Lecture Notes in Computational Optimization

Jianer Chen

Department of Computer Science
Texas A&M University
College Station, TX 77843-3112
U. S. A.
chen@cs.tamu.edu

December, 1995

Preface

This manuscript consists of lecture notes for *CPSC-669 Computational Optimization* as it was taught by me in the Fall of 1995 at Department of Computer Science, Texas A&M University. The notes were first taken by students in the class then were rewritten by myself. The notes were not meant at all to be in polished form and they probably contain many errors. I will appreciate that readers let me know their corrections and comments.

Because of the time limit, I was not able to cover many other recent interesting and important results in this set. The first few of them in my mind are the probabilistic method and derandomization, recent improved approximation algorithms for Max-Sat and Max-Cut, and approximability of Steiner trees. The discussion on linear programming should certainly be in more detail and in more depth. My plan is to add at least these topics in the next revision.

Help from the following list of scribes is acknowledged: M. Chatterjee, X. Chen, S. Lu, L. Shao, B. Varanasi, J. Walter, W. Zhang, and H. Zheng. I also appreciate encouraging discussion and comments from Professors D. Friesen and C. Papadimitriou.

CPSC-669 Computational Optimization

Lecture #1, August 30, 1995

Lecturer: Professor Jianer Chen
Scribe: Jennifer Walter
Revision: Jianer Chen

1 Introduction

Most computational optimization problems come from practice in industry and other fields.

Definition 1.1 An *optimization problem* Q is a 4-tuple $\langle I_Q, S_Q, f_Q, opt_Q\rangle$, where I_Q is the set of *input instances*, S_Q is a function such that for each input $x \in I_Q$, $S_Q(x)$ is a set of *solutions* to x, f_Q is the *objective function* such that for each pair $x \in I_Q$ and $y \in S_Q(x)$, $f_Q(x, y)$ is an integer, and $opt_Q \in \{\max, \min\}$ specifies the problem to be a *maximum problem* or a *minimum problem*.

Therefore, an optimization problem can be defined as follows: given an input instance x, find a solution y in $S_Q(x)$ such that the objective function value $f_Q(x, y)$ is optimized (maximized or minimized depending on opt_Q) among all solutions in $S_Q(x)$.

Remark 1.2 The 4-tuple must satisfy the following conditions for an optimization problem:

1. It should be testable in polynomial time whether a given x is an input instance of Q.

2. It should be testable in polynomial time given x and y whether y is a solution in $S(x)$.

3. The objective function $f(x, y)$ should be computable in polynomial time, given $x \in I_Q$ and $y \in S_Q(x)$.

Examples of combinatorial optimization problems:
1. Minimum Spanning Tree
2. Shortest Path
3. Knapsack

4. Bin Packing
5. Vertex Cover
6. Traveling Salesman Problem

This list is not exhaustive. There are many other optimization problems.

Example 1.3 *How do we formulate the Minimum Spanning Tree problem using the above formulation?*

By using the definition of an optimization problem, we can formulate the MST problem as follows:
I_Q: weighted graph G
S_Q: all spanning trees of G
f_Q: $f_Q(G,T)$ = sum of weights of edges of T, a spanning tree of G
opt_Q: min

Example 1.4 *How do we formulate the Shortest Path problem?*

I_Q: weighted graph G with two specified vertices u, v
S_Q: all paths connecting vertices u, v in G
f_Q: $f_Q(G,u,v,p)$ = length of p, a path connecting u, v
opt_Q: min

Example 1.5 *How do we formulate the Knapsack problem?*

I_Q: Set $S = \{x_1, x_2, \ldots, x_n\}$, where each x has size s_i and profit f_i. Bound B on size is also defined.
S_Q: $S' \subseteq S$, $\sum_{x_i \in S'} s_i \leq B$
f_Q: $f_Q(S, S') = \sum_{x_i \in S'} f_i$
opt_Q: max

Example 1.6 *How do we formulate the Bin Packing problem?*

I_Q: Set $S = \{x_1, x_2, \ldots, x_n\}$, where $0 < x_i < 1$.
S_Q: Partition P of S into $S_1 \cup S_2 \cup \ldots \cup S_r$ such that $\sum_{x \in S_i} x \leq 1$
f_Q: $f_Q(S, P) = r$
opt_Q: min

Example 1.7 *How do we formulate the Vertex Cover problem?*

I_Q: A graph $G = (V, E)$.
S_Q: A subset S of V such that every edge e in E has at least one end in S.
f_Q: $f_Q(G, S) =$ the number of vertices in S.
opt_Q: min

Example 1.8 *How do we formulate the Traveling Salesman problem?*

I_Q: A weighted complete graph $G = (V, E)$.
S_Q: A path P in G that goes through all vertices of G.
f_Q: $f_Q(G, P) =$ the weight of the path P.
opt_Q: min

Examples 1.3 and 1.4 can be solved in polynomial time. Examples 1.5 to 1.8 are known to be NP-hard, which means it is unlikely to have efficient algorithms for solving them precisely. For these problems, we will discuss efficient *approximation algorithms* that find solutions "close" to the optimal ones. We will see that for Knapsack problem, there is a very good approximation algorithm that produce solutions arbitrarily close to the optimal solutions. For Bin Packing problem and Vertex Cover, we will see that approximation algorithms of constant ratio will be possible while it is unlikely for them to have further better approximation algorithm. For Traveling Salesman problem, we will see that any reasonable approximation will be infeasible.

The course will start with optimization problems that can be solved in polynomial time. Examples are Maximum Flow, Matching, and Linear Programming. Then we discuss approximation algorithms on NP-hard optimization problems. We first discuss techniques that approximate NP-hard optimization problems with solutions that are arbitrarily close to optimal solutions. This class of optimization problems includes Knapsack and many scheduling problems. Then we present approximation algorithms with constant ratio for certain optimization problems and show that no much better approximation algorithms are possible for these problems. Bin Packing and Vertex Cover belong to this class. We will also discuss optimization problems such as Traveling Salesman Problem, which are very hard to approximate.

CPSC-669 Computational Optimization

Lecture #2, September 1, 1995

Lecturer: Professor Jianer Chen
Scribe: Jennifer Walter
Revision: Jianer Chen

2 Max-Flow Problem

Definition 2.1 A *flow graph* $G = (V, E)$ is a directed and positively weighted graph with two distinguished vertices s (the source) and t (the sink). The weight on an edge (u, v) is called the *capacity* of the edge, and is designated by $cap(u, v)$. If there is no edge from vertex u to vertex v, then we define $cap(u, v) = 0$.

Remark 2.2 Edges can be directed into the source and out of the sink.

Definition 2.3 A *flow* f on a flow graph $G = (V, E)$ is a function on pairs of vertices of G satisfying the following conditions:

1. For all $u, v \in V$, $cap(u, v) \geq f(u, v)$.
2. For all $u, v \in V$, $f(u, v) = -f(v, u)$.
3. For all $u \neq s, t$, $\sum_{v \in V} f(u, v) = 0$.

Question 2.4 *What is the flow value from u to v if there is no edge between u and v ?*

By the definition, if there is no edge between u and v, then we have $cap(u, v) = cap(v, u) = 0$. By the first condition of a flow f, we must have $cap(u, v) \geq f(u, v)$ and $cap(v, u) \geq f(v, u)$. These together with the second condition of the flow $f(u, v) = -f(v, u)$ give immediately $f(u, v) = 0$.

Remark 2.5 Note the following about capacities and flows:

- $cap(u, v)$ is always defined.
- If $cap(u, v) = 0$, then $f(u, v)$ can be negative.
- $cap(u, v)$ is in general not equal to $cap(v, u)$.

Definition 2.6 Given a flow graph $G = (V, E)$ and given a flow f on G, the *residual graph* $G_f = (V, E')$ of G (with respect to the flow f) has the same vertex set as G. Moreover, for each vertex pair u, v, if $cap(u, v) > f(u, v)$, then (u, v) is an edge in G_f with capacity $cap(u, v) - f(u, v)$.

Remark 2.7 New edges may be created in the residual graph G_f that were not originally present in the original graph G.

Remark 2.8 Max-Flow problem can be formulated using our definition of optimization problems as a 4-tuple Max-Flow = $\langle I, S, f, opt \rangle$

I:	flow graphs G with source s and sink t
S:	$S(G)$ is the set of valid flows f on G
f:	$f(G, f) = \sum_{v \in V} f(s, v)$
opt:	max

Remark 2.9 The goal in the Maximum Flow Problem is to find the maximum flow from source to sink. Solving the Max-Flow problem involves finding paths from s to t and pushing the maximum flow over those paths. Formally, the goal of Max-Flow is to maximize $\sum_{v \in V} f(s, v)$, the amount of flow coming out of the source. Alternatively, the goal could be specified as maximizing $\sum_{w \in V} f(w, t)$, the amount of flow going into the sink. It can be proved that these two definitions are equivalent. The proof is not very hard and left to the students.

CPSC-669 Computational Optimization

Lecture #3, September 4, 1995

Lecturer: Professor Jianer Chen
Scribe: Mitrajit Chatterjee
Revision: Jianer Chen

3 Max-Flow Problem (Contd.)

Theorem 3.1 *Let G be a flow-graph and let f be a flow in G. The flow f is a maximum flow in G if and only if the residual graph G_f has no positive flow.*

PROOF.

($\Rightarrow$). Assume that there is a positive flow f^* in the residual graph G_f, i.e. $|f^*| = \sum_{v \in V} f^*(s,v) > 0$.

Define a function f^+ on each pair (u,v) of vertices in the flow-graph G as follows:

$$f^+(u,v) = f(u,v) + f^*(u,v)$$

Claim: f^+ is a valid flow in G.

Proof for the Claim: A flow is valid if it satisfies all the three conditions as described in Definition 2.3. The conditions are verified as follows.

(a) For all $u, v \in V$, $cap(u,v) \geq f^+(u,v)$:

We compute the value $cap(u,v) - f^+(u,v)$. By definition we have

$$cap(u,v) - f^+(u,v) = cap(u,v) - f(u,v) - f^*(u,v)$$

Now by the definition of cap_f, we have $cap(u,v) - f(u,v) = cap_f(u,v)$. Moreover, since $f^*(u,v)$ is a valid flow in the residual graph G_f, $cap_f(u,v) - f^*(u,v) \geq 0$. Consequently, we have $cap(u,v) - f^+(u,v) \geq 0$. The condition is thus satisfied.

(b) For all $u, v \in V$, $f^+(u,v) = -f^+(v,u)$:

Since both $f(u,v)$ and $f^*(u,v)$ are valid flows in flow-graphs G and G_f, respectively, we have $f(u,v) = -f(v,u)$ and $f^*(u,v) = -f^*(v,u)$. Thus,

$$f^+(u,v) = f(u,v) + f^*(u,v) = -f(v,u) - f^*(v,u) = -f^+(v,u)$$

(c) For all $u \neq s, t$, $\sum_{v \in V} f^+(u,v) = 0$:

Again, since both $f(u,v)$ and $f^*(u,v)$ are valid flows in flow-graphs G and G_f, respectively, we have for all $u \neq s,t$

$$\sum_{v \in V} f(u,v) = \sum_{v \in V} f^*(u,v) = 0$$

Thus

$$\sum_{v \in V} f^+(u,v) = \sum_{v \in V} f(u,v) + \sum_{v \in V} f^*(u,v) = 0$$

Thus, the function f^+ satisfies all three conditions for a flow in G and is a valid flow in the flow-graph G. Now we compute $|f^+|$ and note that $|f^*| > 0$, we get

$$|f^+| = \sum_{v \in V} f^+(s,v) = \sum_{v \in V} f(s,v) + \sum_{v \in V} f^*(s,v) = |f| + |f^*| > |f|$$

Hence f is not a maximum flow in G.

($\Leftarrow$). Here, we assume that f is not a maximum flow in G. Let $f_{\max}$ be a maximum flow in G. Thus, $|f_{\max}| - |f| > 0$. Now define a function f^- on each pair (u,v) of vertices in the flow-graph G_f as follows.

$$f^-(u,v) = f_{\max}(u,v) - f(u,v)$$

Claim: f^- is a valid flow in G_f.

Proof for the Claim: Again we verify the three conditions of a flow in G_f.

(a) For all $u,v \in V$, $cap_f(u,v) \geq f^-(u,v)$:

$$cap_f(u,v) - f^-(u,v) = cap(u,v) - f(u,v) - f^-(u,v)$$

Note that $f(u,v) + f^-(u,v) = f_{\max}(u,v)$. Since $f_{\max}$ is a valid flow in G, we have $cap(u,v) - f_{\max}(u,v) \geq 0$. Consequently, we have $cap_f(u,v) - f^-(u,v) \geq 0$.

(b) For all $u,v \in V$, $f^-(u,v) = -f^-(v,u)$:

$$f^-(u,v) = f_{\max}(u,v) - f(u,v) = -f_{\max}(v,u) + f(v,u) = -f^-(v,u)$$

(c) For all $u \neq s,t$, $\sum_{v \in V} f^-(u,v) = 0$:

$$\sum_{v \in V} f^-(u,v) = \sum_{v \in V} f_{\max}(u,v) - \sum_{v \in V} f(u,v) = 0$$

This, f^- is a valid flow in the flow-graph G_f. Moreover, since we have

$$|f^-| = \sum_{v \in V} f^-(s,v) = \sum_{v \in V} f_{max}(s,v) - \sum_{v \in V} f(s,v) = |f_{max}| - |f| > 0$$

We conclude that the residual graph G_f has a positive flow.

This completes the proof of the theorem. $\square$

Theorem 3.1 ensures the correctness of the following algorithm.

Algorithm 3.1 Max-Flow

```
Input:  A flow-graph $G$.
Output:  A maximum flow $f$ on $G$.
1.  Let $f(u,v) = 0$ for all pairs $(u,v)$ of vertices in $G$;
2.  Construct the residual graph $G_f$;
3.  while there is a positive flow $f^*$ in $G_f$ do
        Construct a positive flow $f^*$ in $G_f$;
        Let $f = f + f^*$ be the new flow on $G$.
        Construct the residual graph $G_f$;
```

Remark 3.1 Whenever there is a positive flow f^* in G_f, there is at least one directed path in G_f from s to t on which all the edges have a positive capacity. There can be several approaches to find such paths in the residual graph G_f. An algorithm by *Ford-Fulkerson* finds a path of maximum capacity. This algorithm is efficient in most cases, but can perform badly in some cases. In this context, *Dinic's* (Dinitz) algorithm has a stronger bound on the time complexity. This algorithm tries to find the shortest path from s to t. The path length is based on the number of edges in the path. The shortest path can be determined by using *breadth first search* (BFS) algorithm. In each iteration of the **while** loop in Algorithm 3.1, Dinic's algorithm will push the flow through *all* the shortest paths, so that in the next iteration, the length of the shortest path increases at least by one. Dinic's algorithm and its analysis will be presented in the next lecture.

CPSC-669 Computational Optimization

Lecture #4, September 6, 1995

Lecturer: Professor Jianer Chen
Scribe: Mitrajit Chatterjee
Revision: Jianer Chen

4 Max-Flow Problem (Contd.)

Algorithm 4.1 BFS_Dinic

```
Input:  A flow-graph $G$.
Output:  A leveled graph $L = (V_L, E_L)$ containing all shor-
         test paths in $G$ from $s$ to $t$.
1.  C_level = $-1$; { C_level = current level }
2.  For all vertices $v$, level[$v$] $= n+1$;
3.  level[$s$] $= 0$;  $Q \leftarrow s$;  $V_L = \{s\}$;
    {$Q$ is a queue.}
4.  while $Q$ is non-empty and C_level < level[$t$] do
      $v \leftarrow Q$;
      if (C_level < level[$v$]) then C_level = level[$v$];
      for each edge $(v, w)$ in $G$ do
        if (level[$w$] $= n+1$)
          then $V_L = V_L \bigcup \{w\}$;  $Q \leftarrow w$;
        if (level[$w$] > level[$v$]) then
            $E_L = E_L \bigcup \{(v, w)\}$;
            level[$w$] = level[$v$] $+ 1$;
```

Remark 4.1 The above algorithm is a modification of the famous breadth first search algorithm. The analysis can be performed similarly as for breadth first search. Thus, we conclude that the time complexity of the algorithm is $O(e)$, where e is the number of edges in the flow-graph G. This algorithm stops either when it reaches t (in this case, the leveled graph L is constructed), or when it exhausts all the edges (in this case, the vertices s and t are disconnected).

Given the leveled graph L, we find all paths in L from the source s to the sink t as follows. Starting from the vertex s, we follow the edges of L to find a path p of length level[t]. Since the graph L is leveled, the path p

can be found in a straightforward way (i.e., at each vertex, simply follow an arbitrary edge from the vertex). Thus, the path p can be constructed in time $O(\text{level}[t]) = O(n)$, where n is the number of vertices in G. Now if the ending vertex is t, then we have found a path from s to t. We trace back the path p to find the edge e on p with minimum capacity c. Now we can push c amount of flow along the path p. Note that this cuts at least one edge, e.g. the edge e, from the path p. On the other hand, if the ending vertex v of p is not t, then v must be a "deadend". Thus, we can cut all incoming edges to v. In conclusion, in the above process of time $O(n)$, at least one edge is removed from the leveled graph L. Thus, after at most e such processes, the vertices s and t are disconnected, i.e., all shortest paths from s to t are saturated. This totally takes time $O(ne)$. We give a formal description for the above process.

Algorithm 4.2 SATURATING

```
Input:  Leveled graph $L$.
1.  **while** there is an edge from $s$ **do**
      find a path $p$ of maximal length from $s$
      **if** $p$ leads to $t$
        **then** saturate $p$ and delete at least one edge on $p$.
        **else** delete the last edge on $p$.
```

Now the complete version for Dinic's algorithm can be given as follows.

Algorithm 4.3 Max-Flow_Dinic

```
Input:  A flow-graph $G$.
Output:  A maximum flow $f$ on $G$.
1.  Let $f(u,v) = 0$ for all vertex pairs $(u,v)$;
2.  Construct the residual graph $G_f$;
3.  **while** there is a positive flow in $G_f$ **do**
      Call BFS_Dinic on $G_f$ to construct the leveled graph $L$;
      Call SATURATING to saturate all paths in $L$;
      Let $f^*$ be the flow in $G_f$ constructed by SATURATING;
      Let $f = f + f^*$ be the new flow in $G$;
      Construct the residual graph $G_f$;
```

By the above discussion, each execution of the body of the **while** loop in Algorithm 4.3 takes time $O(ne)$. Now we study the number of times the body of the **while** loop is executed.

Theorem 4.1 *On a flow-graph G of n vertices, the body of the* **while** *loop in Step 3 of Algorithm 4.3 is executed at most $n-1$ times.*

PROOF. We first prove that after each execution of the body of the **while** loop, the length of the shortest path in the flow-graph is increased by at least one. We need some notations. Let G be a flow-graph, let f be the flow obtained by one execution of the body of the **while** loop on the flow-graph G, and let G_f be the residual graph of G on the flow f. For any vertex v of G, let level(v) be the distance from s to v in the graph G, and let level$_f(v)$ be the distance from s to v in the graph G_f.

Claim 1: Suppose (v, w) is an edge in G_f, then $\text{level}(w) \leq \text{level}(v) + 1$.

Proof for Claim 1: (v, w) can be an edge in G_f due to two cases:

Case 1: (v, w) is an edge in G. Then either the vertex w is seen before we start the search from the vertex v — in this case the level of w cannot be larger then level$(v) + 1$, or the vertex w is discovered in the search from v — in this case, the level of w is exactly one plus the level of v.

Case 2: (v, w) is not an edge in G. Since (v, w) is an edge in the residual graph G_f of G on the flow f, we must have that (w, v) is an edge in G and there is a positive flow in f from the vertex w to the vertex v. Since we only push flow in the leveled graph L on edges that only connect consecutive levels of vertices, we conclude that level(v) is one plus level(w). Thus, certainly we also have $\text{level}(w) \leq \text{level}(v) + 1$.

Claim 2: For all vertices v, we have $\text{level}(v) \leq \text{level}_f(v)$.

Proof for Claim 2: Let $r = \text{level}_f(v)$ be the distance from s to v in the graph G_f. Let $(s, x_1, x_2, \ldots, x_{r-1}, v)$ be a shortest path in G_f from s to v. Then

$$\begin{array}{rcll} \text{level}(v) & \leq & \text{level}(x_{r-1}) + 1 & \{\text{due to Claim1}\} \\ & \leq & \text{level}(x_{r-2}) + 2 & \\ & \cdots & & \\ & \leq & \text{level}(x_1) + (r-1) & \\ & \leq & \text{level}(s) + r & \\ & = & r \;\; = \;\; \text{level}_f(v) & \end{array}$$

In particular, we have $\text{level}(t) \leq \text{level}_f(t)$, which implies that the length of the shortest path from s to t is *not decreased* after each execution of the body of the **while** loop.

Claim 3: level$(t) <$ level$_f(t)$.

Proof for Claim 3: It has been already shown that level$(t) \leq$ level$_f(t)$ in Claim 2. Hence, to prove Claim 3, we only need to show that level(t) and level$_f(t)$ are different. Let us assume the contrary that level$(t) =$ level$_f(t) = r$ and derive a contradiction.

Let $P = (s, x_1, x_2, \ldots, x_{r-1}, t)$ be a shortest path in the graph G_f from the source s to the sink t. Then we must have

$$\text{level}_f(t) = \text{level}_f(x_{r-1}) + 1 = \ldots = \text{level}_f(s) + r = r$$

By Claim 1, we have

$$\begin{array}{rcl} \text{level}(t) & \leq & \text{level}(x_{r-1}) + 1 \\ & \leq & \text{level}(x_{r-2}) + 2 \\ & \cdots & \\ & \leq & \text{level}(x_1) + (r-1) \\ & \leq & \text{level}(s) + r \\ & = & r \end{array}$$

By our assumption, we also have level$(t) = r$, thus all inequalities "$\leq$" in the above formula should be equality "=". This gives level$(x_{i+1}) =$ level$(x_i) + 1$ for all $i = 1, \ldots, r-2$, level$(x_1) =$ level$(s) + 1$, and level$(t) =$ level$(x_{r-1}) + 1$. Now we show that P is also a path in the graph G. In fact, if (s, x_1) is not an edge in G, then since (s, x_1) is an edge in G_f, (x_1, s) must be an edge in G and we have pushed a flow in f along the edge (x_1, s). But this implies that (x_1, s) is an edge in the leveled graph L so level$(x_1) + 1 =$ level(s), contradicting the fact that level$(x_1) =$ level$(s) + 1$, Thus, (s, x_1) is an edge in G. Similarly, all edges on the path P are edges in the graph G. Therefore, the path P is also a path in the graph G. Since the length of the path P is $r =$ level(t), P is a shortest path in G. By our SATURATING algorithm, at least one of the edges on P is saturated, thus at least one of the edges on P should not appear in the residual graph G_f. This contradicts the assumption that P is also a path in the graph G_f. The contradiction proves level$(t) <$ level$_f(t)$.

Thus, each execution of the body of the **while** loop in Algorithm 4.3 increases the length of the shortest path from s to t in the flow graph G_f by at least 1.

Now we can complete the proof of the theorem. Since we start with the original flow-graph G in which the length of the shortest paths from s to

t is at least one (we can always assume that the source s and the sink t are different), if the body of the **while** loop were executed more than $n-1$ times, Claim 3 says that the length of the shortest path from s to t in the resulting residual graph G_f would be at least n, i.e., would consist of more than n vertices. But this contradicts the fact that the graph G_f has only n vertices. $\square$

Theorem 4.2 *The running time of Dinic's Maximum Flow algorithm (Algorithm 4.3) is* $O(n^2 e)$.

CPSC-669 Computational Optimization

Lecture #5, September 8, 1995

Lecturer: Professor Jianer Chen
Scribe: Weijie Zhang
Revision: Jianer Chen

5 Max-Flow Problem (Contd.)

5.1 Edmonds-Karp's Algorithm

We first give a formal proof for a claim we made in the last lecture. Recall that we denote by $\text{level}(v)$ and $\text{level}_f(v)$ the distance from the source node s to the node v in the flow-graphs G and G_f, respectively.

Lemma 5.1 *Let G be a flow-graph and let f be the flow generated by an execution of the body of the* **while** *loop in Dinic's algorithm. If (u, v) is an edge in the residual graph G_f and $\text{level}(u) = \text{level}(v) - 1$ in G, then (u, v) is also an edge in the original flow-graph graph G.*

PROOF. Suppose (u, v) is not an edge in G. Since (u, v) is an edge in the residual graph G_f, we must have that (v, u) is an edge in G and we pushed a flow in f from vertex v to vertex u. However, since each execution of the **while** of Dinic's algorithm pushes flow only in the leveled graph L, we conclude that

$$\text{level}(v) + 1 = \text{level}(u)$$

This contradicts the condition given in the lemma that $\text{level}(u) = \text{level}(v) - 1$. $\square$

In the last lecture, we have proved that if $\text{level}_f(t) = \text{level}(t)$, then for a shortest path $P = (s, x_1, \ldots, x_{r-1}, t)$ in the graph G_f, we must have $\text{level}(x_i) = \text{level}(x_{i+1}) - 1$, $\text{level}(s) = \text{level}(x_1) - 1$, and $\text{level}(x_{r-1}) = \text{level}(t) - 1$ in G. Applying Lemma 5.1 claims that P is also a path in the original graph G. Since P is a shortest path in G_f and $\text{level}(t) = \text{level}_f(t)$, P is also a shortest path in the original graph G. Consequently, P is contained in the leveled graph L. By the subroutine SATURATING, all paths in the leveled graph L are saturated. Thus, the path P in G should have also been saturated, and at least one of the edges on P should have not appeared in the residual graph G_f. But this contradicts the assumption that P is a path in G_f. This contradiction combined with the inequality $\text{level}_f(t) \geq \text{level}(t)$ gives

$$\text{level}_f(t) > \text{level}(t)$$

Therefore, each execution of the body of the **while** loop in Dinic's algorithm (Algorithm 4.3) increases the length of the shortest path in the flow-graph G_f by at least 1. Since the lengths of the shortest paths in G_f cannot be larger than $n - 1$, the **while** loop can be executed at most $n - 1$ times. Moreover, as we have discussed before, each execution of the body of the **while** loop takes time $O(ne)$. This concludes that Dinic's algorithm runs in time $O(n^2 e)$.

It will be interesting to compare Dinic's algorithm with Edmonds-Karp's algorithm, which also uses the strategy of finding shortest augmenting path. Instead of finding all shortest paths, Edmonds-Karp's algorithm finds just one shortest path each time and saturates the path. The algorithm can be given as follows.

Algorithm 5.1 `Edmonds-Karp`

```
Input:  a flow-graph $G$
Output:  a maximum flow on $G$

1.  let f be the zero flow;
2.  construct the residual graph $G_f$;
3.  **while** there is a positive capacity path $P$ in $G_f$ **do**
      find a shortest positive capacity path $P_0$;
      increase the flow $f$ along the $P_0$ as much as possible;
      construct $G_f$ for the new $f$;
```

We omit the detailed analysis here. An informal analysis can be given as follows. Finding a single shortest path from s to t can be done using breadth first search in time $O(e)$. Other steps in the loop can easily be done in time $O(e)$. Thus, each execution of the body of the **while** loop takes time $O(e)$. Each execution of the body of the **while** loop in the above algorithm cuts at least one edge from a shortest path. Therefore, after at most e executions, all shortest paths of the same length have been cut so that the length of the shortest paths in the flow-graph G_f must be increased by at least 1. Now using the same argument as above, the length of the shortest paths cannot be larger than $n - 1$. Therefore, after at most $O(en)$ executions of the body of the **while** loop in the above algorithm, there will be no positive capacity path from s to t in G_f and the algorithm stops with a maximum flow. This concludes that Edmonds-Karp's algorithm runs in time $O(ne^2)$, which is slightly worse than Dinic's algorithm.

5.2 Multiple source-sink flow problem

We say that a flow-graph G is a *multiple source-sink* flow-graph if G has more than one source or more than one sink (or both). The multiple source-sink flow problem can be reduced to the single source-sink flow problem as follows.

1. add a new source S and add a new sink T;
2. add directed edges which goes from the new source S to all old sources in the original flow-graph, and add an directed edge from every old sink to the new sink T.
3. define the capacity of every new added edge. We can simply let the capacity be a very large number. For example, this number can be the sum of the capacities of all edges in the original flow graph.

This is easy to see that a maximum flow in the new constructed single source-sink flow-graph gives a maximum flow in the original multiple source-sink flow-graph.

5.3 Graph Matching

Definition 5.1 Given an undirected graph $G = (V, E)$, a *maximum matching* is a maximum subset of edges $E^{'}$ of E such that no two edges in $E^{'}$ share a common endpoint.

Using the formal definition of an optimization problem, we can formulate the Graph Matching problem as a 4-tuple $Q = (I_Q, S_Q, f_Q, opt_Q)$, where:

I_Q: the set of all undirected graphs $G = (V, E)$;

S_Q: given $G = (V, E) \in I_Q$, $S_Q(G)$ is the collection of all subsets E' of E such that no two edges in E' share a common endpoint;

f_Q: given $G \in I_Q$ and $E' \in S_Q(G)$, $f_Q(G, E')$ is equal to the number of edges in E';

opt$_Q$: max

In this lecture we will discuss a special case: to find maximum matchings in bipartite graphs.

Definition 5.2 A *bipartite graph* is an undirected graph $G = (V, E)$ in which V can be partitioned into two sets V_1 and V_2 such that $(u, v) \in E$ implies either $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$. That is, all edges go between the two sets V_1 and V_2.

There are several approaches to solve the maximum matching problem in bipartite graphs.

- We can use the method of augmenting paths, which is described in our Algorithm Analysis course. The time complexity for this method is $O(ne)$. We will give a more detailed and careful study on this method for general non-bipartite graphs.
- We can use Dinic's Algorithm to find a maximum matching in an undirected bipartite graph $G = (V, E)$ by constructing a flow graph in which flows correspond to matchings. We define the corresponding flow graph G' as follows:

  a. add two new vertices, let them be the source s and the sink t,
  b. add new directed edges from the source s to the vertices in V_1 and new directed edges from the vertices in V_2 to the sink t,
  c. give each edge in the original graph G a direction so all these edges go from V_1 to V_2,
  d. assign unit capacity to each edge in the graph G'.

The proof of the following Theorem is straightforward and left for the reader.

Theorem 5.2 *A maximum matching in a bipartite graph G corresponds directly to a maximum flow in the flow-graph G'.*

If we apply Dinic's Algorithm directly to the above flow-graph G', we can only claim a time bound $O(n^2e)$, which is worse than the augmenting path method. However, a more careful analysis plus a slight modification will show that the running time of Dinic's Algorithm on the above flow-graph G' is bounded by $O(\sqrt{n}e)$, thus a better result than the direct augmenting path method. The details of this analysis and the modification will be given later in this course.

CPSC-669 Computational Optimization

Lecture #6, September 11, 1995

Lecturer: Professor Jianer Chen
Scribe: Weijie Zhang
Revision: Jianer Chen

6 Karzanov's Algorithm

In this lecture, we present Karzanov's Algorithm to get a maximum flow. This approach runs in $O(n^3)$ time, thereby an improvement upon Dinic's Algorithm which runs in $O(n^2e)$ time. Let us review Dinic's Algorithm first.

Algorithm 6.1 Dinic's Algorithm

```
Input:  a flow-graph $G$
Output:  a maximum flow on $G$

1.  let $f$ be the zero flow;
2.  construct the residual graph $G_f$;
3.  **while** there is a positive capacity path $P$ in $G_f$ **do**
     **begin**
3.1    find all shortest paths of positive capacity from $s$
       to $t$ in $G_f$
3.2    increase the flow $f$ along these paths as much as
       possible;
3.3    construct $G_f$ for the new flow $f$;
     **end**
```

Step 3.1 can be done in $O(e)$ time by Breadth-First Search, and Step 3.3 can easily done in time $O(e)$. Moreover, we have already proved that the while loop can be executed at most $n-1$ times. Finally, our early implementation shows that Step 3.2 takes time $O(en)$. Therefore if we want to improve the time complexity from $O(n^2e)$ to $O(n^3)$, what we need to do is to improve the running time of Step 3.2. Now the question is how to improve it.

Let us have a closer look at our implementation of Step 3.2 in Dinic's algorithm. With the leveled graph L being constructed, we iterate the process of searching a path in L from the source s to the sink t, pushing flow along the path, and saturating (thus cutting) at least one edge on the path. In the worst case, for each such a path, we may only be able to cut one edge. Therefore, to ensure that the leveled graph L is eventually cut, we may have to perform the above iteration e times.

The basic idea of Karzanov's algorithm is to reduce the number of times of the above iteration from e to n. In each iteration, instead of saturating an edge in L, Karzanov saturates a vertex in L. Since there are at most n vertices in the leveled graph L, the number of iterations is bounded by n.

Definition 6.1 Let v be a vertex in the leveled graph $L = (V_0, E_0)$. Define the *capacity* $cap(v)$ of the vertex v to be

$$cap(v) = \min\left(\sum_{(w,v)\in E_0} cap(w,v), \sum_{(v,u)\in E_0} cap(v,u)\right)$$

That is, $cap(v)$ is the maximum amount of flow we can push through the vertex v. For the source s and the sink t, we naturally define

$$cap(s) = \sum_{(s,u)\in E_0} cap(s,u) \qquad \text{and} \qquad cap(t) = \sum_{(w,t)\in E_0} cap(w,t)$$

If we start from an arbitrary vertex v and try to push a flow of amount $cap(v)$ through v, it may not always be possible. For example, pushing $cap(v) = 10$ units flow through a vertex v may require to push 5 units flow along an edge (v, w), which requires that $cap(w)$ is at least 5. But the capacity of the vertex w may be less than 5, thus we would be blocked at the vertex w. However, if we always pick the vertex w in L with the smallest capacity, this problem will disappear. In fact, trying to push a flow of amount $cap(w)$ will require no more than $cap(v)$ amount of flow to go through a vertex v for all vertex v. Therefore, we can always push the flow all the way to the sink t (assuming we have no deadend vertices). Similarly, we can *pull* this amount $cap(w)$ of flow from the incoming edges of w all the way back to the source s. Note that this process saturates the vertex w. Thus, the vertex w can be removed from the leveled graph L in the rest of the iterations of the algorithm `SATURATING` on L.

Now we can formally describe Karzanov's Algorithm. The first subroutine deletes all deadends in the leveled graph L and computes the capacity for each vertex in L.

Algorithm 6.2 `INITIALIZATION`

```
Input:  the leveled graph $L$
1.  Perform a depth first search on $L$ to delete all
     vertices that are not on a path from $s$ to $t$;
2.  for each vertex $v \neq s, t$ do
       $in[v] = 0$;  $out[v] = 0$;  $f[v] = 0$;
3.  $in(s) = +\infty$;  $out(t) = +\infty$;
4.  for each edge $(u, v)$ do
       $in[v] = in[v] + cap(u, v)$;
       $out[u] = out[u] + cap(u, v)$;
5.  for each vertex $v$ do
       $cap(v) = \min\{in[v], out[v]\}$
```

Here, $in[v]$ is the sum of capacities of all incoming edges of vertex v, $out[v]$ is the sum of capacities of all outgoing edges of vertex v, and $f[v]$ is the amount of flow we want to push (or pull) through vertex v.

We will always start with a vertex v with the smallest $cap(v)$ and push a flow of amount $cap(v)$ through it all the way to the sink t. This process is similar to the breadth first search algorithm, starting from the vertex v. We use the array $f[\cdot]$ to record the amount of flow we need to push through the corresponding vertex. $f[w] = 0$ implies that the vertex w has not been seen in the breadth first search.

Algorithm 6.3 PUSH(v)

```
Input:  the leveled graph $L$
{$Q$ is a queue used for the breadth first search.}
1.   $Q \leftarrow v$;     $f[v] = cap(v)$;
2.   while $Q$ is not empty do
3.       $u \leftarrow Q$;     $f_0 = f[u]$;
4.       while $f_0 > 0$ do
5.           let $(u, w)$ be the next edge from $u$
6.           if $f[w] = 0$ and $w \neq t$ then $Q \leftarrow w$;
7.           if $cap(u, w) < f_0$ then
8.             cut edge $(u, w)$;
9.             $f[w] = f[w] + cap(u, w)$;    $f_0 = f_0 - cap(u, w)$;
10.          else
11.            push $f_0$ along $(u, w)$;
12.            $cap(u, w) = cap(u, w) - f_0$;    $f[w] = f[w] + f_0$;    $f_0 = 0$;
13.      if $u \neq v$ then $cap(u) = cap(u) - f_0$;
```

```
14.      if $u \neq v$ and $cap(u) = 0$
          then delete $u$ from the leveled graph $L$.
```

Note that we neither change the value $cap(v)$ nor remove the vertex v from the leveled graph L. This is because the vertex v will be used again in the following PULL algorithm.

The algorithm PULL is very similar to algorithm PUSH. We start from the vertex v and pull $cap(v)$ amount of flow all the way back to the source vertex s. Note that now the breadth first search is on the reversed directions of the edges of the leveled graph L. This can be easily done by a reorganization of the adjacency list representation of the graph L and the process can be done in time $O(e)$ (this only needs to be done once for all calls to PULL). Moreover, note that the only vertex that can be seen in both PUSH subroutine and PULL subroutine is the vertex with the smallest capacity. Therefore, no updating is needed for array $f[\cdot]$.

Algorithm 6.4 PULL(v)

```
Input: the leveled graph $L$
{$Q'$ is a queue used for the breadth first search.}
1.  $Q' \leftarrow v$;    $f[v] = cap(v)$;
2.  while $Q'$ is not empty do
3.      $u \leftarrow Q'$;    $f_0 = f[u]$;
4.      while $f_0 > 0$ do
5.          let $(w, u)$ be the next edge into $u$
6.          if $f[w] = 0$ and $w \neq s$ then $Q' \leftarrow w$;
7.          if $cap(w, u) < f_0$ then
8.            cut edge $(w, u)$;
9.            $f[w] = f[w] + cap(w, u)$;    $f_0 = f_0 - cap(w, u)$;
10.         else
11.           push $f_0$ along $(w, u)$;
12.           $cap(w, u) = cap(w, u) - f_0$;    $f[w] = f[w] + f_0$;    $f_0 = 0$;
13.     $cap(u) = cap(u) - f_0$;
14.     if $cap(u) = 0$
          then delete $u$ from the leveled graph $L$.
```

Again note that after the execution of the PULL algorithm, the vertex v with minimum capacity always gets removed.

With the subroutines PUSH and PULL, a new saturating subroutine can be given as follows.

Algorithm 6.5 `SATURATING-Karzanov`

```
Input: the leveled graph $L$
Output: a flow $f$ on $L$ that saturates all paths in $L$
1.  call INITIALIZATION;
2.  **while** there is a path from $s$ to $t$ in $L$ **do**
3.     let $v$ be the vertex in $L$ with minimum $cap(v)$;
4.     call PUSH($v$);
5.     call PULL($v$);
```

We now analyze the algorithm SATURATING-Karzanov.

Lemma 6.1 *The algorithm SATURATING-Karzanov takes time $O(n^2)$.*

PROOF. Step 1 takes time $O(e) = O(n^2)$. Steps 3 takes time $O(n)$. Since each execution of the loop body Steps 3-5 deletes at least one vertex from L, the **while** loop body (Steps 3-5) is executed at most n times. Therefore, all executions of Step 3 in the algorithm SATURATING-Karzanov take time $O(n^2)$.

Now we study the complexity of Steps 4 and 5. Let us first consider the subroutine PUSH. To push a flow of amount $f[u]$ through a vertex u, we take each outgoing edge from u. If the capacity of the edge is smaller than the amount of flow we need to push, we saturate the edge, and if the capacity of the edge is not smaller than the amount of flow we need to push, we let all remaining flow go along that edge and jump out from the **while** loop of Steps 4-12 in the algorithm PUSH. Moreover, once an edge gets cut at Step 8 of the algorithm, the edge will never appear in the leveled graph L for the later calls for PUSH in the **while** loop of Algorithm SATURATING-Karzanov. Thus, each execution of the **while** loop body Steps 5-12 in the algorithm PUSH, except the last one, deletes an edge from the leveled graph L. Therefore, the number of total such executions cannot be larger than e. Consequently, all such executions in the algorithm SATURATING-Karzanov take time $O(e) = O(n^2)$. Besides these executions, the subroutine PUSH spends constant time on each vertex u, thus $O(n)$ time on the graph L. Since there are only $O(n)$ calls to the PUSH in the algorithm SATURATING-Karzanov, we conclude that the algorithm SATURATING-Karzanov takes time $O(n^2)$ on all calls to PUSH. Similarly, the total time spent on the calls to PULL is also bounded by $O(n^2)$. □

Algorithm 6.6 `Karzanov's Algorithm`

```
Input:  a flow-graph $G$
Output:  a maximum flow on $G$

1.  let $f$ be the zero flow;
2.  construct the residual graph $G_f$;
3.  while there is a path from $s$ to $t$ in $G_f$ do
3.1.     construct the leveled graph $L$;
3.2.     call SATURATING-Karzanov to find a flow $f^*$ to
           saturate $L$;
3.3.     $f = f + f^*$; construct $G_f$ for the new flow $f$;
```

Theorem 6.2 *Karzanov's Algorithm (Algorithm 6.6) runs in time* $O(n^3)$

PROOF. According to the discussion of Dinic's algorithm, we know that the body of the **while** loop in Algorithm 6.6 is executed at most $n-1$ times. Moreover, Steps 3.1 and 3.3 takes time $O(e) = O(n^2)$. By Lemma 6.1, each call to the subroutine SATURATING-Karzanov takes time $O(n^2)$. We conclude that Karzanov's algorithm takes time $O(n^3)$. □

CPSC-669 Computational Optimization

Lecture #7, September 13, 1995

Lecturer: Professor Jianer Chen
Scribe: Li Shao
Revision: Jianer Chen

7 Maximum matching on bipartite graphs

In this lecture, we study maximum matching problem on bipartite graphs. We show that the problem can be reduced to a special form of the max-flow problem, for which Dinic's algorithm runs very efficiently.

7.1 Max-Flow Min-Cut Theorem

Definition 7.1 Let $G = (V, E)$ be a flow graph with source s and sink t. A partition of $V = V_1 \cup V_2$ (i.e. $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \phi$) is a *cut* if $s \in V_1$, $t \in V_2$.

Definition 7.2 *The capacity of a cut* (V_1,V_2) is defined by the value:

$$cap(V_1, V_2) = \sum_{v \in V_1, w \in V_2} cap(v, w)$$

The following lemma will be used in our later discussion.

Lemma 7.1 *Let $G = (V, E)$ be a flow graph and let (V_1, V_2) be a cut of G. Then for any flow f on G we have*

$$|f| = \sum_{v \in V_1, w \in V_2} f(v, w)$$

PROOF. By definition, we have $|f| = \sum_{w \in V} f(s, w)$. By the definition of a flow, we have $\sum_{w \in V} f(v, w) = 0$ for all vertices $v \in V_1 - \{s\}$. Therefore, we have

$$\begin{aligned} |f| &= \sum_{w \in V} f(s, w) = \sum_{v \in V_1, w \in V} f(v, w) \\ &= \sum_{v \in V_1, w \in V_1} f(v, w) + \sum_{v \in V_1, w \in V_2} f(v, w) \end{aligned}$$

Now since $f(v, w) = -f(w, v)$ for all vertices $v, w \in V_1$, the first term in the last expression of the above equation is equal to 0. The lemma follows. $\square$

Lemma 7.1 implies one direction of the following fundamental theorem in the study of maximum flow problem.

Theorem 7.2 (Max-Flow Min-Cut Theorem) *For any flow graph $G = (V, E)$,*

$$\max\{|f| : f \textit{ is a flow on } G\} = \min\{cap(V_1, V_2) : (V_1, V_2) \textit{ is a cut of } G\}$$

PROOF. Let f be a flow on G and let (V_1, V_2) be a cut of G. By Lemma 7.1 and note that f is a flow on G thus $f(v, w) \le cap(v, w)$ for all vertices $v \in V_1$ and $w \in V_2$, we have

$$|f| = \sum_{v \in V_1, w \in V_2} f(v, w) \le \sum_{v \in V_1, w \in V_2} cap(v, w) = cap(V_1, V_2)$$

Since f is an arbitrary flow on G and (V_1, V_2) is an arbitrary cut of G, we conclude

$$\max\{|f| : f \text{ is a flow on } G\} \le \min\{cap(V_1, V_2) : (V_1, V_2) \text{ is a cut of } G\}$$

To prove the other direction, let f be a maximum flow on the flow graph G. Let G_f be the residual graph of G with respect to f. By Theorem 3.1, there is no path from the source s to the sink t in the residual graph G_f. Define V_1 to be the set of vertices that are reachable from the source s in the graph G_f. Thus, $s \in V_1$ and $t \notin V_1$. Therefore, if we let $V_2 = V - V_1$, then (V_1, V_2) is a cut of the flow graph G. Now let $e = (v, w)$ be an edge in G such that $v \in V_1$ and $w \in V_2$. Since e is not an edge in the residual graph G_f (otherwise, the vertex w would be reachable from s in G_f), the edge e must be saturated by the flow f. That is, $f(v, w) = cap(v, w)$. Therefore, we have

$$cap(V_1, V_2) = \sum_{v \in V_1, w \in V_2} cap(v, w) = \sum_{v \in V_1, w \in V_2} f(v, w)$$

By Lemma 7.1, the last expression is equal to $|f|$. This proves

$$\max\{|f| : f \text{ is a flow on } G\} \ge \min\{cap(V_1, V_2) : (V_1, V_2) \text{ is a cut of } G\}$$

The proof of the theorem is thus completed. □

7.2 Hopcroft and Karp's analysis

Now we describe an analysis given first by Hopcroft and Karp, which gives an $O(\sqrt{n}e)$ time algorithm for maximum matching on bipartite graphs, which is the best algorithm known so far for the problem.

Given a bipartite graph $B = (V_1, V_2)$, we can construct a flow-graph G by adding two vertices s and t, adding a directed edge from s to each of the vertices in V_1, adding a directed edge from each of the vertices in V_2 to t, giving each original edge in B a direction from V_1 to V_2, and setting the capacity of each edge in G to 1. The resulting flow-graph G has some very interesting properties that can be characterized as follows.

Definition 7.3 A flow graph G is a *simple flow-graph* if it satisfies the following two conditions:

1. the capacity of each edge of G is 1; and
2. every vertex $v \neq s, t$ either has only one incoming edge or has only one outgoing edge.

Clearly, the flow-graph G constructed above from a bipartite graph is a simple flow-graph. Now consider Dinic's algorithm on a simple flow-graph G.

Algorithm 7.1 `Dinic's Algorithm`

```
1.  $f = 0$;
2.  Construct $G_f$;
3.  while there is a path from $s$ to $t$ in $G_f$ do
        construct the leveled graph $L$;
        saturate all the paths in $L$ from $s$ to $t$;
        update the flow $f$;
        construct the new $G_f$;
```

Lemma 7.3 *Let $G = (V, E)$ be a simple flow-graph and let f be a flow on G such that $f(v, w)$ is either 1 or 0 for all pairs (v, w) of vertices in G. Then the residual graph G_f is also a simple flow-graph.*

PROOF. Consider any vertex w in G, $w \neq s, t$. Suppose that the vertex w has only one incoming edge $e = (v, w)$.

If $f(v, w) = 0$ then $f(w, u) = 0$ for all $u \in V$. Thus, in the residual graph G_f, e is still the only incoming edge for the vertex w.

If $f(v, w) = 1$ then there must be an outgoing edge (w, u) of w such that $f(w, u) = 1$, and for all other outgoing edges (w, u') we must have $f(w, u') = 0$. Therefore, in the residual graph G_f, the edge (v, w) disappears and we add another outgoing edge (w, v), and the edge (w, u) disappears and we add a new incoming edge (u, w), which is the unique incoming edge of the vertex w in G_f.

The case that the vertex w has only one outgoing edge can be proved similarly. □

Lemma 7.4 *Let $G = (V, E)$ be a simple flow-graph, and let f be a max-flow on G, let l be the length of the shortest path from s to t in G. then $l \le n/|f| + 1$, where n is the number of vertices in G.*

PROOF. Define V_i to be the set of vertices of distance i from s in G.

Fix an i, $0 \le i \le l - 1$. Define C_1 and C_2 by

$$C_1 = \bigcup_{j=0}^{i} V_j \qquad \text{and} \qquad C_2 = V - C_1$$

It is clear that (C_1, C_2) is a cut of the flow graph G.

We claim that for any edge $e = (v, w)$ of G such that $v \in C_1$ and $w \in C_2$, we have $v \in V_i$ and $w \in V_{i+1}$. In fact, if $v \in V_h$ for some $h < i$, then the distance from s to w cannot be larger than $h + 1 \le i$. This would imply that w is in C_1. Thus, v must be in V_i. Now since $e = (v, w)$ is an edge in G, w is in V_k for some $k \ge i + 1$, and v is in V_i, we must have $w \in V_{i+1}$. This observation together with Lemma 7.1 gives us

$$|f| = \sum_{v \in C_1, w \in C_2} f(v, w) \le \sum_{v \in V_i, w \in V_{i+1}} f(v, w)$$

Now since G is a simple flow graph, there is at most one unit flow through a vertex $v \neq s, t$. Therefore, if $i = 0$ (i.e., $V_i = \{s\}$), then $|f| \le |V_{i+1}|$, and if $i = l + 1$ (i.e., $V_{i+1} = \{t\}$), then $|f| \le |V_i|$, and for $0 < i < l + 1$, we have both $|f| \le |V_{i+1}|$ and $|f| \le |V_i|$. Summarizing these inequalities for all i, we get

$$n = |V| \ge |V_1| + |V_2| + \cdots + |V_{l-1}| \ge (l - 1)|f|$$

which gives immediately $l \le n/|f| + 1$. □

Now we are ready for analyzing the complexity of Dinic's algorithm on simple flow-graphs.

Lemma 7.5 *For simple flow-graphs, the constructed leveled graph L in Dinic's algorithm can be saturated in time $O(e)$.*

PROOF. The saturating is based on a depth first search process, starting from the source s. Any subtree constructed during the depth first search can be entirely deleted if it does not lead to the sink t. Moreover, once a path from s to t is found, all edges on the path will be saturated because all edges in a simple graph have capacity 1. Therefore, in this process, each edge is processed at most twice then will be deleted from the leveled graph L. This concludes that the running time of the saturating process can be done in time $O(e)$. $\square$

Since other steps in the **while** loop body of Dinic's algorithm can be easily done in time $O(e)$, we conclude that each execution of the **while** loop body of Dinic's algorithm takes time $O(e)$.

Lemma 7.6 *On a simple flow-graph, the* **while** *loop body of Dinic's algorithm is executed at most $2\sqrt{n}+1$ times, where n is the number of vertices in the simple flow-graph.*

PROOF. Let h be the number of times the **while** loop body of Dinic's algorithm is executed on a simple flow graph G of n vertices. Let $f_{\max}$ be a maximum flow on G.

If $|f_{\max}| \leq 2\sqrt{n}$, then of course the loop body is executed at most $2\sqrt{n}$ times since each execution of the loop body increases the flow value by at least 1.

Now assume $|f_{\max}| > 2\sqrt{n}$. Let k_0 be the largest integer such that after k_0 executions of the **while** loop body, the flow f_0 constructed in Dinic's algorithm is still less than $|f_{\max}| - \sqrt{n}$. A few interesting facts about k_0 are

- $k_0 < h$;
- after $(k_0 + 1)$st execution of the **while** loop body in Dinic's algorithm, the constructed flow is at least $|f_{\max}| - \sqrt{n}$;
- the value of the maximum flow in the flow graph G_{f_0} is larger than $\sqrt{n}$.

By the third fact, the length of the shortest path from s to t in the flow graph G_{f_0} is bounded by $n/\sqrt{n} + 1 = \sqrt{n} + 1$. Now since each execution of the **while** loop body increases the length of the shortest path from s to

t by at least 1 (see Claim 3 in the proof of Theorem 4.1), we conclude that $k_0 \leq \sqrt{n}$.

By the second fact, after $(k_0 + 1)$st execution of the **while** loop body in Dinic's algorithm, the constructed flow f_1 is at least $|f_{\max}| - \sqrt{n}$. Therefore, with another $\sqrt{n}$ executions of the **while** loop body, starting from the flow-graph G_{f_1}, Dinic's algorithm must reach the maximum flow value $f_{\max}$ because each execution of the **while** loop body increases the flow value by at least 1.

In conclusion, we have $h \leq k_0 + 1 + \sqrt{n} \leq 2\sqrt{n} + 1$. This completes the proof. $\square$

Theorem 7.7 *Dinic's algorithm runs in time $O(\sqrt{n}e)$ on a simple flow graph of n vertices and e edges.*

PROOF. Follows directly from Lemma 7.5 and Lemma 7.6. $\square$

Corollary 7.8 *The maximum matching problem on bipartite graphs can be solved in time $O(\sqrt{n}e)$.*

CPSC-669 Computational Optimization

Lecture #8, September 15, 1995

Lecturer: Professor Jianer Chen
Scribe: Li Shao
Revision: Jianer Chen

8 Maximum matching for general graphs

Now we study the maximum matching problem on general graphs. Recall that a matching M on a graph $G = (V, E)$ is a subset of edges in E such that no two edges in M share a common endpoint. A vertex v is a *matched vertex* if v is an endpoint of an edge in M, otherwise, the vertex is an *unmatched vertex.*

Definition 8.1 Let M be a matching in a graph G. An *alternating path* is a simple path $p = \{u_0, u_1, u_2, \ldots\}$ such that the vertex u_1 is unmatched and that the edges (u_{2i-1}, u_{2i}) are in M, for $i = 1, 2, \ldots$. An alternating path is an *augmenting path* if it starts and ends with unmatched vertices.

Note that alternating paths and augmenting paths are relative to a fixed matching M. The following theorem serves as a fundamental theorem in graph matching.

Theorem 8.1 *Let G be a graph and let M be a matching in G. M is maximum if and only if there is no augmenting path in G.*

PROOF. Suppose that there is an augmenting path $p = (u_0, u_1, \ldots, u_r)$ in the graph G with respect to the matching M.

It is easy to see that the length r of p is odd. Let $r = 2h + 1$, where h is an integer. Consider the set of edges $M' = M \oplus p$, where $\oplus$ is the *symmetric difference* defined by $A \oplus B = (A - B) \cup (B - A)$. Since the number of edges on p that are in M is one less than the number of edges on p that are not in M, the number of edges in M' is one more than that in M. It is also easy to check that M' is also a matching in G: $M' = M \oplus p = (M - p) \cup (p - M)$, for any two edges e_1 and e_2 in M', (1) if both e_1 and e_2 are in $M - p$ then they are in M so have no common endpoint because M is a matching; (2) if both e_1 and e_2 are in $p - M$ then e_1 and e_2 have no common endpoint because p is alternating; and (3) if e_1 is in $M - p$ and e_2 is in $p - M$ then e_1 cannot have an endpoint on p since the two endpoints of p are unmatched and all other vertices on p are matched by edges on p.

Therefore, M' is a matching larger than the matching M. This proves that if there is an augmenting path p, then the matching M cannot be maximum.

Conversely, suppose that the matching M is not maximum. Let $M_{\max}$ be a maximum matching. Then $|M_{\max}| > |M|$. Consider the graph $G_0 = M_{\max} \oplus M = (M - M_{\max}) \cup (M_{\max} - M)$. No vertex in G_0 has degree larger than 2. In fact, if a vertex v in G_0 had degree larger than 2, then at least two edges incident on v belong to either M or $M_{\max}$, contradicting the fact that both M and $M_{\max}$ are matchings in G. Therefore, each component of G_0 must be either a simple path, or a simple cycle. In each simple cycle in G_0, the number of edges in $M_{\max} - M$ should be exactly the same as the number of edges in $M - M_{\max}$. For each simple path in G_0, either the number of edges in $M - M_{\max}$ is the same as the number of edges in $M_{\max} - M$ (in this case, the path has an even length), or the number of edges in $M - M_{\max}$ is one more than the number of edges in $M_{\max} - M$, or the number of edges in $M_{\max} - M$ is one more than the number of edges in $M - M_{\max}$. Since $|M_{\max}| > |M|$, we conclude that there is at least one path $p = (u_1, u_2, \ldots, u_{2h+1})$ in G_0 in which the number of edges in $M_{\max} - M$ is one more than the number of edges in $M - M_{\max}$. Note that the endpoint u_1 of the path p must be unmatched in M. In fact, since $(u_1, u_2) \in M_{\max} - M$, if u_1 is matched in M by an edge e, we must have $e \neq (u_1, u_2)$. Now since u_1 has degree 1 in G_0, $e \notin G_0$, e is also contained in $M_{\max}$. This would make the vertex u_1 incident on two edges (u_1, u_2) and e in the matching $M_{\max}$. Similar reasoning shows that the vertex u_{2h+1} is also unmatched in M. In consequence, the path p is an augmenting path in the graph G with respect to the matching M.

This completes the proof. $\square$

Based on the above theorem, a maximum matching algorithm can be given as follows.

Algorithm 8.1 `Max-matching for general graphs`

```
1.  $M = \phi$;
2.  **while** there is an augumenting path in $G$ **do**
      find an augmenting path $p$;
      construct the matching $M = M \oplus p$ with one more edge;
```

Since a matching in a graph G of n vertices cannot contain more than $n/2$ edges, the **while** loop in the above algorithm will be executed at most $n/2$ times. In the next lectures, we will show how an augmenting path can be constructed when a matching is given for a graph.

CPSC-669 Computational Optimization

Lecture #9, September 18,1995

Lecture: Professor Jianer Chen
Scribe: Shijin Lu
Revision: Jianer Chen

9 Theorems on maximum matching problem

Let us first review the fundamental theorem and algorithm for maximum matching for general graphs.

Theorem 9.1 *Let G be a graph and let M be a matching in G. M is maximum if and only if there is no augmenting path in G.*

Algorithm 9.1 `Max-matching for general graphs`

```
1.  $M = \phi$;
2.  while there is an augumenting path in $G$ do
        find an augmenting path $p$;
        Let $M = M \oplus p$;
```

All known algorithms for maximum matching of general graphs are based on Theorem 9.1 and Algorithm 9.1. The main point here is how an augmenting path can be found. For the rest of the discussion, we assume that G is a fixed graph and that M is a fixed matching in G.

Observe that an augmenting path P must start with an unmatched vertex v_0. The next vertex v_1 must be a neighbor of v_0. If v_0 is also unmatched, then the edge (v_0, v_1) constitutes an augmenting path. On the other hand, if the length of P is larger than 1, then the third vertex v_2 on P must be the one that matches v_1 in M. Now since the path $\{v_0, v_1, v_2\}$ does not make an augmenting path, the fourth vertex v_3 must be a neighbor of the vertex v_2, and so on. Therefore, it seems that we can search the augmenting path using a breadth first search manner: start with v_0, then search all neighbors of v_0, then search all vertices that match the neighbors of v_0, and so on until we find an unmatched vertex. In this search, we give each vertex v a level number $level[v]$ such that all roots of the breadth first search trees are at level 0, and that the children of a vertex at level i are at level $i+1$. A vertex will be called an *even level vertex* or an *odd level vertex* according to its level number. On each even level vertex v_{2h}, we search all neighbors of v_{2h}, and on each odd level vertex v_{2h+1}, we only take the unique vertex v_{2h+2} such that the edge (v_{2h+1}, v_{2h+2}) is in the matching M.

Another modification we will made is that we will perform this BFS fashion search starting from *all* unmatched vertices at the same time, instead of starting from a single vertex. Implementation of this modification is simple: as for the standard BFS, we use a first-in first-out queue. However, we first put all unmatched vertices in the queue then perform the BFS fashion search until either an augmenting path is found or the queue Q is empty. It is easy to see that this search will first construct the first level for all BFS trees rooted at the unmatched vertices, then the second level for all BFS trees, and so on.

Remark 9.1 These modifications make the BFS trees lose many of their well-known and nice properties. The following lost properties should be mentioned:

(1) in the modified BFS process, a cross-edge (i.e., an edge of G that links two vertices in the BFS trees that do not have a father-son relation) may link two vertices whose level numbers differ by an arbitrarily large number. On the other hand, in the standard BFS process, each cross-edge links two vertices whose level numbers differ by at most 1;

(2) in the modified BFS process, a tree path from an ancestor to a descendent may no longer be a shortest path between the two vertices; and

(3) in the modified BFS process, a tree may not necessarily contain all vertices in a connected component of the graph G. In fact, now there may be cross-edges that link two vertices in two different BFS trees.

We present our first draft of the algorithm.

Algorithm 9.2 `Modified BFS (Version 1)`

```
1.  put all unmatched vertices in the queue $Q$;
2.  **while** no augmenting path has been found **do**
        Let $v$ be the next vertex in the queue $Q$;
        **if** $v$ is an even level vertex
        **then** make all unvisited neighbors of $v$ the children
           of $v$ and add them to $Q$
        **else** {$v$ is an odd level vertex.}
           if the vertex $w$ that matches $v$ is unvisited,
           then make $w$ a child of $v$ and add $w$ to $Q$.
```

Algorithm 9.2 does not describe the details of how an augmenting path can be found, which is discussed in the rest of this section.

Definition 9.2 In the modified BFS process, a cross-edge e is a *good cross-edge* if either $e \in M$ and e links two odd level vertices in two different BFS trees, or $e \notin M$ and e links two even level vertices in two different BFS trees.

Lemma 9.2 *If a good cross-edge is given in the modified BFS process, then an augmenting path can be constructed in linear time.*

PROOF. Let $e = (v_{2s+1}, u_{2t+1})$ be a good cross-edge such that $e \in M$, $\{v_0, v_1, \ldots, v_{2s+1}\}$ is a tree path in a BFS tree T_v from root v_0 to v_{2s+1}, and $\{u_0, u_1, \ldots, u_{2t+1}\}$ is a tree path in a BFS tree T_u from root u_0 to u_{2t+1}, where $v_0 \neq u_0$. By the modified BFS process, v_0 and u_0 are unmatched vertices, the edges (v_{2i}, v_{2i+1}) and (u_{2j}, u_{2j+1}) are not in M, for all $i = 0, \ldots, s$ and $j = 0, \ldots, t$, and the edges (v_{2i+1}, v_{2i+2}) and (u_{2j+1}, u_{2j+2}) are in M, for all $i = 0, \ldots, s-1$ and $j = 0, \ldots, t-1$. Therefore, the path

$$\{v_0, v_1, \ldots, v_{2s}, v_{2s+1}, u_{2t+1}, u_{2t}, \ldots, u_1, u_0\}$$

is an augmenting path that can be easily constructed in linear time.

The case that the good cross-edge is not in M and links two even level vertices can be proved similarly. □

Lemma 9.2 suggests a refinement of Algorithm 9.2.

Algorithm 9.3 `Modified BFS (Version 2)`

```
1.  put all unmatched vertices in the queue $Q$;
2.  while no augmenting path has been found do
        Let $v$ be the next vertex in the queue $Q$;
        if $v$ is an even level vertex then
            for each neighbor $w$ of $v$ do
               if $(v, w)$ is a good cross-edge then
                  construct an augmenting path and stop;
               if $w$ is unvisited then
                  make $w$ a child of $v$ and add $w$ to $Q$;
        else {$v$ is an odd level vertex.}
            let $w$ be the vertex matching $v$;
            if $(v, w)$ is a good cross-edge then
```

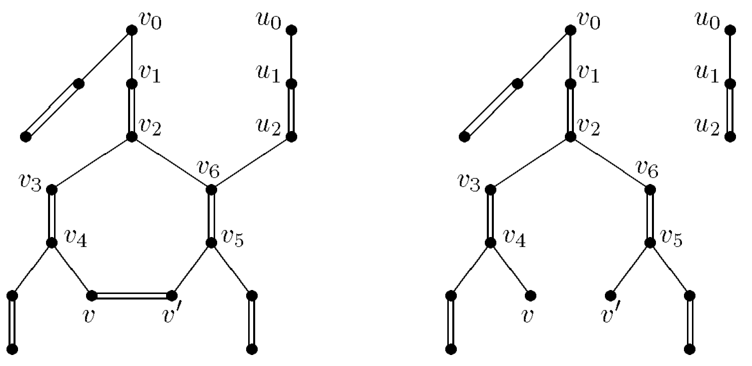

Figure 1: The structure of a blossom

```
        construct an augmenting path and stop;
    if $w$ is unvisited then
        make $w$ a child of $v$ and add $w$ to $Q$;
```

for years people believed that Algorithm 9.3 was sufficient for constructing an augmenting path until the following structure was discovered.

Definition 9.3 In the modified BFS process, a cross-edge e is a *bad cross-edge* if either $e \in M$ and e links two odd level vertices in the same BFS tree, or $e \notin M$ and e links two even level vertices in the same BFS tree.

Let us consider how a bad cross-edge may make the modified BFS process fail to find an existing augmenting path. Let $e = \{v, v'\}$ be an edge in M such that both v and v' are odd level vertices in the same BFS tree. When we first encounter the edge e in the modified BFS process, both vertices v and v' have been visited. Therefore, the edge cannot be added to the BFS tree. However, the edge cannot be simply ignored since it may "hide" an augmenting path from our BFS process. Consider the case in Figure 1(a), where each single line represents an edge not in the matching M and each double line represents a matched edge in M.

The only two unmatched vertices in Figure 1(a) are v_0 and u_0. Thus, the modified BFS process starts with v_0 and u_0 being in the queue Q and stops as in Figure 1(b) without finding an augmenting path. Note that the edge (v_6, u_2) is not included in the BFS trees because at time u_2 is expanding, v_6 has been visited through v_2, and at time v_6 is expanding, only the matched edge (v_6, v_5) is considered. The edge (v, v') is also not included in the BFS trees because at time v (resp. v') is processed, v' (resp. v) has been visited. However, there is clearly an augmenting path from v_0 to u_0:

$$\{v_0, v_1, v_2, v_3, v_4, v, v', v_5, v_6, u_2, u_1, u_0\}$$

In a similar case when e is a bad edge such that e is not in M and e links two even level vertices in the same BFS tree, we can also construct a configuration in which the modified BFS process fails to find an existing augmenting path.

This discussion motivates the following definition.

Definition 9.4 In the modified BFS process, a *blossom* is a simple cycle consisting of a bad cross-edge $e = (v, v')$ together with the two unique tree paths from v and v' to their least common ancestor v''. The vertex v'' will be called the *base* of the blossom.

For example, the cycle $\{v_2, v_3, v_4, v, v', v_5, v_6, v_2\}$ is a blossom whose base is v_2.

Remark 9.5 There are a number of interesting properties for blossoms. We list those that are related to our later discussion.

(1). A blossom consists of an odd number of vertices. This is because either both ends v and v' of the bad cross-edge are odd level vertices or both v and v' are even level vertices.

(2). Suppose that the cycle $b = \{v_0, v_1, \ldots, v_{2s}, v_0\}$ is a blossom, where v_0 is the base, then the edges (v_{2s}, v_0) and (v_{2i}, v_{2i+1}) for all $i = 0, \ldots, s-1$, are not in the matching M, and the edges (v_{2j-1}, v_{2j}) for all $j = 1, \ldots, s$ are in the matching M.

(3). If an edge e_0 is not contained in a blossom but is incident to a vertex v in the blossom, then the edge e_0 cannot be in the matching M unless the incident vertex v is the base of the blossom. This is because each vertex, except the base, in a blossom is incident on a matched edge in the blossom.

Identifying and constructing a blossom is easy, as stated in the following lemma.

Lemma 9.3 *In linear time, we can identify a bad cross-edge and construct the corresponding blossom.*

PROOF. In the modified BFS process, we keep track of the level number for each visited vertex. Once a cross-edge $e = (v, v')$ is found, we can follow the tree edges back to the root to check whether the two ends v and v' of e belong to the same BFS tree. This together with the level numbers of v and v' is sufficient for deciding if e is a bad cross-edge. For a bad cross-edge, we trace the two tree paths back from the common root to find the last common vertex v'' on the paths. The vertex v'' is the base for the blossom. $\square$

Thus, blossoms are structures that may make the modified BFS process fail. Is there any other structure that can also fool the modified BFS process? Fortunately, blossoms are the only such structures, as we will discuss below. We start with the following lemma.

Lemma 9.4 *If a matched edge in M is a cross-edge, then it is either a good cross-edge or a bad cross-edge.*

PROOF. Let $e = (v, v')$ be a matched edge that is a cross-edge. The vertices v and v' cannot be roots of the BFS trees since roots of the BFS trees are unmatched vertices. Let w and w' be the fathers of v and v', respectively. The tree edges (w, v) and (w', v') are not matched edges since (v, v') is a matched edge. Thus, v and v' must be odd level vertices. Now if v and v' belong to different BFS trees, then the edge e is a good cross-edge, otherwise e is a bad cross-edge. $\square$

Lemma 9.5 *If there is no blossom in the modified BFS process, then there is a good cross-edge if and only if there is an augmenting path.*

PROOF. By Lemma 9.2, if there is a good cross-edge, then there is an augmenting path that can be constructed from the good cross-edge in linear time.

Conversely, suppose there is an augmenting path $p = \{u_0, u_1, \ldots, u_{2t+1}\}$. If $t = 0$, then the path p itself is a good cross-edge and we are done. Thus, assume $t > 0$. Let v_1, ..., v_h be the roots of the BFS trees, processed in that order by the modified BFS process. Without loss of generality, we assume $u_0 = v_b$ where b is the smallest index such that v_b is an end of an augmenting path. With this assumption, the vertex u_1 is a child of u_0 in the BFS tree rooted at u_0. If any matched edge e on p is a cross-edge, then by Lemma 9.4, e is either a good cross-edge or a bad cross-edge. Since there is no blossom, e must be a good cross-edge again the lemma is proved.

Thus, we assume that the augmenting path p has length larger than 1, no matched edges on p are cross-edges, and u_1 is a child of u_0 in the BFS tree rooted at u_0.

Case 1. Suppose that all vertices on p are contained in the BFS trees.

Both u_0 and u_{2t+1} are even level vertices. Since the path p is of odd length, there must be an index i such that $level[u_{i-1}] = level[u_i] \bmod 2$. Without loss of generality, assume i is the smallest index satisfying this condition. The edge (u_{i-1}, u_i) must be a cross-edge. Thus, by our assumption, (u_{i-1}, u_i) is not a matched edge.

Suppose that both u_{i-1} and u_i are odd level vertices, then $i \geq 2$. Since (u_{i-2}, u_{i-1}) is a matched edge, $u_{i-2} \neq u_0$. Moreover, by our assumption, (u_{i-2}, u_{i-1}) is a tree edge. Thus, u_{i-2} is an even level vertex. Moreover, since (u_{i-2}, u_{i-1}) is a matched edge, the index $i-2$ is an odd number. Now the partial path

$$p_{i-2} = \{u_0, u_1, \ldots, u_{i-2}\}$$

is of odd length and has both ends being even level vertices. This implies that there is an index j such that $j \leq i-2$ and $level[u_{j-1}] = level[u_j] \bmod 2$. But this contradicts the assumption that i is the smallest index satisfying this condition.

Thus, u_{i-1} and u_i must be even level vertices. So (u_{i-1}, u_i) is either a good cross-edge or a bad cross-edge. By the assumption of the lemma, there is no blossom. Consequently, (u_{i-1}, u_i) must be a good cross-edge and the lemma is proved for this case.

Case 2. Some vertices on p are not contained in any BFS trees.

Let u_i be the vertex on p with minimum i such that u_i is not contained in any BFS trees. Then $i \geq 2$.

Suppose $(u_{i-1}, u_i) \in M$. If u_{i-1} is an odd level vertex then u_i would have been made a child of u_{i-1}. Thus u_{i-1} is an even level vertex. However, since u_{i-1} cannot be a root of a BFS tree, u_{i-1} would have matched its father in the BFS tree, this contradicts the assumption that u_{i-1} matches u_i and u_i is not contained in any BFS trees.

Thus we must have $(u_{i-1}, u_i) \notin M$. Then (u_{i-2}, u_{i-1}) is in M. Thus, the index $i-2$ is an odd number. By our assumption, (u_{i-2}, u_{i-1}) is a tree edge. If u_{i-1} is an even level vertex, then u_i would have been made a child of u_{i-1}. Thus, u_{i-2} is an even level vertex. Now in the partial path of odd length

$$p_{i-2} = \{u_0, u_1, \ldots, u_{i-2}\},$$

all vertices are contained in the BFS trees, and the two ends are even level vertices. Now the proof goes exactly the same as for **Case 1** — we can find a smallest index $j \leq i-2$ such that $level[u_{j-1}] = level[u_j] \bmod 2$ and (u_{j-1}, u_j) is a good cross-edge.

This completes the proof of the claim. $\square$

By Lemma 9.5, if there is an augmenting path and if no bad cross-edge is found (thus no blossom is found), then the modified BFS process will eventually find a good cross-edge. By Lemma 9.2, an augmenting path can be constructed in linear time from this good cross-edge. In particular, if the graph is bipartite, then the modified BFS process will always be able to construct an augmenting path if one exists, since a bipartite graph contains no odd length cycle, thus no blossom can appear in the modified BFS process. This gives the well-known algorithm of running time $O(ne)$ for maximum matching on bipartite graphs.

In order to develop an efficient algorithm for maximum matching on general graphs, we need to resolve the problem of blossoms. Surprisingly, the solution to this problem is not very difficult, based on the following "blossom shrinking" technique.

Definition 9.6 Let G be a graph and M a matching in G. Let b be a blossom found in the modified BFS process. Define G/b to be the graph obtained from G by "shrinking" the blossom b. That is, G/b is a graph obtained from G by deleting all vertices (and their incident edges) of the blossom b then adding a new vertex v_b that is connected to all vertices that are adjacent to some vertices in b in the original graph G.

It is easy to see that given the graph G and the blossom b, the graph G/b can be constructed in linear time.

Since there is at most one matched edge that is incident to but not contained in a blossom, for a matching M in G, the edge set $M - b$ is a matching in the graph G/b.

Theorem 9.6 (Edmond) *Let G be a graph and M a matching in G. Let b be a blossom in G. Then there is an augmenting path in G with respect to M if and only if there is an augmenting path in G/b with respect to $M - b$.*

PROOF. Suppose that the blossom is $b = \{v_0, v_1, \ldots, v_s, v_0\}$, where v_0 is the base. We first show that the existence of an augmenting path in G/b

implies an augmenting path in G. Let $p = \{u_0, u_1, \ldots, u_t\}$ be an augmenting path in G/b and let v_b be the new vertex in G/b obtained by shrinking b.

Case 1. If the vertex v_b is not on the path p, then clearly p is also an augmenting path in G.

Case 2. Suppose $v_b = u_t$. Then v_b is an unmatched vertex in the matching $M - b$. Consequently, the base v_0 of the blossom b is unmatched in the matching M.

If the edge (u_{t-1}, u_t) in G/b corresponds to the edge (u_{t-1}, v_0) in G, then the path

$$p_1 = \{u_0, u_1, \ldots, u_{t-1}, v_0\}$$

is an augmenting path in G.

If the edge (u_{t-1}, u_t) in G/b corresponds to the edge (u_{t-1}, v_h) in G, where v_h is not the base of b, then one of the edges (v_{h-1}, v_h) and (v_h, v_{h+1}) is a matched edge. Without loss of generality, suppose that (v_h, v_{h+1}) is a matched edge. Then, the path

$$p_2 = \{u_0, u_1, \ldots, u_{t-1}, v_h, v_{h+1}, \ldots, v_s, v_0\}$$

is an augmenting path in G.

The case $v_b = u_0$ can be proved similarly.

Case 3. Suppose that $v_b = u_d$, where $0 < d < t$. Then without loss of generality, we assume that (u_{d-1}, u_d) is a matched edge in $M - b$ and (u_d, u_{d+1}) is an unmatched edge. The edge (u_{d-1}, u_d) in G/b must correspond to the matched edge (u_{d-1}, v_0) in G. Let the edge (u_d, u_{d+1}) in G/b correspond to the edge (v_h, u_{d+1}) in G.

If $v_h = v_0$, then the path

$$p_2 = \{u_0, \ldots, u_{d-1}, v_0, u_{d+1}, \ldots, u_t\}$$

is an augmenting path in G.

If $v_h \neq v_0$, then as we proved in **Case 2**, we can assume that (v_{h-1}, v_h) is a matched edge. Thus, the path

$$p_3 = \{u_0, \ldots, u_{d-1}, v_0, v_1, \ldots, v_{h-1}, v_h, u_{d+1}, \ldots, u_t\}$$

is an augmenting path in G.

Therefore, given an augmenting path in G/b, we are always able to construct an augmenting path in G.

The proof for the other direction that the existence of an augmenting path in G implies an augmenting path in G/b is rather complicated based on a case by case analysis. We omit the proof here. $\square$

Corollary 9.7 *Let G be a graph of n vertices and let b be a blossom. Given an augmenting path in G/b, an augmenting path in G can be constructed in time $O(n)$.*

PROOF. Directly follows from the construction given in the proof of Theorem 9.6. $\square$

CPSC-669 Computational Optimization

Lecture #10, September 20, 1995

Lecturer: Professor Jianer Chen
Scribe: Shijin Lu
Revision: Jianer Chen

10 Algorithms for maximum matching problem

We first review two theorems given in the last lecture.

Theorem 10.1 *Suppose we perform the modified BFS process. If there is an augmenting path and there are no blossoms, then the BFS process will find an augmenting path.*

Theorem 10.2 *If a blossom b is found in the modified BFS process, then there is an augmenting path in G with respect to the matching M if and only if there is an augmenting path in G/b with respect to the matching $M - b$. Moreover, an augmenting path in G can be constructed in time $O(n)$ if an augmenting path in G/b is given.*

Now the idea is fairly clear for how we can find an augmenting path: we perform the modified BFS process, either we find an augmenting path in G then we are done, or we find a blossom then we shrink the blossom and search an augmenting path in G/b. Once an augmenting path in G/b is found, we can easily convert it into an augmenting path in G, as stated in Corollary 9.7. Theorem 9.6 ensures that if we cannot find an augmenting path in G/b then there is no augmenting path in G.

The main algorithm for constructing a maximum matching for a general graph now can be rewritten as follows.

Algorithm 10.1 Maximum Matching

```
Input: a graph $G$;
Output: a maximum matching $M$ in $G$
1. $M = \phi$.
2. repeat
     if there is an augmenting path in $G$ w.r.t.$M$
     then
```

```
        construct an augmenting path $p$;
        let $M = M \oplus p$;
    until no augmenting path is found
```

The process of finding an augmenting path is implemented by the modified BFS process as follows.

Algorithm 10.2 Finding An Augmenting Path

```
1.  Perform the modified BFS process;
2.  if an augmenting path is found then
        convert it to an augmenting path for the original
        graph $G$, stop;
3.  if a blossom is found then
        construct the graph $G/b$, resume the modified BFS
        process;
```

To give a more detailed description for the modified BFS process, we give a level number $level[v]$ to each vertex v. Initially, $level[v] = -1$ for all vertices v. Thus, a vertex v is visited if and only if its level number is larger than -1.

Algorithm 10.3 Finding An Augmenting Path (Refined)

```
Input:  a graph $G$ and a matching $M$ in $G$
Output:  an augmenting path in $G$, or report no such a path

1.  for all vertices $w$ of $G$ do $level[w] = -1$;
2.  for all unmatched vertices $w$ in $G$ do
      $level[w] = 0$;     $Q \longleftarrow w$;
3.  while the queue $Q$ is not empty do
      $v \longleftarrow Q$;
      if $level[v]$ is even then
          for each neighbor $w$ of $v$ do
              if $(v, w)$ is a good cross-edge then
                  construct an augmenting path;
                  convert it into an augmenting path in $G$;
                  stop;
              if $(v, w)$ is a bad cross-edge then
                  construct the blossom $b$ based on $(v, w)$;
                  construct the graph $G/b$;
```

```
                update the queue $Q$ properly
                go back to the beginning of Step 3;
              if $level[w] = -1$ then
                make $w$ a child of $v$;
                $level[w] = level[v] + 1$;
                $Q \longleftarrow w$;
        else {$v$ is an odd level vertex.}
            let $w$ be the vertex matching $v$;
            if $(v, w)$ is a good cross-edge then
              construct an augmenting path;
              convert it into an augmenting path in $G$;
              stop;
            if $(v, w)$ is a bad cross-edge then
              construct the blossom $b$ based on $(v, w)$;
              construct the graph $G/b$;
              update the queue $Q$ properly
              go back to the beginning of Step 3;
            if $level[w] = -1$ then
              make $w$ a child of $v$;
              $level[w] = level[v] + 1$;
              $Q \longleftarrow w$;
4.  {At this point, the modified BFS is finished without
    finding an augmenting path}
     return ("no augmenting path")
```

The correctness of Algorithm 10.3 is ensured by Lemma 9.5 and Theorem 9.6.

Lemma 10.3 *Algorithm 10.3 runs in time $O(ne)$ on a graph of n vertices and e edges.*

PROOF. A BFS process takes time $O(e)$.

If a blossom b is found, the graph G/b is constructed and the queue Q is updated. It is easy to see that constructing G/b and updating Q can be done in time $O(e)$. Moreover, the the number of vertices in the graph G/b is at least two less than the number of vertices in the graph G. Therefore, there are at most $n/2$ blossoms found in Algorithm 10.3, and for each blossom it takes time $O(e)$ for Algorithm 10.3 to update the graph and the queue. Therefore, the total time spent by Algorithm 10.3 on processing blossoms is bounded by $O(ne)$.

Once an augmenting path is found, by Corollary 9.7, in time $O(n)$ we can expand a vertex back to a blossom and construct an augmenting path for the new graph. Since there are at most $n/2$ such blossom restoration operations, the total time Algorithm 10.3 spends on constructing an augmenting path for the original graph is bounded by $O(n^2)$. This proves the time bound for the algorithm stated in the lemma. □

Theorem 10.4 *The maximum matching problem on general graphs can be solved in time $O(n^2e)$.*

PROOF. By Lemma 10.3, an augmenting path can be found in time $O(ne)$. Since each augmenting path increases one edge for the matching, and there are no more than $n/2$ edges in a matching for a graph of n vertices, the **repeat** loop in Algorithm 10.1 is executed at most $O(n)$. The theorem follows. □

We should point out that $O(n^2e)$ is not the best upper bound for the maximum matching problem. In fact, a moderate change in Algorithm 10.3 gives an algorithm of running time $O(n^3)$ for the problem. The basic idea for this change is that instead of actually shrinking the blossoms, we keep track of all vertices in a blossom by "marking" them. A careful bookkeeping technique shows that this can be done in time $O(n)$ per blossom. The best known algorithm for the maximum matching problem on general graphs runs in time $O(\sqrt{n}e)$, thus matching the best known algorithm for the maximum matching problem on bipartite graphs.

CPSC-669 Computational Optimization

Lecture #11, September 22, 1995

Lecturer: Professor Jianer Chen
Scribe: Hao Zheng
Revision: Jianer Chen

11 Linear programming problem

The *Linear Programming Problem* is to find a vector $(x_1, x_2, ..., x_n) \in R^n$ such that a linear function $c_1x_1 + c_2x_2 + ... + c_nx_n$, which is called an *objective function*, is optimized (maximized or minimized) and the vector $(x_2, x_2, ..., x_n)$ satisfies a given set of conditions (these conditions are called *linear constraints*).

$$\begin{array}{rcl}
a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n & \geq & a_1 \\
\cdots\cdots & & \\
a_{r1}x_1 + a_{r2}x_2 + ... + a_{rn}x_n & \geq & a_r \\
b_{11}x_1 + b_{12}x_2 + ... + b_{1n}x_n & \leq & b_1 \\
\cdots\cdots & & \\
b_{s1}x_1 + b_{s2}x_2 + ... + b_{sn}x_n & \leq & b_s \\
d_{11}x_1 + d_{12}x_2 + ... + d_{1n}x_n & = & d_1 \\
\cdots\cdots & & \\
d_{t1}x_1 + d_{t2}x_2 + ... + d_{tn}x_n & = & d_t
\end{array}$$

This is called the *general form* of Linear Programming Problem.

Using our 4-tuple formulation, the Linear Programming Problem is given as $LP = \langle I_Q, S_Q, f_Q, opt_Q \rangle$, where

- I_Q is the set of 7-tuples (c, A, B, D, a, b, d), where $c = (c_1, \ldots, c_n)$, $a = (a_1, \ldots, a_r)$, $b = (b_1, \ldots, b_s)$, and $d = (d_1, \ldots, d_t)$ are vectors of real numbers, $A = (a_{i,j})_{r \times n}$, $B = (b_{i,j})_{s \times n}$, and $D = (d_{i,j})_{t \times n}$ are matrices of real numbers, for some positive integers r, s, t, and n.
- for a given $\alpha = (c, A, B, D, a, b, d) \in I_Q$, the solution set $S_Q(\alpha)$ consists of the set of vectors $x = (x_1, \ldots, x_n)$ of real numbers that satisfies the conditions $Ax \geq a$, $Bx \leq b$, and $Dx = d$.

- for a given input instance $\alpha \in I_Q$ and a solution $x \in S_Q(\alpha)$, the objective function value is defined by $f_Q(\alpha, x) = c_1x_1 + \cdots c_nx_n$.

- opt_Q is either max or min.

For many combinatorial optimization problems, the objective function and the constraints on a solution to an input instance are linear, i.e., they can be formulated by linear equations and linear inequalities. Therefore, optimal solutions for these combinatorial optimization problems can be derived from optimal solutions for the corresponding instance in Linear Programming Problem. This is one of the main reasons why Linear Programming Problem receives so much attention from researchers.

Example 11.1 (Maximum Flow) As an example, we show how the Max-Flow Problem is formulated in terms of the Linear Programming Problem.

A flow-graph G of n vertices can be given by n^2 non-negative real numbers $c_{i,j}$, $1 \leq i, j \leq n$, where $c_{i,j}$ is the capacity of the edge from vertex i to vertex j (recall that $c_{i,j} = 0$ if and only if there is no edge from vertex i to vertex j). Here we assume that vertex 1 is the source and vertex n is the sink. Now a flow on G (i.e., a solution to the instance G of Max-Flow Problem) can be given by an n^2-dimensional vector

$$\alpha = (f_{1,1}, \ldots, f_{1,n}, f_{2,1}, \ldots, f_{2,n}, \ldots, f_{n,1}, \ldots, f_{n,n})$$

where $f_{i,j}$ is the amount of flow from vertex i to vertex j. The three conditions that a flow should satisfy are trivially given by

$$\begin{aligned} f_{i,j} &\leq c_{i,j} && \text{for } 1 \leq i, j \leq n \\ f_{i,j} &= -f_{j,i} && \text{for } 1 \leq i, j \leq n \\ \sum_{j=1}^{n} f_{i,j} &= 0 && i \neq 1, n \end{aligned}$$

and the objective function is to maximize the linear function $f_{1,2} + f_{1,3} + \cdots + f_{1,n}$ (or equivalently, to maximize $f_{1,n} + f_{2,n} + \cdots + f_{n-1,n}$).

The *standard form* for Linear Programming Problem is given by

$$\begin{aligned} &\text{minimize } c_1x_1 + c_2x_2 + \cdots + c_nx_n \\ &\qquad a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = a_1 \\ &\qquad a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = a_2 \end{aligned}$$

$$\begin{aligned} &\cdots\cdots \qquad (1)\\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n &= a_m \\ x_1 \geq 0, \quad x_2 \geq 0, \quad \ldots, \quad x_n \geq 0 & \end{aligned}$$

The general form of Linear Programming Problem can be converted into the standard form through the following steps.

1. **Converting Max to Min**

   Maximization $\max\{c_1x_1 + c_2x_2 + \ldots + c_nx_n\}$ can be replaced by the equivalent condition $\min\{(-c_1)x_1 + (-c_2)x_2 + \cdots + (-c_n)x_n\}$.

2. **Eliminating $\leq$ inequalities**

   Each inequality $b_{i1}x_1 + b_{i2}x_2 + \ldots + b_{in}x_n \leq b_i$ is replaced by the equivalent inequality $(-b_{i1})x_1 + (-b_{i2})x_2 + \ldots + (-b_{in})x_n \geq (-b_i)$.

3. **Eliminating $\geq$ inequalities**

   Each inequality $a_{j1}x_1 + a_{j2}x_2 + \ldots + a_{jn}x_n \geq a_j$ is replaced by the inequality $a_{j1}x_1 + a_{j2}x_2 + \ldots + a_{jn}x_n - y_j = a_j$, where y_j is a new variable satisfying $y_j \geq 0$.

4. **Eliminating unconstrained variables**

   For each variable x_i for which $x_i \geq 0$ is not present, introduce two new variables u_i and v_i satisfying $u_i \geq 0$ and $v_i \geq 0$, and replace the variable x_i by $u_i - v_i$.

It is easy to see that the above process will convert Linear Programming Problem from an arbitrary general form to the standard form. It is also easy to verify that an optimal solution for the general form can be easily derived from an optimal solution for the corresponding standard form. Thus, we only need to concentrate on the standard form for Linear Programming Problem.

A classical method, called *Simplex Method* was derived for solving *Linear Programming Problem*. It is based on the following observations. Each equation in the constraints (1) defines a *hyperplane* in the n-dimensional space R^n, so the set of all points in R^n that satisfy the constraints (1) forms a polytope in R^n, which is a convex set.[1] Moreover, the objective

[1]A set S in R^n is *convex* if for any two points x and y in S, the line segment $\overline{xy}$ is entirely in S.

function $c_1x_1 + \cdots + c_nx_n$ is a convex function.[2] Therefore, there is a vertex of the polytope at which the objective function achieves its optimal value, and this vertex can be found using greedy method. Roughly speaking, the Simplex Method starts from an arbitrary vertex of the polytope defined by the linear constraints (1), and uses greedy method to traverse the vertices of the polytope until reaching a vertex at which local improvement is no longer possible. This vertex then is an optimal solution.

In most practical cases, Simplex Method is fast enough to construct an optimal solution for a given instance of Linear Programming Problem. It took a while for researchers to be able to formally prove that in the worst case, Simplex Method runs in exponential time.

It was an outstanding open problem whether Linear Programming Problem could be solved in polynomial time, until the spring of 1979, the Russian mathematician L.G. Khachian published a proof that an algorithm, called *the Ellipsoid Algorithm*, solves Linear Programming Problem in polynomial time. Despite the great theoretical value of the Ellipsoid Algorithm, it is not clear at all that this algorithm can be practically useful. The most obvious among many obstacles is the large precision apparently required.

Another polynomial time algorithm for Linear Programming Problem, called the *Projective Algorithm*, or more generally, the *Interior Point Algorithm*, was published by N. Karmarkar in 1984. The Projective Algorithm, and its derivatives, have great impact in the study of Linear Programming Problem.

[2] A function f from R^n to R is *convex* if for any two points x and y in R^n and for any real number $0 \leq c \leq 1$, we have $f(cx + (1-c)y) \leq cf(x) + (1-c)f(y)$.

CPSC-669 Computational Optimization

Lecture #12, September 25, 1995

Lecturer: Professor Jianer Chen
Scribe: Hao Zheng
Revision: Jianer Chen

12 Integer Linear Programming Problem

Suppose that in Linear Programming Problem, we further require that all numbers are integers, then we get *Integer Linear Programming Problem.* More formally, Integer Linear Programming Problem is to find an n-dimensional vector $\alpha = (x_1, x_2, \ldots, x_n)$ of integers such that

$$\begin{array}{rcl} \text{minimize } c_1x_1 + c_2x_2 + \cdots + c_nx_n & & \\ a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n & = & a_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n & = & a_2 \\ \cdots\cdots & & \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n & = & a_m \\ x_1 \geq 0, \quad x_2 \geq 0, \quad \ldots, \quad x_n \geq 0 & & \end{array} \qquad (2)$$

where all numbers c_i, a_{ij}, a_j, $1 \leq i \leq n$, $1 \leq j \leq m$, are integers. The equations (2) give the *standard form* for Integer Linear Programming Problem. We can similarly define the *general form* for Integer Linear Programming Problem. Moreover, it is not hard to verify that the translation steps described in the previous Lecture Notes convert Integer Linear Programming Problem in general form into Integer Linear Programming Problem in standard form.

It might seem that Integer Linear Programming Problem is easier since we are working on simpler numbers. This intuition is, however, not true. In fact, Integer Linear Programming Problem is computationally harder than general Linear Programming Problem. This may be seen from the following fact: now the point set defined by the constraints (2) is no longer a convex set. It consists of discrete points in the n-dimensional Euclidean space. Therefore, greedy algorithms based on local search do not seem to work any more.

To formally prove the difficulty of Integer Linear Programming Problem, we need introduce some definitions.

Definition 12.1 An optimization problem $Q_1 = (I_1, S_1, f_1, opt_1)$ is *polynomial time reducible* to an optimization problem $Q_2 = (I_2, S_2, f_2, opt_2)$ if there are two polynomial time algorithms A_1 and A_2 such that (1) given an input instance $x_1 \in I_1$ of Q_1, the algorithm A_1 constructs an input instance $x_2 \in I_2$ of Q_2, and (2) for any optimal solution y_2 for the input instance x_2 of Q_2, the algorithm A_2 constructs an optimal solution y_1 for the input instance x_1 of Q_1.

The following theorem follows directly from the definition.

Theorem 12.1 *Suppose that an optimization problem Q_1 is polynomial time reducible to an optimization problem Q_2, then*
(1) *If Q_2 can be solved in polynomial time, then so can Q_1;*
(2) *If Q_1 cannot be solved in polynomial time, then neither can Q_2.*

A problem is a *decision problem* if each input instance of the problem requires only a YES/NO answer. Note that a decision problem can also be regarded as an optimization problem in which the objective function takes only value 0 (NO) or 1 (YES).

Recall that a decision problem is in NP if it can be solved by a nondeterministic algorithm running in polynomial time, and that a problem L in NP is NP-complete if all problems in NP can be polynomial time reducible to L. By Theorem 12.1, if any NP-complete problem is polynomial time solvable, then P = NP. Since people commonly believe that P $\neq$ NP, every NP-complete problem is regarded as not solvable in polynomial time (though there is no formal proof for this conjecture).

Definition 12.2 An optimization problem Q is *NP-hard* if there is an NP-complete problem that is polynomial time reducible to Q.

According to Theorem 12.1, we accept the conjecture that every NP-hard optimization problem is not solvable in polynomial time.

Now we are ready to show the hardness of Integer Linear Programming Problem.

Theorem 12.2 *Integer Linear Programming Problem is NP-hard.*

PROOF. We show that the well known NP-complete problem, the Satisfiability Problem, is polynomial time reducible to the Integer Linear Programming Problem.

Formally, an instance α of the Satisfiability Problem is given by a Boolean expression in conjunctive normal form (CNF):

$$\alpha = C_1 \wedge C_2 \wedge \ldots \wedge C_m \tag{3}$$

where each C_i (called a *clause*) is an OR of Boolean literals. The question is whether there is a Boolean assignment to the Boolean variables x_1, x_2, ..., x_n in α that makes the expression true.

We show how the input instance (3) of Satisfiability Problem is translated into an input instance for Integer Linear Programming Problem.

Suppose that the clause C_i is

$$C_i = (x_{i_1} \vee \cdots \vee x_{i_s} \vee \overline{x}_{j_1} \vee \cdots \vee \overline{x}_{j_t})$$

We then construct a linear constraint

$$x_{i_1} + \cdots + x_{i_s} + (1 - x_{j_1}) + \cdots + (1 - x_{j_t}) \geq 1$$

Moreover, for each Boolean variable x_j in α, we have the constraints

$$x_j \geq 0 \quad \text{and} \quad x_j \leq 1$$

Here we let $x_j = 1$ simulate the assignment $x_j =$ true and let $x_j = 0$ simulate the assignment $x_j =$ false. Therefore, the clause C_i is true if and only if the corresponding linear constraint is satisfied for a 0-1 assignment to the variables x_1, x_2, $\cdots$, x_n.

The objective function of the Integer Linear Programming Problem is irrelevant in this reduction and can be defined arbitrarily. For example, we can define the objective function as

$$\min\{x_1 + x_2 + \cdots + x_n\}$$

which corresponds to finding a truth assignment for the Boolean expression α such that the assignment has a minimum weight (i.e., the number of 1's in the assignment is minimized).

It is easy to see that for the given input instance α for Satisfiability Problem, the corresponding input instance $\pi(\alpha)$ for Integer Linear Programming Problem can be constructed in polynomial time. Moreover, if an optimal solution is found for $\pi(\alpha)$, then the Boolean expression α is certainly satisfiable thus the answer to α is YES. On the other hand, if no feasible solution can be constructed for $\pi(\alpha)$ then α has no truth assignment so the answer to α should be NO.

Therefore, Satisfiability Problem is polynomial time reducible to Integer Linear Programming Problem. Consequently, Integer Linear Programming Problem is NP-hard. $\square$

As we have described in the previous Lecture Notes, the general Linear Programming Problem can be solved in polynomial time. The above discussion shows that Integer Linear Programming Problem is much harder than the general Linear Programming Problem. Our latter study will show that Integer Linear Programming Problem is actually one of the hardest optimization problems.

CPSC-669 Computational Optimization

Lecture #13, September 27, 1995

Lecturer: Professor Jianer Chen
Scribe: Xiaotao Chen
Revision: Jianer Chen

13 NP-hard optimization problems

Recall the decision problem PARTITION that is defined as follows.

> PARTITION
>
> INPUT: A set $S = \{x_1, x_2, \ldots, x_n\}$ of n integers
>
> QUESTION: Is there a subset $S' \subseteq S$ such that
> $\sum_{i \in S'} x_i = \sum_{j \in S - S'} x_j$?

It is well-known that the problem PARTITION is NP-complete.

We now introduce several optimization problems that are at least as hard as PARTITION problem.

We start with an optimization version for the problem PARTITION that is given by SUBSETSUM $= (I, S, f, opt)$, where

- $I = \{\langle x_1, x_2, \ldots, x_n; B\rangle \mid x_i, B : \text{integers}\}$
- $S(\langle x_1, \ldots, x_n; B\rangle) = \{S' \subseteq \{x_1, \ldots, x_n\} \mid \sum_{x_i \in S'} x_i \leq B\}$
- $f(\langle x_1, \ldots, x_n; B\rangle, S') = \sum_{x_i \in S'} x_i$
- opt : max

Theorem 13.1 *The* SUBSETSUM *problem is NP-hard.*

PROOF. We show a polynomial time reduction f from the problem PARTITION to the problem SUBSETSUM.

Given an input instance $\alpha = \{x_1, x_2, \ldots, x_n\}$ for the problem PARTITION, $f(\alpha) = \langle x_1, x_2, \ldots, x_n; B\rangle$ is an instance for the problem SUBSETSUM, where $B = (\sum_{i=1}^{n} x_i)/2$. Now it is obvious that if an optimal solution to the instance $f(\alpha)$ of SUBSETSUM is a subset S' of $\{x_1, x_2, \ldots, x_n\}$ such that

$$\sum_{x_i \in S'} x_i = \frac{1}{2} \sum_{i=1}^{n} x_i$$

then $\alpha = \{x_1, \ldots, x_n\}$ is a YES-instance for PARTITION, otherwise α is a NO-instance for the problem. $\square$

Another popular optimization problem is KNAPSACK problem that is formally defined by KNAPSACK $= (I, S, f, opt)$, where

- $I = \{\langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle \mid s_i, v_j, B : \text{integers}\}$
- $S(\langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle) = \{S \subseteq \{1, \ldots, n\} \mid \sum_{i \in S} s_i \leq B\}$
- $f(\langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle, S) = \sum_{i \in S} v_i$
- opt : max

An "application" of KNAPSACK problem can be described as follows. A thief robbing a store finds n items. The ith item is worth v_i dollars and weighs s_i pounds. The thief wants to take as valuable a load as possible, but he can carry at most B pounds in his knapsack. Now the thief wants to decide what items he should take. Fortunately, the problem is NP-hard, as we prove in the following theorem.

Theorem 13.2 *The* KNAPSACK *problem is NP-hard.*

PROOF. We construct a polynomial time reduction f from the problem SUBSETSUM to the problem KNAPSACK.

Given an input instance $\alpha = \langle x_1, \ldots, x_n; B\rangle$ for the problem SUBSET-SUM, $f(\alpha) = \langle x_1, \ldots, x_n; x_1, \ldots, x_n; B\rangle$ is an input instance for the problem KNAPSACK. Clearly, an optimal solution to the instance $f(\alpha)$ is a subset S of $\{1, \ldots, n\}$ that satisfies the condition $\sum_{i \in S} x_i \leq B$ and maximizes the sum $\sum_{i \in S} x_i$. Thus, an optimal solution to the instance $f(\alpha)$ of KNAPSACK is also an optimal solution to the instance α of SUBSETSUM. $\square$

We say that a collection of c subsets $\langle S_1, \ldots, S_c\rangle$ of $\{1, \ldots, n\}$ is a *c-partition* of $\{1, \ldots, n\}$ if $S_1 \cup \cdots \cup S_c = \{1, \ldots, n\}$ and all subsets S_1, …, S_c are pairwisely disjoint.

We consider the optimization problem c-PROCESSOR SCHEDULING, which is formally defined by c-SCHEDULE $= (I, S, f, opt)$, where

- $I = \{\langle t_1, \ldots, t_n\rangle \mid t_i\text{'s are integers}\}$
- $S(\langle t_1, \ldots, t_n\rangle) = \{\langle S_1, \ldots, S_c\rangle \mid \langle S_1, \ldots, S_c\rangle \text{ a } c\text{-partition of } \{1, \ldots, n\}\}$

- $f(\langle t_1, \ldots, t_n \rangle, \langle S_1, \ldots, S_c \rangle) = \max_i \{\sum_{k \in S_i} t_k\}$
- *opt* : min

Intuitively, suppose we are given n jobs such that the ith job takes execution time t_i, and we want to distribute these jobs to c identical processors so that the parallel finish time (i.e., the time at which all processors finish their work) is minimized.

Theorem 13.3 *The c-*Processor Scheduling *problem is NP-hard, for $c \geq 2$.*

PROOF. We give a polynomial time reduction f from Partition problem to c-Processor Scheduling problem.

Let $\alpha = \langle x_1, \ldots, x_n \rangle$ be an input instance for Partition problem. Without loss of generality, we assume that $\sum_{i=1}^n x_i$ is an even number — otherwise α is clearly a NO-instance for Partition Problem. We define $f(\alpha)$ to be $\langle x_1, \ldots, x_n, B_3, \ldots, B_c \rangle$, where $B_r = (\sum_{i=1}^n t_i)/2$ for all $r = 3, \ldots, c$. Clearly, $f(\alpha)$ is an input instance for the c-Processor Scheduling problem. Now it is easy to verify that if an optimal solution to $f(\alpha)$ gives a parallel finish time $(\sum_{i=1}^n t_i)/2$, then $\langle x_1, \ldots, x_n \rangle$ is a YES-instance for Partition problem, otherwise, $\langle x_1, \ldots, x_n \rangle$ is a NO-instance for the problem. $\Box$

Thus, all these three optimization problems described above are NP-hard. By our believing that P $\neq$ NP, they cannot be solved in polynomial time. However, this does not obviate the need for solving these problems because of their obvious applications. One possible approach is that we could relax the requirement that we always find the optimal solution. In practice, a near-optimal solution will work fine in many cases. Of course, we expect that the algorithms for finding the near-optimal solutions are efficient.

Definition 13.1 An algorithm A is an *approximation algorithm* for an optimization problem $Q = (I_Q, S_Q, f_Q, opt_Q)$, if on any input instance $x \in I_Q$, the algorithm A produces a solution $y \in S_Q(x)$.

Note that here we have put no requirement on the approximation quality for an approximation algorithm. Thus, an algorithm that always produces a "trivial" solution (for example, it simply returns the first item for the Knapsack problem) is an approximation algorithm. To measure the quality of an approximation algorithm, we introduce the following concept.

Definition 13.2 An approximation algorithm A for an optimization problem $Q = (I_Q, S_Q, f_Q, opt_Q)$ has an *approximation ratio* $r(n)$, if on any input instance $x \in I_Q$, the solution y produced by the algorithm A satisfies

$$\frac{Opt(x)}{f(x,y)} \leq r(|x|) \quad \text{if } opt_Q = \max$$
$$\frac{f(x,y)}{Opt(x)} \leq r(|x|) \quad \text{if } opt_Q = \min$$

where $Opt(x)$ is defined to be $\max\{f(x,y) \mid y \in S_Q(x)\}$ if $opt_Q = \max$ and to be $\min\{f(x,y) \mid y \in S_Q(x)\}$ if $opt_Q = \min$.

Remark 13.3 By the definition, an approximation ratio is at least as large as 1. It is easy to se that the closer the approximation ratio to 1, the better the approximation quality of the approximation algorithm.

CPSC-669 Computational Optimization

Lecture #14, September 29, 1995

Lecturer: Professor Jianer Chen
Scribe: Xiaotao Chen
Revision: Jianer Chen

14 The Knapsack problem

We start with an approximation algorithm for the Knapsack problem. Recall that the Knapsack problem is defined as

> Knapsack
>
> Input: $\langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle$ where all s_i, v_j, B are integers
>
> Output: A subset S of $\{1, \ldots, n\}$, such that $\sum_{i\in S} s_i \leq B$
> and $\sum_{i\in S} v_i$ is maximized

We first present an algorithm that solves the Knapsack problem precisely. To simplify the description, for a subset S of $\{1, \ldots, n\}$, we will call $\sum_{i\in S} s_i$ the *size* of S and $\sum_{i\in S} v_i$ the *value* of S. Let $V = v_1 + v_2 + \cdots + v_n$. Thus, there is no subset of $\{1, \ldots, n\}$ that can have value larger than V. The algorithm goes as follows. For each index i and for each value $j \leq V$, we try to answer the question

> **Question** $K(i, j)$
>
> Is there a subset S of $\{1, \ldots, i\}$ such that the size of S is not larger than B and the value of S is equal to j?

The answer to Question $K(i, j)$ is "yes" *if and only if* at least one of the following two cases is true: (1) there is a subset S' of $\{1, \ldots, i-1\}$ such that the size of S' is not larger than B and the value of S is equal to j (in this case, simply let S be S'), and (2) there is a subset S'' of $\{1, \ldots, i-1\}$ such that the size of S'' is not larger than $B - s_i$ and the value of S'' is equal to $j - v_i$ (in this case, let $S = S'' \cup \{i\}$). Therefore, if we are able to answer Question $K(i-1, j)$ for all j, $0 \leq j \leq V$, we can answer Question $K(i, j)$ easily.

For small values i, the Question $K(i, j)$ seems easy. In particular, the answer to $K(0, j)$ is always "no" for $j > 0$ and the answer to $K(0, 0)$ is "yes".

The above discussion motivates the following dynamic programming algorithm for solving the KNAPSACK problem. We first compute $K(0, j)$ for all j, then, inductively, compute each $K(i, j)$ based on the answer to $K(i-1, j')$ for all j'. For each item $K(i, j)$, we associate it with a subset S in $\{1, \ldots, i\}$ such that the size of S is not larger than B and the value of S is equal to j.

Now a potential problem arises. How do we handle two different witnesses for a "yes" answer to the Question $K(i, j)$? More specifically, suppose that we find two subsets S_1 and S_2 of $\{1, \ldots, i\}$ such that both of S_1 and S_2 have size bounded by B and value equal to j, should we keep both of them with $K(i, j)$, or ignore one of them? Keeping both can make $K(i, j)$ exponentially grow as i increases, which will significantly slow down our algorithm. Thus, we intend to ignore one of S_1 and S_2. Which one do we want to ignore? Intuitively, the one with larger size should be ignored (recall that S_1 and S_2 have the same value). However, would ignoring the set cause a final loss of the optimal solution? Fortunately, the following theorem ensures that optimal solutions cannot get lost when we ignore the set with larger size.

Theorem 14.1 *Let S_1 and S_2 be two subsets of $\{1, \ldots, i\}$ such that S_1 and S_2 have the same value, and the size of S_1 is at least as large as the size of S_2. If S_1 leads to an optimal solution $S = S_1 \cup S_3$ for the* KNAPSACK *problem, where $S_3 \subseteq \{i+1, \ldots, n\}$, then $S' = S_2 \cup S_3$ is also an optimal solution for the* KNAPSACK *problem.*

PROOF. Let $size(S)$ and $value(S)$ denote the size and value of a subset S of $\{1, \ldots, n\}$, respectively. We have

$$size(S') = size(S_2) + size(S_3) \quad \text{and} \quad size(S) = size(S_1) + size(S_3)$$

By the assumption that $size(S_1) \geq size(S_2)$, we have $size(S) \geq size(S')$. Since S is an optimal solution, we have $size(S') \leq B$. Thus S' is also a solution to the KNAPSACK problem. Moreover,

$$value(S') = value(S_2) + value(S_3) = value(S_1) + value(S_3) = value(S)$$

Thus, S' is also an optimal solution. $\square$

By Theorem 14.1, for two subsets S_1 and S_2 of $\{1, \ldots, i\}$ that both witness the "yes" answer to Question $K(i, j)$, if the one of larger size leads to an optimal solution, then the one with smaller size also leads to an optimal solution. Therefore, ignoring the set of larger size will not lead to loss of all optimal solutions. More specifically, if we can derive an optimal solution based on the set of larger size, then we can also derive an optimal solution based on the set of smaller size using exactly the same procedure.

Now we are ready for the algorithm.

Algorithm 14.1 `Knapsack-Dyn`

```
Input:  $s_1, \ldots, s_n; v_1, \ldots, v_n; B$, all integers
Output:  A subset $S \subseteq \{1, ..., n\}$, such that $\sum_{i \in S} s_i \le B$
         and $\sum_{i \in S} v_i$ is maximized

1.  for $i = 0$ to $n$ do
      for $j = 0$ to $V$ do
        $K[i,j] = *$;
2.  $K[0,0] = \phi$;          {$\phi$ is the empty set}
3.  for $i = 0$ to $n-1$ do
      for $j = 0$ to $V$ do
        if $K[i,j] \neq *$ then
          Put($K[i,j], K[i+1,j]$);
          if $size(K[i,j]) + s_{i+1} \le B$ then
            $v = j + v_{i+1}$;       {$j$ is the value of $K[i,j]$}
            Put($K[i,j] \cup \{i+1\}, K[i+1,v]$);
4.  $j = V$;
    while $K[n,j] = *$ do
      $j = j - 1$;
5.  return $K[n,j]$.
```

Step 4 of the algorithm `Knapsack-Dyn` searches the last row from the last column to find the first $K[n,j]$ that is not $*$. Obviously, the value j is the largest value a subset S of $\{1, \ldots, n\}$ can make under the restriction that S has size bounded by B.

The subroutine `Put`$(S_0, K[i,j])$ is used to solve the multiple witness problem, where S_0 is a subset of $\{1, \ldots, i\}$ such that S_0 has value j. Details of this subroutine is given as follows.

Algorithm 14.2 `Put`$(S_0, K[i,j])$

```
1.  if $K[i,j] = *$ then
        $K[i,j] = S_0$;
      else if $size(S_0) < size(K[i,j])$
        then $K[i,j] = S_0$.
```

According to our discussion, it should be clear that the Algorithm 14.1 solves the KNAPSACK problem.

Theorem 14.2 *The algorithm* `Knapsack-Dyn` *runs in time* $O(nV)$.

PROOF. We show data structures on which the **if** statement in Step 3 can be executed in constant time. The theorem follows directly from this discussion.

For each item $K[i, j]$, which is for a subset S_{ij} of $\{1, \ldots, i\}$, we associate three parameters: (1) the size of S_{ij}, (2) a marker m_{ij} indicating whether i is contained in S_{ij}, and (3) a pointer p_{ij} to an item $K[i-1, j']$ in the previous row such that the set S_{ij} is derived from the set $K[i-1, j']$. Note that the actual set S_{ij} is *not* stored in $K[i, j]$.

With these parameters, the size of the set S_{ij} can be directly read from $K[i, j]$ in constant time. Moreover, it is also easy to verify that the subroutine calls `Put`$(K[i, j], K[i+1, j])$ and `Put`$(K[i, j] \cup \{i+1\}, K[i+1, v])$ can also be performed in constant time by updating the parameters in $K[i+1, j]$ and $K[i+1, v]$.

This shows that steps 1-4 of the algorithm `Knapsack-Dyn` take time $O(nV)$.

We must show how the actual optimal solution $K[n, j]$ is returned in step 5. After we have decided the item $K[n, j]$ in step 5, which corresponds to an optimal solution S_{nj} that is a subset of $\{1, \ldots, n\}$, we first check the marker m_{nj} to see if S_{nj} contains n, then follow the point p_{nj} to an item $K[n-1, j']$, where we can check whether the set S_{nj} contains $n-1$ and a pointer to an item in the $(n-2)$nd row, and so on. In time $O(n)$, we will be able to "collect" all elements in S_{nj} and return the actual set S_{nj}. □

It seems that we have developed a polynomial time algorithm that solves the NP-hard optimization problem KNAPSACK. This is, in fact, not true since the value V can be much larger than any polynomial of n.

Remark 14.1 We point out that the problem SUBSETSUM can be solved by an algorithm very similar to `Knapsack-Dyn`, with running time $O(nB)$ on input instance $\{x_1, \ldots, x_n; B\}$. We leave the detailed implementation for this algorithm as an exercise to the reader.

CPSC-669 Computational Optimization

Lecture #15, October 2, 1995

Lecturer: Professor Jianer Chen
Scribe: Balarama Varanasi
Revision: Jianer Chen

15 Approximating Knapsack

We re-visit Knapsack problem and attempt to develop an approximation algorithm that provides a solution of acceptable quality. Recall that the Knapsack problem is defined as

> Knapsack
>
> Input: $\langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle$ where all s_i, v_j, B are integers
>
> Output: A subset S of $\{1, \ldots, n\}$, such that $\sum_{i \in S} s_i \leq B$ and $\sum_{i \in S} v_i$ is maximized

In the last lecture, we presented an algorithm `Knapsack-Dyn` that, on an input instance $X = \langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle$ of the Knapsack problem, constructs an optimal solution for X in time $O(nV)$, where $V = \sum_{i=1}^n v_i$. If V is not bounded by any polynomial function of n, then the running time of the algorithm is not polynomial. Is there a way to lower the value of V? Well, an obvious way is to divide each value v_i by a sufficiently large number K so that V is replaced by a smaller value $V' = V/K$. In order to let the algorithm `Knapsack-Dyn` to run in polynomial time, we must have $V' \leq cn^d$ for some constants c and d, or equivalently, $K \geq V/(cn^d)$. Another problem is that the value v_i/K may not be an integer while by our definition, all input values in an instance of Knapsack problem are integers. Thus, we will take $v_i' = \lfloor v_i/K \rfloor$. This gives a new instance X' for Knapsack problem

$$X' = \langle s_1, \ldots, s_n; v_1', \ldots, v_n'; B\rangle$$

where $v_i' = \lfloor v_i/K \rfloor$, for $i = 1, \ldots, n$. For $K \geq V/(cn^d)$ for some constants c and d, the algorithm `Knapsack-Dyn` finds an optimal solution for X' in polynomial time. Note that a solution to X' is also a solution to X and we intend to "approximate" the optimal solution to X by an optimal solution to X'. Since the application of the floor function $\lfloor \cdot \rfloor$, we lose precision thus an optimal solution for X' may not be an optimal solution for X. How much precision have we lost? Intuitively, the larger the value K, the more precision we would lose. Thus, we want K to be as small as possible. On the other hand, we want K to be as large as possible so that the running time of the algorithm `Knapsack-Dyn` can be bounded by a polynomial. Now a natural question is whether there is a value K that makes the algorithm `Knapsack-Dyn` run in polynomial time and cause not much precision loss so that the optimal solution to the instance X' is "close" to the optimal solution to the instance X. For this, we need the following formal analysis.

Let $S \subseteq \{1, \ldots, n\}$ be an optimal solution to the instance X, and let $S' \subseteq \{1, \ldots, n\}$ be the optimal solution to the instance X' produced by the algorithm `Knapsack-Dyn`. Note that S is also a solution to the instance X' and that S' is also a solution to the instance X. Let $Opt(X) = \sum_{i \in S} v_i$ and $Apx(X) = \sum_{j \in S'} v_i$ be the objective function values of the solutions S and S', respectively. Therefore, $Opt(X)/Apx(X)$ is the approximation ratio for the algorithm we proposed. In order to bound the approximation ratio by a given constant ϵ, we consider

$$\begin{aligned} Opt(X) &= \sum_{i \in S} v_i \\ &= K \sum_{i \in S} \frac{v_i}{K} \\ &\leq K \sum_{i \in S} (\lfloor \frac{v_i}{K} \rfloor + 1) \\ &\leq Kn + K \sum_{i \in S} \lfloor \frac{v_i}{K} \rfloor \\ &= Kn + K \sum_{i \in S} v'_i \end{aligned}$$

The last inequality is because the cardinality of the set S is bounded by n.

Now since S' is an optimal solution to $X' = \langle s_1, \ldots, s_n; v'_1, \ldots, v'_n; B \rangle$, we must have

$$\sum_{i \in S} v'_i \leq \sum_{i \in S'} v'_i$$

Thus,

$$\begin{aligned} Opt(X) &\leq Kn + K \sum_{i \in S'} v'_i \\ &= Kn + K \sum_{i \in S'} \lfloor \frac{v_i}{K} \rfloor \end{aligned}$$

$$\begin{aligned} &\leq Kn + K \sum_{i \in S'} \frac{v_i}{K} \\ &= Kn + Apx(X) \qquad (4) \end{aligned}$$

This gives us the approximation ratio.

$$\frac{Opt(X)}{Apx(X)} \leq 1 + \frac{Kn}{Apx(X)}$$

Without loss of generality, we can assume that $s_i \leq B$ for all $i = 1, \ldots, n$ (otherwise, the index i can be simply deleted from the input instance since it can never make contribution to a feasible solution to X). Thus, $Opt(X)$ is at least as large as $\max_{1 \leq i \leq n}\{v_i\} \geq V/n$. From inequality (4), we have

$$Apx(X) \geq Opt(X) - Kn \geq \frac{V}{n} - Kn$$

It follows that

$$\begin{aligned} \frac{Opt(X)}{Apx(X)} &\leq 1 + \frac{Kn}{\frac{V}{n} - Kn} \\ &= 1 + \frac{Kn^2}{V - Kn^2} \end{aligned}$$

Thus, in order to bound the approximation ratio by $1 + \epsilon$, it should be such that

$$\frac{Kn^2}{V - Kn^2} \leq \epsilon$$

This leads to $K \leq (\epsilon V)/(n^2(1 + \epsilon))$.

Recall that to make the algorithm `Knapsack-Dyn` run in polynomial time on the input instance X', we must have $K \geq V/(cn^d)$ for some constants c and d. Combining these two relations, we get $c = 1 + 1/\epsilon$, and $d = 2$, and the value

$$K = V/(cn^d) = \frac{V}{(1 + 1/\epsilon)n^2}$$

makes the algorithm `Knapsack-Dyn` run in time $O(n^3(1+1/\epsilon))$ and produces a solution S' to the instance X with approximation ratio bounded by ϵ.

We summarize the above discussion in the following algorithm and theorems.

Algorithm 15.1 `Knapsack-Approx`

```
Input:  $\langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle$, and a constant $\epsilon$.
Output:  A subset $S' \subseteq \{1, ..., n\}$, such that $\sum_{i \in S'} s_i \leq B$

1.  Let $K = \frac{V}{(1+1/\epsilon)n^2}$;
2.  for $i = 1$ to $n$ do     $v_i' = \lfloor v_i/K \rfloor$;
3.  Apply algorithm Knapsack-Dyn on $\langle s_1, \ldots, s_n; v_1', \ldots, v_n'; B\rangle$
        and find a subset $S' \subseteq \{1, ..., n\}$;
4.  Output $S'$;
```

Theorem 15.1 *For any input instance of the* Knapsack *problem, the algorithm* `Knapsack-Approx` *runs in time $O(n^3(1+1/\epsilon))$ and produces a solution with approximation ratio bounded by $1 + \epsilon$.*

Theorem 15.2 *For any fixed constant ϵ, there is an algorithm of running time $O(n^3)$ that, on an input instance of the* Knapsack *problem, produces a solution with approximation ratio bounded by $1 + \epsilon$.*

This lecture concludes with the above result.

CPSC-669 Computational Optimization

Lecture #16, October 4, 1995

Lecturer: Professor Jianer Chen
Scribe: Balarama Varanasi
Revision: Jianer Chen

16 Approximating Processor Scheduling

We continue the discussion of approximation algorithms.

We presented an $O(n^3/\epsilon)$ time approximation algorithm with approximation ratio ϵ for any $\epsilon > 0$ for the KNAPSACK problem. Note that the time complexity of this algorithm is polynomial in both the input size n and the value $1/\epsilon$, which seems the best we can expect for an approximation algorithm for an NP-hard optimization problem. This motivates the following definition.

Definition 16.1 An optimization problem Q has a *fully polynomial time approximation scheme* (FPTAS) if it has an approximation algorithm A such that given $\langle x, \epsilon \rangle$, where x is an input instance of Q and ϵ is a positive constant, A finds a solution for x with approximation ratio bounded by $1+\epsilon$ in time polynomial in both n and $1/\epsilon$.

By the definition, the KNAPSACK problem has a fully polynomial time approximation scheme. In this lecture, we illustrate the techniques for developing fully polynomial time approximation schemes for optimization problems by studying another important optimization problem, the c-PROCESSOR SCHEDULING problem.

The approach for developing a fully polynomial time approximation scheme for the c-PROCESSOR SCHEDULING problem is very similar to that for the KNAPSACK problem: we first develop a precise algorithm for the problem such that the algorithm runs in time polynomial in both n and T, where T is a large number obtained from the input. Then we try to scale T by dividing all numbers in the input by a large number K. By properly choosing the value K, we can make the precise algorithm to run in polynomial time and keep the approximation ratio bounded by a given constant ϵ. Because of the similarity, some details in the algorithms and in the analysis are omitted. The reader is advised to refer to corresponding parts in the study of the KNAPSACK problem and complete the omitted parts for a better understanding.

Recall that the optimization problem c-PROCESSOR SCHEDULING is defined by c-SCHEDULE $= (I, S, f, opt)$, where

- $I = \{\langle t_1, \ldots, t_n \rangle \mid t_i$'s are integers$\}$
- $S(\langle t_1, ..., t_n \rangle) = \{\langle S_1, ..., S_c \rangle | \langle S_1, ..., S_c \rangle$ is a c-partition of $\{1, ..., n\}\}$
- $f(\langle t_1, \ldots, t_n \rangle, \langle S_1, \ldots, S_c \rangle) = \max_{1 \le i \le c}\{\sum_{k \in S_i} t_k\}$
- opt : min

Let $T = \sum_{i=1}^{n} t_i$. Note that every scheduling $(S_1, \ldots, S_c)$ of the n jobs $\langle t_1, \ldots, t_n \rangle$, where S_d is the subset of $\{1, \ldots, n\}$ that corresponds to the jobs assigned to the dth processor, can be written as a c-tuple $(T_1, \ldots, T_c)$ with $0 \le T_d \le T$ for all $1 \le d \le c$, where $T_d = \sum_{h \in S_d} t_h$ is the total execution time assigned to the dth processor. The c-tuple $(T_1, \ldots, T_c)$ will be called the *time list* for the scheduling $(S_1, \ldots, S_c)$. Moreover, each c-tuple $(T_1, \ldots, T_c)$ with $0 \le T_d \le T$ for all $d = 1, \ldots, c$ can be uniquely written as a non-negative integer j less than or equal to $(T+1)^c$ by the following formula

$$j = T_1(T+1)^{c-1} + T_2(T+1)^{c-2} + \cdots + T_{c-1}(T+1) + T_c \tag{5}$$

Conversely, each non-negative integer j less than or equal to $(T+1)^c$ can be uniquely decomposed into a c-tuple $(T_1, \ldots, T_c)$ with $0 \le T_d \le T$ for all $d = 1, \ldots, c$, using the formula (5).

Now as for the KNAPSACK problem, for each i, $0 \le i \le n$, and for each non-negative integer j, where we suppose that the integer j is decomposed into a c-tuple $(T_1, \ldots, T_c)$ by the formula (5), we ask the question

> Is there a scheduling of the first i jobs $\{t_1, \ldots, t_i\}$ that gives the time list $j = (T_1, \ldots, T_c)$?

Note that for two different schedulings of the i jobs $\{t_1, \ldots, t_i\}$ that have the same time list $(T_1, \ldots, T_c)$, we can pick either of them without loss of correctness.

Now we are ready to present the algorithm.

Algorithm 16.1 `c-Scheduling-Dyn`

```
Input: $n$ jobs with execution time $t_1, \ldots, t_n$, all integers
```

```
Output:  A scheduling of the $n$ jobs on $c$ processors such
         that the parallel finish time is minimized

{ $H[0..n, 0..(T+1)^c]$ is a table such that the element $H[i,j]$
  is a scheduling $(S_1, \ldots S_c)$ of the first $i$ jobs $t_1$, ..., $t_i$
  whose time list is $j = (T_1, \ldots, T_c)$.  }

1.  $T = \sum_{i=1}^{n} t_i$;
2.  for $i = 0$ to $n$ do
      for $j = 0$ to $(T+1)^c$ do
        $H[i,j] = *$;
3.  $H[0,0] = (\phi, \ldots, \phi)$;          {$\phi$ is the empty set}
4.  for $i = 0$ to $n-1$ do
      for $j = 0$ to $(T+1)^c$ do
        if $H[i,j] \neq *$ then
          Let $H[i,j] = (S_1, \ldots, S_c)$ is a scheduling of $t_1$, ...,
          $t_i$, and $j = (T_1, \ldots, T_c)$ is the time list for $H[i,j]$;
          for $d = 1$ to $c$ do
            $H[i+1, j_d] = (S_1, \ldots, S_{d-1}, S_d \cup \{i+1\}, S_{d+1}, \ldots, S_c)$;
            where $j_d = (T_1, \ldots, T_{d-1}, T_d + t_{i+1}, T_{d+1}, \ldots, T_c)$;
5.  Scan the $n$th row of the table $H$ to find the scheduling
    $H[n,j] \neq *$ such that $j = (T_1, \ldots, T_c)$ has the minimum
    parallel time;
6.  Return $H[n,j]$.
```

We analyze the algorithm. As we did for the algorithm `Knapsack-Dyn`, instead of storing the entire c-tuple $(S_1, \ldots, S_c)$ in $H[i,j]$, we simply keep a marker that indicates which processor is assigned the ith job and a pointer to the element $H[i-1, j']$ such that the scheduling $H[i,j]$ of t_1, ..., t_i is obtained from the scheduling $H[i-1, j']$ of t_1, ..., t_{i-1} by assigning the job t_i to a proper processor. By these data structures, each assignment to the elements of the table H can be done in constant time. Moreover, for each non-negative integer j less than or equal to $(T+1)^c$, by formula (5), j can be uniquely decomposed using a constant number of division and modulo operations into a c-tuple $(T_1, \ldots, T_c)$ with $0 \leq T_d \leq T$ for all $d = 1, \ldots, c$ (recall that c is a constant). Similarly, each c-tuple $(T_1, \ldots, T_c)$ with $0 \leq T_d \leq T$ for all $d = 1, \ldots, c$ can be converted in constant time into a unique non-negative integer j less than or equal to $(T+1)^c$. In conclusion, each execution of the **if** statement in the loop in step 4 takes constant time. Consequently, the algorithm c-`Scheduling-Dyn` takes time $O(nT^c)$.

A fully polynomial time approximation scheme now is derived for the c-PROCESSOR SCHEDULING problem based on algorithm c-`Scheduling-Dyn`. The idea is the same as for the KNAPSACK problem: we first scale the input numbers to make T smaller then apply algorithm c-`Scheduling-Dyn` to the scaled input.

Algorithm 16.2 c-`Scheduling-Apx`

```
Input:  $\langle t_1, \ldots, t_n; \epsilon \rangle$, all $t_i$'s are integers
Output:  A scheduling of the $n$ jobs on $c$ processors

1.  Let $K = \epsilon \sum_{i=1}^{n} t_i/(cn)$;
2.  for $i = 1$ to $n$ do    $t'_i = \lceil t_i/K \rceil$;
3.  Apply algorithm $c$-Scheduling-Dyn on input $\langle t'_1, \ldots, t'_n \rangle$ to
      produce a scheduling $(S'_1, \ldots, S'_c)$ on $\langle t_1, \ldots, t_n \rangle$;
4.  Output $(S'_1, \ldots, S'_c)$;
```

Theorem 16.1 *The algorithm* c-`Scheduling-Apx` *on input* $\langle t_1, \ldots, t_n; \epsilon \rangle$ *produces a scheduling* $(S'_1, \ldots, S'_c)$ *with approximation ratio bounded by* $1 + \epsilon$ *and runs in time* $O(n^{c+1}/\epsilon^c)$.

PROOF. It is easy to see that the time complexity of the algorithm c-`Scheduling-Apx` is dominated by step 3.

Since $T_0 = \sum_{i=1}^{n} t'_i = O(\sum_{i=1}^{n} t_i/K) = O(n/\epsilon)$, by our analysis, the algorithm c-`Scheduling-Dyn` in step 3, thus the algorithm c-`Scheduling-Apx`, runs in time $O(nT_0^c) = O(n^{c+1}/\epsilon^c)$.

Now let $(S_1, \ldots, S_c)$ be an optimal solution to the input instance $X = \langle t_1, \ldots, t_n \rangle$ of the c-PROCESSOR SCHEDULING problem, and let $(S'_1, \ldots, S'_c)$ be the optimal solution to the input instance $X' = \langle t'_1, \ldots, t'_n \rangle$ obtained by the algorithm c-`Scheduling-Dyn`. Note that $(S_1, \ldots, S_c)$ is also a solution to the instance $\langle t'_1, \ldots, t'_n \rangle$ and $(S'_1, \ldots, S'_c)$ is also a solution to the instance $\langle t_1, \ldots, t_n \rangle$.

For all d, $1 \le d \le c$, let

$$T_d = \sum_{h \in S_d} t_h \qquad V_d = \sum_{h \in S_d} t'_h$$

$$T'_d = \sum_{h \in S'_d} t_h \qquad V'_d = \sum_{h \in S'_d} t'_h$$

Without loss of generality, suppose

$$T_1 = \max_{1 \le d \le c} \{T_d\} \qquad V_2 = \max_{1 \le d \le c} \{V_d\}$$

$$T_3' = \max_{1\le d\le c}\{T_d'\} \qquad V_4' = \max_{1\le d\le c}\{V_d'\}$$

Therefore, on instance $\langle t_1, \ldots, t_n\rangle$, the scheduling $(S_1, \ldots, S_c)$ has parallel finish time T_1 and the scheduling $(S_1', \ldots, S_c')$ has parallel finish time T_3'; and on instance $\langle t_1', \ldots, t_n'\rangle$, the scheduling $(S_1, \ldots, S_c)$ has parallel finish time V_2 and the scheduling $(S_1', \ldots, S_c')$ has parallel finish time V_4'. The approximation ratio given by the algorithm c-`Processor-Apx` is T_3'/T_1.

We have

$$T_3' = \sum_{h\in S_3'} t_h = K\sum_{h\in S_3'}(t_h/K) \le K\sum_{h\in S_3'} t_h' = KV_3' \le KV_4'$$

The last inequality is by the assumption $V_4' = \max_{1\le d\le c}\{V_d'\}$.

Now since $(S_1', \ldots, S_c')$ is an optimal scheduling on instance $\langle t_1', \ldots, t_n'\rangle$, we have $V_4' \le V_2$. Thus,

$$T_3' \le KV_2 = K\sum_{h\in S_2} t_h' = K\sum_{h\in S_2}\lceil t_h/K\rceil$$
$$\le K\sum_{h\in S_2}(\frac{t_h}{K}+1) \le T_2 + Kn \le T_1 + Kn$$

The last inequality is by the assumption $T_1 = \max_{1\le d\le c}\{T_d\}$.

This gives us immediately

$$T_3'/T_1 \le 1 + Kn/T_1$$

It is easy to see that $T_1 \ge \sum_{i=1}^n t_i/c$, and recall that $K = \epsilon\sum_{i=1}^n t_i/(cn)$, we obtain $Kn/T_1 \le \epsilon$. That is, the scheduling $(S_1', \ldots, S_c')$ produced by the algorithm c-`Scheduling-Apx` has approximation ratio bounded by $1+\epsilon$. $\square$

Corollary 16.2 *For a fixed constant c, the c-*PROCESSOR SCHEDULING *problem has a fully polynomial time approximation scheme.*

CPSC-669 Computational Optimization

Lecture #17, October 6, 1995

Lecture: Professor Jianer Chen
Scribe: Jennifer Walter
Revision: Jianer Chen

17 Which optimization problem has a FPTAS?

Let us first review the definition of a fully polynomial-time approximation scheme.

Definition 17.1 An optimization problem Q has a *fully polynomial time approximation scheme* (FPTAS) if it has an approximation algorithm A that on input $\langle x, \epsilon \rangle$, where x is an input instance of Q and ϵ is a positive number, gives a solution of approximation ratio bounded by $1 + \epsilon$ in time polynomial in $|x|$ and $1/\epsilon$.

This definition says that if a fully polynomial time approximation scheme exists for an optimization problem Q, then there is a polynomial time approximation algorithm for the problem that can approximate the optimal solution for the problem Q to any arbitrary precision. A fully polynomial time approximation scheme seems the best solution we can hope to derive for an NP-hard optimization problem. By our discussion in the previous lectures, the optimization problems Subset Sum, Knapsack, and c-Processor Scheduling have fully polynomial time approximation schemes.

Natural questions are how many problems we can devise a fully polynomial time approximation scheme for, what are the kinds of possible fully polynomial time approximation scheme solutions, and how it can be determined that a problem does not have a fully polynomial time approximation scheme. To discuss these questions, we first introduce a notation.

Definition 17.2 Let $Q = \langle I_Q, S_Q, f_Q, opt_Q \rangle$ be an optimization problem. For each input instance $x \in I_Q$, define $Opt_Q(x) = opt_Q\{f_Q(x,y) | y \in S_Q(x)\}$. That is, $Opt_Q(x)$ is the value of the objective function f_Q on input instance x and an optimal solution to x.

We have the following very useful theorem.

Theorem 17.1 *Let $Q = \langle I_Q, S_Q, f_Q, opt_Q\rangle$ be an optimization problem. If there is a fixed polynomial p such that for all input instances $x \in I_Q$, $Opt_Q(x)$ is bounded by $p(|x|)$, then Q does not have a fully polynomial time approximation scheme unless Q can be precisely solved in polynomial-time.*

PROOF. Let A be an approximation algorithm that is a fully polynomial time approximation scheme for the optimization problem Q. We show that Q can be precisely solved in polynomial time.

By the definition, we can suppose that the running time of A is n^c/ϵ^d, where c and d are fixed constants. Moreover, by the condition given in the theorem, we can assume that $Opt_Q(x) \leq n^h$, where h is also a fixed constant.

First assume that $opt_Q = \min$. For an input instance $x \in I_Q$, let $A(x)$ be the objective function value on the input x and the solution to x produced by the algorithm A. Thus, we know that for any $\epsilon > 0$, the algorithm A produces in time n^c/ϵ^d a solution with approximation ratio $A(x)/Opt(x) \leq 1 + \epsilon$. Also note that $A(x)/Opt(x) \geq 1$.

Now, let $\epsilon = 1/n^{h+1}$, then the algorithm A produces a solution with approximation ratio bounded by

$$1 \leq \frac{A(x)}{Opt(x)} \leq 1 + \frac{1}{n^{h+1}}$$

which gives

$$Opt(x) \leq A(x) \leq Opt(x) + Opt(x)/n^{h+1}$$

Since both $Opt(x)$ and $A(x)$ are integers, and $Opt(x) \leq n^h$ implies that $Opt(x)/n^{h+1}$ is a number strictly less than 1, we conclude that

$$Opt(x) = A(x)$$

That is, the algorithm A actually produces an optimal solution to the input instance x. Moreover, the running time of A is bounded by $n^c/(1/n^{h+1})^d = n^{c+hd+d}$, which is a polynomial of n.

The case that $opt_Q = \max$ can be proved similarly. Note that in this case, we should also have $A(x) \leq n^h$. Thus, in time $n^c/(1/n^{h+1})^d = n^{c+hd+d}$, the algorithm A produces a solution to x with the value $A(x)$ such that

$$1 \leq Opt(x)/A(x) \leq 1 + 1/n^{h+1}$$

which gives

$$A(x) \leq Opt(x) \leq A(x) + A(x)/n^{h+1}$$

Now since $A(x)/n^{h+1} < 1$, we conclude $Opt(x) = A(x)$. $\square$

In particular, this theorem says that if $Opt_Q(x)$ is bounded by a polynomial of the input length $|x|$ and Q is known to be NP-hard, then Q does not have a fully polynomial time approximation scheme unless P = NP.

Theorem 17.1 is actually very powerful. Most NP-hard optimization problems satisfy the condition stated in the theorem, thus we can derive directly that these problems have no fully polynomial time approximation scheme. We will give a few examples below to illustrate the power of Theorem 17.1.

Consider the following problem:

> INDEPENDENT SET IS = $\langle I, S, f, opt \rangle$
>
> I: set of all graphs $G = (V, E)$
>
> $S(G)$: the collection of subsets S of vertices of G such that no two vertices in S are adjacent
>
> $f(G, S)$: the number of vertices in S
>
> *opt*: max

INDEPENDENT SET problem has many applications in networking design and scheduling. A trivial solution to the INDEPENDENT SET problem is to pick one single vertex, or a small number of vertices from the graph which are not adjacent. The problem is more difficult for a very large set of vertices. In fact, this is well-known that the INDEPENDENT SET problem is NP-hard.

It is easy to apply Theorem 17.1 to show that the INDEPENDENT SET problem has no fully polynomial time approximation scheme. In fact, the value of the objective function is bounded by the number of vertices in the input graph G, which is certainly bounded by a polynomial of the input length $|G|$.

There are many other graph problems (actually, most graph problems) like the INDEPENDENT SET problem that ask to optimize a subset of vertices or edges of the input graph. For all these problems, we can conclude directly from Theorem 17.1 that they do not have a fully polynomial time approximation scheme unless they can be solved precisely in polynomial time.

Let us consider another example of a problem for which no fully polynomial time approximation scheme exists.

> BOUNDED-TIME PROCESSOR SCHEDULING

INPUT: $\{t_1, t_2, \ldots, t_n; B\}$, all integers where each t_i is the execution time for the ith job and B is a restriction on the parallel finish time

OUTPUT: A scheduling of the n jobs on m processors such that the parallel finish time is bounded by B and m is minimized

The BOUNDED-TIME PROCESSOR SCHEDULING problem is commonly called BIN PACKING problem, which is known to be NP-hard. Given an input instance for the BOUNDED-TIME PROCESSOR SCHEDULING problem, either we can conclude immediately that there is no such scheduling (if any input job has execution time larger than B), or we know the output value m is bounded by n (i.e., in the worst case, each processor is assigned with a single job). In any case, we have $Opt(x)$ bounded by n. By Theorem 17.1, we conclude directly that the BOUNDED-TIME PROCESSOR SCHEDULING problem has no fully polynomial time approximation scheme unless P = NP.

Remark 17.3 Although this version of the scheduling problem has no fully polynomial time approximation scheme, the majority of the problems for which a fully polynomial time approximation scheme exists are scheduling problems.

What if the condition of Theorem 17.1 does not hold? Can we still derive a conclusion of nonexistence of a fully polynomial time approximation scheme for an optimization problem? We study this problem starting with the famous TRAVELING SALESMAN problem (TSP), and will derive general rules for this kind of optimization problems.

TRAVELING SALESMAN (TSP)

INPUT: a weighted complete graph G

OUTPUT: a simple cycle through all vertices of G (such a simple cycle is called a *traveling salesman tour*) and the weight of the cycle is minimized

The TRAVELING SALESMAN problem obviously does not satisfy the condition stated in Theorem 17.1. For example, if all edges of the input graph G of n vertices have weight of order $\Theta(2^n)$, then the weight of the minimum traveling salesman tour is $\Omega(n2^n)$ while a binary representation of the input graph G has length bounded by $O(n^3)$ (note that the length of the binary representation of a number of order $\Theta(2^n)$ is $O(n)$ and G has $O(n^2)$ edges). Therefore, Theorem 17.1 does not apply to the TRAVELING SALESMAN problem.

To show the non-approximability of the TRAVELING SALESMAN problem, we first consider a simpler version of the TRAVELING SALESMAN problem, which is defined as follows.

> TRAVELING SALESMAN 1-2 (TSP(1,2))
>
> INPUT: a weighted complete graph G such that the weight of each edge of G is either 1 or 2
>
> OUTPUT: a traveling salesman tour of minimum weight

Theorem 17.2 *The* TRAVELING SALESMAN 1-2 *problem is NP-hard.*

PROOF. We present a polynomial time reduction that transforms the well-known NP-complete problem HAMILTONIAN CIRCUIT to the TRAVELING SALESMAN 1-2 problem.

By the definition, for each undirected unweighted graph G of n vertices, the HAMILTONIAN CIRCUIT problem asks if G contains a Hamiltonian circuit, i.e., a simple cycle of length n.

Given an input instance $G = (V, E)$ for the HAMILTONIAN CIRCUIT problem, we add edges to G to make a weighted complete graph $G' = (V, E \cup E')$ such that for each edge $e \in E$ of G' that is in the original graph G, we assign a weight 1 and for each edge $e' \in E'$ of G' that is not in the original graph G, we assign a weight 2. The graph G' is certainly an input instance of the TRAVELING SALESMAN 1-2 problem. Now, let T be a minimum weighted traveling salesman tour in G'. It is easy to verify that the weight of T is equal to n if and only if the original graph G contains a Hamiltonian circuit.

This completes the proof. $\Box$

Theorem 17.1 can apply to the TRAVELING SALESMAN 1-2 problem directly.

Theorem 17.3 *The* TRAVELING SALESMAN 1-2 *problem has no fully polynomial time approximation scheme unless $P = NP$.*

PROOF. Since the weight of a traveling salesman tour for an input instance G of the TRAVELING SALESMAN 1-2 problem is at most $2n$, assuming that

G has n vertices, the condition stated in Theorem 17.1 is satisfied by the Traveling Salesman 1-2 problem. Now the theorem follows from Theorem 17.1 and Theorem 17.2. $\square$

Now we are ready for a conclusion on the approximability of the Traveling Salesman problem in its general form.

Theorem 17.4 *The* Traveling Salesman *problem has no fully polynomial time approximation scheme unless $P = NP$.*

proof. Since each input instance for the Traveling Salesman 1-2 problem is also an input instance for the Traveling Salesman problem, a fully polynomial time approximation scheme for the Traveling Salesman problem should also be a fully polynomial time approximation scheme for the Traveling Salesman 1-2 problem. Now the theorem follows from Theorem 17.3. $\square$

CPSC-669 Computational Optimization

Lecture #18, October 9, 1995

Lecture: Professor Jianer Chen
Scribe: Jennifer Walter
Revision: Jianer Chen

18 Strong NP-hardness

We continue the discussion on what conditions will make an optimization problem have a fully polynomial time approximation scheme. In the last lecture, we have seen that if the optimal value $Opt_Q(x)$ is always bounded by a polynomial of the input length of x, then the problem Q has no fully polynomial time approximation scheme unless Q can be solved precisely in polynomial time. We have also studied the TRAVELING SALESMAN problem, which does not satisfy the above condition, and developed a technique to show that the TRAVELING SALESMAN problem has no fully polynomial time approximation scheme. We started by a restricted version of the TRAVELING SALESMAN problem, the TRAVELING SALESMAN 1-2 problem, and showed that it satisfies the above condition and is also NP-hard. Thus, the TRAVELING SALESMAN 1-2 problem has no fully polynomial time approximation scheme. From this we derived that the original TRAVELING SALESMAN problem does not have a fully polynomial time approximation scheme. In this lecture, we will formalize this technique and extend it to other optimization problems.

For many optimization problems, such as those we have previously discussed as SUBSET SUM, KNAPSACK, c-PROCESSOR SCHEDULING, and TRAVELING SALESMAN, an input instance is always associated with numbers. Indeed, it is natural to define a number in the problem statement for these problems.

Definition 18.1 Suppose $Q = \langle I_Q, S_Q, f_Q, opt_Q \rangle$ is an optimization problem. For each input instance $x \in I_Q$ we can define:

- length(x) = the length of binary representation of x; and
- max(x) = the largest number that appears in input x.

In particular, if no number appears in the input instance x, we define $\max(x) = 0$.

Definition 18.1 can vary by some degree without loss of the generality of our discussion. For example, length(x) can also denote the length of the decimal representation in the input x or of any other fixed base representation in the input x, and max(x) can be defined to be the sum of all numbers appearing in the input x. Our discussion below will be valid for any of these variations. The point is that for two different definition systems (length(x), max(x)) and (length$'(x)$, max$'(x)$), we require that length(x) and length$'(x)$ are polynomially related and that max(x) and max$'(x)$ are polynomially related for all input instances x.

Definition 18.2 An optimization problem $Q = \langle I_Q, S_Q, f_Q, opt_Q\rangle$ is a *non-number problem* if for all $x \in I_Q$, $\max(x) \leq p(\text{length}(x))$, where p is a fixed polynomial. If there is no such a polynomial p exists, then Q is called a *number problem.*

According to the definition, Subset Sum, Knapsack, c-Processor Scheduling, and Traveling Salesman problems are all number problems. Independent Set is a non-number problem.

Definition 18.3 Let $Q = \langle I_Q, S_Q, f_Q, opt_Q\rangle$ be an optimization problem and let q be any function. Define an optimization problem Q_q to be the sub-problem of Q such that $Q_q = \langle I'_Q, S'_Q, f'_Q, opt'_Q\rangle$, where $I'_Q \subseteq I_Q$, $S'_Q = S_Q$, $f'_Q = f_Q$ and $opt'_Q = opt_Q$, and for all $x \in I'_Q$, $\max(x) \leq q(\text{length}(x))$. In other words, for all input instances x of Q_q, $\max(x)$ is bounded by $q(\text{length}(x))$.

The following definition was first introduced and studied by Garey and Johnson.

Definition 18.4 An optimization problem $Q = \langle I_Q, S_Q, f_Q, opt_Q\rangle$ is *NP-hard in the strong sense* if Q_q is NP-hard for some polynomial q.

The Traveling Salesman problem is an example of optimization problems that are NP-hard in the strong sense, as shown by the following theorem.

Theorem 18.1 *The* Traveling Salesman *problem is NP-hard in the strong sense.*

PROOF. If we denote by Q the TRAVELING SALESMAN problem, then Q_2 corresponds to the TRAVELING SALESMAN 1-2 problem. By Theorem 17.2, the TRAVELING SALESMAN 1-2 problem is NP-hard. Now by the above definition, the TRAVELING SALESMAN problem is NP-hard in the strong sense. $\square$

Remark 18.5 Every non-number NP-hard optimization problem Q is NP-hard in the strong sense. This is because for every non-number NP-hard optimization problem Q, $Q = Q_p$ for some polynomial function p. This implies that Q_p is NP-hard, which, by the definition, further implies that Q is NP-hard in the strong sense.

Theorem 18.2 SUBSET-SUM, KNAPSACK, *and* c-PROCESSOR SCHEDULING *problems are not NP-hard in the strong sense unless P = NP.*

PROOF. Let Q be any one of these problems. From previous lectures, we know that there is an algorithm A such that for an input instance x of Q, the algorithm A constructs an optimal solution to x in time $O(n^c V^d)$ for some constants c and d, where $V \leq n \cdot \max(x)$. Therefore, the algorithm A solves the optimization problem Q in time $O((\text{length}(x))^{c'}(\max(x))^d)$, where c' is a constant.

If Q is NP-hard in the strong sense, then Q_p is NP-hard for some fixed polynomial p. However, for all input instances x of Q_p, $\max(x) \leq p((length)(x))$. Thus, the algorithm A constructs an optimal solution for each input instance x of Q_q in time

$$O((\text{length}(x))^{c'}(\max(x))^d) = O((\text{length}(x))^{c'}((p(\text{length}(x)))^d))$$

which is bounded by a polynomial of length(x). Thus, the problem Q_p is solvable in polynomial time, which implies P = NP. $\square$

The following theorem serves as a fundamental theorem for showing which number problem has no fully polynomial time approximation scheme. We say that a two-parameter function $f(x, y)$ is a polynomial of x and y if $f(x, y)$ can be written as a finite sum of the terms of form $x^c y^d$, where c and d are non-negative integers.

Theorem 18.3 *Let* $Q = \langle I_Q, S_Q, f_Q, opt_Q \rangle$ *be an optimization problem that is NP-hard in the strong sense. Suppose that for all* $x \in I_Q$, $Opt_Q(x)$ *is* *bounded by a polynomial of length(x) and* $\max(x)$. *Then Q has no fully polynomial time approximation scheme unless P = NP.*

PROOF. The proof of this theorem is very similar to the discussion we have given for the TRAVELING SALESMAN problem in the last lecture.

Since Q is NP-hard in the strong sense, Q_q is NP-hard for a polynomial q. Let $Q_q = \langle I'_Q, S_Q, f_Q, opt_Q \rangle$ such that for each input instance $x \in I'_Q$, we have $\max(x) \leq q(\text{length}(x))$. Combining this condition with the condition stated in the theorem that $Opt_Q(x)$ is bounded by a polynomial of length(x) and $\max(x)$, we derive that $Opt_Q(x)$ is bounded by a polynomial of length(x) for all input instances $x \in I'_Q$. Now by Theorem 17.1, the problem Q_q has no fully polynomial time approximation scheme unless P = NP. Since each input instance of Q_q is also an input instance of Q, a fully polynomial time approximation scheme for Q is also a fully polynomial time approximation scheme for Q_q. Now the theorem follows. $\square$

Remark 18.6 How common is the situation that $Opt_Q(x)$ is bounded by a polynomial of length(x), and $\max(x)$? In fact, this situation is fairly common because for most optimization problems, the objective function value is defined through additions or constant number of multiplications on the numbers appearing in the input instance x, which is certainly bounded by a polynomial of length(x) and $\max(x)$. Of course, the condition is not universely true for general optimization problems. For example, an objective function can be simply defined to be the exponentiation of the sum of a subset of input values, which cannot be bounded by any polynomial of length(x) and $\max(x)$.

A general technique for showing the strong NP-hardness for an optimization problem Q is to pick an NP-complete problem L and show that L is polynomial time reducible to Q_q for some polynomial q. Our polynomial time reduction from the HAMILTONIAN CIRCUIT problem to the TRAVELING SALESMAN 1-2 problem given in the last lecture well illustrates this idea.

We give another example of optimization problems that are NP-hard in the strong sense.

MULTI-PROCESSOR SCHEDULING (MPS)

INPUT: $\{t_1, t_2, \ldots, t_n; m\}$, all integers, where t_i is the execution time for the ith job

OUTPUT: a scheduling of the n jobs on m identical processors such that the parallel finish time is minimized

The MULTI-PROCESSOR SCHEDULING problem is NP-hard in the strong sense. In fact, the following restricted version of the MULTI-PROCESSOR SCHEDULING problem is NP-hard in the strong sense.

> THREE-PARTITION
>
> INPUT: $\{t_1, t_2, \ldots, t_{3m}; m\}$, all integers, where t_i is the execution time for the ith job
>
> OUTPUT: a scheduling of the $3m$ jobs on m identical processors such that the parallel finish time is minimized

The reader is advised to read Section 4.2.2 in *Computers and Intractability: A Guide to the Theory of NP-Completeness* by M. Garey and D. Johnson, for a detailed proof that the THREE-PARTITION is NP-hard in the strong sense. Chapter 4 of the above book also contains excellent discussion on strong NP-hardness of optimization problems.

We should point out that for the MULTI-PROCESSOR SCHEDULING problem, when the number of processors is fixed by a constant c, the problem has a fully polynomial time approximation scheme, as we discussed in the previous lectures on the c-PROCESSOR SCHEDULING problem. However, if the number m of processors is given as a variable in the input, then the problem becomes NP-hard in the strong sense. By Theorem 18.3, the problem has no fully polynomial time approximation scheme (it is easy to verify that the condition that $Opt(x)$ is bounded by a polynomial of length(x) and max(x) is satisfied by this problem).

CPSC-669 Computational Optimization

Lecture #19, October 11, 1995

Lecture: Professor Jianer Chen
Scribe: Jennifer Walter
Revision: Jianer Chen

19 Absolute approximability

We have seen a number of NP-hard optimization problems for which we can derive a polynomial time approximation algorithm with approximation ratio bounded by an arbitrary constant $\epsilon > 0$. In this section, we will discuss the approximability of optimization problems in terms of a different measure — the absolute difference.

Definition 19.1 Let $Q = \langle I_Q, S_Q, f_Q, opt_Q \rangle$ be an optimization problem and let $d(n)$ be a function. We say that Q can be approximated with an *absolute difference* $d(n)$ in polynomial time if there is a polynomial time approximation algorithm A for Q such that for any input instance x of Q, the algorithm A produces a solution y to x such that

$$|Opt(x) - f_Q(x, y)| \leq d(|x|)$$

We start the discussion with the famous planar graph coloring problem.

> Planar Graph Coloring
>
> Input: a planar graph G
>
> Output: a coloring of the vertices of G such that no two adjacent vertices are colored with the same color and the number of colors used is minimized.

Theorem 19.1 *The* Planar Graph Coloring *problem is NP-hard.*

PROOF. In fact, the decision problem Planar Graph 3-Colorability: "given a planar graph G, can G be colored with at most 3 colors?" is NP-complete. It is straightforward that the Planar Graph 3-Colorability problem is polynomial time reducible to the Planar Graph Coloring problem. $\square$

Theorem 19.2 *The* Planar Graph Coloring *problem can be approximated in polynomial time with an absolute difference 2.*

proof. First note that there is a well-known and simple process that colors any planar graph with at most 5 colors. Moreover, the process can be implemented by a polynomial time algorithm.

Therefore, given a planar graph G, we first check if G is 2-colorable — this is equivalent to checking if G is a bipartite graph and can be done in linear time. If G is 2-colorable, then we color G with 2 colors and obtain an optimal solution. Otherwise, we need at least 3 colors and we call the above algorithm to color G with at most 5 colors. □

Remark 19.2 By the famous Four-Color Theorem, every planar graph can be colored with at most 4 colors. Therefore, the absolute difference in Theorem 19.2 can actually be replaced by 1. However, since the Four-Color Theorem is too involved, we rather use a much simpler Five-Color Theorem here.

Thus, the Planar Graph Coloring problem can be approximated with a constant absolute difference. On the other hand, the approximation algorithm does not seem very good in term of the approximation ratio. For example, for a planar graph that is 3-colorable, the algorithm can only guarantee a 4-coloring solution. Thus, the approximation ratio for this algorithm is at least $4/3 > 1.3$. The reason for this is that the optimal value of an instance of the problem is always bounded by a constant. Thus, even a small absolute difference makes a significant error in the approximation ratio. Next we give another example for which optimal values are not bounded while the problem still has very good approximation algorithm in terms of the absolute difference.

> Graph Edge Coloring
>
> Input: a graph G
>
> Output: a coloring of the edges of G such that no two adjacent edges are colored with the same color and the number of colors used is minimized.

Remark 19.3 The Graph Edge Coloring problem is NP-hard.

Given a graph G, let v be a vertex of G. Define $deg(v)$ to be the degree of the vertex and define $deg(G)$ to be the maximum $deg(v)$ over all vertices v of G.

The following lemma follows directly from the definition.

Lemma 19.3 *Every edge coloring of a graph G uses at least $deg(G)$ colors.*

Since $deg(G)$ can be arbitrarily large, the optimal value for an instance of the Graph Edge Coloring problem is not bounded by any constant. This is the difference to the case of the Planar Graph Coloring problem.

The next lemma may look more surprising.

Lemma 19.4 *There is a polynomial time algorithm that colors a given graph G with at most $deg(G) + 1$ colors.*

PROOF. Let G be the input graph. To simplify the expression, let $d = deg(G)$. We present an algorithm that colors the edges of G using at most $d + 1$ colors.

The algorithm has the following framework.

Algorithm 19.1 Edge-Coloring

```
Input:  a graph $G$
Output:  an edge coloring of $G$

1.  let $G_0 = G$ with all edges deleted;
    { Suppose that the edges of $G$ are $e_1$, ..., $e_m$ }
2.  for $i = 1$ to $m$ do
      $G_i = G_{i-1} \cup \{e_i\}$;
      color the edges of $G_i$ using at most $d + 1$ colors;
```

We need to explain how the graph G_i can be colored with at most $d+1$ colors. Inductively, suppose that we have colored the edges of G_{i-1} using at most $d + 1$ colors. Now $G_i = G_{i-1} \cup \{e_i\}$, where suppose $e_i = (v_1, w)$. Thus, we have all edges of G_i except e_i colored properly using at most $d+1$ colors.

We say that a vertex u in G_i *misses* a color c if no edge incident on u is colored with c. Since we have $d + 1$ colors and each vertex of G_i has degree at most d, every vertex of G_i misses at least one color.

If both vertices v_1 and w miss a common color c, then we simply color $e_i = (v_1, w)$ with c and we obtain a valid coloring for the graph G_i.

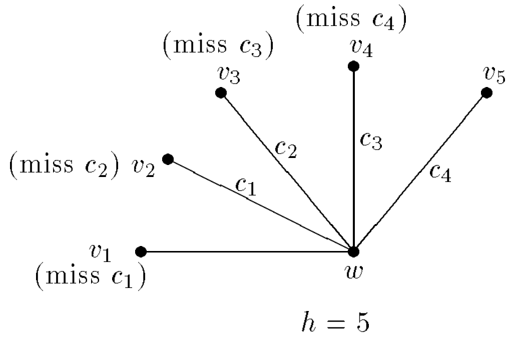

Figure 2: A fan structure

So we suppose that there is no color that is missed by both v_1 and w. Let c_1 be the color missed by v_1 and c_0 be the color missed by w, $c_1 \neq c_0$.

Since c_1 is not missed by w, there is an edge (v_2, w) colored with c_1. Now if v_2 and w have a common missed color, we stop. If v_2 and w have no common missed color, then let c_2 be a color missed by v_2 — c_2 is not missed by w. Now let (v_3, w) be the edge colored with c_2.

Inductively, suppose that we have constructed a "fan" that consists of h neighbors v_1, ..., v_h of w and $h-1$ different colors c_1, ..., c_{h-1}, such that (see Figure 2)

- for all $j = 1, \ldots, h-1$, the vertex v_j misses color c_j and the edge (v_{j+1}, w) is colored with the color c_j;
- none of the vertices v_1, ..., v_{h-1} have a common missed color with w;
- for all $j = 1, \ldots, h-1$, the vertex v_j does not miss any of the colors c_1, ..., c_{j-1}.

There are three possible cases.

Case 1. the vertex v_h does not miss any of the colors c_1, ..., c_{h-1} and v_h has no common missed color w.

Then let c_h be a color missed by v_h. Since c_h is not missed by w, there is an edge (v_{h+1}, w) colored with c_h. Thus, we have expanded the fan structure by one more edge.

Since the degree of the vertex w is finite, **Case 1** must fail at some stage and one of the following two cases should happen.

Case 2. the vertex v_h has a common missed color c_0 with w.

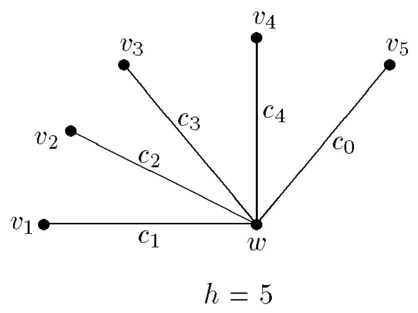

Figure 3: In case v_h and w miss a common color c_0

Then we change the coloring of the fan by coloring (v_h, w) with c_0, and coloring (v_i, w) with c_i, for $i = 1, \ldots, h-1$ (see Figure 3). It is easy to verify that this gives a valid edge coloring for the graph $G_i = G_{i-1} \cup \{e_i\}$.

Case 3. the vertex v_h misses a color c_s, $1 \leq s \leq h-1$.

Let c_0 be a color missed by w. We start from the vertex v_s. Since v_s has no common missed color with w, there is an edge (v_s, u_1) colored with c_0. Now if u_1 does not miss c_s, there is an edge (u_1, u_2) colored with c_s, now we look at vertex u_2 and see if there is an edge colored with c_0, and so on. By this, we obtained a path P_s whose edges are alternatively colored by c_0 and c_s. The path has the following properties: (1) the path P_s must be a simple path since each vertex of the graph G_{i-1} has at most two edges colored with c_0 and c_s. Thus, the path P_s must be a finite path; (2) the path P_s cannot be a cycle since the vertex v_s misses the color c_s; and (3) the vertex w is not an interior vertex of P_s since w misses the color c_0.

Let $P_s = \{v_s, u_1, \ldots, u_t\}$, where v_s misses the color c_s, u_t misses one of colors c_s and c_0, and u_j, $j = 1 \ldots, t-1$, misses neither c_s nor c_0.

If $u_t \neq w$, then interchange the colors c_0 and c_s on the path P_s to make the vertex v_s miss c_0. Then color (v_s, w) with c_0 and color (v_j, w) with c_j, for $j = 1, \ldots, s-1$ (see Figure 4). It is easy to verify that this gives a valid edge coloring for the graph $G_i = G_{i-1} \cup \{e_i\}$.

If $u_t = w$, we must have $u_{t-1} = v_{s+1}$. Then we grow a c_0-c_s alternating path P_h starting from the vertex v_h, which also misses the color c_s. Again P_h is a finite simple path. Moreover, the path P_h cannot end at the vertex w since no vertex in G_{i-1} is incident on more than two edges colored with c_0 and c_s and the vertex w misses the color c_0. Therefore, similar to what we did for vertex v_s, we interchange the colors c_0 and c_s on the path P_h to make v_h miss c_0. Then color (v_h, w) with c_0 and color (v_j, w) with c_j for

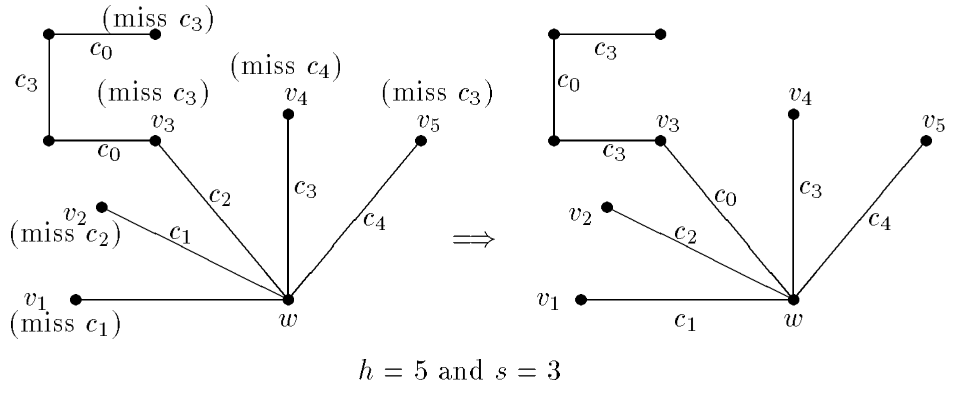

Figure 4: Extending a c_0-c_s alternating path P_s from v_s not ending at w

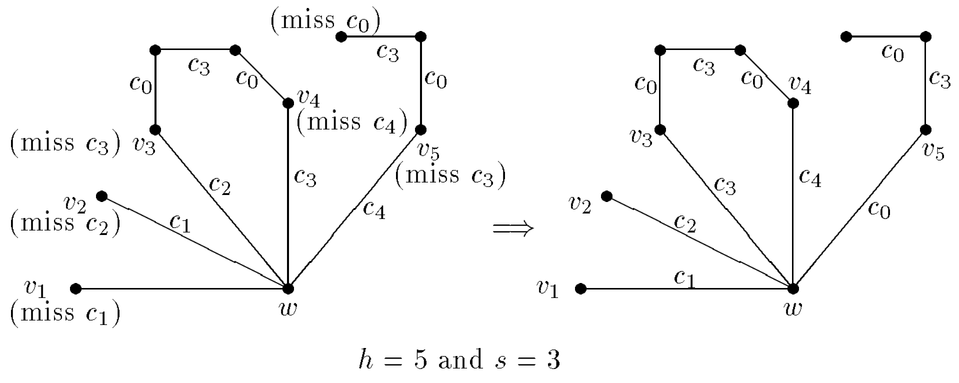

Figure 5: Extending a c_0-c_s alternating path P_h from v_h not ending at w

$j = 1, \ldots, h-1$ (see Figure 5). It is easy to verify that this gives a valid edge coloring for the graph $G_i = G_{i-1} \cup \{e_i\}$.

Therefore, starting with an edge coloring of the graph G_{i-1} using at most $d+1$ colors, we can always derive a valid edge coloring for the graph $G_i = G_{i-1} \cup \{e_i\}$ using at most $d+1$ colors. It is also easy to see that this process can be implemented by a polynomial time algorithm. We leave the detailed implementation of this process to the interested reader.

Now we conclude that the algorithm `Edge-Coloring` runs in polynomial time and produces a valid edge coloring using at most $d+1$ colors for the graph G. $\Box$

Theorem 19.5 *The* Graph Edge Coloring *problem can be approximated within an absolute difference of 1 in polynomial-time.*

PROOF. Follows directly from Lemma 19.3 and 19.4. □

Remark 19.4 Theorem 19.5 seems to give the best possible polynomial time approximation algorithm for the NP-hard GRAPH EDGE COLORING problem. On the other hand, this algorithm does not provide a fully polynomial time approximation scheme for the problem. Indeed, the decision problem GRAPH EDGE 3-COLORABILITY "given a graph G, can the edges of G be colored using no more than 3 colors" is NP-complete. Thus, the algorithm from Theorem 19.5 can only guarantee a 4-coloring for an instance of the GRAPH EDGE 3-COLORABILITY problem, which has an approximation ratio at least $4/3 > 1.3$.

It is natural to ask whether the optimization problems that have fully polynomial time approximation scheme should have good approximation algorithms in terms of absolute difference. It is, in fact, not very difficult to show that this is not always the case.

Recall the KNAPSACK problem.

> KNAPSACK
>
> INPUT: $(s_1, \ldots, s_n; v_1, \ldots, v_n; B)$, all integers
>
> OUTPUT: a subset S of $\{1, \ldots, n\}$ such that $\sum_{i \in S} s_i \leq B$ and $\sum_{i \in S} v_i$ is maximized

Theorem 19.6 *There is no polynomial time approximation algorithm for the* KNAPSACK *problem that guarantees an absolute difference* 2^n *unless P = NP.*

PROOF. Suppose that A is a polynomial time approximation algorithm for the KNAPSACK problem $Q = \langle I, S, f, opt \rangle$ such that for any input instance X of Q, A produces a solution S such that $|Opt(X) - A(X)| \leq 2^n$, where $A(X) = f(X, S)$. We show how we can use this algorithm to solve the KNAPSACK problem in polynomial time.

Given an input instance $X = (s_1, \ldots, s_n; v_1, \ldots, v_n; B)$ for Q, we construct $X' = (s_1, \ldots, s_n; v_1 2^{n+1}, \ldots, v_n 2^{n+1}; B)$ (i.e. scale the values v_i to be a multiple of 2^{n+1} so that a difference of 2^n between two values makes no difference).

Now apply the algorithm A to X' to get a solution S with value $A(X') = f(X', S)$. According to our assumption, $|Opt(X') - A(X')| \leq 2^n$. Since both $Opt(X')$ and $A(X')$ are multiples of 2^{n+1}, we conclude that $Opt(X') = A(X')$, that is, the solution S is an optimal solution to the instance X'. Moreover, it is easy to see that S is also a solution to the instance X and $Opt(X') = 2^{n+1}Opt(X)$ and $A(X') = 2^{n+1}A(X)$. Therefore, S is also an optimal solution for the instance X.

By our assumption, the algorithm A runs in polynomial time. It is also easy to see that we can construct the instance X' from the instance X in polynomial time. Therefore, the above process constructs an optimal solution for X in polynomial time. Consequently, the Knapsack problem can be solved in polynomial time. Since the Knapsack problem is NP-hard, it follows that P = NP. □

This proof for Theorem 19.6 can be easily extended to other number problems such as the c-Processor Scheduling problem and the Subset Sum problems.

The main reason that Theorem 19.6 holds for many number problems is that we can scale the numbers in the input instances so that a small absolute difference would make no difference for the scaled instance. However, what about non-number problems? In particular, is there a similar theorem for optimization problems whose instances contain no number at all? We demonstrate a technique for this via the study of an optimization problem related to graph embeddings.

Graph embeddings can be studied using *graph rotation systems*. A *rotation* at a vertex v is a cyclic permutation of the edge-ends incident on v. A list of rotations, one for each vertex of the graph, is called a *rotation system.*

An embedding of a graph G in an orientable surface induces a rotation system, as follows: the rotation at vertex v is the cyclic permutation corresponding to the order in which the edge-ends are traversed in an orientation-preserving tour around v. Conversely, it is known that every rotation system induces a unique embedding of G into an orientable surface. In the following, we will interchangeably use the phrases "an embedding of a graph" and "a rotation system of a graph".

The *genus* $\gamma(\Pi(G))$ *of the rotation system* $\Pi(G)$ is defined by the Euler polyhedral equation

$$|V| - |E| + |F| = 2 - 2\gamma(\Pi(G))$$

where $|F|$ is the number of faces in the embedding $\Pi(G)$. It can be proved that the value $\gamma(\Pi(G))$ is actually the number of "holes" of the surface on which the embedding $\Pi(G)$ is realized. Consequently, $\gamma(\Pi(G))$ is always a non-negative integer. There is a linear time algorithm that, given a rotation system $\Pi(G)$ for a graph G, traces the boundary walks of all faces in the rotation system. Therefore, given a rotation system $\Pi(G)$, the genus $\gamma(\Pi(G))$ of $\Pi(G)$ can be computed in linear time.

Now we are ready to state the following problem.

> Graph Genus
>
> Input: a graph G
>
> Output: an embedding $\Pi(G)$ of G such that the genus $\gamma(\Pi(G))$ is minimized. Such a value is called the *minimum genus* of the graph G, written as $\gamma_{\min}(G)$

It is known that the Graph Genus problem is NP-hard. The Graph Genus problem has applications in circuit layouts and distributed computation.

Let G and G' be two graphs. The *bar-amalgamation* of G and G', denoted $G * G'$, is the result of running a new edge (called the "bar") from a vertex of G to a vertex of G'. The definition of bar-amalgamation on two graphs can be extended to more than two graphs. Inductively, a *bar-amalgamation* of r graphs $G_1, \ldots, G_r$, written $G_1 * G_2 * \cdots * G_r$, is the bar-amalgamation of the graph G_1 and the graph $G_2 * \cdots * G_r$.

Let G be a graph and let H be a subgraph of G. Let $\Pi(G)$ be a rotation system of G. A rotation system $\Pi'(H)$ of H can be obtained from $\Pi(G)$ by deleting all edges that are not in H. The rotation system $\Pi'(H)$ of H will be called an *induced rotation system* of H from the rotation system $\Pi(G)$.

The proofs for the following theorem and corollary are omitted.

Theorem 19.7 *Let $G_1, \cdots, G_r$ be graphs and let $\Pi(G_1 * \cdots * G_r)$ be a rotation system of a bar-amalgamation $G_1 * \cdots * G_r$ of $G_1, \cdots, G_r$. Then*

$$\gamma(\Pi(G_1 * \cdots * G_r)) = \sum_{i=1}^{r} \gamma(\Pi_i(G_i))$$

where $\Pi_i(G_i)$ is the induced rotation system of G_i from $\Pi(G_1 * \cdots * G_r)$, $1 \leq i \leq r$.

Corollary 19.8 *Let $G_1, \cdots, G_r$ be graphs and let G' be an arbitrary bar-amalgamation of $G_1, \cdots, G_r$. Then*

$$\gamma_{\min}(G') = \sum_{i=1}^{r} \gamma_{\min}(G_i)$$

Now we are ready for the main theorem.

Theorem 19.9 *For any fixed constant ϵ, $0 \le \epsilon < 1$, the* Graph Genus *problem cannot be approximated in polynomial time with an absolute difference n^ϵ unless $P = NP$.*

PROOF. Suppose that A is an approximation algorithm that, given a graph G of n vertices, constructs an embedding of G of genus at most $\gamma_{\min}(G) + n^\epsilon$.

Let k be an integer such that $\epsilon < \frac{k}{k+1}$. Then for sufficiently large n, we have $n^\epsilon < n^{\frac{k}{k+1}}$. Thus $n^{\epsilon(k+1)} \le n^k - 1$.

Let n^kG be a graph that is an arbitrary bar amalgamation of n^k copies of G. Then the number of vertices of n^kG is $N = n^{k+1}$. The graph n^kG can be obviously constructed from G in polynomial time. Moreover, by Corollary 19.8

$$\gamma_{\min}(n^kG) = n^k \cdot \gamma_{\min}(G)$$

Now running the algorithm A on the graph n^kG gives us an embedding $\Pi(n^kG)$ of n^kG, which has genus at most $\gamma_{\min}(n^kG) + N^\epsilon$. Therefore,

$$\begin{aligned} \gamma(\Pi(n^kG)) &\le \gamma_{\min}(n^kG) + N^\epsilon \\ &= n^k\gamma_{\min}(G) + n^{\epsilon(k+1)} \\ &\le n^k\gamma_{\min}(G) + n^k - 1 \end{aligned} \qquad (6)$$

On the other hand, if we let $\Pi_1(G), \cdots, \Pi_{n^k}(G)$ be the n^k induced rotation systems of G from $\Pi(n^kG)$, then by Theorem 19.7

$$\gamma(\Pi(n^kG)) = \sum_{i=1}^{n^k} \gamma(\Pi_i(G)) \qquad (7)$$

Combining Equations (6) and (7) and noticing that the genus of $\Pi_i(G)$ is at least as large as $\gamma_{\min}(G)$ for all $1 \le i \le n^k$, we conclude that at least one induced rotation system $\Pi_i(G)$ of G achieves the minimum genus $\gamma_{\min}(G)$. This rotation system of G can be found by calculating the genus for each induced rotation system $\Pi_i(G)$ from $\Pi(n^kG)$ and selecting the one with the smallest genus. This can be accomplished in polynomial time.

Therefore, using the algorithm A, we would be able to construct in polynomial time a minimum genus embedding for the graph G. Consequently, the Graph Genus problem can be solved in polynomial time. Since the Graph Genus problem is NP-hard, we would derive P = NP. □

The technique of Theorem 19.9 can be summarized as follows. Let $Q = \langle I_Q, S_Q, f_Q, opt_Q \rangle$ be an optimization problem such that there is an operator $\oplus$ implementable in polynomial time that can "compose" input instances, i.e., for any two input instances x and y of Q, $x \oplus y$ is also an input instance of Q such that $|x \oplus y| = |x| + |y|$ (in the case of Theorem 19.9, $\oplus$ is the bar-amalgamation). Moreover, suppose that from a solution $s_{x \oplus y}$ to the instance $x \oplus y$, we can construct in polynomial time solutions s_x and s_y for the instances x and y, respectively such that

$$f_Q(x \oplus y, s_{x \oplus y}) = f_Q(x, s_x) + f_Q(y, s_y)$$

(this corresponds to Theorem 19.7) and

$$Opt(x \oplus y) = Opt(x) + Opt(y)$$

(this corresponds to Corollary 19.8), then using the technique of Theorem 19.9, we can prove that the problem Q cannot be approximated in polynomial time with a absolute difference n^ϵ for any constant $\epsilon < 1$ unless Q can be solved precisely in polynomial time. In particular, if Q is NP-hard, then Q cannot be approximated in polynomial time with a absolute difference n^ϵ for any constant $\epsilon < 1$ unless P = NP.

As an easy exercise, readers are advised to use this technique to prove that the INDEPENDENT SET problem cannot be approximated in polynomial time with an absolute difference n^ϵ for any constant $\epsilon < 1$.

CPSC-669 Computational Optimization

Lecture #20, October 13, 1995

Lecturer: Professor Jianer Chen
Scribe: Mitrajit Chatterjee
Revision: Jianer Chen

20 Planar Independent Set

The algorithm `Knapsack-Dyn` for the KNAPSACK problem and the algorithm c-`Scheduling-Dyn` for the c-PROCESSOR SCHEDULING problem share a common property that the algorithms run in time polynomial in length(x) and max(x) on an input instance x, where length(x) and max(x) are as defined in Definition 18.1. This motivates the following definition.

Definition 20.1 An algorithm A that solves an optimization problem $Q = \langle I, S, f, opt\rangle$ is a *pseudo-polynomial time algorithm* if on any input instance $x \in I$, the running time of A is bounded by a polynomial of length(x) and max(x). In this case, we say that the optimization problem Q can be solved in pseudo-polynomial time.

Most fully polynomial time approximation scheme algorithms are derived from pseudo-polynomial time algorithms for the same problem by properly scaling and rounding the input data. On the other hand, the following theorem shows that under a very general condition, the existence of a fully polynomial time approximation scheme implies the existence of a pseudo-polynomial time algorithm.

Theorem 20.1 *Let $Q = \langle I, S, f, opt\rangle$ be an optimization problem such that for all input instance $x \in I$ we have $Opt(x) \leq p(length(x), \max(x))$, where p is a two variable polynomial. If Q has a fully polynomial time approximation scheme, then Q can be solved in pseudo-polynomial time.*

PROOF. Suppose Q is a minimization problem, i.e., $opt = \min$. Since Q has a fully polynomial time approximation scheme, there is an approximation algorithm A for Q such that for any input instance $x \in I$, the algorithm A produces a solution $y \in S(x)$ in time $p_1(|x|, 1/\epsilon)$ satisfying

$$\frac{f(x,y)}{Opt(x)} \leq 1 + \epsilon$$

where p_1 is a two variable polynomial.

In particular, let $\epsilon = 1/(p(\text{length}(x), \max(x)) + 1)$, then the solution y satisfies

$$f(x,y) \leq Opt(x) + \frac{Opt(x)}{p(\text{length}(x), \max(x)) + 1} < Opt(x) + 1$$

Now since both $f(x,y)$ and $Opt(x)$ are integers and $f(x,y) \geq Opt(x)$, we get immediately $f(x,y) = Opt(x)$. That is, the solution produced by the algorithm A is actually an optimal solution. Moreover, the running time of the algorithm A for producing the solution y is bounded by

$$p_1(|x|, p(\text{length}(x), \max(x)) + 1)$$

which is a polynomial of length(x) and $\max(x)$. We conclude that the optimization problem Q can be solved in pseudo-polynomial time. $\square$

Theorem 17.1 gives a fairly convenient way for checking that an optimization problem has no fully polynomial time approximation scheme. How well can this kind of problems be approximated? In the following, we will show that for certain problems that have no fully polynomial time approximation scheme, polynomial time approximation algorithms with approximation ratio $1 + \epsilon$ are still possible, for any fixed constant $\epsilon > 0$.

The first problem to be considered is the Independent Set problem on planar graphs, defined as follows.

Planar Independent Set (IS)= $\langle I, S, f, opt \rangle$

I: the set of all planar graphs G

$S(G)$: the collection of all subsets D of the vertices of the graph G such that no two vertices in D are adjacent

$f(G, D)$: the number of vertices in D

opt: max

It is known that the Planar Independent Set problem is NP-hard. Moreover, by Theorem 17.1, the Planar Independent Set problem has no fully polynomial time approximation scheme unless P = NP.

The following theorem by Lipton and Tarjan plays a key role in the approximation algorithm for the Planar Independent Set problem. The proof of the theorem is omitted.

Theorem 20.2 *(Separator Theorem) For any planar graph $G = (V, E)$, $|V| = n$, one can partition the vertex set V of G into three disjoint sets, A, B, and C, such that*

1. *$|A|, |B| \leq 2n/3$;*
2. *$|C| \leq \sqrt{8n}$; and*
3. *C separates A and B, i.e. there is no edge between A and B.*

Moreover, there is a linear time algorithm that, given a planar graph G, constructs the triple (A, B, C) as above.

Let $G = (V, E)$ be a planar graph and let (A, B, C) be a triple satisfying the conditions of Theorem 20.2. We will say that *the graph G is split into two smaller pieces A and B* (using the separator C). A simple observation is that if D_A and D_B are independent sets of the graphs induced by the vertex sets A and B, respectively, then the union $D_A \cup D_B$ is an independent set of the graph G. Moreover, since the sizes of the sets A and B are of order $\Omega(n)$ while the size of the separator C is of order $O(\sqrt{n})$, ignoring the vertices in the separator C does not seem to lose too much precision. This idea is implemented by the following algorithm, where K is a constant to be determined later.

Algorithm 20.1 `PlanarIndSet`(K)

```
Input:  a planar graph $G = (V, E)$
Output:  an independent Set $S$ in $G$

1.  **If** ($|V| \leq K$) **then**
       find a maximum indepenent set $D$ in $G$ using
          exhaustive search;
       Return($D$);
{At this point $|V| > K$.}
2.  split $V$ into $(A, B, C)$ as in Theorem 20.2;
3.  recursively find an independent set $D_A$ for $A$ and an
       independent set $D_B$ for $B$;
4.  return($D_A \cup D_B$).
```

By the discussion above, the algorithm `PlanarIndSet` correctly returns an independent set for the graph G. Thus, it is an approximation algorithm for the PLANAR INDEPENDENT SET problem. We first study a few properties of this algorithm.

The algorithm splits the graph G into small pieces. If the size of a piece is larger than K, then the algorithm splits the piece into two smaller pieces in linear time according to Theorem 20.2. Otherwise, it finds a maximum independent set for the piece using brute force method. We first discuss the number of pieces whose size is within a certain region.

A piece is *at level* 0 if its size is not larger than K. For a general $i \geq 0$, a piece is *at level* i if its size (i.e., the number of vertices in the piece) is in the region $((3/2)^{i-1}K, (3/2)^i K]$, i.e., if its size is larger than $(3/2)^{i-1}K$ but not larger than $(3/2)^i K$. Note that the largest level number is bounded by $\log(n/K)/\log(3/2) = O(\log(n/K))$.

Lemma 20.3 *For a fixed i, each vertex of the graph G belongs to at most one piece at level i.*

PROOF. Fix a vertex v of the graph G.

Suppose that the largest level number is h and that the graph G is at level h. By the definition, $n \leq (3/2)^h K$. Now according to Theorem 20.2, G is split into two pieces A and B, whose size is bounded by

$$2n/3 \leq (2/3)(3/2)^h K = (3/2)^{h-1} K$$

Thus, both pieces A and B do not belong to level h. Consequently, G is the only piece at level h. Thus, there is only one piece at level h that contains the vertex v.

Inductively, suppose that for each $i \geq j$, at most one piece at level i contains the vertex v and there is a piece P at level j that contains the vertex v. If $j = 0$, then we are done. Otherwise, let P_1 and P_2 be the two smaller pieces obtained by splitting the piece P according to Theorem 20.2. As we proved above for level h, no P_1 and P_2 can be at level j. Moreover, at most one of P_1 and P_2 can contain the vertex v. Without loss of generality, suppose that P_1 contains v and that P_1 is at level $j' < j$. Now for each $i \geq j'$, at most one piece at level i contains the vertex v. The induction goes through. $\square$

Therefore, all pieces at level i are disjoint. Since each piece at level i consists of more than $(3/2)^{i-1}K$ vertices, there are no more than $(2/3)^{i-1}(n/K)$ pieces at level i, for all i. We summarize these discussions as follows.

- There are no more than n pieces at level 0, each is of size at most K;
- For each fixed $i > 0$, there are no more than $(2/3)^{i-1}(n/K)$ pieces at level i, each is of size bounded by $(3/2)^i K$; and
- There are at most $O(\log n)$ levels.

Now we are ready to analyze the algorithm.

Lemma 20.4 *The running time of the algorithm* `PlanarIndSet` *is bounded by* $O(n \log n + 2^K n)$.

PROOF. For each piece at level $i > 0$, we apply Theorem 20.2 to split it into two smaller pieces in time linear to the size of the piece. Since the total number of vertices belonging to pieces at level i is bounded by n, we conclude that the total time spent by the algorithm `PlanarIndSet` on pieces at level i is bounded by $O(n)$ for each $i > 0$. Since there are only $O(\log n)$ levels, the algorithm `PlanarIndSet` takes time $O(n \log n)$ on piece splitting.

For each piece P at level 0, which has size bounded by K, the algorithm finds a maximum independent set by checking all subsets of vertices of the piece P. There are at most 2^K such subsets in P, and each such a subset can be checked in time linear to the size of the piece. We conclude that the algorithm `PlanarIndSet` spends time $O(2^K n)$ on pieces at level 0. In summary, the running time of the algorithm `PlanarIndSet` is bounded by $O(n \log n + 2^K n)$. $\square$

Let us consider the approximation ratio for the algorithm `PlanarIndSet`.

Fix an $i > 0$. Suppose that we have l pieces of size n_1, n_2, ..., n_l at level i. For each such a piece of size n_q, a separator of size less than $3\sqrt{n_q}$ is constructed to split the piece into two smaller pieces. The vertices in the separator will be ignored in the further consideration. There are at most

$$3\sqrt{n_1} + 3\sqrt{n_2} + \cdots + 3\sqrt{n_l}$$

vertices that belong to separators for pieces at level i. It is well-known that the above summation will be maximized when all n_1, n_2, ..., n_l are equal. As $n_1 + n_2 + \cdots + n_l \le n$, each n_q can be at most (n/l). Hence, the above summation is bounded by

$$\underbrace{3\sqrt{n/l} + 3\sqrt{n/l} + \cdots + 3\sqrt{n/l}}_{l \text{ terms}} = 3l\sqrt{n/l} = 3\sqrt{nl}$$

Now, since $l \leq (2/3)^{i-1}(n/K)$ (as derived above), the total number of vertices belonging to separators for pieces at level i is bounded by

$$3\sqrt{n \times \left(\frac{2}{3}\right)^{i-1} \frac{n}{K}} = \frac{3n}{\sqrt{K}} \left(\frac{2}{3}\right)^{\frac{i-1}{2}}$$

Let F denote the set of all vertices that belong to a separator at some level. We derive

$$|F| \leq \sum_{i=1}^{h} \left(\frac{3n}{\sqrt{K}}\right) \left(\sqrt{\frac{2}{3}}\right)^{i-1} \leq \left(\frac{3n}{\sqrt{K}}\right) \sum_{i=1}^{\infty} \left(\sqrt{\frac{2}{3}}\right)^{i-1} = \frac{3nd}{\sqrt{K}}$$

where $d = \sum_{i=1}^{\infty}(\sqrt{2/3})^{i-1}$ is a constant.

Lemma 20.5 *Let S be the solution produced by the algorithm* `PlanarIndSet`*. Then $Opt(G) \leq |S| + |F|$.*

PROOF. Let P be a piece at level 0 and let $S_{\max}$ be a maximum independent set of the graph G. It is easy to see that $S_{\max} \cap P$ is an independent set in the piece P, which cannot be larger than the maximum independent set $S_{\max}^P$ of P constructed by the algorithm `PlanarIndSet`. Note that S is the union of $S_{\max}^P$ over all pieces at level 0. We have

$$S_{\max} = \bigcup_{P:\ \text{level 0 piece}} (S_{\max} \cap P) \cup (S_{\max} \cap F)$$

Therefore,

$$\begin{aligned} Opt(G) &= |S_{\max}| \leq \sum_{P:\ \text{level 0 piece}} (|S_{\max} \cap P|) + |S_{\max} \cap F| \\ &\leq \sum_{P:\ \text{level 0 piece}} |S_{\max}^P| + |F| \\ &= |S| + |F| \end{aligned}$$

The lemma is proved. $\square$

From Lemma 20.5, we get immediately

$$\frac{Opt(G)}{|S|} \leq 1 + \frac{|F|}{Opt(G) - |F|}$$

Since the graph G is planar, by the famous Four-Color Theorem, G can be colored with at most 4 colors such that no two adjacent vertices in G are of the same color. It is easy to see that all vertices with the same color form an independent set for G. We conclude that the size $Opt(G)$ of the maximum independent set $S_{\max}$ of G is at least a quarter of the size n of the graph G.

Combining $Opt(G) \geq n/4$ with $|F| \leq 3nd/\sqrt{K}$, we obtain

$$\begin{aligned}\frac{Opt(G)}{|S|} &\leq 1 + \frac{|F|}{Opt(G) - |F|} \leq 1 + \frac{|F|}{(n/4) - |F|} \\ &\leq 1 + \frac{3nd/\sqrt{K}}{(n/4) - 3nd/\sqrt{K}} = 1 + \frac{12d}{\sqrt{K} - 12d}\end{aligned}$$

Now for any fixed constant ϵ, if we let

$$K = (12d(1 + 1/\epsilon))^2 = 144d^2(1 + 1/\epsilon)^2$$

then the algorithm `PlanarIndSet`(K) produces an independent set S for G with approximation ratio

$$\frac{Opt(G)}{|S|} \leq 1 + \epsilon$$

in time $O(n \log n + n2^{144d^2(1+1/\epsilon)^2})$. For a fixed $\epsilon > 0$, this is a polynomial time algorithm. However, this is *not* a fully polynomial time approximation scheme since its time complexity is not bounded by a polynomial of n and $1/\epsilon$.

CPSC-669 Computational Optimization

Lecture #21, October 16, 1995

Lecturer: Professor Jianer Chen
Scribe: Mitrajit Chatterjee
Revision: Jianer Chen

21 Δ-TSP: first algorithm

The approximation algorithm `PlanarIndSet` for the Planar Independent Set problem motivates the following definition.

Definition 21.1 An optimization problem Q has a *polynomial time approximation scheme* (PTAS), if for any fixed constant $\epsilon > 0$, there is a polynomial time approximation algorithm for Q with approximation ratio bounded by $1 + \epsilon$.

Note that a polynomial time approximation scheme does not require the running time of the approximation algorithm to be bounded by a polynomial of $1/\epsilon$.

The previous lecture shows that the Planar Independent Set problem has a polynomial time approximation scheme. According to Theorem 17.1, the Planar Independent Set problem has no fully polynomial time approximation scheme unless P = NP. Thus, a polynomial time approximation scheme seems the best we can hope for the problem.

Other optimization problems that have polynomial time approximation schemes but have no fully polynomial time approximation schemes include the Planar Vertex Cover problem and some other optimization problems on planar graphs. Most of these polynomial time approximation scheme algorithms use the similar technique as the one we described for the Planar Independent Set problem, i.e., using Separator Theorem (Theorem 20.2) to separate a planar graph into small pieces by separators of small size and using brute force method to solve the problem for the small pieces. Students are encouraged to apply this technique to derive a polynomial time approximation scheme for the Planar Vertex Cover problem.

A difference separating technique has been proposed by Baker (1994) to derive polynomial time approximation scheme for optimization problems on planar graphs. We briefly describe the idea here based on the Planar

Independent Set. Let G be a planar graph. Embed G into the plane. Now the vertices on the unbounded face of the embedding give the first *layer* of the graph G. By peeling the first layer, i.e., deleting the vertices in the first layer, we obtain (maybe more than one) several separated pieces, each of which is a planar graph embedded in the plane. Now the first layers of these pieces form the second layer for the graph G. By peeling the second layer of G, we obtain the third layer, and so on. Define *depth* of the planar graph G to be the maximum number of layers of the graph. Baker observed that for a graph of constant depth, a maximum independent set can be constructed in polynomial time by dynamic programming. Moreover, for any graph G of arbitrary depth, if we remove one layer out of every K consecutive layers, we obtain a set of separated planar graphs of constant depth. Now for each such graph of constant depth, we construct a maximum independent set. The union of these maximum independent sets forms an independent set for the original graph G. For sufficiently large K, the number of vertices belonging to the removed layers is very small and thus gives only a small error in the approximation. Baker demonstrated a polynomial time approximation scheme for the Planar Independent Set problem with running time bounded by $O(8^{1/\epsilon}n/\epsilon)$.

Another optimization problem that has a polynomial time approximation scheme but has no fully polynomial time approximation scheme is the Multi-Processor Scheduling problem. The polynomial time approximation scheme algorithm for this problem is closely related to approximation algorithms for the Bin Packing problem. We will discuss this after the study of approximation algorithms for the Bin Packing problem.

We will study in this lecture a restricted version of the Traveling Salesman problem.

Definition 21.2 Let $G = (V, E)$ be a weighted, undirected, and complete graph. We say that the graph G satisfies the *triangle inequality* if for any three vertices u, v, and w of G we have

$$\text{weight}(u, w) \leq \text{weight}(u, v) + \text{weight}(v, w)$$

The Traveling Salesman problem under triangle inequality is defined as follows.

Δ-Traveling Salesman Problem (Δ-TSP)

Input: a weighted, undirected, and complete graph G
satisfying the triangle inequality

OUTPUT: a simple cycle of minimum weight that contains all vertices of G

Let G be an input instance of the Δ-TSP. Every solution to G, i.e., every simple cycle in G that contains all vertices of G, will be called a *traveling salesman tour*.

An important case of the Δ-TSP is the EUCLIDEAN TSP, in which each vertex is a point in the Euclidean plane and the weight of an edge (w, u) equals the Euclidean distance between w and u.

Remark 21.3 Both Δ-TSP and EUCLIDEAN TSP are NP-hard.

We present the first approximation algorithm for the Δ-TSP based on minimum spanning trees.

Algorithm 21.1 `EasyTSP`

```
Input:  an input instance $G$ of $\Delta$-TSP
Output:  a traveling salesman tour $L$

1.  construct a minimum spanning tree $T$ for $G$;
2.  perform a depth first search on the tree $T$ to compute
      the dfs number for each vertex of $T$;
3.  let $L$ be the list of vertices of $T$ sorted by their dfs
      numbers;
4.  return $L$ as a traveling salesman tour for $G$.
```

The analysis of the time complexity of the above algorithm is pretty simple. It is well-known that a minimum spanning tree of a graph of n vertices can be constructed in time $O(n^2)$. It is also easy to see that each of the steps 2, 3, and 4 takes time bounded by $O(n^2)$. Therefore, the algorithm `EasyTSP` runs in time $O(n^2)$.

Now we consider the approximation ratio for the algorithm. A depth first search process that computes the dfs numbers for vertices of a tree can be implemented by the following simple algorithm.

Algorithm 21.2 `DFS(`v`)`

```
1.  counter = counter + 1;
2.  dfs[$v$] = counter;
3.  for each child $w$ of $v$ do
        DFS($w$);
```

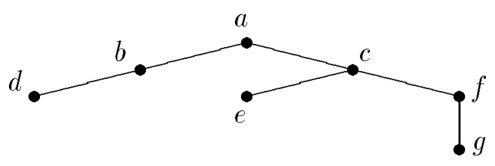

Figure 6: The minimum spanning tree T

This recursive subroutine is called by the following main program.

Algorithm 21.3 `Main`

```
{suppose that vertex 1 is the root of the tree $T$}
1.  counter = 0;
2.  DFS(1).
```

The depth first search process on the tree T can be regarded as a closed walk L_0 of the tree (a *closed walk* is a cycle in T in which each vertex may appear more than once). Each edge (u, v), where u is the father of v in T, is traversed exactly twice in the walk L_0: the first time when `DFS`(u) calls `DFS`(v) we traverse the edge from u to v, and the second time when `DFS`(v) is finished and returns back to `DFS`(u) we traverse the edge from v to u. Therefore, the walk L_0 has weight exactly twice the weight of the tree T. It is also easy to see that the list L produced by the algorithm `EasyTSP` can be obtained from the walk L_0 by deleting for each vertex v all but the first occurrences of v in the list L_0. Since each vertex appears exactly once in the list L and G is a complete graph, L corresponds to a traveling salesman tour.

Example 21.4 Consider the tree T in Figure 6, where a is the root of the tree T. The depth first search process traverses the tree T in the order

$$a, b, d, b, a, c, e, c, f, g, f, c, a$$

By deleting for each vertex v all but the first vertex occurrences for v, we obtain the list of vertices of the tree T sorted by their dfs numbers

$$a, b, d, c, e, f, g$$

Deleting a vertex occurrence of v in the list $\{\cdots uvw \cdots\}$ is equivalent to replacing the path $u \to v \to w$ by a single edge (u, w). Since the graph G satisfies the triangle inequality, deleting vertex occurrences from a walk does not increase the weight of the walk. Consequently, the weight of the traveling salesman tour L is not larger than the weight of the closed walk L_0, which is bounded by 2 times the weight of the minimum spanning tree T.

Note that for the TRAVELING SALESMAN problem, we can assume without loss of generality that all edge weights are non-negative integers (otherwise we add a sufficiently large weight to each edge). Observe that the weight of any traveling salesman tour is at least as large as the weight of the minimum spanning tree T — removing any edge (of non-negative weight) from the traveling salesman tour results in a spanning tree of the graph G. In conclusion, the traveling salesman tour L constructed by the algorithm `EasyTSP` has weight bounded by 2 times the weight of a minimum traveling salesman tour. We conclude with the following theorem.

Theorem 21.1 *The approximation ratio of the algorithm* `EasyTSP` *is bounded by* 2.

We give a simple example to show that the ratio 2 is tight for the approximation algorithm `EasyTSP` in the sense that there are input instances for the Δ-TSP for which the algorithm `EasyTSP` produces a solution with approximation ratio arbitrarily close to 2. This kind of input instances can actually appear for the EUCLIDEAN TSP. Consider the figures in Figure 7. Suppose we are given $2n$ points on the Euclidean plane with polar coordinates $x_k = (b, 2k\pi/n)$ and $y_k = (b + d, 2k\pi/n)$, $k = 1, \ldots, n$, where d is much smaller than b. See Figure 7(a), where $n = 8$. Then it is not hard (for example, by Kruskal's algorithm for minimum spanning tree) to see that the edges (x_k, x_{k+1}), $k = 1, \ldots, n-1$ and (x_j, y_j), $j = 1, \ldots, n$ form a minimum spanning tree T for the set of points. See Figure 7(b). Now if we perform a depth first search on T starting from the vertex x_1 and construct a traveling salesman tour, we will get a tour L that is shown in Figure 7(c) while an optimal traveling salesman tour L_0 is shown in Figure 7(d).

The weight of the tour L is about $2a(n-1) + 2d$, where a is the distance between two adjacent points x_k and x_{k+1} (note that when d is sufficiently small compared with a, the distance between two adjacent points y_k and y_{k+1} is roughly equal to the distance between the two corresponding points x_k and x_{k+1}), while the optimal traveling salesman tour has weight roughly $nd + na$. When d is sufficiently small compared with a and when n is sufficiently large, the ratio of the weight of the tour L and the weight of the tour L_0 can be arbitrarily close to 2.

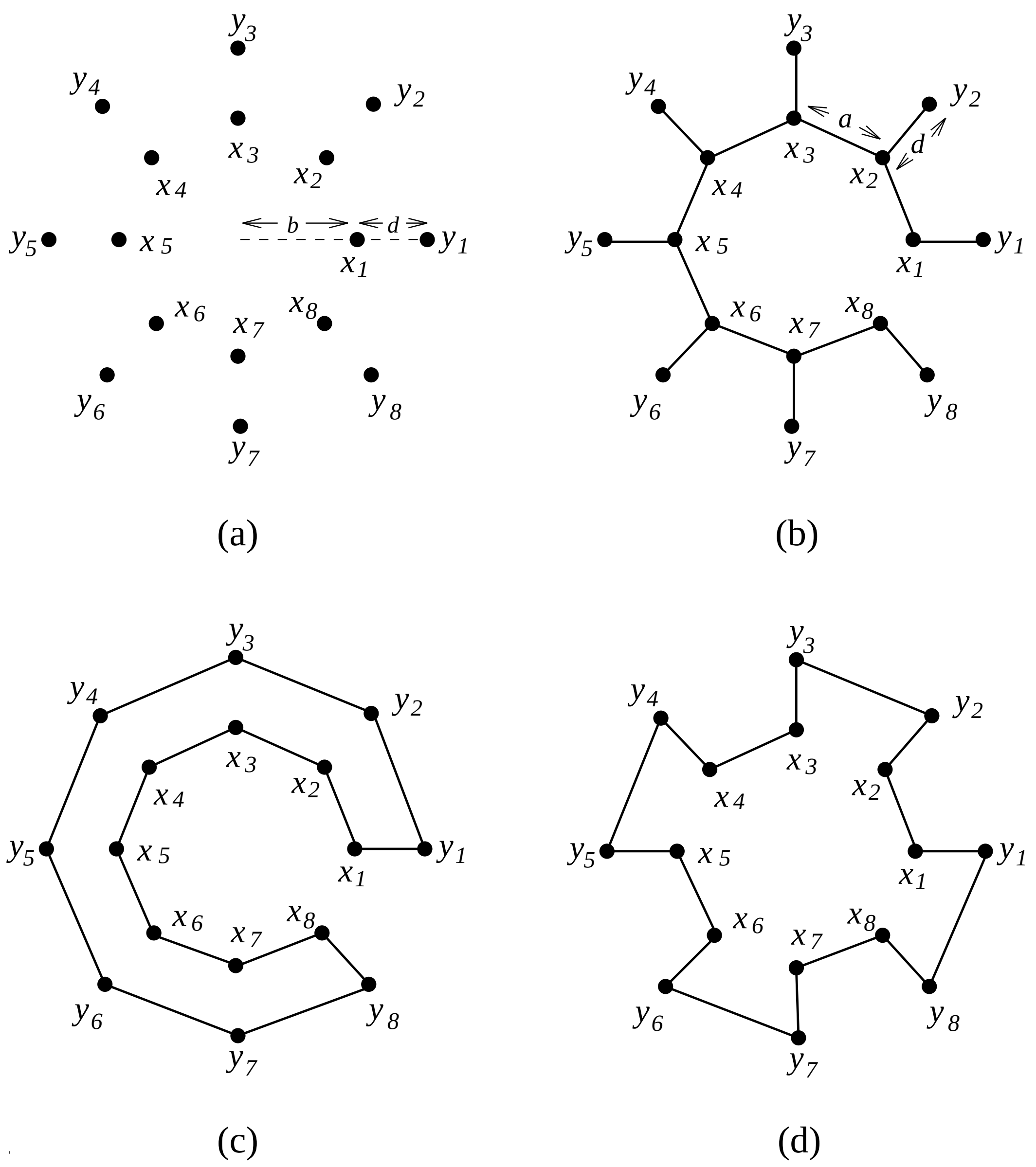
y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8
b
d
a
(a)
(b)
(c)
(d)

Figure 7: Δ-TSP Example.

CPSC-669 Computational Optimization

Lecture #22, October 18, 1995

Lecturer: Professor Jianer Chen
Scribe: Mitrajit Chatterjee
Revision: Jianer Chen

22 Δ-TSP: Christofides algorithm

In this lecture, we will allow a graph to have "multiple edges", i.e., each pair of vertices of a graph can be connected by more than one edge.

Let us reconsider the approximation algorithm `EasyTSP` for the Δ-TSP problem. As we pointed out, after the minimum spanning tree T is constructed, we traverse the tree T by a depth first search process in which each edge of T is traversed exactly twice. This process can be re-interpreted as follows:

1. construct a minimum spanning tree;
2. double each edge of T into two edges, each of which has the same weight as the original edge. Let the resulting graph be D;
3. make a closed walk W in the graph D such that each edge of D is traversed exactly once in W;
4. use "shortcuts", i.e., delete all but the first occurrences for each vertex in the walk W to make a traveling salesman tour L.

There are three crucial facts that make the above algorithm correctly produce a traveling salesman tour with approximation ratio 2: (1) the graph D gives a closed walk in the graph G and D contains all vertices of G; (2) the total weight of the graph D is bounded by 2 times the weight of an optimal traveling salesman tour; and (3) the shortcuts do not increase the weight of a closed walk so that we can derive a traveling salesman tour L from D without increasing the weight of the walk.

Therefore, if we can construct a better graph D_1 whose weight is smaller than the graph D constructed by the algorithm `EasyTSP` such that D_1 forms a closed walk of G and that D_1 contains all vertices of G, then using the shortcuts on D_1 should derive a better approximation to the minimum traveling salesman tour.

For this, we need introduce a definition.

Definition 22.1 An undirected connected graph G is an *Euler graph* if there is a closed walk in G that traverses each edge of G exactly once.

Recent research has shown that Euler graphs play an important role in designing efficient parallel graph algorithms.

Theorem 22.1 *An undirected connected graph G is an Euler graph if and only if every vertex of G has an even degree.*

PROOF. Suppose that G is an Euler graph. Let W be a closed walk in G that traverses each edge of G exactly once.

Let v be a vertex of G. Since W is a closed walk, each time W enters the vertex v from an edge, W must leave the vertex v by another edge incident on v. Therefore, each edge incident on v that is an "incoming" edge for W must be paired with an edge incident on v that is an "outgoing" edge for W. Since W traverses each edge exactly once, we conclude that the number of edges incident on v, i.e., the degree of v, is even.

Conversely, suppose that all vertices of the graph G have even degree. We prove the theorem by induction on the number of edges in G. The minimum such a graph G in which all vertices have even degree consists of two vertices connected by two (multiple) edges. This graph is clearly an Euler graph.

Now suppose that G has more than two edges. Let v_0 be any vertex of G. We construct a maximal walk W_0 starting from the vertex v_0. That is, starting from v_0, on each vertex if there is an unused edge, then we extend W_0 along that edge (if there are more than one such edge, we pick any one). The process stops when we hit a vertex u on which there is no unused incident edge. We claim that the ending vertex u must be the starting vertex v_0. In fact, for each interior vertex w in the walk W_0, each time W_0 passes through, W_0 uses one edge to enter w and uses another edge to leave w. Therefore, if the process stops at u and $u \neq v_0$, then the walk W_0 has only used an odd number of edges incident on u. This contradicts our assumption that the vertex u is of even degree. This proves the claim. Consequently, the walk W_0 is a closed walk.

The closed walk W_0 can also be regarded as a graph. By the definition, the graph W_0 itself is an Euler graph. According to the proof for the first part of this theorem, all vertices of the graph W_0 have even degree. Now removing all edges in the walk W_0 from the graph G results in a graph $G_0 = G - W_0$. The graph G_0 may not be connected. However, all vertices of G_0 must have an even degree because each vertex of the graphs G and W_0 has an even degree.

Let C_1, C_2, …, C_h be the connected components of the graph G_0. By the inductive hypothesis, each connected component C_i is an Euler graph. Let W_i be a closed walk in C_i that traverses each edge of C_i exactly once, for $i = 1, \ldots, h$. Moreover, for each i, the closed walk W_0 contains at least one vertex v_i in the connected component C_i (if W_0 does not contain any vertex from C_i, then the vertices of C_i have no connection to the vertices in the walk W_0 in the original graph G, this contradicts the assumption that the graph G is connected).

Therefore, it is easy to insert each closed walk W_i into the closed walk W_0 (by replacing any vertex occurrence of v_i in W_0 by the list W_i, where W_i is given by beginning and ending with v_i), for all $i = 1, \ldots, h$. This forms a closed walk W for the original graph G such that the walk W traverses each edge of G exactly once. Thus, the graph G is an Euler graph. □

The proof of Theorem 22.1 suggests an algorithm that constructs a closed walk W for an Euler graph G such that the walk W traverses each edge of G exactly once. This walk will be called an *Euler tour*. By a careful implementation, one can make this algorithm run in linear time. We leave the detailed implementation to the reader. Instead, we state this result without a proof as follows.

Theorem 22.2 *There is an algorithm that, given an Euler graph, constructs an Euler tour in linear time.*

Now we are ready to show how a better Euler graph D_1 can be constructed based on a minimum spanning tree, from which a better approximation for the minimum traveling salesman tour can be derived.

Let G be an input instance of the Δ-TSP problem and let T be a minimum spanning tree in G. We have

Lemma 22.3 *The number of vertices of the tree T that has an odd degree in T is even.*

PROOF. Let v_1, …, v_n be the vertices of the tree T. Since each edge $e = (v_i, v_j)$ of T contributes one degree to v_i and one degree to v_j, we must have

$$\sum_{i=1}^{n} deg_T(v_i) = 2(n-1)$$

where $deg_T(v_i)$ is the degree of the vertex v_i in the tree T. Note that $n-1$ is the number of edges in the tree T. We partition the set of vertices of T into odd degree vertices and even degree vertices. Then we have

$$\sum_{v_i\text{: even degree}} deg_T(v_i) + \sum_{v_j\text{: odd degree}} deg_T(v_j) = 2(n-1)$$

Since both $\sum_{v_i\text{: even degree}} deg_T(v_i)$ and $2(n-1)$ are even numbers, the value $\sum_{v_j\text{: odd degree}} deg_T(v_j)$ is also an even number. Consequently, the number of vertices that have odd degree in T must be even. $\square$

By Lemma 22.3, we can suppose, without loss of generality, that v_1, v_2, $\ldots$, v_{2h} be the odd degree vertices in the tree T. The vertices v_1, v_2, $\ldots$, v_{2h} induce a complete subgraph H in the original graph G. Now construct a minimum weight complete matching E_h in H. The matching E_h consists of h edges such that each of the vertices v_1, v_2, $\ldots$, v_{2h} is incident on exactly one edge in E_h. Thus, adding the edges in E_h to the tree T results in a graph $D_1 = T + E_h$ in which all vertices have an even degree. By Theorem 22.1, the graph D_1 is an Euler graph. Moreover, the graph D_1 contains all vertices of the graph G. We are now able to derive a traveling salesman tour L_1 from D_1 by using shortcuts.

We formally present this in the following algorithm. The algorithm is due to N. Christofides.

Algorithm 22.1 `Christofides`

```
Input:  an input instance $G$ of $\Delta$-TSP
Output:  a traveling salesman tour $L$

1.  construct a minimum spanning tree $T$ for $G$;
2.  let $v_1$, ..., $v_{2h}$ be the odd degree vertices in $T$,
     construct a minimum weight matching $E_h$ in the
     complete graph induced by $v_1$, ..., $v_{2h}$;
3.  construct an Euler tour $W_1$ in the Euler graph
     $D_1 = T + E_h$;
4.  use shortcuts to derive a traveling salesman tour $L_1$
     from $W_1$;
5.  return $L_1$.
```

It is known that a minimum weight matching can be constructed in time $O(n^3)$ (see lecture notes 7-10 for discussion on graph matchings). The other steps of the algorithm `Christofides` clearly take time $O(n^3)$. Therefore, the algorithm `Christofides` runs in time $O(n^3)$.

Now let us study the approximation ratio for the algorithm `Christofides`.

Lemma 22.4 *The weight of the minimum weight matching E_h on v_1, v_2, …, v_{2h}, $\sum_{e \in E_h}$ weight(e), is at most $1/2$ of the weight of an optimal traveling salesman tour in the graph G.*

PROOF. Let L be an optimal traveling salesman tour in the graph G. By using shortcuts, i.e., by removing the vertices that are not in $\{v_1, v_2, \ldots, v_{2h}\}$ from the tour L, we obtain a simple cycle L' that contains exactly the vertices v_1, …, v_{2h}. Since G satisfies the triangle inequality, the weight of L' is not larger than the weight of L.

Moreover, the simple cycle L' can be decomposed into two disjoint matchings of $\{v_1, \ldots, v_{2h}\}$ — one matching is obtained by taking every other edge in the cycle L, and the other matching is formed by the rest of the edges. Of course, both of these two matchings have weight at least as large as the minimum weight matching E_h on $\{v_1, \ldots, v_{2h}\}$. This gives

$$\text{weight}(L) \geq \text{weight}(L') \geq 2 \cdot \text{weight}(E_h)$$

This completes the proof. $\square$

Now the analysis is clear. We have $D_1 = T + E_h$. Thus

$$\text{weight}(D_1) = \text{weight}(T) + \text{weight}(E_h)$$

By the analysis for the algorithm `EasyTSP` (Algorithm 21.1), the weight of T is not larger than the weight of an optimal traveling salesman tour for G. Combining this with Lemma 22.4, we conclude that the weight of the graph D_1 is bounded by 1.5 times the weight of an optimal traveling salesman tour in G. Moreover, the traveling salesman tour L_1 constructed by the algorithm `Christofides` is obtained by using shortcuts on the graph D_1 and thus has weight not larger than the weight of D_1. We close this lecture with the following theorem.

Theorem 22.5 *The algorithm* `Christofides` *for the Δ-TSP problem runs in time $O(n^3)$ and has approximation ratio* 1.5.

As for the algorithm `EasyTSP`, one can show that the ratio 1.5 is tight for the algorithm `Christofides` in the sense that there are input instances of Δ-TSP for which the algorithm `Christofides` produces traveling salesman tours with approximation ratio arbitrarily close to 1.5.

CPSC-669 Computational Optimization

Lecture #23, October 20, 1995

Lecturer: Professor Jianer Chen
Scribe: Weijie Zhang
Revision: Jianer Chen

23 Bin Packing problem

In the previous lectures, we have presented several approximation algorithms for NP-hard optimization problems. In this lecture, we study approximation algorithms for the BIN PACKING problem. Recall that the BIN PACKING problem is defined as

> BIN PACKING
>
> INPUT: $\langle t_1, t_2, \ldots, t_n; B\rangle$, all integers and $t_i \leq B$ for all i
>
> OUTPUT: a packing of the n objects of size t_1, …, t_n into the minimum number of bins of size B

Since the number of bins used by any packing cannot be larger than the number of objects in the input, according to Theorem 17.1, the BIN PACKING problem has no fully polynomial time approximation scheme. On the other hand, it is fairly easy to design a polynomial time approximation algorithm for the BIN PACKING problem with a reasonably good approximation ratio. Consider the following simple approximation algorithm for the BIN PACKING problem.

Algorithm 23.1 `First-Fit (FF)`

```
Input:  $I = \langle t_1, t_2, \cdots, t_n; B\rangle$
Output:  a packing of the $n$ objects into bins of size $B$

1.  for $i = 1$ to $n$ do
2.      $j = 1$
3.      notput = true;
4.      while notput do
5.          if object $i$ can be put in bin $j$
6.          then put $i$ in bin $j$; notput = false;
7.          else $j = j + 1$
```

The **for** loop in the algorithm is executed n times, and in each execution, the **while** loop can be done in $O(n)$ time since $t_i \leq B$ for all i. This concludes that the algorithm `First-Fit` runs in time $O(n^2)$. What is the approximation ratio for the algorithm?

Theorem 23.1 *The algorithm* `First-Fit` *has approximation ratio 2.*

PROOF. We observe that there is at most one used bin whose content is not larger than $B/2$. In fact, suppose that there are two used bins B_i and B_j whose contents are bounded by $B/2$. Without loss of generality, let $i < j$. Then the algorithm `First-Fit` would have put the objects in the bin B_j into the bin B_i since the bin B_i has enough room for them and the bin B_i is considered before the bin B_j by the algorithm `First-Fit`.

Now the theorem can be proved in two cases.

Suppose that the contents of all used bins are not less than $B/2$. Let m be the number of bins used by the algorithm FIRST-FIT. We have

$$\sum_{i=1}^{n} t_i \geq \frac{mB}{2}$$

Since the bin size is B, we need at least

$$\lceil (\sum_{i=1}^{n} t_i)/B \rceil \geq \lceil (mB)/(2B) \rceil \geq m/2$$

bins to pack the n objects, i.e., $Opt(I) \geq m/2$. Therefore, the approximation ratio is bounded in this case by

$$\frac{m}{Opt(I)} \leq \frac{m}{m/2} = 2$$

Now suppose that there is a used bin whose content x is less than $B/2$. Again let m be the number of bins used by the algorithm `First-Fit`. Therefore, there are $m - 1$ bins with contents at least $B/2$. This gives us

$$\sum_{i=1}^{n} t_i \geq \frac{(m-1)B}{2} + x > \frac{(m-1)B}{2}$$

Thus, $\lceil (\sum_{i=1}^{n} t_i)/B \rceil > (m-1)/2$

If $m-1$ is an even number, then since both $\lceil(\sum_{i=1}^n t_i)/B\rceil$ and $(m-1)/2$ are integers, we get

$$\lceil(\sum_{i=1}^n t_i)/B\rceil \geq (m-1)/2+1 > m/2$$

If $m-1$ is an odd number, then

$$\lceil(\sum_{i=1}^n t_i)/B\rceil \geq \lceil(m-1)/2\rceil = (m-1)/2+1/2 = m/2$$

Note that any packing should use at least $\lceil(\sum_{i=1}^n t_i)/B\rceil$ bins. In particular,

$$Opt(I) \geq \lceil(\sum_{i=1}^n t_i)/B\rceil$$

The above analysis shows that the approximation ratio is bounded by

$$\frac{m}{Opt(I)} \leq \frac{m}{\lceil(\sum_{i=1}^n t_i)/B\rceil} \leq \frac{m}{m/2} = 2$$

This proves the theorem. □

Therefore, the Bin Packing problem can be approximated in polynomial time with approximation ratio 2. Can we do better than 2? In particular, does the Bin Packing problem have a polynomial time approximation scheme? A negative answer to this question can be easily derived, as shown in the following theorem.

Theorem 23.2 *There is no polynomial time approximation algorithm for the* Bin Packing *problem with approximation ratio less than 1.5 unless* $P = NP$.

PROOF. Suppose that we have a polynomial time approximation algorithm A with approximation ratio less than 1.5 for the Bin Packing problem. We show how we can use this algorithm to solve in polynomial time the Partition problem, which is NP-complete.

Recall that Partition is a decision problem defined as follows.

Partition

Input: A set $\{x_1, x_2, \ldots, x_n\}$ of n integers

Question: Is there a subset $S' \subseteq S$ such that
$\sum_{i \in S'} x_i = \sum_{j \in S-S'} x_j$?

Given an input instance $X = \{t_1, t_2, \cdots, t_n\}$ for the PARTITION problem, if $\sum_{i=1}^n t_i$ is an odd number, then we know X is a NO-instance. Otherwise, let $B = (\sum_{i=1}^n t_i)/2$, and let $g(X) = \langle t_1, t_2, \cdots, t_n; B\rangle$ be an instance for the problem BIN PACKING. Now apply the approximation algorithm A for the BIN PACKING problem on the input $g(X)$. Suppose that the approximation algorithm A uses m bins for this input instance $g(X)$. There are two different cases.

If $m \geq 3$ bins, then since we have

$$m/Opt(g(X)) < 1.5$$

we get $Opt(g(X)) > 2$. That is, the objects t_1, …, t_n cannot be packed into two bins of size $B = (\sum_{i=1}^n t_i)/2$. Consequently, the instance $X = \{t_1, t_2, \cdots, t_n\}$ is a NO-instance for the PARTITION problem.

On the other hand, if $m \leq 2$, then we must have $m = 2$. Thus, the objects t_1, …, t_h can be evenly split into two sets of equal size. That is, the instance $X = \{t_1, t_2, \cdots, t_n\}$ is a YES-instance for the PARTITION problem.

Therefore, the instance X is a YES-instance for the PARTITION problem if and only if the approximation algorithm A uses two bins to pack the instance $g(X)$. Since by our assumption, the approximation algorithm A runs in polynomial time, we conclude that the PARTITION problem can be solved in polynomial time.

Since the PARTITION problem is NP-complete, this implies P = NP. The theorem is proved. $\square$

We observe that the 1.5 lower bound on approximation ratio for the BIN PACKING problem occurs when the optimal value $Opt(X)$ is very small. Similar lower bounds on approximation ratio can be derived for optimization problems that remain NP-hard even when the optimal value is very small. Examples include GRAPH COLORING and GRAPH EDGE COLORING problems.

In some cases, we may be interested in the *asymptotic lower bounds* on approximation ratio of an optimization problem. For instance, we may want to ask whether the 1.5 lower bound can still be achieved when the optimal value is sufficiently large for a input instance for the BIN PACKING problem. This question is closely related to the concept of *asymptotic approximation scheme* defined as follows.

Definition 23.1 An optimization problem $Q = \langle I, S, f, opt\rangle$ has a *asymptotic polynomial time approximation scheme* (APTAS) if for any fixed constant $\epsilon > 0$, there is a constant c_ϵ and a polynomial time approximation algorithm A_ϵ for Q such that for all input instances $x \in I$ with $Opt(x) \geq c_\epsilon$, the algorithm A_ϵ produces a solution for x with approximation ratio bounded by $1 + \epsilon$.

We will show that the Bin Packing problem has an asymptotic polynomial time approximation scheme.

Let us start with a restricted version of the Bin Packing problem, which will be called the (δ, π)-Bin Packing problem. There are two restrictions. First, we assume that the input objects have at most a constant number π of different sizes. Second, we assume that the size of each input object is at least as large as a δ factor of the bin size. The following is a formal definition.

> (δ, π)-Bin Packing
>
> Input: $\langle t_1 : n_1, t_2 : n_2, \ldots, t_\pi : n_\pi; B\rangle$, where $\delta B \leq t_i \leq B$ for all i, interpreted as: for the $n = \sum_{i=1}^{\pi} n_i$ input objects, n_i of them are of size t_i, for $i = 1, \ldots, \pi$
>
> Output: a packing of the n objects into the minimum number of bins of size B

We first study the properties of the (δ, π)-Bin Packing problem. Let $I = \langle t_1 : n_1, \ldots, t_\pi : n_\pi; B\rangle$ be an input instance for the (δ, π)-Bin Packing problem. Suppose that an optimal packing packs the objects in I into m bins B_1, B_2, ..., B_m. Consider the first bin B_1. Suppose that the bin B_1 contains b_1 objects of size t_1, b_2 objects of size t_2, ..., and b_π objects of size t_π. We then call

$$(b_1, b_2, \ldots, b_\pi)$$

the *configuration* of the bin B_1. Since each object has size at least δB and the bin size is B, the bin B_1 contains at most $1/\delta$ objects. In particular, we have $b_i \leq 1/\delta$ for all i. Therefore, the total number of different bin configurations is bounded by $(1/\delta)^\pi$.

Now consider the set I' of objects that is obtained from the set I with all objects packed in the bin B_1 removed. The set I' can be written as

$$I' = \langle t_1 : (n_1 - b_1), t_2 : (n_2 - b_2), \ldots, t_\pi : (n_\pi - b_\pi); B\rangle$$

Note that I' is also an input instance for the (δ, π)-Bin Packing problem and the packing $(B_2, B_3, \ldots, B_m)$ is an optimal packing for I' (I' cannot be packed into less than $m - 1$ bins otherwise the set I can be packed into less than m bins). Therefore, if we can pack the set I' into a minimum number of bins then an optimal packing for the set I can be obtained by packing the rest of the objects into a single bin B_1.

Now the problem is that we do not know the configuration for the bin B_1. Therefore, we will try all possible configurations for a single bin, and recursively find an optimal packing for the rest of the objects. As pointed out above, the number of bin configurations is bounded by $(1/\delta)^\pi$, which is a constant when both δ and π are fixed. In the next lecture, we will present a dynamic programming algorithm that constructs an optimal packing for an input instance for the (δ, π)-Bin Packing problem.

CPSC-669 Computational Optimization

Lecture #24, October 23, 1995

Lecturer: Professor Jianer Chen
Scribe: Weijie Zhang
Revision: Jianer Chen

24 The (δ, π)-Bin Packing problem

Recall that the (δ, π)-Bin Packing problem is defined as follows:

> (δ, π)-Bin Packing
>
> Input: $\langle t_1 : n_1, t_2 : n_2, \ldots, t_\pi : n_\pi; B\rangle$, where $\delta B \leq t_i \leq B$ for all i, interpreted as: for the $n = \sum_{i=1}^{\pi} n_i$ input objects, n_i of them are of size t_i, for $i = 1, \ldots, \pi$
>
> Output: a packing of the n objects into the minimum number of bins of size B

Fix an input instance $I = \langle t_1 : n_1, \ldots, t_\pi : n_\pi; B\rangle$ of the (δ, π)-Bin Packing problem. Each subset of objects in I can be written as a π-tuple $[h_1, \ldots, h_\pi]$ with $h_i \leq n_i$ to specify that the subset contains h_i objects of size t_i for all i. In particular, the input instance I itself can be written as $[n_1, \ldots, n_\pi]$.

Let $\#H[h_1, \ldots, h_\pi]$ denote the minimum number of bins needed to pack the subset $[h_1, \ldots, h_\pi]$ of the input instance I for the (δ, π)-Bin Packing problem. Suppose that $\#H[h_1, \ldots, h_\pi] \geq 1$. According to the discussion in the last lecture, we know that $\#H[h_1, \ldots, h_\pi]$ is equal to 1 plus $\#H[h_1 - b_1, \ldots, h_\pi - b_\pi]$ for some bin configuration $(b_1, b_2, \ldots, b_\pi)$. On the other hand, since $\#H[h_1, \ldots, h_\pi]$ corresponds to an optimal packing of the subset $[h_1, \ldots, h_\pi]$, $\#H[h_1, \ldots, h_\pi]$ is actually equal to 1 plus the minimum of $\#H[h_1 - b_1, \ldots, h_\pi - b_\pi]$ over all consistent bin configurations $(b_1, \ldots, b_\pi)$. This suggests an algorithm that uses the dynamic programming technique to compute the value of $\#H[h_1, \ldots, h_\pi]$. In particular, $\#H[n_1, \ldots, n_\pi]$ gives the optimal value for the input instance I for the (δ, π)-Bin Packing problem.

Definition 24.1 Fix an input instance $I = \langle t_1 : n_1, \ldots, t_\pi : n_\pi; B\rangle$ for the (δ, π)-Bin Packing problem. Let $I' = [h_1, \ldots, h_\pi]$ be a subset of the input objects in I, where $h_i \leq n_i$ for all i. A π-tuple $(b_1, \ldots, b_\pi)$ is an *addable bin configuration to I'* if

1. $h_i + b_i \leq n_i$ for all $i = 1, \ldots, \pi$; and
2. $\sum_{i=1}^{\pi} t_i b_i \leq B$.

Intuitively, an addable bin configuration specifies a bin configuration that can be obtained using the objects in I that are not in the subset I'.

Now we are ready for presenting the following dynamic programming algorithm. We use a π-dimensional array $H[1..n_1, \ldots, 1..n_\pi]$ (note that π is a fixed constant) such that $H[i_1, \ldots, i_\pi]$ records an optimal packing for the subset $[i_1, \ldots, i_\pi]$ of I. We use the notation $\#H[i_1, \ldots, i_\pi]$ to denote the number of bins used in the packing $H[i_1, \ldots, i_\pi]$. For a packing $H[i_1, \ldots, i_\pi]$ and a bin configuration $(b_1, \ldots, b_\pi)$, we will use

$$H[i_1, \ldots, i_\pi] \oplus (b_1, \ldots, b_\pi)$$

to represent the packing for the subset $[i_1 + b_1, \ldots, i_\pi + b_\pi]$ that is obtained from $H[i_1, \ldots, i_\pi]$ by adding a new bin with configuration $(b_1, \ldots, b_\pi)$.

Algorithm 24.1 (δ, π)-Precise

```
INPUT:  $I = \langle t_1 : n_1, \ldots, t_\pi : n_\pi; B \rangle$, where $t_i \geq \delta B$ for all $i$
OUTPUT:  a bin packing of $I$ using minimum number of bins.

1.  $\#H[i_1, \ldots, i_\pi] = +\infty$ for all $0 \leq i_j \leq n_j$, $1 \leq j \leq \pi$;
2.  $H[0, \ldots, 0] = \phi$;     $\#H[0, \ldots, 0] = 0$;
3.  for $i_1 = 0$ to $n_1$ do
4.     for $i_2 = 0$ to $n_2$ do
          ⋮
5.        for $i_\pi = 0$ to $n_\pi$ do
6.           for each bin configuration $(b_1, \ldots, b_\pi)$
                   addable to the subset $[i_1, \ldots, i_\pi]$   do
7.              if  $\#H[i_1 + b_1, \ldots, i_\pi + b_\pi] > 1 + \#H[i_1, \ldots, i_\pi]$
8.              then
                  $H[i_1 + b_1, \ldots, i_\pi + b_\pi] = H[i_1, \ldots, i_\pi] \oplus (b_1, \ldots, b_\pi)$;
                  $\#H[i_1 + b_1, \ldots, i_\pi + b_\pi] = \#H[i_1, \ldots, i_\pi] + 1$
```

Steps 7-8 can obviously be done in time $O(n)$. Since $b_i \leq 1/\delta$ for all $i = 1, \ldots \pi$, there are at most $(1/\delta)^\pi$ addable bin configurations for each subset $[i_1, \ldots, i_\pi]$. Moreover, $n_i \leq n$ for all $i = 1, \ldots, \pi$. Therefore, steps 7-8 can be executed at most $n^\pi (1/\delta)^\pi$ times. We conclude that the running time of the algorithm (δ, π)-`Precise` is bounded by $O(n^{\pi+1}(1/\delta)^{\pi})$, which is a polynomial of n when δ and π are fixed.

The algorithm (δ, π)-`Precise` is not very satisfying. In particular, even for a moderate constant π of different sizes, the factor $n^{\pi+1}$ in the complexity makes the algorithm not practically useful. On the other hand, we will see that our approximation algorithm for the general Bin Packing problem is based on solving the (δ, π)-Bin Packing problem with a very large constant π and a very small constant δ. Therefore, we need, if possible, to improve the above time complexity. In particular, we would like to see if there is an algorithm that solves the (δ, π)-Bin Packing problem such that in the time complexity of the algorithm, the exponent of n is independent of the values of π and δ.

Fix an input instance $I = \langle t_1 : n_1, \ldots, t_\pi : n_\pi; B\rangle$ for the (δ, π)-Bin Packing problem. We say that a π-tuple $(b_1, \ldots, b_\pi)$ is a *feasible bin configuration* if $b_i \leq n_i$ for all i and $t_1 b_1 + \cdots t_\pi b_\pi \leq B$. Since $t_i \geq \delta B$ for all i, we get $b_i \leq 1/\delta$ for all i. Therefore, there are totally at most $(1/\delta)^\pi$ feasible bin configurations. Let all feasible bin configurations be

$$\begin{array}{rcl} T_1 & = & (b_{11}, b_{12}, \cdots, b_{1\pi}) \\ T_2 & = & (b_{21}, b_{22}, \cdots, b_{2\pi}) \\ & \vdots & \\ T_q & = & (b_{q1}, b_{q2}, \cdots, b_{q\pi}) \end{array} \tag{8}$$

where $q \leq (1/\delta)^\pi$. Note that the above list of feasible bin configurations can be constructed in time independent of the number $n = \sum_{i=1}^{\pi} n_i$ of objects in the input instance I. Now each bin packing P of the input instance I can be written as a q-tuple $\langle x_1, x_2, \ldots, x_q\rangle$, where x_j is the number of bins of bin configuration T_j used in the packing P. Moreover, there is essentially only one bin packing that corresponds to the q-tuple $\langle x_1, x_2, \ldots, x_q\rangle$, if we ignore the ordering of the bins used. An optimal packing corresponds to a q-tuple $\langle x_1, x_2, \ldots, x_q\rangle$ with $x_1 + \cdots x_q$ minimized.

Conversely, in order to let a q-tuple $\langle x_1, x_2, \ldots, x_q\rangle$ to describe a real pin packing, we need to make sure that the q-tuple uses exactly the input objects given in I. For each feasible bin configuration T_j, there are b_{jh} objects of size t_h. Therefore, if x_j bins are of bin configuration T_j, then for the bin configuration T_j, the q-tuple assumes $x_j b_{jh}$ objects of size t_h. Now adding these over all bin configurations, we conclude that the total number of objects of size t_h assumed by the q-tuple $\langle x_1, x_2, \ldots, x_q \rangle$ is

$$x_1 b_{1h} + x_2 b_{2h} + \cdots + x_q b_{qh}$$

This should match the number n_h of objects of size t_h in the input instance I. This formulates the conditions into the following linear programming problem.

$$\begin{array}{rcl} \min \quad x_1 + x_2 + \cdots + x_q & & \\ x_1 b_{11} + x_2 b_{21} + \cdots + x_q b_{q1} & = & n_1 \\ x_1 b_{12} + x_2 b_{22} + \cdots + x_q b_{q2} & = & n_2 \\ & \vdots & \\ x_1 b_{1\pi} + x_2 b_{2\pi} + \cdots + x_q b_{q\pi} & = & n_\pi \\ x_i \geq 0, \text{ for } i = 1, \ldots, q & & \end{array} \tag{9}$$

Since all x_is must be integers, this is an integer linear programming problem. It is easy to see that if a q-tuple $\langle x_1, \ldots, x_q \rangle$ corresponds to a valid bin packing of the input instance I, then the vector $(x_1, \ldots, x_q)$ satisfies the constraints in the system (9). Conversely, any vector $(x_1, \ldots, x_q)$ satisfying the constraints in the system (9) describes a valid bin packing for the input instance I. Moreover, it is easy to see that if a vector $(x_1, \ldots, x_q)$ satisfying the constraints in the system (9) is given, the corresponding bin packing can be constructed in linear time.

Therefore, to construct an optimal solution for the input instance I for the (δ, π)-Bin Packing problem, we only need to construct an optimal solution for the integer linear programming system (9). As we discussed before, the Integer Linear Programming problem in general is NP-hard. But here the nice thing is that both the number q of variables and the number $q+\pi$ of constraints in the system (9) are independent of $n = \sum_{i=1}^{\pi} n_i$. However, this does not immediately imply that the system can be solved in time independent of n — the numbers n_i appearing on the right side of the system may be as large as n.

Anyway, the above system has at least suggested a polynomial time algorithm for solving the problem: we know that an optimal solution must satisfy $x_1 + \cdots + x_q \leq n$. Thus, $0 \leq x_i \leq n$ for all $i = 1, \ldots, q$ in an optimal solution. Therefore, we could enumerate all vectors $(x_1, \ldots, x_q)$ satisfying $0 \leq x_i \leq n$ and solve the system (9). Note that there are totally $(n+1)^q$ such vectors and q is independent of n. However, since q has order $(1/\delta)^\pi$, this enumerating algorithm gives a polynomial time algorithm whose complexity is even worse than that of the algorithm (δ, π)-`Precise`.

Fortunately, Lenstra in 1983 has described an algorithm that solves the system (9) in time $h(q, \pi)$, where $h(q, \pi)$ is a function depending only on q and π. Since the algorithm involves complicated analysis on integer linear programming, we omit the description of the algorithm.

We summarize the above discussion.

Algorithm 24.2 (δ, π)-`Precise2`

```
INPUT:   $I = \langle t_1 : n_1, \ldots, t_\pi : n_\pi; B\rangle$, where $t_i \geq \delta B$ for all $i$
OUTPUT:  a bin packing of $I$ using the minimum number of bins.

1.  construct the list (8) of all feasible configurations
                        $T_1, T_2, \ldots, T_q$;
2.  solve the system (9) using Lenstra's algorithm;
3.  return the solution $\langle x_1, \ldots, x_q\rangle$ of step 2.
```

Algorithm (δ, π)-`Precise2`, as discussed above, runs in time $h_1(q, \pi) = h_2(\pi, \delta)$, where h_2 is a function depending only on δ and π. This may seem a bit surprising since the algorithm packs $n = \sum_{i=1}^{\pi} n_i$ objects in time independent of n! This is really a matter of coding. Note that the input $I = \langle t_1 : n_1, \ldots, t_\pi : n_\pi; B\rangle$ of the algorithm (δ, π)-`Precise2` actually consists of $2\pi + 1$ integers, and the solution $\langle x_1, \ldots, x_q\rangle$ given by the algorithm consists of $q = (1/\epsilon)^\pi$ integers. To convert the vector $\langle x_1, \ldots, x_q\rangle$ into an actual packing of the $n = \sum_{i=1}^{\pi} n_i$ input objects, an extra step of time $O(n)$ should be added.

Theorem 24.1 *The* (δ, π)-BIN PACKING *problem can be solved in time* $O(n) + h(\delta, \pi)$, *where* $h(\delta, \pi)$ *is a function independent of* n.

CPSC-669 Computational Optimization

Lecture #25, October 25, 1995

Lecturer: Professor Jianer Chen
Scribe: Weijie Zhang
Revision: Jianer Chen

25 Approximating Bin Packing

In the last lecture, we have shown that the (δ, π)-Bin Packing problem can be solved in time $O(n) + h(\delta, \pi)$, where $h(\delta, \pi)$ is a function independent of n.

In today's lecture, we use the solution for the (δ, π)-Bin Packing problem to develop an approximation algorithm for the general Bin Packing problem. Let us first roughly describe the basic idea of the approximation algorithm.

An input instance of the general Bin Packing problem may contain objects of small size and objects of many different sizes. To convert an input instance $I = \langle t_1, \ldots, t_n; B\rangle$ of the general Bin Packing problem to an input instance of the (δ, π)-Bin Packing problem, we perform two preprocessing steps:

1. ignore the objects of small size, i.e., the objects of size less than δB; and

2. sort the rest of the objects by their sizes in decreasing order, then partition the sorted list into π groups G_1, ..., G_π. For each group G_i, replace every object by the one with the largest size t'_i in G_i.

After the preprocessing steps, we obtain an instance $I' = \langle t'_1 : m, \ldots, t'_\pi : m; B\rangle$ of the (δ, π)-Bin Packing problem, where $m \le n/\pi$. Now we use the algorithm we have described to construct a solution Y', which is a packing, for the instance I'. To obtain a solution Y to the original input instance I of the Bin Packing problem, we first replace each object in Y' by the corresponding object in I, then add the objects in I that have size smaller than δB using greedy method.

The intuition is that an optimal solution to I' is an over-estimation of the optimal solution to I (since each object in I is replaced by a larger object in I', the number of bins used by an optimal packing of I' is at least as large as the number of bins used by an optimal packing of I); while an optimal solution to $I'' = \langle t'_2 : m, \ldots, t'_\pi : m; B\rangle$ is an under-estimation of the optimal solution to I (I'' can be regarded as an instance obtained by replacing each object of I in the group G_i by a smaller object of size t'_{i+1} for $1 \leq i \leq \pi - 1$ and deleting the objects in the last group G_π). Since the instance I'' can also be obtained by deleting the m largest objects in I', an optimal packing of I' uses at most m more bins than an optimal packing of I'' (the m bins are used to pack the m largest objects in I'). Therefore, an optimal packing of I' uses at most m more bins than an optimal packing of I, with the objects of size less than δB ignored. When the value π is sufficiently large, the value $m = n/\pi$ is small so that an optimal solution to I' will be a good approximation to the optimal solution to I with objects of size less than δB ignored.

Finally, after a good approximation of the optimal solution to the instance I minus the small objects is obtained, we add the small objects to this solution using greedy method. Since the small objects have small size, the greedy method will not leave much room in each bin. Thus, the resulting packing will be a good approximation for the input instance I of the general Bin Packing problem.

We present the formal algorithm and formal analysis as follows.

Algorithm 25.1 `ApprxBinPacking`

```
Input:  $I = \langle t_1, \ldots, t_n; B\rangle$ and $\epsilon > 0$
Output:  a packing of the objects in $I$

1.  sort $t_1, \ldots, t_n$; without loss of generality, let
                 $t_1 \geq t_2 \geq \cdots \geq t_n$
2.  let $h$ be the largest index such that $t_h \geq \epsilon B/2$; let
                 $I_0 = \langle t_1, t_2, \ldots, t_h; B\rangle$
3.  let $\pi = \lceil 4/\epsilon^2 \rceil$, partition the objects in $I_0$ into $\pi$
     groups $G_1$, ..., $G_\pi$, such that the group $G_i$ consists
     of the objects
                 $t_{(m-1)i+1}, t_{(m-1)i+2}, \ldots, t_{mi}$
     where $m = \lceil h/\pi \rceil$ (the last group $G_\pi$ contains $m' \leq m$
     objects).
4.  construct an optimal solution $Y'$ to the instance
      $I' = \langle t_1 : m, t_{m+1} : m, t_{2m+1} : m, \ldots, t_{(\pi-1)m+1} : m'; B\rangle$
     for the $(\epsilon/2, \pi)$-Bin Packing problem;
5.  replace each object in $Y'$ of size $t_{jm+1}$ by a proper
```

```
     object in the group $G_{j+1}$ of $I_0$, for $j = 0, \ldots, \pi - 1$,
     to construct a packing $Y_0$ for the instance $I_0$;
6.   add the objects $t_{h+1}, \ldots, t_n$ in $I$ to the packing $Y_0$ by
     greedy method (i.e., no new bin will be used until no
     used bin has enough space for the current object).
     This results in a packing for the instance $I$.
```

According to Theorem 24.1 and note that $\pi = \lceil 4/\epsilon \rceil$, the $(\epsilon/2, \pi)$-Bin Packing problem can be solved in time $O(n) + h(\epsilon/2, \pi) = O(n) + h_0(\epsilon)$, where $h_0(\epsilon)$ is a function depending only on ϵ, we conclude that the algorithm `ApprxBinPacking` runs in time $O(n \log n) + h_0(\epsilon)$, if an $O(n \log n)$ time sorting algorithm is used for step 1.

We discuss the approximation ratio for the algorithm `ApprxBinPacking`. As before, we denote by $Opt(I)$ the optimal value, i.e., the number of bins used by an optimal packing, of the input instance I of the Bin Packing problem.

Lemma 25.1 *Let I_0 be the input instance constructed by step 2 of the algorithm* `ApprxBinPacking`*. Then*

$$Opt(I_0) \le Opt(I)$$

PROOF. This is because I_0 is a subset of I so I takes at least as many bins as I_0. □

Lemma 25.2 *Let I_0 and I' be the input instances constructed by step 2 and step 4 of the algorithm* `ApprxBinPacking`*, respectively. Then*

$$Opt(I') \le Opt(I_0)(1 + \epsilon) + 1$$

PROOF. Note that the instance I' is obtained from the instance I_0 by replacing each object in a group G_i by the largest object $t_{(i-1)m+1}$ in the group. Therefore, an optimal packing for the instance I' uses at least as many bins as that used by an optimal packing for the instance I_0. This gives

$$Opt(I_0) \le Opt(I')$$

Now let

$$I'' = \langle t_{m+1} : m, t_{2m+1} : m, \ldots, t_{(\pi-1)m+1} : m'; B \rangle$$

I'' can be regarded as an instance obtained from I_0 by (1) replacing each object in the group G_i by a smaller object t_{im+1} (recall that t_{im+1} is the largest object in group G_{i+1}), for all $i = 1, \ldots, \pi - 2$; (2) replacing m' objects in group $G_{\pi-1}$ by a smaller object $t_{(\pi-1)m+1}$; and (3) eliminating rest of the objects in group $G_{\pi-1}$ and all objects in group G_π. Therefore, an optimal packing for I_0 uses at least as many bins as an optimal packing for I''. This gives

$$Opt(I'') \le Opt(I_0)$$

Finally, the difference between the instances I' and I'' are m objects of size t_1. Since an object can fit into a bin, we must have

$$Opt(I') \le Opt(I'') + m$$

Combining all these we obtain

$$Opt(I_0) \le Opt(I') \le Opt(I_0) + m$$

This gives us

$$\begin{aligned} Opt(I') &\le Opt(I_0) + m = Opt(I_0) + \lceil h/\pi \rceil \\ &\le Opt(I_0) + h/\pi + 1 = Opt(I_0) + h\epsilon^2/4 + 1 \end{aligned} \tag{10}$$

Now since each object of I_0 has size at least $\epsilon B/2$, each bin can hold at most $\lfloor 2/\epsilon \rfloor$ objects. Thus, the number of bins $Opt(I_0)$ used by an optimal packing for the instance I_0 is at least as large as $\epsilon h/2$:

$$\epsilon h/2 \le Opt(I_0)$$

Use this in Equation (10), we get

$$Opt(I') \le Opt(I_0) + \epsilon \cdot Opt(I_0)/2 + 1 \le Opt(I_0)(1+\epsilon) + 1$$

The lemma is proved. □

Lemma 25.3 *The solution Y_0 constructed by step 5 of* `ApprxBinPacking` *is a packing for the instance I_0. Moreover, the number of bins used by Y_0 is at most $Opt(I_0)(1+\epsilon)+1$.*

PROOF. First note that the instances I_0 and I' have the same number of objects. The solution Y_0 to I_0 is obtained from the optimal solution Y' to the instance I' by replacing each of the m objects of size $t_{(i-1)m+1}$ in I'

by an object in group G_i of I_0. Since no object in group G_i has size larger than $t_{(i-1)m+1}$, we actually replace objects in the bins in Y' by objects of the same or smaller size. Therefore, no bin would get content more than B in the packing Y_0. This shows that Y_0 is a packing for the instance I_0.

Finally, since Y_0 uses exactly the same number of bins as Y' and Y' is an optimal packing for I'. By Lemma 25.2, the number of bins used by Y_0, i.e., the number of bins $Opt(I')$ used by Y', is at most $Opt(I_0)(1+\epsilon)+1$. $\square$

Now we are ready for deriving our main theorem.

Theorem 25.4 *For any input instance $I = \langle t_1, \ldots, t_n; B\rangle$ of the* BIN PACKING *problem and for any $0 < \epsilon \leq 1$, the algorithm* `ApprxBinPacking` *constructs a bin packing of I that uses at most $Opt(I)(1+\epsilon)+1$ bins.*

PROOF. According to Lemma 25.3, the solution Y_0 constructed by step 5 of the algorithm `ApprxBinPacking` is a packing for the instance I_0. Now step 6 of the algorithm simply adds the objects in $I - I_0$ to Y_0 using greedy method. Therefore, the algorithm `ApprxBinPacking` constructs a packing for the input instance I. Let Y be the packing constructed by the algorithm `ApprxBinPacking` for I and let r be the number of bins used by Y. There are two cases.

If in step 6 of the algorithm `ApprxBinPacking`, no new bin is introduced. Then r equals the number of bins used by Y_0. According to Lemma 25.3 and Lemma 25.1, we get

$$r \leq Opt(I_0)(1+\epsilon)+1 \leq Opt(I)(1+\epsilon)+1$$

and the theorem is proved.

Thus, we assume that in step 6 of the algorithm `ApprxBinPacking`, new bins are introduced. According to our greedy strategy, no new bin is introduced unless no used bin has enough room for the current object. Since all objects added by step 6 have size less than $\epsilon/2$, we conclude that all of the r bins in Y, except maybe one, have content larger than $B(1-\epsilon/2)$. This gives us

$$t_1 + \cdots + t_n > B(1-\epsilon/2)(r-1)$$

Therefore, an optimal packing of the instance I uses more than $(1-\epsilon/2)(r-1)$ bins. From

$$Opt(I) > (1-\epsilon/2)(r-1)$$

we derive

$$r < Opt(I)/(1-\epsilon/2) + 1 \le Opt(I)(1+\epsilon) + 1$$

The last inequality is because $\epsilon \le 1$.

Therefore, in any case, the packing Y constructed by the algorithm `ApprxBinPacking` for the input instance I of the Bin Packing problem uses at most $Opt(I)(1+\epsilon)+1$ bins. The theorem is proved. □

Note that the condition that ϵ must be less than or equal to 1 loses no generality. In particular, if we are interested in an approximation algorithm for the Bin Packing problem with approximation ratio $1+\epsilon$ with $\epsilon > 1$, we simply use the **`First-Fit`** algorithm (Algorithm 23.1).

We conclude the lecture by the following theorem.

Theorem 25.5 *The* Bin Packing *problem has an asymptotic polynomial time approximation scheme.*

PROOF. For any $\epsilon > 0$, let $c_\epsilon = 2/\epsilon$. For each input instance I of the Bin Packing problem, let the algorithm `ApprxBinPacking` construct in time $O(n \log n) + h_0(\epsilon/2)$ a packing that uses at most $r \le Opt(I)(1+\epsilon/2)+1$ bins. Now for input instances I with $Opt(I) \ge c_\epsilon = 2/\epsilon$, we have

$$\frac{r}{Opt(I)} \le 1 + \frac{\epsilon}{2} + \frac{1}{Opt(I)} \le 1 + \epsilon$$

By the definition, the Bin Packing problem has an asymptotic polynomial time approximation scheme. □

The algorithm `ApprxBinPacking` runs in time $O(n \log n) + h_0(\epsilon)$, which is not a polynomial of $1/\epsilon$. When ϵ is small, the value $h_0(\epsilon)$ can be huge. Therefore, a further improvement on the algorithm `ApprxBinPacking` is an algorithm of the similar approximation ratio but with the running time bounded by a polynomial of n and $1/\epsilon$. This kind of algorithms is characterized by the following definition.

Definition 25.1 An optimization problem $Q = \langle I, S, f, opt \rangle$ has an *asymptotic fully polynomial time approximation scheme* (AFPTAS) if there is an approximation algorithm A for Q such that for any $\epsilon > 0$, there is a constant c_ϵ such that for all input instances $x \in I$ with $Opt(x) \ge c_\epsilon$, the algorithm A produces in time polynomial in both n and $1/\epsilon$ a solution for x with approximation ratio bounded by $1+\epsilon$.

The question whether the BIN PACKING problem has an asymptotic fully polynomial time approximation scheme was answered by Karmakar and Karp, who use a similar approach that reduces the bin packing problem to the linear programming problem. The algorithm uses some deep observations on the linear programming problem. We omit the detailed description here. Instead, we state the result directly.

Theorem 25.6 (Karmakar and Karp) *There is an approximation algorithm A for the* BIN PACKING *problem such that for any $\epsilon > 0$, the algorithm A produces in time polynomial in n and $1/\epsilon$ a packing in which the number of bins used is bounded by*

$$Opt(x)(1+\epsilon) + 1/\epsilon^2 + 3$$

Corollary 25.7 *The* BIN PACKING *problem has an asymptotic fully polynomial time approximation scheme.*

PROOF. For any $\epsilon > 0$, let $c_\epsilon = (8 + 6\epsilon^2)/\epsilon^3$. For each input instance I of the BIN PACKING problem, let the Karmakar-Karp algorithm construct a packing that uses at most

$$r \leq Opt(I)(1 + \epsilon/2) + (2/\epsilon)^2 + 3$$

bins. Now for input instances I with

$$Opt(I) \geq c_\epsilon = (8 + 6\epsilon^2)/\epsilon^3$$

we have

$$\frac{r}{Opt(I)} \leq 1 + \frac{\epsilon}{2} + \frac{(2/\epsilon)^2 + 3}{Opt(I)} \leq 1 + \epsilon$$

Moreover, the algorithm runs in time polynomial in n and $2/\epsilon$, which is also in polynomial in n and $1/\epsilon$. $\square$

CPSC-669 Computational Optimization

Lecture #26, October 27, 1995

Lecturer: Professor Jianer Chen
Scribe: Li Shao
Revision: Jianer Chen

26 Multi-processor scheduling

In the next few lectures, we study how the techniques developed for the BIN PACKING problem can be used to develop a polynomial time approximation scheme for the MULTI-PROCESSOR SCHEDULING problem.

Recall that the MULTI-PROCESSOR SCHEDULING problem is defined as follows.

> MULTI-PROCESSOR SCHEDULING
>
> INPUT: $\langle t_1, t_2, \ldots, t_n; m\rangle$, all integers, where t_i is the processing time for the ith job
>
> OUTPUT: a scheduling of the n jobs on m identical processors such that the parallel finish time is minimized

We point out a few properties for the MULTI-PROCESSOR SCHEDULING problem:

1. Even if we fix the number m of processors to be any constant larger than 1, the problem is still NP-hard (Theorem 13.3);
2. If the number m of processors is a fixed constant, then the problem has a fully polynomial time approximation scheme (Corollary 16.2).
3. If m is not fixed, the problem is strongly NP-hard and has no fully polynomial time approximation scheme unless P = NP (see Lecture Notes #18).

Therefore, the best we can expect for the MULTI-PROCESSOR SCHEDULING problem is a (non-fully) polynomial time approximation scheme.

The MULTI-PROCESSOR SCHEDULING problem can also be regarded as a variation of the BIN PACKING problem in which we are given n objects of sizes t_1, ..., t_n, respectively, and the number m of bins, and we are asked to pack the objects into the m bins such that the bin size is minimized. Therefore, there are two parameters: the number of bins and the bin size. Each of the Multi-Processor Scheduling problem and the Bin Packing problem fixes one parameter and optimizes the other parameter. In this sense, the Multi-Processor Scheduling problem is "dual" to the Bin Packing problem. Therefore, it is not very surprising that the techniques developed for approximation algorithms for the Bin Packing problem can be useful in deriving approximation algorithms for the Multi-Processor Scheduling problem.

Consider the following problem, where for an input instance I of the Bin Packing problem, we use $Opt(I)$ to denote the optimal value of I, i.e., the number of bins used by an optimal packing of the instance I.

> $(1+\epsilon)$-Bin Packing
>
> Input: $I = \langle t_1, t_2, \ldots, t_n; B \rangle$, all integers
>
> Output: a packing of the n objects into at most $Opt(I)$ bins such that the content of each bin is at most $(1+\epsilon)B$

We first show that the $(1+\epsilon)$-Bin Packing problem can be solved in polynomial time for a fixed constant $\epsilon > 0$. Then we show how this solution can be used to derive a polynomial time approximation scheme for the Multi-Processor Scheduling problem.

The idea for solving the $(1+\epsilon)$-Bin Packing problem is very similar to the one for the approximation algorithm `ApprxBinPacking` for the general Bin Packing problem. We first perform two preprocessing steps:

1. ignore the objects of size less than ϵB; and

2. partition the rest of the objects into π groups $G_1, \ldots, G_\pi$ so that the objects in each group have a very small difference in size. For each group G_i, replace every object by the one with the *smallest* size in G_i.

The preprocessing steps give us an instance I' of the (ϵ, π)-Bin Packing problem, for which an optimal solution can be constructed in polynomial time. Note that the optimal solution for I' is an under-estimation of the optimal solution for I and thus it uses no more than $Opt(I)$ bins. Then we restore the object sizes and add the small objects by greedy method to get a packing for the instance I. Since the difference in sizes of the objects in each group is very small, the restoring of object sizes will not increase the content for each bin very much. Similarly, adding small objects using greedy method will not induce much error.

The formal algorithm is given as follows.

Algorithm 26.1 `VaryBinPacking`

```
Input:  $I = \langle t_1, \ldots, t_n; B\rangle$ and $\epsilon > 0$
Output:  a packing of the objects of $I$ into bins of size
         $(1+\epsilon)B$

1.  sort $t_1, \ldots, t_n$; without loss of generality, let
                  $t_1 \ge t_2 \ge \cdots \ge t_n$
2.  let $h$ be the largest index such that $t_h > \epsilon B$; let
                  $I_0 = \langle t_1, t_2, \ldots, t_h; B\rangle$
3.  let $\pi = \lceil 1/\epsilon^2 \rceil$, divide the line segment $(\epsilon B, B]$ into $\pi$
     subsegments of equal length
           $(l_1, h_1],\ \ (l_2, h_2],\ \ \ldots,\ \ (l_\pi, h_\pi]$
     where $h_i = l_{i+1}$ and $h_i - l_i = (B - \epsilon B)/\pi$;
4.  partition the objects in $I_0$ into $\pi$ groups $G_1$, ..., $G_\pi$,
     such that an object is in group $G_i$ if and only if its
     size is in the range $(l_i, h_i]$; let $t'_i$ be the size of the
     smallest object in group $G_i$ (if $G_i$ is empty, let
     $t'_i = l_i$), and let $m_i$ be the number of objects in $G_i$;
5.  construct an optimal solution $Y'$ to the instance
           $I' = \langle t'_1 : m_1, t'_2 : m_2, \ldots, t'_\pi : m_\pi; B\rangle$
     for the $(\epsilon, \pi)$-Bin Packing problem;
6.  replace each object in $Y'$ of size $t'_j$ by a proper object
     in the group $G_j$ of $I_0$, for $j = 1, \ldots, \pi$, to construct a
     packing $Y_0$ for the instance $I_0$;
7.  add the objects $t_{h+1}, \ldots, t_n$ in $I$ to the packing $Y_0$ by
     greedy method (i.e., no new bin will be used until
     adding the current object to any used bins would
     exceed the size $(1+\epsilon)B$).  This results in a packing
     $Y$ for the instance $I$.
```

According to Theorem 24.1 and note that $\pi = \lceil 1/\epsilon^2 \rceil$, the (ϵ, π)-Bin Packing problem can be solved in time $O(n) + h_0(\epsilon)$, where $h_0(\epsilon)$ is a function depending only on ϵ. We conclude that the algorithm `VaryBinPacking` runs in time $O(n \log n) + h_0(\epsilon)$, if an $O(n \log n)$ time sorting algorithm is used for step 1.

As before, we denote by $Opt(I)$ the optimal value, i.e., the number of bins used by an optimal packing, of the input instance I of the general Bin Packing problem.

Lemma 26.1 *The packing Y' for the instance I' constructed by step 5 of the algorithm* `VaryBinPacking` *uses no more than $Opt(I)$ bins.*

PROOF. The instance I_0 is a subset of the instance I. Thus, $Opt(I_0) \leq Opt(I)$. The instance I' is obtained from I_0 by replacing each object in I_0 by a smaller object. Thus, $Opt(I') \leq Opt(I_0) \leq Opt(I)$. Since Y' is an optimal packing for I', Y' uses $Opt(I') \leq Opt(I)$ bins. □

Lemma 26.2 *In the packing Y_0 constructed by step 6 of* `VaryBinPacking`*, no bin has content larger than $(1+\epsilon)B$, and Y_0 uses no more than $Opt(I)$ bins.*

PROOF. According to step 6 of the algorithm `VaryBinPacking`, the number of bins used by Y_0 is the same as that used by Y'. By Lemma 26.1, the packing Y_0 uses no more than $Opt(I)$ bins.

Each object of size t'_i in I' corresponds to an object in group G_i in I_0. The packing Y_0 for I_0 is obtained from the packing Y' by restoring each object of I' to the corresponding object in I_0. Since t'_i is the size of the smallest object in G_i and no object in G_i has size larger than

$$t'_i + (h_i - l_i) = t'_i + (B - \epsilon B)/\pi$$

the size increase for each object from Y' to Y_0 is bounded by $(B - \epsilon B)/\pi$.

Moreover, since all objects in I' have size at least ϵB, and the packing Y' has bin size B, each bin in the packing Y' holds at most $\lfloor 1/\epsilon \rfloor$ objects. Therefore, the size increase for each bin from Y' to Y_0 is bounded by

$$((B - \epsilon B)/\pi) \cdot \lfloor 1/\epsilon \rfloor = ((B - \epsilon B)/\lceil 1/\epsilon^2 \rceil) \cdot \lfloor 1/\epsilon \rfloor \leq (B/(1/\epsilon^2)) \cdot (1/\epsilon) = \epsilon B$$

Since the content of each bin of the packing Y' is at most B, we conclude that the content of each bin of the packing Y_0 is at most $(1+\epsilon)B$. □

Lemma 26.3 *The packing Y constructed by step 7 of* `VaryBinPacking` *uses no more than $Opt(I)$ bins, and each bin of Y has content at most $(1+\epsilon)B$.*

PROOF. By Lemma 26.2, each bin of the packing Y_0 has content at most

$(1+\epsilon)B$. The packing Y is obtained from Y_0 by adding the objects of size bounded by ϵB using greedy method. That is, suppose we want to add an object of size not larger than ϵB and there is a used bin whose content will not exceed $(1+\epsilon)B$ after adding the object to the bin, then we add the object to the bin. A new bin is introduced only if no used bin can have the object added without exceeding the content $(1+\epsilon)B$. The greedy method ensures that the content of each bin in Y is bounded by $(1+\epsilon)B$. Note that since all added objects have size bounded by ϵB, when a new bin is introduced, all used bins have content larger than B.

If no new bin was introduced in the process of adding small objects in step 7, then the number of bins used by the packing Y is the same as the number of bins used by the packing Y_0. By Lemma 26.2, in this case the packing Y uses no more than $Opt(I)$ bins.

Now suppose that new bins were introduced in the process of adding small objects in step 7. Let r be the number of bins used by the packing Y. By the above remark, at least $r-1$ bins in the packing Y have content larger than B. Therefore, we have

$$t_1 + \cdots + t_n > B(r-1)$$

This shows that we need more than $r-1$ bins of size B to pack the objects in I in any packing. Consequently, the value $Opt(I)$ is at least $(r-1)+1 = r$. That is, the packing Y uses no more than $Opt(I)$ bins. $\square$

We conclude this lecture with the following theorem.

Theorem 26.4 *Given an instance $I = \langle t_1, \ldots, t_n; B\rangle$ for the* Bin Packing *problem and a constant $\epsilon > 0$, The algorithm* `VaryBinPacking` *constructs in time $O(n \log n) + h_0(\epsilon)$ a packing for I that uses no more than $Opt(I)$ bins and the content of each bin is bounded by $(1+\epsilon)B$, where $Opt(I)$ is the number of bins used by an optimal packing of I using bins of size B and $h_0(\epsilon)$ is a function depending only on ϵ.*

Corollary 26.5 *The $(1+\epsilon)$-*Bin Packing *problem can be solved in polynomial time for a fixed constant ϵ.*

CPSC-669 Computational Optimization

Lecture #27, October 30, 1995

Lecturer: Professor Jianer Chen
Scribe: Li Shao
Revision: Jianer Chen

27 Approximating multi-processor scheduling

In the last lecture, we developed an algorithm `VaryBinPacking` that, given an input instance $I = \langle t_1, \ldots, t_n; B\rangle$ of the BIN PACKING problem and a constant $\epsilon > 0$, constructs in time $O(n \log n) + h_0(\epsilon)$ a packing using at most $Opt(I)$ bins such that the content of each bin is bounded by $(1+\epsilon)B$, where $h_0(\epsilon)$ is a function depending only on ϵ.

We use this algorithm to develop a polynomial time approximation scheme for the MULTI-PROCESSOR SCHEDULING problem. We first re-formulate the MULTI-PROCESSOR SCHEDULING problem in the language of bin packing.

> MULTI-PROCESSOR SCHEDULING (Bin Packing version)
>
> INPUT: $\langle t_1, t_2, \ldots, t_n; m\rangle$, all integers, where t_i is the size of the ith object
>
> OUTPUT: a packing of the n objects into m bins of size B with B minimized

We use the idea of binary search to find the optimal bin size B. In general, suppose that we try bin size B, and find out that the input instance $\langle t_1, \ldots, t_n; B\rangle$ for the BIN PACKING problem needs more than m bins in its optimal packing, then the tried bin size B is too small. So we will try a larger bin size. On the other hand, if the instance $\langle t_1, \ldots, t_n; B\rangle$ needs no more than m bins, then we may want to try a smaller bin size because we are minimizing the bin size. Note that the algorithm `VaryBinPacking` can be used to estimate the number of bins used by an optimal packing of the instance $\langle t_1, \ldots, t_n; B\rangle$.

We first discuss the initial bounds for the bin size in the binary search. Fix an input instance $\langle t_1, \ldots, t_n; m\rangle$ for the MULTI-PROCESSOR SCHEDULING problem. Let

$$\text{Avg} = \max\{\sum_{i=1}^{n} t_i/m, t_1, t_2, \ldots, t_n\}$$

Lemma 27.1 *The minimum bin size of the input instance* $\langle t_1, \ldots, t_n; m\rangle$ *for the* MULTI-PROCESSOR SCHEDULING *problem is at least Avg.*

PROOF. Since $\sum_{i=1}^n t_i/m$ is the average content of the m bins for packing the n objects of size t_1, …, t_n, any packing of the n objects into the m bins has at least one bin with content at least $\sum_{i=1}^n t_i/m$. That is, the bin size of the packing is at least $\sum_{i=1}^n t_i/m$.

Moreover, the bin size of the packing should also be at least as large as any t_i since every object has to be packed into a bin in the packing.

This shows that for any packing of the n objects of size t_1, …, t_n into the m bins, the bin size is at least Avg. The lemma is proved. □

This gives a lower bound on the bin size for the input instance I of the MULTI-PROCESSOR SCHEDULING problem. We also have the following upper bound.

Lemma 27.2 *The minimum bin size of the input instance* $\langle t_1, \ldots, t_n; m\rangle$ *for the* MULTI-PROCESSOR SCHEDULING *problem is bounded by* $2 \cdot$ *Avg.*

PROOF. Suppose that the lemma is false. Let r be the minimum bin size for packing $I = \langle t_1, \ldots, t_n; m\rangle$ into m bins, and $r > 2 \cdot \text{Avg}$.

Let Y be a packing of I into m bins such that the bin size of Y is r. Furthermore, we suppose that Y is the packing in which the least number of bins have content r. Let B_1, B_2, …, B_m be the bins used by Y, where the bin B_1 has content $r > 2 \cdot \text{Avg}$. Then at least one of the bins B_2, …, B_m has content less than Avg — otherwise, the sum of total contents of the bins B_1, B_2, …, B_m would be larger than $m\text{Avg} \geq \sum_{i=1}^n t_i$. Without loss of generality, suppose that the bin B_2 has content less than Avg. Now remove any object t_i in the bin B_1 and add t_i to the bin B_2. We have

1. the content of the bin B_1 in the new packing is less than r;
2. the content of the bin B_2 in the new packing is less than
$$\text{Avg} + t_i \leq 2 \cdot \text{Avg} < r$$
3. the contents of the other bins are unchanged.

Thus, in the new packing, the number of bins that have content r is one less than the number of bins of content r in the packing Y. This contradicts our assumption that Y has the least number of bins of content r.

This contradiction proves the lemma. □

Therefore, the minimum bin size for packing the instance I into m bins is in the range [Avg, 2Avg]. We apply binary search on this range to find an approximation for the optimal solution of I for the Multi-Processor Scheduling problem.

Algorithm 27.1 `ApprxMPS`

```
Input:  $I = \langle t_1, \ldots, t_n; m \rangle$, all integers, and $\epsilon > 0$
Output:  a scheduling of the $n$ jobs of processing time $t_1$,
           $t_2$, ..., $t_n$ on $m$ identical processors.

1.  Avg $= \max\{\sum_{i=1}^n t_i/m, t_1, t_2, \ldots, t_n\}$;
2.  lower $= \lfloor$Avg$\rfloor$;     upper $= \lceil 2 \cdot$ Avg$\rceil$;
3.  while upper $-$ lower $> \epsilon \cdot$ Avg$/4$ do
        $B = \lfloor$(lower + upper)$/2\rfloor$;
        call the algorithm VaryBinPacking on the input
          $\langle t_1, \ldots, t_n; B \rangle$ and $\epsilon/4$; suppose that the algorithm
          uses $r$ bins on the input;
        if $r > m$
        then lower $= B$
        else upper $= B$;
4.  let $B^* = \lfloor$upper$(1 + \epsilon/4)\rfloor$;
5.  call the algorithm VaryBinPacking on the input
      $\langle t_1, \ldots, t_n; B^* \rangle$ and $\epsilon/4$ to construct a scheduling
      of $I$.
```

We first study the complexity of the above algorithm `ApprxMPS`. The complexity of the algorithm is dominated by step 3. We start with

$$\text{upper} - \text{lower} = 2 \cdot \text{Avg} - \text{Avg} = \text{Avg}$$

Since we are using binary search, each execution of the body of the **while** loop will half the difference (upper $-$ lower). Therefore, after $O(\log(1/\epsilon))$ executions of the body of the **while** loop in step 3, we must have

$$\text{upper} - \text{lower} \le \epsilon \cdot \text{Avg}/4$$

That is, the body of the **while** loop is executed at most $O(\log(1/\epsilon))$ times.

In each execution of the body of the **while** loop in step 3, we call the algorithm `VaryBinPacking` on input $\langle t_1, \ldots, t_n; B \rangle$ and $\epsilon/4$, which takes time $O(n \log n) + h_0(\epsilon/4) = O(n \log n) + h_1(\epsilon)$, where $h_1(\epsilon)$ is a function depending only on ϵ. Therefore, the running time of the algorithm `ApprxMPS` is bounded by

$$O(\log(1/\epsilon))(O(n \log n) + h_1(\epsilon)) = O(n \log n \log(1/\epsilon)) + h_2(\epsilon)$$

where $h_2(\epsilon)$ is a function depending only on ϵ.

Theorem 27.3 *The running time of the algorithm* `ApprxMPS` *on input instance* $I = \langle t_1, \ldots, t_n; m \rangle$ *and* $\epsilon > 0$ *is bounded by* $O(n \log n \log(1/\epsilon)) + h_2(\epsilon)$. *In particular, for a fixed constant* $\epsilon > 0$, *the algorithm* `ApprxMPS` *runs in polynomial time.*

We will present the analysis for the approximation ratio for the algorithm `ApprxMPS` in the next lecture.

CPSC-669 Computational Optimization

Lecture #28, November 1, 1995

Lecturer: Professor Jianer Chen
Scribe: Li Shao
Revision: Jianer Chen

28 More on multi-processor scheduling

In the last lecture, we presented the following algorithm for the MULTI-PROCESSOR SCHEDULING problem.

Algorithm 28.1 `ApprxMPS`

```
Input:  $I = \langle t_1, \ldots, t_n; m\rangle$, all integers, and $\epsilon > 0$
Output:  a scheduling of the $n$ jobs of processing time $t_1$,
         $t_2$, ..., $t_n$ on $m$ identical processors.

1.  Avg $= \max\{\sum_{i=1}^{n} t_i/m, t_1, t_2, \ldots, t_n\}$;
2.  lower $= \lfloor$Avg$\rfloor$;     upper $= \lceil 2 \cdot$ Avg$\rceil$;
3.  while upper $-$ lower $> \epsilon \cdot$ Avg$/4$ do
        $B = \lfloor$(lower $+$ upper)$/2\rfloor$;
        call the algorithm VaryBinPacking on the input
          $\langle t_1, \ldots, t_n; B\rangle$ and $\epsilon/4$; suppose that the algorithm
          uses $r$ bins on the input;
        if $r > m$
        then lower $= B$
        else upper $= B$;
4.  let $B^* = \lfloor$upper$(1 + \epsilon/4)\rfloor$;
5.  call the algorithm VaryBinPacking on the input
      $\langle t_1, \ldots, t_n; B^*\rangle$ and $\epsilon/4$ to construct a scheduling
      of $I$.
```

we also showed that the algorithm runs in time $O(n \log n \log(1/\epsilon)) + h_2(\epsilon)$, where $h_2(\epsilon)$ is a function depending only on ϵ. Now we discuss the approximation ratio of the algorithm.

Fix an input instance $I = \langle t_1, \ldots, t_n; m\rangle$ for the MULTI-PROCESSOR SCHEDULING problem. Let $Opt(I)$ be the optimal solution, i.e., the parallel finish time of an optimal scheduling, of the instance I.

Lemma 28.1 *In the whole execution of the algorithm* `ApprxMPS`*, we always have*

$$lower \leq Opt(I) \leq upper(1 + \epsilon/4)$$

PROOF. Initially, lower = $\lfloor \text{Avg} \rfloor$ and upper = $\lceil 2 \cdot \text{Avg} \rceil$. By Lemmas 27.1 and 27.2, we have lower $\leq Opt(I) \leq$ upper$(1 + \epsilon/4)$.

Now for each execution of the **while** loop in step 3, we start with a bin size B and call the algorithm `VaryBinPacking` on input $\langle t_1, \ldots, t_n; B \rangle$ and $\epsilon/4$, which uses r bins.

If $r > m$, by the algorithm `VaryBinPacking`, the minimum number of bins used by a packing to pack the objects into bins of size B is at least as large as r. Therefore, if the bin size is B, then we need more then m bins to pack the objects t_1, ..., t_n. Thus, in order to pack the objects t_1, ..., t_n into m bins, the bin size B is too small. That is, $Opt(I) > B$. Since in this case we set lower = B, the relation lower $\leq Opt(I) \leq$ upper$(1 + \epsilon/4)$ still holds.

If $r \leq m$, then the objects t_1, ..., t_n can be packed in r bins of size $(1 + \epsilon/4)B$. Certainly, the objects can also be packed in m bins of size $(1 + \epsilon/4)B$. This gives $Opt(I) \leq (1 + \epsilon/4)B$. Thus, setting upper = B still keeps the relation lower $\leq Opt(I) \leq$ upper$(1 + \epsilon/4)$.

This proves the lemma. $\square$

Now we are ready to show that the algorithm `ApprxMPS` is a polynomial time approximation scheme for the MULTI-PROCESSOR SCHEDULING problem.

Theorem 28.2 *On any input instance* $I = \langle t_1, \ldots, t_n; m \rangle$ *for the* MULTI-PROCESSOR SCHEDULING *problem and for any* ϵ*,* $0 < \epsilon \leq 1$*, the algorithm* `ApprxMPS` *constructs in time* $O(n \log n \log(1/\epsilon)) + h_2(\epsilon)$ *a scheduling of the* n *jobs on the* m *processors with approximation ratio* $1 + \epsilon$*, where* $h_2(\epsilon)$ *is a function depending only on* ϵ*.*

PROOF. The time complexity of the algorithm `ApprxMPS` is given by Theorem 27.3.

By Lemma 28.1, the relation

$$\text{lower} \leq Opt(I) \leq \text{upper}(1 + \epsilon/4)$$

always holds. In particular, at step 4 of the algorithm, we have

$$Opt(I) \leq \text{upper}(1 + \epsilon/4)$$

Since $Opt(I)$ is an integer, we should also have

$$Opt(I) \leq \lfloor \text{upper}(1+\epsilon/4) \rfloor = B^*$$

Therefore, the bin of size B^* is at least as large as the bin of size $Opt(I)$. Since the objects t_1, ..., t_n can be packed into m bins of size $Opt(I)$, we conclude that the objects t_1, ..., t_n can also be packed into m bins of size B^*. By the property of the algorithm `VaryBinPacking`, on input instance $I = \langle t_1, \ldots, t_n; B^* \rangle$ and $\epsilon/4$, the algorithm `VaryBinPacking` packs the objects t_1, ..., t_n into at most m bins, with each bin of content at most $B^*(1+\epsilon/4)$. Therefore, the packing is a scheduling of the n jobs on the m processors.

Now let us consider the content bound $B^*(1+\epsilon/4)$ for the bins in the packing constructed by the algorithm `VaryBinPacking`. At step 4, we have

$$\text{upper} - \text{lower} \leq \epsilon \cdot \text{Avg}/4$$

Since lower = Avg initially, and lower is never decreased, we have

$$\text{upper} \leq \text{lower} + \epsilon \cdot \text{Avg}/4 \leq \text{lower} + \epsilon \cdot \text{lower}/4 = \text{lower}(1+\epsilon/4)$$

By Lemma 28.1, we always have

$$\text{lower} \leq Opt(I) \leq \text{upper}(1+\epsilon/4)$$

Thus

$$\text{upper}(1+\epsilon/4) \leq \text{lower}(1+\epsilon/4)^2 \leq Opt(I)(1+\epsilon/4)^2$$

Therefore, the content bound $B^*(1+\epsilon/4)$ is bounded by

$$\begin{aligned} B^*(1+\epsilon/4) &= \lfloor \text{upper}(1+\epsilon/4) \rfloor (1+\epsilon/4) \\ &\leq \text{upper}(1+\epsilon/4)^2 \leq Opt(I)(1+\epsilon/4)^3 \end{aligned}$$

Now $Opt(I)(1+\epsilon/4)^3 \leq Opt(I)(1+\epsilon)$ for $\epsilon \leq 1$. Recall that in the scheduling, the number m of bins corresponds to the number of processors, and the maximum bin content $B^*(1+\epsilon/4)$ corresponds to the parallel finish time. In conclusion, the scheduling of the n jobs on the m processors constructed by the algorithm `ApprxMPS` has parallel finish time bounded by $Opt(I)(1+\epsilon)$. In other words, the algorithm `ApprxMPS` has approximation ratio $1+\epsilon$. □

Corollary 28.3 *The* MULTI-PROCESSOR SCHEDULING *problem has a polynomial time approximation scheme.*

Again, the condition $\epsilon \leq 1$ is not crucial. In particular, we will see below that if $\epsilon > 1$, a much simpler approximation algorithm for the Multi-Processor Scheduling problem can be designed to have approximation ratio bounded by $1 + \epsilon$.

Algorithm 28.2 SimpleMPS

```
Input:  $I = \langle t_1, \ldots, t_n; m \rangle$, all integers
Output:  a scheduling of the $n$ jobs of processing time $t_1$,
         $t_2$, ..., $t_n$ on $m$ identical processors

for $i = 1$ to $n$ do
    assign $t_i$ to the processor with the lightest load;
```

Using a data structure such as a 2-3 tree to organize the m processors using their loads as the keys, we can always find the lightest loaded processor, update its load, and re-insert it back to the data structure in time $O(\log m)$. With this implementation, the algorithm `SimpleMPS` runs in time $O(n \log m)$.

Now we study the approximation ratio of the algorithm `SimpleMPS`.

Theorem 28.4 *Algorithm* `SimpleMPS` *for the* Multi-Processor Scheduling *problem has approximation ratio bounded by* 2*.*

PROOF. Let $I = \langle t_1, \ldots, t_n; m \rangle$ be an input instance to the Multi-Processor Scheduling problem. Suppose that the algorithm `SimpleMPS` constructs a scheduling S for I with parallel finish time T. Let P_1 be a processor that has the execution time T assigned by the scheduling S.

If the processor P_1 is assigned only one job, then the job has processing time T, and any scheduling on I has parallel finish time at least T. In this case, the scheduling S is a optimal scheduling with approximation ratio 1.

So suppose that the processor P_1 is assigned at least two jobs. Let t_0 be the last job assigned to the processor P_1. We have $T - t_0 > 0$. By our strategy, at the time the job t_0 is about to be assigned to the processor P_1, all processors have load at least $T - t_0$. This gives:

$$\sum_{i=1}^{n} t_i \geq m(T - t_0) + t_0 = mT - (m-1)t_0$$

This gives

$$T \leq \frac{\sum_{i=1}^{n} t_i + (m-1)t_0}{m}$$

$$
\begin{aligned}
&= \frac{\sum_{i=1}^{n} t_i}{m} + \frac{m-1}{m} t_0 \\
&\leq \frac{\sum_{i=1}^{n} t_i}{m} + t_0
\end{aligned}
$$

Now since the optimal value $Opt(I)$ is at least as large as $(\sum_{i=1}^{n} t_i)/m$, and at least as large as t_0, we conclude that

$$T \leq 2 \cdot Opt(I)$$

Consequently, the approximation algorithm `SimpleMPS` has approximation ratio bounded by 2. $\square$

There are certainly many possible ways to improve the performance of the algorithm `SimpleMPS`. For example, it seems that if we sort the jobs first so that the larger jobs will be assigned first, then we may get an improvement on the approximation ratio. In fact, it can be shown that such a modification makes the algorithm have an approximation ratio of 4/3. Students are encouraged to think of other possible improvements.

CPSC-669 Computational Optimization

Lecture #29, November 3, 1995

Lecturer: Professor Jianer Chen
Scribe: Shijin Lu
Revision: Jianer Chen

29 Approximability with a constant ratio

So far we have seen many optimization problems that can be approximated in polynomial time to approximation ratio $1+\epsilon$, for any given constant ϵ. These problems are classified into the following two classes.

Definition 29.1 An optimization problem is in the class FPTAS if it has a fully polynomial time approximation scheme. An optimization problem is in the class PTAS if has a polynomial time approximation scheme.

Obviously, FPTAS is a subclass of PTAS.

On the other hand, there are many other optimization problems that do not seem to have such nice approximability. There is a large class of optimization problems of practical importance, which do not seem to have polynomial time approximation schemes. The rest of this course will be centered on the study of these optimization problems.

Let us first consider the Vertex Cover problem. Given a graph $G = (V, E)$, we say that a subset V_0 of V is a *vertex cover* of the graph G if every edge of the graph G has at least one endpoint in V_0.

Vertex Cover

Input: an undirected graph $G = (V, E)$

Output: a vertex cover V_0 of minimum cardinality

The Vertex Cover problem has applications in computer networks, VLSI design, and circuit testing. For example, in computer network, we are given a network, which can be regarded as a graph, and we are asked to pick a set of nodes in the network so that all connections of the network are monitored by the nodes in the set. To economize the resources, we expect to have as few nodes as possible in the set. This is exactly the Vertex Cover problem.

We have a very efficient and simple approximation algorithm for the Vertex Cover problem. The algorithm is given below.

Algorithm 29.1 `ApprxVC`

```
Input:  an undirected graph $G = (V, E)$
Output:  a vertex cover of $G$

1.  Let $V_0 = \phi$;
2.  **for** each edge $e$ of $G$ **do**
       **if** $e$ has no ends in $V_0$
       **then** add *both* ends of $e$ to $V_0$
```

From the `ApprxVC` algorithm, we can easily get two observations.

Observation 29.2 The set V_0 constructed by the algorithm `ApprxVC` is a vertex cover of the graph G.

As we can see, the algorithm makes sure that all edges of the graph G are covered by the set V_0.

Observation 29.3 The algorithm `ApprxVC` actually constructs a maximal matching for the graph G.

When the algorithm `ApprxVC` includes two endpoints u and v of an edge e in the set V_0, we can regard that the algorithm matches the two endpoints u and v by the edge e. By the algorithm, if the endpoints u and v are matched by e, no other edge incident on either u or v would be used for matching. That is, the set

$$E_0 = \{e \mid e \text{ is picked by } \texttt{ApprxVC} \text{ for matching its two ends}\}$$

is a matching in G. Moreover, the matching is maximal because every edge has at least one end in V_0.

Theorem 29.1 *The algorithm* `ApprxVC` *is an approximation algorithm for the* Vertex Cover *problem and has approximation ratio* 2.

PROOF. By Observation 29.2, the algorithm `ApprxVC` always constructs a vertex cover for the input graph G.

By Observation 29.3, a maximal matching E_0 is constructed by the algorithm `ApprxVC`. Let

$$E_0 = \{e \mid e \text{ is picked by } \texttt{ApprxVC} \text{ for matching its two ends}\}$$

and let C be any minimum vertex cover. Then every edge in E_0 should be covered by C, i.e., each edge in E_0 should have at least one end in C. Since no two edges in E_0 share a common end, we should have

$$|C| \geq |E_0|$$

Since each edge in E_0 has two ends in V_0 and no two edges in E_0 share a common end, we have

$$2|E_0| = |V_0|$$

In conclusion

$$Opt(G) = |C| \geq |V_0|/2$$

This gives the approximation ratio

$$|V_0|/Opt(G) \leq 2$$

and the theorem is proved. □

The algorithm `ApprxVC` looks very simple. However, it gives the best approximation ratio known for the VERTEX COVER problem. Actually, it is an outstanding open problem whether the VERTEX COVER problem has a polynomial time approximation algorithm with approximation ratio $r < 2$, for a fixed constant $r > 0$.

I assign the following as one of the project problems.

Project problem: Improve the approximation ratio 2 for the VERTEX COVER problem on graph classes with some reasonable restrictions.

There are many optimization problems like the VERTEX COVER problem that have polynomial time approximation algorithms with approximation ratio bounded by a fixed constant c ($c = 2$ for the VERTEX COVER problem). On the other hand, for many of them, it is unknown whether the constant c can be arbitrarily close to 1, i.e., whether the problems have polynomial time approximation schemes. We discuss another example as follows.

Let X, Y, and Z be three finite sets. Given a subset $S \subseteq X \times Y \times Z$, a *matching* M in S is a subset of S such that no two triples in M have the same coordinate at any dimension. The 3-DIMENSIONAL MATCHING problem is defined as follows.

3-D MATCHING

INPUT: a set $S \subseteq X \times Y \times Z$ of triples

OUTPUT: a matching M in S with $|M|$ maximized

The 3-D Matching problem is a generalization of the classical "marriage problem": Given n unmarried men and m unmarried women, along with a list of all male-female pairs who would be willing to marry one another, find the largest number of pairs so that polygamy is avoided and every paired person receives an acceptable spouse. Analogously, in the 3-D Matching problem, the sets X, Y, and Z correspond to three sexes, and each triple in S corresponds to a 3-way marriage that would be acceptable to all three participants.

Remark 29.4 The 2-D Matching problem can be similarly defined: given a set $S \subseteq X \times Y$ of pairs, find a maximum subset M of S such that no two pairs in M agree in any coordinate. The 2-D Matching problem is the standard graph matching problem. In fact, the sets X and Y can be regarded as the vertices of a graph G, and each pair in the set S corresponds to an edge in the graph G. Now a matching M in S is simply a subset of edges in which no two edges share a common end. That is, a matching in S is a graph matching in the corresponding graph G. As we have studied in Lectures 8-10, the graph matching problem, i.e., the 2-D Matching problem can be solved in polynomial time.

Remark 29.5 The 3-D Matching problem is NP-hard. This is from the fact that the decision version of the 3-D Matching problem is NP-complete (see Garey and Johnson's book) and can be reduced to the optimization version of the 3-D Matching problem. In fact, the decision version of the 3-D Matching problem is listed by Garey and Johnson as one of the six basic NP-complete problems.

We present two polynomial time approximation algorithms for the 3-D Matching problem.

Let $S \subseteq X \times Y \times Z$ be a set of triples and let M be a matching of S. We say that a triple (x, y, z) in $S - M$ *does not contradict the matching* M if no triple in M has x as its first coordinate, or has y as its second coordinate, or has z as its third coordinate. In other words, (x, y, z) does not contradict the matching M if $M \cup \{(x, y, z)\}$ is still a matching.

Algorithm 29.2 `Apprx3D-First`

```
Input:  a set $S \subseteq X \times Y \times Z$ of triples
Output:  a matching $M$ in $S$

1.  let $M = \phi$.
```

```
2.  for each triple $(x, y, z)$ in $S$ do
      if $(x, y, z)$ does not contradict $M$
      then add $(x, y, z)$ to $M$.
```

It is easy to verify that the algorithm `Apprx3D-First` runs in polynomial time. In fact, if we use three arrays for the symbols in X, Y, and Z, and mark the symbols as "in M" or "not in M", then in constant time we can decide whether a triple (x, y, z) contradicts the matching M. With these data structures, the algorithm `Apprx3D-First` runs in linear time.

Theorem 29.2 *The algorithm* `Apprx3D-First` *constructs a matching in the set S and has approximation ratio* 3*.*

PROOF. From the algorithm `Apprx3D-First`, it is clear that the set M constructed is a matching in the given set S.

Let $M_{\max}$ be a maximum matching in S and let (x, y, z) be a triple in $M_{\max}$. By the algorithm `Apprx3D-First`, the triple (x, y, z) contradicts the matching M (otherwise, it would have been added to M by the algorithm). Therefore, either x is the first coordinate of a triple in M, or y is the second coordinate of a triple in M, or z is the third coordinate of a triple in M. Therefore, the total number of symbols appearing in the matching M (in either the first dimension, or the second dimension, or the third dimension) is at least $|M_{\max}|$. Since each triple in M uses exactly three symbols, we conclude that the number of triples in the matching M is at least $|M_{\max}|/3$. That is,

$$Opt(S)/|M| = |M_{\max}|/|M| \leq 3$$

The theorem is proved. $\square$

CPSC-669 Computational Optimization

Lecture #30, November 6, 1995

Lecturer: Professor Jianer Chen
Scribe: Shijin Lu
Revision: Jianer Chen

30 3-dimensional matching

We continue our discussion on the 3-D Matching problem.

Let $S \subseteq X \times Y \times Z$ be a set of triples. Without loss of generality, we assume that the symbol sets X, Y, and Z are all pairwise disjoint. Therefore, it makes no ambiguity to say that a triple t contains a symbol w in $X \cup Y \cup Z$. Recall that a *matching* M in S is a subset of S in which no two triples agree in any coordinate. We say that a symbol $w \in X \cup Y \cup Z$ is in the matching M if a triple in M contains the symbol w. A triple t in $S - M$ *contradicts* the matching M if a symbol in t is also in the matching M. We say that a matching M in S is *maximal* if every triple in $S - M$ contradicts M.

Before we present another approximation algorithm for the 3-D Matching problem, we diverge to a related problem.

> k-Triple Matching
>
> Given a set $S \subseteq X \times Y \times Z$ of n triples and an integer $k > 0$, find a matching in S with k triples or report that no such a matching exists in S.

It is clear that the k-Triple Matching problem can be solved in time $O(n^k)$ if we pick every k triples in S and check whether they make a matching. However, the algorithm will be very time-consuming even for a small value of k. We would like to have a better algorithm for the problem. In particular, we would like to have an algorithm for the problem such that in the time complexity of the algorithm, the exponent of n is independent of the value k.

We present an algorithm solving the k-Triple Matching problem as follows. The algorithm is first given as a nondeterministic algorithm, i.e., an algorithm that can "guess" a desired object in a set without exhaustively searching the set. Then we show how the nondeterministic algorithm can be converted into a deterministic one.

We first suppose that a maximum matching in the set S contains at least k triples. Fix a matching $M_0 = \{t_1, t_2, \ldots, t_k\}$ of k triples in S (the matching M_0 is unknown to our algorithm).

Let M_1 be a maximal matching in S. M_1 can be found in time $O(n)$ by, say, the algorithm `Apprx3D-First` given in the last lecture. If $|M_1| \geq k$, then we are done — any k triples in M_1 make a matching of k triples in S. Thus, we assume $|M_1| < k$.

Let t_i be any triple in the matching M_0. If t_i is in M_1, then certainly the symbols in t_i are also in M_1. If t_i is not in M_1, then t_i contradicts M_1 because M_1 is maximal. Thus, in any case, for each triple t_i in M_0 at least one symbol in t_i is in the matching M_1.

Thus, our algorithm guesses k symbols a_1, …, a_k in M_1 such that a_i is a symbol in the triple t_i, $i = 1, \ldots, k$. This gives us a "pseudo-matching"

$$M_2 = \{t_1^{(2)}, t_2^{(2)}, \ldots, t_k^{(2)}\}$$

where $t_i^{(2)}$ is the triple t_i in the matching M_0 with the symbol a_i present and the other two symbols replaced by a special symbol '$*$', for $i = 1 \ldots, k$. This gives us the initial pseudo-matching. Note that the pseudo-matching M_2 can be constructed from the matching M_1 in time $O(k)$ if the guessed symbols a_1, …, a_k are given.

Inductively, suppose that we have obtained a pseudo-matching

$$M_j = \{t_1^{(j)}, t_2^{(j)}, \ldots, t_k^{(j)}\}$$

where $t_i^{(j)}$ is the triple t_i in the matching M_0 with at least one symbol present and the other symbols replaced by the symbol '$*$', for $i = 1 \ldots, k$. We say that a triple t in S is *consistent* with a triple $t_i^{(j)}$ if t and $t_i^{(j)}$ agree in all coordinates except those on which $t_i^{(j)}$ has the symbol '$*$'.

Now we try to fill the missing symbols in the pseudo-matching M_j using a greedy algorithm. Formally, we start with $M' = \phi$ then scan the triples in S. We add a triple t in S to M' if (1) t is consistent with a triple $t_i^{(j)}$ in M_j; (2) no symbols in t appear in other triples in M_j; and (3) the triple t does not contradict the matching M'. Note that this process is equivalent to filling the missing symbols '$*$' in the triple $t_i^{(j)}$ by the corresponding symbols in the triple t.

The above process ends up with a matching M' in S. Note that the matching M' can be constructed from the matching M_j in time $O(n)$. If $|M'| = k$, i.e., if all missing symbols in M_j are filled, then we are done (note that the matching M' may not necessarily be the matching M_0). Otherwise, $|M'| < k$. Without loss of generality, suppose that the triples in M_j whose missing symbols are filled are the triples $t_1^{(j)}$, …, $t_h^{(j)}$, $h < k$. Now consider the triple $t_{h+1}^{(j)}$, which corresponds to the triple t_{h+1} in the matching M_0. It is clear that the only reason that the triple t_{h+1} was not included in the matching M' is that the triple t_{h+1} contradicts the matching M'. According to the way we construct the matching M', the symbols in t_{h+1} that also appear in $t_{h+1}^{(j)}$ cannot be in M'. Thus, the symbols in t_{h+1} that are in M' must correspond to the symbol '$*$' in $t_{h+1}^{(j)}$. Now we guess a symbol b_{h+1} in M' such that b_{h+1} is in t_{h+1} and corresponds to a '$*$' in $t_{h+1}^{(j)}$, and replace the corresponding symbol '$*$' in $t_{h+1}^{(j)}$ by b_{h+1}. This gives us a new pseudo-matching

$$M_{j+1} = \{t_1^{(j+1)}, t_2^{(j+1)}, \ldots, t_k^{(j+1)}\}$$

where $t_i^{(j+1)} = t_i^{(j)}$ for all $i \neq h+1$, and $t_{h+1}^{(j+1)}$ is the triple $t_{h+1}^{(j)}$ with a symbol '$*$' replaced by the symbol b_{h+1}.

Therefore, both pseudo-matchings M_j and M_{j+1} are the matching M_0 with some symbols replaced by the symbol '$*$'. Moreover, the pseudo-matching M_{j+1} has one less '$*$' than the pseudo-matching M_j. It is clear that the matching M_{j+1} can be constructed from the matching M_j in time $O(n)$ if the guessed symbol b_{h+1} is given. Now our algorithm applies the same process on the matching M_{j+1}.

Since we started with the matching M_2 with $2k$ '$*$' symbols and the above algorithm reduces the number of '$*$' symbols by one from M_j to M_{j+1}, the algorithm must end up with a matching M_g of k triples that contains no '$*$' symbols, where $g \leq 2k + 2$. This completes the description of our nondeterministic algorithm. Our nondeterministic algorithm runs in time $O(kn)$ if the guessed symbols are all given.

We point out that our nondeterministic algorithm reports a matching of k triples in S only if it actually finds a matching of k triples. Therefore, if a maximum matching in S contains less than k triples, then our nondeterministic algorithm will be stuck at some point without having a matching of k triples. An incorrect guess may also spoil the process. However, if the maximum matching in S has at least k triples, and if all our guesses in the process are correct, then the nondeterministic algorithm will produce a matching of k triples.

Now we explain how the nondeterminism in the above algorithm can be eliminated. Each guess in the algorithm corresponds to a sequence of nondeterministic binary bits. We first calculate how many nondeterministic binary bits are needed in the algorithm.

In constructing the pseudo-matching M_2 from the maximal matching M_1, we need guess k symbols in M_1. Since M_1 contains $3|M_1| < 3k$ symbols, k symbols in M_1 can be represented by a binary vector of length $3|M_1|$, in which exactly k bits are 1. Therefore, guessing k symbols in M_1 takes no more than $3k$ nondeterministic binary bits.

When we construct the pseudo-matching M_{j+1} from the pseudo-matching M_j, we need to guess the symbol b_{h+1} from the matching M'. First we need at most one nondeterministic binary bit to decide which '$*$' symbol in $t_{h+1}^{(j)}$ should be filled (recall that $t_{h+1}^{(j)}$ contains at most two '$*$' symbols). Once the '$*$' symbol in $t_{h+1}^{(j)}$ is decided, we only need to look at the triples in M' on the corresponding dimension. Since the matching M' contains less than k triples, M' contains less than k symbols in each dimension. Therefore, guessing a symbol in M' corresponding to the chosen '$*$' in $t_{h+1}^{(j)}$ is equivalent to deciding a position out of $|M'|$ positions. Thus, guessing the symbol b_{h+1} totally takes no more than $1 + \log k$ nondeterministic binary bits.

Since the nondeterministic algorithm ends up with a matching M_g, with $g \leq 2k + 2$, we conclude that the total number of nondeterministic binary bits used by the nondeterministic algorithm is bounded by (note that the pseudo-matching starts from M_2)

$$3k + 2k(1 + \log k) = k(5 + 2\log k)$$

To convert the nondeterministic algorithm into a deterministic algorithm, we run the nondeterministic algorithm using each of the $2^{k(5+2\log k)}$ binary vectors of length $k(5 + 2\log k)$ as the $k(5 + 2\log k)$ nondeterministic binary bits. Since for a fixed such binary vector, the algorithm runs in time $O(kn)$, we conclude that the running time of the resulting deterministic algorithm is bounded by $O(n2^{3k\log k})$.

Theorem 30.1 *There is an algorithm A such that given a set S of n triples and an integer k, the algorithm A runs in time $O(n2^{3k\log k})$, either finds a matching of k triples in S or reports no such a matching exists in S.*

Now we come back to approximation algorithms for the 3-D MATCHING problem. In the last lecture, we presented an algorithm `Apprx3D-First` that runs in linear time and constructs a maximal matching for a given set of triples. We proved that the number of triples in a maximal matching is at least 1/3 the number of triples in a maximum matching (Theorem 29.2). Thus, the algorithm `Apprx3D-First` is an approximation algorithm of approximation ratio 3 for the 3-D MATCHING problem. Now we present another polynomial time approximation algorithm with a better approximation ratio for the 3-D MATCHING problem.

Let S be a set of triples and let M be a maximal matching in S. Since the matching M is maximal, no triple in $S - M$ can be added directly to M to obtain a larger matching. However, it is possible that if we remove one triple from M, then we are able to add two triples from $S - M$ to M to obtain a larger matching. We say that the matching M is *1-optimal* if no such a triple in M exists. More formally, we say that a matching M is *1-optimal* if M is maximal and it is impossible to find a triple t_1 in M and two triples t_2 and t_3 in $S - M$ such that $M - \{t_1\} \cup \{t_2, t_3\}$ is a matching in S.

We present an algorithm that constructs a 1-optimal matching for a given set of triples.

Algorithm 30.1 `Apprx3D-Second`

```
Input:  a set $S$ of $n$ triples
Output:  a matching $M$ in $S$

1.  construct a maximal matching $M$ using Apprx3D-First;
2.  change = true;
3.  **while** change **do**
      **for** each triple $t$ in $M$ **do**
        $M = M - \{t\}$;
        let $S_t$ be the set of triples not contradicting $M$;
        construct a maximum matching $M_t$ in $S_t$;
        **if** $M_t$ contains more than one triple
        **then**  $M = M \cup M_t$;  change = true;
        **else** $M = M \cup \{t\}$;
```

Lemma 30.2 *After each execution of the* **for** *loop in step 3 of the algorithm* `Apprx3D-Second`*, the matching M is a maximal matching.*

PROOF. Before the algorithm enters step 3, the matching M is maximal.

Since the set S_t has no common symbol with the matching M after the triple t is removed from M, for any matching M' in S_t, $M \cup M'$ is a matching in S. Moreover, since all triples in $S - S_t$ contradict M, and all triples in

$S_t - M_t$ contradict M_t, we conclude that all triples in $S - (M \cup M_t)$ contradict $M \cup M_t$. That is, the matching $M \cup M_t$ is a maximal matching in S, which is assigned to M if M_t has more than one triple. In case M_t has only one triple, the triple t is put back to M, which by induction is also maximal. $\Box$

Lemma 30.3 *The matching constructed by the algorithm* `Apprx3D-Second` *is 1-optimal.*

PROOF. It is easy to see that there are a triple t in M and two triples t_1 and t_2 in $S - M$ such that $M - \{t\} \cup \{t_1, t_2\}$ is a matching in S if and only if the matching M_t in S_t contains more than one triple. Therefore, the algorithm `Apprx3D-Second` actually goes through all triples in M and checks whether each of them can be traded for more than one triple in $S - M$. In other words, the algorithm `Apprx3D-Second` ends up with a 1-optimal matching M. $\Box$

Lemma 30.4 *The maximum matching M_t in the set S_t can be constructed in time $O(n)$.*

PROOF. We first show that a maximum matching in S_t contains at most 3 triples. Suppose that t_1, t_2, t_3, and t_4 are four triples in a maximum matching in S_t. Then at least one of them, say t_1, contains no symbol in the triple t. Since t_1 does not contradict $M - \{t\}$, t_1 does not contradict M even before t is removed from M. Therefore, before the triple t is removed, the matching M is not maximal. This contradicts Lemma 30.2.

Therefore, a maximum matching in S_t contains at most 3 triples. Now according to Theorem 30.1, we can find a maximum matching M_t in S_t in time $O(n)$. $\Box$

Since each execution of the **while** loop in algorithm `Apprx3D-Second` increases the number of triples in M by at least 1 and a maximum matching in S contains at most n triples, we have the following theorem.

Theorem 30.5 *The algorithm* `Apprx3D-Second` *runs in time $O(n^3)$.*

We analyze the approximation ratio for the algorithm `Apprx3D-Second`.

Theorem 30.6 *The algorithm* `Apprx3D-Second` *has approximation ratio* 2.

PROOF. We denote by M the matching in S constructed by the algorithm `Apprx3D-Second` and let $M_{\max}$ be a maximum matching in S.

Based on the matchings M and $M_{\max}$, we introduce a weighting function $w(\cdot)$ on symbols in $X \cup Y \cup Z$ as follows.

- if a symbol a is not in both M and $M_{\max}$, then the symbol a has weight 0: $w(a) = 0$;
- if a symbol a is in both M and $M_{\max}$, and a is in a triple of $M_{\max}$ that contains two symbols not in M, then a has weight 1: $w(a) = 1$;
- if a symbol a is in both M and $M_{\max}$, and a is in a triple of $M_{\max}$ that contains only one symbol not in M, then a has weight 1/2: $w(a) = 1/2$;
- if a symbol a is in both M and $M_{\max}$, and a is in a triple of $M_{\max}$ that contains no symbol not in M, then a has weight 1/3: $w(a) = 1/3$;

The *weight* $w(t)$ *of a triple* t is the sum of the weights of its components. According to the definition, each triple in the matching $M_{\max}$ has weight exactly 1.

Now let $t = (a, b, c)$ be a triple in M. If $w(t) > 2$, then at least two components of t have weight 1. Without loss of generality, suppose that $w(a) = w(b) = 1$. By the definition, there are two triples $t_1 = (a, b', c')$ and $t_2 = (a'', b, c'')$ in the matching $M_{\max}$ such that the symbols b', c', a'', c'' do not appear in M. However, this would imply that $M - \{t\} \cup \{t_1, t_2\}$ is a matching and the matching M constructed by the algorithm `Apprx3D-Second` would not be 1-optimal. This contradicts Lemma 30.3.

Thus, each triple in the matching M has weight at most 2. Since only symbols in both matchings M and $M_{\max}$ have nonzero weight, we must have

$$\sum_{t \in M_{\max}} w(t) = \sum_{t \in M} w(t)$$

Since each triple in $M_{\max}$ has weight 1, we have $\sum_{t \in M_{\max}} w(t) = |M_{\max}|$. Moreover, since each triple in M has weight at most 2, we have $\sum_{t \in M} w(t) \leq 2|M|$. This gives us

$$|M_{\max}| \leq 2|M|$$

or $|M_{\max}|/|M| \leq 2$. This completes the proof. $\square$

Corollary 30.7 *The* 3-D MATCHING *problem has an approximation algorithm that runs in time* $O(n^3)$ *and has approximation ratio* 2.

Remark 30.1 A natural extension of the algorithm `Apprx3D-Second` is to consider 2-optimal, or in general k-optimal. That is, we construct a maximal matching M in S such that no k triples in M can be traded for $k+1$ triples in $S - M$. It is not very hard to see that a k-optimal matching in S can be constructed in polynomial time when k is a fixed constant. In fact, using Theorem 30.1, we can develop an algorithm of running time $O(n^{k+2})$ that constructs a k-optimal matching for a set S of n triples. We can show that a k-optimal matching gives an approximation ratio smaller than 2 when $k > 1$. For example, a 2-optimal matching has approximation ratio 9/5 while a 3-optimal matching has approximation ratio 5/3. It can also been shown that the approximation algorithm for the `3-D Matching` problem by constructing k-optimal matchings for a fixed constant k cannot have approximation ratio less than or equal to 3/2.

Course Project Problem: Develop an approximation algorithm for the `3-D Matching` problem that uses an approach different from the k-optimality method and has approximation ratio better than 2.

CPSC-669 Computational Optimization

Lecture #31, November 8, 1995

Lecturer: Professor Jianer Chen
Scribe: Shijin Lu
Revision: Jianer Chen

31 Maximum satisfiability

We have studied the VERTEX COVER problem and the 3-D MATCHING problem. These two problems have a common property that both of them have polynomial time approximation algorithms whose approximation ratio is bounded by a constant $c > 1$. It is unknown how close this constant c can be to the value 1. In particular, do these problems have a polynomial time approximation scheme? Very recent progress in computational optimization has shown that a large class of optimization problems of practical importance falls into this category. For this reason, we introduce another class of optimization problems.

Definition 31.1 An optimization problem is *approximable with a constant ratio in polynomial time* if it has a polynomial time approximation algorithm with approximation ratio c, where c is a fixed constant. Let APX be the class of all optimization problems approximable with a constant ratio in polynomial time.

It is clear that the class PTAS is a subclass of the class APX.

It is well-known that the SATISFIABILITY problem plays a fundamental role in the study of NP-completeness. An optimization version of the SATISFIABILITY problem, the MAX-SAT problem, plays a similar role in the study of the optimization class APX.

Let $X = \{x_1, \ldots, x_n\}$ be a set of boolean variables. A *literal* in X is either a boolean variable x_i or its negation $\overline{x_i}$, for some $1 \le i \le n$. A *clause* on X is an OR of a set of literals in X. The SATISFIABILITY problem is formally defined as follows.

SATISFIABILITY (SAT)

INPUT: a set of clauses $C_1, C_2, \ldots, C_m$ on $\{x_1, \ldots, x_n\}$

QUESTION: does there exist a truth assignment on $\{x_1, \ldots, x_n\}$ that satisfies all clauses?

By the famous Cook's Theorem, the SATISFIABILITY problem is NP-complete.

If we have further restrictions on the number of literals in each clause, we obtain another two interesting complexity classes.

> 3-SATISFIABILITY (3SAT)
>
> INPUT: a set of clauses $C_1, C_2, \ldots, C_m$ on $\{x_1, \ldots, x_n\}$ such that each clause has exactly 3 literals
>
> QUESTION: does there exist a truth assignment on $\{x_1, \ldots, x_n\}$ that satisfies all clauses?

> 2-SATISFIABILITY (2SAT)
>
> INPUT: a set of clauses $C_1, C_2, \ldots, C_m$ on $\{x_1, \ldots, x_n\}$ such that each clause has exactly 2 literals
>
> QUESTION: does there exist a truth assignment on $\{x_1, \ldots, x_n\}$ that satisfies all clauses?

It is well-known that the 3-SATISFIABILITY problem is still NP-complete, while the 2-SATISFIABILITY problem can be solved in polynomial time (in fact, in linear time).

An optimization version of the SATISFIABILITY problem can be defined as follows.

> MAX-SAT
>
> INPUT: a set of clauses $C_1, C_2, \ldots, C_m$ on $\{x_1, \ldots, x_n\}$
>
> OUTPUT: a truth assignment on $\{x_1, \ldots, x_n\}$ that satisfies the maximum number of the clauses

The optimization versions for the 3-SATISFIABILITY problem and for the 2-SATISFIABILITY problem are

> MAX-3SAT
>
> INPUT: a set of clauses $C_1, C_2, \ldots, C_m$ on $\{x_1, \ldots, x_n\}$ such that each clause has at most 3 literals
>
> OUTPUT: a truth assignment on $\{x_1, \ldots, x_n\}$ that satisfies the maximum number of the clauses

MAX-2SAT

INPUT: a set of clauses $C_1, C_2, \ldots, C_m$ on $\{x_1, \ldots, x_n\}$ such that each clause has at most 2 literals

OUTPUT: a truth assignment on $\{x_1, \ldots, x_n\}$ that satisfies the maximum number of the clauses

It is easy to see that the SATISFIABILITY problem can be reduced in polynomial time to the MAX-SAT problem: an instance $\{C_1, \ldots, C_m\}$ is a Yes-instance for the SATISFIABILITY problem if and only if when it is regarded as an instance of the MAX-SAT problem, its optimal value is m. Therefore, the MAX-SAT problem is NP-hard. Similarly, the 3-SATISFIABILITY problem can be reduced in polynomial time to the MAX-3SAT problem so that the MAX-3SAT problem is NP-hard.

Since the 2-SATISFIABILITY problem can be solved in linear time, one may expect that the corresponding optimization problem MAX-2SAT is also easy. However, the following theorem gives a bit surprising result.

Lemma 31.1 *The* MAX-2SAT *problem is NP-hard.*

PROOF. We show that the 3-SATISFIABILITY problem can be reduced in polynomial time to the MAX-2SAT problem.

Let $E = \{C_1, \ldots, C_m\}$ be an instance for the 3-SATISFIABILITY problem, where each C_i is a clause of three literals in $\{x_1, \ldots, x_n\}$. Consider the clause $C_i = (a_i \vee b_i \vee c_i)$, where a_i, b_i, and c_i are literals in $\{x_1, \ldots, x_n\}$. We construct ten clauses:

$$\begin{aligned} \overline{C_i} \quad = \quad & \{(a_i), \quad (b_i), \quad (c_i), \quad (y_i), \\ & (\overline{a_i} \vee \overline{b_i}), \quad (\overline{a_i} \vee \overline{c_i}), \quad (\overline{b_i} \vee \overline{c_i}), \\ & (a_i \vee \overline{y_i}), \quad (b_i \vee \overline{y_i}), \quad (c_i \vee \overline{y_i})\} \end{aligned} \tag{11}$$

where y_i is a new created boolean variable. It is easy to verify the following facts.

- if none of a_i, b_i, c_i is true, then any assignment to y_i can make at most six out of the ten clauses in (11) true;
- if one of a_i, b_i, c_i is true and two of a_i, b_i, c_i are false, then no assignment to y_i can make more than seven of the ten clauses in (11) true and there is an assignment to y_i that makes seven out of the ten clauses in (11) true;

- if two of a_i, b_i, c_i are true and one of a_i, b_i, c_i is false, then no assignment to y_i can make more than seven of the ten clauses in (11) true and there is an assignment to y_i that makes seven out of the ten clauses in (11) true;
- if all a_i, b_i, c_i are true, then no assignment to y_i can make more than seven of the ten clauses in (11) true and there is an assignment to y_i that makes seven out of the ten clauses in (11) true;

Based on the above analysis, we conclude that if we set any of the three literals a_i, b_i, and c_i true, then no assignment to y_i can make more than seven of the ten clauses in (11) true and there is an assignment to y_i that makes seven out of the ten clauses in (11) true, and if we set all three literals a_i, b_i, c_i false, then any assignment to y_i can make at most six out of the ten clauses in (11) true.

Now let $\overline{E}$ be the set of $10m$ clauses in $\overline{C_1}$, ..., $\overline{C_m}$, where each $\overline{C_i}$ is given as in (11). $\overline{E}$ is an instance for the MAX-2SAT problem. It is easy to see that the instance $\overline{E}$ can be constructed in polynomial time from the instance E.

Suppose that E is a Yes-instance for the 3-SATISFIABILITY problem. Then there is an assignment S_x to $\{x_1, \ldots, x_n\}$ that makes at least one literal in each C_i of the clauses C_1, ..., C_m true. According to the analysis given above, this assignment S_x together with a proper assignment S_y to $\{y_1, \ldots, y_m\}$ will make seven out of the ten clauses in $\overline{C_i}$ true, for all $i = 1, \ldots, m$. Therefore, the assignment $S_x + S_y$ to the boolean variables $\{x_1, \ldots, x_n, y_1, \ldots, y_m\}$ makes $7m$ clauses in $\overline{E}$ true. This gives $Opt(\overline{E}) \geq 7m$.

Now suppose that E is a No-instance for the 3-SATISFIABILITY problem. Let $\overline{S}$ be an assignment to $\{x_1, \ldots, x_n, y_1, \ldots, y_m\}$ and we analyze how many clauses in $\overline{E}$ the assignment $\overline{S}$ can satisfy. The assignment $\overline{S}$ can be decomposed into an assignment S_x to $\{x_1, \ldots, x_n\}$ and an assignment S_y to $\{y_1, \ldots, y_m\}$. Since E is a No-instance for the 3-SATISFIABILITY problem, for at least one clause C_i in E, the assignment S_x makes all literals false. According to our previous analysis, any assignment to y_i together with the assignment S_x can make at most six out of the ten clauses in $\overline{C_i}$ true. Moreover, since no assignment to $\{x_1, \ldots, x_n, y_1, \ldots, y_m\}$ can make more than seven clauses in each $\overline{C_j}$ true, we conclude that the assignment $\overline{S}$ can make at most $7(m-1) + 6 = 7m - 1$ clauses in $\overline{E}$ true. Since $\overline{S}$ is arbitrary, we conclude that in this case, no assignment to $\{x_1, \ldots, x_n, y_1, \ldots, y_m\}$ can make more than $7m - 1$ clauses in $\overline{E}$ true.

Summarizing the discussion above, we conclude that E is a Yes-instance for the 3-SATISFIABILITY problem if and only if the optimal value $Opt(\overline{E})$ for the instance $\overline{E}$ for the MAX-2SAT problem is equal to $7m$. Consequently, the 3-SATISFIABILITY problem can be reduced in polynomial time to the MAX-2SAT problem. This completes the proof that the MAX-2SAT problem is NP-hard. $\square$

Now we describe an approximation algorithm for the MAX-SAT problem. Consider the following algorithm. For each clause, we give it a weight $w(C_i)$. We use $|C_i|$ to denote the number of literals in the clause C_i.

Algorithm 31.1 `ApprxMaxSat`

```
Input:  a set of clauses $\{C_1, \ldots, C_m\}$ on $\{x_1, \ldots, x_n\}$
Output:  a truth assignment to $\{x_1, \ldots, x_n\}$

1.  LEFT $= \{C_1, \ldots, C_m\}$;
2.  for each clause $C_i$ do
      $w(C_i) = 1/2^{|C_i|}$
3.  for $i = 1$ to $n$ do
      find all clauses $C_1^t, \ldots, C_r^t$ in LEFT that contain $x_i$;
      find all clauses $C_1^f, \ldots, C_s^f$ in LEFT that contain $\overline{x_i}$;
      if $\sum_{i=1}^r w(C_i^t) > \sum_{j=1}^s w(C_j^f)$
      then  $x_i = 1$
            delete $C_1^t, \ldots, C_r^t$ from LEFT;
            for $j = 1$ to $s$ do  $w(C_j^f) = 2w(C_j^f)$
      else  $x_i = 0$
            delete $C_1^f, \ldots, C_r^f$ from LEFT;
            for $j = 1$ to $r$ do  $w(C_j^t) = 2w(C_j^t)$
```

The algorithm `ApprxMaxSat` runs in polynomial time. Now we analyze the approximation ratio for the algorithm.

Lemma 31.2 *If each of the input clauses $\{C_1, \ldots, C_m\}$ contains at least k literals, then the algorithm* `ApprxMaxSat` *constructs an assignment that satisfies at least $m(1 - 1/2^k)$ of the clauses.*

PROOF. In the algorithm `ApprxMaxSat`, once a clause is satisfied, the clause is deleted from the set `LEFT`. Therefore, the number of clauses that are not satisfied by the constructed assignment is equal to the number of clauses left in the set LEFT at the end of the algorithm. We calculate the number of clauses in LEFT using the weighting function $w(\cdot)$.

Initially, each clause C_i has weight $1/2^{|C_i|}$. By our assumption, the clause C_i contains at least k literals. So we have

$$\sum_{C_i \in \mathtt{LEFT}} w(C_i) = \sum_{C_i \in \mathtt{LEFT}} 1/2^{|C_i|} \leq \sum_{C_i \in \mathtt{LEFT}} 1/2^k = m/2^k$$

In step 3, we update the set LEFT and the weight for the clauses in LEFT. It can be easily seen that we never increase the value $\sum_{C_i \in \mathtt{LEFT}} w(C_i)$ — each time we update the set LEFT, we delete a heavier set of clauses in LEFT and double the weight for a lighter set of clauses in LEFT. Therefore, at end of the algorithm, we should have

$$\sum_{C_i \in \mathtt{LEFT}} w(C_i) \leq m/2^k \tag{12}$$

At the end of the algorithm, all boolean variables $\{x_1, \ldots, x_n\}$ have been assigned a value. A clause C_i in the set LEFT has been considered by the algorithm exactly $|C_i|$ times and each time the corresponding literal in C_i was assigned 0. Therefore, for each literal in C_i, the weight of the clause C_i is doubled once. Since initially the clause C_i has weight $1/2^{|C_i|}$ and its weight is doubled exactly $|C_i|$ times in the algorithm, we conclude that at the end of the algorithm, the clause C_i in LEFT has weight 1. Combining this with the inequality (12), we conclude that at the end of the algorithm, the number of clauses in the set LEFT is no more than $m/2^k$. In other words, the number of clauses satisfied by the constructed assignment is at least $m(1 - 1/2^k)$. The lemma is proved. □

Theorem 31.3 *For an input of m clauses each containing at least $k > 0$ literals, the algorithm* ApprxMaxSat *constructs an assignment with approximation ratio $1 + 1/(2^k - 1)$. In particular, the algorithm* ApprxMaxSat *is an approximation algorithm with approximation ratio 2 for the* Max-Sat *problem.*

PROOF. According to Lemma 31.2, on an input of m clauses each containing at least k literals, the algorithm ApprxMaxSat constructs an assignment that satisfies at least $m(1 - 1/2^k)$ clauses. Since no assignment can satisfy more than m clauses, the approximation ratio must be bounded by

$$\frac{m}{m(1 - 1/2^k)} = 1 + \frac{1}{2^k - 1}$$

Since for each input instance for the MAX-SAT problem, each clause contains at least 1 literal, the second statement in the theorem follows directly. $\square$

Remark 31.2 The approximation ratio 2 for the MAX-SAT problem is due to a classical work of David Johnson about 20 years ago. The bound 2 stood for more than 20 years until recently, Yannakakis developed a polynomial time approximation algorithm with approximation ratio 4/3 for the MAX-SAT problem.

CPSC-669 Computational Optimization

Lecture #32, November 10, 1995

Lecturer: Professor Jianer Chen
Scribe: Hao Zheng
Revision: Jianer Chen

32 Probabilistically Checkable Proofs

We have seen a number of optimization problems that are in APX, that is, that have polynomial time approximation algorithms with approximation ratio bounded by a constant: the Δ-TRAVELING SALESMAN problem can be approximated in polynomial time with approximation ratio 1.5, the VERTEX COVER problem can be approximated in polynomial time with approximation ratio 2, the 3-D MATCHING problem can be approximated in polynomial time with approximation ratio 2, and the MAX-SAT problem can be approximated in polynomial time with approximation ratio 2. The ratios for the first two problems are still the best results known today, and the ratios for the last two problems have be somehow improved recently to a constant $c > 1$ ($c = 1.5 + \epsilon$ for the 3-D MATCHING problem and $c = 1.325$ for the MAX-SAT problem). The question is whether further improvement on the approximation ratio is possible. In particular, how close can this approximation ratio be to the value 1? Can they have a polynomial time approximation scheme?

The questions turn out to be very deep in the study of computational optimization. We will see later that from a viewpoint of complexity theory, these questions are equivalent to the famous P = NP problem. Moreover, our algorithmic practice also suggests the possibility for either directions. Take the Δ-TRAVELING SALESMAN problem as an example. It has been more than 15 years that the bound 1.5 on the approximation ratio has stood for the problem. On the other hand, very recent research (still in manuscript version) has shown that the GRAPH TRAVELING SALESMAN problem, in which the distance metric between two vertices is the shortest path metric of an unweighted graph, has a polynomial time approximation scheme. Note that the GRAPH TRAVELING SALESMAN problem is a restricted version of the Δ-TRAVELING SALESMAN problem. From this progress, researchers have even conjectured that the EUCLIDEAN TRAVELING SALESMAN problem, which seems the most naturally restricted version of the Δ-TRAVELING SALESMAN problem, has a polynomial time approximation scheme.

Researchers were not able to answer these questions for more than 20 years until a very recent breakthrough in complexity theory that gives a new characterization of the complexity class NP. In the rest of this lecture, we will describe this new characterization.

We need to review a few fundamental definitions in complexity theory.

Definition 32.1 A *language* L is a subset of Σ^*, where Σ is a fixed alphabet. With a proper coding scheme, we can assume that $\Sigma = \{0, 1\}$. For an instance $x \in \Sigma^*$, if $x \in L$ then we say that x is a Yes-instance of L while if $x \notin L$ then we say that x is a No-instance of L.

A language L is also called a "decision problem" in which for each instance x, we need to decide a "Yes/No" conclusion for the question "$x \in L$?"

Definition 32.2 A language L is *accepted* by an algorithm A if on any input instance $x \in \Sigma^*$, the algorithm A outputs "Yes" if $x \in L$ (or we say that A *accepts* x), and "No" if $x \notin L$ (or we say that A *rejects* x).

Definition 32.3 An algorithm A is *nondeterministic* if it works as follows: on an input instance $x \in \Sigma^*$, the algorithm A is also provided with another "guessed" string $y_x \in \Sigma^*$ (by some magic way). Thus, the algorithm A can work on x with the "hints" given in the guessed string y. The nondeterministic algorithm A *accepts* a language L if for each $x \in L$, there is a guessed string y_x such that A accepts x when y_x is provided, and for each $x \notin L$, the algorithm A rejects x for any guessed string y.

A nondeterministic algorithm A *runs in polynomial time* if the running time of A is bounded by a polynomial of the input length $|x|$. Note that the time complexity of a nondeterministic algorithm is not measured in terms of the guessed string y. Since a polynomial time nondeterministic algorithm A can read at most polynomial many bits in y, we can assume, without loss of generality, that the length of the guessed string y is bounded by a polynomial of the input length $|x|$.

We say that an algorithm A *reduces* a language L_1 to another language L_2 if on any input instance x_1 for L_1, the algorithm A produces an input instance x_2 for L_2 such that $x_1 \in L_1$ if and only if $x_2 \in L_2$.

Definition 32.4 A language L is in NP if it is accepted by a polynomial time nondeterministic algorithm. A language L is NP-complete if L is in NP and for each language L' in NP, there is a polynomial time algorithm that reduces L' to L.

Take the SATISFIABILITY problem as an example. Given a set x of clauses C_1, ..., C_m on $\{x_1, \ldots, x_n\}$, a polynomial time nondeterministic algorithm A can work as follows: A interprets the first n binary bits in the guessed string y as a truth assignment to the boolean variables $\{x_1, \ldots, x_n\}$ and replaces each literal in the clauses by the corresponding boolean value. A accepts x if all clauses are evaluated true on this assignment, otherwise A rejects x. It is easy to see that if x is a Yes-instance for the SATISFIABILITY problem, then for the guessed string y_x whose first n bits give the assignment that satisfies x, the algorithm A will accept. On the other hand, if x is a No-instance for the SATISFIABILITY problem, then no matter which guessed string y is provided, the algorithm A will reject anyway since no assignment on $\{x_1, \ldots, x_n\}$ can satisfy all the clauses in x. Therefore, the SATISFIABILITY problem is in NP. By the famous Cook's theorem, the SATISFIABILITY problem is actually NP-complete.

A nondeterministic algorithm A accepting a language L can also be interpreted as a proof system. The given input instance x can be regarded as a statement of a theorem "the string x is in the language L," while the guessed string y can be regarded as a proof for the theorem. The algorithm A is a very trusty "verifier", who may not be able to derive a proof for the theorem x, but can verify whether y is a valid proof for the theorem x. Therefore, if the theorem x is true and the guessed string y is a valid proof for the theorem x, then the algorithm A will say "Yes", and if the theorem x is not true then the algorithm A will disprove any pseudo-proof y and say "No". In this sense, each problem in NP is a set of theorems that have valid proofs that are "easily checkable", i.e., that can be checked in polynomial time.

An interesting question is how many bits of the proof y a polynomial time nondeterministic algorithm needs to read in order to verify the theorem x. In real life, it seems that most of the theorems simply need a single "hint" and the other parts of the proof can be easily derived from the hint. Is this also true for the problems in NP? For this, we introduce the following definition.

Definition 32.5 A language L is in the class $PCP(0, b(n))$ if L is accepted by a polynomial time nondeterministic algorithm A such that on each input instance x, the algorithm A reads at most $O(b(|x|)$ bits from the guessed string y.

We have the following easy observations.

Lemma 32.1 *Every language in NP is in the class PCP$(0, n^c)$ for some constant c.*

PROOF. Suppose that a language L is in NP. Then L is accepted by a polynomial time nondeterministic algorithm A. Let the running time of the algorithm A be $O(n^c)$. Then the algorithm A on input instance x reads at most $O(|x|^c)$ bits from a guessed string y. That is, the language L is in the class PCP$(0, n^c)$. $\Box$

It is interesting to ask whether it is possible to have a polynomial time algorithm that accepts a language in NP, in particular an NP-complete language, by reading less than $\Omega(n)$ bits from the guessed string y. The conjecture is No. However, it seems that our current knowledge is still far from a formal proof of this conjecture. A (much) weaker result can be formally proved: if a language is in PCP$(0, \log\log n)$, then it is in P.

Another extension of our deterministic algorithms is *probabilistic algorithms*, defined as follows.

Definition 32.6 An algorithm A is a *probabilistic algorithm* if on any input instance x, the algorithm A first generates a random string r, then deterministically works on the input x.

If the outcome of a probabilistic algorithm does not depend on the generated random string, then the probabilistic algorithm is just a normal deterministic algorithm. If the computation of the probabilistic algorithm does depend on the generated random string, then each outcome of the computation will happen with a certain probability. Some very interesting practical problems can be solved by probabilistic algorithms in such a way that correct solutions are produced by the algorithm with very high probability.

If we allow both probabilism and nondeterminism, we obtain the following class.

Definition 32.7 A language L is in the class *PCP$(r(n), b(n))$* if it is accepted by a polynomial time algorithm A with two constants c and d such that

1. On an input instance x, the random string generated by the algorithm A is of length $c \cdot r(|x|))$, a guessed string y of length $O(|x|^d)$ is provided, and the algorithm A reads at most $O(b(|x|))$ bits from a guessed string;

2. For each input instance $x \in L$, there is a guessed string y_x of length $O(|x|^d)$ such that the algorithm A accepts x with probability 1 (i.e., A accepts x based on y_x for every generated random string of length $c \cdot r(|x|))$;

3. For each input instance $x \notin L$, on any guessed string y of length $O(|x|^d)$, the algorithm A rejects x with probability at least 1/2 (i.e., A rejects x based on y for at least half of the generated random strings of length $c \cdot r(|x|))$.

The algorithm A is called a *PCP*$(r(n), b(n))$ *system* accepting the language L.

The name "PCP" here refers to the "probabilistically checkable proof" as the model involves a checkable proof system (i.e., guessed strings) and probabilistic computation (i.e., the random string generation).

It was a very active research topic that for what functions $r(n)$ and $b(n)$, the class PCP$(r(n), b(n))$ precisely describes the class NP. The question was eventually settled down recently, as a result stated as follows.

Theorem 32.2 *A language L is in the class NP if and only if L is in the class PCP*$(\log n, 1)$.

The current proof for Theorem 32.2 is rather involved. It borrows significantly from results on polynomial checking, proof verification, program result checking, and coding theory. Giving the details of these results goes far beyond the scope of this course and we refer the interested students to the papers that originate them (talk to the instructor).

CPSC-669 Computational Optimization

Lecture #33, November 13, 1995

Lecturer: Professor Jianer Chen
Scribe: Hao Zheng
Revision: Jianer Chen

33 MAX-3SAT has no PTAS

In the previous lecture, we defined the PCP systems. The following theorem was stated.

Theorem 33.1 *A language L is in the class NP if and only if L is accepted by a PCP*$(\log n, 1)$ *system.*

An outstanding application of Theorem 33.1 is a proof that many optimization problems, such as MAX-3SAT, have no polynomial time approximation scheme unless P = NP. Before we present the proof, we first make a closer look at the PCP systems. This investigation should let us have a better understanding on the PCP systems.

By the definition, a PCP$(\log n, 1)$ system is a polynomial time algorithm A that on input of length n, generates a random string of length $O(\log n)$ and reads at most b bits from the guessed string y, where b is a fixed constant. It should be noted that which b bits of the guessed string y are read by the algorithm A may depend on the values of the bits read from y. For example, suppose that the algorithm A reads the first bit from the guessed string y. Now the algorithm A may calculate the address of the second bit to be read from y based on the value of the first bit. In general, the address of the ith bit to be read by A from y may depend on the values of the first $i-1$ bits read by A.

Definition 33.1 A PCP$(r(n), b(n))$ system A is *nonadaptive* if on an input instance x and a fixed randomly generated string of length $O(r(|x|))$, the addresses of the $O(b(n))$ bits to be read by A from the guessed string y are independent of the content of the guessed string y.

Therefore, the process of a nonadaptive PCP$(r(n), b(n))$ system A can be regarded as follows: on input instance x, the polynomial time algorithm

A first generates a random string of length $O(r(|x|))$, then generates the $O(b(n))$ addresses for the bits to be read from the guessed string y, and then reads the bits from y. There will be no other computation performed during the reading of the bits from the guessed string y.

Lemma 33.2 *A language L is accepted by a PCP$(\log n, 1)$ system if and only if it is accepted by a nonadaptive PCP$(\log n, 1)$ system.*

PROOF. By the definition, if the language L is accepted by a nonadaptive PCP$(\log n, 1)$ system, then L is accepted by a general PCP$(\log n, 1)$ system.

Now suppose that the language L is accepted by a PCP$(\log n, 1)$ system A_1. By the definition, on each input instance x, the polynomial time algorithm A_1 first generates a random string R of length $O(\log n)$, then works on x based on R and a guessed string y. The algorithm A_1 reads at most b bits from the guessed string y, where b is a fixed constant. If $x \in L$, then there is a guessed string y_x such that on all randomly generated strings R of length $O(\log n)$, A_1 accepts x based on y_x; and if $x \notin L$, then for any guessed string y, for at least half of the randomly generated strings of length $O(\log n)$, the algorithm A_1 rejects x based on y.

We construct a nonadaptive PCP$(\log n, 1)$ system A_2 that accepts the language L.

Let x be an input instance of L. Fix a randomly generated string R of length $O(\log n)$. The algorithm A_2 works as follows. A_2 first enumerates all the 2^b boolean vectors of length b. Note that each boolean vector $(v_1, \ldots, v_b)$ of length b gives a possible set of values for the b bits to be read from the guessed string y. For the fixed input instance x and the fixed random string R, the boolean vector $(v_1, \ldots, v_b)$ also uniquely determines the addresses of the bits to be read by the algorithm A_1 from the guessed string y: the address of the first bit depends only on x and R, the address of the second bit depends on x, R, and the value of the first bit, which is supposed to be v_1, and the address of the third bit depends on x, R, and the values of the first two bits, which are supposed to be v_1 and v_2, respectively, and so on. Therefore, based on the input instance x, the random string R, and the given boolean vector $(v_1, \ldots, v_b)$ of length b, the algorithm A_2 can uniquely determine the addresses of the b bits to be read by the algorithm A_1. The algorithm A_2 simulates the algorithm A_1 on this computation, records the b addresses on the guessed string, and records the decision of A_1 on x and R based on this boolean vector. Note that so far, no bits have been actually read from the guessed string y, the values of the bits on the guessed string

y are assumed in the boolean vector $(v_1, \ldots, v_b)$.

The algorithm A_2 performs the above operation on each of the 2^b boolean vectors of length b. At the end, the algorithm A_2 has recorded $d \leq b2^b$ addresses on the guessed string y. Now the algorithm A_2 reads all these d bits at once from the guessed string y. With these d values available, the algorithm A_2 can easily decide precisely which boolean vector gives the correct sequence of values of the bits read from y by the algorithm A_1 on x and R. Note that there is exactly one such vector. With this correct boolean vector, now the algorithm A_2 can find out whether the algorithm A_1 accepts x on the random string R and the guessed string y. The algorithm A_2 accepts x on R if and only if the algorithm A_1 accepts x on R.

Now we can describe the algorithm A_2 in a complete version. Given an input instance x for L, the algorithm A_2 simulates the algorithm A_1 by first generating a random string R of length $O(\log n)$. Then, as described above, the algorithm A_2 simulates the algorithm A_1 on x and R, and A_2 accepts if and only if A_1 accepts.

By the construction, the algorithm A_2 is clearly nonadaptive. The number of bits read by A_2 from the guessed string y is bounded by $d \leq b2^b$, which is still a constant. The running time of A_2 is bounded by $b2^b$ times the running time of the algorithm A_1. Thus, A_2 is also a polynomial time algorithm. Finally, on any input instance x, any randomly generated string R, and any guessed string y, the algorithm A_2 makes the same decision as the algorithm A_1. Thus, the algorithm A_2 is a nonadaptive PCP$(\log n, 1)$ system that accepts the language L. $\square$

Now we are ready for our main theorem in this lecture.

Theorem 33.3 *The* MAX-3SAT *problem has no polynomial time approximation scheme unless $P = NP$.*

PROOF. Let L be any language that is NP-complete. We show that if the MAX-3SAT problem has a polynomial time approximation scheme, then the language L can be solved in polynomial time by a deterministic algorithm, which implies that P = NP.

Since the language L is in NP, by Theorem 33.1, L is accepted by a PCP$(\log n, 1)$ system A. Without loss of generality, we assume that on an input instance x, the polynomial time algorithm A generates a random string R of length $c \log n$, and reads at most b bits from a guessed string y, where c and b are fixed constants. If the input instance x is in L, then there is a guessed string y_x of length n^d such that for all randomly generated strings R of length $c \log n$, the algorithm A accepts x; if the input instance x is not in L, then for any guessed string y, the algorithm A rejects x on at least half of the randomly generated strings R of length $c \log n$. According to Lemma 33.2, we can assume that the PCP($\log n, 1$) system A is nonadaptive.

Let x be a given input instance of L. Fix a random string R of length $c \log n$. We simulate the algorithm A on the input x and the random string R. Note that the outcome of the algorithm A on input x and the random string R depends on the values of the b bits to be read from the guessed string y. Since the algorithm A is nonadaptive, the addresses i_1, ..., i_b of the b bits to be read from the guessed string y can be computed without knowing the actual content of the guessed string y. Formally, the outcome of the algorithm A on input x and the random string R is a boolean function of b boolean variables:

$$F_{x,R}(y_{i_1}, \ldots, y_{i_b})$$

where y_{i_j} stands for the i_jth bit of the guessed string y. The boolean function $F_{x,R}$ can be constructed by simulating the algorithm A on input x and the random string R and on each of the possible assignments of y_{i_1}, ..., y_{i_b}. Note that there are only 2^b different possibilities. Now convert the function $F_{x,R}(y_{i_1}, \ldots, y_{i_b})$ into the conjunctive normal form. Note that in the conjunctive normal form, each clause has at most b literals and there are at most 2^b clauses. Now for each clause $C = (z_1 \vee \ldots \vee z_a)$ containing more than 3 literals, we use the standard transformation, by introducing $a - 3$ new variables w_1, ..., w_{a-3}, to convert it into a set of $a - 3$ clauses of 3 literals:

$$\begin{aligned} &(z_1 \vee z_2 \vee w_1) \wedge (\overline{w_1} \vee z_3 \vee w_2) \wedge (\overline{w_2} \vee z_4 \vee w_3) \wedge \\ &\wedge \cdots \wedge (\overline{w_{a-4}} \vee z_{a-2} \vee w_{a-3}) \wedge (\overline{w_{a-3}} \vee z_{a-1} \vee z_a) \end{aligned}$$

After this transformation, the boolean function $F_{x,R}$ has been converted into a set $S_{x,R}$ of clauses, in which each clause contains at most 3 literals. One important fact is that the number of clauses contained in $S_{x,R}$ is bounded by $b2^b$, a constant independent of the input instance x. Note that there is an assignment to the boolean variables $y_{i_1}, \ldots, y_{i_b}$ that makes the function $F_{x,R}$ true if and only if there is an assignment to the variables in $S_{x,R}$ that satisfies all clauses in $S_{x,R}$. It is also clear that the set $S_{x,R}$ can be constructed from the input x and the random string R in polynomial time.

Now for each random string R of length $c \log n$, we construct the set $S_{x,R}$ of clauses that contain at most 3 literals. The union of all these sets gives us an input instance for the MAX-3SAT problem:

$$S(x) = \bigcup_{|R| = c\log n} S_{x,R}$$

Since there are totally $2^{c\log n} = n^c$ binary strings of length $c\log n$, we can rename the subsets $S_{x,R}$ in $S(x)$ as S_1, ..., S_m, where $m = n^c$:

$$S(x) = S_1 \cup S_2 \cup \cdots \cup S_m$$

Since each set S_i can be constructed in polynomial time and $m = n^c$, the set $S(x)$ can be constructed from the input instance x in polynomial time. Moreover, there is a constant $h \leq b2^b$ such that each subset S_i contains at most h clauses. Suppose that the set $S(x)$ has N clauses. Then $N \leq hm$.

If the input x is in the language L, then according to the definition of the PCP($\log n, 1$) system A, there is a guessed string y_x of length n^d such that on every randomly generated string R, the algorithm A accepts x. Consequently, if we let this y_x be the assignment on the variables in the set $S_{x,R}$, then all clauses in $S_{x,R}$ are satisfied. That is, if we regard $S(x)$ as an input instance of the MAX-3SAT problem, then the optimal value $Opt(S(x))$ of $S(x)$ is N.

If the input instance x is not in the language L, then given any guessed string y, for at least half of the random strings R of length $c\log n$, the algorithm A rejects x. That is, there are at least $m/2$ of the subsets S_1, ..., S_m in $S(x)$, for which the assignment y cannot satisfy all clauses in the subset. Therefore, on any assignment to the variables in the set $S(x)$, at least $m/2 \geq N/(2h)$ clauses in the set $S(x)$ are not satisfied.

In conclusion, the set $S(x)$ is an input instance of the MAX-3SAT problem with the following properties:

> *Either $x \in L$ and there is an assignment to the boolean variables in $S(x)$ that satisfies all clauses in $S(x)$, or $x \notin L$ and no assignment to the boolean variables in $S(x)$ can satisfy more than $N(1 - 1/(2h))$ clauses in $S(x)$, where $h > 0$ is a fixed constant. Moreover, the set $S(x)$ of clauses can be constructed from x in polynomial time.*

Now it is straightforward to prove the theorem. Suppose that the MAX-3SAT problem has a polynomial time approximation scheme. Then let A' be a polynomial time approximation algorithm with approximation ratio $1 + 1/(4h)$ for the MAX-3SAT problem. We describe a polynomial time deterministic algorithm A_0 that accepts the language L. Given an input instance x, the algorithm A_0 first constructs the instance $S(x)$ in polynomial time for the Max-3Sat problem. Suppose that $S(x)$ has N clauses. The algorithm A_0 then applies the polynomial time approximation algorithm A' on $S(x)$ to produce a solution s for $S(x)$, where s is an assignment to the boolean variables in $S(x)$. The algorithm A_0 accepts x if and only if the assignment s satisfies more than $N(1 - 1/(2h))$ clauses in $S(x)$.

It is clear that the algorithm A_0 is a polynomial time deterministic algorithm. We prove that the algorithm A_0 accepts precisely the language L. In case $x \in L$, by the above analysis, $Opt(S(x)) = N$. Since the approximation algorithm A' has approximation ratio $1 + 1/(4h)$, the assignment s produced by the algorithm A' must satisfy at least

$$\frac{N}{1 + 1/(4h)} > N(1 - 1/(2h))$$

clauses in $S(x)$. In this case, the algorithm A_0 accepts x. On the other hand, if $x \notin L$, then by the above analysis, no assignment to the variables in $S(x)$ can satisfy more than $N(1-1/(2h))$ clauses in $S(x)$. In particular, the assignment s produced by the algorithm A' satisfies no more than $N(1 - 1/(2h))$ clauses in $S(x)$. Thus, in this case, the algorithm A_0 rejects x. This proves that the polynomial time algorithm A_0 accepts precisely the language L. We conclude that the NP-complete problem L is accepted by a polynomial time deterministic algorithm. Consequently, P = NP.

This completes the proof. $\square$

CPSC-669 Computational Optimization

Lecture #34, November 15, 1995

Lecturer: Professor Jianer Chen
Scribe: Hao Zheng
Revision: Jianer Chen

34 INDEPENDENT SET has no PTAS

The famous Cook theorem that the SATISFIABILITY problem is NP-complete serves as a fundamental theorem for the study of NP-completeness of decision problems and gives the first NP-complete problem. Because of this first NP-complete problem, the proofs for the NP-completeness of other decision problems become much simpler by means of a proper "reduction" from the SATISFIABILITY problem.

Theorem 33.3 plays the same role in the study of approximability of optimization problems as does Cook's theorem in the study of NP-completeness. By Theorem 33.3, the hardness of approximability for the MAX-3SAT problem is established. The hardness of approximability for other optimization problems now can be established from the MAX-3SAT problem by a proper reduction that preserves the approximability. We will demonstrate a few such reductions in this lecture. A formal definition of such a reduction among optimization problems will be given in the next lecture.

The first reduction is straightforward.

Lemma 34.1 *If the* MAX-SAT *problem has a polynomial time approximation scheme, then so does the* MAX-3SAT *problem.*

PROOF. Each instance x of the MAX-3SAT problem is also an instance for the MAX-SAT problem. Therefore, any approximation algorithm for the MAX-SAT problem is also an approximation algorithm for the MAX-3SAT problem with the same approximation ratio. Therefore, if the MAX-SAT problem has a polynomial time approximation scheme, then so does the MAX-3SAT problem. □

Theorem 34.2 *The* MAX-SAT *problem has no polynomial time approximation scheme unless* $P = NP$.

PROOF. This follows directly from Theorem 33.3 and Lemma 34.1. □

Now let us consider a less simple reduction. Let G be a graph. Recall that an *independent set* in G is a subset S of vertices in G in which no two vertices are adjacent. The INDEPENDENT SET problem is defined as follows.

INDEPENDENT SET

INPUT: a graph G

OUTPUT: an independent set S of G with the cardinality of S maximized

We present a reduction from the MAX-3SAT problem to the INDEPENDENT SET problem.

The reduction is the one that is used in the NP-completeness theory to show that the decision version of the INDEPENDENT SET is NP-complete. However, we need a more careful quantitative analysis.

Given an instance $E = \{C_1, C_2, \ldots, C_m\}$ of the MAX-3SAT problem, where each C_i is a clause of at most 3 literals in $\{x_1, \ldots, x_n\}$. We construct a graph G_E as follows.

Every literal occurrence l in a clause C_i in E induces a vertex in the graph G_E, which will be named by $l^{(i)}$. Note that if the same literal appears in two different clauses in E, then there will be two corresponding vertices in the graph G_E. For any pair of vertices $l_1^{(i)}$ and $l_2^{(j)}$, there is an edge connecting them if and only if either

1. $i = j$, i.e., the literals $l_1^{(i)}$ and $l_2^{(j)}$ belong to the same clause in E; or
2. $i \neq j$ and $\overline{l_1^{(i)}} = l_2^{(j)}$, i.e., the literals $l_1^{(i)}$ and $l_2^{(j)}$ belong to different clauses in E and they negate each other.

This completes the description of the graph G_E.

Lemma 34.3 *If α is an assignment to the variables $\{x_1, \ldots, x_n\}$ that satisfies k clauses in E, then an independent set S_α of at least k vertices in the graph G_E can be constructed in polynomial time based on the assignment α.*

PROOF. Without loss of generality, suppose that the assignment α to the variables $\{x_1, \ldots, x_n\}$ satisfies the k clauses C_1, ..., C_k in E. Then under this assignment α, each clause C_i, $i = 1, \ldots, k$, has (at least) one literal $l^{(i)}$ that is set to true by α. We claim that the subset $S_\alpha = \{l^{(1)}, \ldots, l^{(k)}\}$ of vertices in the graph G_E forms an independent set. In fact, for any pair of vertices $l^{(i)}$ and $l^{(j)}$, $1 \leq i, j \leq k$, $i \neq j$, since the literals $l^{(i)}$ and $l^{(j)}$ belong to different clauses C_i and C_j in E, the vertices $l^{(i)}$ and $l^{(j)}$ are adjacent in the graph G_E only if the literal $l^{(i)}$ is the negation of the literal $l^{(j)}$. Thus, any assignment will set one of the literals $l^{(i)}$ and $l^{(j)}$ true and the other false. By our assumption, the assignment α sets both the literals $l^{(i)}$ and $l^{(j)}$ true. Thus, the literal $l^{(i)}$ cannot be the negation of the literal $l^{(j)}$. In consequence, the vertices $l^{(i)}$ and $l^{(j)}$ in the graph G_E are not adjacent. This proves that the set $S_\alpha = \{l^{(1)}, \ldots, l^{(k)}\}$ is an independent set in the graph G_E. It is easy to see that the independent set $S_\alpha = \{l^{(1)}, \ldots, l^{(k)}\}$ in the graph G_E can be constructed in linear time when the assignment α to the variables $\{x_1, \ldots, x_n\}$ is given. $\Box$

Lemma 34.4 *If the graph G_E has an independent set S of k vertices, then an assignment α_S to the variables $\{x_1, \ldots, x_n\}$ can be constructed in polynomial time such that α_S satisfies at least k clauses in E.*

PROOF. Let $S = \{l_1, \ldots, l_k\}$ be an independent set in the graph G_E. Since no two vertices in S are adjacent, by the construction of the graph G_E,

1. no two literals l_i and l_j in the set S belong to the same clause in E; and

2. no two literals l_i and l_j in the set S negate each other. Thus, for each variable x_h, at most one of x_h and $\overline{x_h}$ is in S.

Thus, the set $S = \{l_1, \ldots, l_k\}$ induces an assignment α_S to the variables $\{x_1, \ldots, x_n\}$ such that α_S sets all literals l_1, $\ldots$, l_k in S true. That is, if x_h is in S then α_S sets $x_h = 1$ and if $\overline{x_h}$ is in S then α_S sets $x_h = 0$. For variables x_h such that neither of x_h and $\overline{x_h}$ appears in S, the assignment α_S sets x_h arbitrarily.

Now since the k literals l_1, $\ldots$, l_k, which are set true by the assignment α_S, belong to k different clauses in E, we conclude that the assignment α_S satisfies at least k clauses in E. The assignment α_S to $\{x_1, \ldots, x_n\}$ can be easily constructed in linear time from the independent set S in the graph G_E. $\Box$

Corollary 34.5 *The number of vertices in a maximum independent set in the graph G_E is equal to the maximum number of clauses in E that can be satisfied by an assignment to $\{x_1, \ldots, x_n\}$.*

Now we are ready to prove:

Lemma 34.6 *If the* INDEPENDENT SET *problem has a polynomial time approximation scheme, then so does the* MAX-3SAT *problem.*

PROOF. Suppose that the INDEPENDENT SET problem has a polynomial time approximation scheme, we show how a polynomial time approximation scheme for the MAX-3SAT problem can be constructed.

For a given constant $\epsilon > 0$. Let `ApxIS` be a polynomial time approximation algorithm for the INDEPENDENT SET problem with approximation ratio $1 + \epsilon$. Consider the following algorithm.

Algorithm 34.1 `Apx3Sat`

```
Input:  a set of clauses $\{C_1, \ldots, C_m\}$, where each $C_i$ is
        a clause of at most 3 literals in $\{x_1, \ldots, x_n\}$
Output:  a truth assignment to $\{x_1, \ldots, x_n\}$

1.  construct the graph $G_E$;
2.  call the algorithm ApxIS on the graph $G_E$ to find
     an independent set $S$ in $G_E$;
3.  construct an assignment $\alpha_S$ to $\{x_1, \ldots, x_n\}$ from $S$
     such that $\alpha_S$ satisfies at least $|S|$ clauses in $E$.
```

It is clear that step 1 and step 2 of the algorithm `Apx3Sat` take polynomial time. Lemma 34.4 proves that step 3 of the algorithm `Apx3Sat` also takes polynomial time. Therefore, the algorithm `Apx3Sat` is a polynomial time approximation algorithm for the MAX-3SAT problem.

Now we analyze the approximation ratio for the algorithm `Apx3Sat`.

Let $Opt_{SAT}(E)$ be the optimal value of the set $E = \{C_1, \ldots, C_m\}$ of clauses, where E is treated as an instance of the MAX-3SAT problem, and let $Opt_{IS}(G_E)$ be the optimal value of the graph G_E, where G_E is treated as an instance of the INDEPENDENT SET problem. By Corollary 34.5,

$$Opt_{SAT}(E) = Opt_{IS}(G_E)$$

Let $Apx(\alpha_S)$ be the number of clauses in E that are satisfied by the assignment α_S. According to step 3 of the algorithm `Apx3Sat`, we have

$$\frac{Opt_{SAT}(E)}{Apx(\alpha_S)} \leq \frac{Opt_{SAT}(E)}{|S|} = \frac{Opt_{IS}(G_E)}{|S|} \tag{13}$$

Since S is the independent set produced by the approximation algorithm `ApxIS` for the INDEPENDENT SET problem, by our assumption,

$$\frac{Opt_{IS}(G_E)}{|S|} \leq 1 + \epsilon$$

We conclude that the approximation ratio of the algorithm `Apx3Sat` for the MAX-3SAT problem is bounded by

$$\frac{Opt_{SAT}(E)}{Apx(\alpha_S)} \leq 1 + \epsilon$$

This proves that the polynomial time algorithm `Apx3Sat` is an approximation algorithm of approximation ratio $1 + \epsilon$ for the MAX-3SAT problem. Since $\epsilon > 0$ is arbitrary, we have proved that the MAX-3SAT problem has a polynomial time approximation scheme. □

Theorem 34.7 *The* INDEPENDENT SET *problem has no polynomial time approximation scheme unless $P = NP$.*

PROOF. This follows directly from Lemma 34.6 and Theorem 33.3. □

Another important optimization problem, CLIQUE, is very closely related to the INDEPENDENT SET problem. Let G be a graph. A subset C of vertices in G is a *clique* in G if all vertices in C are mutually adjacent.

CLIQUE

INPUT: a graph G

OUTPUT: a clique C in G with the cardinality of C maximized

Let $G = (V, E)$ be a graph. The graph $G^c = (V, E')$ with the same vertex set V is called the *complement graph* of G if for any pair of vertices u and v in V, u and v are adjacent in G^c if and only if u and v are not adjacent in G. Note that the complement graph of the complement graph G^c is the original graph G.

Lemma 34.8 *Let $G = (V, E)$ be a graph and let $G^c = (V, E')$ be the complement graph of G. Let S be a subset of vertices in V. Then, S is an independent set in the graph G if and only if S is a clique in the graph G^c.*

PROOF. This follows directly from the definitions. □

The approximabilities of the CLIQUE problem and the INDEPENDENT SET problem are related by the following theorem.

Theorem 34.9 *Let $t(n)$ be a function that is at least as large as n. The* Clique *problem has an approximation algorithm of running time $O(t(n))$ and approximation ratio $r(n)$ if and only if the* Independent Set *problem has an approximation algorithm of running time $O(t(n))$ and approximation ratio $r(n)$).*

PROOF. Let `A-Clique` be an approximation algorithm for the Clique problem such that `A-Clique` has running time $O(t(n))$ and approximation ratio $r(n)$. Consider the following algorithm for the Independent Set problem:

Algorithm 34.2 `A-IS`

```
Input:  a graph $G$
Output:  an independent set $S$ in $G$

1.  construct the complement graph $G^c$;
2.  call the algorithm A-Clique on the graph $G^c$ to find
     a clique $S$ in $G^c$;
3.  return $S$ as an independent set in the graph $G$.
```

By Lemma 34.8, the set S constructed by the algorithm `A-IS` is an independent set in the graph G. It is clear that the running time of the algorithm `A-IS` is bounded by the running time of the algorithm `A-Clique` plus $O(n)$. By our assumption, the algorithm `A-Clique` runs in time $O(t(n))$ and $t(n) = \Omega(n)$, thus we conclude that the running time of the algorithm `A-IS` is also bounded by $O(t(n))$.

Again by Lemma 34.8, the number of vertices in a maximum independent set in the graph G is equal to the number of vertices in a maximum clique in the graph G^c. Therefore, if we let $Opt_{cl}(G^c)$ be the optimal value for the graph G^c treated as an instance for the Clique problem and let $Opt_{is}(G)$ be the optimal value for the graph G treated as an instance for the Independent Set problem, then we have

$$\frac{Opt_{is}(G)}{|S|} = \frac{Opt_{cl}(G^c)}{|S|}$$

By our assumption, the approximation ratio for the approximation algorithm `A-Clique` is bounded by $r(n)$, thus $Opt_{cl}(G^c)/|S| \leq r(n)$. This gives immediately that the approximation ratio of the approximation algorithm `A-IS` is also bounded by $r(n)$.

The other direction that if the Independent Set problem has an approximation algorithm of running time $O(t(n))$ and approximation ratio $r(n)$ then so does the Clique problem can be proved in a similar way. $\square$

Corollary 34.10 *The* Clique *problem has no polynomial time approximation scheme unless* $P = NP$.

PROOF. This follows directly from the combination of Theorem 34.9 and Theorem 34.7. $\square$

According to Theorem 34.9, we can say that essentially there is no difference in the approximability between the Clique problem and the Independent Set problem. In fact, a result can be obtained which is much stronger than Theorem 34.7 and Corollary 34.10 on the approximability of these two optimization problems. We will discuss this in the next lecture.

CPSC-669 Computational Optimization

Lecture #35, November 17, 1995

Lecturer: Professor Jianer Chen
Scribe: Xiaotao Chen
Revision: Jianer Chen

35 Independent Set is not in APX

In this lecture, We present results on the approximability of the Independent Set problem and the Clique problem that strengthen Theorem 34.7 and Corollary 34.10.

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs. The *composition graph* $G = G_1[G_2]$ of these two graphs is the graph $G = (V, E)$ that has vertex set $V = V_1 \times V_2$. Two vertices $[u_1, u_2]$ and $[v_1, v_2]$ in the graph G are adjacent, where u_1 and v_1 are vertices in G_1 and u_2 and v_2 are vertices in G_2, if and only if either (u_1, v_1) is an edge in G_1, or $u_1 = v_1$ and (u_2, v_2) is an edge in G_2. A convenient way to view the composition graph $G = G_1[G_2]$ is as being constructed by replacing each vertex of G_1 by a copy of G_2 and then replacing each edge of G_1 by a complete bipartite subgraph that joins every vertex in the copy corresponding to one endpoint to every vertex in the copy corresponding to the other endpoint.

Lemma 35.1 *Let $\{u_1, \ldots, u_k\}$ be an independent set of k vertices in the graph G_1 and let $\{v_1, \ldots, v_h\}$ be an independent set of h vertices in the graph G_2, then the kh vertices*

$$[u_i, v_j] \qquad 1 \le i \le k \quad \textit{and} \quad 1 \le j \le h$$

form an independent set in the composition graph $G_1[G_2]$.

PROOF. By the definition, no two vertices in $\{u_1, \ldots, u_k\}$ are adjacent in the graph G_1 and no two vertices in $\{v_1, \ldots, v_h\}$ are adjacent in the graph G_2. According to the definition of the composition graph $G_1[G_2]$, it is easy to check that no two vertices $[u_i, v_j]$ and $[u_{i'}, v_{j'}]$ in the list given in the lemma are adjacent in the graph $G_1[G_2]$. $\Box$

Let G be any graph. We define a graph dG by iterating the composition operation. Inductively, $1G = G$, and for $d > 1$, $dG = (d-1)G[G]$. Note that if the graph G has n vertices, then the graph dG has n^d vertices.

Lemma 35.2 *Let d be a fixed positive integer and K be an integer satisfying $K > (k-1)^d$. If the composition graph dG has an independent set S_d of K vertices, then the graph G has an independent set S that contains at least k vertices. Moreover, the independent set S of the graph G can be constructed from the independent set S_d of the graph dG in polynomial time.*

PROOF. First note that since d is a fixed constant, the number of vertices in the graph dG is n^d, which is a polynomial of the number n of vertices in the graph G. Therefore, the running time of any polynomial time algorithm on the graph dG is also bounded by a polynomial of n.

We prove the lemma by induction on the integer d. The lemma is certainly true for the case $d = 1$.

Now consider the graph $dG = (d-1)G[G]$. Suppose that the independent set S_d in the graph dG is

$$S_d = \{[u_1, v_1], [u_2, v_2], \ldots, [u_K, v_K]\}$$

where u_i's are vertices in the graph $(d-1)G$ and v_j's are vertices in the graph G.

We partition the vertices in the set S_d into groups H_1, …, H_m such that all vertices in each group H_j have the same first coordinate. There are two possible cases.

If one H_i of the groups contains at least k vertices:

$$H_i = \{[u_i, w_1], [u_i, w_2], \ldots, [u_i, w_{k'}]\}$$

where $k' \geq k$, then since no two of these k' vertices are adjacent in the graph $dG = (d-1)G[G]$, by the definition, no two of the k' vertices w_1, …, $w_{k'}$ are adjacent in the graph G. That is, the set $S = \{w_1, \ldots, w_{k'}\}$ is an independent set of at least k vertices in the graph G. It is also easy to see that if such a group H_i exists in the set S_d, then the set $S = \{w_1, \ldots, w_{k'}\}$ can be constructed in polynomial time.

If none of the groups H_1, …, H_m in the set S_d contains more than $k-1$ vertices, then we have $m(k-1) \geq K$, which implies $m \geq K/(k-1) > (k-1)^{d-1}$. Pick any vertex $[u_i, w_i]$ from the group H_i, for $i = 1, \ldots, m$, since no two of the vertices

$$[u_1, w_1], [u_2, w_2], \ldots, [u_m, w_m]$$

are adjacent in the graph dG, by the definition, no two of the vertices u_1, …, u_m are adjacent in the graph $(d-1)G$. Thus, we obtain an independent set S_{d-1} of $m > (k-1)^{d-1}$ vertices in the graph $(d-1)G$:

$$S_{d-1} = \{u_1, u_2, \ldots, u_m\}$$

It is easy to see that the independent set S_{d-1} of the graph $(d-1)G$ can be constructed in polynomial time from the independent set S_d of the graph dG. Now by our inductive hypothesis, an independent set S of at least k vertices in the graph G can be constructed in polynomial time from the independent set S_{d-1} of more than $(k-1)^{d-1}$ vertices in the graph $(d-1)G$. Because d is a fixed constant, we conclude that an independent set S of at least k vertices in the graph G can be constructed in polynomial time from the independent set S_d of more than $(k-1)^d$ vertices in the graph dG.

This completes the proof of the lemma. $\square$

Theorem 35.3 *The number of vertices in a maximum independent set of the graph G is k if and only if the number of vertices in a maximum independent set of the graph dG is k^d.*

PROOF. Suppose that a maximum independent set of the graph G contains k vertices. Applying induction on Lemma 35.1, we can easily derive that a maximum independent set of the graph dG has at least k^d vertices. If a maximum independent set of the graph dG contains more than k^d vertices, then by Lemma 35.2, the graph G would contain an independent set of more than k vertices. This contradicts the assumption that a maximum independent set in G has k vertices. Thus, a maximum independent set of the graph dG contains exactly k^d vertices.

Conversely, if a maximum independent set of the graph dG has k^d vertices, then by Lemma 35.2, a maximum independent set of the graph G contains at least k vertices. A maximum independent set of G cannot contain more than k vertices since otherwise, by Lemma 35.1, a maximum independent set of dG would contain more than k^d vertices. $\square$

Now we come back to the approximability of the INDEPENDENT SET problem and the CLIQUE problem.

Lemma 35.4 *If the* INDEPENDENT SET *problem has a polynomial time approximation algorithm* $\mathtt{A}_c$ *with approximation ratio bounded by a fixed constant $c \geq 1$, then the* INDEPENDENT SET *problem has a polynomial time approximation scheme.*

PROOF. We construct a polynomial time approximation scheme for the INDEPENDENT SET problem from the polynomial time approximation algorithm $\texttt{A}_c$ of constant approximation ratio for the problem. Consider the following algorithm for the INDEPENDENT SET problem.

Algorithm 35.1 `PTAS-IS`

```
Input:  a graph $G$ and a constant $\epsilon > 0$
Output:  an independent set $S$ in $G$

1.  $d = \lceil (c-1)/\epsilon \rceil$;
2.  construct the composition graph $dG$;
3.  apply algorithm $\texttt{A}_c$ to construct an independent set $S_d$
     in the graph $dG$;
4.  construct an independent set $S$ of $k$ vertices in the
     graph $G$ from the set $S_d$, where $k$ is the largest
     integer such that $|S_d| > (k-1)^d$.
```

Since $d \geq (c-1)/\epsilon$, we have $(1+\epsilon)^d \geq 1 + d\epsilon \geq c$. Note that when c and ϵ are fixed constants, d is also a fixed constant. Thus, if the graph G has n vertices, then the number of vertices of the composition graph dG is bounded by a polynomial of n, and the composition graph dG can be constructed in polynomial time. Consequently, the independent set S_d in the graph dG can be constructed in time polynomial in n. Finally, according to Lemma 35.2, the independent set S in the graph G can be constructed from S_d in time polynomial in n. In conclusion, the algorithm `PTAS-IS` is a polynomial time algorithm.

Let $Opt(G)$ be the number of vertices in a maximum independent set of the graph G. By Theorem 35.3, the number $Opt(dG)$ of vertices in a maximum independent set of the graph dG is $(Opt(G))^d$. By the construction we have $|S_d| \leq k^d = |S|^d$. Finally, since the independent set S_d of the graph dG is produced by the algorithm $\texttt{A}_c$, by our assumption, $Opt(dG)/|S_d| \leq c$. With all these relations, we obtain

$$\left(\frac{Opt(G)}{|S|}\right)^d = \frac{(Opt(G))^d}{|S|^d} \leq \frac{Opt(dG)}{|S_d|} \leq c \leq (1+\epsilon)^d$$

which gives

$$\frac{Opt(G)}{|S|} \leq 1 + \epsilon$$

Therefore, for any fixed constant $\epsilon > 0$, the algorithm `PTAS-IS` constructs in polynomial time an independent set S for the graph G such that the approximation ratio $Opt(G)/|S|$ is bounded by $1 + \epsilon$. That is, the INDEPENDENT SET problem has a polynomial time approximation scheme. □

Since we already know that the INDEPENDENT SET problem has no polynomial time approximation scheme unless P = NP (Theorem 34.7), Lemma 35.4 derives the following theorem that is stronger than Theorem 34.7.

Theorem 35.5 *Unless P = NP, the* INDEPENDENT SET *problem is not in the class APX. In other words, unless P = NP, the* INDEPENDENT SET *problem has no polynomial time approximation algorithm whose approximation ratio is bounded by a fixed constant c.*

Corollary 35.6 *The* CLIQUE *problem has no polynomial time approximation algorithm whose approximation ratio is bounded by a fixed constant c, unless P = NP.*

PROOF. This follows from Theorem 34.9 and Theorem 35.5. □

CPSC-669 Computational Optimization

Lecture #36, November 20, 1995

Lecturer: Professor Jianer Chen
Scribe: Xiaotao Chen
Revision: Jianer Chen

36 Vertex Cover has no PTAS

After the establishment of the fundamental theorem (Theorem 33.3) that claims the non-approximability of the Max-3Sat problem, we have seen that the non-approximability of other optimization problems, such as the Max-Sat problem and the Independent Set problem, can be derived by reducing the Max-3Sat problem to them. The reductions used for these problems are essentially the same as the ones that are used in the proofs for the NP-completeness of the corresponding decision versions of the problems. Since there has been a long line of reductions that reduce one NP-complete problem to another, we would wonder whether we can somehow simply modify these reductions for decision problems to the world of optimization problems so that the non-approximability of all optimization problems can be derived.

Researchers quickly realized the difference in these two worlds. To illustrate the difference, we use the Vertex Cover as an example. Recall that given a graph G, a subset S of vertices of G forms a *vertex cover* of G if every edge in G has at least one end in the set S. The Vertex Cover problem is formulated as follows.

Vertex Cover

Input: a graph G

Output: a vertex cover S of G with $|S|$ minimized

Lemma 36.1 *Let $G = (V, E)$ be a graph. A set $S \subseteq V$ of vertices is a vertex cover of the graph G if and only if the set $V - S$ is an independent set in the graph G. In particular, S is a minimum vertex cover of the graph G if and only if $V - S$ is a maximum independent set in the graph G.*

PROOF. If S is a vertex cover of the graph G, then every edge in G has at least one end in S. Therefore, no edge in G has both ends in $V - S$. That is, the set $V - S$ is an independent set in the graph G.

If $V - S$ is an independent set of the graph G, then no two vertices in $V - S$ are adjacent. That is, no edge has both ends in the set $V - S$, or equivalently, every edge has at least one end in S. Thus, the set S is a vertex cover.

The rest of the lemma follows directly. $\square$

Therefore, suppose that we know that the decision version of the INDEPENDENT SET problem "given a graph G and k, does G have an independent set of k vertices?" is NP-complete, then we can use Lemma 36.1 to show the NP-completeness of the decision version of the VERTEX COVER problem "given a graph G and k, does G have a vertex cover of k vertices?" by reducing the decision version of the INDEPENDENT SET problem to it: asking if a given graph G has an independent set of k vertices is equivalent to asking if the graph G has a vertex cover of $n - k$ vertices. Therefore, the problem of finding an independent set of k vertices in a graph is not harder than the problem of finding a vertex cover of $n - k$ vertices in the same graph. Thus, the hardness of the first problem implies the hardness of the second problem.

However, the above reduction would not work if we *approximate* the optimal solutions in the problems. For instance, let $G = (V, E)$ be a graph of 1000 vertices in which a maximum independent set has 10 vertices while a minimum vertex cover contains 990 vertices. Now suppose we want to derive an approximation for the maximum independent set from an approximation for the minimum vertex cover. Let S be a vertex cover of 950 vertices in G. Then S is a pretty good approximation for the minimum vertex cover (with approximation ratio $990/950 < 1.05$). However, if we use the above reduction to get the independent set $V - S$ for the graph$G$, we obtain a very bad approximation $V - S$ for the maximum independent set (with approximation ratio $50/10 = 5$). Therefore, even Lemma 36.1 suggests a reduction that reduces the INDEPENDENT SET problem to the VERTEX COVER problem, the reduction does not preserve the approximation ratio when we apply approximation algorithms. The hardness of approximability of the INDEPENDENT SET thus does not imply the hardness of approximability of the VERTEX COVER problem. In fact, as we have shown, the INDEPENDENT SET has no polynomial time approximation algorithm with a constant approximation ratio unless P = NP (Theorem 35.5) while the VERTEX COVER problem has a simple approximation algorithm of approximation ratio 2 (Theorem 29.1).

Therefore, to study approximability of optimization problems, we need to consider reductions that somehow preserve the approximability. Before we present the formal definition for the reduction, we first introduce a notation.

Definition 36.1 Let $Q = \langle I, S, f, opt \rangle$ be an optimization problem. Let $x \in I$ be an instance of Q and $y \in S(x)$ be a solution to the instance x. The *relative error* $E_Q(x, y)$ of the solution y is defined by

$$E_Q(x,y) = \begin{cases} \frac{Opt(x)}{f(x,y)} - 1 & \text{if } opt = \max \\ \frac{f(x,y)}{Opt(x)} - 1 & \text{if } opt = \min \end{cases}$$

Simply speaking, the relative error $E_Q(x, y)$ of the solution y is the approximation ratio of y minus 1. Note that the relative error $E_Q(x, y)$ is always a non-negative number.

Now we are ready for giving the definition of our reduction that preserves the approximation ratio.

Definition 36.2 An optimization problem $Q_1 = \langle I_1, S_1, f_1, opt_1 \rangle$ can be *E-reducible* to an optimization problem $Q_2 = \langle I_2, S_2, f_2, opt_2 \rangle$, in written $Q_1 \leq_E Q_2$, if there are two polynominal time computable functions $g(\cdot)$ and $h(\cdot, \cdot)$, a polynomial $p(\cdot)$ and a constant β, such that

1. for any $x_1 \in I_1$, $g(x_1) = x_2 \in I_2$, satisfying $Opt_2(x_2) \leq p(|x_1|)Opt_1(x_1)$ and $Opt_1(x_1) \leq p(|x_1|)Opt_2(x_2)$;

2. for any $y_2 \in S_2(x_2)$, $h(x_1, y_2) = y_1 \in S_1(x_1)$ such that

$$E_1(x_1, y_1) \leq \beta E_2(x_2, y_2)$$

where $Opt_1(x_1)$ and $Opt_2(x_2)$ are the optimal values for the instances x_1 and x_2 of the problems Q_1 and Q_2, respectively, and E_1 and E_2 are the relative errors for the problems Q_1 and Q_2, respectively.

The definition of the E-reduction seems very natural from the viewpoint of polynomial time approximability of optimization problems. Condition 1 and the polynomial time computability of the functions g and h ensure that the reduction is a polynomial time transformation, while condition 2 requires that the transformation preserves the approximability. The only thing that looks a bit less natural is the requirement that the optimal values of the instances are related by a polynomial factor. We will see that this requirement makes the E-reduction the canonical reduction for an important class of optimization problems. Moreover, in most cases, this requirement is naturally satisfied. For example, if the optimal values of both optimization problems Q_1 and Q_2 are bounded by a polynomial of their input instance length, then this requirement is automatically satisfied.

The E-reduction from the optimization problem Q_1 to the optimization problem Q_2 provides a systematic technique for designing approximation algorithms for the problem Q_1 based on approximation algorithms for the problem Q_2, as shown by the following lemmas.

Lemma 36.2 *Let Q_1 and Q_2 be two optimization problems. If $Q_1 \leq_E Q_2$ and Q_2 has a fully polynomial time approximation scheme, then so does Q_1.*

PROOF. Let $Q_1 = \langle I_1, S_1, f_1, Opt_1 \rangle$ and $Q_2 = \langle I_2, S_2, f_2, Opt_2 \rangle$. Let A_2 be a fully polynomial time approximation scheme for the problem Q_2. Suppose that the reduction $Q_1 \leq_E Q_2$ is given as stated in Definition 36.2.

We design an approximation algorithm for the problem Q_1 as follows. For any constant $\epsilon > 0$, given $x_1 \in I_1$, we (1) compute $x_2 = g(x_1) \in I_2$; (2) apply the algorithm A_2 for Q_2 on x_2 to get a solution y_2 for x_2 satisfying $E_2(x_2, y_2) \leq \epsilon/\beta$; and finally (3) construct the solution $y_1 = h(x_1, y_2)$ for the instance x_1 of Q_1.

Since A_2 is a fully polynomial time approximation scheme for Q_2, the solution y_2 can be constructed in time polynomial in $|x_2|$ and β/ϵ, thus in time polynomial in $|x_1|$ and $1/\epsilon$. The other steps of the above algorithm take time polynomial in $|x_1|$. Therefore, the total running time of the above algorithm is bounded by a polynomial of $|x_1|$ and $1/\epsilon$.

Since the reduction is an E-reduction, we have

$$E_1(x_1, y_1) \leq \beta E_2(x_2, y_2) \leq \epsilon$$

Thus, the above algorithm has approximation ratio $1 + \epsilon$, thus, is a fully polynomial time approximation scheme for the problem Q_1. $\square$

In a similar way, we can prove

Lemma 36.3 *Let Q_1 and Q_2 be two optimization problems. If $Q_1 \leq_E Q_2$ and Q_2 has a polynomial time approximation scheme, then so does Q_1.*

Lemma 36.4 *Let Q_1 and Q_2 be two optimization problems. If $Q_1 \leq_E Q_2$ and Q_2 is in the class APX (i.e., Q_2 has a polynomial time approximation algorithm whose approximation ratio is bounded by a fixed constant), then so is Q_1.*

In particular, Lemma 36.3 implies

Lemma 36.5 *Let Q_1 and Q_2 be two optimization problems and $Q_1 \leq_E Q_2$. If Q_1 does not have a polynomial time approximation scheme unless P = NP, then Q_2 does not have a polynomial time approximation scheme unless P = NP.*

Example 36.3 We give an example to illustrate these ideas. In Lecture 34, we presented a reduction from Q_1, the Max-3Sat problem, to Q_2, the Independent Set problem as follows. Given an instance x_1 of the Max-3Sat problem, where x_1 is a set of clauses of at most 3 literals (x_1 was written as E in the discussion of Lecture 34), we construct a graph x_2 that is an instance for the Independent Set problem (x_2 was written as G_E in the discussion of Lecture 34). This corresponds to the polynomial time computable function $g(\cdot)$: $g(x_1) = x_2$. The condition that the optimal values of x_1 and x_2 are related by a polynomial factor is automatically satisfied since both of x_1 and x_2 have their optimal values bounded by a polynomial of their input instance length. Now for any solution y_2 to x_2, where y_2 is an independent set in the graph x_2 (y_2 was written as S in the discussion of Lecture 34), a truth assignment y_1 to the clauses in x_1 can be constructed in polynomial time (y_1 was written as α_S in the discussion of Lecture 34). This corresponds to the polynomial time computable function $h(\cdot,\cdot)$: $y_1 = h(x_1, y_2)$. Since we have $Opt_1(x_1) = Opt_2(x_2)$ (Corollary 34.5), $f_1(x_1, y_1) \geq f_2(x_2, y_2)$ (Lemma 34.4), we eventually have

$$E_1(x_1, y_1) = \frac{Opt_1(x_1)}{f_1(x_1, y_1)} - 1 \leq \frac{Opt_2(x_2)}{f_2(x_2, y_2)} - 1 = E_2(x_2, y_2)$$

Thus, if we let $\beta = 1$, then the above reduction is an E-reduction from the Max-3Sat problem to the Independent Set problem. Since the Max-3Sat problem has no polynomial time approximation scheme unless P = NP (Theorem 33.3), by Lemma 36.5, we derive directly that the Independent Set problem has no polynomial time approximation scheme unless P = NP.

Now we apply the E-reduction to show the non-approximability of the Vertex Cover problem. As we explained at the beginning of this lecture, the reduction from the Independent Set problem to the Vertex Cover problem does not seem to preserve approximation ratio. Thus, a reduction from another problem seems more proper. We present an E-reduction from the Max-3Sat problem to the Vertex Cover problem.

The reduction from the Max-3Sat problem to the Vertex Cover problem is the same as the one that reduces the Max-3Sat problem to the Independent Set problem.

Given an instance $E = \{C_1, C_2, \ldots, C_m\}$ of the Max-3Sat problem, where each C_i is a clause of at most 3 literals in $\{x_1, \ldots, x_n\}$. The graph $G_E = (V_E, A_E)$ is constructed as follows.

Every literal occurrence l in a clause C_i in E induces a vertex in the graph G_E, which will be named by $l^{(i)}$. For any pair of vertices $l_1^{(i)}$ and $l_2^{(j)}$ in G_E, there is an edge connecting them if and only if either

1. $i = j$, i.e., the literals $l_1^{(i)}$ and $l_2^{(j)}$ belong to the same clause in E; or
2. $i \neq j$ and $\overline{l_1^{(i)}} = l_2^{(j)}$, i.e., the literals $l_1^{(i)}$ and $l_2^{(j)}$ belong to different clauses in E and they negate each other.

This completes the transformation from the instance E of the Max-3Sat problem to an instance G_E of the Vertex Cover problem. Note that again the condition that the optimal values of E and G_E are related by a polynomial factor is automatically satisfied since both E and G_E have their optimal value bounded by their input length.

Now we show how the transformation from a solution S for the instance G_E, where S is a vertex cover of G_E, to a solution α_S for the instance E, where α_S is an assignment to $\{x_1, \ldots, x_n\}$, is constructed.

Given a vertex cover S of the graph $G_E = (V_E, A_E)$, we first construct the independent set $V_E - S$ in the graph G_E. Then based on the independent set $V_E - S$, we apply Lemma 34.4 to construct an assignment α' to the variables $\{x_1, \ldots, x_n\}$ such that α' satisfies at least $|V_E - S|$ clauses in E. If the assignment α' satisfies at least $m/2$ clauses in E, then we let $\alpha_S = \alpha'$. If the assignment α' satisfies less than $m/2$ clauses in E, then we apply Algorithm 31.1 `ApprxMaxSat` to construct the assignment α_S that satisfies at least $m/2$ clauses in E (see Lemma 31.2), ignoring the assignment α'. Therefore, the assignment α_S always satisfies at least $\max\{m/2, |V_E - S|\}$ clauses in E. According to Lemma 34.4, the assignment α' can be constructed from the independent set $V_E - S$ in polynomial time. Moreover, Algorithm 31.1 `ApprxMaxSat` runs in polynomial time. We conclude that the assignment α_S

to $\{x_1, \ldots, x_n\}$ can be constructed from the vertex cover S in polynomial time. This completes the transformation from a solution S for the instance G_E of the VERTEX COVER problem to a solution α_S for the instance E of the MAX-3SAT problem.

Now we analyze the relative errors of the solutions S and α_S.

Lemma 36.6 *A minimum vertex cover of the graph $G_E = (V_E, A_E)$ contains at most $5|V_E|/6$ vertices.*

PROOF. Each clause of the set E contains at most 3 literals, and there are m clauses in the set E. Since there is a one-to-one correspondence between the literal occurrences in E and the vertices in the graph G_E, the number $|V_E|$ of vertices of the graph G_E is bounded by $3m$, which gives $m/2 \geq |V_E|/6$.

By Algorithm 31.1 and Lemma 31.2, we know that there is an assignment to $\{x_1, \ldots, x_n\}$ that satisfies at least $m/2$ clauses in E. By Corollary 34.5, a maximum independent set in the graph G_E contains at least $m/2$ vertices. Now by Lemma 36.1, a minimum vertex cover of the graph G_E contains at most

$$|V_E| - m/2 \leq |V_E| - |V_E|/6 = 5|V_E|/6$$

vertices. The lemma is proved. $\Box$

Let $Opt_{sat}(E)$ be the optimal value for the instance E of the MAX-3SAT problem, let $Opt_{is}(G_E)$ be the optimal value for the instance G_E of the INDEPENDENT SET problem, and let $Opt_{vc}(G_E)$ be the optimal value for the instance G_E of the VERTEX COVER problem. According to Corollary 34.5 and Lemma 36.1

$$Opt_{sat}(E) = Opt_{is}(G_E) = |V_E| - Opt_{vc}(G_E) \tag{14}$$

Let $Apx_{sat}(\alpha_S)$ be the number of clauses in E that are satisfied by the assignment α_S. By the construction of the assignment α_S, we have

$$Apx_{sat}(\alpha_E) \geq \max\{m/2, |V_E - S|\}$$

Let $E_{sat}(E, \alpha_S)$ be the relative error of the solution α_S to the instance E of the MAX-3SAT problem and let $E_{vc}(G_E, S)$ be the relative error of the solution S to the instance G_E of the VERTEX COVER problem, we have

$$E_{sat}(E, \alpha_S) = \frac{Opt_{sat}(E)}{Apx_{sat}(\alpha_S)} - 1 \qquad \text{and} \qquad E_{vc}(G_E, S) = \frac{|S|}{Opt_{vc}(G_E)} - 1$$

Now from

$$E_{sat}(E, \alpha_S) = \frac{Opt_{sat}(E)}{Apx_{sat}(\alpha_S)} - 1 = \frac{Opt_{sat}(E) - Apx_{sat}(\alpha_S)}{Apx_{sat}(\alpha_S)}$$

we combine the relation $Apx_{sat}(\alpha_S) \geq \max\{m/2, |V_E - S|\}$ with Equation (14) and obtain

$$\begin{aligned} E_{sat}(E, \alpha_S) &\leq \frac{Opt_{sat}(E) - |V_E - S|}{m/2} = \frac{|S| - (|V_E| - Opt_{sat}(E))}{m/2} \\ &= \frac{|S| - (|V_E| - Opt_{is}(G_E))}{m/2} = \frac{|S| - Opt_{vc}(G_E)}{m/2} \end{aligned}$$

Now by Lemma 36.6 $Opt_{vc}(G_E) \leq 5|V_E|/6$ and note $|V_E| \leq 3m$, we get

$$\begin{aligned} E_{sat}(E, \alpha_S) &\leq \frac{|S| - Opt_{vc}(G_E)}{Opt_{vc}(G_E)/5} \\ &= 5\left(\frac{|S|}{Opt_{vc}(G_E)} - 1\right) = 5E_{vc}(G_E, S) \end{aligned}$$

We summarize this analysis in the following lemma.

Lemma 36.7 *The reduction constructed above that reduces the* MAX-3SAT *problem to the* VERTEX COVER *problem is an E-reduction.*

Combining Lemma 36.7 with Theorem 33.3 and Lemma 36.5, we obtain the following result for the VERTEX COVER problem.

Theorem 36.8 *The* MAX-3SAT *problem is E-reducible to the* VERTEX COVER *problem. In consequence, the* VERTEX COVER *problem has no polynomial time approximation scheme unless P = NP.*

CPSC-669 Computational Optimization

Lecture #37, November 27, 1995

Lecturer: Professor Jianer Chen
Scribe: Xiaotao Chen
Revision: Jianer Chen

37 The E-reducibility and the class APX-PB

The E-reduction introduced in the last lecture plus Theorem 33.3 have led to significant progress recently in the study of the approximability of optimization problems. Here we mention some of the recent results related to this direction.

Definition 37.1 An optimization problem $Q = \langle I, S, f, opt \rangle$ is in the class *APX-PB* if Q is in the class APX and there is a polynomial $p(\cdot)$ such that for all instances $x \in I$, we have $Opt(x) \leq p(|x|)$. Here the letters PB stand for "polynomially bounded".

Not all optimization problems in APX are in APX-PB. For example, the KNAPSACK problem, the MULTI-PROCESSOR SCHEDULING problem, and the Δ-TRAVELING SALESMAN problem are all in the class APX but none of them is in the class APX-PB. On the other hand, a very large class of important optimization problems are in the class APX-PB, such as the MAX-SAT problem, VERTEX COVER problem, 3-D MATCHING problem, and BIN PACKING problem. By Theorem 17.1, we know that an optimization problem in APX-PB has no fully polynomial time approximation scheme unless the problem can be solved in polynomial time. Therefore, for an NP-hard optimization problem in the class APX-PB, the most interesting thing is to decide whether it admits a polynomial time approximation scheme.

Since the E-reducibility requires that the optimal values of the instances in the original problem and in the transformed problem be related by a polynomial factor, it is impossible to reduce an optimization problem in APX but not in APX-PB to an optimization problem in APX-PB. In fact, this requirement in the E-reducibility makes the class APX-PB closed under the E-reducibility, as shown in the following lemma.

Lemma 37.1 *Let Q_1 and Q_2 be two optimization problems. If $Q_1 \leq_E Q_2$ and Q_2 is in the class APX-PB, then so is Q_1.*

PROOF. Let $Q_1 = \langle I_1, S_1, f_1, opt_1 \rangle$ and $Q_2 = \langle I_2, S_2, f_2, opt_2 \rangle$. By Lemma 36.4, since the optimization problem Q_2 is in the class APX, the optimization problem Q_1 is also in the class APX. Moreover, for any $x_1 \in I_1$, since the E-reduction transforms x_1 into an instance $x_2 \in I_2$ in polynomial time, we have $|x_2|$ bounded by a polynomial of $|x_1|$. Now since Q_2 is in the class APX-PB, the value $Opt_2(x_2)$ is bounded by a polynomial of $|x_2|$ thus by a polynomial of $|x_1|$. Now from the definition of the E-reducibility, the optimal value $Opt_1(x_1)$ of the instance x_1 is bounded by $Opt_2(x_2)$ times a polynomial of $|x_1|$. We conclude that $Opt_1(x_1)$ is bounded by a polynomial of $|x_1|$. That is, the problem Q_1 is in the class APX-PB. $\square$

The following important result, which was a little unexpected, has been derived recently. The proof of the theorem is omitted. Interested students can talk to the instructor for a discussion of the proof.

Theorem 37.2 *An optimization problem Q is in the class APX-PB if and only if it is E-reducible to the* MAX-3SAT *problem.*

Remark 37.2 Theorem 37.2 is derived from a modification of Theorem 33.1. Theorem 37.2 is a very powerful theorem. For example, our fundamental theorem, Theorem 33.3, can be derived directly from Theorem 37.2 without using Theorem 33.1. We give a complete proof for this.

According to Algorithm 23.1 `First-Fit` and Theorem 23.1, the BIN PACKING problem is in the class APX-PB. By Theorem 37.2, the BIN PACKING problem is E-reducible to the MAX-3SAT problem. Now if the MAX-3SAT problem has a polynomial time approximation scheme, then by Lemma 36.3, the BIN PACKING problem has a polynomial time approximation scheme. However, by Theorem 23.2, there is no polynomial time approximation algorithm of approximation ratio less than 1.5 for the BIN PACKING problem unless P = NP. In particular, the BIN PACKING problem has no polynomial time approximation scheme unless P = NP. This proves that the existence of a polynomial time approximation scheme for the MAX-3SAT problem implies P = NP. This is Theorem 33.3.

Remark 37.3 According to Lemma 37.1, the E-reduction cannot transform an optimization problem in APX but not in APX-PB to an optimization problem in APX-PB. Thus, the class APX-PB in Theorem 37.2 cannot be replaced by the class APX since the MAX-3SAT problem is in the class APX-PB. However, if the E-reduction is replaced by another reduction, called the

PTAS-reduction, then Theorem 37.2 is also true for the class APX, that is, an optimization problem is in the class APX if and only if it is PTAS-reducible to the MAX-3SAT problem. The definition of the PTAS-reducibility is a bit more technical, but it still preserves PTAS approximability and APX approximability. More specifically, suppose that a problem Q_1 is PTAS-reducible to a problem Q_2. Then if Q_2 has a polynomial time approximation scheme then so does Q_1, and if Q_2 is in the class APX then so is Q_1. I am not going to give a detailed discussion along this line. Instead, I refer the interested students to the related literature.

Theorem 37.2 motivates the following definition.

Definition 37.4 An optimization problem Q is *ApxPB-hard* if every optimization problem in the class APX-PB is E-reducible to Q. An optimization problem Q is *ApxPB-complete* if Q is in the class APX-PB and Q is ApxPB-hard.

According to the definition, the MAX-3SAT problem is ApxPB-complete.

Theorem 37.3 *An ApxPB-hard optimization problem Q has no polynomial time approximation scheme unless $P = NP$.*

PROOF. If the problem Q is ApxPB-hard, then by the definition, the MAX-3SAT problem can be E-reducible to the problem Q. Now the theorem follows directly from Lemma 36.5 and Theorem 33.3. □

Thus, the ApxPB-hardness implies the difficulty of approximation of an optimization problem. This provides a systematic technique for deriving non-approximability for optimization problems. To derive the ApxPB-hardness for optimization problems, we need the following lemma.

Lemma 37.4 *If $Q_1 \leq_E Q_2$ and $Q_2 \leq_E Q_3$, then $Q_1 \leq_E Q_3$.*

PROOF. Let $Q_1 = \langle I_1, S_1, f_1, opt_1 \rangle$, $Q_2 = \langle I_2, S_2, f_2, opt_2 \rangle$, and $Q_3 = \langle I_3, S_3, f_3, opt_3 \rangle$. Let $g_1(\cdot)$ and $h_1(\cdot,\cdot)$ be the functions, $p_1(\cdot)$ be the polynomial, and β_1 be the constant that constitute the E-reduction from Q_1 to Q_2, and let $g_2(\cdot)$ and $h_2(\cdot,\cdot)$ be the functions, $p_2(\cdot)$ be the polynomial, and β_2 be the constant that constitute the E-reduction from Q_2 to Q_3. Without loss of generality, suppose that both functions $g_1(\cdot)$ and $g_2(\cdot)$ are computable in time $p(n)$, where $p(\cdot)$ is a polynomial. Then it is easy to check that the functions

$$\overline{g}(x_1) = g_2(g_1(x_1)) \qquad \text{and} \qquad \overline{h}(x_1, y_3) = h_1(x_1, h_2(g_1(x_1), y_3))$$

the polynomial

$$\overline{p}(n) = p_1(n) \cdot p_2(p(n))$$

and the constant

$$\overline{\beta} = \beta_1 \beta_2$$

constitute an E-reduction from the optimization problem Q_1 to the optimization problem Q_3. □

Lemma 37.4 immediately gives

Lemma 37.5 *If an optimization problem Q_1 is ApxPB-hard, and $Q_1 \leq_E Q_2$, then the optimization problem Q_2 is ApxPB-hard.*

Now our previous study gives the following theorem.

Theorem 37.6 *The* MAX-3SAT *problem, the* MAX-SAT *problem, and the* VERTEX COVER *problem are ApxPB-complete. The* INDEPENDENT SET *problem and the* CLIQUE *problem are ApxPB-hard.*

PROOF. Algorithm 31.1 `ApprxMaxSat` shows that the MAX-3SAT problem and the MAX-SAT problem are in the class APX-PB. Theorem 37.2 shows that the MAX-3SAT problem is ApxPB-hard. The proof of Lemma 34.1 shows the ApxPB-hardness for the MAX-SAT problem. Theorem 36.8 gives the ApxPB-hardness of the VERTEX COVER problem, and Theorem 29.1 shows that the VERTEX COVER problem is in the class APX-PB.

Finally, Example 36.3 shows the ApxPB-hardness for the INDEPENDENT SET problem, and Theorem 34.9 gives the ApxPB-hardness for the CLIQUE problem. □

There are a number of more restricted optimization problems that are also ApxPB-complete. Note that the ApxPB-hardness of more restricted optimization problems sometimes is more useful in our derivation of non-approximability for our own optimization problems: our own problem may not be strong enough to have a simple E-reduction from an unrestricted ApxPB-hard optimization problem. On the other hand, a restricted version of the problem may look more similar to our own problem and an E-reduction may be readily available.

Let d be a fixed positive integer. We define the following restricted optimization problems.

> d-Occurrence Max-3Sat
>
> Input: a set E of clauses $C_1, C_2, \ldots, C_m$ on $\{x_1, \ldots, x_n\}$ such that each clause has at most 3 literals and each variable x_i appears, either as x_i or as $\overline{x_i}$, at most d times in E
>
> Output: a truth assignment on $\{x_1, \ldots, x_n\}$ that satisfies the maximum number of the clauses in E

> d-Occurrence Max-2Sat
>
> Input: a set E of clauses $C_1, C_2, \ldots, C_m$ on $\{x_1, \ldots, x_n\}$ such that each clause has at most 2 literals and each variable x_i appears, either as x_i or as $\overline{x_i}$, at most d times in E
>
> Output: a truth assignment on $\{x_1, \ldots, x_n\}$ that satisfies the maximum number of the clauses in E

> d-Degree Vertex Cover
>
> Input: a graph G in which each vertex has degree at most d
>
> Output: a vertex cover S of G with $|S|$ minimized

> d-Degree Independent Set
>
> Input: a graph G in which each vertex has degree at most d
>
> Output: an independent set S of G with $|S|$ maximized

Theorem 37.7 *The* 3-Occurrence Max-3Sat *problem is ApxPB-complete.*

proof. The 3-Occurrence Max-3Sat problem is a restricted version of the Max-Sat problem, which is in the class APX-PB (Theorem 37.6). Thus, the 3-Occurrence Max-3Sat problem is in the class APX-PB.

The proof that the 3-Occurrence Max-3Sat problem is ApxPB-hard is more complicated, which uses the techniques called *Amplifier* and *Expander*. We omit the detailed proof here. Interested students are encouraged to talk to the instructor. □

Theorem 37.8 *The* 4-Degree Vertex Cover *problem and the* 4-Degree Independent Set *problem are ApxPB-complete.*

PROOF. We first consider the 4-Degree Independent Set problem.

It is a bit surprising that the 4-Degree Independent Set problem is in the class APX-PB. As we have seen, the unrestricted Independent Set problem is not in the class APX unless P = NP (Theorem 35.5). However, when the degree of the vertices of a graph G is bounded by a fixed constant c, an independent set of the graph G with approximation ratio bounded by $c+1$ can be constructed easily by the following process: start with an empty set S. Each time we pick one vertex v from the graph G and put it in S, and delete all neighbors of v in the graph G. We iterate this until there is no vertex in the graph G. It is easy to see that the set S constructed by this process forms an independent set in the graph G. Since the degree of the vertices of the graph G is bounded by c, each time we add a vertex v to the set S, we delete at most $c+1$ vertices from the graph (including the vertex v). Therefore, the set S contains at least $n/(c+1)$ vertices, where n is the number of vertices in the graph G. Since an independent set of the graph G contains at most n vertices, the independent set S has approximation ratio at most $n/(n/(c+1)) = c+1$. In particular, when $c = 4$, we have that the 4-Degree Independent Set problem is in the class APX-PB.

To show the ApxPB-hardness of the 4-Degree Independent Set problem, we E-reduce the 3-Occurrence Max-3Sat problem to it. The reduction is exactly the same as the one we used to reduce the Max-3Sat problem to the Independent Set problem (see Lecture Notes 34). That is, given a set E of clauses, we construct a graph G_E such that each literal occurrence in E corresponds to a vertex in G_E, and two vertices in G_E are adjacent if and only if either they are in the same clause in E or they negate each other. Note that if E is an instance of the 3-Occurrence Max-3Sat problem, then each vertex l in G_E has at most 4 neighbors — two of them correspond to the literals in the same clause, and two of them correspond to the other occurrences of l in E. Therefore, the graph G_E is an instance of the 4-Degree Independent Set problem. As we have shown in Lecture 34, this reduction is an E-reduction. We conclude that the 4-Degree Independent Set problem is ApxPB-complete.

The problem 4-Degree Vertex Cover problem is in the class APX-PB because the unrestricted version of the Vertex Cover problem is in the class APX-PB. As we have seen in Lecture Notes 36, the E-reduction from the Max-3Sat problem to the Independent Set problem can be modified to an E-reduction from the Max-3Sat problem to the Vertex Cover problem. In particular, if E is an instance of the 3-Occurrence Max-3Sat problem, then the corresponding graph G_E is an instance of the 4-Degree Vertex Cover problem. Thus, the 3-Occurrence Max-3Sat problem is E-reducible to the 4-Degree Vertex Cover problem, which gives the ApxPB-hardness of the 4-Degree Vertex Cover problem. $\square$

Finally, we consider the 5-Occurrence Max-2Sat problem. We first present an E-reduction from the 4-Degree Independent Set problem to the 5-Occurrence Max-2Sat problem.

Let $G = (V, E)$ be a graph in which each vertex has degree at most 4, where $V = \{v_1, \ldots, v_n\}$, and $E = \{e_1, \ldots, e_m\}$. We construct an instance S_G for the 5-Occurrence Max-2Sat problem as follows. The boolean variable set of S_G is $\{v_1, \ldots, v_n\}$. For each vertex v_i in G, the set S_G has a 1-literal clause (v_i), and for each edge $e_h = [v_i, v_j]$ in G, S_G has a 2-literal clause $(\overline{v_i} \vee \overline{v_j})$. Note that each vertex of G appears in one 1-literal clause and in at most four 2-literal clauses. Thus, the set S_G is an instance of the 5-Occurrence Max-2Sat problem. This completes the construction of the instance S_G for the 5-Occurrence Max-2Sat problem.

We call an assignment to $\{v_1, \ldots, v_n\}$ a *setting assignment* if it satisfies all 2-literal clauses in the set S_G.

Lemma 37.9 *Let α be an assignment to the boolean variables $\{v_1, \ldots, v_n\}$. Then there is a setting assignment α_0 that satisfies at least as many clauses in S_G as α does. Moreover, the setting assignment α_0 can be constructed from the assignment α in polynomial time.*

PROOF. If α is a setting assignment, then simply let α_0 be α. Otherwise, suppose that the clause $(\overline{v_i} \vee \overline{v_j})$ is not satisfied by the assignment α, then α sets $v_i = 1$ and $v_j = 1$. Now we set $v_i = 0$. Note this makes false the 1-literal clause (v_i), which was set true by α, but makes true the 2-literal clause $(\overline{v_i} \vee \overline{v_j})$, which was set false by α. No other 1-literal clauses in S_G are turned from true to false. Moreover, since only the negation of v_i appears in 2-literal clauses in S_G, no 2-literal clauses in S_G are turned from true to false. Therefore, setting $v_i = 0$ will not decrease the number of clauses satisfied by the assignment. Now we repeat the above process on each 2-literal clause that is not satisfied by the assignment. Eventually, we obtain a setting assignment α_0 that satisfies at least as many clauses in S_G as the assignment α does. $\square$

Lemma 37.10 *Let α be a setting assignment to S_G and let (u_1), ..., (u_k) be the 1-literal clauses satisfied by the assignment α, then $\{u_1, \ldots, u_k\}$ is an independent set in the graph G.*

PROOF. Consider any pair u_i and u_j. If the vertices u_i and u_j are adjacent in G, then we have a 2-literal clause $(\overline{u_i} \vee \overline{u_j})$ in S_G. Since α sets both u_i and u_j true, the clause $(\overline{u_i} \vee \overline{u_j})$ is not satisfied by the assignment α. This contradicts the assumption that α is a setting assignment. $\square$

Let $Opt_{is}(G)$ be the optimal value of the instance G for the 4-DEGREE INDEPENDENT SET problem, and let $Opt_{sat}(S_G)$ be the optimal value of the instance S_G for the 5-OCCURRENCE MAX-2SAT problem.

Lemma 37.11 *Let m be the number of edges in the graph G, then*

$$Opt_{is}(G) + m = Opt_{sat}(S_G)$$

PROOF. Let $D = \{u_1, \ldots, u_k\}$ be a maximum independent set in the graph G. Consider the assignment α_D to $\{v_1, \ldots, v_n\}$ that sets $u_j = 1$, for $j = 1, \ldots, k$, and sets all other variables 0. Thus, the assignment α_D satisfies exactly k 1-literal clauses in S_G. For each 2-literal clause $(\overline{v_i} \vee \overline{v_j})$, which corresponds to an edge $[v_i, v_j]$, since at least one of v_i and v_j is not in D, the assignment α_D sets $(\overline{v_i} \vee \overline{v_j})$ true. That is, the assignment α_D satisfies all 2-literal clauses. In conclusion, the assignment α_D satisfies $k + m$ clauses in S_G. This gives

$$Opt_{is}(G) + m \leq Opt_{sat}(S_G)$$

Now let α be an assignment to $\{v_1, \ldots, v_n\}$ that satisfies the largest number of clauses in the set S_G. By Lemma 37.9, we can assume that the assignment α is a setting assignment. Let (u_1), ..., (u_k) be the 1-literal clauses satisfied by α. By Lemma 37.10, the set $\{u_1, \ldots, u_k\}$ is an independent set in the graph G. Since the assignment α satisfies all m 2-literal clauses in S_G, we get

$$Opt_{is}(G) + m \geq Opt_{sat}(S_G)$$

This completes the proof. $\square$

Now we are ready for an E-reduction from the the 4-DEGREE INDEPENDENT SET problem to the 5-OCCURRENCE MAX-2SAT problem.

Lemma 37.12 *The* 4-Degree Independent Set *problem is E-reducible to the* 5-Occurrence Max-2Sat *problem.*

proof. Given an instance G of the 4-Degree Independent Set problem, we use the transformation described above to construct an instance S_G for the 5-Occurrence Max-2Sat problem. The instance S_G can certainly be constructed from the instance G in polynomial time.

Now suppose that α is a solution to the instance S_G, i.e., α is assignment to the boolean variables $\{v_1, \ldots, v_n\}$ in S_G. We construct a solution D_α to the instance G, where D_α is an independent set in the graph G, as follows. We first construct a setting assignment α_0 that satisfies at least as many clauses as α does. According to Lemma 37.9, the assignment α_0 can be constructed from the assignment α in polynomial time. Let (v_1), ..., (v_k) be the 1-literal clauses satisfied by α_0. Then by Lemma 37.10, $D = \{v_1, \ldots, v_k\}$ is an independent set in the graph G. Now if $|D| \geq n/5$, we let $D_\alpha = D$, otherwise, we let D_α be an independent set of at least $n/5$ vertices in G. Note that by the proof of Theorem 37.8, an independent set of at least $n/5$ vertices in G can be constructed in polynomial time when the degree of vertices in the graph G is bounded by 4. This completes the construction of the transformation that transforms the solution α of S_G, which is an instance of the 5-Occurrence Max-2Sat problem, to a solution D_α of G, which is an instance of the 4-Degree Independent Set problem. By the above discussion, the solution D_α can be constructed from the solution α in polynomial time.

Now we analyze the relative errors. Let $Apx(\alpha)$ be the number of clauses satisfied by the assignment α. By the construction of the independent set D_α, we have

$$|D_\alpha| \geq \max\{Apx(\alpha) - m, n/5\} \tag{15}$$

Since each vertex of G has degree at most 4, the number m of edges in the graph G, which also equals the number of 2-literal clauses in S_G, is bounded by $2n$. Therefore,

$$Apx(\alpha) \leq n + m \leq 3n \tag{16}$$

Let $E_{is}(G, D_\alpha)$ be the relative error of the solution D_α to the instance G of the 4-Degree Independent Set problem, and $E_{sat}(S_G, \alpha)$ be the relative error of the solution α for the instance S_G of the 5-Occurrence Max-2Sat problem. Using Lemma 37.11, together with Equations (15) and

(16), We have

$$\begin{aligned}
E_{is}(G, D_\alpha) &= \frac{Opt_{is}(G)}{|D_\alpha|} - 1 = \frac{Opt_{is}(G) - |D_\alpha|}{|D_\alpha|} \\
&\le \frac{Opt_{is}(G) - (Apx(\alpha) - m)}{n/5} = \frac{Opt_{is}(G) + m - Apx(\alpha)}{n/5} \\
&= \frac{Opt_{sat}(S_G) - Apx(\alpha)}{n/5} \le \frac{Opt_{sat}(S_G) - Apx(\alpha)}{Apx(\alpha)/15} \\
&= 15E_{sat}(S_G, \alpha)
\end{aligned}$$

This completes the proof that the above reduction from the 4-Degree Independent Set problem to the 5-Occurrence Max-2Sat problem is an E-reduction. □

Theorem 37.13 *the* Max-2Sat *problem and the* 5-Occurrence Max-2Sat *problem are ApxPB-complete.*

proof. It is easy to see that both of these two problems are in the class APX-PB.

The ApxPB-hardness of the 5-Occurrence Max-2Sat problem is derived from Lemma 37.12

Finally, since the 5-Occurrence Max-2Sat problem is a restricted version of the Max-2Sat problem, we conclude that the Max-2Sat problem is also ApxPB-hard. □

We will study more ApxPB-complete optimization problems in the rest of this course.

CPSC-669 Computational Optimization

Lecture #38, November 29, 1995

Lecturer: Professor Jianer Chen
Scribe: Balarama Varanasi
Revision: Jianer Chen

38 3-D Matching has no PTAS

In today's lecture, we study the approximability of the 3-D Matching problem. Recall that a *matching* M in a set T of triples is a subset of T such that no two triples in M have the same coordinate at any dimension.

> 3-D Matching
>
> Input: a set $S \subseteq X \times Y \times Z$ of triples
>
> Output: a matching M in S with $|M|$ maximized

According to Algorithm 30.1 `Apprx3D-Second` and Theorem 30.6, the 3-D Matching problem is in the class APX-PB. We show below that the 3-D Matching problem is ApxPB-complete.

We construct an E-reduction from an ApxPB-complete problem, the 3-Occurrence Max-3Sat, to the 3-D Matching problem.

Let S be a set of clauses $\{C_1, \ldots, C_m\}$ on boolean variables $\{x_1, \ldots, x_n\}$ in which each clause contains at most three literals and each variable x_i appears, either as x_i or as $\overline{x_i}$, at most 3 times in S. The set S is an instance of the 3-Occurrence Max-3Sat problem. We construct an instance T_S of the 3-D Matching problem based on S.

We will use graphs to represent the triples in T_S. Each triple will be given as a triangle whose three vertices correspond to the three components of the triple. Therefore, if two triples have a common component, then the two corresponding triangles will have a shared vertex.

For each boolean variable u in $\{x_1, \ldots, x_n\}$ we have a ring structure. If u has three occurrences in S, then the ring structure of u consists of six triples, connected as in Figure 8(a). Similarly, if u has two occurrences or one occurrence in S, then the ring structure of u consists of four triples or two triples, connected as in Figure 8(b) and Figure 8(c), respectively.

In the following, we first consider the case that the boolean variable u has three occurrences in the set S. The discussion for the cases that u

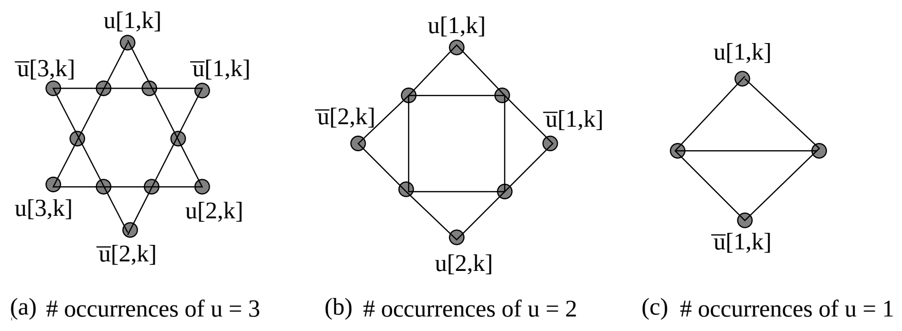

Figure 8: The ring structure for the boolean variable u

has two occurrences or one occurrence in S is very similar, we will briefly describe them after we complete the discussion on the case that u has three occurrences.

For the boolean variable u that has three occurrences in the set S, there are four identical rings of six triples. For $k = 1, \ldots, 4$, the kth ring has its outer vertices labeled by $u[1,k]$, $\overline{u}[1,k]$, $u[2,k]$, $\overline{u}[2,k]$, $u[3,k]$, and $\overline{u}[3,k]$ (see Figure 8). For each $i = 1, 2, 3$, the four vertices $u[i,1]$, $u[i,2]$, $u[i,3]$ and $u[i,4]$ are connected by three new triples

$$(u[i,1], u[i,2], u1[i]), \quad (u[i,3], u[i,4], u2[i]), \quad (u1[i], u2[i], u[i])$$

in a binary tree manner. Similarly, for each $i = 1, 2, 3$, the four vertices $\overline{u}[i,1]$, $\overline{u}[i,2]$, $\overline{u}[i,3]$ and $\overline{u}[i,4]$ are connected by three new triples

$$(\overline{u}[i,1], \overline{u}[i,2], \overline{u1}[i]), \quad (\overline{u}[i,3], \overline{u}[i,4], \overline{u2}[i]), \quad (\overline{u1}[i], \overline{u2}[i], \overline{u}[i])$$

in a binary tree manner. Figure 9 shows the four rings and the three new triples connecting the vertices $u[1,1]$, $u[1,2]$, $u[1,3]$ and $u[1,4]$. Note that the new triples connecting the other 20 triples in the rings are not shown in Figure 9.

The triples contained in each ring will be called *ring triples*. The triples

$$\begin{gathered}(u[i,1], u[i,2], u1[i]), \quad (u[i,3], u[i,4], u2[i]), \\ (\overline{u}[i,1], \overline{u}[i,2], \overline{u1}[i]), \quad (\overline{u}[i,3], \overline{u}[i,4], \overline{u2}[i])\end{gathered}$$

for $i = 1, 2, 3$, will be called *leaf triples*, and the triples

$$(u1[i], u2[i], u[i]) \quad (\overline{u1}[i], \overline{u2}[i], \overline{u}[i])$$

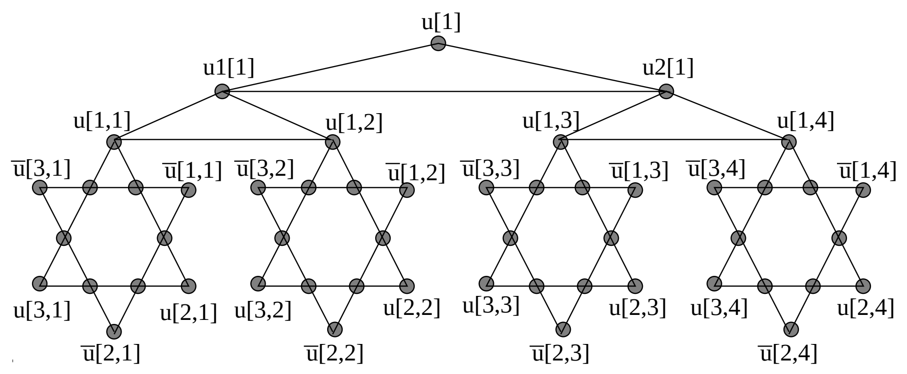

Figure 9: The set T_u of triples for the boolean variable u

for $i = 1, 2, 3$, will be called *root triples*. Moreover, a triple will be called a *positive triple* if it contains a component labeled as $u[\cdot]$ or $u[\cdot,\cdot]$, and a triple will be called a *negative triple* if it contains a component labeled as $\overline{u}[\cdot]$ or $\overline{u}[\cdot,\cdot]$. Note that by this definition, every triple constructed above is either a positive triple or a negative triple.

Therefore, there is a set T_u of 42 triples corresponding to each boolean variable u in $\{x_1, \ldots, x_n\}$: 24 of them are ring triples, 12 of them are leaf triples, and 6 of them are root triples.

To make the set T_u a valid set of triples, i.e., a subset of $X \times Y \times Z$, we must label the vertices in T_u with X, Y, or Z properly. All trees will be labeled identically, so we only describe the labeling for the tree rooted at $u[1]$. Label $u[1]$ with X, label $u1[1]$ with Y and $u2[1]$ with Z, and label $u[1,1]$ with Z, $u[1,2]$ with X, $u[1,3]$ with X, and $u[1,4]$ with Y. Note that for each fixed ring, this labeling process labels all outer vertices of the ring with the same symbol. Thus, the inner vertices in the ring can be properly labeled using the other two symbols. It is not hard to verify that in this labeling process, no triangle in T_u has two vertices labeled with the same symbol. Therefore, the set T_u represents a set of triples.

We first study the matching problem of the set T_u. Note that the set T_u is also an instance of the 3-D MATCHING problem.

It is easy to check that the following two sets M_u^+ and M_u^- in the set T_u are matchings in T_u.

The set M_u^+ consists of: (1) the 12 ring triples that contain $\overline{u}[i,k]$, for $i = 1, 2, 3$ and $k = 1, 2, 3, 4$, respectively; (2) the 6 leaf triples $(u[i,1], u[i,2], u1[i])$, and $(u[i,3], u[i,4], u2[i])$, for $i = 1, 2, 3$; and

(3) the 3 root triples $(\overline{u1}[i], \overline{u2}[i], \overline{u}[i])$, for $i = 1, 2, 3$.

The set M_u^- consists of: (1) the 12 ring triples that contain $u[i, k]$, for $i = 1, 2, 3$ and $k = 1, 2, 3, 4$, respectively; (2) the 6 leaf triples $(\overline{u}[i, 1], \overline{u}[i, 2], \overline{u1}[i])$, and $(\overline{u}[i, 3], \overline{u}[i, 4], \overline{u2}[i])$, for $i = 1, 2, 3$; and (3) the 3 root triples $(u1[i], u2[i], u[i])$, for $i = 1, 2, 3$.

Each of the matchings M_u^+ and M_u^- contains 21 triples. The two matchings M_u^+ and M_u^- will be called the *canonical matchings* in T_u.

Lemma 38.1 *The canonical matchings M_u^+ and M_u^- are maximum matchings in T_u.*

PROOF. If we regard the set T_u as a graph, then a matching in T_u corresponds to a set of disjoint triangles in the graph. We first count the number of vertices in this graph.

Each ring contains 12 different vertices, each leaf triple adds a new vertex, and each root triple adds another new vertex. Since there are 4 rings, 12 leaf triples, and 6 root triples, we conclude that there are totally 66 vertices in the graph.

Since 66 vertices can make at most 22 disjoint triangles, the number of triples in a maximum matching in T_u is at most 22. Now suppose that M_u is a matching of 22 triples in T_u. Then every vertex is contained in a triangle in M_u. In particular, the six vertices labeled with $u[i]$ and $\overline{u}[i]$, $i = 1, 2, 3$, should appear in M_u. Since the six root triples are the only triples that contain these vertices, all these six root triples should be in the matching M_u. In consequence, no leaf triples can be in the matching M_u since every leaf triple shares a vertex with a root triple. Now by the structure of the rings, each ring can have at most 3 triples in M_u. Thus, the matching M_u contains at most 12 ring triples. Summarizing all these, we derive that the matching M_u would contain at most 18 triples, contradicting the assumption that M_u contains 22 triples. Therefore, no matching in T_u can contain 22 triples.

Since the canonical matchings M_u^+ and M_u^- each contains 21 triples in T_u, we conclude that the canonical matchings are maximum matchings in the set T_u. □

Let M_u be a matching in T_u. We call a triple *matched* (by M_u) if it is contained in M_u. Otherwise, we say that the triple is *unmatched.*

Lemma 38.2 *Let M_u be a matching in the set T_u. If M_u is not a canonical matching, then M_u is not maximum.*

PROOF. Suppose that M_u is a maximum matching in T_u. We show that M_u must be one of the canonical matchings M_u^+ and M_u^-.

By Lemma 38.1, $|M_u| = 21$. Let r_u, l_u, and t_u be the number of ring triples, leaf triples, and root triples in M_u, respectively. Then

$$|M_u| = 21 = r_u + l_u + t_u$$

Since each matched root triple must be connected with two unmatched leaf triples and since each leaf triple is connected with exactly one root triple, we must have

$$t_u \leq \lfloor (12 - l_u)/2 \rfloor \tag{17}$$

Since each matched leaf triple must be connected with two unmatched ring triples and since each ring triple is connected with exactly one leaf triple, we must have

$$r_u \leq 24 - 2l_u \tag{18}$$

Another trivial upper bound for r_u is 12 since each ring can have at most 3 matched ring triples. This gives us

$$\begin{aligned} |M_u| &= r_u + l_u + t_u \\ &\leq \min\{24 - l_u + \lfloor (12 - l_u)/2 \rfloor,\ 12 + l_u + \lfloor (12 - l_u)/2 \rfloor\} \end{aligned}$$

From this relation, it is easy to verify that in order to make $|M_u| = 21$, we must have $l_u = 6$. Combining $l_u = 6$ and $|M_u| = 21$ together with Equations (17) and (18), we also get $t_u = 3$ and $r_u = 12$.

From $r_u = 12$, we derive that each ring in T_u must have exactly three ring triples in M_u. Thus, each ring either has all its three positive triples in M_u but none of its negative triples in M_u, or has all its three negative triples in M_u but none of its positive triples in M_u.

We show that it is impossible that one ring has all its positive ring triples in M_u while another ring has all its negative ring triples in M_u.

If the first ring has all its positive ring triples in M_u while the second ring has all its negative ring triples in M_u, then none of the six leaf triples that are connected to the ring triples in the first and the second rings can be in the matching M_u. Since M_u contains six leaf triples and T_u has totally 12 leaf triples, all six leaf triples that are connected to ring triples in the third and the fourth rings must be in M_u. But this is impossible because it would imply that no ring triples in the third and the fourth rings are in M_u. This proves that we must either have all positive ring triples in the first and the second rings in M_u or have all negative ring triples in the first and the second rings in M_u. Similarly, either all positive ring triples in the third and the fourth rings are in M_u or all negative triples in the third and the fourth rings are in M_u.

Now suppose that the first and the second rings have all their positive ring triples in M_u while the third and the fourth rings have all their negative ring triples in M_u. Since the matching M_u has 6 leaf triples, the 3 negative leaf triples connecting to triples in the first and the second rings and the 3 positive leaf triples connecting to triples in the third and the fourth rings must be in the matching M_u. However, it would imply that none of the root triples can be in the matching M_u, contradicting the fact that the matching M_u contains 3 root triples.

Therefore, we must either have all positive ring triples in M_u but no negative ring triples in M_u, or have all negative ring triples in M_u but no positive ring triples in M_u. Whenever this is decided, the six leaf triples and the three root triples in M_u are uniquely determined. In fact, if all positive ring triples are in M_u, then M_u must be the canonical matching M_u^-, while if all negative ring triples are in M_u, then M_u must be the canonical matching M_u^+. $\Box$

This completes the discussion on the set T_u of triples, where u is a boolean variable in $\{x_1, \ldots, x_n\}$ that has 3 occurrences in the given set S of clauses.

If u is a boolean variable that has 2 occurrences in S, then 4 rings of 4 triples, which has the structure shown in Figure 8(b), are used. Four binary tree structures are constructed by adding 8 leaf triples and 4 root triples. Thus, the set T_u of triples corresponding to u contains 28 triples. There are two canonical matchings M_u^+ and M_u^- of 14 triples in T_u such that M_u^+ contains all negative root triples but no positive root triples, while M_u^- contains all positive root triples but no negative root triples. Moreover, M_u^+ and M_u^- are the only maximum matchings in the set T_u.

Similarly, if u is a boolean variable that has 1 occurrence in S, then 4 rings of 2 triples, which has the structure shown in Figure 8(c), are used. Four binary tree structures are constructed by adding 4 leaf triples and 2 root triples. Thus, the set T_u of triples corresponding to u contains 14 triples. There are two canonical matchings M_u^+ and M_u^- of 7 triples in T_u such that M_u^+ contains all negative root triples but no positive root triples, while M_u^-

contains all positive root triples but no negative root triples. Moreover, M_u^+ and M_u^- are the only maximum matchings in T_u.

We summarize these discussions into the following theorem.

Theorem 38.3 *Let u be a boolean variable in $\{x_1, \ldots, x_n\}$ such that u has d occurrences in the set S, $1 \leq d \leq 3$. Then one can construct a set T_u of at most 42 triples with the following properties:*

1. *T_u has d positive root triples that contain the d components $u[1]$, ..., $u[d]$, respectively, and d negative root triples that contain the d components $\overline{u}[1]$, ..., $\overline{u}[d]$, respectively;*
2. *T_u has only two maximum matchings M_u^+ and M_u^-, such that M_u^- contains all d positive root triples but no negative root triples while M_u^+ contains all d negative root triples but no positive root triples.*

CPSC-669 Computational Optimization

Lecture #39, December 1, 1995

Lecturer: Professor Jianer Chen
Scribe: Balarama Varanasi
Revision: Jianer Chen

39 3-D MATCHING has no PTAS (contd.)

In the last lecture, we have shown the following theorem.

Theorem 39.1 *Let u be a boolean variable in $\{x_1, \ldots, x_n\}$ such that u has d occurrences in the set S, $1 \leq d \leq 3$. Then one can construct a set T_u of at most 42 triples with the following properties:*

1. *T_u has d positive root triples that contain the d components $u[1]$, ..., $u[d]$, respectively, and d negative root triples that contain the d components $\overline{u}[1]$, ..., $\overline{u}[d]$, respectively;*
2. *T_u has only two maximum matchings M_u^+ and M_u^-, such that M_u^- contains all d positive root triples but no negative root triples while M_u^+ contains all d negative root triples but no positive root triples.*

Now let us complete the construction of the set T_S of triples, which is an instance for the 3-D MATCHING problem, from the set S of clauses, which is an instance for the 3-OCCURRENCE MAX-3SAT problem.

Let S be the set of clauses on the boolean variable set $\{x_1, \ldots, x_n\}$. The set T_S is the union of all sets T_{x_i}, $i = 1, \ldots, n$, which satisfies the properties stated in Theorem 39.1, plus the *clause triples* desribed as follows. For each clause $C_h = (u \vee v \vee w)$, where u, v, and w are literals in $\{x_1, \ldots, x_n\}$ and we assume that this is the ith occurrence of u, the jth occurrence of v, and the kth occurrence of w, the set T_S contains three triples

$$(u[i], y[h], z[h]), \quad (v[j], y[h], z[h]), \quad (w[k], y[h], z[h])$$

where $y[h]$ and $z[h]$ are two new symbols introduced for the clause C_h. Similarly, if the clause C_h consists of 2 literals or 1 literal, the set T_S introduces two new symbols $y[h]$ and $z[h]$ and has 2 or 1 new triples. Figure 10 illustrates this construction. This completes the construction of the set T_S.

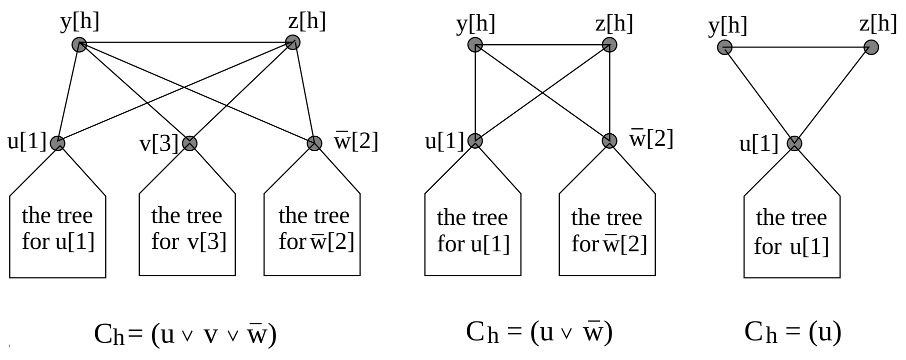

Figure 10: The clause triples in T_S

Since each clause in S has at most 3 literals, the set T_S contains at most $3m$ clause triples. Moreover, since each set T_{x_i} contains at most 42 triples, we conclude that the set T_S contains at most $42n + 3m$ triples. It is not difficult to see that the set T_S can be constructed from the set S of clauses in polynomial time.

Let x_i be a boolean variable. Each matching M in the set T_S induces a matching M/T_{x_i} in the set T_{x_i} of triples corresponding to the boolean variable x_i. We say that the matching M is a *canonical matching* for the set T_S if for all boolean variables x_i, the induced matching M/T_{x_i} is a canonical matching in the set T_{x_i}.

Lemma 39.2 *Let M be a matching in the set T_S. Then there is a canonical matching M' in the set T_S that contains at least as many triples as M does. Moreover, the canonical matching M' can be constructed from the matching M in polynomial time.*

PROOF. For each variable x_i, we consider the set M_i of clause triples in M that contain an occurrence of x_i.

If no clause triples in M_i contain a negative occurrence of x_i, or M_i is empty, then we replace the induced matching M/T_{x_i} by the canonical matching $M^+_{x_i}$ in T_{x_i}. Note that since the canonical matching $M^+_{x_i}$ contains only negative root triples but no positive root triples, this replacement still gives a matching in T_S. Moreover, since $M^+_{x_i}$ is a maximum matching in T_{x_i}, this replacement does not decrease the number of triples in the matching.

Similarly, if no clause triples in M_i contain a positive occurrence of x_i, then we replace the induced matching M/T_{x_i} by the canonical matching

$M_{x_i}^-$ in T_{x_i}, which gives a matching in T_S that is at least as large as M.

Finally, suppose that the set M_i has a clause triple that contains a positive occurrence of x_i and a clause triple that contains a negative occurrence of x_i. Then at least one positive root triple and at least one negative root triple in the set T_{x_i} are not contained in the matching M. Consequently, the induced matching M/T_{x_i} is not canonical, thus not maximum by Theorem 39.1. Moreover, since the variable x_i has at most three occurrences in the set S, we have either at most one clause triple in M_i that contains a positive occurrence of x_i or at most one clause triple in M_i that contains a negative occurrence of x_i. Without loss of generality, we assume that only one clause triple in M_i contains a positive occurrence of x_i. Then we perform the following operation: (1) delete the clause triple containing the positive occurrence of x_i, and (2) replace the induced matching M/T_{x_i} in T_{x_i} by the canonical matching $M_{x_i}^-$. Since after deleting the unique clause triple containing the positive occurrence of x_i, the matching M contains no clause triples containing positive occurrences of x_i while the canonical matching $M_{x_i}^-$ in T_{x_i} contains only positive root triples in T_{x_i}, we conclude that this operation still gives a matching in the set T_S. Moreover, since the induced matching M/T_{x_i} is not maximum in T_{x_i}, replacing M/T_{x_i} by the maximum matching $M_{x_i}^-$ in T_{x_i} increases the number of matched triples in T_{x_i} by at least one, which can be used to make up the clause triple deleted from M_i. In consequence, this operation replaces the induced matching M/T_{x_i} in T_{x_i} by a canonical matching in T_{x_i} and does not descrease the number of triples in the matching.

If we apply the above process to each of the boolean variables x_i, $i = 1, \ldots, n$, we will eventually get a canonical matching M' in the set T_S such that the matching M' is at least as large as the matching M. It is also easy to verify that the canonical matching M' can be constructed from the matching M in polynomial time. $\square$

For each boolean variable x_i, let n_i be the number of triples contained in a maximum matching in the set T_{x_i}, and let $N_0 = \sum_{i=1}^n n_i$.

Now we are ready to construct a solution for the instance S of the 3-Occurrence Max-3Sat problem based on a solution for the instance T_S of the 3-D Matching problem.

Lemma 39.3 *Given a matching M in the set T_S, an assignment α_M to the boolean variables $\{x_1, \ldots, x_n\}$ can be constructed in polynomial time such that α_M satisfies at least $|M| - N_0$ clauses in the set S.*

PROOF. Let M be a matching in the set T_S. By Lemma 39.2, we can construct in polynomial time a canonical matching M' in T_S such that $|M| \leq |M'|$.

Let M'_c be the subset of M' such that M'_c consists of all clause triples in M'. Then we have

$$|M'_c| = |M'| - N_0$$

No boolean variable x_i can have both its positive occurrence and its negative occurrence contained in the clause triples in M'_c — otherwise, the induced matching M'/T_{x_i} would not be canonical in T_{x_i}. Therefore, we can construct an assignment α_M to the boolean variables x_i, ..., x_n as follows: if M'_c has a clause triple that contains a positive occurrence of x_i then α_M assigns $x_i = 1$; if M'_c has a clause triple that contains a negative occurrence of x_i then α_M assigns $x_i = 0$. For variables that have no occurrences in M'_c, α_M assigns them arbitrarily. By the construction of the clause triples, for each clause C_h, at most one corresponding clause triple is contained in the set M'_c. Moreover, if a clause C_h has a corresponding clause triple in the set M'_c, then the assignment α_M sets the clause C_h true. In conclusion, the assignment α_M satisfies at least $|M'_c|$ clauses in S. From $|M'_c| + N_0 = |M'| \geq |M|$, we derive $|M_c| \geq |M| - N_0$. The lemma follows. □

Lemma 39.4 *Let $Opt_{sat}(S)$ be the optimal value of the instance S for the* 3-OCCURRENCE MAX-3SAT *problem and let $Opt_{3dm}(T_S)$ be the optimal value of the instance T_S for the* 3-D MATCHING *problem. Then*

$$Opt_{sat}(S) = Opt_{3dm}(T_S) - N_0$$

PROOF. Lemma 39.3 shows $Opt_{sat}(S) \geq Opt_{3dm}(T_S) - N_0$.

Now suppose that α is an assignment to $\{x_1, \ldots, x_n\}$ that satisfies the largest number of clauses in the set S. Without loss of generality, let the clauses satisfied by α be C_1, ..., C_k, where $k = Opt_{sat}(S)$. Suppose that the assignment α sets the literal l_i true in the clause C_i, for $i = 1, \ldots, k$. If α sets more than one literal in C_i true, pick any of them as l_i. Then we construct a matching M_α in the set T_S as follows. For $i = 1, \ldots k$, we pick the clause triple $(l_i, y[i], z[i])$. If l_i is a positive occurrence of a boolean variable x_j, then we also pick all triples in the canonical matching $M^+_{x_j}$ in the set T_{x_j}, and if l_i is a negative occurrence of a boolean variable x_j, then we pick all triples in the canonical matching $M^-_{x_j}$ in the set T_{x_j}. Note that for each boolean variable x_j, the assignment α either sets all its positive occurrences true or sets all its negative occurrences true. Therefore, the above selection of triples cannot result in any conflict. Moreover, since no positive root triple in T_{x_j} is contained in $M^+_{x_j}$ and no negative root triple in T_{x_j} is contained in $M^-_{x_j}$, the above selection of triples makes a matching in the set T_S. Finally, for those boolean variables x_j with no occurrence in $\{l_1, \ldots, l_k\}$, we pick all the triples in the canonical matching $M^+_{x_j}$. This constructs a canonical matching M_α in the set T_S, and M_α contains k clause triples in T_S. That is,

$$|M_\alpha| = k + N_0 = Opt_{sat}(S) + N_0$$

Since $Opt_{3dm}(T_S) \geq |M_\alpha|$, we derive $Opt_{sat}(S) \leq Opt_{3dm}(T_S) - N_0$. The lemma is proved. $\square$

Now we can describe how one can construct a solution α_M to the instance S of the 3-Occurrence Max-3Sat problem from a solution M to the instance T_S of the 3-D Matching problem. Recall that S is a set of m clauses. Consider the following algorithm.

Algorithm 39.1 `3DM-to-3SAT`

```
Input:  a matching $M$ in the set $T_S$
Output:  an assignment $\alpha_M$ to $\{x_1, \ldots, x_n\}$

1.  construct an assignment $\alpha$ to $\{x_1, \ldots, x_n\}$ that
     satisfies at least $|M| - N_0$ clauses in $S$;
2.  if $\alpha$ satisfies less than $m/2$ clauses in $S$
    then construct an assignment $\alpha_M$ that satisfies
          at least $m/2$ clauses in $S$
    else let $\alpha_M$ be $\alpha$;
3.  output $\alpha_M$.
```

By Lemma 39.3, the assignment α in step 1 can be constructed in polynomial time. Moreover, Algorithm 31.1 `ApprxMaxSat` and Lemma 31.2 show that an assignment that satisfies at least $m/2$ clauses in S can be constructed in polynomial time. In consequence, Algorithm 39.1 `3DM-to-3SAT` runs in polynomial time.

To study the relative errors, let $Apx(\alpha_M)$ be the number of clauses in S that are satisfied by the assignment α_M. By the construction of the assignment α_M, we have

$$Apx(\alpha_M) \geq \max\{|M| - N_0, m/2\} \tag{19}$$

Let $E_{sat}(S, \alpha_M)$ be the relative error of the solution α_M to the instance S of the 3-Occurrence Max-3Sat problem, and let $E_{3dm}(T_S, M)$ be the relative error of the solution M to the instance T_S of the 3-D Matching problem. Note that the set T_S has at most $3m + 42n \leq 126m$ triples. Thus, $|M| \leq 126m$. Combining this fact with Equation (19) and Lemma 39.4, we have

$$\begin{aligned}
E_{sat}(S, \alpha_M) &= \frac{Opt_{sat}(S)}{Apx(\alpha_M)} - 1 = \frac{Opt_{sat}(S) - Apx(\alpha_M)}{Apx(\alpha_M)} \\
&\leq \frac{Opt_{sat}(S) - (|M| - N_0)}{m/2} = \frac{Opt_{sat}(S) + N_0 - |M|}{m/2} \\
&= \frac{Opt_{3dm}(T_S) - |M|}{m/2} \leq \frac{Opt_{3dm}(T_S) - |M|}{|M|/252} \\
&= 252 E_{3dm}(T_S, M)
\end{aligned}$$

This shows that the reduction we constructed from the 3-Occurrence Max-3Sat problem to the 3-D Matching problem is an E-reduction. We conclude with the following theorem.

Theorem 39.5 *The* 3-Occurrence Max-3Sat *problem is E-reducible to the* 3-D Matching *problem.*

By Theorem 37.7, the 3-Occurrence Max-3Sat problem is ApxPB-complete. We get

Theorem 39.6 *The* 3-D Matching *problem is ApxPB-complete. Therefore, the* 3-D Matching *problem has no polynomial time approximation scheme unless* $P = NP$.

Remark 39.1 The set T_S of triples constructed from the set S of clauses in our E-reduction is actually an instance of a more restricted version of the 3-D Matching problem. Note that in the construction of the set T_S, each symbol in $X \cup Y \cup Z$ appears in at most 3 triples in T_S. In fact, all symbols in the set T_{x_i}, $i = 1, \ldots, n$, appear in at most 2 triples in T_S, only the symbols $y[h]$ and $z[h]$ introduced for the clause C_h, $h = 1, \ldots, m$, may appear in 3 triples in T_S. We can naturally define a problem called the 3-Occurrence 3-D Matching by requiring that in an instance of the 3-D Matching problem, each symbol appears in at most 3 triples. The set T_S is an instance of the 3-Occurrence 3-D Matching problem. Thus, our E-reduction constructed in these two lectures actually reduces the 3-Occurrence Max-3Sat problem to the 3-Occurrence 3-D Matching problem. In consequence, the 3-Occurrence 3-D Matching problem is also ApxPB-complete.

Remark 39.2 There is another optimization problem Triangle Packing whose ApxPB-completeness can be easily obtained from an E-reduction from the 3-D Matching problem. Let G be a graph. A *triangle* in G consists of three mutually adjacent vertices in G. Two triangles in G are *disjoint* if they do not share any common vertex. The Triangle Packing problem is formulated as follows.

> Tirangle Packing
>
> Input: a graph G
>
> Output: a set S of disjoint triangles in G with $|S|$ maximized

It is not very hard to see that if an instance S of the 3-D Matching problem is given as a graph, as we did in the last lecture, then a matching in S is a set of disjoint triangles in the graph. This observation leads to an E-reduction from the 3-D Matching problem to the Triangle Packing problem. We leave the details to the interested students. On the other hand, the best polynomial time approximation ratio for the Triangle Packing problem is 2. Therefore, the Triangle Packing problem is ApxPB-complete.

CPSC-669 Computational Optimization

Lecture #40, December 4, 1995

Lecturer: Professor Jianer Chen
Scribe: Balarama Varanasi
Revision: Jianer Chen

40 Max-Cut is ApxPB-complete

The last problem we will study in this course is the Max-Cut problem. Let $G = (V, E)$ be a graph. A *cut* of the graph G is a partition $D = (V_1, V_2)$ of the vertex set V of G. That is, $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \phi$. We say that an edge e of G is *crossing* in the cut D if one end of e is in V_1 and the other end of e is in V_2. The Max-Cut problem is defined as follows.

> Max-Cut
>
> Input: a graph $G = (V, E)$
>
> Output: a cut (V_1, V_2) of G that maximizes the number of crossing edges

While the Max-Cut problem is NP-hard, it has a very simple approximation algorithm, as shown below.

Algorithm 40.1 `Large-Cut`

```
Input:  a graph $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$
Output:  a cut $(V_1, V_2)$ of the graph $G$

1.  let $V_1 = \phi$  and  $V_2 = \phi$;
2.  for $i = 1$ to $n$ do
       if $v_i$ has more adjacent vertices in $V_1$ than in $V_2$
       then $V_2 = V_2 \cup \{v_i\}$
       else $V_1 = V_1 \cup \{v_i\}$;
3.  output $(V_1, V_2)$.
```

Theorem 40.1 *The approximation algorithm* `Large-Cut` *for the* Max-Cut *problem has approximation ratio* 2.

PROOF. In the algorithm, each time the vertex v_i is considered, the edges connecting v_i to the vertices v_1, …, v_{i-1} are counted. According to the algorithm, at least half of these edges become crossing edges. Therefore, at the end of the algorithm, at least half of the edges of the graph G are crossing edges. Since no cut can have the number of crossing edges larger than the number of edges in the graph G, the theorem follows. □

Remark 40.1 This simple algorithm provided the best approximation ratio for the MAX-CUT problem for over 20 years. Very recently (1994), the approximation ratio has been improved to 1.14. The complete version of the paper is still not available yet. Interested students may ask the instructor for a copy of the preliminary version of the paper.

To show that the MAX-CUT problem is ApxPB-hard. we construct an E-reduction from a ApxPB-complete problem, the 5-OCCURRENCE MAX-2SAT problem, to the MAX-CUT problem.

Let $S = \{C_1, \ldots, C_m\}$ be an instance of the 5-OCCURRENCE MAX-2SAT problem. That is, S is a set of clauses of at most two literals in the boolean variable set $\{x_1, \ldots, x_n\}$ and each variable x_i appears, either as x_i or as $\overline{x_i}$, at most five times in the set S. We construct an instance G_S for the MAX-CUT problem, where $G_S = (V_S, E_S)$ is a graph.

The vertex set V_S of the graph G_S is

$$V_S = \{x_1, \overline{x_1}, \ldots, x_n, \overline{x_n}, z, \overline{z}\}$$

where z is a new symbol.

For each $i = 1, \ldots, n$, there are 10 multiple edges connecting the vertices x_i and $\overline{x_i}$, and there are $2m$ multiple edges connecting the vertices z and $\overline{z}$. These edges connecting x_i and $\overline{x_i}$ or z and $\overline{z}$ will be called the *pairing edges* of the graph G_S. For each clause $C_i = (u \vee w)$ of two literals in S, we have a triangle consisting of three edges $[u, w]$, $[u, z]$, and $[w, z]$; and for each clause $C_i = (u)$ of one literal in S, we have two multiple edges connecting u and z. These edges will be called the *clause edges* corresponding to the clause C_i. Note that if a literal u appears in two different clauses C_i and C_j, then there are two multiple incident clause edges connecting u and z, corresponding to the two clauses C_i and C_j, respectively.

This completes the construction of the instance G_S of the MAX-CUT problem from the instance S of the 5-OCCURRENCE MAX-2SAT problem. It is clear that the graph G_S can be constructed from the set S of clauses in polynomial time.

A cut $D = (V_1, V_2)$ of G_S is *regular* if for any $u \in \{x_1, \ldots, x_n, z\}$, one of u and $\overline{u}$ is in the set V_1 and the other is in the set V_2. For a cut D of the graph G_S, denote by $|D|$ the number of crossing edges in the cut D.

Lemma 40.2 *There is a polynomial time algorithm that, given a cut D of the graph G_S, constructs a regular cut D_0 of G_S such that $|D| \leq |D_0|$.*

PROOF. Suppose $D = (V_1, V_2)$, where $V_1 \cup V_2 = V_S$ and $V_1 \cap V_2 = \phi$.

If for any boolean variable x_i, both vertices x_i and $\overline{x_i}$ are in V_1, then we remove x_i from V_1 and add it to V_2. We show this does not decrease the number of crossing edges in the cut. In fact, since S is an instance of the 5-Occurrence Max-2Sat problem and there are at most 2 clause edges incident to x_i corresponding to each clause containing x_i, there are at most 10 clause edges incident on x_i. Since both x_i and $\overline{x_i}$ are in V_1, these clause edges are the only edges incident on x_i that may be crossing edges in the cut D. On the other hand, there are 10 pairing edges connecting x_i and $\overline{x_i}$, which are not crossing edges in the cut D. Therefore, moving the vertex x_i from V_1 to V_2 will convert all these 10 pairing edges incident on x_i from non-crossing edges into crossing edges and may convert at most 10 clause edges incident on x_i from crossing edges into non-crossing edges. No other crossing edges in the cut are changed. In consequence, this process does not decrease the number of crossing edges in the cut.

If both vertices z and $\overline{z}$ are in V_1, then we move the vertex z from the set V_1 to the set V_2. Again, since z is incident to $2m$ clause edges and to $2m$ pairing edges, moving z from V_1 to V_2 does not decrease the number of crossing edges in the cut.

The case when both x_i and $\overline{x_i}$, or both z and $\overline{z}$, are in the set V_2 can be dealt with in a completely similar way.

Thus, applying this process on each boolean variable x_i and on z gives a regular cut D_0 without decreasing the number of crossing edges. Moreover, it is easy to verify that the cut D_0 can be constructed from the cut D in polynomial time. □

Lemma 40.3 *There is a polynomial time algorithm that, given a cut D of the graph G_S, constructs an assignment α_0 to $\{x_1, \ldots, x_n\}$ such that α_0 satisfies at least $(|D| - 10n - 2m)/2$ clauses in the set S.*

PROOF. We first convert the cut D into a regular cut D_0 of G_S such that $|D| \leq |D_0|$. By Lemma 40.2, the cut D_0 can be constructed in polynomial time.

Let $D_0 = (V_1, V_2)$. Then for each $u \in \{x_1, \ldots, x_n, z\}$, exactly one of u and $\overline{u}$ is in the set V_1. Thus, all $(10n + 2m)$ pairing edges in G_S are crossing edges in the cut D_0. Without loss of generality, we assume that the vertex

$\overline{z}$ is in the set V_1 while the vertex z is in the set V_2.

We let α_0 be the assignment to $\{x_1, \ldots, x_n\}$ that sets all literals in the set $V_1 - \{\overline{z}\}$ true. Note that since $D_0 = (V_1, V_2)$ is a regular cut, α_0 is always a valid assignment to $\{x_1, \ldots, x_n\}$.

Now consider a crossing edge e in D_0 that is a clause edge corresponding to a clause C_i in the set S. If $C_i = (u \vee w)$ consists of 2 literals, then since the crossing edge e is one of the three clause edges $[u, w]$, $[u, z]$ and $[w, z]$ corresponding to the clause C_i, exactly two of these three clause edges are crossing edges. In particular, by our assumption that the vertex z is in the set V_2, at least one of the vertices u and w is in V_1. In consequence, the assignment α_0 sets this literal true and satisfies the clause C_i. If $C_i = (u)$ consists of 1 literal, since the crossing edge e is one of the multiple clause edges connecting u and z, both multiple clause edges connecting u and z are crossing edges. Moreover, the vertex u is in the set V_1 since we assume that the vertex z is in the set V_2. Thus the assignment α_0 satisfies the clause C_i. We conclude that in either case, there are exactly two clause edges corresponding to C_i that are crossing edges in the cut D_0, and the assignment α_0 satisfies the clause C_i.

Since there are $|D_0| - 10n - 2m$ crossing edges in D_0 that are clause edges, and no three of them correspond to the same clause in S, we conclude that the assignment α_0 satisfies at least

$$(|D_0| - 10n - 2m)/2 \geq (|D| - 10n - 2m)/2$$

clauses in the set S. $\square$

Lemma 40.4 *Let $Opt_{sat}(S)$ be the optimal value of S, regarded as an instance of the* 5-OCCURRENCE MAX-2SAT *problem, and let $Opt_{cut}(G_S)$ be the optimal value of G_S, regarded as an instance of the* MAX-CUT *problem, then*

$$Opt_{sat}(S) = (Opt_{cut}(G_S) - 10n - 2m)/2$$

PROOF. Let D be a maximum cut of the graph G_S. By Lemma 40.3, there is an assignment that satisfies at least

$$(|D| - 10n - 2m)/2 = (Opt_{cut}(G_S) - 10n - 2m)/2$$

clauses in the set S. Consequently,

$$Opt_{sat}(S) \geq (Opt_{cut}(G_S) - 10n - 2m)/2$$

Conversely, let α be an assignment to $\{x_1, \ldots, x_n\}$ that satisfies the largest number of clauses in the set S. Construct a cut $D_\alpha = (V_1, V_2)$ of the graph G_S such that a literal u is in V_1 if and only if the assignment α sets u true. Moreover, the vertex $\overline{z}$ is in the set V_1 and the vertex z is in the set V_2. It is easy to verify that the cut D_α is a regular cut. Thus, all $10n + 2m$ pairing edges in G_S are crossing edges in the cut D_α.

Moreover, suppose without loss of generality that the assignment α satisfies the clauses C_i, $i = 1, \ldots, k$, in the set S, where $k = Opt_{sat}(S)$. Thus, the assignment α makes at least one literal u_i true in the clause C_i, so the literal u_i is in the set V_1. Since the vertex z is in the set V_2, exactly two of the clause edges corresponding to the clause C_i are crossing edges. Thus, there are at least $2k = 2Opt_{sat}(S)$ clause edges that are crossing edges in the cut D_α. This implies that the number of crossing edges in the cut D_α is at least $2Opt_{sat}(S) + 10n + 2m$, which should not be larger than $Opt_{cut}(G_S)$. Consequently,

$$Opt_{sat}(S) \leq (Opt_{cut}(G_S) - 10n - 2m)/2$$

This completes the proof of the lemma. □

Now we are ready to show how a solution D to the instance G_S of the Max-Cut problem, where D is a cut of the graph G_S, can be transformed into a solution α_D to the instance S of the 5-Occurrence Max-2Sat problem, where α_D is an assignment to the boolean variables $\{x_1, \ldots, x_n\}$. Consider the following algorithm.

Algorithm 40.2 `CUT-to-2SAT`

```
Input:  a cut $D$ of the graph $G_S$
Output:  an assignment $\alpha_D$ to $\{x_1, \ldots, x_n\}$

1.  construct an assignment $\alpha_0$ to $\{x_1, \ldots, x_n\}$ such that $\alpha_0$
     satisfies at least $(|D| - 10n - 2m)/2$ clauses in $S$;
2.  **if** $\alpha_0$ satisfies less than $m/2$ clauses in $S$
    **then** construct an assignment $\alpha_D$ that satisfies
           at least $m/2$ clauses in $S$
    **else** let $\alpha_D$ be $\alpha$;
3.  output $\alpha_D$.
```

By Lemma 40.3, the assignment α_0 in step 1 can be constructed in polynomial time. Moreover, Algorithm 31.1 `ApprxMaxSat` and Lemma 31.2 show that an assignment that satisfies at least $m/2$ clauses in S can be constructed in polynomial time. In consequence, Algorithm 40.2 `CUT-to-2SAT` runs in polynomial time.

To study the relative errors, let $Apx(\alpha_D)$ be the number of clauses in S that are satisfied by the assignment α_D. By the construction of the assignment α_D, we have

$$Apx(\alpha_D) \geq \max\{(|D| - 10n - 2m)/2, m/2\} \tag{20}$$

Let $E_{sat}(S, \alpha_D)$ be the relative error of the solution α_D to the instance S of the 5-OCCURRENCE MAX-2SAT problem, and let $E_{cut}(G_S, D)$ be the relative error of the solution D to the instance G_S of the MAX-CUT problem. Since each clause in S results in at most 3 clause edges in G_S, the number of edges in the graph G_S is bounded by $10n + 2m + 3m$, which is bounded by $25m$. Thus, $|D| \leq 25m$. Combining this fact with Equation (20) and Lemma 40.4, we have

$$\begin{aligned}
E_{sat}(S, \alpha_D) &= \frac{Opt_{sat}(S)}{Apx(\alpha_D)} - 1 = \frac{Opt_{sat}(S) - Apx(\alpha_D)}{Apx(\alpha_D)} \\
&\leq \frac{Opt_{sat}(S) - (|D| - 10n - 2m)/2}{m/2} \\
&= \frac{2Opt_{sat}(S) - (|D| - 10n - 2m)}{m} \\
&= \frac{(2Opt_{sat}(S) + 10n + 2m) - |D|}{m} \\
&= \frac{Opt_{cut}(G_S) - |D|}{m} \\
&\leq \frac{Opt_{cut}(G_S) - |D|}{|D|/25} \\
&= 25\left(\frac{Opt_{cut}(G_S)}{|D|} - 1\right) \\
&= 25E_{cut}(G_S, D)
\end{aligned}$$

This shows that the reduction we constructed from the 5-OCCURRENCE MAX-2SAT problem to the MAX-CUT problem is an E-reduction. We conclude with the following theorem.

Theorem 40.5 *The* 5-OCCURRENCE MAX-2SAT *problem is E-reducible to the* MAX-CUT *problem.*

By Theorem 37.13, the 5-OCCURRENCE MAX-2SAT problem is ApxPB-complete. Combining this with Theorem 40.1 and Theorem 40.5, we get

Theorem 40.6 *The* MAX-CUT *problem is ApxPB-complete. Therefore, the* MAX-CUT *problem has no polynomial time approximation scheme unless P = NP.*

To close the course, we point out that for most of the optimization problems studied in this course, we have precisely classified each of them into a proper class: some of them are polynomial time solvable, some of them are NP-hard but have fully polynomial time approximation schemes, some of them have polynomial time approximation schemes but have no fully polynomial time approximation schemes unless P = NP, and some of them have polynomial time approximation algorithms with constant ratio but have no polynomial time approximation scheme unless P = NP.

There are some optimization problems that even do not have a polynomial time approximation algorithm with constant ratio. Examples are the INDEPENDENT SET problem, the CLIQUE problem, and the TRAVELING SALESMAN problem. For example, recent research has shown that there is a constant $\epsilon > 0$ such that the INDEPENDENT SET has no polynomial time approximation algorithm with approximation ratio n^ϵ unless P = NP.

One problem for which we did not study the non-approximability is the Δ-TSP problem. Theorem 22.5 shows that the problem has a polynomial time approximation algorithm of approximation ratio 1.5. Using the E-reduction, we can show that the Δ-TSP problem is ApxPB-complete. In fact, even a weaker version, the TRAVELING SALESMAN 1-2 problem (see Lecture 17) is ApxPB-complete.

References

[1] A. V. Aho, J. E. Hopcropt, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[2] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy, Proof verification and hardness of approximation problems, *Proc. 33rd Ann. IEEE Symp. on the Foundation of Computer Science*, (1992), pp. 14-23.

[3] S. Arora and S. Safra, Probabilistic checking of proofs: a new characterization of NP, *Proc. 33rd Ann. IEEE Symp. on the Foundation of Computer Science*, (1992), pp. 2-13.

[4] G. Ausiello, P. Crescenzi, and M. Protasi, Approximate solution of NP optimization problems, *Theoretical Computer Science 150*, (1995), pp. 1-55.

[5] B. S. Baker, Approximation algorithms for NP-complete problems on planar graphs, *Journal of ACM 41*, (1994), pp. 153-180.

[6] P. Berman and G. Schnitger, On the complexity of approximating the independent set problem, *Information and Computation 96*, (1992), pp. 77-94.

[7] P. Berman and V. Ramaiyer, Improved approximations for the Steiner tree problem, *Proc. 3rd Ann. ACM-SIAM Symp. on Discrete Algorithms*, (1992), pp. 325-334.

[8] L. Cai and J. Chen, On the amount of nondeterminism and the power of verifying, *SIAM Journal on Computing*, to appear.

[9] L. Cai and J. Chen, On fixed-parameter tractability and approximability of NP-hard optimization problems, *Journal of Computer and System Sciences*, to appear.

[10] L. Cai, J. Chen, R. Downey, and M. Fellows, On the structure of parameterized problems in NP, *Information and Computation 123*, (1995), pp. 38-49.

[11] J. Chen and D. K. Friesen, The complexity of 3-dimensional matching, *Tech. Report*, Dept. Computer Science, Texas A&M University, (1995).

[12] J. Chen, S. P. Kanchi, and A. Kanevsky, On the complexity of graph embeddings, *Lecture Notes in Computer Science 709*, (1993), pp. 234-245.

[13] N. Christofides, Worst-case analysis of a new heuristic for the traveling salesman problem, *Tech. Report*, GSIA, Carnegie-Mellon University, (1976).

[14] E. G. Coffman, M. R. Garey, and D. S. Johnson, Approximation algorithms for bin packing – an updated survey, in *Algorithm Design for Computer System Design*, (ed. G. Ausiello, M. Lucertini, and P. Serafini), Springer-Verlag, 1984.

[15] P. Crescenzi and V. Kann, A compendium of NP optimization problems, *Manuscript*, (1995).

[16] P. Crescenzi and A. Panconesi, Completeness in approximation classes, *Information and Computation 93*, (1991), pp. 241-262.

[17] G. B. Dantzig, *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1963.

[18] E. A. Dinits, Algorithm for solution of a problem of maximum flow in a network with power estimation, *Soviet Math. Dokl. 11*, (1970), pp. 1277-1280.

[19] J. Edmonds, Paths, trees and flowers, *Canad. J. Math. 17*, (1965), pp. 449-467.

[20] J. Edmonds and R. M. Karp, Theoretical improvements in algorithmic efficiency for network flow problems, *Journal of ACM 19*, (1972), pp. 248-264.

[21] R. Fagin, Generalized first-order spectra and polynomial-time recognizable sets, *SIAM-AMS Proc.*, (1974), pp. 43-73.

[22] W. Fernandez de la Vega and G. S. Lueker, Bin packing can be solved within $1+\epsilon$ in linear time, *Combinatorica 1*, (1981), pp. 349-355.

[23] L. R. Ford and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.

[24] D. K. Friesen, Tighter bounds for the multifit processor scheduling algorithm, *SIAM Journal on Computing 13*, (1984), pp. 170-181.

[25] M. R. Garey and D. S. Johnson, Strong NP-completeness results: motivation, examples, and implications, *Journal of ACM 25*, (1978), pp. 499-508.

[26] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, Freeman, San Fransico, CA, 1979.

[27] M. X. Goemans and D. P. Williamson, .878-approximation algorithms for MAX CUT and MAX 2SAT, *Proc. 26st Ann. ACM Symp. on Theory of Computing*, (1994), pp. 422-431.

[28] R. L. Graham, Bounds for certain multiprocessing anomalies, *Bell Systems Technical Journal 45*, (1966), pp. 1563-1581.

[29] M. Grigni, E. Koutsoupias, and C. Papadimitriou, An approximation scheme for planar graph TSP, *Proc. 36st Ann. IEEE Symp. on the Foundation of Computer Science*, (1995), to appear.

[30] D. S. Hochbaum, Approximation algorithms for the set covering and vertex cover problems, *SIAM Journal on Computing 3*, (1982), pp. 555-556.

[31] D. S. Hochbaum and D. B. Shmoys, Using dual approximation algorithms for scheduling problems: theoretical and practical results, *Journal of ACM 34*, (1987), pp. 144-162.

[32] D. S. Hochbaum and D. B. Shmoys, A polynomial approximation scheme for scheduling on uniform processors: using the dual approximation approach, *SIAM Journal on Computing 17*, (1988), pp. 539-551.

[33] I. Holyer, The NP-completeness of edge coloring, *SIAM Journal on Computing 10*, (1981), pp. 718-720.

[34] J. E. Hopcroft and R. M. Karp, A $n^{5/2}$ algorithm for maximum matching in bipartite graphs, *SIAM Journal on Computing 2*, (1973), pp. 225-231.

[35] O. H. Ibarra and C. E. Kim, Fast approximation algorithms for the knapsack and sum of subset problems, *Journal of ACM 22*, (1975), pp. 463-468.

[36] D. S. Johnson, Approximation algorithms for combinatorial problems, *Journal of Computer and System Sciences 9*, (1974), pp. 256-278.

[37] D. S. Johnson, The NP-completeness column: an ongoing guide, *Journal of Algorithms 13*, (1992), pp. 502-524.

[38] V. Kann, Maximum bounded 3-dimensional matching is MAX SNP-complete, *Information Processing Letters 37*, (1991), pp. 27-35.

[39] N. Karmakar, A new polynomial-time algorithm for linear programming, *Combinatorica 4*, (1984), pp. 373-395.

[40] N. Karmakar and R. M. Karp, An efficient approximation scheme for the one-dimensional bin packing problem, *Proc. 23rd Ann. IEEE Symp. on Foundation of Computer Science*, (1982), pp. 312-320.

[41] D. Karger, R. Motwani, and G. D. S. Ramkumar, On approximating the longest path in a graph, *Lecture Notes in Computer Science 709*, (1993), pp. 421-432.

[42] A. V. Karzanov, Determining the maximum flow in the network with the method of preflows, *Soviet Math. Dokl. 15*, (1974), pp. 434-437.

[43] S. Khanna, R. Motwani, M. Sudan, and U. Vazirani, On syntactic versus computational views of approximability, *Proc. 35th Ann. IEEE Symp. on Foundation of Computer Science*, (1994), pp. 819-836.

[44] D. E. Knuth, *The Art of Computer Programming. Volume III: Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973.

[45] P. G. Kolaitis and M. N. Thakur, Logical definability of NP optimization problems, *Information and Computation 115*, (1994), pp. 321-353.

[46] P. G. Kolaitis and M. N. Thakur, Approximation properties of NP minimization classes, *Journal of Computer and System Sciences 50*, (1995), pp. 391-411.

[47] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart&Winston, 1976.

[48] H. W. Lenstra, Integer programming with a fixed number of variables, *Mathematics of Operations Research 8*, (1983), pp. 538-548.

[49] R. J. Lipton and R. E. Tarjan, A separator theorem for planar graphs, *SIAM J. Appl. Math. 36*, (1979), pp. 177-189.

[50] R. J. Lipton and R. E. Tarjan, Applications of a planar separator theorem, *SIAM Journal on Computing 9*, (1980), pp. 615-627.

[51] C. Lund and M. Yannakakis, On the hardness of approximating minimization problems, *Journal of ACM 41*, (1994), pp. 960-981.

[52] S. Micali and V. V. Vazirani, An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs, *Proc. 21st Ann. IEEE Symp. on the Foundation of Computer Science*, (1980), pp. 17-27.

[53] B. Monien, How to find long paths efficiently, *Annals of Discrete Mathematics 25*, (1985), pp. 239-254.

[54] R. Motwani, *Lecture Notes on Approximation algorithms*, Dept. of Computer Science, Stanford University, 1995.

[55] C. H. Papadimitriou, *Combinatorial Complexity*, Addison-Wesley, Reading, MA, 1993.

[56] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Englewood Cliffs, NJ: Prentice Hall, 1982.

[57] C. H. Papadimitriou and M. Yannakakis, Optimization, approximation, and complexity classes, *Journal of Computer and System Sciences 43*, (1991), pp. 425-440.

[58] C. H. Papadimitriou and M. Yannakakis, The traveling salesman problem with distances one and two, *Mathematics of Operations Research 18*, (1993), pp. 1-11.

[59] C. H. Papadimitriou and M. Yannakakis, On limited nondeterminism and the complexity of the V-C dimension, *Journal of Computer and System Sciences*, (1995), to appear.

[60] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985.

[61] S. Sahni, Algorithms for scheduling independent tasks, *Journal of ACM 23*, (1976), pp. 116-127.

[62] S. Sahni and T. Gonzalez, P-complete approximation problems, *Journal of ACM 23*, (1976), pp. 555-565.

[63] D. B. Shmoys, Computing near-optimal solutions to combinatorial optimization problems, *DIMACS Series in Discrete Mathematics*, (1995), to appear.

[64] V. G. Vizing, On an estimate of the chromatic class of a p-graph (in Russian), *Diskret. Analiz 3*, (1964), pp. 23-30.

[65] M. Yannakakis, On the approximation of maximum satisfiability, *Journal of Algorithms 17*, (1994), pp. 475-502.

[66] D. Zuckerman, On unapproximable versions of NP-complete problems, *SIAM Journal on Computing*, (1995), to appear.

Contents

List of Figures