RC 21273 (09/03/98)
Computer Science/Mathematics

IBM Research Report

Lecture Notes on Approximation Algorithms Spring 1998

David P. Williamson

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

Lecture Notes on Approximation Algorithms

David P. Williamson

Spring 1998

Contents

Preface

The contents of this book are lecture notes from a class taught in the Department of Industrial Engineering and Operations Research of Columbia University during the Spring 1998 term (IEOR 6610E: Approximation Algorithms). The notes were created via the "scribe" system, in which each lecture one student was responsible for turning their notes into a LaTeX document. I then edited the notes, and made copies for the entire class. The students in the class who served as scribes were M. Tolga Cezik, David de la Nuez, Mayur Khandewal, Eurico Lacerda, Yiqing Lin, Jörn Meißner, Olga Raskina, R. N. Uma, Xiangdong Yu, and Mark Zuckerberg. Any errors which remain (or were there to begin with!) are, of course, entirely my responsibility.

David P. Williamson
Yorktown Heights, NY

IEOR E6610 Approximation Algorithms	January 21, 1998
Lecture 1	
Lecturer: David P. Williamson	*Scribe: David de la Nuez*

1.1 First Plunge: A look at the Traveling Salesman Problem (TSP)

The first problem we will consider today is the Traveling salesman problem.

Traveling salesman problem

- **Input:**
  - Undirected graph $G = (V, E)$
  - costs $c_e \geq 0 \;\; \forall \; e \in E$
- **Goal:** Find a *tour* of minimum cost which visits each "city" (vertex in the graph) exactly once.

There are many applications – at IBM the problem has been encountered in working on batches of steel at a steel mill.

Naive Algorithm: Try all tours!

Why is it so naive? It would run too slowly, because the running time is $O(n!)$ where $n = |V|$. We need a better algorithm. Edmonds and Cobham were the first to suggest that a "good" algorithm is one whose running time is a polynomial in the "size" of the problem. Unfortunately, we don't know if such an algorithm exists for the TSP. What we do know, thanks to Cook and Karp, is that the existence of such an algorithm implies that $P = NP$. A lot of very intelligent people don't believe this is the case, so we need an alternative! We have a couple of options:

1. Give up on polynomial-time algorithms and hope that in practice our algorithms will run fast enough on the instances we want to solve (*e.g.* IP branch-and-bound methods).
2. Give up on optimality and try some of these approaches:
   (a) heuristics
   (b) local search
   (c) simulated annealing

(d) tabu search

(e) genetic algorithms

(f) *approximation algorithms*

1.2 Some definitions and examples

Definition 1.1 An algorithm is an α-*approximation algorithm* for an optimization problem Π if

1. The algorithm runs in polynomial time
2. The algorithm always produces a solution which is within a factor of α of the value of the optimal solution.

Note that throughout the course we use the following convention: For minimization problems, $\alpha > 1$, while for maximization problems, $\alpha < 1$ (α is known as the "performance guarantee"). Also, keep in mind that in the literature, researchers often speak of $1/\alpha$ for maximization problems.

So, why do we study approximation algorithms?

1. As algorithms to solve problems which need a solution.
2. As ideas for #1.
3. As a mathematically rigorous way of studying heuristics.
4. Because it's fun!
5. Because it tells us how hard problems are.

Let us briefly touch on item 5 above, beginning with another definition:

Definition 1.2 A *polynomial-time approximation scheme (PTAS)* is a family of algorithms $A_\epsilon : \epsilon > 0$ such that for each $\epsilon > 0$, A_ϵ is a $(1+\epsilon)$-approximation algorithm which runs in polynomial time in input size for fixed ϵ.

Some problems which have the *PTAS* property are knapsack, Euclidean TSP (Arora 1996, Mitchell 1996), and some scheduling problems. Other problems like *MAX SAT* and *MAX CUT* are harder:

Theorem 1.1 (Arora, Lund, Motwani, Sudan, Szegedy 1992) There does not exist a *PTAS* for any *MAX SNP-hard* problem unless $P = NP$.

There is a similarly exotic result with respect to *CLIQUE*:

Theorem 1.2 (Håstad 1996) There does not exist a $O(n^{1-\epsilon})$ approximation algorithm for any $\epsilon > 0$ for *MAX CLIQUE* unless $NP = RP$.

What is *MAX CLIQUE*? Given a graph $G = (V, E)$, find the clique $S \subset V$ of maximum size $|S|$. And what is a clique?

Definition 1.3 A *clique* S is a set of vertices for which each vertex pair has its corresponding edge included (that is, $i \in S$, $j \in S$ implies $(i, j) \in E$).

1.3 Vertex Cover: Three ways to skin a cat

1.3.1 The Vertex Cover Problem

Consider the following problems:

Vertex Cover (VC)

- **Input:** An undirected graph $G = (V, E)$
- **Goal:** Find a set $C \in V$ of minimum size $|C|$ such that $\forall (i, j) \in E$, we have either $i \in C$ or $j \in C$.

Weighted Vertex Cover (WVC)

- **Input:**
  - An undirected graph $G = (V, E)$
  - Weights $w_i \geq 0 \ \forall i \in V$
- **Goal:** Find a vertex cover C which minimizes $\sum_{i \in C} w_i$.

1.3.2 First attempt: a greedy algorithm

Greedy

$C \leftarrow \emptyset$
while $E \neq \emptyset$
    $(*)$ choose $(i, j) \in E$
    $C \leftarrow C \cup \{i, j\}$
    Delete (i, j) from E and all edges adjacent to i and j.

Now we prove that the above is a 2-approximation algorithm:

Lemma 1.3 Greedy returns a vertex cover.

Proof: Edges are only deleted when they are covered, and at termination, $E = \emptyset$. $\square$

Theorem 1.4 (Gavril) Greedy is a 2-approximation algorithm.

Proof: It is clear that it runs in polynomial time. Now, suppose the algorithm goes through the while loop X times. By construction, each edge chosen in $(*)$ must be covered by a different vertex in the optimal solution, so $X \leq OPT$, and thus $|C| = 2X \leq 2OPT$. $\square$

This algorithm, however, will not be very useful for the weighted problem, so we skin the cat another way now.

1.3.3 Second attempt: formulate as an integer program (IP)

The technique we use here is as follows:

1. Formulate the problem as an IP
2. Relax to a Linear Program (LP)
3. Use the LP (and its solution) to get a solution to the IP

We will use this technique 1 million times in this course.

So, let's apply it to the WVC problem. Here is the IP formulation

$$
\begin{aligned}
&\text{Min} \quad \sum_{i \in V} w_i x_i \\
&\text{subject to:} \\
&\qquad \sum_{i: e_i \in S_j} x_i + x_j \geq 1 \qquad\qquad \forall (i,j) \in E \\
&\qquad x_i \in \{0,1\} \qquad\qquad \forall i \in V,
\end{aligned}
$$

and the corresponding LP relaxation

$$
\begin{aligned}
&\text{Min} \quad \sum_{i \in V} w_i x_i \\
&\text{subject to:} \\
&\qquad \sum_{i: e_i \in S_j} x_i + x_j \geq 1 \qquad\qquad \forall (i,j) \in E \\
&\qquad x_i \geq 0 \qquad\qquad \forall i \in V.
\end{aligned}
$$

Now, suppose that Z_{LP} is the optimal value of the LP. Then

$$Z_{LP} \leq OPT.$$

This follows since any solution feasible for the IP is feasible for the LP. Thus the value of the optimal LP will be no greater than that for the IP. This is a fact we will use many, many times throughout the course. In addition, we use it in analyzing the algorithm below:

Round

```
Solve LP relaxation to get optimal x*
C ← ∅
∀i ∈ V
    if x*_i ≥ 1/2
        C ← C ∪ {i}.
```

Lemma 1.5 Round produces a vertex cover.

Proof: Suppose by way of contradiction $\exists (i,j) \in E$ *s.t.* $i, j \notin C$. Then $x_i^*, x_j^* < 1/2$. But then $x_i^* + x_j^* < 1$, which contradicts the feasibility of x^*. $\square$

Theorem 1.6 (Nemhauser, Trotter 1975; Hochbaum 1983) Round is a 2-approximation algorithm.

Proof: It is clear that it runs in polynomial time. Now, observe that

$$\begin{aligned} i \in C &\Rightarrow x_i^* \geq 1/2 \\ &\Rightarrow 2x_i^* \geq 1 \\ &\Rightarrow 2x_i^* w_i \geq w_i. \end{aligned}$$

Then

$$\sum_{i \in C} w_i \leq 2 \sum_{i \in V} w_i x_i^*,$$

but then combining with our relation between the LP optimal and the true optimal we have

$$\sum_{i \in C} w_i \leq 2OPT.$$

$\square$

1.3.4 Third (and final) attempt: Dual-LP

We now consider our third and final way of skinning the Vertex Cover cat. Our first step is to take the dual of our LP:

$$\text{Max} \quad \sum_{(i,j)\in E} y_{i,j}$$

subject to:

$$\sum_{j:(i,j)\in E} y_{i,j} \leq w_i \qquad \forall i \in V$$

$$y_{(i,j)} \geq 0 \qquad \forall (i,j) \in E.$$

Observe that for any feasible dual solution y, we have

$$\sum_{(i,j)\in E} y_{i,j} \leq Z_{LP} \leq OPT,$$

by weak duality. This motivates the following algorithm:

Dual-LP

Solve dual LP to get optimal y^*
$C \leftarrow \emptyset$
$\forall i \in V$
  if $\sum_{j:(i,j)\in E} y^*_{i,j} = w_i$
    $C \leftarrow C \cup \{i\}$.

Lemma 1.7 Dual-LP produces a vertex cover.

Proof: Suppose by way of contradiction $\exists (k,l) \in E$ $s.t.$ $k, l \notin C$. Let

$$\epsilon_1 := w_k - \sum_{j:(k,j)\in E} y^*_{k,j}$$

$$\epsilon_2 := w_l - \sum_{j:(l,j)\in E} y^*_{l,j}$$

Note that by assumption, $\epsilon_1, \epsilon_2 > 0$. But then we increase $y^*_{k,l}$ by $\min\{\epsilon_1, \epsilon_2\}$ (because $y^*_{k,l}$ only appears in the constraints corresponding to ϵ_1 and ϵ_2), which leads to an increase of the objective function and contradicts the optimality of y^*. $\square$

Theorem 1.8 (Hochbaum 1982) Dual-LP is a 2-approximation algorithm.

Proof: It is clear that it runs in polynomial time. Now, by construction of C,

$$\begin{aligned}\sum_{i\in C} w_i &= \sum_{i\in C}\Big(\sum_{j:(i,j)\in E} y^*_{i,j}\Big) \\ &\leq 2\sum_{(i,j)\in E} y^*_{i,j} \\ &\leq 2OPT.\end{aligned}$$

The second inequality follows from the first since each edge $(i, j) \in E$ can appear at most twice in the double summation. The final line follows from our weak duality relation above. $\square$

To wrap up, note the following interesting result:

Theorem 1.9 (Håstad) If an α-approximation algorithm exists for vertex cover with $\alpha < 7/6$ then $P = NP$.

IEOR E6610 Approximation Algorithms January 30, 1998

Lecture 2

Lecturer: David P. Williamson *Scribe: Mayur Khandelwal*

2.1 Set Cover: description and approximation algorithms

In this lecture, we discuss the set cover problem and three approximation algorithms, as well as an application.

Set Cover

- **Input:**
  - Ground set $E = \{e_1, \ldots, e_i, \ldots, e_n\}$
  - Subsets $S_1, S_2, \ldots, S_m \subseteq E$
  - Costs w_j for each subset S_j
- **Goal:** Find a set of indices of subsets $I \subseteq \{1, \ldots, m\}$ that minimizes the sum of the weights such that each element is in at least one subset. More precisely, find a set I that minimizes $\sum_{j \in I} w_j$ such that $\bigcup_{j \in I} S_j = E$.

2.1.1 Method I: Rounding

Apply the general method of rounding described for vertex cover.

1. Formulate problem as an integer program.
2. Relax the integer requirement and solve using a polynomial-time linear programming solver (e.g. interior-point method).
3. Use the linear program solution in some undefined way to obtain an integer solution that is close in value to the linear programming solution.

Here's how we can apply the process above to the set cover problem.

1. Formulate problem as an integer program. Here, we create a variable x_j for each subset S_j. If $S_j \in I$, then $x_j = 1$, otherwise $x_j = 0$.

$$\begin{aligned} \text{Min} \quad & \sum_{j=1}^{m} w_j x_j \\ \text{subject to:} \quad & \\ & \sum_{j: e_i \in S_j} x_j \geq 1 \qquad\qquad \forall e_i \in E \\ & x_j \in \{0, 1\}. \end{aligned}$$

2. Relax the integer requirement by changing the last constraint to $x_j \geq 0$. Let OPT equal the optimal objective value for the integer program. Let z_{LP} be the optimal objective value for the linear program. Note, $z_{LP} \leq OPT$ because the solution space for the integer program is a subset of the solution space of the linear program.

3. Use the linear program solution in some undefined way to obtain a solution that is close in value to the linear programming solution. To go further, we must first define f as the following:

$$f = \max_i |\{j : e_i \in S_j\}|.$$

In plain English, f is maximum number of sets that contain any given element. Now, we have the following algorithm:

Rounding

```
Solve the linear program to get solution x*.
I ← ∅
for each S_j
    if x*_j ≥ 1/f
        I ← I ∪ {j}
```

Lemma 2.1 Rounding produces a set cover.

Proof: Suppose there is an element e_i such that $e_i \notin \bigcup_{j \in I} S_j$. Then for each set S_j, which e_i is a member of, $x_j^* < 1/f$. Then

$$\begin{aligned} \sum_{j: e_i \in S_j} x_j^* &< \frac{1}{f} \cdot |\{j : e_i \in S_j\}| \\ &\leq 1, \end{aligned}$$

since $|\{j : e_i \in S_j\}| \leq f$. But this violates the linear programming constraint for e_i.
□

Theorem 2.2 (Hochbaum '82) Rounding is an f-approximation algorithm for set cover.

Proof: It is clear that the rounding algorithm is a polytime algorithm. Furthermore,

$$\begin{aligned} \sum_{j \in I} w_j &\leq \sum_j w_j \cdot x_j^* \cdot f \\ &= f \sum_j w_j \cdot x_j^* \\ &\leq f \cdot OPT. \end{aligned}$$

The first inequality follows since $j \in I$ only if $x_j^* \cdot f \geq 1$. □

Recall that in the last lecture, we discussed 2-approximation algorithms for vertex cover.

Weighted Vertex Cover (WVC)

- **Input:**
  - An undirected graph $G = (V, E)$
  - Weights $w_i \geq 0$ $\forall i \in V$
- **Goal:** Find a vertex cover C which minimizes $\sum_{i \in C} w_i$.

We can translate this problem to the set cover problem: the edges correspond to the ground set and vertices correspond to subsets (i.e., the subset corresponding to vertex i is the set of all edges adjacent to i). Since each edge is in exactly two sets, in this case $f = 2$; thus the f-approximation algorithms for set cover translate to 2-approximation algorithms for vertex cover.

2.1.2 Method II: Dual LP

Another way to use the rounding method is to apply it to the dual solution. The dual of the linear programming relaxation for set cover is:

$$\text{Max} \quad \sum_i y_i$$

subject to:

$$\begin{aligned} \sum_{i: e_i \in S_j} y_i \leq w_j & \qquad \forall S_j \\ y_i \geq 0 & \qquad \forall e_i \in E. \end{aligned}$$

If we have feasible dual solution y, then

$$\sum_i y_i \leq z_{LP} \leq OPT$$

by weak duality. The algorithm to find an minimum-cost set cover using a dual LP is as follows:

Dual-LP

Solve the Dual linear program to get optimal solution y^*
$I \leftarrow \emptyset$
for each S_j
  if $\sum_{i:e_i \in S_j} y_i^* = w_j$
    $I \leftarrow I \cup \{j\}$.

Theorem 2.3 Dual-LP is an f-approximation algorithm.

Proof: Left as a homework exercise. □

2.1.3 Method III: Primal-Dual

The problem with the previous algorithms is that they require solving a linear program. While this can be done relatively quickly in practice, we would like algorithms that are even faster. We now turn to an algorithm that behaves much like Dual-LP above, but constructs its own dual solution, rather than finding the optimal dual LP solution.

Primal-Dual

$I \leftarrow \emptyset$
$\tilde{y}_i \leftarrow 0 \ \ \forall i$
while $\exists e_{i'} : e_{i'} \notin \bigcup_{j \in I} S_j$
  $j' = \arg\min_{j:e_{i'} \in S_j} \left\{ w_j - \sum_{k:e_k \in S_j} \tilde{y}_i \right\}$
  $\epsilon_{j'} \leftarrow w_{j'} - \sum_{i:e_i \in S_{j'}} \tilde{y}_i$
  $\tilde{y}_{i'} \leftarrow \tilde{y}_{i'} + \epsilon_{j'}$
  $I \leftarrow I \cup \{j'\}$.

Note that the function arg min returns the argument (index, in this case) that attains the minimum value.

Lemma 2.4 Primal-Dual returns a set cover.

Proof: Trivial, since the algorithm does not terminate until it does return a set cover. □

Lemma 2.5 Primal-Dual constructs a dual feasible solution.

Proof: By induction. The base case is trivial since initially

$$\sum_{i:e_i \in S_j} \tilde{y}_i = 0 \leq w_j \ \ \forall j.$$

Now assume that

$$\sum_{i:e_i \in S_j} \tilde{y}_i \leq w_j \ \ \forall j.$$

Consider first the case of S_j, where $j \neq j'$, and $e_{i'} \in S_j$. Then since $\tilde{y}_{i'}$ is increased by $\epsilon_{j'}$,

$$\begin{aligned}
\sum_{i:e_i \in S_j} \tilde{y}_i + \epsilon_{j'} &= \sum_{i:e_i \in S_j} \tilde{y}_i + (w_{j'} - \sum_{i:e_i \in S_{j'}} \tilde{y}_i) \\
&\leq \sum_{i:e_i \in S_j} \tilde{y}_i + (w_j - \sum_{i:e_i \in S_j} \tilde{y}_i) \\
&\leq w_j.
\end{aligned}$$

In the case of $S_{j'}$,

$$\begin{aligned}
\sum_{i:e_i \in S_{j'}} \tilde{y}_i + \epsilon_{j'} &= \sum_{i:e_i \in S_{j'}} \tilde{y}_i + (w_{j'} - \sum_{i:e_i \in S_{j'}} \tilde{y}_i) \\
&= w_{j'}.
\end{aligned}$$

Notice that we don't need to consider the case of a set S_j such that $e_{i'} \notin S_j$, since $\sum_{i:e_i \in S_j} \tilde{y}_i$ does not change for these sets. $\square$

Theorem 2.6 (Bar-Yehuda, Even '81) Primal-Dual is an f-approximation algorithm for the set cover problem.

Proof:

$$\begin{aligned}
\sum_{j \in I} w_j &= \sum_{j \in I} \sum_{e_i \in S_j} \tilde{y}_i \\
&\leq \sum_{1 \leq i \leq n} \tilde{y}_i |\{j : e_i \in S_j\}| \\
&\leq f \cdot \sum_i \tilde{y}_i \\
&\leq f \cdot OPT.
\end{aligned}$$

The first equality follows from the case of the set $S_{j'}$ in the previous lemma: whenever we choose a set $S_{j'}$, we know that $\sum_{i:e_i \in S_{j'}} \tilde{y}_i = w_{j'}$. The next inequality follows since each $\tilde{y}_i$ can appear in the double sum at most $|\{j : e_i \in S_j\}|$ times. The next inequality follows by the definition of f, and the last inequality follows from weak duality. $\square$

2.1.4 Method IV: Greedy Algorithm

So far every technique we have tried has led to the same result: every technique we looked at last time gave a 2-approximation algorithm for the vertex cover problem, and every technique we have looked at this time has given an f-approximation algorithm for the set cover time. It seems that we get the same results no matter what we do! This is not true in general: often cleverness will give us improved performance guarantees. And sometimes the cleverest thing to do is the most obvious thing, which is what we will try to do next, in devising a "greedy" algorithm for the set cover problem.

The intuition here is straightforward. Go through and pick sets which will give the most 'bang for the buck' – will cover the most elements yet to be covered at the lowest cost. Before examining the algorithm, we define two new items:

$$H_n \equiv 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \approx \ln n$$

$$g \equiv \max_j |S_j|$$

Greedy

$I \leftarrow \emptyset$
$\tilde{S}_j \leftarrow S_j \;\; \forall j$
while $\bigcup_{j \in I} S_j \neq E$
  $j' \leftarrow \arg\min_{j : \tilde{S}_j \neq \emptyset} \frac{w_j}{|\tilde{S}_j|}$
  $I \leftarrow I \cup \{j'\}$
  $\tilde{y}_i \leftarrow \frac{w_{j'}}{|\tilde{S}_j| \cdot H_g} \;\; \forall e_i \in \tilde{S}_{j'}$ (†)
  $\tilde{S}_j \leftarrow \tilde{S}_j - S_{j'} \;\; \forall j.$

Note: The † step has been added only for the proof and is not performed in practice.

Lemma 2.7 Greedy constructs a feasible dual solution $\tilde{y}$.

Proof: First, note one observation from the † step in the algorithm:

$$w_{j'} = H_g \sum_{i \in \tilde{S}_{j'}} \tilde{y}_i.$$

Now, pick an arbitrary set $S_j = \{e_1, e_2, \ldots, e_k\}$ and assume that greedy covers this set in index order. Thus when e_i is covered, $|\tilde{S}_j| \geq k - i + 1$. Let j' be the index of the first set chosen that covers e_i. It follows that

$$\begin{aligned}\tilde{y}_i &= \frac{w_{j'}}{|\tilde{S}_{j'}|H_g} \\ &\leq \frac{w_j}{|\tilde{S}_j|H_g} \\ &\leq \frac{w_j}{(k-i+1)H_g}\end{aligned}$$

The first inequality follows since at the step of the algorithm in which j' is chosen, it must be the case that

$$\frac{w_{j'}}{|\tilde{S}_{j'}|} \leq \frac{w_j}{|\tilde{S}_j|}.$$

We can now show that the variables $\tilde{y}$ form a feasible solution to the dual of the linear programming relaxation for set cover, since

$$\begin{aligned}\sum_{i:e_i \in S_j} \tilde{y}_i &= \sum_{i=1}^{k} \tilde{y}_i \\ &\leq \frac{w_j}{H_g}\left(\frac{1}{k} + \frac{1}{k-1} + \ldots + \frac{1}{1}\right) \\ &= \frac{w_j}{H_g} H_k \\ &\leq w_j,\end{aligned}$$

since $k = |S_j| \leq g$. □

Theorem 2.8 (Chvatal '79) Greedy is a H_g-approximation algorithm.

Proof: In the previous lemma, we observed that for any set S_j chosen to be in the set cover, $w_j = H_g \sum_{e_i \in \tilde{S}_j} \tilde{y}_i$, where $\tilde{S}_j$ is the set of elements in the set cover at the time set j is chosen.

Thus

$$\sum_{j \in I} w_j = H_g \sum_i \tilde{y}_i,$$

since each e_i will be in exactly one of the $\tilde{S}_j$ when that set is chosen.

By weak duality it follows that

$$\sum_{j \in I} w_j \leq H_g \cdot OPT.$$

□

What else can we say about set cover?

Theorem 2.9 (Lund-Yannkakis '92, Feige '96, Raz-Safra '97, Arora-Sudan '97)

- If there exists a $c \ln n$-approximation algorithm where $c < 1$ then $NP \subseteq DTIME(n^{\log^k n})$ for some k.
- There exists some $c < 1$ such that if there exists a $c \log n$-approximation algorithm for set cover, then $P = NP$.

2.2 Set Cover: An Application

In the IBM labs, the set cover problem came up in the development of the IBM AntiVirus product. Antivirus programs typically detect viruses by checking programs to see if they contain substrings of 20 or more bytes from a known virus. In terms of the set cover problem, the problem was as follows:

- Sets: Substrings of 20 or more consecutive bytes (900 in number).
- Ground Set: Known viruses. (500 in number).

By using the greedy algorithm, a solution of 190 strings was found. The value of the linear programming relaxation was 185, so the optimal solution had at least 185 strings in it. Thus the greedy solution was fairly close to optimal.

IEOR E6610 Approximation Algorithms	February 4, 1998
Lecture 3	
Lecturer: David P. Williamson	*Scribe: Olga Raskina*

3.1 Metric Traveling Salesman Problem

We now turn to a well-known problem.

Metric Traveling Salesman Problem

- **Input:**
  - Complete graph $G = (V, E)$ (graph is **complete** if $\forall i, j \in V,\ \exists (i, j) \in E$)
  - Costs $c_{i,j} \equiv c(i, j) \geq 0\ \forall (i, j) \in E$
  - $c_{i,j} = c_{j,i}\ \forall (i, j) \in E$ (symmetry property)
  - $c_{i,j} + c_{j,k} \geq c_{i,k}\ \forall i, j, k \in V$ (triangle inequality)
- **Goal:** Find a *tour* of minimum cost which visits each vertex in V exactly once.

Definition 3.1 A *tour* in (V, E) is a sequence $v_1, v_2, \ldots, v_k$ such that $v_i \in V, (v_i, v_{i+1}) \in E$ and $v_k = v_1$. The *cost* of the tour is $\sum_{1 \leq i \leq k-1} c(v_i, v_{i+1})$.

Definition 3.2 A *Hamiltonian tour* of (V, E) is a tour which visits each vertex in V exactly once.

Definition 3.3 An *Eulerian tour* of (V, E) is a tour which visits each edge in E exactly once.

Theorem 3.1 (Euler) The graph (V, E) has an Eulerian tour if and only if it is connected and every vertex has an even degree.

Lemma 3.2 Let $G = (V, E)$ be a complete graph such that edge costs obey the triangle inequality. If graph (V, E'), $E' \subseteq E$, has an Eulerian tour, then G has a Hamiltonian tour of no greater cost.

Proof: Let's take the Eulerian tour of (V, E'), $v_1, v_2, \ldots, v_i, \ldots, v_{j-1}, v_j \equiv v_i, v_{j+1}, \ldots, v_k$. We can "shortcut" the visit to v_j by considering the tour $v_1, v_2, \ldots, v_i, \ldots, v_{j-1}, v_{j+1}, \ldots, v_k$. The change in the cost of the tour is $c(v_{j-1}, v_{j+1}) - c(v_{j-1}, v_j) - c(v_j, v_{j+1}) \leq 0$ (by the triangle inequality). We can continue shortcutting vertices from the tour that are visited more than once until we have a Hamiltonian tour of no greater cost. $\square$

So we can find an Eulerian tour first and then get from it to a Hamiltonian tour.

Now consider Minimum-cost Spanning Tree (MST) of graph G. Let T be the edges of MST. Let OPT be the cost of optimal tour in G.

Lemma 3.3

$$\sum_{(i,j)\in T} c_{ij} \leq (1 - \frac{1}{n})OPT$$

Proof: Let $v_1, v_2, \ldots, v_n, v_{n+1} \equiv v_1$ be an optimal tour. Since $\sum_{1\leq i\leq n} c(v_i, v_{i+1}) \equiv OPT$, there must exist some j such that $c(v_j, v_{j+1}) \geq \frac{OPT}{n}$. But we know that $\{(v_1, v_2), (v_2, v_3)...(v_n, v_{n+1})\} - \{(v_j, v_{j+1})\}$ is a spanning tree. So

$$\sum_{(i,j)\in T} c_{i,j} \leq \sum_{1\leq i\leq n} c(v_j, v_{j+1}) - c(v_j, v_{j+1}) \leq OPT - \frac{1}{n}OPT = (1 - \frac{1}{n})OPT.$$

$\square$

DoubleTree (DT)

Find MST T of $G = (V, E)$.
Let E' be T with each edge repeated. Now every vertex has even degree, and we can find Eulerian tour.
Find Eulerian tour of (V, E').
Shortcut it to TSP tour of $G = (V, E)$. Now we get Hamiltonian tour of no greater cost.

Theorem 3.4 DoubleTree is a 2-approximation algorithm for Metric TSP.

Proof: Because of the duplication of edges the cost of the Eulerian tour of (V, E') is $2\sum_{(i,j)\in T} c_{ij}$, and thus the Hamiltonian tour is no more than this amount. By the previous lemma it is at most $2(1 - \frac{1}{n})OPT$. $\square$

DT algorithm is wasteful since it doubles all the edges. To avoid it we can try to use the fact that in every graph (and thus every spanning tree) the number of odd-degree vertices is even.

In order to get a better algorithm, we need to first define another problem, the miminum-cost perfect matching problem (MCPM).

Minimum-cost Perfect Matching (MCPM)

- **Input:** $G = (V, E)$, cost c_{ij} $\forall(i, j) \in E$, $|V|$ is even
- **Goal:** Find a min-cost set of edges M such that each vertex is adjacent to exactly one edge. (M does not necessarily exist)

The following theorem will prove useful.

Theorem 3.5 (Edmonds '67) There is a poly-time algorithm to find MCPM (if one exists).

We can now give the following algorithm.

Christofides
Find MST T Let V' be the set of all odd-degree vertices in T Let G' be the complete graph induced by V' Find a MCPM M in G' Shortcut $T+M$ to a TSP tour of G.

Lemma 3.6 $T+M$ has an Eulerian tour.

Proof: T was connected and we took all odd-degree vertices and added exactly one edge adjacent to them. □

Theorem 3.7 (Christofides '76) Christofides is a $\frac{3}{2}$-approximation algorithm.

To prove the theorem we need the following lemma.

Lemma 3.8 $\sum_{(i,j)\in M} c_{ij} \leq \frac{1}{2}OPT$

Proof: Let $v_1, v_2, \ldots, v_n, v_{n+1} \equiv v_1$ be the optimal tour. Shortcut any v_i such that $v_i \notin V'$. So we get the same tour $v_{i_1}, v_{i_2}, \ldots, v_{i_k}, v_{i_{k+1}} \equiv v_{i_1}$ and all of these vertices are in V'. Since we obtained it by shortcutting, we have $\sum_{1\leq j\leq k} c(v_{i_j}, v_{i_{j+1}}) \leq OPT$.

Since V' contains an even number of vertices, it must be the case that $k = |V'|$ is even. Then $(v_{i_1}, v_{i_2}), (v_{i_3}, v_{i_4}), \ldots, (v_{i_{k-1}}, v_{i_k})$ and $(v_{i_2}, v_{i_3}), (v_{i_4}, v_{i_5}), \ldots, (v_{i_k}, v_{i_{k+1}})$ are both matchings of V'. Thus the cost of one of them is no more than $\frac{OPT}{2}$ (since their sum is no more than OPT). □

This proves the theorem, since the cost of the Eulerian tour $T+M$ is then no more than $\frac{3}{2}OPT$.

3.2 Dynamic Programming: Knapsack

Here we consider the "knapsack problem", and show that the technique of dynamic programming is useful in designing approximation algorithms.

Knapsack

- **Input:** Set of items $\{1,\dots,n\}$. Item i has a value v_i and size s_i. Total "capacity" is B. $v_i, s_i, B \in Z^+$.
- **Goal:** Find a subset of items S that maximizes the value of $\sum_{i\in S} v_i$ subject to the constraint $\sum_{i\in S} s_i \leq B$.

We assume that $s_i \leq B\ \forall i$, since if $s_i > B$ it can never be included in any feasible solution.

We now show that dynamic programming can be used to solve the knapsack problem exactly.

Definition 3.4 Let $A(i,v) \equiv$ size of "smallest" subset of $\{1,\dots,i\}$ with value **exactly** v. (∞ if no such subset exists).

Now consider the following dynamic programming algorithm. Note that if $V = \max_i v_i$, then nV is an upper bound on the value of any solution.

DynProg

```
V = max_i v_i
For i ← 1 to n
    A(i,0) ← 0
For v ← 1 to nV
    A(1,v) ← { s_1   if v_1 = v
             { ∞     otherwise
For i ← 2 to n
    For v ← 1 to nV
        if v_i ≤ v
            A(i,v) ← min(A(i−1,v), s_i + A(i_1, v − v_i))
        else
            A(i,v) ← A(i−1,v).
```

This algorithm computes all As correctly and returns $\arg\max_v\{A(n,v) : A(n,v) \leq B\}$, which is the largest value set of items that fits in the knapsack. The running time of the algorithm is $O(n^2V)$.

It is known that knapsack problem is NP-hard. But the running time of the algorithm seems to be polynomial. Have we proven that $P = NP$? No, since input is usually represented in binary; that is, it takes $\lceil \log v_i \rceil$ bits to write down v_i. Since the running time is polynomial in $\max_i v_i$, it is exponential in the input size of the v_i. We could think of writing the input to the problem in unary (i.e., v_i bits to encode v_i), in which case the running time would be polynomial in the size of the input.

Definition 3.5 An algorithm for a problem Π with running time polynomial of input encoded in unary is called *pseudopolynomial*.

If V were some polynomial in n, then the running time would be polynomial in the input size (encoded in binary). We will now get an *approximation scheme* for knapsack by rounding the numbers so that V is a polynomial in n and applying the dynamic programming algorithm. This rounding implies some loss of precision, but we will show that it doesn't affect the final answer by too much.

Definition 3.6 A *polynomial-time approximation scheme (PTAS)* is a family of algotithms $\{A_\epsilon\}$ for a problem Π such that for each $\epsilon > 0$, A_ϵ is a $(1+\epsilon)$-approximation algorithm (for min problems) or $(1-\epsilon)$-approximation algorithm (for max problems). If the running time is also a polynomial in $\frac{1}{\epsilon}$, then A_ϵ is a fully polynomial approximation scheme (FPAS, FPTAS).

Here is our new algorithm.

DynProg2

$K \leftarrow \frac{\epsilon V}{n}$
$v_i' \leftarrow \lfloor \frac{v_i}{K} \rfloor \ \forall i$
Run DynProg on (s_i, v_i').

Theorem 3.9 DynProg2 is an FPAS for knapsack.

Proof: Let S be the set of items found by DynProg2. Let O be the optimal set. We know $V \leq OPT$, since one possible knapsack is to simply take the most valuable item. We also know, by the definition of v_i',

$$Kv_i' \leq v_i \leq K(v_i' + 1),$$

which implies

$$Kv_i' \geq v_i - K.$$

Then

$$\begin{aligned}
\sum_{i \in S} v_i &\geq K \sum_{i \in S} v_i' \\
&\geq K \sum_{i \in O} v_i' \qquad (3.1) \\
&\geq \sum_{i \in O} v_i - |O|K \\
&\geq \sum_{i \in O} v_i - nK \\
&= \sum_{i \in O} v_i - \epsilon V \\
&\geq OPT - \epsilon OPT \\
&= (1-\epsilon) OPT.
\end{aligned}$$

Inequality (3.1) follows since the set of items in S is the optimal solution for the values v'.

Furthermore, the running time is $O(n^2V') = O(n^2\lfloor\frac{V}{K}\rfloor) = O(n^3\frac{1}{\epsilon})$, so it is an FPAS. $\square$

(Lawler '79) has given an FPAS which runs in time $O(n\log\frac{1}{\epsilon}+\frac{1}{\epsilon^4})$.

3.3 Scheduling Identical Machines: List Scheduling

We now turn to a scheduling problem. In the next lecture, we will see that dynamic programming can be used to produce a PTAS for this problem as well.

Scheduling Identical Machines

- **Input:**
  - m identical machines
  - n jobs $J_1, \ldots, J_n$ to be scheduled on the machines
  - $p_1, \ldots, p_n$ the processing times of the jobs
- **Goal:** Find a schedule of jobs that minimizes the completion time of the last job.

Before we get to the PTAS, we give what is quite possibly the first known approximation algorithm.

List-scheduling

```
For i ← 1 to n
    Schedule job i on the machine that has the least work assigned to it so far.
```

Theorem 3.10 (Graham '66) List-scheduling is a 2-approximation algorithm.

Proof: We will use two lower bounds on the length of the optimal schedule. The average load $\frac{1}{m}\sum_{1\le j\le n} p_j$ is a lower bound: the best possible schedule would be if we could equally split all the processing time among all the machines, so the optimal schedule will be at least as long as this. Furthermore, the optimal schedule has to be as long as any processing time p_j.

Suppose job J_k is the last job to finish in the List-Scheduling schedule. It must be the case that no other machine is idle prior to the start of job J_k, otherwise we would have scheduled J_k on that machine. So J_k must start no later than $\frac{1}{m}\sum_{1\le j\le n} p_j \le OPT$. Then J_k must finish no later than $\frac{1}{m}\sum_{1\le j\le n} p_j + p_k \le OPT + OPT = 2OPT$. $\square$

IEOR E6610 Approximation Algorithms	February 11, 1998
Lecture 4	
Lecturer: David P. Williamson	*Scribe: Eurico Lacerda*

4.1 Scheduling Identical Machines: A PTAS

Scheduling Identical Machines

- **Input:**
  - m identical machines
  - n jobs $J_1, J_2, \ldots, J_n$
  - processing time of each job $p_1, p_2, \ldots, p_n$
- **Goal:** Schedule jobs on machines to minimize maximum completion time; i.e. partition $\{1, \ldots, n\}$ jobs into m sets $M_1, \ldots, M_m$ so as to minimize $\max_i \sum_{j \in M_i} p_j$.

In the last class, we saw that List Scheduling is a 2-approximation algorithm. In particular, we saw that we could produce a schedule of length no more than $\frac{1}{m}\sum_j p_j + \max_j p_j$, and that both $\frac{1}{m}\sum_j p_j$ and $\max_j p_j$ are lower bounds on the length of the optimal schedule. The value $\frac{1}{m}\sum_j p_j$ is sometimes called the *average load.*

We will now give a PTAS for the problem of scheduling identical machines. To do this, we will use a $(1+\epsilon)$-relaxed decision procedure.

Definition 4.1 Given ϵ and time T, a $(1+\epsilon)$-*relaxed decision procedure* returns:

no: if there is no schedule of length $\leq T$

yes: if there is a schedule of length $\leq (1+\epsilon)T$ and it returns such a schedule

Let $L \equiv \max(\max_j p_j, \dfrac{1}{m}\sum_{j=1}^{n} p_j)$. We know that $OPT \in [L, 2L]$. Our algorithm will perform *binary search* on $[L, 2L]$ with the decision procedure above, with $\epsilon' \leftarrow \frac{\epsilon}{3}$ down to an interval of size $\epsilon' L$. The first step of the binary search is:

$T = \frac{3}{2}L, \quad \epsilon' \leftarrow \frac{\epsilon}{3}$
Call relaxed decision procedure using T, ϵ'
If **no**: $OPT \in [\frac{3}{2}L, 2L]$
yes: $OPT \in [L, \frac{3}{2}L]$, we get schedule of length $(1+\epsilon')\frac{3}{2}L$.

We continue in this way until we have an interval of length $\epsilon' L$; suppose it is $[T, T+\epsilon' L]$ By induction we know that $OPT \in [T, T+\epsilon' L]$, and we also have obtained a schedule of length no more than $(T+\epsilon' L)(1+\epsilon')$. It follows that

$$\begin{aligned}
(T+\epsilon' L)(1+\epsilon') &\leq \left(T+\frac{\epsilon}{3}L\right)\left(1+\frac{\epsilon}{3}\right) \\
&\leq T\left(1+\frac{\epsilon}{3}\right)+L\left(\frac{\epsilon}{3}+\frac{\epsilon^2}{9}\right) \\
&\leq OPT\left(1+\frac{\epsilon}{3}\right)+OPT\left(\frac{\epsilon}{3}+\frac{\epsilon^2}{3}\right) \\
&= OPT\left(1+2\frac{\epsilon}{3}+\frac{\epsilon^2}{9}\right) \\
&\leq OPT(1+\epsilon),
\end{aligned}$$

which holds for $\epsilon < 1$. Computationally, $O(\log \frac{1}{\epsilon'})$ calls to the procedure are required.

We now see how to get the necessary decision procedure by reducing it to yet another decision procedure.

DecisionProcedure1

	Split jobs into small jobs ($p_j \leq \epsilon T$) and large jobs ($p_j > \epsilon T$)
	Call a $(1+\epsilon)$-*relaxed decision procedure* on large jobs (with parameters T, ϵ)
$(*)$	If it returns *no* then return **no**
	else use list scheduling of small jobs to complete schedule
$(**)$	If schedule has length $> (1+\epsilon)T$ then return **no**
$(***)$	else return **yes**

Lemma 4.1 DecisionProcedure1 is a $(1+\epsilon)$-decision procedure.

Proof: It is trivial when **no** is returned in line $(*)$ or **yes** is returned in line $(***)$. We only need to consider the case in which **no** is returned in line $(**)$. This implies that some machine is busy at time $(1+\epsilon)T$, which implies that all machines are busy at time T, since small jobs have length no more than ϵT. This means that the average load is greater than T, which implies that there can be no schedule of length less than or equal to T. □

Now, we only need to find a $(1+\epsilon)$-relaxed decision procedure for large jobs. Suppose only k different-sized large jobs exist. Then, we can give a decision procedure based on dynamic programming that returns **yes**/**no** if schedule of length $\leq T$ exists (and return such a schedule if it does.)

To do this, let a_i denote the number of jobs of size i, let $(a_1, \ldots, a_k)$ denote a set of jobs, and let $M(a_1, \ldots, a_k)$ denote the number of machines needed to schedule this set of jobs by time T. Suppose there are n_i large jobs of size i. Clearly, $\sum_i n_i \leq n$.

```
BigJobDecProc

Let Q = {(a_1, ..., a_k) : (a_1, ..., a_k) can be scheduled on one machine
                           by time T, a_i ≤ n_i, ∀i}
M(a_1, ..., a_k) ← 1, ∀(a_1, ..., a_k) ∈ Q
M(0, 0, ..., 0) ← 0
For a_1 ← 1 to n_1
    For a_2 ← 1 to n_2
        ...
            For a_k ← 1 to n_k
                If (a_1, ..., a_k) ∉ Q
                    M(a_1, ..., a_k) ← 1 + min_{(b_1,...,b_k)∈Q : b_i ≤ a_i} M(a_1 − b_1, a_2 − b_2, ..., a_k − b_k)
If M(n_1, ..., n_k) ≤ m return yes
                else return no
```

The running time of this algorithm is $O(n^{2k})$: we execute the innermost statement of the nested loop at most $O(n^k)$ times, and the innermost statement takes $O(n^k)$ time, since there can be at most $O(n^k)$ elements in Q.

This does not yet give us a polynomial-time algorithm, since we might have as many as n different job sizes. But as with the Knapsack problem, we can cut down the number of job sizes by rounding the data; this will lead only to a small loss of precision. We do that with the following procedure.

```
RealBigJobDecProc

For each large job j
    If p_j ∈ [Tε(1 + ε)^i, Tε(1 + ε)^{i+1})
        p'_j ← Tε(1 + ε)^i
Run BigJobDecProc on p'_j
If it returns no then return no
    else return yes and same schedule, with p_j substituted for p'_j
```

Note that the job lengths p'_j are: $T\epsilon, T\epsilon(1+\epsilon), T\epsilon(1+\epsilon)^2, \ldots, T\epsilon(1+\epsilon)^l = T$, so that $l = O(\log_{1+\epsilon} \epsilon)$. Thus the number of different job sizes is at most $l+1$, which implies that the running time is $O(n^{2(l+1)}) = O(n^{\log_{(1+\epsilon)} 1/\epsilon})$.

Lemma 4.2 RealBigJobDecProc is a $(1+\epsilon)$-relaxed decision procedure for the large jobs.

Proof: If it returns **no** the algorithm is correct since $p'_j \leq p_j$. If it returns **yes** then each job's running time goes from $p'_j \to p_j$, an increase of at most a factor of $(1+\epsilon)$. Since schedule for p'_j has length no more than T, the same schedule for p_j has length no more than $(1+\epsilon)T$. □

The algorithm and analysis given above is due to Hochbaum and Shmoys (1982).

Up until now, we have studied fairly "classical" approximation algorithms (whatever this means in a field less than 40 years old). From this point in the course onwards, we will look at more modern techniques.

4.2 Randomization

We begin by looking at randomized approximation algorithms. We allow our algorithms the additional step `random(p)` which has the property

$$\text{random}(p) = \begin{cases} 1 & \text{w/ prob. } p \\ 0 & \text{w/ prob. } 1-p \end{cases}$$

Definition 4.2 A *randomized α-approximation algorithm* runs in randomized polynomial time and outputs a solution of value within a factor of α of the optimal value:

- With high probability ($\geq 1 - \frac{1}{n^c}$, for some c)

  OR

- In expectation over random choices of algorithm

4.2.1 The Maximum Cut Problem

We begin looking at randomized algorithms by looking at the maximum cut problem, sometimes called MAX CUT for short.

Maximum cut (MAX CUT)

- **Input:**
  - Undirected graph $G = (V, E)$
  - Weights $w_{ij} \geq 0$ for $(i,j) \in E$; assume $w_{ij} = 0$ for $(i,j) \notin E$
- **Goal:** Find $S \subseteq V$ that maximizes $\sum_{\substack{i \in S \\ j \notin S}} w_{ij}$.

Unlike MIN CUT, this is an NP-hard problem, even if $w_{ij} \in \{0, 1\}$. For notational simplicity, assume $V = \{1, \ldots, n\}$.

What is the dumbest algorithm you could possibly have?

DumbRandom

$S \leftarrow \emptyset$
For $i \leftarrow 1$ to n
  If random $(1/2) = 1$
    $S \leftarrow S \cup \{i\}$

Theorem 4.3 ($\approx$ Sahni, Gonzalez '76) DumbRandom is a 1/2-approximation algorithm.

Proof: Let the random variables:

$$X_{ij} = \begin{cases} 1 & \text{if } i \in S, j \notin S \text{ or } i \notin S, j \in S \\ 0 & \text{otherwise} \end{cases}$$

and let

$$W = \sum_{i<j} w_{ij} X_{ij}.$$

Let us consider the expected value of W, which is the value of the cut obtained by the randomized algorithm. Then

$$\begin{aligned} E[W] &= E[\sum_{i<j} w_{ij} X_{ij}] \\ &= \sum_{i<j} w_{ij} E[X_{ij}] \\ &= \sum_{i<j} w_{ij} Pr[i \in S, j \notin S \text{ or } i \notin S, j \in S] \\ &= \frac{1}{2} \sum_{i<j} w_{ij} \\ &\geq \frac{1}{2} OPT, \end{aligned}$$

since $\sum_{i<j} w_{ij}$ must certainly be an upper bound on the value of a maximum cut.

$\square$

4.2.2 The Maximum Satisfiability Problem

We now turn to yet another problem, the maximum satisfiability problem, sometimes called MAX SAT for short.

Maximum satisfiability (MAX SAT)

- **Input:**
  - n Boolean variables $x_1, x_2, \ldots, x_n$ (i.e., $x_i =$ TRUE or FALSE)
  - m Clauses $C_1, C_2, \ldots, C_m$ (e.g., $C_3 = x_1 \vee x_3 \vee \overline{x}_5 \vee x_{17}$)
  - Weights w_j for each clause C_j
- **Goal:** Find an assignment of TRUE/FALSE to each x_i that maximizes weight of satisfied clauses (i.e., has a positive literal set to TRUE or a negative literal set to FALSE).

We consider a randomized approximation algorithm for this problem that looks suspiciously familar.

DumbRandom2

```
For i ← 1 to n
    If random (1/2) = 1
        x_i = TRUE
    else
        x_i = FALSE
```

Theorem 4.4 ($\approx$ Johnson '74) DumbRandom2 is a $1/2-$approximation algorithm.

Proof: Let the random variable:

$$X_j = \begin{cases} 1 & \text{if clause } C_j \text{ is satisfied} \\ 0 & \text{otherwise} \end{cases}$$

We let the random variable W denote the value of the satisfied clauses:

$$W = \sum_j w_j X_j.$$

Suppose clause C_j has l_j literals. Then

$$\begin{aligned} E[W] &= E[\sum_j w_j X_j] \\ &= \sum_j w_j E[X_j] \\ &= \sum_j w_j Pr[C_j \text{ is satisfied}] \\ &= \sum_j w_j \left(1 - \left(\frac{1}{2}\right)^{l_j}\right) \\ &\geq \frac{1}{2}\sum_j w_j \\ &\geq \frac{1}{2} OPT, \end{aligned}$$

since $l_j \geq 1$ and certainly $\sum_j w_j \geq OPT$. $\square$

Observe that if we know that all clauses have at least k literals, then DumbRandom2 is an α-approximation algorithm with $\alpha = 1 - (\frac{1}{2})^k$. This will come in handy next time when we discuss improved approximation algorithms for MAX SAT.

IEOR E6610 Approximation Algorithms	February 18, 1998
Lecture 5	
Lecturer: David P. Williamson	*Scribe: Yiqing Lin*

5.1 Review of Johnson's Algorithm

We begin by reviewing the maximum satisfiability problem (MAX SAT) and the "dumb" randomized algorithm for it.

MAX SAT

- **Input:**
  - n boolean variables $x_1, x_2, \ldots, x_n$
  - m clauses $C_1, C_2, \ldots, C_m$ (e.g. $x_3 \vee \bar{x}_5 \vee \bar{x}_7 \vee x_{11}$)
  - weight $w_i \geq 0$ for each clause C_i
- **Goal:** Find an assignment of TRUE/FALSE for the x_i that maximizes total weight of satisfied clauses. (e.g. $x_3 \vee \bar{x}_5 \vee \bar{x}_7 \vee x_{11}$ is satisfied if x_3 is set TRUE, x_5 is set FALSE, x_7 is set FALSE, or x_{11} is set TRUE).

DumbRandom

For $i \leftarrow 1$ to n
    If random$(\frac{1}{2}) = 1$
        $x_i \leftarrow$ TRUE
    else
        $x_i \leftarrow$ FALSE.

Theorem 5.1 ($\approx$ Johnson '74) *DumbRandom* is a $\frac{1}{2}$-approximation algorithm.

Proof: Consider a random variable X_j such that

$$X_j = \begin{cases} 1 & \text{if clause } j \text{ is satisfied} \\ 0 & \text{otherwise.} \end{cases}$$

Let

$$W = \sum_j w_j X_j.$$

Then

$$\begin{aligned} E[W] &= \sum_j w_j E[X_j] = \sum_j w_j \Pr[\text{clause } j \text{ is satisfied}] \\ &= \sum_j w_j \left(1 - \left(\frac{1}{2}\right)^{l_j}\right) \geq \frac{1}{2}\sum_j w_j \geq \frac{1}{2} OPT, \end{aligned}$$

where $l_j = \#$ literals in clause j, since $l_j \geq 1$ and the sum of the weights of all clauses is an upper bound on the value of an optimal solution. □

Observe that if $l_j \geq k\ \forall j$, then we have a $(1 - (\frac{1}{2})^k)-$approximation algorithm. Thus Johnson's algorithm is bad when clauses are short, but good if clauses are long.

Although this seems like a pretty naive algorithm, a recent theorem shows that in fact this is the best that can be done in some cases. MAX E3SAT is the subset of MAX SAT instances in which each clause has exactly three literals in it. Note that Johnson's algorithm gives a $\frac{7}{8}$-approximation algorithm in this case.

Theorem 5.2 (Håstad '97) If *MAX E3SAT* has an α-approximation algorithm, $\alpha > \frac{7}{8}$, then $P = NP$.

5.2 Derandomization

We can make the algorithm deterministic using Method of Conditional Expectations (Spencer, Erdös). This method is very general, and allows for the derandomization of many randomized algorithms.

Derandomized Dumb

```
For i ← 1 to n
    W_T ← E[W|x_1, x_2, ..., x_{i-1}, x_i ← TRUE]
    W_F ← E[W|x_1, x_2, ..., x_{i-1}, x_i ← FALSE]
    If W_T ≥ W_F
        x_i ← TRUE
    else
        x_i ← FALSE.
```

How do we calculate $E[W|x_1, x_2, \ldots, x_i]$ in this algorithm? By linearity of expectations, we know that

$$E[W \mid x_1, x_2, \ldots, x_i] = \sum_j w_j E[X_j \mid x_1, x_2, \ldots, x_i].$$

Furthermore, we know that

$$E[X_j \mid x_1, x_2, \ldots, x_i] = \Pr[\text{clause } j \text{ is satisfied } \mid x_1, \ldots, x_i].$$

It is not hard to determine that

$$\Pr(\text{clause } j \text{ is satisfied } \mid x_1, \ldots, x_i) = \begin{cases} 1 & \text{if } x_1, \ldots, x_i \text{ already satisfy clause } j \\ 1 - (\frac{1}{2})^k & \text{otherwise when } k{=}\# \text{ variables of } x_{i+1}, \ldots, x_n \text{ in clause } j \end{cases}$$

Consider, for example, the clause $x_3 \vee \bar{x}_5 \vee \bar{x}_7 \vee x_{11}$. It is not hard to see that

$$\Pr[\text{clause satisfied } \mid x_1 \leftarrow T, x_2 \leftarrow F, x_3 \leftarrow T, x_4 \leftarrow F] = 1,$$

since $x_3 \leftarrow T$ satisfies the clause. On the other hand,

$$\Pr[\text{clause satisfied } \mid x_1 \leftarrow T, x_2 \leftarrow F, x_3 \leftarrow F, x_4 \leftarrow F] = 1 - \left(\frac{1}{2}\right)^3 = \frac{7}{8},$$

since only the "bad" settings of x_5, x_7, and x_{11} will make the clause unsatisfied.

Why does this give a $\frac{1}{2}$-approximation algorithm?

$$\begin{aligned} E[W|x_1, x_2, \ldots, x_{i-1}] &= \Pr[x_i = TRUE] E[W|x_1, \ldots, x_{i-1}, x_i \leftarrow TRUE] \\ &+ \Pr[x_i = FALSE] E[W|x_1, \ldots, x_{i-1}, x_i \leftarrow FALSE]. \end{aligned}$$

By construction of the algorithm, after setting x_i

$$E[W|x_1, x_2, \ldots, x_i] \geq E[W|x_1, x_2, \ldots, x_{i-1}].$$

Therefore,

$$E[W|x_1, \ldots, x_n] \geq E[W] \geq \frac{1}{2} OPT.$$

Notice that $E[W \mid x_1, \ldots, x_n]$ is the value of the solution using the algorithm.

The Method of Conditional Expectations allows us to give deterministic variants of randomized algorithms for most of the randomized algorithms we discuss. Why discuss the randomized algorithms, then? It turns out that usually the randomized algorithm is easier to state and analyze that its corresponding deterministic variant.

5.3 Flipping Bent Coins

As a stepping stone to better approximation algorithms for the maximum satisfiability problem, we consider what happens if we bias the probabilities for each boolean variable. To do this, we restrict our attention for the moment to MAX SAT instances in which all length 1 clauses are not negated.

Bent Coin

For $1 \leftarrow 1$ to n
    If random(p) $= 1$
        $x_i \leftarrow$ TRUE
    else
        $x_i \leftarrow$ FALSE.

We assume $p \geq \frac{1}{2}$.

Lemma 5.3 Pr[clause j is satisfied] $\geq \min(p, 1-p^2)$

Proof: If $l_j = 1$ then

$$\Pr[C_j \text{ is satisfied}] = p,$$

since every length 1 clause appears positively. If $l_j \geq 2$ then

$$\Pr[C_j \text{ is satisfied}] \geq 1 - p^2.$$

This follows since $p \geq \frac{1}{2} \geq (1-p)$. For example, for the clause $\bar{x}_1 \vee \bar{x}_2$,

$$\Pr[\text{clause is satisfied}] = 1 - p \cdot p = 1 - p^2,$$

while for $\bar{x}_1 \vee x_2$,

$$\Pr[\text{clause is satisfied}] = 1 - p(1-p) \geq 1 - p^2.$$

$\square$

We set $p = 1 - p^2 \Rightarrow p = \frac{1}{2}(\sqrt{5} - 1) \approx 0.618$.

Theorem 5.4 (Lieberherr, Specker '81) *Bent Coin* is a p-approximation algorithm for MAX SAT when all length 1 clauses are not negated.

Proof:

$$E[W] = \sum_j w_j \Pr[C_j \text{ is satisfied}] \geq p \sum_j w_j \geq p \cdot OPT.$$

$\square$

5.4 Randomized Rounding

We now consider what would happen if we tried to give different biases to determine each x_i. To do that, we go back to our general technique for deriving approximation algorithms. Recall that our general technique is:

1. Formulate the problem as an integer program.

2. Relax it to a linear program and solve.

3. Use the solution (somehow) to obtain an integer solution close in value to LP solution.

We now consider a very general technique introduced by Raghavan and Thompson, who use randomization in Step 3.

Randomized Rounding (Raghavan, Thomson '87)

1. Create an integer program with decision variables $x_i \in \{0,1\}$.
2. Get an LP solution with $0 \le x_i^* \le 1$.
3. To get an integer solution:

   If random$(x_i^*) = 1$
   $\quad x_i \leftarrow 1$
   else
   $\quad x_i \leftarrow 0$

We now attempt to apply this technique to MAX SAT.

Step 1: We model MAX SAT as the following integer program, in which we introduce a variable z_j for every clause and a variable y_i for each boolean variable x_i.

$$\text{Max} \quad \sum_j w_j z_j$$

subject to:

$$\sum_{i \in I_j^+} y_i + \sum_{i \in I_j^-} (1 - y_i) \ge z_j \qquad \forall C_j : \bigvee_{i \in I_j^+} x_i \vee \bigvee_{i \in I_j^-} \bar{x}_i$$

$$y_i \in \{0,1\}$$

$$0 \le z_j \le 1.$$

Step 2: To obtain an LP, we relax $y_i \in \{0,1\}$ to $0 \le y_i \le 1$. Note that if z_{LP} is the LP optimum and OPT is the integral optimum, then $z_{LP} \ge OPT$.

Step 3: Now applying randomized rounding gives the following algorithm:

Random Round

Solve LP, get solution (y^*, z^*)
For $i \leftarrow 1$ to n
$\quad$ If random$(y_i^*) = 1$
$\quad\quad x_i \leftarrow$ TRUE
$\quad$ else
$\quad\quad x_i \leftarrow$ FALSE.

Theorem 5.5 (Goemans, W '94) *Random Round* is a $(1 - \frac{1}{e})$-approximation algorithm, where $1 - \frac{1}{e} \approx 0.632$.

Proof: We need two facts to prove this theorem.

Fact 5.1

$$\sqrt[k]{a_1 a_2 \dots a_k} \le \frac{1}{k}(a_1 + a_2 + \cdots + a_k)$$

for nonnegative a_i.

Fact 5.2 If $f(x)$ is concave on $[l,u]$ (that is, $f''(x) \leq 0$ on $[l,u]$), and $f(l) \geq al+b$ and $f(u) \geq au+b$, then

$$f(x) \geq ax+b \quad \text{on } [l,u].$$

Consider first a clause C_j of the form $x_1 \vee x_2 \vee \ldots \vee x_k$. Notice that the corresponding LP constraint is $\sum_{i=1}^k y_i^* \geq z_j^*$.

$$\begin{aligned}
\Pr[\text{clause is satisfied}] &= 1 - \prod_{i=1}^{k}(1-y_i^*) \\
&\geq 1 - \left(\frac{k - \sum_{i=1}^k y_i^*}{k}\right)^k && (5.1) \\
&\geq 1 - \left(1 - \frac{z_j^*}{k}\right)^k && (5.2) \\
&\geq \left[1 - \left(1 - \frac{1}{k}\right)^k\right] z_j^*, && (5.3)
\end{aligned}$$

where (5.1) follows from Fact 1, (5.2) follows by the LP constraint, and (5.3) follows by Fact 2, since

$$\begin{aligned}
z_j^* = 0 \quad &\Rightarrow \quad 1-(1-z_j^*/k)^k = 0 \\
z_j^* = 1 \quad &\Rightarrow \quad 1-(1-z_j^*/k)^k = 1 - \left(1 - \frac{1}{k}\right)^k
\end{aligned}$$

and $1-(1-z_j^*/k)^k$ is concave.

We now claim that this inequality holds for any clause, and we prove this by example. Consider now the clause $x_1 \vee x_2 \vee \ldots \vee x_{k-1} \vee \bar{x}_k$. Then

$$\begin{aligned}
\Pr[\text{clause is satisfied}] &= 1 - \prod_{i=1}^{k-1}(1-y_i^*)y_k^* \\
&= 1 - \left(\frac{(k-1) - \sum_{i=1}^{k-1} y_i^* + y_k^*}{k}\right).
\end{aligned}$$

However, since $\sum_{i=1}^{k-1} y_i^* + (1-y_k^*) \geq z_j^*$ for this clause, the result is the same.

Therefore,

$$\begin{aligned}
E[W] &= \sum_j w_j \Pr[\text{clause } j \text{ is satisfied}] \\
&\geq \min_k \left[1 - \left(1 - \frac{1}{k}\right)^k\right] \sum_j w_j z_j^* \\
&\geq \min_k \left[1 - \left(1 - \frac{1}{k}\right)^k\right] \cdot OPT \geq \left(1 - \frac{1}{e}\right) \cdot OPT,
\end{aligned}$$

since $(1-\frac{1}{x})^x$ converges to e^{-1} from below. $\square$

Observe that this algorithm does well when all clauses are short. If $l_j \leq k$ for all j, then the performance guarantee becomes $1-(1-1/k)^k$.

5.5 A Best-of-Two Algorithm for MAX SAT

In the previous section we used the technique of randomized rounding to improve a .618-approximation algorithm (for a subclass of MAX SAT) to a .632-approximation algorithm for MAX SAT. This doesn't seem like much of an improvement.

But notice that Johnson's algorithm and the randomized rounding algorithm have conflicting bad cases. Johnson's algorithm is bad when clauses are short, whereas randomized rounding is bad when clauses are long. It turns out we can get an approximation algorithm that is much better than either algorithm just by taking the best solution of the two produced by the two algorithms.

Best-of-two

> Run *DumbRandom*, get assign x^1 of weight W_1
> Run *RandomRound*, get assign x^2 of weight W_2
> If $W_1 \geq W_2$
>   return x^1
> else
>   return x^2.

Theorem 5.6 (Goemans, W'94) *Best-of-two* is a $\frac{3}{4}$-approximation algorithm for MAX SAT.

Proof:

$$
\begin{aligned}
E[\max(W_1, W_2)] &\geq E\left[\frac{1}{2}W_1 + \frac{1}{2}W_2\right] \\
&= \sum_j w_j \Big(\frac{1}{2}\Pr[\text{clause } j \text{ is satisfied by DumbRandom}] \\
&\qquad + \frac{1}{2}\Pr[\text{clause } j \text{ is satisfied by RandomRound}]\Big) \\
&\geq \sum_j w_j \left[\frac{1}{2}\left(1 - \left(\frac{1}{2}\right)^{l_j}\right) + \frac{1}{2}\left[1 - \left(1 - \frac{1}{l_j}\right)^{l_j}\right] z_j^*\right] \\
&\geq \sum_j w_j \left[\frac{3}{4} z_j^*\right] \qquad (5.4) \\
&= \frac{3}{4}\sum_j w_j z_j^* \\
&\geq \frac{3}{4} OPT.
\end{aligned}
$$

We need to prove inequality (5.4), which follows if

$$\frac{1}{2}\left(1 - \left(\frac{1}{2}\right)^{l_j}\right) + \frac{1}{2}\left[1 - \left(1 - \frac{1}{l_j}\right)^{l_j}\right] z_j^* \geq \frac{3}{4} z_j^*.$$

The cases $l_j = 1, 2$ are easy:

$$l_j = 1 \quad \Rightarrow \quad \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} z_j^* \geq \frac{3}{4} z_j^*$$
$$l_j = 2 \quad \Rightarrow \quad \frac{1}{2} \cdot \frac{3}{4} + \frac{1}{2} \cdot \frac{3}{4} z_j^* \geq \frac{3}{4} z_j^*$$

For the case $l_j \geq 3$, we take the minimum possible value of the two terms:

$$l_j \geq 3 \quad \Rightarrow \quad \frac{1}{2} \cdot \frac{7}{8} + \frac{1}{2}(1 - \frac{1}{e}) z_j^* \geq \frac{3}{4} z_j^*$$

□

The best known approximation algorithm for MAX SAT so far: ≈ 0.77-approximation algorithm.

Research question: Can you get a $\frac{3}{4}$-approximation algorithm for MAX SAT without solving an LP?

IEOR E6610 Approximation Algorithms February 25, 1998

Lecture 6

Lecturer: David P. Williamson *Scribe: Xiangdong Yu*

6.1 Randomized Rounding continued

Recall the maximum satisfiability problem (MAX SAT) that we looked at last time.

MAX SAT

- **Input:**
  - n boolean variables $x_1, x_2, \ldots, x_n$
  - m clauses $C_1, C_2, \ldots, C_m$ (e.g. $x_3 \vee \bar{x}_5 \vee \bar{x}_7 \vee x_{11}$)
  - weight $w_i \geq 0$ for each clause C_i
- **Goal:** Find an assignment of TRUE/FALSE for the x_i that maximizes total weight of satisfied clauses. (e.g. $x_3 \vee \bar{x}_5 \vee \bar{x}_7 \vee x_{11}$ is satisfied if x_3 is set TRUE, x_5 is set FALSE, x_7 is set FALSE, or x_{11} is set TRUE).

Recall that last time we introduced the concept of randomized rounding, and tried to apply it to MAX SAT. We did this by setting up the following linear programming relaxation of the MAX SAT problem:

$$\text{Max} \quad \sum_j w_j z_j$$

subject to:

$$\sum_{i \in I_j^+} y_i + \sum_{i \in I_j^-} (1 - y_i) \geq z_j \qquad \forall C_j : \bigvee_{i \in I_j^+} x_i \vee \bigvee_{i \in I_j^-} \bar{x}_i$$

$$0 \leq y_i \leq 1$$

$$0 \leq z_j \leq 1$$

We considered what happened if we set x_i TRUE with probability y_i^*, where y^* is an optimal solution. We showed that this gave a $(1 - \frac{1}{e})$-approximation algorithm for MAX SAT. We further showed that by taking the better of the solutions given by this algorithm and Johnson's algorithm led to a $\frac{3}{4}$-approximation algorithm for MAX SAT.

Today we show that we can obtain a $\frac{3}{4}$-approximation algorithm directly by a variation of randomized rounding in which we set x_i TRUE with probability $g(y_i^*)$, where $g : [0, 1] \rightarrow [0, 1]$.

Consider the following algorithm:

Nonlinear-Round

Solve LP, get (y^*, z^*).
Pick any function $g(y)$ such that $1 - 4^{-y} \le g(y) \le 4^{y-1}$ for $y \in [0,1]$.
For $i \leftarrow 1$ to n
    If random$(g(y_i^*)) = 1$
        $x_i \leftarrow$ TRUE
    else
        $x_i \leftarrow$ FALSE.

Theorem 6.1 (Goemans, W '94) Nonlinear-Round is a 3/4-approximation algorithm for MAX SAT.

Proof: Recall **Fact 2** in previous lecture: If $f(x)$ is concave in $[l,u]$, and $f(l) \ge al + b$, $f(u) \ge au + b$, then $f(x) \ge ax + b$ on $[l,u]$.

First consider a clause of form $x_1 \vee \ldots \vee x_k$. Then

$$\begin{aligned}
\Pr[C_j \text{ satisfied}] &= 1 - \prod_{i=1}^{k}(1 - g(y_i^*)) \\
&\ge 1 - \prod_{i=1}^{k} 4^{-y_i^*} \\
&= 1 - 4^{-\sum_{i=1}^{k} y_i^*} \\
&\ge 1 - 4^{-z_j^*} \qquad (6.1)\\
&\ge \frac{3}{4} z_j^*, \qquad (6.2)
\end{aligned}$$

where (6.1) follows from the LP constraint $\sum_{i=1}^{k} y_i^* \ge z_j^*$, and (6.2) follows from Fact 2.

To show that this result holds in greater generality, suppose we negate the last variable, and have a clause of form $x_1 \vee \ldots \vee \bar{x}_k$. Then

$$\begin{aligned}
\Pr[C_j \text{ satisfied}] &= 1 - \prod_{i=1}^{k-1}(1 - g(y_i^*)) \times g(y_k^*) \\
&\ge 1 - \prod_{i=1}^{k-1} 4^{-y_i^*} \times 4^{-(1-y_k^*)} \\
&\ge 1 - 4^{-z_j^*} \qquad (6.3)\\
&\ge \frac{3}{4} z_j^*, \qquad (6.4)
\end{aligned}$$

where again (6.3) follows from the LP constraint $\sum_{i=1}^{k-1} y_i^* + (1 - y_k^*) \ge z_j^*$ and (6.4) follows from Fact 2. Clauses of other forms are similar.

Hence,

$$E[W] = \sum_j w_j \Pr[C_j \text{ satisfied}] \geq \frac{3}{4} \sum_j w_j z_j^* \geq \frac{3}{4} OPT$$

$\square$

6.2 MAX CUT in Dense Graphs

To show that the use of randomization can get quite sophisticated, we turn to again to the maximum cut problem.

MAX CUT

- **Input:** Undirected Graph $G = (V, E)$, and weights $w_{ij} \geq 0, \forall (i,j) \in E$.
- **Goal:** Find subset $S \subseteq V$ that maximizes $w(S) = \sum_{(i,j) \in E, i \in S, j \notin S \text{ or } i \notin S, j \in S} w_{ij}$

This lecture considers the case that the graph is *unweighted* (i.e. $w_{ij} = 1\ \forall (i,j) \in E$) and the graph is *dense*, i.e. $|E| \geq \alpha n^2$ for some $\alpha > 0$, where $n = |V|$. We give a result of Arora, Karger, and Karpinski that gives a PTAS for MAX CUT in this case.

An implication of this case is that $OPT \geq \frac{\alpha}{2} n^2$. To see this, recall our Dumb-Random algorithm for the maximum cut problem that produced a cut with expected value at least $\frac{1}{2} \sum_{(i,j) \in E} w_{ij}$. Since the expected value of a random cut is this large, the value of the maximum cut must also be this large.

Let us consider a particular model of the maximum cut problem. The problem can be restated as to find an assignment x which maps each vertex in V to 0 or 1, with $x_i = 1$ iff $i \in S$, and the objective function becomes

$$\max_{x_i \in \{0,1\}} \sum_{i \in V} x_i \sum_{(i,j) \in E} (1 - x_j).$$

To see this, note that $\sum_{(i,j) \in E} (1 - x_j)$ counts the number of edges adjacent to x_i that have endpoints of value 0. Since we multiply each such term by x_i, we only count these edges when $x_i = 1$. So for each vertex x_i we count all the edges that have an endpoint on the other side of the cut.

Since we will be using the term $\sum_{(i,j) \in E} (1 - x_j)$ quite frequently, we define some notation for it. Let $ZN(x, i)$ as the "Number of Zero Neighbors of i under x", i.e., $\sum_{(i,j) \in E} (1 - x_j)$.

Let x^* denote an optimal solution. Suppose there is a Genie that gives us values Z_i such that $Z_i - \epsilon n \leq ZN(x^*, i) \leq Z_i + \epsilon n$. Can we make use of this information to obtain a near-optimal solution?

The answer is "yes", and we do this by using randomized rounding. Consider the following linear program:

$$\begin{aligned}
&\text{Max} \quad \sum_{i \in V} Z_i y_i \\
&\text{subject to:} \\
&\qquad Z_i - \epsilon n \le \sum_{(i,j)\in E} (1 - y_j) \le Z_i + \epsilon n \qquad\qquad \forall i \\
&\qquad 0 \le y_i \le 1
\end{aligned}$$

Notice by the definition of Z_i, the optimal solution x^* is feasible for this LP. And the objective function value for x^* is close to OPT, as we see below:

$$\begin{aligned}
\sum_{i\in V} Z_i x_i^* &\ge \sum_{i \in V}(ZN(x^*, i) - \epsilon n)x_i^* \\
&= OPT - \epsilon n \sum_{i \in V} x_i^* \\
&\ge OPT - \epsilon n^2 \\
&\ge \left(1 - \frac{2\epsilon}{\alpha}\right) OPT
\end{aligned}$$

So the LP optimal $Z_{LP} \ge (1 - \frac{2\epsilon}{\alpha})OPT$.

Now consider the following randomized rounding algorithm.

AKK (Arora, Karger and Karpinski '95)

> Get Z_i from genie. Solve LP, get y^*.
> For all $i \in V$,
>   If random$(y_i^*) = 1$
> $x_i' \leftarrow 1$
>   else
> $x_i' \leftarrow 0$.

Observe that the value of the cut obtained is $\sum_{i\in V} x_i' ZN(x', i)$.

We need the following well-known result in our proof. This theorem is extremely important, and is used repeatedly in the analysis of randomized algorithms.

Theorem 6.2 (Chernoff) Let $X_1, \ldots, X_n$ be n independent 0-1 random variables (not necessarily from the same distribution). Then for $X = \sum_{i=1}^n X_i$ and $\mu = E[X]$, and $0 \le \delta < 1$,

$$\Pr[X \le (1+\delta)\mu] > 1 - e^{-\mu\delta^2/3},$$

and

$$\Pr[X \ge (1-\delta)\mu] > 1 - e^{-\mu\delta^2/2}.$$

Let's first calculate the expected value of ZN for the solution x'.

$$\begin{aligned}
E[ZN(x',i)] &= E\left[\sum_{(i,j)\in E}(1-x'_j)\right] \\
&= \sum_{(i,j)\in E}(1-E[x'_j]) \\
&= \sum_{(i,j)\in E}(1-y^*_j) \\
&= ZN(y^*,i)
\end{aligned}$$

We now show that with high probability this expected value is close to the value from the linear programming solution by applying Chernoff bounds. Set $\delta_i = \sqrt{\frac{2c\log n}{ZN(y^*,i)}}$. Then

$$\begin{aligned}
\Pr[ZN(x',i) < (1-\delta_i)ZN(y^*,i)] &\le e^{-\mu\delta^2/2} \\
&= e^{-ZN(y^*,i)\frac{2c\log n}{ZN(y^*,i)}} \\
&= e^{-c\log n} = 1/n^c
\end{aligned}$$

Then with high probability (w.h.p.) $1 - 1/n^{c-1}$,

$$\begin{aligned}
\sum_i x'_i ZN(x',i) &\ge \sum_i x'_i(1-\delta_i)ZN(y^*,i) \\
&\ge \sum_i x'_i\left(ZN(x',i) - \sqrt{c\log n ZN(y^*,i)}\right) \\
&\ge \sum_i x'_i\left(Z_i - \epsilon n - \sqrt{2c\log n ZN(y^*,i)}\right) \\
&\ge \sum_i x'_i Z_i - \left(\epsilon n + \sqrt{2cn\log n}\right)\sum_i x'_i
\end{aligned}$$

Now we stop to bound $\sum_i x'_i Z_i$. Since

$$E\left[\sum_i x'_i Z_i\right] = \sum_i Z_i E[x'_i] = \sum_i Z_i y^*_i$$

Using a Chernoff bound[1] with $\delta = \sqrt{\frac{2c\log n}{\sum_i y^*_i Z_i}}$, we have,

$$\Pr[\sum_i x'_i Z_i < (1-\delta)\sum_i Z_i y^*_i] \le \frac{1}{n^c}.$$

[1] Note that the variables are not 0-1, so the given theorem does not apply. However, another Chernoff-style theorem does apply. Thanks to the scribe for pointing this out. – DPW

So w.h.p.,

$$\begin{aligned}
\sum_i x'_i Z_i &\geq (1-\delta)\sum_i Z_i y^*_i \\
&= \left(1-\sqrt{\frac{2c\log n}{\sum_i y^*_i Z_i}}\right)\sum_i Z_i y^*_i \\
&= \sum_i Z_i y^*_i - \sqrt{2c\log n \sum_i y^*_i Z_i} \\
&\geq \sum_i Z_i y^*_i - n\sqrt{2c\log n}
\end{aligned}$$

Use this result to continue,

$$\begin{aligned}
\sum_i x'_i ZN(x',i) &\geq \sum_i Z_i y^*_i - n\sqrt{2c\log n} - (\epsilon n + \sqrt{2cn\log n})\sum_i x'_i \\
&\geq \left(1-\frac{2\epsilon}{\alpha}\right)OPT - n\sqrt{2c\log n} - \epsilon n^2 - n\sqrt{2cn\log n} \\
&\geq \left(1-\frac{2\epsilon}{\alpha}\right)OPT - \frac{2\epsilon}{\alpha}OPT - o(1)OPT \\
&\geq \left(1-\frac{5\epsilon}{\alpha}\right)OPT,
\end{aligned}$$

where the last line follows for n large enough to swamp out the $o(1)$ term by $\frac{\epsilon}{\alpha}OPT$. Then if we set $\epsilon' = \frac{5\epsilon}{\alpha}$, Algorithm AKK produces solution of value $\geq (1-\epsilon')OPT$ with high probability for sufficiently large n.

6.2.1 Degenie-izing the algorithm

We need to show how the "genie" works, which is based on the theorem below.

Theorem 6.3 Given $a_1, \ldots, a_n \in \{0,1\}$, $Z = \sum_{i=1}^n a_i$. If we pick a random set $S \subseteq \{1, \ldots, n\}$, with $|S| = c\log n/\epsilon^2$, then, w.h.p.

$$Z - \epsilon n \leq \frac{n}{|S|}\sum_{i\in S} a_i \leq Z + \epsilon n.$$

Pick random subset S of $c\log n/\epsilon^2$ vertices. Set $Z_i = \frac{n}{|S|}\sum_{(i,j)\in E, j\in S}(1-x^*)$. By the theorem, w.h.p.,

$$ZN(x^*,i) - \epsilon n \leq Z_i \leq ZN(x^*,i) + \epsilon n$$

But we still don't know x^*! In order to get around this problem, we run the algorithm for all $2^{|S|} = n^{O(1/\epsilon^2)}$ possible settings of $x^*_j \in \{0,1\}$. We don't know which one gives the optimal solution, but it doesn't matter; we simply return the largest cut found, and that will be guaranteed to be within a $(1-\epsilon')$ factor of OPT, since at least one of the cuts will be this large.

IEOR E6610 Approximation Algorithms	March 4, 1998
Lecture 7	
Lecturer: David P. Williamson	*Scribe: R.N. Uma*

7.1 Semidefinite Programming

Definition 7.1 A matrix $X \in \Re^{n \times n}$ is positive semidefinite (psd) iff $\forall a \in \Re^n$, $a^T X a \geq 0$.

Sometimes we will write $X \succeq 0$ to denote that X is psd.

If $X \in \Re^{n \times n}$ is a symmetric matrix, then the following are equivalent:

1. X is psd;
2. X has non-negative eigenvalues;
3. $X = V^T V$ for some $V \in \Re^{m \times n}$, where $m \leq n$.

A semidefinite program (SDP) can be formulated as

$$
\begin{array}{ll}
\text{Max or Min} & \sum c_{ij} x_{ij} \\
\text{subject to:} & \\
& \sum_{i,j} a_{ijk} x_{ij} = b_k \qquad \forall k \\
& X = (x_{ij}) \succeq 0 \qquad \text{and } X \text{ is symmetric}
\end{array}
$$

SDP's have the useful property that they can be solved in polynomial time using

- the ellipsoid method
- modifications of interior-point methods that are used to solve LP's

to within an additive error of ϵ. The running time depends on ϵ, but in a nice way. This additive error of ϵ is necessary because sometimes the solutions to SDP's may be irrational numbers.

SDP is equivalent to *vector programming* (VP) which can be formally stated as

$$\text{Max or Min} \quad \sum c_{ij}(\vec{v_i} \cdot \vec{v_j})$$

subject to:

$$\begin{aligned} \sum_{i,j} a_{ijk}(\vec{v_i} \cdot \vec{v_j}) &= b_k && \forall k \\ \vec{v_i} &\in \Re^n && \forall i \end{aligned}$$

This follows since X is psd and X is symmetric iff $X = V^T V$ i.e., iff $x_{ij} = \vec{v_i} \cdot \vec{v_j}$ where

$$V = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \vec{v_1} & \vec{v_2} & \cdots & \vec{v_3} \\ \vdots & \vdots & & \vdots \end{pmatrix}$$

(That is, the $\vec{v_i}$ are the column vectors of V). So if we have a feasible solution to SDP, then we have a solution to VP with the same value and vice versa.

7.1.1 MAX CUT using Semidefinite Programming

We now consider applying semidefinite programming to the MAX CUT problem. Let us consider the following formulation of the MAX CUT problem which we denote by (A).

$$(A) \qquad \begin{aligned} \text{Max} \quad & \frac{1}{2}\sum_{i<j} w_{ij}(1 - y_i \cdot y_j) \\ & y_i \in \{-1, +1\} \qquad \forall i \end{aligned}$$

We claim that if we can solve (A), then we can solve the MAX CUT problem.

Claim 7.1 The formulation (A) models MAX CUT.

Proof: Consider the cut given by $S = \{i \in V | y_i = -1\}$. We have

$$\frac{1}{2}\sum_{i<j} w_{ij}(1 - y_i \cdot y_j) = \frac{1}{2}\sum_{i<j:y_i=y_j} w_{ij}(1 - y_i \cdot y_j) + \frac{1}{2}\sum_{i<j:y_i\neq y_j} w_{ij}(1 - y_i \cdot y_j)$$

This is because $y_i \in \{-1, +1\}$ for all i. So for a given i and j, either $y_i = y_j$ or $y_i \neq y_j$. If $y_i = y_j$, then $1 - y_i \cdot y_j = 0$ and if $y_i \neq y_j$, then $1 - y_i \cdot y_j = 2$. So in the above expression, the first term becomes zero. So we have

$$\begin{aligned} \frac{1}{2}\sum_{i<j} w_{ij}(1 - y_i \cdot y_j) &= \frac{1}{2}\sum_{i<j:y_i\neq y_j} w_{ij}(1 - y_i \cdot y_j) \\ &= \frac{1}{2}\sum_{i<j:y_i\neq y_j} w_{ij} \cdot 2 \\ &= \sum_{i<j:i\in S, j\notin S (\text{or}) i\notin S, j\in S} w_{ij} \end{aligned}$$

□

Let us now consider a vector programming relaxation (denoted (B)) of (A).

$$
(B) \quad
\begin{aligned}
Z_{SDP} \quad = \quad \text{Max} \quad & \frac{1}{2}\sum_{i<j} w_{ij}(1 - \vec{v_i}\cdot\vec{v_j}) & \\
& \vec{v_i}\cdot\vec{v_i} = 1 & \forall i \\
& \vec{v_i} \in \Re^n & \forall i.
\end{aligned}
$$

To see that (B) is indeed a relaxation of (A), we can view the y_i's in (A) as 1-dimensional vectors and so anything that is feasible for (A) is feasible for (B). Also note that this implies $Z_{SDP} \geq OPT$.

We can solve (B) in polynomial time, but not (A). So how do we convert a solution of (B) to a solution of (A) ? To do this, we would like to apply randomized rounding in some way.

Consider the following algorithm:

VectorRound

> Solve vector programming problem (B) and get vectors $\vec{v^*}$
> Choose a random vector $\vec{r}$ uniformly from the unit n-sphere
> $S \leftarrow \emptyset$
> for $i \leftarrow 1$ to n
>   if $\vec{v_i^*}\cdot\vec{r} \geq 0$
> $S \leftarrow S \cup \{i\}$

The vector $\vec{r}$ is a normal to some hyperplane. So everything that has a non-negative dot product with $\vec{r}$ will be on one side of the hyperplane and everything that has a negative dot product with $\vec{r}$ will be on the other side (see Figure 7.1). To get vector $\vec{r}$, choose $\vec{r} = (r_1, r_2, \cdots, r_n)$, such that $r_i \in \mathcal{N}(0,1)$ where $\mathcal{N}$ is a normal distribution. (The normal distribution $\mathcal{N}(0,1)$ can be simulated using uniform distribution on $[0,1]$.)

Theorem 7.2 VectorRound is a 0.878-approximation algorithm.

To prove this theorem we require a couple of facts and a couple of lemmas which we give below.

Fact 7.1 $\frac{\vec{r}}{||\vec{r}||}$ (i.e., normalization of $\vec{r}$) is uniformly distributed over a unit sphere.

Fact 7.2 The projection of $\vec{r}$ onto two lines l_1 and l_2 are independent and normally distributed iff l_1 and l_2 are orthogonal.

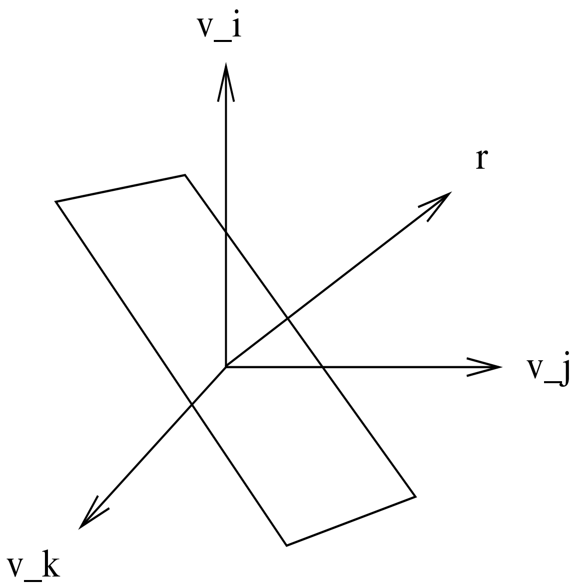

Figure 7.1: A random hyperplane example

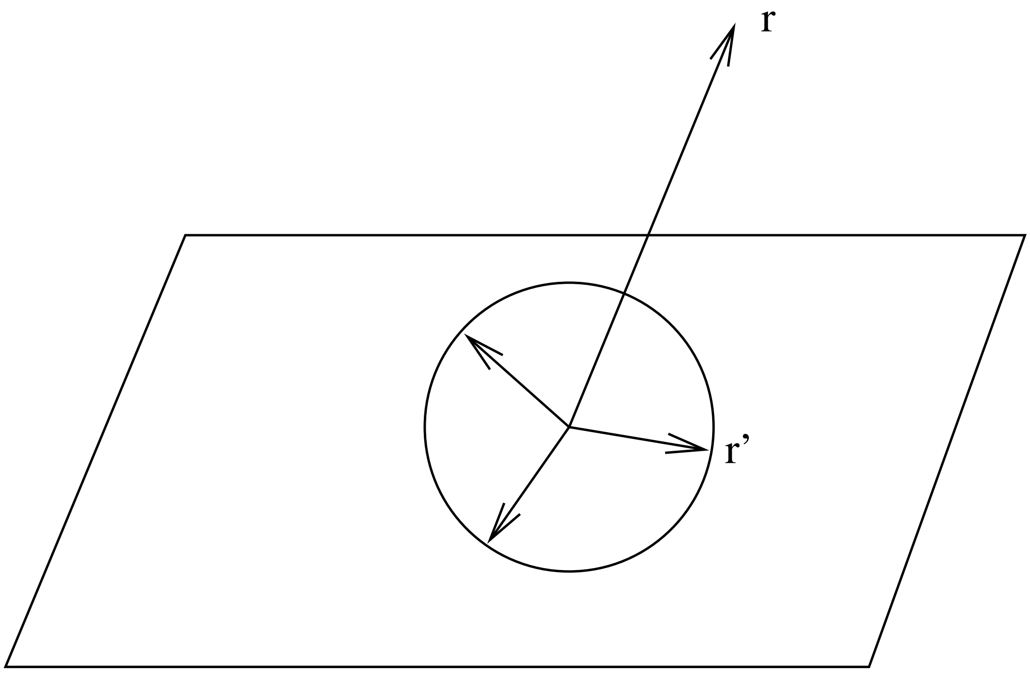

Figure 7.2: Projection of r to r'

Corollary 7.3 Let $\vec{r'}$ be the projection of $\vec{r}$ onto a plane. $\frac{\vec{r'}}{||\vec{r'}||}$ is uniformly distributed on a unit circle on the plane (see Figure 7.2).

Lemma 7.4 $\Pr[i \in S, j \notin S$ or $i \notin S, j \in S] = \frac{1}{\pi}\arccos(\vec{v_i^*} \cdot \vec{v_j^*})$.

Proof:

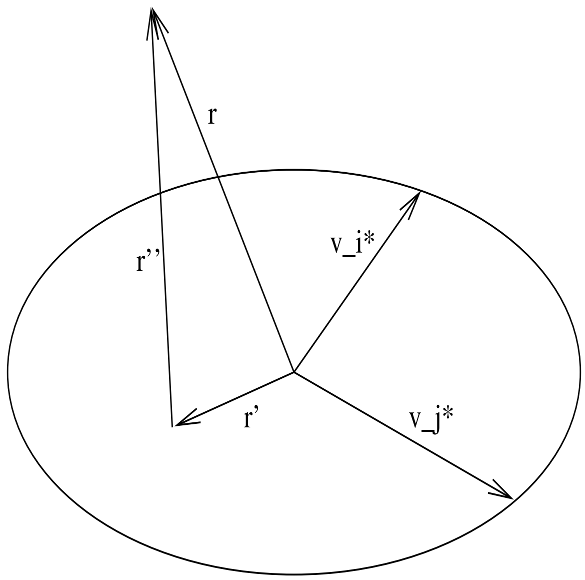

Figure 7.3: Projection of r into plane defined by v_i^* and v_j^*

Let $\vec{r'}$ be the projection of $\vec{r}$ onto the plane defined by $\vec{v_i^*}$ and $\vec{v_j^*}$.

If $\vec{r} = \vec{r'} + \vec{r''}$ (Figure 7.3), then

$$\begin{aligned} \vec{v_i^*} \cdot \vec{r} &= \vec{v_i^*} \cdot (\vec{r'} + \vec{r''}) \\ &= \vec{v_i^*} \cdot \vec{r'} \end{aligned}$$

The second equality follows because $\vec{r''}$ is orthogonal to $\vec{v_i^*}$. Similarly, $\vec{v_j^*} \cdot \vec{r} = \vec{v_j^*} \cdot \vec{r'}$.

There are a total of 2π possible orientations of $\vec{r'}$. If $\vec{r'}$ lies on the semi-circular plane AFB, see Figure 7.4, then $\vec{v_i^*} \cdot \vec{r'} \geq 0$ and so $i \in S$. If $\vec{r'}$ lies on the semi-circular plane AEB, then $i \notin S$. Likewise, if $\vec{r'}$ lies on the semi-circular plane CFD, then $j \in S$ and if $\vec{r'}$ lies on the semi-circular plane CED, then $j \notin S$. Let θ be the angle between the vectors $\vec{v_i^*}$ and $\vec{v_j^*}$. So by construction, $\widehat{AOC} = \widehat{BOD} = \theta$. Note that in the sector AOC, $i \notin S$ and $j \in S$ and in the sector BOD, $i \in S$ and $j \notin S$. Therefore, 2θ of the orientations out of a total of 2π orientaions for $\vec{r'}$ cause $i \in S, j \notin S$ or $i \notin S, j \in S$. Therefore the required probability is $\frac{2\theta}{2\pi}$. We have $\vec{v_i^*} \cdot \vec{v_j^*} = \|\vec{v_i^*}\| \cdot \|\vec{v_j^*}\| \cos\theta$. Therefore, $\theta = \arccos(\vec{v_i^*} \cdot \vec{v_j^*})$, since the $\vec{v_i^*}$'s are unit vectors. Hence the result. $\square$

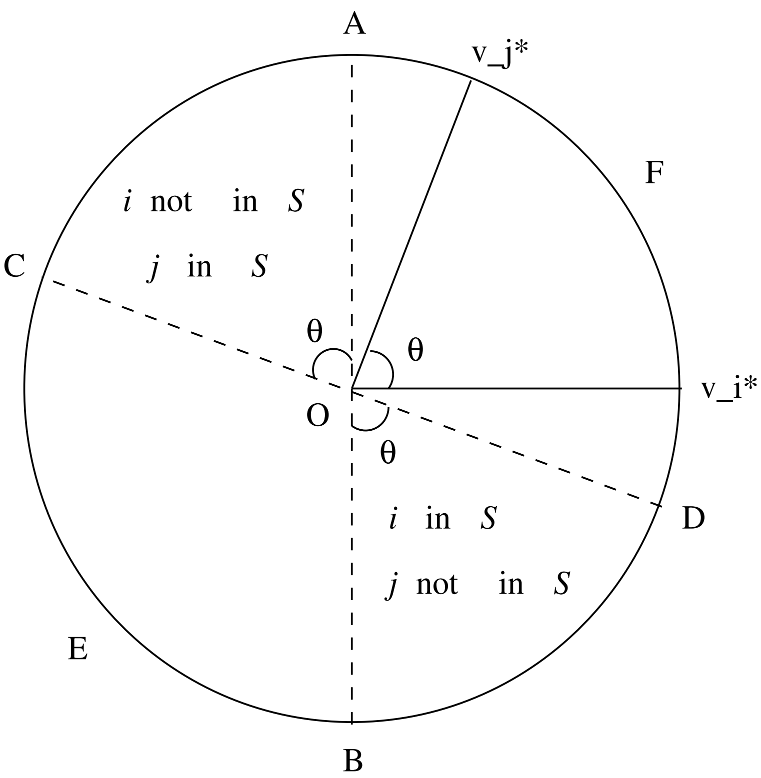

Figure 7.4: Determining the probability

Lemma 7.5
$$\min_{-1\le x\le 1} \frac{\frac{1}{\pi}\arccos(x)}{\frac{1}{2}(1-x)} \ge 0.878.$$

Proof: Using Mathematica!!! $\square$

So now we can prove the theorem.

Theorem 7.6 (Goemans, W '95) VectorRound is a 0.878-approximation algorithm.

Proof: Consider the random variables
$$X_{ij} = \begin{cases} 1 & \text{if } i \in S, j \notin S \text{ or } i \notin S, j \in S \\ 0 & \text{otherwise} \end{cases}$$

and
$$W = \sum_{i<j} w_{ij} X_{ij}.$$

Then
$$\begin{aligned} E[W] &= \sum_{i<j} w_{ij} \cdot \Pr[i \in S, j \notin S \text{ or } i \notin S, j \in S] \\ &= \sum_{i<j} w_{ij} \frac{1}{\pi} \arccos(\vec{v}_i^* \cdot \vec{v}_j^*) && (7.1) \\ &\ge 0.878 \cdot \frac{1}{2} \sum_{i<j} w_{ij}(1 - \vec{v}_i^* \cdot \vec{v}_j^*) && (7.2) \\ &= 0.878 \cdot Z_{SDP} \\ &\ge 0.878 \cdot OPT, \end{aligned}$$

where (7.1) follows by Lemma 7.4 and (7.2) follows by Lemma 7.5. $\square$

Can we do better than this? It turns out that we will have to do something quite different, as the following theorems attest.

Corollary 7.7 For any graph with non-negative weights,
$$\frac{OPT}{Z_{SDP}} \ge 0.878.$$

Theorem 7.8 (Delorme, Poljak '93) For the 5-cycle,
$$\frac{OPT}{Z_{SDP}} = \frac{32}{25 + 5\sqrt{5}} \approx 0.884.$$

Theorem 7.9 (Karloff '95) There exists graphs G such that,

$$\frac{E[W]}{OPT} = \frac{E[W]}{Z_{SDP}} \to \min_{-1 \le x \le 1} \frac{\frac{1}{\pi}\arccos(x)}{\frac{1}{2}(1-x)}.$$

This is true even if any valid inequality is added to SDP.

The theorem by Delorme and Poljak implies that we can't do much better than a performance guarantee of .878 using this SDP. The theorem of Karloff implies that we can't do any better at all with this SDP or anything obtained by adding valid inequalities as long as we obtain the cut by choosing a random hyperplane.

It turns out that there is a limit on how well we can do anyway.

Theorem 7.10 (Håstad '97) If $\exists$ an α-approximation algorithm for MAX CUT, $\alpha > \frac{16}{17} \approx 0.941$, then $P = NP$.

Research Question: Can you get a 0.878-approximation algorithm without solving an SDP?

7.1.2 Quadratic Programming

We now show that we can get an approximation algorithm for some kinds of quadratic programming by using the same techniques. Consider quadratic programs of the form

$$(C) \qquad \begin{array}{lll} \text{Max} & \sum_{i,j} a_{ij}(x_i \cdot x_j) & \\ & x_i \in \{-1, +1\} & \forall i, \end{array}$$

where we assume that the values in the objective function form a positive semidefinite matrix: i.e., $A \succeq 0$.

As before we can relax this to the following vector program:

$$(D) \qquad \begin{array}{lll} \text{Max} & \sum_{i,j} a_{ij}(\vec{v_i} \cdot \vec{v_j}) & \\ & \vec{v_i} \cdot \vec{v_j} = 1 & \forall i \\ & \vec{v_i} \in \Re^n. & \end{array}$$

VectorRound2

Solve the vector program and get vectors $\vec{v^*}$
Choose a random vector $\vec{r}$ uniformly from the unit n-sphere
for $i \leftarrow 1$ to n
  if $\vec{v_i^*} \cdot \vec{r} \ge 0$
    $\bar{x}_i \leftarrow 1$
  else
    $\bar{x}_i \leftarrow -1$

We will need to do something a little bit different here because some of the entries of A may be negative.

We will now give two lemmas similar to Lemmas 7.4 and 7.5.

Lemma 7.11 $E[\bar{x}_i \cdot \bar{x}_j] = \frac{2}{\pi}\arcsin(\vec{v_i^*} \cdot \vec{v_j^*})$

Proof:

$$\begin{aligned}
E[\bar{x}_i \cdot \bar{x}_j] &= \Pr[\bar{x}_i \cdot \bar{x}_j = 1] - \Pr[\bar{x}_i \cdot \bar{x}_j = -1] \\
&= \left(1 - \frac{1}{\pi}\arccos(\vec{v_i^*} \cdot \vec{v_j^*})\right) - \left(\frac{1}{\pi}\arccos(\vec{v_i^*} \cdot \vec{v_j^*})\right) \\
&= 1 - \frac{2}{\pi}\arccos(\vec{v_i^*} \cdot \vec{v_j^*}) \\
&= 1 - \frac{2}{\pi}\left[\frac{\pi}{2} - \arcsin(\vec{v_i^*} \cdot \vec{v_j^*})\right] \\
&= \frac{2}{\pi}\arcsin(\vec{v_i^*} \cdot \vec{v_j^*})
\end{aligned}$$

The second term in the second equality follows from Lemma 7.4, and the fourth inequality follows since $\arcsin(x) + \arccos(x) = \frac{\pi}{2}$. □

We would like our proof to proceed as follows. We would like to prove an equivalent of Lemma 7.5.

Lemma 7.12

$$\min_{-1 \le x \le 1} \frac{\frac{2}{\pi}\arcsin(x)}{x} \ge \alpha.$$

We would then like the proof to go the same as before:

$$\begin{aligned}
E[\sum_{i,j} a_{ij}(\bar{x}_i \cdot \bar{x}_j)] &= \sum_{i,j} a_{ij} E[\bar{x}_i \cdot \bar{x}_j] \\
&= \sum_{i,j} a_{ij} \frac{2}{\pi}\arcsin(\vec{v_i^*} \cdot \vec{v_j^*}) \\
&\ge \alpha \sum_{i,j} a_{ij}(\vec{v_i^*} \cdot \vec{v_j^*}) \qquad (7.3) \\
&\ge \alpha \cdot OPT. \qquad (7.4)
\end{aligned}$$

But we cannot do this because the inequality (7.3) is not correct. This is because some of the a_{ij}'s may be negative. That is the inequality $\frac{2}{\pi}\arcsin(x) \ge \alpha x$ will become $\frac{2}{\pi} a_{ij} \arcsin(x) \le \alpha x a_{ij}$ if $a_{ij} < 0$.

So we will now switch back to matrix notation. So the vector program that we saw earlier (D) is equivalent to

$$
\begin{aligned}
&\text{Max} \quad A \bullet X \\
&\text{subject to:} \\
&\qquad x_{ii} = 1 \qquad\qquad \forall i \\
&\qquad X = (x_{ij}) \succeq 0 \qquad \text{and } X \text{ is symmetric.}
\end{aligned}
$$

Note that $A \bullet X = \sum_{i,j} a_{ij} x_{ij}$ is sometimes referred to as the *outer product* as opposed to *inner product*. Let $X^* = (x^*_{ij})$ be the optimal solution. Let $f[X] \equiv (f(x_{ij})_{ij})$ and let $\bar{X} = ((\bar{x}_i \cdot \bar{x}_j)_{ij})$.

We can restate Lemma 7.11 by the following corollary.

Corollary 7.13 $E[A \bullet X] = A \bullet (\frac{2}{\pi} \arcsin[X^*])$

Proof:

$$E[A \bullet X] = E[\sum_{i,j} a_{ij}(\bar{x}_i \bar{x}_j)] = \sum_{i,j} a_{ij} \left(\frac{2}{\pi} \arcsin(x^*_{ij}) \right) = A \bullet \left(\frac{2}{\pi} \arcsin[X^*] \right).$$

□

Fact 7.3 If $A \succeq 0$, $B \succeq 0$, then $A \bullet B \geq 0$.

Fact 7.4 If $X \succeq 0$, $|x_{ij}| \leq 1$, then $\arcsin[X] - X \succeq 0$.

Theorem 7.14 (Nesterov '97) VectorRound2 is a $\frac{2}{\pi}$-approximation algorithm.

Proof: We want to show

$$E[A \bullet \bar{X}] - \frac{2}{\pi} A \bullet X^* \geq 0$$

because this would imply

$$E[A \bullet \bar{X}] \geq \frac{2}{\pi} A \bullet X^* \geq \frac{2}{\pi} \cdot OPT$$

We know

$$E[A \bullet \bar{X}] = A \bullet \left(\frac{2}{\pi} \arcsin[X^*] \right) \qquad \text{from Corollary 7.13}$$

So we want to show

$$A \bullet \left(\frac{2}{\pi} \arcsin[X^*] \right) - \frac{2}{\pi} (A \bullet X^*) \geq 0$$

Hence we want to show

$$A \bullet \left(\frac{2}{\pi} (\arcsin[X^*] - X^*) \right) \geq 0$$

That is, we want to show

$$\frac{2}{\pi} (A \bullet (\arcsin[X^*] - X^*)) \geq 0$$

But $\arcsin[X^*] - X^*$ is psd by Fact 7.4 and thus $A \bullet (\arcsin[X^*] - X^*) \geq 0$ by Fact 7.3. Hence the result. □

IEOR E6610 Approximation Algorithms	March 11, 1998
Lecture 8	
Lecturer: David P. Williamson	*Scribe: Jörn Meißner*

8.1 Semidefinite Programming: Graph Coloring

In the last lecture we have seen that a semidefinite program (SDP) can be formulated as

$$\text{Max or Min} \quad \sum c_{ij} x_{ij}$$

subject to:

$$\sum_{i,j} a_{ijk} x_{ij} = b_k \qquad \forall \quad k$$

$$X = (x_{ij}) \succeq 0 \qquad \text{and } X \in \Re^{n \times n} \text{ is symmetric,}$$

and is equivalent to a *vector programming* (VP) which can be formally stated as

$$\text{Max or Min} \quad \sum c_{ij} (\vec{v_i} \cdot \vec{v_j})$$

subject to

$$\sum_{i,j} a_{ijk} (\vec{v_i} \cdot \vec{v_j}) = b_k \qquad \forall k$$

$$\vec{v_i} \in \Re^n \qquad \forall i.$$

This follows since X is psd and X is symmetric iff $X = V^T V$ i.e., iff $x_{ij} = \vec{v_i} \cdot \vec{v_j}$ where

$$V = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \vec{v_1} & \vec{v_2} & \cdots & \vec{v_3} \\ \vdots & \vdots & & \vdots \end{pmatrix}.$$

Today we will show that semidefinite programming can be applied to the graph coloring problem. In particular, we will show the following result:

Theorem 8.1 (Karger, Motwani, Sudan '94) There is a polytime algorithm to color a 3-colorable graph with $\tilde{O}(n^{1/4})$ colors.

The previous best algorithm used $\tilde{O}(n^{3/8})$ colors (Blum '94).

Definition 8.1 A function $f(n) = \tilde{O}(g(n))$ (sometimes also called $O^*(g(n))$ when the following is valid:

$$\exists \quad n_0, c_1, c_2 \quad \text{s.t.} \quad \forall \quad n \geq n_0 \qquad f(n) \leq c_1 g(n) \log^{c_2} n$$

This doesn't seem very good; we know the graph is 3-colorable but we take more than $n^{1/4}$ colors! But the previously known result is worse still. In terms of the hardness of this problem, the theorem below is the best thing known so far.

Theorem 8.2 (Khanna, Linial, Safra '93) It is NP-hard to color a 3-colorable graph with 4 colors.

The following facts are known about coloring graphs.

- We can color 2-colorable (aka bipartite) graphs in polytime.
- We can color G with $\Delta + 1$ colors (Δ = max degree of G) in polytime.

Proof: Color greedily (color with color not used by neighbors yet). $\Box$

The following coloring algorithm for 3-colorable graphs was also previously known.

Color1

```
While ∃v ∈ G   s.t.  deg(v) ≥ √n
      Color v with color #1
      Color neighbors of v in polytime with 2 new colors
      Remove v and its neighbors from graph
Color remaining G with √n new colors
```

Theorem 8.3 (Widgerson '83) Color1 colors 3-colorable graphs in polytime with $O(\sqrt{n})$ colors.

Proof: We can execute the while loop at most $\frac{n}{\sqrt{n}}$ times, since we remove at least $\sqrt{n}$ vertices from the graph every time. Hence we use $1 + 2\frac{n}{\sqrt{n}}$ colors in the while loop.

The last step takes $\sqrt{n}$ colors by the fact above (since the maximum degree is $\sqrt{n} - 1$), so the total numbers of colors needed is $3\sqrt{n} + 1$. $\Box$

We now think about applying semidefinite programming to the problem of coloring 3-colorable graphs. Consider the following vector programm:

$$
\begin{aligned}
\text{Min} \quad & \lambda \\
\text{subject to:} \quad & \\
& v_i \cdot v_j \leq \lambda \qquad \forall \quad (i,j) \in E \\
& v_i \cdot v_i = 1 \qquad \forall \quad i \\
& v_i \in R^n
\end{aligned}
$$

Claim 8.4 For a 3-colorable graph $\lambda \leq -\frac{1}{2}$.

Proof: Consider an equilateral triangle, and associate the vectors for the three different colors with the three different vertices of the triangle. Note that the angle between any two vectors of the same color is 0, while the angle between any two vectors of different color is $2\pi/3$. Then for v_i, v_j such that $(i,j) \in E$

$$\begin{aligned} v_i \cdot v_j &= \|v_i\| \, \|v_j\| \cos\theta \\ &= \cos\left(\frac{2\pi}{3}\right) \\ &= -\frac{1}{2}, \end{aligned}$$

so that this solution is a feasible solution to the vector program with $\lambda = -1/2$. Thus in the optimal solution, $\lambda \leq -1/2$. $\square$

As before we will consider randomized algorithms. It turns out that it is too much to expect that we will get an algorithm that colors the whole graph correctly with high probability. Instead, we will aim for an algorithm that colors mostly correctly. In particular, we want a *semicoloring*, which means that at most $\frac{n}{4}$ edges have the same colored endpoints. In such a solution at least $\frac{n}{2}$ of the vertices are colored "correctly" (any edge between these vertices has differently colored endpoints).

Note then if we can semicolor a graph with k colors, then we can color the graph with $k \log n$ colors: we obtain a semicoloring of the graph with k colors, and take the half of the graph colored correctly. We then semicolor the remaining half of the graph with k new colors, and take the half colored correctly, and so on. This takes $\log n$ iterations, after which the graph is colored correctly with $k \log n$ colors.

Consider now the following algorithm.

KMS1

Solve vector program, get v_i
Pick $t = 2 + \log_3 \Delta$ random vectors $r_1, \ldots, r_t$
Let $R_1 = \{v_i : r_1 \cdot v_i \geq 0, r_2 \cdot v_i \geq 0, \ldots, r_t \cdot v_i \geq 0\}$
$R_2 = \{v_i : r_1 \cdot v_i < 0, r_2 \cdot v_i \geq 0, \ldots, r_t \cdot v_i \geq 0\}$
$\vdots$
$R_{2^t} = \{v_i : r_1 \cdot v_i < 0, r_2 \cdot v_i < 0, \ldots, r_t \cdot v_i < 0\}$
Color vertices of vectors in R_i with color i

Theorem 8.5 (Karger, Motwani, Sudan) The algorithm KMS1 gives a semicoloring of $O(\Delta^{\log_3 2})$ colors with probability $\frac{1}{2}$.

Proof: Since we used 2^t colors, this is $2^t = 4 \times 2^{\log_3 \Delta} = 4\Delta^{\log_3 2}$ colors.

Now

$$
\begin{aligned}
&\Pr[i \text{ and } j \text{ get the same color for edge } (i,j)]\\
&= \left(1 - \frac{1}{\pi}\arccos(v_i \cdot v_j)\right)^t\\
&\le \left(1 - \frac{1}{\pi}\arccos(\lambda)\right)^t\\
&\le \left(1 - \frac{1}{\pi}\arccos(-\frac{1}{2})\right)^t\\
&= \left(1 - \frac{1}{\pi}\frac{2\pi}{3}\right)^t\\
&= \left(\frac{1}{3}\right)^t\\
&\le \frac{1}{9\Delta}.
\end{aligned}
$$

This follows since for any particular random vector r_k, we know (from the analysis for MAX CUT) that the probability that $v_i \cdot r_k \ge 0$ and $v_j \cdot r_k \ge 0$ OR that $v_i \cdot r_k < 0$ and $v_j \cdot r_k < 0$ is $1 - \frac{1}{\pi}\arccos(v_i \cdot v_j)$. Thus the probability that v_i and v_j get the same color is the probability that this happens for each of the t vectors, which is $(1 - \frac{1}{\pi}\arccos(v_i \cdot v_j))^t$, since each of these events is independent.

Let m denote the number of edges in the graph. Note that $m \ge n\Delta/2$. Thus

$$E[\#\text{ bad edges}] \le \frac{m}{9\Delta} \le \frac{\frac{\Delta n}{2}}{9\Delta} \le \frac{n}{8},$$

and therefore

$$\Pr[\text{ more than } \tfrac{n}{4} \text{ bad edges}] \le \frac{1}{2}.$$

$\square$

If we just plug in n for Δ, this gives us an algorithm that colors with $\tilde{O}(n^{\log_3 2}) = \tilde{O}(n^{.631})$, which is worse than Widgerson's algorithm. But we can use Widgerson's technique to make things better:

Color2

While $\exists v \in G$ s.t. $\deg(v) \ge \sigma$
    Color v with color #1
    2-color neighbors of v in polytime with 2 new colors
    Remove v & neighbors
Apply KMS1 to color remaining graph with $O(\sigma^{\log_3 2})$ colors

Let's analyze this algorithm. The While loop uses $O(\frac{n}{\sigma})$ colors, since we remove σ vertices from the graph each time. The final step uses $\tilde{O}(\sigma^{\log_3 2})$ colors. If we set $\sigma = n^{0.613}$ (to balance the two terms), we get a coloring with $\tilde{O}(n^{0.387})$ colors.

But this algorithm is still worse than Blum (whose algorithm uses $\tilde{O}(n^{\frac{3}{8}})$ colors)! We consider next the following algorithm:

KMS2

> Solve vector program, get vectors v_i
> Pick $t = \tilde{O}(\Delta^{\frac{1}{3}})$ random vectors $r_1, \ldots, r_t$
> Assign vector v_i to random r_j that maximizes $v_i \cdot r_j$
> Color vectors assigned to r_j with color j

Theorem 8.6 (Karger, Motwani, Sudan)

$$\Pr[i \text{ and } j \text{ get same color for edge } (i,j)] = \tilde{O}(t^{-3})$$

We omit the proof of this theorem. To see that this theorem leads to a better algorithm, note that if we use $t = \tilde{O}(\Delta^{\frac{1}{3}})$ vectors, for an appropriate choice of the right constants, we get that

$$\Pr[i \text{ and } j \text{ get same color for edge } (i,j)] \leq \frac{1}{9\Delta},$$

just as with the previous algorithm, and the previous analysis goes through, except now our algorithm uses $\tilde{O}(\Delta^{\frac{1}{3}})$ colors. If we now apply Widgerson's technique using this algorithm, we get an algorithm that colors the graph with $\tilde{O}(n^{\frac{1}{4}})$ colors.

8.2 The Primal-Dual Method

Recall our meta-method for designing approximation algorithms:

1. Formulate problem as an integer program
2. Relax to an LP
3. Use LP to obtain a near optimal solution

We will now look at another way of carrying out the third step, a technique known as the primal-dual method for approximation algorithms. To illustrate this we look at the following problem:

Hitting Set

- **Input:**
  - ground set $E = \{e_1, e_2, \ldots, e_n\}$
  - subsets $T_1, T_2, \ldots, T_p \subseteq E$
  - costs $c_e \geq 0 \quad e \in E$
- **Goal:** Find min-cost $A \subseteq E$ s.t. $A \cap T_i \neq \emptyset \quad \forall i$.

We now carry out our meta-method for the hitting set problem. First we formulate the problem as an integer program:

- Step 1

$$\begin{aligned} & \text{Min} \quad \sum_{e \in E} c_e x_e \\ \text{subject to:} & \\ & \sum_{e \in T_i} x_e \geq 1 \qquad \forall i \\ & x_e \in \{0,1\}. \end{aligned}$$

- Step 2. We then relax it to a linear program:

$$x_e \in \{0,1\} \rightarrow x_e \geq 0.$$

- Step 3. For the third step, we are going to consider the dual of the linear programming relaxation, which is the following:

$$\begin{aligned} & \text{Max} \quad \sum_{i} y_i \\ \text{subject to:} & \\ & \sum_{i: e \in T_i} y_i \leq c_e \qquad \forall e \in E \\ & y_i \geq 0 \qquad \forall i. \end{aligned}$$

We can now state the general primal-dual method for approximation algorithms:

Primal-Dual Method for Approximation Algorithms

$y \leftarrow 0$
While there does not exist an integral solution obeying primal complementary slackness conditions ($x_e > 0 \Rightarrow \sum_{i:e \in T_i} y_i = c_e$)
    Get direction of increase for dual
Return feasible integral solution x obeying primal complementary slackness.

The primal-dual method for approximation algorithms differs from the classical primal-dual method in that the dual complementary slackness conditions are not enforced.

We now flesh out the steps of the general method. To check the condition of the while loop, we simply look at all edges e such that corresponding dual constraint is tight, put them in our solution, and check whether the solution is feasible. That is, we consider $A = \{e \in E : \sum_{i:e \in T_i} y_i = c_e\}$ and see if $A \cap T_i \neq \emptyset$ for each i.

The algorithm states that if A is not a feasible solution then there should be some way to increase the dual objective function. This follows since if A is not a feasible solution, then there exists T_k s.t. $A \cap T_k = \emptyset$. By the definition of A, we know that for each $e \in T_k$, it must be the case that $\sum_{i:e \in T_i} y_i < c_e$. Thus each inequality involving y_k is not tight, so we can increase y_k by $\min_{e \in T_k}\{c_e - \sum_{i:e \in T_k} y_i\}$. This increases the value of the dual objective function.

We can now restate the primal-dual method as follows:

Primal-Dual (Take 2)

> $y \leftarrow 0$
> $A \leftarrow \emptyset$
> While A is not feasible
>   Choose some *violated* set T_k (i.e. some set T_k s.t. $A \cap T_k = \emptyset$)
>   Increase y_k until $\exists e \in T_k : \sum_{i:e \in T_i} y_i = c_e$
> $A \leftarrow A \cup \{e\}$
> Return A

We now consider the value of the solution found by the algorithm. We know that

$$\sum_{e \in A} c_e = \sum_{e \in A} \sum_{i:e \in T_i} y_i = \sum_i |A \cap T_i| y_i,$$

where the first equality follows since $c_e = \sum_{i:e \in T_i} y_i$ for each $e \in A$, and where the second equality follows from rewriting the double sum.

Thus if we can show that whenever $y_i > 0$ then $|A \cap T_i| \leq \alpha$, it follows that

$$\sum_{e \in A} c_e \leq \alpha \sum_i y_i \leq \alpha \text{OPT},$$

since the value of the dual objective function for any feasible dual solution is a lower bound on OPT.

As an example, we apply this algorithm to the vertex cover problem. Note that vertex cover can be translated into a hitting set problem, where V is the ground set of elements, the costs c_i of the elements are the weights of the vertices, and we must hit the sets $T_i = \{u, v\}$ for each $(u, v) \in E$. Then since $|T_i| = 2$ for each set, it follows that $|A \cap T_i| \leq 2$ for all i, and by the reasoning above we have a 2-approximation algorithm for vertex cover.

Let us consider another example:

Feedback Vertex Set in Undirected Graphs

- **Input:**
  - Undirected graph $G = (V, E)$
  - Weights $w_i \geq 0 \quad \forall i \in V$
- **Goal:** Find $S \subseteq V$ minimizing $\sum_{i \in S} w_i$ such that for every cycle C in G, $C \cap S \neq \emptyset$. (Equivalently, find a min-weight set of vertices S such that removing S from the graph causes the remaining graph to be acyclic).

We claim that the feedback vertex set problem is just a hitting set problem with:

- Ground set V
- Cost w_i
- Sets to hit: $T_i = C_i$ for each cycle C_i in graph

We now have a hitting set problem with potentially an exponential number of sets to hit. How do we deal with this problem? The answer is that we do not need to enumerate or find all cycles: the algorithm only needs to find a violated set when one exists.

To apply the primal-dual method to this problem, we first need the following observation: we can reduce the input graph G to an equivalent graph G' with no degree 1 vertices and such that every degree 2 vertex is adjacent to a vertex of higher degree. To see this suppose we have two vertices of degree two adjacent to each other, i and j, and WLOG $w_i \leq w_j$. Note that every cycle which goes through i must also go through j. Thus there is no reason to include j in any solution: we should always choose i. We can then shortcut j out of the graph.

To get our algorithm, we need the following lemma:

Lemma 8.7 (Erdös, Posa) In every non-empty graph in which there are no degree 1 vertices and such that each vertex of degree 2 is adjacent to a vertex of higher degree, there is a cycle of length no longer than $4 \log_2 n$.

Thus in our algorithm, we always choose as our violated set any unhit cycle of length no longer than $4 \log_2 n$ (such a cycle can be found via a breadth-first search of the graph).

Theorem 8.8 (Bar-Yehuda, Geiger, Naor, Roth '94) If we choose a cycle of length no more than $4 \log_2 n$ as our violated set, we get a $4 \log_2 n$-approximation algorithm for the feedback vertex set problem in undirected graphs.

Proof: By construction, whenever $y_i > 0$, $|T_i| = |C_i| \leq 4 \log_2 n$. Thus $|A \cap T_i| \leq 4 \log_2 n$, and by the reasoning above this implies that we have a $4 \log_2 n$-approximation algorithm. $\square$

IEOR E6610 Approximation Algorithms	March 25, 1998
Lecture 9	
Lecturer: David P. Williamson	*Scribe: Mark Zuckerberg*

9.1 The Primal-Dual Method

We continue the discussion of the primal-dual method for approximation algorithms that we started last time. Last time, we started looking at the following:

Hitting Set Problem

- **Input:**
  - The ground set $E=\{e_1, e_2, \ldots, e_n\}$
  - p subsets $T_1, T_2, \ldots, T_p \subseteq E$
  - Costs $c_e \geq 0$ for each edge $e \in E$
- **Goal:** Find a minimum-cost $A \subseteq E$ such that $A \cap T_i \neq \emptyset \quad \forall i$

The integer programming formulation of this problem is:

$$\text{Min} \quad \sum_{e \in E} c_e x_e$$

subject to:

$$\sum_{e \in T_i} x_e \geq 1 \quad \forall i$$

$$x_e \in \{0, 1\}.$$

Here a feasible x satisfies that for every T_i at least one $x_e = 1, e \in T_i$. Therefore if $A = \{e : x_e = 1\}$ then A hits every T_i and has cost $\sum_{e \in E} c_e x_e$.

The LP relaxation of the problem relaxes the x bound to $x_e \geq 0, \forall e \in E$. Its dual is as follows.

$$\text{Max} \quad \sum_{i=1}^{p} y_i$$

subject to:

$$\sum_{i: e \in T_i} x_e \leq c_e \quad \forall e \in E$$

$$y_i \geq 0.$$

The general primal-dual method for approximation algorithms is:

General primal-dual method for approximation algorithms

> $y \leftarrow 0$ (dual feasible)
> While there is no feasible integral x obeying primal complementary slackness ($x_e \neq 0 \Rightarrow \sum_{i:e\in T_i} y_i = c_e$)
>   Get a direction of increase for the dual
> Return feasible x obeying primal complementary slackness.

In order to check the while condition, we note that the while condition only allows $x_e > 0$ for $e : \sum_{i:e\in T_i} y_i = c_e$. Thus if setting $x_e = 1$ (i.e. $e \in A$) for all $e : \sum_{i:e\in T_i} y_i = c_e$ still does not satisfy feasibility (i.e. $\exists\ T_i$ such that $A \cap T_i = \emptyset$) then there is no feasible integral x obeying primal complementary slackness conditions. So we need only check if $A = \{e \in E : \sum_{i:e\in T_i} y_i = c_e\}$ is feasible (i.e. it hits every subset, $A \cap T_i \neq \emptyset, \forall i$).

We claimed that if A is not feasible, then there is some direction of increase for the dual. Note that if A is not feasible then it does not hit every set, so there is some T_k such that $A \cap T_k = \emptyset$. By construction of A this means that $\forall e \in T_k, \sum_{i:e\in T_i} y_i < c_e$. Now y_k appears in these constraints and only in these constraints. It thus follows that every constraint in which y_k participates is not tight ($c_e - \sum_{i:e\in T_i} y_i > 0$), and we can therefore increase y_k by the minimum of these differences while keeping all of these differences ≥ 0. Thus none of the constraints containing y_k are violated - though the one corresponding to that minimum is now tight. Given this new dual feasible solution we may now set x_e for e corresponding to that newly tight constraint to 1, i.e. we may add e to A and the new A will hit T_k as well.

Thus we can translate the general primal-dual method for approximation algorithms to the following algorithm:

Primal-Dual1

> $y \leftarrow 0$
> $A \leftarrow \emptyset$
> While A is not feasible
>   Find violated T_k (i.e. T_k s.t. $A \cap T_k = \emptyset$)
>   Increase y_k until $\exists\ e \in T_k$ such that $\sum_{i:e\in T_i} y_i = c_e$
> $A \leftarrow A \cup \{e\}$
> Return A.

We consider now the performance guarantee of this algorithm. Note that by the construction of A,

$$\sum_{e\in A} c_e = \sum_{e\in A} \sum_{i:e\in T_i} y_i = \sum_i |A \cap T_i| y_i,$$

since each y_i is counted once for each $e \in A$ that is also in T_i. If we could find an α such that whenever $y_i > 0$ then $|A \cap T_i| \leq \alpha$ then it would follow that

$$\sum_{e\in A} c_e \leq \alpha \sum_i y_i \leq \alpha OPT,$$

(since $OPT \geq OPT_{primal} \geq \sum_i y_i$ for any dual feasible solution y) and the above algorithm would be an α-approximation algorithm.

The above algorithm has the shortcoming that though at any particular iteration the edge added to A was needed for feasibility, by the time the algorithm terminates it may no longer be necessary. These unnecessary edges increase the cost of A. Consider the following refinement to remove the unnecessary edges in A:

Primal-Dual2

```
y ← 0
A ← ∅
l ← 0 (l is a counter)
While A is not feasible
    l ← l + 1
    Find violated T_k
    Increase y_k until ∃ e_l ∈ T_k such that Σ_{i:e_l ∈ T_i} y_i = c_{e_l}
    A ← A ∪ {e_l}
For j ← l down to 1
    If A − {e_j} is still feasible
        A ← A − {e_j}
Return A.
```

Let A_f denote the solution returned by the algorithm. The algorithm performs a total of l iterations (where l refers to the value of the counter at termination). Iteration j finds the violated set T_{k_j}, increases the dual variable y_{k_j}, and adds the edge e_j to A. It follows then that $T_{k_j} \cap \{e_1, e_2, \ldots, e_{j-1}\} = \emptyset$ by construction.

To analyze more carefully the performance guarantee of this algorithm, we need the following definition.

Definition 9.1 A set $Z \subseteq E$ is a *minimal augmentation* of a set $X \subseteq E$ if:

1. $X \cup Z$ is feasible, and
2. for any $e \in Z$, $X \cup Z - \{e\}$ is not feasible.

We claim that $A_f - \{e_1, e_2, \ldots, e_{j-1}\}$ is a minimal augmentation of $\{e_1, e_2, \ldots, e_{j-1}\}$. By definition the union is feasible, satisfying condition (1). Now note that $A_f \subseteq \{e_1, e_2, \ldots, e_l\}$ implies that $A_f - \{e_1, e_2, \ldots, e_{j-1}\} \subseteq \{e_j, e_{j+1}, \ldots, e_l\}$. For any $e_t \in \{e_j, e_{j+1}, \ldots, e_l\}$ such that $e_t \in A_f$ as well, letting A_t be the version of A considered by the algorithm in the iteration of its for-loop which attempted (unsuccessfully) to remove e_t we know that $A_t - e_t$ is not feasible (or else e_t would have been removed), but since $A_f \subseteq A_t$ then $A_f - e_t$ is certainly infeasible and condition (2) is satisfied as well.

It follows then that $|A_f \cap T_{k_j}| \leq \max |B \cap T_{k_j}|$ where the maximum is taken over all B such that B is a minimum augmentation of $\{e_1, e_2, \ldots, e_{j-1}\}$.

Theorem 9.1 Let $T(A)$ be the violated set the algorithm chooses given an infeasible A. If

$$\beta = \max_{\text{infeasible } A \subseteq E} \; \max_{\text{minimal augmentations } B \text{ of } A} |B \cap T(A)|,$$

then

$$\sum_{e \in A_f} c_e \;=\; \sum_i |A_f \cap T_i| y_i \;\leq\; \beta \sum_i y_i \;\leq\; \beta OPT.$$

Proof: This follows from $|A_f \cap T_{k_j}| \leq \max |B \cap T_{k_j}| \leq \beta$. □

We see that if we can find a bound β (the maximum number of elements of any violated set chosen by the algorithm that could possibly be introduced under a minimal augmentation) then the above algorithm is a β-approximation algorithm. We now consider two problems in which the above procedure can be implemented.

9.1.1 Finding the shortest s-t path

Here we consider the problem of finding the shortest s-t path in an undirected graph. This problem can be seen as a hitting set problem as follows:

Ground Set : the set of edges E
Costs : $c_e \geq 0, \ \forall e \in E$
Sets to Hit : $T_i = \delta(S_i)$, $s \in S_i$, $t \notin S_i$

where $\delta(S) = \{(u,v) \in E : \ u \in S \text{ and } v \notin S_i\}$. That is, the sets S_i are the s-t cuts and the sets $T_i = \delta(S_i)$ are the edges crossing the s-t cuts.

To see that this hitting set problem captures the shortest s-t path problem, we need that a set of edges contains an s-t path if and only if it hits every s-t cut[1]. First, if a set of edges A does not cross some s-t cut S_i then A must consist exclusively of edges joining two vertices of S_i or joining two vertices of the complement of S_i. Thus any path starting from $s \in S_i$ consisting of such edges can only bring us to vertices that are also in S_i, but $t \notin S_i$. Conversely, if a set of edges does not contain an s-t path then let S_i be the largest connected component (corresponding to those edges) containing s. By assumption $t \notin S_i$ and the set of edges could not contain any edge from $\delta(S_i)$ or else we could have found a larger connected component containing s by including the other vertex incident on that edge. Thus the absence of an s-t path implies that some s-t cut was not hit. We find then that a set of edges contains an s-t path if and only if it hits every s-t cut.

We now wish to apply the algorithm **Primal-Dual2** to this problem. Suppose that whenever A is infeasible, the algorithm chooses the violated set $T_k = \delta(S_k)$, where S_k is the connected component of (V, A) containing s. As is shown above, $A \cap T_k = \emptyset$. We can then prove the following theorem.

[1]This follows directly from the max-flow/min-cut theorem, but for completeness we prove it here.

Theorem 9.2 Given the choice of T_k in each iteration, **Primal-Dual2** is a 1-approximation (optimal) algorithm for the shortest s-t path problem.

Proof: We need only show that $\beta = 1$ for the β defined in Theorem 9.1. Let A be an infeasible solution, and let B be a minimal augmentation of A. Now let $s, v_1, v_2, \ldots, v_l, t$ be an s-t path in $(V, A \cup B)$. Choose i such that $v_i \in S_k$, $v_{i+1} \notin S_k$ where i is as large as possible. Since S_k is a connected component there must be a s-v_i path exclusively in S_k of the form $s, w_1, w_2, \ldots, w_j, v_i$, where $w_\ell \in S_k$. Thus $s, w_1, w_2, \ldots, w_j, v_i, v_{i+1}, \ldots, v_l, t$ is an s-t path, and if we let $B' = \{(v_i, v_{i+1}), (v_{i+1}, v_{i+2}), \ldots, (v_l, t)\}$, then B' is an augmentation. Since all the edges in B' have at least one endpoint not in S_k then (as above) none of these edges is from A which implies that they are all from B, i.e. $B' \subseteq B$. But minimality of B then implies that $B' = B$. So

$$|B \cap \delta(S_k)| = |B' \cap \delta(S_k)| = |\{(v_i, v_{i+1})\}| = 1,$$

since the first edge is the only one to have an endpoint in S_k. Therefore, $\beta = 1$. $\square$

9.1.2 Generalized Steiner Trees

We now consider another problem for which the primal-dual method gives a good approximation algorithm, the Generalized Steiner Tree problem.

Generalized Steiner Tree Problem

- **Input:**
  - An undirected graph $G = (V, E)$
  - l pairs of vertices $(s_i, t_i), i = 1 \ldots l$
  - Costs $c_e \geq 0$ for each edge $e \in E$
- **Goal:** Find a minimum-cost set of edges F such that for all i, s_i and t_i are in the same connected component of (V, F).

This can be modelled as a hitting set problem:

Ground Set : the set of edges E
Costs : $c_e \geq 0, \forall e \in E$
Sets to Hit : $T_i = \delta(S_i)$ where $|S_i \cap \{s_j, t_j\}| = 1$ for some j (the s_j-t_j cuts).

Note that by the logic we used for the shortest s-t path problem that a set of edges will be feasible for this hitting set problem if and only if it contains a path between s_i and t_i for each i.

Let us consider how to apply the algorithm **Primal-Dual2** to this problem. Suppose we do more or less the same thing here we did for the shortest s-t path problem.

We know that if A is not feasible then there must be some connected component S_k containing s_j but not t_j for some j. Suppose the algorithm picks $T_k = \delta(S_k)$ as the violated set. The difficulty is that the reasoning used above in the s-t path problem will not yield a good bound here since a minimal augmentation may cross the cut many times. Consider the problem for which $s = s_1 = s_2 = \cdots = s_l$ and for which there are edges $(s, t_j), \forall j$ and say that $A = \emptyset$. Then $\{(s, t_j)\}, j = 1 \ldots n$ is a minimum augmentation that crosses the cut $\delta(\{s\})$ l times, which (using Theorem 9.1) would imply a β-approximation algorithm, for $\beta \geq l$. This is not very good.

Notice, however, that even though a bound of l hardly tells us anything at all, those l times that the violated set is hit by the augmentation correspond to hits on l different violated sets as well (each $\delta(\{t_i\})$ is hit by (s, t_i)), each of which is hit only once. So on the average the number of hits per violated set (among the group $\{\delta(\{s\}), \delta(\{t_i\}), \forall i\}$) is only $\frac{l+l}{l+1} < 2$. This observation leads to the following variation of the primal-dual method:

Primal-Dual3

```
y ← 0
A ← ∅
l ← 0 (l is a counter)
While A is not feasible
     l ← l + 1
     𝒱 ← Violated(A) (a subroutine returning several violated sets)
     Increase y_k uniformly for all T_k ∈ 𝒱 until ∃ e_l ∉ A such that Σ_{i:e_l∈T_i} y_i = c_{e_l}
     A ← A ∪ {e}
For j ← l down to 1
     If A − {e_j} is still feasible
          A ← A − {e_j}
Return A.
```

The following theorem can be shown about the algorithm **Primal-Dual3**.

Theorem 9.3 If for any infeasible A and any minimal augmentation B of A,

$$\sum_{T_i \in \text{Violated}(A)} |T_i \cap B| \leq \rho |\text{Violated}(A)|.$$

Then

$$\sum_{e \in A_f} c_e \leq \rho \sum_i y_i \leq \rho OPT.$$

Proof: Homework problem, problem set 4. □

For the Generalized Steiner Tree Problem, we let Violated(A) return $\{T_k = \delta(S_k) : S_k \text{ is a connected component of } (V, A), \exists j \text{ s.t. } |S_k \cap \{s_j, t_j\}| = 1\}$.

Theorem 9.4 (Agrawal, Klein, Ravi '95, Goemans, W '95) For this subroutine Violated, **Primal-Dual3** is a 2-approximation algorithm for the Generalized Steiner Tree Problem.

IEOR E6610 Approximation Algorithms April 3, 1998

Lecture 10

Lecturer: David P. Williamson *Scribe: M. Tolga Cezik*

10.1 The Primal-Dual Method: Generalized Steiner Trees cont.

In this lecture, we finish our discussion of the primal-dual method. Recall that we were considering the following:

Hitting Set Problem

- **Input:**
  - The ground set $E=\{e_1, e_2, \ldots, e_n\}$
  - p subsets $T_1, T_2, \ldots, T_p \subseteq E$
  - Costs $c_e \geq 0$ for each edge $e \in E$
- **Goal:** Find a minimum cost $A \subseteq E$ such that $A \cap T_i \neq \emptyset \quad \forall i$.

At the end of last time, we considered another problem for which the primal-dual method gives a good approximation algorithm, the Generalized Steiner Tree problem.

Generalized Steiner Tree Problem

- **Input:**
  - An undirected graph $G = (V, E)$
  - l pairs of vertices $(s_i, t_i), i = 1 \ldots l$
  - Costs $c_e \geq 0$ for each edge $e \in E$
- **Goal:** Find a minimum-cost set of edges F such that for all i, s_i and t_i are in the same connected component of (V, F).

We showed that this problem can be modelled as a hitting set problem:

Ground Set : the set of edges E
Costs : $c_e \geq 0, \forall e \in E$
Sets to Hit : $T_i = \delta(S_i)$ where $|S_i \cap \{s_j, t_j\}| = 1$ for some j (the s_j-t_j cuts).

Recall that the integer programming formulation of the hitting set problem is:

$$\begin{aligned}
\text{Min} \quad & \sum_{e \in E} c_e x_e \\
\text{subject to:} \quad & \\
& \sum_{e \in T_i} x_e \geq 1 \quad \forall i \\
& x_e \in \{0,1\}.
\end{aligned}$$

and the dual of the LP relaxation can be written as:

$$\begin{aligned}
\text{Max} \quad & \sum_{i=1}^{p} y_i \\
\text{subject to:} \quad & \\
& \sum_{i: e \in T_i} y_i \leq c_e \quad \forall e \in E \\
& y_i \geq 0.
\end{aligned}$$

From last time, our analysis of the hitting set problem and the generalized Steiner tree problem led to the following variation of the primal-dual method:

Primal-Dual3

> $y \leftarrow 0$
> $A \leftarrow \emptyset$
> $l \leftarrow 0$ (l is a counter)
> While A is not feasible
> $l \leftarrow l + 1$
> $\mathcal{V} \leftarrow$ Violated(A) (a subroutine returning several violated sets)
>   Increase y_k uniformly for all $T_k \in \mathcal{V}$ until $\exists\ e_l \notin A$ such that $\sum_{i: e_l \in T_i} y_i = c_{e_l}$
> $A \leftarrow A \cup \{e\}$
> For $j \leftarrow l$ down to 1
>   If $A - \{e_j\}$ is still feasible
> $A \leftarrow A - \{e_j\}$
> Return A.

We claimed that the following theorem can be shown about the algorithm, **Primal-Dual3**.

Theorem 10.1 If for any infeasible A and any minimal augmentation B of A,

$$\sum_{T_i \in \text{Violated}(A)} |T_i \cap B| \leq \rho |\text{Violated}(A)|.$$

Then

$$\sum_{e \in A_f} c_e \leq \rho \sum_i y_i \leq \rho OPT.$$

For the GSTP, we let Violated(A) return $\{T_k = \delta(S_k) : S_k$ is a connected component of (V, A), $\exists j$ s.t. $|S_k \cap \{s_j, t_j\}| = 1\}$.

We now prove the following theorem.

Theorem 10.2 (Agrawal, Klein, Ravi '95, Goemans, W '95) For this subroutine Violated, **Primal-Dual3** is a 2-approximation algorithm for the GSTP.

Proof: Given infeasible A, let $\mathcal{C}(A) = \{S : S$ is a connected component of (V, A) s.t. $|S \cap \{s_j, t_j\}| = 1$ for some $j\}$

All we need to show is, for any minimal augmentation B,

$$\sum_{S \in \mathcal{C}(A)} |B \cap \delta(S)| \leq 2|\mathcal{C}(A)|$$

Suppose we contract every connected component of (V, A) where A is an infeasible set of edges. In this contracted graph, call the nodes corresponding to the connected components in $\mathcal{C}(A)$ red and the rest blue. Now consider the graph $G' = (V', B)$ where V' is the vertex set. We note that G' must be a forest, since if it had a cycle we could remove an edge of the cycle and maintain feasibility, contradicting the minimality of B.

How does the inequality we wish to prove translate to the graph G'? Note that $|B \cap \delta(S)|$ in G for a connected component S is equal to $deg(v)$ in G' for the vertex v corresponding to S. Similarly, $|\mathcal{C}(A)|$ in G is simply the number of red vertices in G'. We let Red and $Blue$ represent the sets of red and blue vertices in G', so that we can rewrite the above inequality as

$$\sum_{v \in Red} deg(v) \leq 2|Red|.$$

We will need the following claim.

Claim 10.3 If $v \in Blue$ then $deg(v) \neq 1$.

Proof: If $deg(v) = 1$ then we claim $B - e$ is feasible for $e \in B$ and adjacent to v. If true, this contradicts the minimality of B. Let S be the connected component in G that corresponds to the vertex v in G'. If $B - e$ is not feasible, then there must be some s_i-t_i pair that is connected in $(V, A \cup B)$ but not in $(V, A \cup B - e)$. Thus either s_i or t_i is in S, and the other vertex is in $V - S$. But then it must have been the case that $S \in \mathcal{C}(A)$ and $v \in Red$, which is a contradiction. $\square$

To complete the proof, we first discard all blue nodes with $deg(v) = 0$. Then

$$\begin{aligned}\sum_{v \in Red} deg(v) &= \sum_{v \in Red \cup Blue} deg(v) - \sum_{v \in Blue} deg(v)\\ &\leq 2(|Red| + |Blue|) - 2|Blue|\\ &= 2|Red|\end{aligned}$$

The inequality follows since the sum of the degrees of nodes in a forest is at most twice the number of nodes, and since every blue node has degree at least two. $\square$

This 2-approximation algorithm for the generalized Steiner tree is just an example of the kind of graph problem for which the primal-dual method can obtain a good approximation algorithm. A generalization of the proof above gives 2-approximation algorithms for many other graph problems.

10.2 Metric Methods: Minimum Multicuts

We now turn from the primal-dual method to yet another technique for obtaining approximation algorithms for a wide range of problems. To illustrate this technique, we look at the Minimum Multicut Problem:

Minimum Multicut Problem

- **Input:**
  - An undirected graph $G = (V, E)$
  - k pairs of vertices $(s_i, t_i), i = 1 \ldots k$
  - Costs $c_e \geq 0$ for each edge $e \in E$
- **Goal:** Find a minimum-cost set of edges F such that for all i, s_i and t_i are in different connected components of $G' = (V, E - F)$.

Note that this problem is $\mathcal{NP}$-hard even if G is a tree.

For a given G, let $\mathcal{P}_i$ denote the set of all paths P from s_i to t_i. As usual, we begin by modelling this problem as an integer program. One integer programming formulation of the problem is:

$$\begin{aligned}&\text{Min} \quad \sum_{e \in E} c_e x_e\\ &\text{subject to:}\\ &\qquad \sum_{e \in P} x_e \geq 1 \quad \forall P \in \mathcal{P}_i, \ \forall i\\ &\qquad x_e \in \{0, 1\}.\end{aligned}$$

We then relax the integer program to a linear program by replacing the constraints $x_e \in \{0, 1\}$ by $x_e \geq 0$. The LP relaxation of the above formulation can be solved in polynomial time by the *ellipsoid method*, given a *separation oracle* that runs in polynomial time. A separation oracle is a subroutine which, given some infeasible solution to the LP, returns a violated constraint of the LP. In this case, suppose we have a solution x to the LP. Let x_e denote the length of edge e. Then the LP says that every path from s_i to t_i has length at least 1; in particular, the shortest path from s_i to t_i must have length at least 1. Furthermore, if the shortest path has length at least 1, then every path has length at least 1. Hence a separation oracle for this LP checks if the shortest path from s_i to t_i is at least length 1 for each s_i-t_i pair; if not, a violated constraint is found and returned.

To get some intuition about the LP, we suppose that the LP formulation defines a pipe system, where

- $e = (i, j)$ is a pipe from i to j
- x_e = length of pipe
- c_e = cross section area of pipe

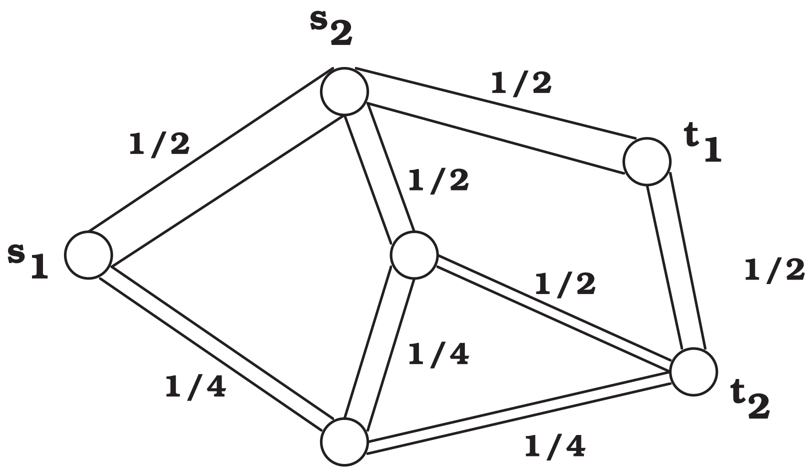

Figure 10.1: LP solution as a pipe system.

For an example of this, see Figure 10.1. Observe that the LP finds the minimum total volume pipe system s.t. $s_i \rightsquigarrow t_i$ length ≥ 1 for all i.

Given an LP solution x, let $dist_x(u, v)$ be the distance from u to v given edge lengths x. We define $B_x(u, r) = \{v \in V : dist_x(u, v) \leq r\}$. This is a ball of radius r around vertex u. See Figure 10.2 for an example.

We use the interpretation of the LP in the following algorithm:

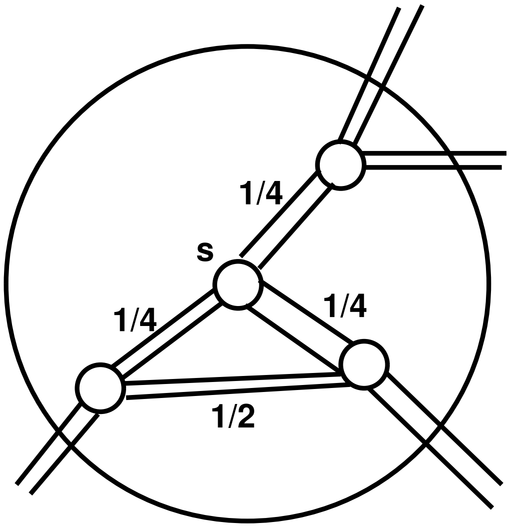

Figure 10.2: Example of a ball of radius 3/8.

GVY

$F \leftarrow \emptyset$
Solve LP and get optimal solution x
While $\exists$ some connected s_i, t_i pair in current graph
    $S = B_x(s_i, r)$ for an appropriate choice of r s.t. $r < 1/2$
    Add $\delta(S)$ to F
    Remove S and edges incident to S from current graph
Return F.

Lemma 10.4 The algorithm terminates.

Proof: Trivial, since each time through the while loop some s_i-t_i pair is disconnected, and there are at most k of them. $\square$

Lemma 10.5 The solution F is a multicut.

Proof: The only possible problem is that when we remove a set S from the graph, it might contain some s_j-t_j pair. But $\exists$no $s_j \rightsquigarrow t_j$ path inside the ball around s_i, $\forall i \neq j$, since

$$\begin{aligned} dist_x(s_j, t_j) &\leq dist_x(s_j, s_i) + dist_x(s_i, t_j) \\ &< 1/2 + 1/2 = 1 \end{aligned}$$

which contradicts the feasibility of the LP solution, since $dist_x(s_j, t_j) \geq 1$. Thus every (s_i, t_i) pair will be separated at the end of algorithm. $\square$

To analyze the algorithm, we will need some notation. We let $V^* = \sum_e c_e x_e \leq OPT$. Fix some iteration of algorithm where (s_i, t_i) pair is chosen. We define

$$\begin{aligned}
V_x(s_i, r) &= \frac{V^*}{k} + \sum_{e=(u,v):u,v \in B_x(s_i,r)} c_e x_e + \sum_{e=(u,v)\in\delta(B_x(s_i,r))} c_e(r - dist_x(s_i, u)), \\
C_x(s_i, r) &= \sum_{e\in\delta(B_x(s_i,r))} c_e.
\end{aligned}$$

That is, $V_x(s_i, r)$ is the total volume of pipe in the ball of radius r around s_i plus an extra term V^*/k. Also, $C_x(s_i, r)$ is the cost of the cut defined by the vertices in the ball of radius r around s_i. The terms V_x and C_x are related, as shown by the following observation.

Observation 10.1 If $\exists$ no $v \in V$ s.t. $dist_x(s_i, v) = r$ then

$$\frac{d}{dr}V_x(s_i, r) \text{ exists and } \frac{d}{dr}V_x(s_i, r) = C_x(s_i, r).$$

The reason for the condition on the observation is that $V_x(s_i, \cdot)$ is possibly discontinuous at values of r such that $dist_x(s_i, v) = r$. Indeed, suppose that there exist u, v and an edge (u, v) such that $dist_x(s_i, v) = dist_x(s_i, u) = r$. Then $V_x(s_i, \cdot)$ is discontinuous at r since at r it has the extra volume of the pipe (u, v) that it does not have for $r - \epsilon$, for any $\epsilon > 0$.

To prove that **GVY** is a good approximation algorithm, we first need the following theorem.

Theorem 10.6

$$\exists \;\; r < 1/2 \quad \text{s.t.} \quad \frac{C_x(s_i, r)}{V_x(s_i, r)} \leq 2\ln 2k.$$

Proof: We prove this by contradiction. First, sort vertices of G according to their distance from s_i. That is, consider vertices $v_1, v_2, \ldots, v_n$ such that $s_i = v_1$, and $r_j = dist_x(s_i, v_j)$ for $r_1 = 0 \leq r_2 \leq \ldots \leq r_l = 1/2$.

Then, for $r \in (r_j, r_{j+1})$ we know that

$$\frac{\frac{d}{dr}V_x(s_i, r)}{V_x(s_i, r)} > 2\ln 2k.$$

Integrating both sides, we obtain

$$\int_{r_j}^{r_{j+1}} \frac{\frac{d}{dr}V_x(s_i, r)}{V_x(s_i, r)} dr > \int_{r_j}^{r_{j+1}} 2\ln 2k \; dr,$$

which is equivalent to

$$\int_{r_j}^{r_{j+1}} \frac{d}{dr}\left(\ln V_x(s_i, r)\right) dr > \int_{r_j}^{r_{j+1}} 2\ln 2k \; dr.$$

Therefore,

$$\ln V_x(s_i, r_{j+1}) - \ln V_x(s_i, r_j) > (r_{j+1} - r_j)(2 \ln 2k).$$

If $V_x(s_i, \cdot)$ is continuous on $[0, 1/2)$, then by summing over $j = 0, 1, \ldots, l-1$ we obtain

$$\ln V_x(s_i, 1/2) - \ln V_x(s_i, 0) > \frac{1}{2}(2 \ln 2k) \quad \text{so that,}$$

$$V_x(s_i, 1/2) > 2k V_x(s_i, 0) = 2k \frac{V^*}{k} = 2V^*$$

which is a contradiction, and proves the theorem. Note that if $V_x(s_i, \cdot)$ is not continuous, then the left-hand side of the inequality can only be greater, and the theorem still follows. $\square$

IEOR E6610 Approximation Algorithms	April 8, 1998
Lecture 11	
Lecturer: David P. Williamson	*Scribe: David de la Nuez*

11.1 Metric Methods

In today's lecture, we continue our discussion of metric methods. We will consider applications to the minimum multicut, balanced cut, and minimum linear arrangement problems.

11.1.1 Minimum Multicut

Recall the minimum multicut problem:

Minimum Multicut Problem

- **Input:**
  - An undirected graph $G = (V, E)$
  - k pairs of vertices $(s_i, t_i), i = 1 \dots k$
  - Costs $c_e \geq 0$ for each edge $e \in E$
- **Goal:** Find a minimum-cost set of edges F such that for all i, s_i and t_i are in different connected components of $G' = (V, E - F)$.

Last time we considered the following LP relaxation of the problem:

$$\text{Min} \quad \sum_{e \in E} c_e x_e$$

subject to:

$$\sum_{e \in P} x_e \geq 1 \quad \forall P \in \mathcal{P}_i, \quad \forall i$$

$$x_e \geq 0,$$

where $\mathcal{P}_i$ denotes the set of all paths P from s_i to t_i. We viewed the solution to the LP as a "pipe system", where

- $e = (i, j)$ is a pipe from i to j
- x_e = length of pipe

- c_e = cross section area of pipe

We defined $dist_x(u,v)$ be the distance from u to v given edge lengths x, $B_x(u,r) = \{v \in V : dist_x(u,v) \leq r\}$, and $V^* = \sum_{e \in E} c_e x_e$ for an optimal LP solution x. We then gave the following algorithm:

GVY

> $F \leftarrow \emptyset$
> Solve LP and get optimal solution x
> While $\exists$ some connected s_i, t_i pair in current graph
> $S = B_x(s_i, r)$ for an appropriate choice of r s.t. $r < 1/2$
>   Add $\delta(S)$ to F
>   Remove S and edges incident to S from current graph
> Return F.

We argued that the algorithm did indeed return a multicut. To prove the performance guarantee of the algorithm, we defined

$$
\begin{aligned}
V_x(s_i, r) &= \frac{V^*}{k} + \sum_{e=(u,v):u,v \in B_x(s_i,r)} c_e x_e + \sum_{e=(u,v) \in \delta(B_x(s_i,r))} c_e(r - dist_x(s_i, u)), \\
C_x(s_i, r) &= \sum_{e \in \delta(B_x(s_i,r))} c_e.
\end{aligned}
$$

That is, $V_x(s_i, r)$ is the total volume of pipe in the ball of radius r around s_i plus an extra term V^*/k. Also, $C_x(s_i, r)$ is the cost of the cut defined by the vertices in the ball of radius r around s_i.

We then proved the following theorem:

Theorem 11.1

$$\exists \;\; r < 1/2 \quad \text{s.t.} \quad \frac{C_x(s_i, r)}{V_x(s_i, r)} \leq 2 \ln 2k.$$

In algorithm GVY, for our choice of an appropriate $r < 1/2$, we choose an $r < 1/2$ such that the theorem is true. How can we find such an r? First, sort vertices of G according to their distance from s_i. That is, consider vertices $v_1, v_2, \ldots, v_n$ such that $s_i = v_1$, and $r_j = dist_x(s_i, v_j)$ for $r_1 = 0 \leq r_2 \leq \ldots \leq r_l = 1/2$. Notice that in any interval (r_j, r_{j+1}) the value of $V_x(s_i, r)$ is increasing, while the value of $C_x(s_i, r)$ stays the same. Thus in any interval (r_j, r_{j+1}) the ratio is largest at the very end of the interval. Thus, given that we know that the theorem is true, we only need check the ratio at $r_{j+1} - \epsilon$ for some tiny value of $\epsilon > 0$, for each j.

We can now prove the final theorem.

Theorem 11.2 (Garg, Vazirani, Yannakakis '96) GVY is a $(4 \ln 2k)$-approximation algorithm for the minimum multicut problem.

Proof: To prove the bound, we charge the cost of the edges in each cut $\delta(B_x(s_i, r))$ added to F in each iteration against the volume removed from the graph plus V^*/k; that is, against $V_x(s_i, r)$. We know that $C_x(s_i, r) \leq (2\ln 2k)V_x(s_i, r)$. Since the edges in $B_x(s_i, r)$ and $\delta(B_x(s_i, r))$ are removed from the graph, we can only charge against these edges once. Thus the total cost of edges in the graph can be no more than $2\ln 2k$ times the total volume of the graph plus $k \times V^*/k$. That is,

$$\begin{aligned}\sum_{e \in F} c_e &\leq 2\ln 2k(V^* + V^*) \\ &\leq (4\ln 2k)V^* \\ &\leq (4\ln 2k)OPT,\end{aligned}$$

since V^* is the value of a linear programming relaxation of the problem. □

11.1.2 Balanced Cut

The method we used to approximate solutions to the minimum multicut problem, namely interpreting an LP solution as a "length" or metric or some clever way, extends nicely to other problems. We consider now one of these, the balanced cut problem:

Balanced Cut Problem

- **Input:**
  - $G = (V, E)$
  - Costs $c_e \geq 0$ for each edge $e \in E$
  - A number, $b \in (0, \frac{1}{2}]$
- **Goal:** Find a set $S \subseteq V$ such that we minimize $\sum_{e \in \delta(S)} c_e$ and satisfies $bn \leq |S| \leq (1-b)n$.

Note that $b = \frac{1}{2}$ gives graph bisections, in which half the nodes of the graph are on each side of the cut. Another typical value is $b = \frac{1}{3}$. But why do we care about balanced cuts? As it turns out (and shall see later today), balanced cuts are useful as subroutines in some divide and conquer strategies. Because each side of the cut contains some constant fraction of the nodes, if we apply this recursively, we can only do this $O(\log n)$ times. Furthermore, the minimization of the edges in the cut makes the "merge" step of such strategies easier or in some way cheaper.

Definition 11.1 By $OPT(b)$, we mean the optimal value of a b-balanced cut.

Innocent as the balanced cut problem sounds and important as it is, there is very little currently know about it. The best result to date is due to Leighton and Rao (1988):

Theorem 11.3 There exists a polynomial-time algorithm for finding a b-balanced cut with $b \leq \frac{1}{3}$ of value $O(\log n) OPT(b')$ for $b' > b + \epsilon$, for any fixed $\epsilon > 0$.

There is a small "cheat" above, in the sense that we don't get an algorithm truly in the spirit of those we have considered in this course: $OPT(b')$ could be quite large compared to $OPT(b)$. Unfortunately, we know no better result. However, we do know of a simplified version of the above result, due to Even, Naor, Rao, and Schieber (1995):

Theorem 11.4 There exists a polynomial time algorithm for finding a $\frac{1}{3}$-balanced cut of value $O(\log n) OPT(\frac{1}{2})$.

As illustration of the fact that we know almost nothing about the balanced cut problem, consider that our current stage of knowledge does not even allow us to disprove the existence of a polynomial-time approximation scheme. So, let us now study the simplified approach, to learn what we can!

Definition 11.2 By $\mathcal{P}_{uv}$ we mean the set of all paths from $u \in V$ to $v \in V$.

Now, consider the following linear program, whose optimal value we can use as a bound:

$$Z_{LP} := \quad \text{Min} \quad \sum_{e \in E} c_e x_e$$

subject to:

$$\sum_{v \in S} \sum_{e \in P_{uv}} x_e \geq \left(\frac{2}{3} - \frac{1}{2}\right) n \quad \forall\ S \ s.t.\ |S| \geq \frac{2}{3} n, u \in S,\ P_{uv} \in \mathcal{P}_{uv}$$

$$x_e \geq 0.$$

The quantification is intended to read that for any S such that $|S| \geq \frac{2}{3}n$, we pick any $u \in S$, and for each $v \in S$, sum the x_e over some u-v path. The total sum over all $v \in S$ should be at least $(\frac{2}{3} - \frac{1}{2})n$.

We show that this LP is a relaxation of the minimum bisection problem.

Lemma 11.5 $Z_{LP} \leq OPT(\frac{1}{2})$.

Proof: Given an optimal bisection S, construct a solution $\bar{x}$ for the LP by setting $\bar{x}_e = 1$ if $e \in \delta(S)$, and $\bar{x}_e = 0$ otherwise. We show that this is a feasible solution, which is enough to prove the lemma.

Consider any S' such that $|S'| \geq \frac{2}{3}\, n$, and any $u \in S'$. Note that there must be at least $(\frac{2}{3} - \frac{1}{2})n$ verticies in $S' - S$, because in the worst case, $S \subseteq S'$ (do the math).

Suppose $u \in S$. As "proof-by-picture", consider Figure 11.1. By our observation about $|S' - S|$, it is easy to see that $\sum_{v \in V} \sum_{e \in P_{uv}} \bar{x}_e \geq (\frac{2}{3} - \frac{1}{2})\, n$.

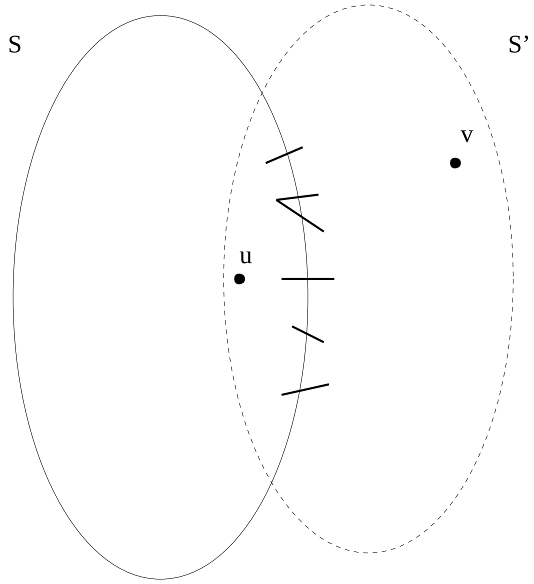

Figure 11.1: Dark lines represent edges whose variable is set to one.

When $u \notin S$, the argument is essentially the same. So, the solution given by $\bar{x}$ is in fact feasible and so $Z_{LP} \leq OPT(\frac{1}{2})$. $\square$

There is a problem we have ignored in all this – the LP could be quite large. However, there are ways around this; in particular, if we can find a so-called "polynomial-time separation oracle", we can apply the ellipsoid method and not worry about the LP being too large. We can get our oracle as follows: given a solution x to check for feasibility, fix a node $u \in V$, run Dijkstra's algotithm (with x_e as edge lengths) to get an ordering of the nodes $\{u = v_1, v_2, \ldots, v_n\}$ from closest to farthest from u. Now consider the sets $S_0 = \left\{v_1, v_2, \ldots, v_{\frac{2}{3}n}\right\}, S_1 = \left\{v_1, v_2, \ldots, v_{\frac{2}{3}n+1}\right\}, \ldots, S_{\frac{1}{3}n} = \{v_1, v_2, \ldots, v_n\}$. If some constraint is violated for this choice of u, then certainly it must be violated for one of these sets S_i since these are the sets of vertices closest to u; that is, the sum of the path lengths can be no smaller for any other S such that $|S| \geq \frac{2}{3}n$.

We are now in position to state an algorithm, and begin its analysis.

ENRS

> Solve the LP for optimal x.
> $S \leftarrow \emptyset$
> $F \leftarrow \emptyset$
> While $|S| < \frac{n}{3}$
>   Choose some u, v pair in the current graph such that
> $dist_x(u,v) \geq \frac{1}{6}$
> $C \leftarrow B_x(u,r)$, for an appropriate $r < \frac{1}{12}$
> $C' \leftarrow B_x(v,r')$, for an appropriate $r' < \frac{1}{12}$
>   Add C or C' to S, whichever gives $\min\{|C|, |C'|\}$
>   and $\delta(C)$ (or $\delta(C')$) to F
>   Remove C (or C') from the graph.

Lemma 11.6 If $|S| < \frac{n}{3}$, there exists a u, v pair in the current graph such that $dist_x(u,v) \geq \frac{1}{6}$.

Proof: Consider $\bar{S} = V - S$. Then $|\bar{S}| > \frac{2n}{3}$. This implies that $\sum_{v \in \bar{S}} \sum_{e \in P_{uv}} x_e \geq (\frac{2}{3} - \frac{1}{2})n = \frac{n}{6}, \forall\ u \in S,\ P_{uv} \in \mathcal{P}_{uv}$. But then the average path length from u to a $v \in \bar{S}$ is at least $\frac{1}{6}\frac{n}{|\bar{S}|} \geq \frac{1}{6}$. Thus by the pigeon-hole principle, there exists some $v \in \bar{S}$ such that $dist_x(u,v) \geq \frac{1}{6}$. □

Lemma 11.7 ENRS outputs a $\frac{1}{3}$-balanced cut.

Proof: $|S| \geq \frac{n}{3}$ at termination, by design. So, we only need to show that $|S| \leq \frac{2n}{3}$. Choose *any* iteration of the while loop. At the begining of the iteration, we certainly have $|S| < \frac{n}{3}$, by design, and at the end of the iteration, $|S| \leftarrow |S| + \min\{|C|, |C'|\}$. Note that because $dist_x(u,v) \geq \frac{1}{6}$ but $C = B_x(u,r)$ and $C' = B_x(v,r')$ for $r, r' < \frac{1}{12}$, it must be the case that C and C' are disjoint. This implies that $\min\{|C|, |C'|\} \leq \frac{1}{2}(n - |S|)$, i.e. the smaller of C and C' can be no more than half the remaining vertices. So,

$$\begin{aligned} |S| + \min\{|C|, |C'|\} &\leq |S| + \frac{1}{2}(n - |S|) \\ &= \frac{1}{2}n + \frac{1}{2}|S| \\ &\leq \frac{2n}{3} \end{aligned}$$

□

We have reached the point where the analysis will begin to look very familiar, i.e. we follow closely the model of the minimum multicut analysis. So, we present the following definitions, analagous to those we have seen before:

Definition 11.3

$$V_x(u,r) := \frac{Z_{LP}}{n} + \sum_{e=(u,v)\ :\ v,w\ \in\ B_x(u,r)} c_e + \sum_{e=(v,w)\ \in\ \delta(B_x(u,r))} c_e(r - dist_x(u,w))$$

Definition 11.4

$$C_x(u,r) = \sum_{e=(v,w)\ \in\ \delta(B_x(u,r))} c_e$$

And now we prove a familiar looking theorem, whose bound is somewhat different, but whose proof is practically identical (and so we omit it):

Theorem 11.8 There exists $r < \frac{1}{12}$ such that $\frac{C_x(u,r)}{V_x(u,r)} \leq\ 12 \ln 2n$

We find the "appropriate choice" of r as before (i.e., one that achieves the bound in the theorem). Now we may state the final theorem, due to Even, Naor, Rao, and Schieber (1995):

Theorem 11.9 There exists a polynomial time algorithm for finding a $\frac{1}{3}$-balanced cut S such that

$$\sum_{e\ \in\ \delta(S)} c_e \leq (24 \ln 2n) Z_{LP} = (24 \ln 2n) OPT \left(\frac{1}{2}\right).$$

We omit the proof due to its similarity to the analagous proof in the analysis of the minimum multicut problem.

11.1.3 Minimum Linear Arrangement

We promised earlier to look at an example of an application of balanced cuts. One application is in solving the minimum linear arrangement problem:

Minimum Linear Arrangement Problem

- **Input:**
  - $G = (V, E)$, undirected.
  - Costs $c_e \geq 0$ for each edge $e \in E$
- **Goal:** Find a bijection $f : V \leftarrow \{1, 2, \ldots, n\}$ which minimizes

$$\max_i \sum_{(u,v) \in E : f(u) \leq i, f(v) > i} c_e.$$

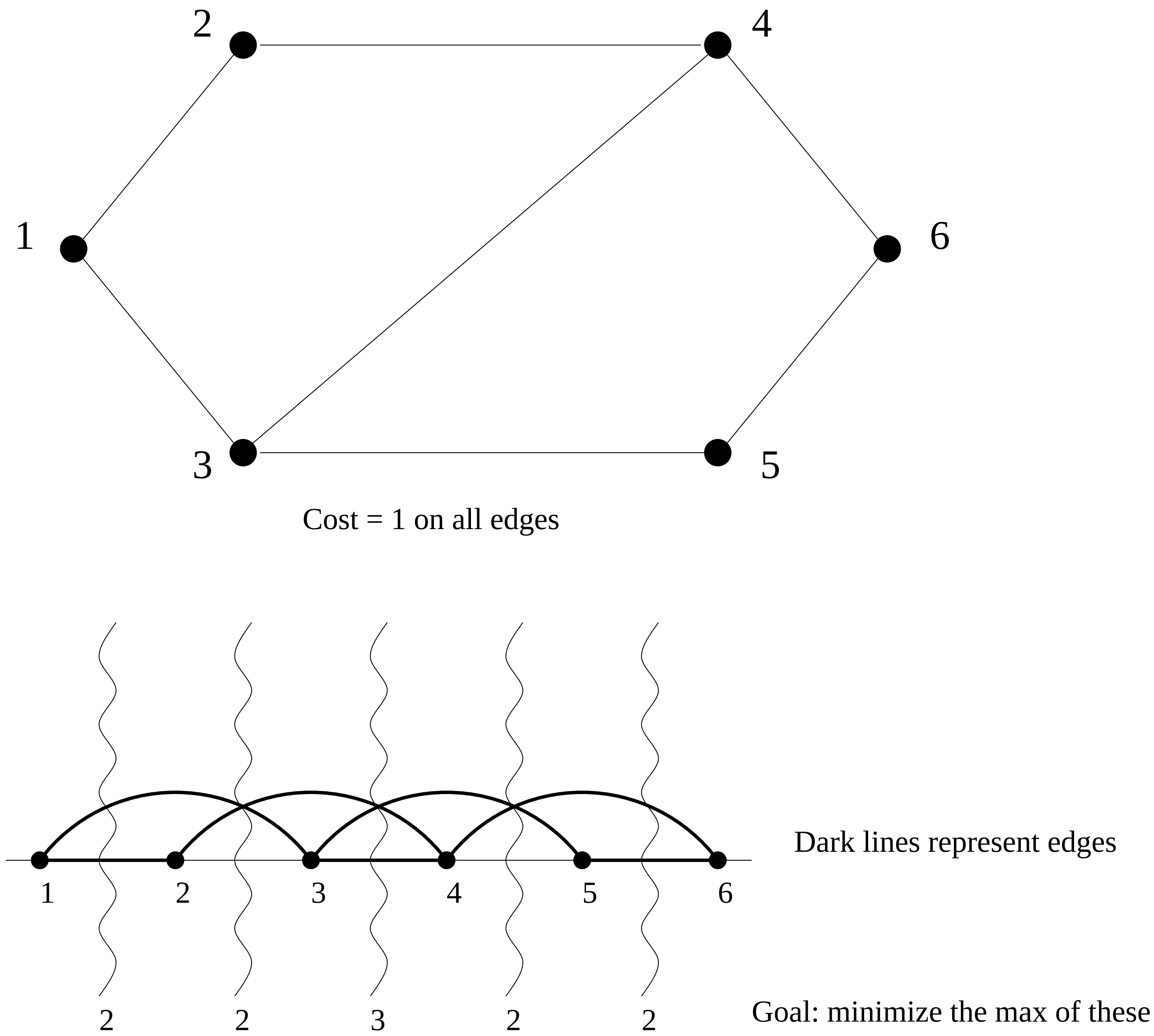

Figure 11.2: A graph and a representation of a corresponding linear arrangement.

As an example, consider Figure 11.2. The curly lines represent "$f(u) \le i, f(v) > i$". The goal is to minimize the maximum number of edges crossing a curly line.

In the past we've used optimal value of a linear programs to bound in some intelligent manner the value of something we're interested in. Balanced cuts will allow us to do a similar trick for the linear arrangement problem.

Lemma 11.10 $OPT(\frac{1}{2}) \le OPT_{LA}$, where OPT_{LA} is the optimal value of the linear arrangement problem.

Proof: The proof here is quite simple: any linear arrangment gives us a bisection by splitting the arrangement at $\frac{n}{2}$. See Figure 11.3 for a "proof-by-picture".

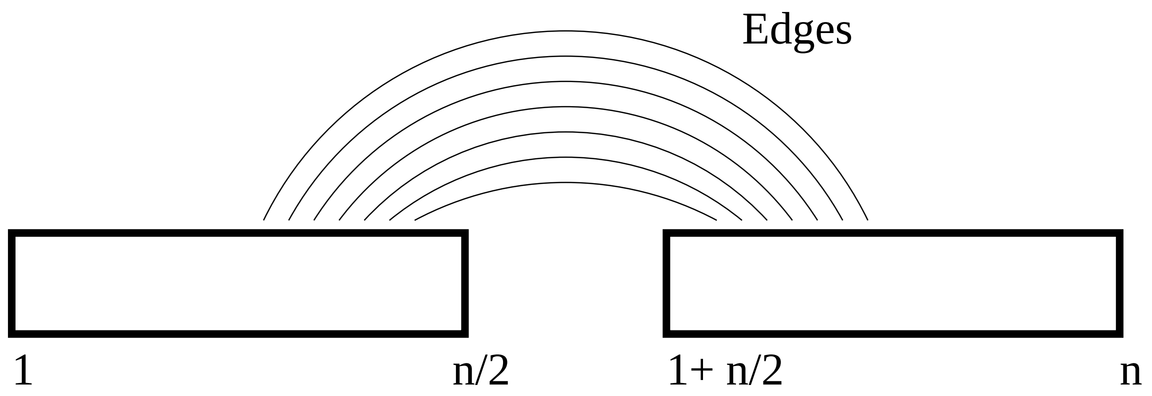

Figure 11.3: Getting a bisection from a linear arrangment

So then $OPT(\frac{1}{2}) \le LA$, where LA is the value of the linear arrangement. In particular, this is true of the optimal arrangment, so we are done. $\square$

This motivates the following algorithm:

LAYOUT$(V, E, \{i_1, i_2, \ldots, i_{|V|}\})$

---

if $V = \{v\}$ (i.e. singleton set)
  $f(v) \leftarrow i_1$;
else
  Find a $\frac{1}{3}$-balanced cut S
  LAYOUT$(S, E_S, \{i_1, i_2, \ldots, i_{|S|}\})$
  LAYOUT$(\bar{S}, E_{\bar{S}}, \{i_1, i_2, \ldots, i_{|\bar{S}|}\})$

Note that we initially call LAYOUT $(V, E, \{1, 2, \ldots, n\})$.

Theorem 11.11 LAYOUT is an $O(\log^2 n)$-approximation algorithm for the linear arrangment problem.

Proof:

Let $L(V,E) :=$ value of LAYOUT(V, E).

Let $B(V,E) :=$ value of $\frac{1}{3}-$balanced cut S of (V, E).

Observe that because in the worst case, all the edges of the balanced cut appear in all the divisions of the layout,

$$\begin{aligned} L(V,E) &\leq \max\left\{L(S,E_S), L(\bar{S},E_{\bar{S}})\right\} + B(V,E) \\ &\leq \max\left\{L(S,E_S), L(\bar{S},E_{\bar{S}})\right\} + O(\log n) OPT\left(\frac{1}{2}\right) \\ &\leq \max\left\{L(S,E_S), L(\bar{S},E_{\bar{S}})\right\} + O(\log n) OPT_{LA}. \end{aligned}$$

But applying the inequality above recursively, combined with the fact that our recursion tree is $O(\log n)$ deep, we get that

$$\begin{aligned} L(V,E) &\leq \max\left\{L(S,E_S), L(\bar{S},E_{\bar{S}})\right\} + O(\log n) OPT_{LA} \\ &\leq O(\log n) O(\log n) OPT_{LA} \\ &= O(\log^2 n) OPT_{LA}. \end{aligned}$$

□

12.1 Scheduling problems and LP

12.1.1 Some deterministic scheduling notation

We turn to discussion of some recent work on approximation algorithms for deterministic scheduling problems. Before we get to that, we will need to review some scheduling notation due to Graham, Lawler, Lenstra, and Rinnooy Kan. In their notation, scheduling problems are represented as *Machine environment*|*Constraints*|*Objective function*. We give some examples of the possibities. Two possible elements in the environment field are 1 (for single machine problems) and P (for identical parallel machine problems). Two possible elements in the constraint field are r_j (when each job j has a "release date". Then job j is not available before time r_j) and *prec* (e.g. $j \prec k$. This implies that we cannot start processing job k before job j is finished). Assume C_j is the completion time of job j. Then two possible elements in the objective field are C_{max} (meaning $\min \max_j C_j$, the maximum completion time of all jobs) and $\sum_j w_j C_j$ (meaning $\min \sum_j w_j C_j$, where w_j is weight of job j).

As an example, the scheduling problem we looked at earlier in the semester, scheduling identical parallel machines to minimize the maximum completion time, is denoted $P||C_{max}$.

12.1.2 $1||\sum_j w_j C_j$

We now turn to the problem $1||\sum_j w_j C_j$.

$1||\sum_j w_j C_j$

- **Input:** n jobs $J_1 ... J_n$ with weights $w_1 ... w_n$ and processing times $p_1 ... p_n$.
- **Goal:** Find a schedule on 1 machine that minimizes $\sum_j w_j C_j$.

To think about a scheduling algorithm for this problem, suppose we have a schedule on 1 machine and jobs j and k are adjacent. What happens if we swap them? Nothing changes for any job before or after j and k. If C' denotes the new completion times after the job swap, then $C'_j = C_j + p_k$, $C'_k = C_k - p_j$, and $C'_l = C_l$ for all other jobs l.

When is this swap an improvement? Obviously, when $\sum_j w_j C'_j - \sum_j w_j C_j < 0$, which in this case is when $w_j p_k - w_k p_j < 0$, which implies $w_j p_k < w_k p_j$ or $\frac{w_j}{p_j} < \frac{w_k}{p_k}$. Therefore no improvement is possible if and only if the jobs are scheduled in nonincreasing order of $\frac{w_j}{p_j}$. This observation is called *Smith's Rule* and is the basis of the following theorem.

Theorem 12.1 (Smith '56) The optimal schedule for $1||\sum_j w_j C_j$ can be found by scheduling jobs in order of non-decreasing $\frac{w_j}{p_j}$ (and thus can be found in $O(n \log n)$ time).

But even minor twists make the problem hard: $1|r_j|\sum_j w_j C_j$ and $1|prec|\sum_j w_j C_j$ are NP-hard.

Corollary 12.2 If $p_j = w_j \; \forall j$ then any schedule without idle time is optimal; i.e. $\sum_j w_j C_j$ is the same for any such schedule.

Using this corollary, we will develop linear constraints on variables C_j that are valid for all schedules. From this, we will be able to develop a linear programming relaxation for $1||\sum_j w_j C_j$ and its various variants. Thus if we schedule jobs in the order of their indices we have that

$$\sum_j w_j C_j = \sum_j p_j C_j = \sum_j \sum_{1 \le k \le j} p_k = \sum_{j,k:k \le j} p_j p_k.$$

Then for *any* schedule (including those with idle time)

$$\sum_j p_j C_j \ge \sum_{j,k:k \le j} p_j p_k,$$

since adding idle time can only increase the left-hand side.

Now, suppose jobs in $S \subseteq \{1, 2, \ldots, n\}$ are all scheduled first. As before, when $w_j = p_j$ the order of the jobs is immaterial, since $\sum_{j \in S} w_j C_j$ will be the same no matter what order the jobs are scheduled in. Then

$$\sum_{j \in S} w_j C_j = \sum_{j \in S} p_j C_j = \sum_{j \in S} p_j \left(\sum_{k \in S: k \le j} p_k \right) = \sum_{j,k \in S: k \le j} p_j p_k.$$

Notice then that if we consider *any* schedule (ones including idle time, or ones in which jobs not in S finish before those in S), the sum $\sum_{j \in S} p_j C_j$ can only increase. Thus we have that

$$\sum_{j \in S} p_j C_j \ge \sum_{j,k \in S: k \le j} p_j p_k$$

is valid for any schedule.

We introduce the following notation: $p(S) = \sum_{j \in S} p_j$, $p(S)^2 = (\sum_{j \in S} p_j)^2$, $p^2(S) = \sum_{j \in S} p_j^2$. Then we can rewrite the inequality as

$$\sum_{j \in S} p_j C_j \ge \sum_{j,k \in S: k \le j} p_j p_k = \frac{1}{2}(p(S)^2 + p^2(S)).$$

We can now write a linear programming relaxation of the scheduling problem $1||\sum_j w_j C_j$.

$$
\begin{aligned}
Z_{LP} = \quad & \text{Min} \quad \sum_j w_j C_j \\
& \text{subject to:} \\
& \sum_j p_j C_j \geq \frac{1}{2}(p(S)^2 + p^2(S)) \qquad \forall S \subseteq \{1, \ldots, n\}.
\end{aligned}
$$

Claim 12.3 There is a polynomial-time separation oracle for these inequalities.

Thus we can solve the LP in polynomial-time using the ellipsoid method.

Although we won't need this fact, it is interesting to note that the LP relaxation completely captures the problem $1||\sum_j w_j C_j$.

Theorem 12.4 (Wolsey '85, Queyranne '93) Z_{LP} gives the optimal value for $1||\sum_j w_j C_j$.

Proof: We will show that the completion times C_j from the schedule given by Smith's rule give an optimal LP solution. Assume $\frac{w_1}{p_1} \geq \cdots \geq \frac{w_n}{p_n}$, so that the optimal schedule is $C_j = \sum_{1 \leq k \leq j} p_k$. By previous arguments, the LP constraints will be tight for the sets $S = \{1\}, \{1,2\}, \ldots, \{1,2,\ldots,n\}$.

Now, consider the dual LP:

$$
\begin{aligned}
Z_{LP} = \quad & \text{Max} \quad \frac{1}{2}\sum_S (p(S)^2 + p^2(S)) y_S \\
& \text{subject to:} \\
& \sum_{S: j \in S} y_S = \frac{w_j}{p_j} \qquad \forall j \\
& y_S \geq 0 \qquad \forall S \subseteq \{1, \ldots, n\}.
\end{aligned}
$$

To prove that the schedule is optimal, we will construct a dual feasible solution that obeys complementary slackness with respect to the C_j. This implies that the C_j give the optimal LP solution.

Set

$$
\begin{aligned}
y_{\{1\}} &= \frac{w_1}{p_1} - \frac{w_2}{p_2} \\
y_{\{1,2\}} &= \frac{w_2}{p_2} - \frac{w_3}{p_3} \\
&\vdots \\
y_{\{1,\ldots,n-1\}} &= \frac{w_{n-1}}{p_{n-1}} - \frac{w_n}{p_n} \\
y_{\{1,\ldots,n\}} &= \frac{w_n}{p_n},
\end{aligned}
$$

and set $y_S = 0$ otherwise. Then note that $y_S \geq 0$ for all S. Also note that for any job j

$$\sum_{S: j \in S} y_S = \sum_{k=j}^{n-1} \left(\frac{w_k}{p_k} - \frac{w_{k+1}}{p_{k+1}} \right) + \frac{w_n}{p_n} = \frac{w_j}{p_j}$$

for all jobs j. Thus these y_S are feasible for the dual LP and obey complementary slackness, since whenever $y_S > 0$ the corresponding primal inequality is tight. $\square$

12.1.3 $1|prec|\sum_j w_j C_j$

We now turn to the problem $1|prec|\sum_j w_j C_j$. To add precedence constraints to the LP, just add the the inequalities $C_k \geq C_j + p_k$ for $j \prec k$. Now consider the following algorithm:

Schedule-by-$\bar{C}_j$
Solve LP; Obtain opt solution $\bar{C}$. (Assume $\bar{C}_1 \leq \bar{C}_2 \leq \ldots \leq \bar{C}_n$). Schedule jobs in order $1, 2, \ldots, n$, i.e. $\tilde{C}_j = \sum_{1 \leq k \leq j} p_k$

Note that the solution will obey the precedence constraints since $\bar{C}_j < \bar{C}_k$ by the LP constraints whenever $j \prec k$ (assuming $p_k \neq 0$).

We can prove the following lemma about the LP solution.

Lemma 12.5 $\bar{C}_j \geq \frac{1}{2}(\sum_{1 \leq k \leq j} p_j)$

Proof: We know for $S = \{1, 2, \ldots, j\}$

$$\sum_{1 \leq k \leq j} p_k \bar{C}_k \geq \frac{1}{2}(p(S)^2 + p^2(S)) \geq \frac{1}{2} p(S)^2 = \frac{1}{2} \left(\sum_{1 \leq k \leq j} p_k \right)^2.$$

Since $\bar{C}_j \geq \bar{C}_k$ for $k \leq j$ then

$$\bar{C}_j \sum_{1 \leq k \leq j} p_k \geq \sum_{1 \leq k \leq j} p_k \bar{C}_k \geq \frac{1}{2} \left(\sum_{1 \leq k \leq j} p_k \right)^2.$$

Dividing both sides by $\sum_{1 \leq k \leq j} p_k$ gives the lemma statement. $\square$

The lemma leads to the following theorem.

Theorem 12.6 (Hall, Schulz, Shmoys, Wein '97) Schedule-by-$\bar{C}_j$ is a 2-approximation algorithm.

Proof:

$$\begin{aligned}
\sum_{1\le j\le n} w_j \tilde{C}_j &= \sum_{1\le j\le n} w_j \left(\sum_{1\le k\le j} p_k\right) \\
&\le 2 \sum_{1\le j\le n} w_j \bar{C}_j \\
&= 2Z_{LP} \\
&\le 2OPT.
\end{aligned}$$

□

12.1.4 $1|r_j|\sum_j w_j C_j$

We finally turn to the problem $1|r_j|\sum_j w_j C_j$. In order to get an approximation algorithm for this problem, we consider a different linear programming relaxation of the problem. We introduce variables y_{jt} where

$$y_{jt} = \begin{cases} 1 & \text{if job } j \text{ processed in time } (t-1, t] \\ 0 & \text{otherwise} \end{cases}$$

Let T denote $\max_j r_j + \sum_{1\le j\le n} p_j$, which is the latest possible completion time for any job. We now formulate a series of constraints for the LP. First note that

$$\sum_{1\le j\le n} y_{jt} \le 1 \text{ for } t = 1, \ldots, T,$$

since at most one job can be processed at any point in time. Also,

$$\sum_{1\le t\le T} y_{jt} = p_j \quad \forall j,$$

since the total amount of processing for job j must be p_j. In order to impose that no job is processed before its release date, we have

$$y_{jt} = 0 \quad \forall t = 1, \ldots, r_j \quad \forall j,$$

but

$$y_{jt} \ge 0 \quad \forall t > r_j \quad \forall j.$$

Finally, because we wish to minimize $\sum_j w_j C_j$, we need to express C_j in terms of the variables y_{jt}. Note that in valid schedule $y_{j,C_j-p_j+1} = y_{j,C_j-p_j+2} = \ldots = y_{j,C_j} = 1$. Thus we can express

$$C_j - \frac{p_j}{2} = \frac{1}{p_j} \sum_{r_j+1\le t\le T} y_{jt}(t - \frac{1}{2}).$$

That is, the midpoint of the execution of the job, $C_j - \frac{p_j}{2}$, is the average over all midpoints of the time intervals that job j is processed.

Thus we have the following linear programming relaxation of the problem $1|r_j|\sum_j w_j C_j$:

$$
\begin{array}{lll}
\text{Min} & \sum_j w_j C_j & \\
\text{subject to:} & & \\
& \sum_{1\le j\le n} y_{jt} \le 1 & t = 1,\ldots,T \\
& \sum_{1\le t\le T} y_{jt} = p_j & \forall j \\
& y_{jt} = 0 & \forall t = 1,\ldots,r_j, \forall j \\
& y_{jt} \ge 0 & \forall t > r_j, \forall j \\
& C_j = \frac{1}{p_j} \sum_{r_j+1\le t\le T} y_{jt}(t-\frac{1}{2}) + \frac{p_j}{2} & \forall j
\end{array}
$$

Thus if Z_{LP} is the value of this LP relaxation, then $Z_{LP} \le OPT$.

This seems very bad because we have an exponential number of variables (in the size of the input) and an exponential number of constraints. But it turns out it is not so bad.

Claim 12.7 (Goemans '96) The LP can be solved in $O(n \log n)$ time.

To get an approximation algorithm given an LP solution, we invoke our old friend, randomized rounding.

RandomRound

> Solve LP, get optimal solution $(\bar{C}_j, \bar{y}_{jt})$.
> Set $T_j = t - \frac{1}{2}$ with probability $\frac{\bar{y}_{jt}}{p_j}$ $\forall j$.
> Schedule jobs as early as possible in same order as T_j
> (because of r_j there can be idle time in the resulting schedule).

Suppose that $T_1 \le T_2 \le \cdots \le T_n$. Let $\tilde{C}_j$ be random variable giving the completion time of job j. We begin the analysis by considering the expected value of $\tilde{C}_j$ given a fixed value of T_j.

Lemma 12.8 $E[\tilde{C}_j|T_j] \le p_j + 2T_j$

Proof: Let I be a random variable giving the total idle time before job j is processed, and let P be a random variable giving the total amount of time spent processing jobs before j starts. Then obviously $\tilde{C}_j = I + P + p_j$.

Since idle time before job j can only result from the release dates of jobs to be scheduled before job j, the idle time is at worst

$$\max_{k:T_k\le T_j} r_k \le \max_{k:T_k\le T_j} T_k \le T_j,$$

so that $I \leq T_j$.

Furthermore,

$$\begin{aligned}
E[P|T_j] &= \sum_{k \neq j} p_k \Pr[\text{job } k \text{ is processed before } j|T_j] \\
&= \sum_{k \neq j} p_k \Pr[T_k \leq T_j|T_j] \\
&= \sum_{k \neq j} p_k \left(\sum_{1 \leq t \leq T_j} \frac{y_{kt}}{p_k} \right) \\
&= \sum_{k \neq j} \sum_{1 \leq t \leq T_j} y_{kt} \\
&= \sum_{1 \leq t \leq T_j} \sum_{k \neq j} y_{kt} \\
&\leq T_j,
\end{aligned}$$

since $\sum_{k \neq j} y_{kt} \leq 1$ by the linear program.

Therefore

$$E[\tilde{C}_j|T_j] = p_j + E[I|T_j] + E[P|T_j] \leq p_j + 2T_j.$$

□

We can now prove the following theorem.

Theorem 12.9 (Schulz, Skutella '96) RandomRound is a 2-approximation algorithm for $1|r_j|\sum_j w_j C_j$

Proof: Notice that

$$\begin{aligned}
E[\tilde{C}_j] &\leq p_j + 2 \sum_{1 \leq t \leq T} \left(t - \frac{1}{2}\right) \Pr\left[T_j = t - \frac{1}{2}\right] \\
&= p_j + 2 \sum_{1 \leq t \leq T} \left(t - \frac{1}{2}\right) \frac{y_{jt}}{p_j} \\
&= 2 \left(\frac{p_j}{2} + \frac{1}{p_j} \sum_{1 \leq t \leq T} \left(t - \frac{1}{2}\right) y_{jt} \right) \\
&= 2\bar{C}_j,
\end{aligned}$$

where the last equality follows from the LP formulation. Thus

$$E[\sum_j w_j \tilde{C}_j] = \sum_j w_j E[\tilde{C}_j] \leq 2 \sum_j w_j \bar{C}_j \leq 2OPT.$$

□

The first 2-approximation algorithm for this problem was due to Goemans.

Here's an alternate perspective on what the algorithm is doing: the LP solution is a preemptive schedule (one in which the processing of any job j need not be continuous). Pick $\alpha_j \in [0, 1]$ uniformly for each j. Let T_j be α_j-point of job j: that is, the time when $\alpha_j p_j$ units of job j have been processed in the preemptive schedule. Then schedule jobs according to T_j as before. The $O(n \log n)$-time algorithm of Goemans to find the LP solution actually finds this preemptive schedule.

The best known approximation algorithm for $1|r_j|\sum_j w_j C_j$, due to Goemans, Queyranne, Schulz, Skutella, and Wang (1998) has a performance guarantee of 1.6853, and is the same as above, except α_j is picked non-uniformly from [0,1]. It is conjectured that there is an $\frac{e}{e-1} \approx 1.58$-approximation algorithm. This is known to be true for $1|r_j|\sum_j C_j$.

13.1 A PTAS for Euclidean TSP

Today, we give a polynomial-time approximation scheme for Euclidean TSP due to Arora (1996, 1997) which finds a tour with cost no more than $(1+\epsilon)OPT$ in $O(n \log^{O(\frac{1}{\epsilon})} n)$ time. Recall that Euclidean TSP is a special case of TSP in which vertices correspond to points in the plane, and the cost of an edge (i,j) is the Euclidean distance between the corresponding points. It turns out that this technique for Euclidean TSP also works for Euclidean Steiner trees, perfect matchings, problems called k-MST and k-TSP (in which one must find the min-cost MST and TSP on k out of the n points), problems in $\Re^d$, and more.

First, some interesting history behind this result. Mitchell (1996) independently discovered a similar PTAS soon after Arora had made a preliminary announcement of his result. Also, Arora is better known for his work showing that for many problems, no PTAS can exist unless $P = NP$. He was trying to prove the same for Euclidean TSP, when he realized it wouldn't work, and figured out a PTAS.

The basic strategy for obtaining the PTAS is (as it has been for most of the PTAS's we've seen) to apply dynamic programming. The main difficulty is getting the problem to a point where we can apply dynamic programming. Our overall strategy for doing so is the following:

1. We perturb the instance to get nice properties, increasing OPT by at most ϵOPT.
2. We subdivide the plane randomly, in a way to be defined later.
3. We show that with probability at least $\frac{1}{2}$, a highly structured tour of cost no more than $(1+\epsilon)OPT$ exists for the perturbed instance with respect to the random subdivision.
4. We use dynamic programming to find cheapest tour with the structure of part 3.

13.1.1 Perturbing the Problem Instance

We want to perturb the problem instance so that the following properties hold:

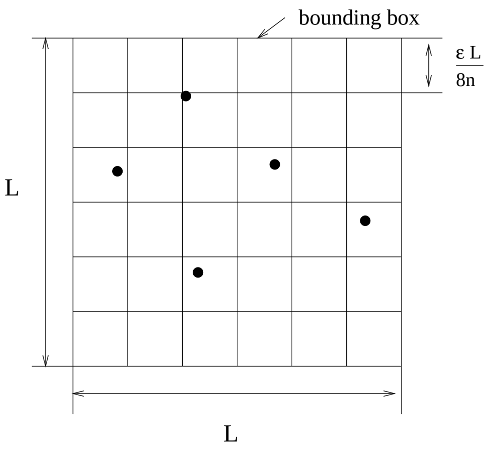

Figure 13.1: Illustration of bounding box and grid.

1. All points have integer coordinates.
2. The minimum nonzero distance is at least 8.
3. The maximum distance is $O(n)$.

Let L denote the length of the "bounding box" around the points in the instance (see Figure 13.1). Observe that $L \leq OPT$. To get a perturbed instance with the properties we want, we first put down a grid of spacing $\frac{\epsilon L}{8n}$ (see Figure 13.1), and move the points to nearest grid points. How does this affect the length of the optimal tour? Well, each edge increases by at most $\frac{2\epsilon L}{8n}$, which implies that the total increase is at most $n\frac{2\epsilon L}{8n} = \frac{\epsilon L}{4} \leq \frac{\epsilon}{4}OPT$. Thus an optimal tour in this instance has length at most $(1+\frac{\epsilon}{4})OPT$.

We now blow up the grid spacing by a factor of $\frac{64n}{\epsilon L}$. This implies that every point is now at some multiple of $\frac{\epsilon L}{8n}(\frac{64n}{\epsilon L}) = 8$, which gives us desired properties 1 and 2. Furthermore, the maximum distance is now $O(L)\frac{64n}{\epsilon L} = O(\frac{64n}{\epsilon}) = O(n)$, assuming ϵ is fixed. Clearly if we find a near-optimal tour in the "blown up" instance, the same tour will also be near optimal in the non-blown-up instance.

13.1.2 Subdividing the Plane

To apply dynamic programming, we need some structure on the problem whereby we can build up a solution from the solution to smaller problems. To do this, we are going to subdivide the plane in a random way. First we will consider the subdivision without randomness, and then we will introduce the randomization.

To create the subdivision, we divide the bounding box into four equally-sized boxes, and then recursively divide the boxes into four boxes, and so on (See Figure

13.2). We will consider two different types of subdivisions, which differ only in when we stop the recursion. For a *quadtree*, we stop when there is at most 1 node per square. For a *dissection*, we stop when each box has side length 1. The subdivision in Figure 13.2 is a quadtree.

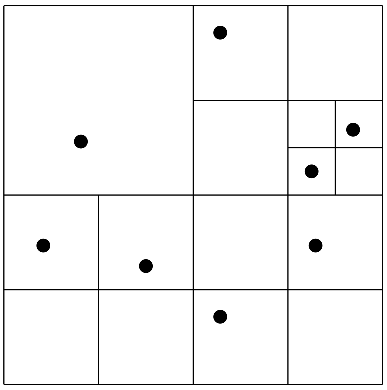

Figure 13.2: Example of a quadtree.

We will need to talk about the different levels of recursion in subdividing the instance. We will say that the bounding box has level 0, and the boxes in the subdivision of a level i box are at level $i+1$. Since the maximum distance between points is $O(n)$ and points at non-zero distance are at least distance 8 apart, the level of smallest box is $O(\log n)$. Thus it is not difficult to see that the total number of boxes in quadtree is $O(n \log n)$.

We introduce randomness into the subdivision by "shifting" the center of dissection. If the side length of the bounding box is L, the original center of the dissection is $(\frac{L}{2}, \frac{L}{2})$. To shift the dissection, we pick a, b from $[0, L)$ randomly, and start dissection at center $(\frac{L}{2} + a(\bmod L), \frac{L}{2} + b(\bmod L))$. We call dissections and quadtrees created in this way (a, b)-dissections and (a, b)-quadtrees. See Figure 13.3 for an example.

13.1.3 The Structure Theorem

Given the subdivisions of the plane, we can now begin discussing how to use them to find a tour that is near optimal. To do this, for each box in the (a, b)-quadtree, we put m regularly spaced points on each side and one at each corner. We call these points *portals.* The main idea that enables us to carry out dynamic programming is that we will look for tours that cross the sides of boxes *only* through portals.

Let's define such a tour. We say we have a *salesman path* if we have a connected set of edges such that degree at portals is even (possibly 0), and degree at nodes is 2. We know from previous discussion of the TSP that if we find a short salesman path, then by shortcutting the portals we can find a tour of cost no greater. We define an

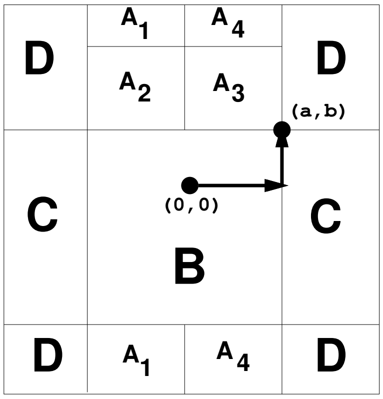

Figure 13.3: Example of the beginning of an (a, b)-dissection. Areas labelled with the same letter are conceptually the same box.

(m, r)-*light salesman path* to be a salesman path that crosses each side of each box at most r times, and crosses the the sides of boxes only at portals.

We then have the following theorem about (m, r)-light salesman paths. We defer the proof for the time being.

Theorem 13.1 [Structure Theorem, Arora '97] Pick $a, b \in [0, L)$ at random. Then with probability at least $\frac{1}{2}$, the (a, b)-dissection has an (m, r)-light salesman path of cost no more than $(1 + \epsilon)OPT$, where $m = O(\frac{1}{\epsilon} \log L)$ and $r = O(\frac{1}{\epsilon})$.

13.1.4 Applying Dynamic Programming

Since we know by the Structure Theorem that there is an (m, r)-light salesman path of cost no more than $(1+\epsilon)OPT$ with probability at least $1/2$, we now apply dynamic programming to find the least cost (m, r)-light salesman path, then convert this into a tour of no greater cost. Then with probability at least $1/2$ we will have found a tour of cost at most $(1 + \epsilon)OPT$.

Now to define the dynamic program. By the definition of an (m, r)-light salesman path, we know for every box that the path crosses each side of it at most r times. Given some Eulerian tour of the salesman path, we can think of the tour entering the box, visiting some of the nodes inside it, then exiting, visiting points elsewhere, re-entering, and so on. Thus for each square, we know that the salesman path

- Uses some even number of portals (possibly repeated), no more than $4r$.
- These portals are paired into entry/exit pairs.

Inside each box, given the portals used and the entry/exit pairing, the cheapest (m, r)-light salesman path uses cheapest paths in the box to visit all nodes in that box.

Thus our dynamic program is as follows: for each box in the (a, b)-quadtree, for each choice of up to r portals per side, for each pairing of these portals into entry/exit pairs, we find the cheapest way to visit all the nodes in the box. We build up a table containing an entry for each box, choice of portals, and pairing of portals, and use some of the entries to help us find the solutions for other entries.

First we calculate the size of the table we will need. There are $O(n \log n)$ boxes in the quadree, $(4m + 4 + 1)^{4r}$ choices of up to r portals on each side of the box (including not choosing any portals) for each box, and $(4r)!$ possible pairings of the portals into entry/exit pairs. So the table has

$$O(n \log n) \times (4m + 4 + 1)^{4r} \times (4r)! = O(n \log^{O(\frac{1}{\epsilon})} n) \text{ entries.}$$

Now we discuss how to build up the table. Our base case considers the boxes at the "lowest" level of the quadtree (i.e. boxes that do not contain any boxes inside of them). This case is fairly simple: for each choice of portals and pairing of portals, we find the shortest paths that enter/exit the box in the designated way and visit the 1 node inside the box. The inductive case builds up a solution for a box B from the solutions for the four boxes $b_1, \ldots, b_4$ it contains. Note that the paths visiting the nodes in B may use portals on the four boundaries of the b_i that are not also a boundary of B. Call these boundaries the "internal" sides of the b_i. To combine the solutions from the b_i to get a solution for box B, we enumerate over all the portals on the internal sides that the paths might have used, and the order in which these portals were visited by the paths, and pick the best solution we find. Notice that specifying a set of portals used on the internal sides and an order in which they are used implies for each b_i a set of portals which are used, in addition to an entry/exit pairing on the portals. Thus we can simply look up the best solution for this subproblem for each b_i from our table and combine them to get a solution for B. We can pick up to r portals from each of the four internal sides (which is no more than $(m + 5)^{4r}$ possibilities) specify which of the $2r$ paths for B on which these portals lie (which is no more than $(2r)^{4r}$ possibilities), and the ordering of these portals on the paths (no more than $(4r)!$ possibilities). Thus it takes at most $O((m+5)^{4r}(2r)^{4r}(4r)!) = O(\log^{O(\frac{1}{\epsilon})} n)$ time to compute the answer for each entry in the table. Therefore, the overall time taken is $O(n \log^{O(\frac{1}{\epsilon})} n)$.

The overall solution to our problem is found in the table entry corresponding to the bounding box with no portals chosen.

13.1.5 Proving the Structure Theorem

We now turn to the proof of Theorem 13.1. The basic idea is to modify the optimal tour to be (m, r)-light salesman path. We will show that modifications don't cost much. To prove this, we first need to show that if the tour crosses a line too much, we can change it at a small increase in cost so that it doesn't cross the line very much.

Lemma 13.2 [Patching Lemma] Given a line segment S of length ℓ, if a tour crosses S three or more times, we can add line segments on S of length no more than $6l$ to get Eulerian tour that contains the previous tour, and crosses S at most twice.

Proof: We take the tour, and break it at the points at which it crosses line S. We put new points just to the "left" and "right" of S where the tour crossed S; see Figure 13.4.

We now add a tour and a matching to the points on the left side of the line, and a tour and matching to the points on the right side of the line, plus edges connecting the last one or two pairs of points. See Figure 13.5 for an example. This gives an Eulerian graph which contains the previous tour and crosses S at most twice. Each tour added has cost at most $2l$ and each matching has cost l, for an overall cost of $6l$.

□

We now apply the patching lemma to the various "grid lines" in the dissection (i.e. the lines parallel with the sides of the bounding box) in order to make the tour into an (m, r)-light salesman path. We need to do this in a particular order, which we give in the following lemma.

Lemma 13.3 Given line ℓ from random (a, b)-dissection, the expected cost of making tour (m, r)-light on line ℓ is $\frac{13t(\ell)}{r}$, where $t(\ell)$ is number of times that the optimal tour crosses line ℓ.

Proof: Define the *level* of line ℓ to be the minimum level over all boxes such that ℓ contains a side of the box. Then observe that since we chose the center of the dissection uniformly at random that this implies that

$$\Pr[\text{level of } \ell \text{ is } i] = \frac{2^i}{L},$$

that is, the probability it has level 0 is $1/L$, etc.

The first thing we need to do is to move all the points at which the optimal tour crosses ℓ to a portal. A box at level i has portals at distance $\frac{L}{2^i m}$, so that the expected increase in the tour for moving the crossings to the nearest portal is

$$\begin{aligned}
\sum_{i \geq i} \Pr[\text{level of } \ell \text{ is } i] \cdot t(\ell) \cdot \frac{L}{2^i m} &= \sum_{i \geq 1} \frac{2^i}{L} \cdot t(\ell) \cdot \frac{L}{2^i m} \\
&= \frac{t(\ell) \log L}{m} \\
&\leq \frac{t(\ell)}{r-1},
\end{aligned}$$

for $m = O(r \log L)$.

We now invoke the patching lemma to make sure that for every side of every box which ℓ contains, it is not crossed more than r times. We consider a vertical line ℓ (the horizontal case is similar), and apply the following procedure to modify ℓ, given that it has level i and we have an (a, b)-dissection:

Modify(ℓ, i, b)

> For $j \leftarrow \log L$ downto i
>
> For $p \leftarrow 0$ to $2^j - 1$
>
> If segment of ℓ from $b + p \cdot \frac{L}{2^j} \text{mod } L$ to $b + (p+1) \cdot \frac{L}{2^j} \text{mod } L$ is crossed more than r times
>
> Apply patching lemma to cross segment twice

Let c_j denote the number of times the patching lemma is applied in iteration j. Then $\sum_{j \geq 1} c_j \leq t(\ell)/(r-1)$, since each time the patching lemma is invoked, it replaces at least $r+1$ crossings with at most 2. Then the total expected increase in the cost of the tour due to the patching of ℓ is

$$
\begin{aligned}
&\sum_{i \geq 1} \Pr[\text{level of } \ell \text{ is } i] \cdot \text{Increase in cost due to } \mathbf{Modify}(\ell, i, b) \\
&\leq \sum_{i \geq 1} \frac{2^i}{L} \sum_{j \geq i} c_j \cdot 6 \cdot \frac{L}{2^j} \\
&= 6 \sum_{j \geq 1} \frac{c_j}{2^j} \sum_{i \leq j} 2^i \\
&= 6 \sum_{j \geq 1} 2c_j \\
&= \frac{12t(\ell)}{r-1}.
\end{aligned}
$$

Thus the total expected increase in cost for making the tour (m, r)-light on line ℓ is

$$\frac{t(\ell)}{r-1} + \frac{12t(\ell)}{r-1} \leq \frac{13t(\ell)}{r}.$$

□

Let T be the sum of $t(\ell)$ over all horizontal and vertical grid lines ℓ. The above lemma tells us that the overall expected increase caused by transforming the optimal tour to an (m, r)-light salesman path is at most $\frac{13}{r}T$. If we set $r \geq \frac{52}{\epsilon}$, then with probability at least $1/2$, the cost is at most $\frac{\epsilon}{2}T$. The final lemma shows that $T \leq 2OPT$, which completes the proof of the Structure Theorem.

Lemma 13.4 *$T \leq 2OPT$.*

Proof: Consider an edge e in the optimal tour from point (x_1, y_1) to (x_2, y_2). The edge contributes at most $|x_1 - x_2| + |y_1 - y_2| + 2$ to T, and has length $s = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. But

$$
\begin{aligned}
|x_1 - x_2| + |y_1 - y_2| + 2 &\leq \sqrt{2[(x_1 - x_2)^2 + (y_1 - y_2)^2]} + 2 \\
&\leq \sqrt{2s^2} + 2 \\
&\leq 2s,
\end{aligned}
$$

since (by our perturbation of the instance) $s \geq 4$. Thus, summing over all edges in the optimal tour, $T \leq 2OPT$. $\square$

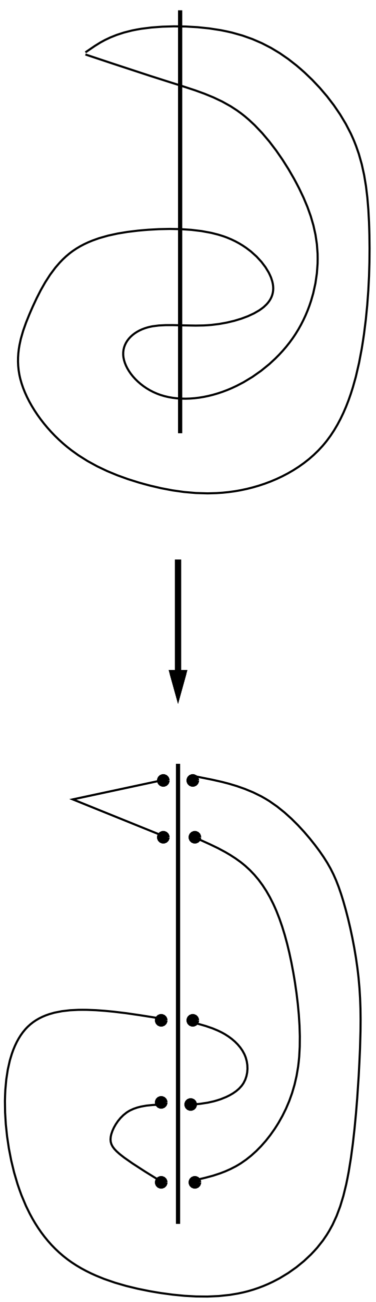

Figure 13.4: Breaking the tour where it crosses the line.

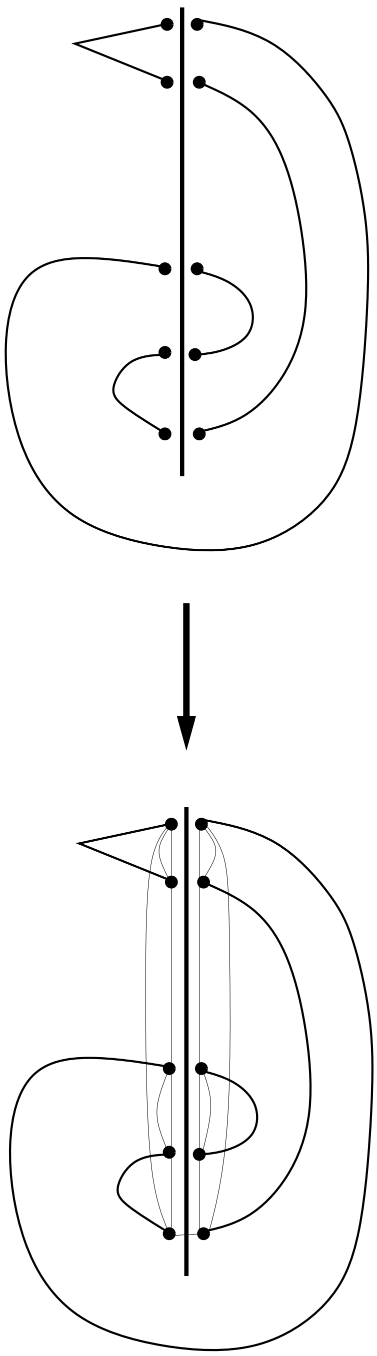

Figure 13.5: Adding tours and matchings to obtain Eulerian tour.

IEOR E6610 Approximation Algorithms	April 29, 1998
Lecture 14	
Lecturer: David P. Williamson	*Scribe: Xiangdong Yu*

14.1 Uncapacitated Facility Location

For our last lecture, we will consider the uncapacitated facility location problem.

Uncapacitated Facility Location Problem (UFL)

- **Input:**
  - Set V of locations.
  - Facilities $F \subset V$, cost f_i, $\forall i \in F$.
  - Clients $D = V - F$, cost c_{ij} for assigning client j to facility i.
  - Costs c_{ij} obey the trangle inequality.
- **Goal:** Find $F' \subseteq F$ and assignment of clients to facilities in F' that minimizes the total cost.

We will use our standard technique, and first formulate the problem as an integer program. To do that, we first introduce some variables. Let

$$y_i = \begin{cases} 1 & \text{if facility } i \text{ is opened} \\ 0 & \text{otherwise} \end{cases}$$

and

$$x_{ij} = \begin{cases} 1 & \text{if client } j \text{ is assigned to facility } i \\ 0 & \text{otherwise.} \end{cases}$$

We can then get the following integer programming formulation for the UFL:

$$\begin{array}{lll} \text{Min} & \sum_{i \in F} f_i y_i + \sum_{i \in F, j \in D} c_{ij} x_{ij} & \\ \text{subject to:} & & \\ & \sum_{i \in F} x_{ij} = 1 & \forall j \in D \\ & x_{ij} \leq y_i & \forall i \in F, j \in D \\ & x_{ij} \in \{0,1\} & \forall i \in F, j \in D \\ & y_i \in \{0,1\} & \forall i \in F. \end{array}$$

The first set of equalities enforce that each client is assigned to some facility. The inequalities $x_{ij} \leq y_i$ enforce that if client j is assigned to facility i, then facility i should be open. We relax this IP to an LP by replacing $x_{ij} \in \{0,1\}$ with $x_{ij} \geq 0$ and $y_i \in \{0,1\}$ with $y_i \geq 0$.

In order to get an approximation algorithm for the problem, we first need the following definition.

Definition 14.1 Let $g \in \Re^{|D|}$, $(\tilde{x}, \tilde{y})$ be a feasible solution for the LP. Then $(\tilde{x}, \tilde{y})$ is *g-close* if $\tilde{x}_{ij} > 0$ implies that $c_{ij} \leq g_j$, $\forall i \in F, j \in D$.

Basically the definition says that for any client j, for any facility i that has a non-zero assignment variable x_{ij}, the facility should be "close"; i.e. within cost g_j.

To get a vector g such that an LP solution is g-close, we look at the dual of our LP. The dual is

$$
\begin{aligned}
\text{Max} \quad & \sum_{j \in D} v_j \\
\text{subject to:} \quad & \\
& \sum_{j \in D} w_{ij} \leq f_i && \forall i \in F \\
& v_j - w_{ij} \leq c_{ij} && \forall i \in F, j \in D \\
& w_{ij} \geq 0 && \forall i \in F, j \in D.
\end{aligned}
$$

The following lemma relates the optimal LP and dual solutions via closeness.

Lemma 14.1 If (x^*, y^*) is an optimal solution to LP and (v^*, w^*) is an optimal solution to its dual, then (x^*, y^*) is v^*-close.

Proof: We observe that

$$
\begin{aligned}
& x^*_{ij} > 0 \;\Rightarrow\; v^*_j - w^*_{ij} = c_{ij} && (14.1) \\
& \qquad\;\;\; \Rightarrow\; c_{ij} \leq v^*_j. && (14.2)
\end{aligned}
$$

(14.1) follows via complementary slackness, and (14.2) follows since $w_{ij} \geq 0$. $\square$

Shmoys, Tardos, and Aardal proved the following useful theorem. We defer the proof of the theorem until after we show how it implies an approximation algorithm for UFL.

Theorem 14.2 (Shmoys, Tardos, Aardal '97) Given a LP solution $(\tilde{x}, \tilde{y})$ that is g-close, one can find in polynomial time a feasible IP solution $(\bar{x}, \bar{y})$ which is $3g$-close such that $\sum_{i \in F} f_i \bar{y}_i \leq \sum_{i \in F} f_i \tilde{y}_i$.

Theorem 14.3 (Chudak '98) Applying the theorem above to an optimal LP solution (x^*, y^*) with $g = v^*$, where v^* is an optimal dual solution, gives a 4-approximation algorithm for UFL.

Proof: The cost of the integral solution produced by the algorithm is $\sum_{i\in F} f_i\bar{y}_i + \sum_{i\in F, j\in D} c_{ij}\bar{x}_{ij}$. We see then that

$$\begin{aligned}
& \sum_{i\in F} f_i\bar{y}_i + \sum_{i\in F, j\in D} c_{ij}\bar{x}_{ij} \\
& \quad \leq \sum_{i\in F} f_i y_i^* + \sum_{j\in D}(3v_j^*) && (14.3)\\
& \quad \leq Z_{LP} + 3Z_D && (14.4)\\
& \quad = 4Z_{LP} && (14.5)\\
& \quad \leq 4OPT.
\end{aligned}$$

The inequality (14.3) follows for several reasons. First, we know that $\sum_{i\in F} f_i\bar{y}_i \leq \sum_{i\in F} f_i y_i^*$. Second, we know that $\sum_{i\in F}\bar{x}_{ij} = 1$ for any j, so that there must be exactly one $i \in F$ such that $\bar{x}_{ij} = 1$ for any j. Third, we know that $(\bar{x}, \bar{y})$ is v^*-close, which means that when $\bar{x}_{ij} = 1$ then $c_{ij} \leq 3v_j^*$. Thus this implies that $\sum_{i\in F, j\in D} c_{ij}\bar{x}_{ij} \leq \sum_{j\in D}(3v_j^*)$.

Inequality (14.4) follows since $\sum_{i\in F} f_i y_i^*$ is less than the LP objection function, and $\sum_{j\in D} v_j^*$ is the dual objective function, and inequality (14.5) follows since at optimality $Z_{LP} = Z_D$. □

We now turn to the proof of Theorem 14.2.

Proof of Theorem 14.2: Given a feasible LP solution $(\tilde{x}, \tilde{y})$ which is g-close, we obtain the IP solution via the following algorithm:

UFL-Round

> $S \leftarrow D$
> while $S \neq \emptyset$
>   Choose client $k \in S$ with smallest g_k.
>   Let $N(k)$ be all facilities i such that $\tilde{x}_{ik} > 0$.
>   Choose facility $l \in N(k)$ of cheapest cost, open l.
>   Assign client k to facility l, $S \leftarrow S - k$.
>   For any $a \in S$ such that $\tilde{x}_{ba} > 0$ and $b \in N(k)$,
>   Assign a to l, $S \leftarrow S - a$.

We will say that a client j *neighbors* a facility i (or vice versa) if $x_{ij} > 0$. Thus, for instance, $N(k)$ is the set of all facilities that neighbor client k. Note that the $N(k)$'s in different iterations are disjoint: this follows since any unassigned client which neighbors a facility in $N(k)$ is assigned in that iteration and removed from S. Thus we cannot pick some client k' in a later iteration that neighbors some facility in $N(k)$.

We now bound the cost of the integer programming solution created. Fix an iteration in which $k \in S$ is chosen. Then for the facility l opened in that iteration, we have

$$
\begin{aligned}
f_l &= f_l \sum_{i \in N(k)} \tilde{x}_{ik} && (14.6)\\
&\leq \sum_{i \in N(k)} f_i \tilde{x}_{ik} && (14.7)\\
&\leq \sum_{i \in N(k)} f_i \tilde{y}_i. && (14.8)
\end{aligned}
$$

Equality (14.6) follows since from the LP solution we know that $\sum_{i \in N(k)} \tilde{x}_{ik} = 1$. Inequality (14.7) follows since we chose l to minimize f_l over all $i \in N(k)$. Inequality (14.8) follows since from the LP we know that $\tilde{x}_{ik} \leq \tilde{y}_i$. Then by the disjointness of the $N(k)$ we have overall that

$$\sum_{i \in F} f_i \bar{y}_i \leq \sum_{i \in F} f_i \tilde{y}_i,$$

as promised.

We now turn to the assignment costs. Again, fix an iteration in which client $k \in S$ is chosen, and facility $l \in N(k)$ is opened. The cost of assigning k to l is c_{lk}, which by hypothesis is no more than g_k, since k neighbors l and $(\tilde{x}, \tilde{y})$ is g-close. Now consider a client $a \in S$ assigned to l since it neighbors a facility $b \in N(k)$. By the triangle inequality, and by closeness we have that

$$c_{la} \leq c_{lk} + c_{bk} + c_{ba} \leq g_k + g_k + g_a.$$

However, since we chose k to minimize g_k among all $k \in S$, this implies that $c_{la} \leq 3g_a$. Thus this proves that the integer assignment $\bar{x}$ is $3g$-close. $\square$

Further progress exists for this problem. In particular, it is not too much more difficult to prove that if we modify the algorithm above to pick k from S to minimize $g_k + \sum_{i \in F} x_{ik}$ and then choose the facility l from $N(k)$ randomly according to the probability distribution $\tilde{x}_{ik}$, we obtain a 3-approximation algorithm. Currently the best result known is due to Chudak (1998), who gives a $(1 + \frac{2}{e})$-approximation algorithm for UFL. There is also a lower bound for the problem.

Theorem 14.4 (Guha, Khuller '97) There is no 1.427-approximation algorithm for UFL unless P=NP.