RC 21409 (02/17/99)
Computer Science/Mathematics

IBM Research Report

Lecture Notes on Approximation Algorithms
Fall 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

Fall 1998

Contents

Preface

The contents of this book are lecture notes from a class taught in the School of Operations Research and Industrial Engineering of Cornell University during the Fall 1998 term (ORIE 634:Combinatorial Optimization – 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 scribe notes were frequently created from a previous version of the notes written in a previous version of the course (IEOR 6610E, Spring 1998, Columbia University). The students in the class who served as scribes were Christina Ahrens, Aaron Archer, Tuğkan Batu, dan brown, Nathan Edwards, Tim Huh, Rif "Andrew" Hutchings, Amit Kumar, Vardges Melkonian, Kathryn Nyman, and Tim Roughgarden. Any errors which remain (or were there to begin with!) are, of course, entirely my responsibility.

David P. Williamson
Yorktown Heights, NY

ORIE 634 Approximation Algorithms		August 27, 1998
	Lecture 1	
Lecturer: David P. Williamson		*Scribe: Woonghee Tim Huh*

1.1 An Introduction to Approximation Algorithms

The first problem we will consider today is the well-known 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! It runs too slowly because the running time is $O((n-1)!)$ where $n = |V|$.
- *Dynamic Programming* runs in $O(2^n)$, which is still slow.

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, Karp and Levin 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, branch-and-cut and branch-and-price methods). Using these methods, a non-trivial instance of the TSP with over 13,000 cities has been solved optimally.
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"). 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)* for a minimization problem 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. For a maximization problem, we require that A_ϵ is a $(1-\epsilon)$-approximation algorithm.

Some problems which have a *PTAS* are knapsack, Euclidean TSP (Arora 1996, Mitchell 1996), and some scheduling problems. Other problems like *MAX SAT*, *MAX CUT* and *Metric TSP* 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 *MAX 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 $NP \subseteq RP$.

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 $(i, j) \in E$).

Note that there is a trivial approximation algorithm for *MAX CLIQUE* with performance guarantee $n = |V|$. Simply take a single vertex; this is trivially a clique. The size of any clique cannot be more than n, so the algorithm has a performance guarantee of $n/1 = n$. Håstad's result tells us that we do not expect to do much better than this trivial algorithm.

Therefore, the theory of approximation algorithms shows that some NP problems are harder than others.

The goal of this class is to understand both the basic algorithms known in the area, and some more recent techniques that lead to approximation algorithms for large classes of problems.

1.3 Set Cover

1.3.1 What is the Set Cover Problem?

Consider the following problem:

Weighted Set Cover (SC)

- **Input:**
  - Ground elements $T = \{t_1, t_2, \cdots, t_n\}$
  - Subsets $S_1, S_2, \cdots, S_m \subseteq T$
  - Weights $w_1, w_2, \cdots, w_m$
- **Goal:** Find a set $I \subseteq \{1, 2, \cdots, m\}$ that minimizes $\sum_{i \in I} w_i$, such that $\bigcup_{i \in I} S_i = T$.

For the **unweighted SC** problem, we take $w_j = 1$ for all j.

Why should we care about the Set Cover Problem? First, the problem shows up in various applications. A colleague of Dr. Williamson at IBM applied the set cover problem to find computer viruses.

- *Elements*: Known viruses (about 5000 of them).
- *Sets*: Substrings of 20 or more consecutive bytes from viruses, not found in "good" code (about 9000 of them).

A set cover of size about 180 was found. It suffices to search for these 180 substrings to verify the existence of known computer viruses.

A second reason to care about the Set Cover Problem is that it is a generalization of other problems we care about. Consider the following problem:

Weighted Vertex Cover (VC)

- **Input:**
  - An undirected graph $G = (V, E)$
  - Weights $w_i \geq 0\ \forall i \in V$
- **Goal:** Find a set C that minimizes $\sum_{i \in C} w_i$, such that for every $(i, j) \in E$, we have either $i \in C$ or $j \in C$.

To see that VC is a special case of SC, consider the following substitution:

- *Elements*: all edges in E.
- *Sets*: $S_i = \{\text{all edges incident with vertex } i\}$.

1.3.2 First Attempt: a Greedy Algorithm

There are quite a few different ways to approach the set cover problem. Here's our first attempt.

Greedy 1

```
I ← ∅
while T ≠ ∅
      (*) Pick t_i ∈ T
      I ← I ∪ {j : t_i ∈ S_j}
      T ← T − ∪_{j∈I} S_j.
```

We let $f = \max_i |\{j : t_i \in S_j\}|$. For the VC problem, we see that $f = 2$ (every edge is in two sets, the ones corresponding to its two endpoints).

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

Lemma 1.3 *Greedy 1* returns a set cover.

Proof: Elements are only deleted when they are covered, we delete at least one each time through the while loop, and at termination, $T = \emptyset$. $\square$

Theorem 1.4 *Greedy 1* is an f-approximation algorithm for the unweighted *SC problem.*

Proof: It is clear that it runs in polynomial time. Now, suppose the algorithm goes through the while loop X times. We claim that each t_i chosen in $(*)$ *must* be covered by a distinct set in the optimal solution. This implies that $X \leq OPT$. To see the claim, suppose the opposite, and there is a t_a chosen in one iteration and a t_b chosen in a later iteration which are both in the same set S_j in the optimal solution. However, by construction, when we pick t_a, we choose all sets that contain t_a, including S_j, and remove all the elements from T that are contained in these sets. In particular, t_b is removed from T and is not available to be picked in a later iteration. Thus we have a contradiction.

Each time we pick t_i in step $(*)$, we add $|\{j : t_i \in S_j\}| \leq f$ sets, so that $I \leq f \cdot X \leq f \cdot OPT$. Thus we have an f-approximation algorithm. $\square$

This algorithm, however, will not be very useful for the weighted problem, so we use the following method.

1.3.3 A General Approach for Approximation Algorithms

We will use the following method over and over again in this course. In fact, we will see four different ways it can be used for the Set Cover Problem.

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.

Let's apply it to the weighted *SC* problem. Here is the IP formulation

$$\begin{aligned} &\text{Min} \quad \sum_{j=1}^{m} w_j \cdot x_j \\ &\text{subject to:} \\ &\qquad \sum_{j: t_i \in S_j} x_j \geq 1 && \forall t_i \in T \\ &\qquad x_j \in \{0,1\} && \forall i \in \{1, 2, \cdots, m\}, \end{aligned}$$

and the corresponding LP relaxation

$$\begin{aligned} &\text{Min} \quad \sum_{j=1}^{m} w_j \cdot x_j \\ &\text{subject to:} \\ &\qquad \sum_{j: t_i \in S_j} x_j \geq 1 && \forall t_i \in T \\ &\qquad x_j \geq 0 && \forall i \in \{1, 2, \cdots, m\}. \end{aligned}$$

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.

Now if we can find an integral solution of cost no more than $\alpha \cdot Z_{LP}$, then its cost is at most $\alpha \cdot OPT$. We have four different ways of completing Step 3, which will be presented in the next class.

ORIE 634 Approximation Algorithms	September 3, 1998
Lecture 2	
Lecturer: David P. Williamson	*Scribe: Aaron Archer*

2.1 Four More Ways to Skin a Cat: Approximation Algorithms for Set Cover

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

Set Cover

- **Input:**
  - Ground set $T = \{t_1, t_2, \ldots, t_n\}$
  - Subsets $S_1, S_2, \ldots, S_m \subseteq T$
  - Weights (costs) $w_j \geq 0$ for each subset S_j
- **Goal:** Find $I \subseteq \{1, \ldots, m\}$ that minimizes $\sum_{j \in I} w_j$ subject to $\bigcup_{j \in I} S_j = T$. That is, select the minimum-weight collection of sets that covers all of T.

Recall the general 3-step approach introduced in Lecture 1 for constructing approximation algorithms:

1. Formulate the problem as an integer program (IP).
2. Relax the integer requirement to obtain a linear program (LP).
3. Use the LP solutions in some clever way to obtain in polynomial time a feasible solution to the IP whose value is close to the optimal LP value.

Since Step 2 can be performed in polynomial time and (for a minimization problem) the optimal value z_{LP} of the LP is a lower bound on the optimal value OPT of the IP, constructing an integer solution of value $\leq \alpha z_{LP} \leq \alpha \cdot OPT$ yields an α-approximation algorithm. The tricky part, of course, is Step 3. Here we perform Steps 1 and 2, and in the next 4 subsections we illustrate 4 different ways of performing Step 3. The first three yield f-approximation algorithms where $f = \max_i |\{j : t_i \in S_j\}|$, and the last yields an H_g-approximation algorithm where $g = \max_j |S_j|$ and $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \approx \ln n$.

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

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

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 that $z_{LP} \leq OPT$ because the solution space for the integer program is a subset of the solution space of the linear program.

2.1.1 Method I: Rounding

Let us define f by

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

That is, f is the maximum number of sets that contain any given element. Now consider the following algorithm:

Rounding

> Solve the linear program to get an optimal solution x^*.
> $I \leftarrow \emptyset$
> for each S_j
>   if $x_j^* \geq 1/f$
> $I \leftarrow I \cup \{j\}$

Lemma 2.1 *Rounding produces a set cover.*

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

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

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

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

Proof: It is clear that the rounding algorithm is a polynomial-time algorithm. Furthermore,

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

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

Note that this yields a 2-approximation algorithm 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 $C \subseteq V$ that minimizes $\sum_{i \in C} w_i$ such that each edge is incident to some vertex in C.

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:

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

If we have a feasible dual solution y, then

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

by weak duality. An algorithm for finding a low-cost set cover using the dual LP follows:

Dual-LP

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

Lemma 2.3 Dual-LP produces a set cover.

Proof: Suppose $\exists t_i \notin \bigcup_{j \in I} S_j$. Then for each S_j containing t_i

$$\sum_{i: t_i \in S_j} y_i^* < w_j,$$

so we can increase y_i^* by some positive amount and remain feasible, which contradicts the optimality of y^*. □

Theorem 2.4 (Hochbaum '82) Dual-LP is an f-approximation algorithm.

Proof: Because we choose set S_j only if its constraint is tight, we have

$$\begin{aligned}\sum_{j \in I} w_j &= \sum_{j \in I} \sum_{i: t_i \in S_j} y_i^* \\ &= \sum_i y_i^* |\{j \in I : t_i \in S_j\}| \\ &\leq f \sum_i y_i^* \\ &\leq f \cdot OPT.\end{aligned}$$

□

Michael Wagner observed in class that complementary slackness guarantees that whenever the Rounding algorithm includes a set S_j (because $x_j^* > \frac{1}{f}$) the corresponding dual constraint is tight so Dual-LP also includes the set S_j in its solution. Thus, Dual-LP never obtains a better solution than Rounding.

2.1.3 Method III: Primal-Dual

One 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 $\exists t_k : t_k \notin \bigcup_{j \in I} S_j$
  $l = \arg\min_{j: t_k \in S_j} \left\{ w_j - \sum_{i: t_i \in S_j} \tilde{y}_i \right\}$
  $\epsilon_l \leftarrow w_l - \sum_{i: t_i \in S_l} \tilde{y}_i$
  $\tilde{y}_k \leftarrow \tilde{y}_k + \epsilon_l$
  $I \leftarrow I \cup \{l\}$.

Note that the function arg min returns the argument (index, in this case) that minimizes the expression.

Lemma 2.5 Primal-Dual returns a set cover.

Proof: Trivial, since this is the termination condition for the while loop. □

Lemma 2.6 Primal-Dual constructs a dual feasible solution.

Proof: We proceed by induction on the loops of the algorithm. The base case is trivial since initially

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

For the inductive step, assume that upon entering an iteration of the while loop we have

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

The only dual variable value changed by the while loop is $\tilde{y}_k$, so the inequalities for S_j where $t_k \notin S_j$ are unaffected. If $t_k \in S_j$, then by our choice of l

$$\begin{aligned} \sum_{i: t_i \in S_j} \tilde{y}_i + \epsilon_l &= \sum_{i: t_i \in S_j} \tilde{y}_i + (w_l - \sum_{i: t_i \in S_l} \tilde{y}_i) \\ &\leq \sum_{i: t_i \in S_j} \tilde{y}_i + (w_j - \sum_{i: t_i \in S_j} \tilde{y}_i) \\ &\leq w_j. \end{aligned}$$

□

Lemma 2.7 If $j \in I$ then $\sum_{i: t_i \in S_j} \tilde{y}_i = w_j$.

Proof: In the step where j was added to I, we increased $\tilde{y}_k$ by exactly enough to make the constraint S_j tight. □

Theorem 2.8 (Bar-Yehuda, Even '81) Primal-Dual is an f-approximation algorithm for the set cover problem.

Proof:

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

The first equality follows from Lemma 2.7. The next inequality follows since each $\tilde{y}_i$ can appear in the double sum at most $|\{j : t_i \in S_j\}|$ times. The next inequality follows by the definition of f, and the last inequality follows from weak duality. □

2.1.4 Method IV: Greedy Algorithm

So far every technique we have tried has led to the same result: an f-approximation algorithm for set cover. It seems that we get the same performance guarantee no matter what we do! But in general cleverness will often give us improved performance guarantees. In the case of set cover, using a natural greedy heuristic yields an improved algorithm.

The intuition here is straightforward. At each step choose the set that gives the most "bang for the buck." That is, select the set that minimizes the cost per additional element covered. Before examining the algorithm, we define two more quantities:

$$H_n \equiv 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \approx \ln n$$
$$g \equiv \max_j |S_j|.$$

Greedy

$I \leftarrow \emptyset$
$\tilde{S}_j \leftarrow S_j \ \ \forall j$
while $\bigcup_{j \in I} S_j \neq T$
  $l \leftarrow \arg\min_{j: \tilde{S}_j \neq \emptyset} \frac{w_j}{|\tilde{S}_j|}$
  $I \leftarrow I \cup \{l\}$
  $\tilde{y}_i \leftarrow \frac{w_l}{|\tilde{S}_l| H_g} \ \ \forall t_i \in \tilde{S}_l$ (†)
  $\tilde{S}_j \leftarrow \tilde{S}_j - S_l \ \ \forall j$.

Note: The † step has been added only to aid the proof and is not actually part of the algorithm.

Lemma 2.9 Greedy constructs a feasible dual solution $\tilde{y}$.

Proof: First, note (from the † step in the algorithm) that at the time we choose l,

$$w_l = H_g \sum_{i \in \tilde{S}_l} \tilde{y}_i.$$

Now, pick an arbitrary set S_j. For convenience we reindex the elements of T such that $S_j = \{t_1, \ldots, t_k\}$ and Greedy covers this set in index order. Thus when t_i is covered, $|\tilde{S}_j| \geq k - i + 1$. Let l be the index of the first set chosen that covers t_i. It follows that

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

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

$$\frac{w_l}{|\tilde{S}_l|} \leq \frac{w_j}{|\tilde{S}_j|}.$$

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

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

since $k = |S_j| \leq g$. □

Theorem 2.10 (Chvatal '79) Greedy is a H_g-approximation algorithm.

Proof: Reindex the sets so that Greedy chooses S_r in the r^{th} iteration. Let $\hat{S}_r$ be the contents of $\tilde{S}_r$ when S_r is chosen. Then the $\hat{S}_r$ partition T, and we know

$$w_r = H_g \sum_{i \in \hat{S}_r} \tilde{y}_i.$$

Say the algorithm takes q iterations. Then

$$\begin{aligned} \sum_{j \in I} w_j &= \sum_{r=1}^{q} w_r \\ &= \sum_{r=1}^{q} H_g \sum_{i \in \hat{S}_r} \tilde{y}_i \\ &= H_g \sum_{i \in T} \tilde{y}_i \\ &\leq H_g \cdot OPT. \end{aligned}$$

The last equality follows because the $\hat{S}_r$ partition T. □

The following complexity results give a sense of how good our approximation algorithms are in relation to what is possible.

Theorem 2.11 (Lund, Yannkakis '92, Feige '96, Raz, Safra '97, Arora, Sudan '97) Let $n = |T|$ = the size of the ground set. Then:

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

Theorem 2.12 (Håstad '97) If there exists an α-approximation algorithm for vertex cover with $\alpha < \frac{7}{6}$ then $P = NP$.

At present no α-approximation algorithm with $\alpha < 2$ is known for vertex cover.

2.2 Set Cover: An Application

Recall the antivirus application mentioned in Section 1.3.1. 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.

ORIE 634 Approximation Algorithms	September 10, 1998
Lecture 3	
Lecturer: David P. Williamson	*Scribe: Tuğkan Batu*

3.1 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 $\{1, \ldots, n\}$. Item i has a value v_i and size s_i. Total "capacity" is B. $v_i, s_i, B \in \mathbf{Z}_+$.
- **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 3.1 Let $A(i, v) \equiv$ size of "smallest" subset of $\{1, \ldots, i\}$ 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 $i \leftarrow 1$ to n
  $A(i, 0) \leftarrow 0$
For $v \leftarrow 1$ to nV
  $A(1, v) \leftarrow \begin{cases} s_1 & \text{if } v_1 = v \\ \infty & \text{otherwise} \end{cases}$
For $i \leftarrow 2$ to n
  For $v \leftarrow 1$ to nV
    if $v_i \leq v$
      $A(i, v) \leftarrow \min(A(i-1, v), s_i + A(i_1, v - v_i))$
    else
      $A(i, v) \leftarrow A(i-1, v)$.

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 $O(n^2 V)$.

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 3.2 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 3.3 A *polynomial-time approximation scheme (PTAS)* is a family of algorithms $\{A_\epsilon\}$ 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 $\{A_\epsilon\}$ is a fully polynomial-time 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 3.1 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',

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

which implies

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

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

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

3.2 Scheduling Identical Machines

3.2.1 List Scheduling

We now turn to a scheduling problem. 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 $J_1, \dots, J_n$ to be scheduled on the machines
  - $p_1, \dots, p_n$ the processing times of the jobs
- **Goal:** Find a schedule of jobs that minimizes the completion time of the last job, i.e., partition $\{1, \dots, n\}$ into $M_1, \dots, M_m$ to minimize $\max_{i \in \{1,\dots,m\}} \sum_{j \in M_i} p_j$.

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

List-scheduling

For $i \leftarrow 1$ to n
    Schedule job i on the machine that has the least work assigned to it so far.

Theorem 3.2 (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*, $\frac{1}{m}\sum_{1 \leq j \leq n} p_j$, is a lower bound: by the pigeonhole principle, some machine must have at least this amount of processing time assigned to it. 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. By the pigeonhole principle, some machine must be finished with its processing in time no more than the average load, so that J_k must start no later than $\frac{1}{m}\sum_{1 \leq j \leq n} p_j \leq OPT$. Then J_k must finish no later than $\frac{1}{m}\sum_{1 \leq j \leq n} p_j + p_k \leq OPT + OPT = 2OPT$. □

3.2.2 A PTAS Using Dynamic Programming

We will now give a PTAS for the problem of scheduling identical machines. We would like to use the same set of ideas that were used for the knapsack problem: that is, given an explicit time T we would like to round the job lengths and use dynamic programming to see if they will fit within time T. Then the unrounded job lengths should fit within time $T(1+\epsilon)$.

We now must show that such an agenda will lead to a PTAS overall. To do this, we define a $(1+\epsilon)$-relaxed decision procedure.

Definition 3.4 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 $L \equiv \max(\max_j p_j, \frac{1}{m}\sum_{j=1}^{n} p_j)$. We know that $OPT \in (L-1, 2L]$. Our algorithm will perform *binary search* on $(L-1, 2L]$ 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.

To implement the decision procedure, we would like to round down the job sizes to powers of $(1+\epsilon)$. That way, if we can use dynamic programming to help us decide whether the rounded job sizes will fit in a schedule of length T, when we take the same schedule and revert to the original sizes the schedule will have length no more than $(1+\epsilon)T$.

However, to make this work, we will need to deal with smaller jobs differently than larger jobs; we also need this to avoid dealing with jobs whose size has been rounded down to zero! To do this, we get our necessary decision procedure by reducing it to yet another decision procedure.

DecisionProcedure1

```
        Split jobs into small jobs (p_j ≤ εT) and large jobs (p_j > εT)
        Call a (1 + ε)-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 + ε)T then return no
(***)       else return yes
```

Lemma 3.3 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. We first show that we can get down to a constant number of job sizes by rounding down to powers of $(1+\epsilon)$. We then hope to apply dynamic programming to tell us whether a schedule of length T exists for the rounded-down jobs.

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 BigJobDynProg on p'_j, T
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 $l = O(\log_{1+\epsilon}\epsilon)$.

Lemma 3.4 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 $(1+\epsilon)T$. □

We are now down to the case in which only $k = l+1$ 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. Since $k = l+1$, which implies that the running time is $O(n^{2(l+1)}) = O(n^{\log_{(1+\epsilon)} 1/\epsilon})$.

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

ORIE 634 Approximation Algorithms	September 17, 1998
Lecture 4	
Lecturer: David P. Williamson	*Scribe: Amit Kumar*

4.1 Bin Packing

We consider the bin packing problem and apply the techniques introduced in previous lectures to get approximation algorithms for this problem.

Bin Packing

- **Input:** Set of items $\{1, \ldots, n\}$. Item i has size s_i. $1 \geq s_1 \geq s_2 \geq \ldots \geq s_n \geq 0$, $s_i \in Q^+$.
- **Goal:** Find a minimum number of bins into which all the items can be packed, where each bin has size 1. (i.e., partition $\{1, 2, \ldots, n\}$ into k sets $B_1, \ldots, B_k$ such that $\sum_{i \in B_j} s_i \leq 1$ and k is minimum.)

4.1.1 A Lower Bound

We show a connection between the bin packing problem and another NP-complete problem:

Partition

- **Input:** Set of items $\{1, \ldots, n\}$. Item i has size s_i. $s_1 \geq s_2 \geq \ldots \geq s_n$, $s_i \in Q^+$.
- **Goal:** Can $\{1, 2, \ldots, n\}$ be partitioned into 2 sets A and B such that $\sum_{i \in A} s_i = \sum_{i \in B} s_i$?

Claim 4.1 The NP-completeness of Partition implies that there doesn't exist a ρ-approximation algorithm for bin packing for $\rho < \frac{3}{2}$ unless P = NP.

Proof: Consider an instance I of partition problem. Scale the size of items such that $\sum_i s_i = 2$ and consider this as an instance I' of bin packing. If all items of I' can fit in two bins, then we have a "yes" answer to I. Otherwise, the items of I' need 3 bins and the answer to I is "no".

OPT for I' is 2 or 3. So, if $\rho < \frac{3}{2}$, then we can determine the value of OPT which implies that we can solve I. Thus, there cannot exist a ρ-approximation algorithm for bin packing for $\rho < \frac{3}{2}$ unless P = NP. $\square$

However, the situation is not so bad. There exist approximation algorithms which guarantee better approximation ratios, but they also entail some small additive factor. We state below some results of this kind.

Theorem 4.2 (Johnson '74) There exists a polynomial-time algorithm FFD such that

$$FFD(I) \leq \frac{11}{9} OPT(I) + 4$$

for all instances I for the bin packing problem.

Today, we shall prove the following theorems.

Theorem 4.3 (Fernandez de la Vega, Lueker '81) There exists a polynomial-time algorithm FL such that

$$FL(I) \leq (1+\epsilon) OPT(I) + 1$$

for all bin packing instances I and any $\epsilon > 0$.

Theorem 4.4 (Karmarkar, Karp '82) There exists a polynomial-time algorithm KK such that

$$KK(I) \leq OPT(I) + O(\log^2(OPT(I)))$$

for all bin packing instances I, where n is the size of instance I.

It is still an open problem whether there exists a polynomial time algorithm A such that

$$A(I) \leq OPT(I) + 1.$$

4.1.2 First Fit Algorithm

This is among the most intuitive algorithms one can think of for the bin packing problem. Consider some ordering on empty bins.

FF

```
For i ← 1 to n
    Let j be the first bin such that i fits in bin j
    Put i in bin j
```

Theorem 4.5 $FF(I) \leq 2OPT(I) + 1$ for all instances I.

Proof: Let $SIZE(I)$ denote $\sum_i s_i$. Then it is easy to see the following lower bound for $OPT(I)$:

$$SIZE(I) \leq OPT(I).$$

We now claim that at most one bin can be half-full in the output of $FF(I)$. This is so because if there were two bins half full, then the last item added to the latter bin should have been added to the first bin. Thus,

$$\frac{1}{2}(FF(I) - 1) \leq SIZE(I)$$

which implies

$$FF(I) \leq 2SIZE(I) + 1$$

i.e.,

$$FF(I) \leq 2OPT(I) + 1.$$

□

4.1.3 A "PTAS" for Bin Packing

We give a "PTAS" for the bin packing problem ("PTAS" in quotes, since we need an additive factor). The ideas involved are similar to those for the PTAS for the scheduling problem we solved in the previous lecture.

Suppose size of items $s_i \leq \frac{\epsilon}{2}$ for all i in an instance I. Using arguments similar to the proof of Theorem 4.5, we can show that all bins except perhaps one must contain items of total size at least $1 - \frac{\epsilon}{2}$. Thus,

$$\left(1 - \frac{\epsilon}{2}\right)(FF(I) - 1) \leq SIZE(I)$$

which implies that (assuming $\epsilon < 1$)

$$\begin{aligned} FF(I) &\leq \frac{1}{1-\frac{\epsilon}{2}} SIZE(I) + 1 \\ &\leq (1+\epsilon) SIZE(I) + 1 \\ &\leq (1+\epsilon) OPT(I) + 1. \end{aligned}$$

We can use this idea to develop the following algorithm for bin packing:

1. Split the items into large (size $> \frac{\epsilon}{2}$) and small (size $\leq \frac{\epsilon}{2}$) pieces.
2. Use some algorithm to pack large pieces into b bins.
3. Use FF to pack the small pieces (possibly opening new bins).

In step 3 above, if FF doesn't open any new bin, then the algorithm uses at most b bins. If FF opens new bins, then by the argument above , we can prove that all bins except perhaps one must contain items of total size at least $1 - \frac{\epsilon}{2}$. Thus by the inequalities above, at most $(1+\epsilon)OPT(I)+1$ bins are used.

So, if we can get a PTAS for bin packing problem when all items are large, then we have a PTAS for the general problem as well.

We will use the dynamic programming routine **BigJobDynProg**, which we developed in the previous lecture for scheduling problems, to get a PTAS for bin packing when pieces are large. Recall the following facts about the procedure :

input : $(a_1, a_2, \ldots, a_k) \equiv$ set of pieces ; a_i items of size i for all i.

output : $M(T, a_1, \ldots, a_k) \equiv$ number of bins (machines) needed to pack (schedule) $(a_1, \ldots, a_k)$ in bins (machines) of size T (by time T).

running time : $O(n^{2k})$, where k is the number of distinct sizes.

To apply **BigJobDynProg**, we must have a constant number of piece sizes. To get a small number of piece sizes, we must round them. Rounding them down is not a good idea because once we pack them in a bin of size 1, we cannot "expand" them to their original size because the size of bin is fixed (compare this with the scheduling problem where the makespan is a variable).

So, we round the sizes up, and we do this via a "grouping" of pieces. Given an instance I, arrange the pieces in decreasing order of size. Place l consecutive pieces in one group (starting from the largest size piece). Thus, the pieces are divided into $\lceil \frac{n}{l} \rceil$ groups, namely, $G_1, \ldots, G_{\lceil \frac{n}{l} \rceil}$, where G_1 contains the largest pieces and so on.

Produce another instance I' from I as follows : Discard G_1. For all other groups G_i, round the size of pieces to size of the largest piece in G_i. Thus, we have at most $\lceil \frac{n}{l} \rceil - 1$ distinct sizes.

Lemma 4.6 $OPT(I') \leq OPT(I) \leq OPT(I') + l$.

Proof: $OPT(I') \leq OPT(I)$ because each piece in I' can be mapped to a unique piece in I of size at least as large, e.g., pieces in G_2 in I' can be mapped to pieces in G_1 in I. So, any packing for I gives a packing for I' also.

$OPT(I) \leq OPT(I') + l$ because each piece in $I - G_1$ can be mapped to a piece in I' of size at least as large, e.g., pieces in G_2 in I_1 have smaller size than the corresponding pieces in G_2 in I'. So, given any packing for I' we need at most l more bins to pack I, for the l pieces in G_1. □

So, the algorithm is

FL

Separate the pieces into large ($> \frac{\epsilon}{2}$) and small ($\leq \frac{\epsilon}{2}$) pieces
$l \leftarrow \lceil \epsilon SIZE(I) \rceil$
Group the large pieces to get another instance I' as above
Pack G_1 into l bins
Apply **BigJobDynProg** to I' to pack it optimally into b bins
Apply FF to small pieces

Note that the number of large piece sizes in FL is at most

$$\frac{n}{l} \leq \frac{n}{\epsilon SIZE(I)} \leq \frac{n}{\epsilon \frac{n\epsilon}{2}} \leq \frac{2}{\epsilon^2}$$

because $SIZE(I) = \sum_i s_i \geq \sum_i \frac{\epsilon}{2} = \frac{n\epsilon}{2}$.

Theorem 4.7 $FL(I) \leq (1+\epsilon)OPT(I) + 1$

Proof: We have already proved the theorem for the case when FF adds new bins. So, assume that FF doesn't add new bins. In this case,

$$\begin{aligned} FL(I) &\leq b + l \\ &\leq OPT(I) + \lceil \epsilon SIZE(I) \rceil, \quad (4.1) \\ &\leq (1+\epsilon)OPT(I) + 1, \quad (4.2) \end{aligned}$$

where (4.1) follows since $b = OPT(I') \leq OPT(I)$ and (4.2) follows since $OPT(I) \geq SIZE(I)$. □

4.1.4 A Better-Than-PTAS for Bin Packing

We turn to linear programming relaxation techniques to get an algorithm for bin packing with only an additive error term. First we formulate bin packing as an integer program. Suppose an instance I consists of m distinct piece sizes and there are b_i pieces of size t_i in I.

Definition 4.1 A set $(a_1, \ldots, a_m)$ is called a *configuration* if it fits in 1 bin, i.e., $\sum_i a_i t_i \leq 1$.

Let N be the number of all possible configurations. Let $A_1, A_2, \ldots$ be an enumeration of all possible configurations. Let a_{ij} denote the i^{th} component of A_j, i.e., $A_j = (a_{1j}, a_{2j}, \ldots, a_{mj})$.

For each configuration A_j, we introduce a variable x_j which denotes the number of bins having configuration A_j. So an integer programming formulation of the bin packing problem is:

$$\begin{aligned}
\text{Min} \quad & \sum_{j=1}^{N} x_j \\
\text{subject to} \quad & \\
& \sum_{j=1}^{N} a_{ij} x_j \geq b_i \quad \text{for all } i = 1, \ldots, m \\
& x_j \in N
\end{aligned}$$

We relax this to a linear program by replacing the constraints $x_j \in N$ by $x_j \geq 0$. Note that the LP has a large number of variables. However, it can be solved efficiently, as the following theorem shows:

Theorem 4.8 (Karmarkar, Karp '82) The LP above can be solved to within an additive error of at most 1 in time polynomial in m and $\log\left(\frac{n}{t_m}\right)$, where t_m is the size of smallest piece in the instance.

We shall ensure that $t_m \geq \frac{1}{SIZE(I)}$, by filling in the smaller pieces afterwards with FF. If FF opens a new bin, this gives a solution which uses at most

$$\left(1 + \frac{2}{SIZE(I)}\right) SIZE(I) + 1 \leq OPT(I) + 3 \quad \text{bins.}$$

Also, since $t_m \geq \frac{1}{SIZE(I)}$, Theorem 4.8 implies that the LP can be solved in polynomial time.

Although the number of variables in the LP are very large, any extreme point will contain at most m non-zero variables. We can also assume that we can get an extreme point solution to the LP. So, if we round up the non-zero variables in an optimal solution to the LP, the number of bins used will be at most $OPT(I) + m$.

To do better than this, we introduce the following scheme:

1. Given LP solution x^* for instance I, pack $\lfloor x_j^* \rfloor$ bins according to configuration A_j. Denote the set of pieces packed by this as I_{int}, and the remaining pieces as $I - I_{int}$.

2. Recurse on $I - I_{int}$.

To prove that this is a good approximation algorithm, we must show how to make progress in every iteration. We begin by letting $LP(I)$ denote the value of LP on instance I.

Claim 4.9 $LP(I - I_{int}) + LP(I_{int}) \leq LP(I)$

Proof: We know that $\lfloor x^* \rfloor$ is feasible for the LP on I_{int} and $(x^* - \lfloor x^* \rfloor)$ is feasible for the LP on $I - I_{int}$. Thus,

$$LP(I - I_{int}) + LP(I_{int}) \leq \sum_j (x_j^* - \lfloor x_j^* \rfloor) + \sum_j \lfloor x_j^* \rfloor = \sum_j x_j^* = LP(I).$$

□

Lemma 4.10 $SIZE(I - I_{int}) \leq m$

Proof: Because x^* is an extreme point, at most m entries of x^* are non-zero. So at most m entries of $x^* - \lfloor x^* \rfloor$ are non-zero, and furthermore, each of them is less than one. Since $x^* - \lfloor x^* \rfloor$ is feasible for the LP of the bin packing instance $I - I_{int}$, by our previous rounding-up argument we can pack this instance into m bins. Thus the total size of the instance must be no more than m. □

The claim above shows that we make progress if $SIZE(I) > m$. To make progress in general, we will use a new type of "geometric grouping" each time before we solve the linear program. We will show that the grouping has the effect of making $m = SIZE(I)/2$, so that by the claim above, the size of the instance will fall by a factor of 2 each time we recurse on the "fractional" part of the instance.

To perform geometric grouping, arrange the pieces in decreasing order of size as before. Starting from the first piece, add pieces to a group until the group size becomes ≥ 2. Then, start the next group and so on until all items have been allocated to some group. Let the groups thus obtained be $G_1, \ldots, G_r$, where G_i has n_i pieces.

Claim 4.11 $n_i \geq n_{i-1}$

Proof: The claim follows from the fact that the items sizes in G_{i-1} are no larger than that of any item in G_i. □

We obtain a new grouping instance I' from I as follows

Obtaining I' from I

Discard G_1 and G_r
For $i \leftarrow 2$ to $r - 1$
    discard $n_i - n_{i-1}$ smallest pieces of G_i
    round the remaining n_{i-1} pieces of G_i to the size of the largest piece of G_i

Lemma 4.12 $OPT(I') \leq OPT(I) \leq OPT(I') + O(\log(SIZE(I)))$ and $LP(I') \leq LP(I)$.

Proof: $OPT(I') \leq OPT(I)$ and $LP(I') \leq LP(I)$ follow from the same kind of reasoning as in Lemma 4.6.

To prove $OPT(I) \leq OPT(I') + O(\log(SIZE(I)))$, we show that the total size of discarded pieces while constructing I' from I is $O(\log(SIZE(I)))$. Packing these discarded pieces by any constant factor approximation algorithm, e.g. FF, gives us the desired result.

Clearly, $SIZE(G_1), SIZE(G_r) \leq 3$, since we stop filling a group when its size exceeds 2.

It is also easy to see that the total size of the $n_i - 1$ largest pieces in G_i is at most 2, because before the addition of the last piece, the total size of G_i did not exceed 2. Using an averaging argument, we have

$$\text{size of } n_i - n_{i-1} \text{ smallest pieces in } G_i \quad \leq \frac{2}{n_i - 1}(n_i - n_{i-1}).$$

So,

$$\begin{aligned}
\text{total size of discards} \quad & \leq 6 + \sum_{i=2}^{r-1} \frac{2}{n_i - 1}(n_i - n_{i-1}) \\
& \leq 6 + 2\sum_{i=2}^{r-1}\left[\frac{1}{n_i - 1} + \frac{1}{n_i - 2} + \ldots + \frac{1}{n_i - (n_i - n_{i-1})}\right] \\
& \qquad (\text{since } \tfrac{1}{n_i - 1} \leq \tfrac{1}{n_i - k} \text{ for any } k \geq 1) \\
& = 6 + 2\sum_{j=n_1}^{n_{r-1}-1} \frac{1}{j} \\
& \leq 6 + 2H_{n_r}
\end{aligned}$$

Since each piece has size $\geq \frac{1}{SIZE(I)}$ and each group has size at most 3, the number of pieces in each group is at most $3SIZE(I)$. In particular, $n_r \leq 3SIZE(I)$. So,

$$\begin{aligned}
\text{total size of discards} \quad & \leq 6 + 2H_{n_r} \\
& = O(\log(n_r)) \\
& = O(\log(SIZE(I)))
\end{aligned}$$

□

Observe that the number of groups in I' is at most $\frac{SIZE(I)}{2}$ since each group has size at least 2. In I', all pieces in the same group get the same size. So,

$$\text{number of piece sizes in } I' \quad \leq \frac{SIZE(I)}{2}$$

which implies by Lemma 4.10

$$SIZE(I' - I'_{int}) \leq \frac{SIZE(I)}{2}.$$

Thus, the size of an instance goes down by a factor of two after grouping, solving the linear program, rounding down, and recursing on the remaining fractional part. We

stop the algorithm when $SIZE(I)$ becomes less than 1. Thus, there can be at most $O(\log(SIZE(I)))$ iterations of the algorithm.

We summarize the algorithm below.

KK

> Let I_0 be large pieces, i.e., pieces of size $\geq \frac{1}{SIZE(I)}$
> $i \leftarrow 0$
> While $(SIZE(I_i) \geq 1)$
>   perform geometric grouping to get I'_i
>   pack the discards in $O(\log(SIZE(I)))$ bins (follows from proof of Lemma 4.12)
>   Run LP on I'_i : pack the integral part $I'_{i,int}$ using the solution to the LP.
> $I_{i+1} \leftarrow I'_i - I'_{i,int}$
> $i \leftarrow i + 1$
> Pack remainder I_i in 1 bin
> Use FF to pack small pieces

Theorem 4.13 (Karmarkar, Karp '82) $KK(I) \leq OPT(I) + O(\log^2(SIZE(I)))$ for any bin packing instance I.

Proof: Using Lemma 4.12 and the fact that there are at most $O(\log(SIZE(I)))$ iterations, we get

$$\begin{aligned}
KK(I) \quad & = O(\log^2(SIZE(I))) + \sum_i LP(I'_{i,int}) \\
& \leq O(\log^2(SIZE(I))) + \sum_i (LP(I'_i) - LP(I'_i - I'_{int})) && (4.3) \\
& = O(\log^2(SIZE(I))) + \sum_i (LP(I'_i) - LP(I_{i+1})) \\
& \leq O(\log^2(SIZE(I))) + \sum_i (LP(I_i) - LP(I_{i+1})) && (4.4) \\
& = O(\log^2(SIZE(I))) + LP(I_0) \\
& \leq O(\log^2(SIZE(I))) + OPT(I),
\end{aligned}$$

where (4.3) follows by Claim 4.9 and (4.4) follows by Lemma 4.12.

□

ORIE 634 Approximation Algorithms September 24, 1998

Lecture 5

Lecturer: David P. Williamson *Scribe: Rif "Andrew" Hutchings*

5.1 Randomization: MAX SAT

5.1.1 Johnson's Algorithm

Today we start looking at some more modern results, as opposed to the 20 year old "classical" results that have occupied us thus far.

For purposes of these randomized algorithms we will make use of a function random(p). This is defined as:

random(p): $[0,1] \to \{0,1\}$

random(p) $= 1$ with probability p

random(p) $= 0$ with probability $1-p$

This leads to the following definition.

Definition 5.1 We have a *randomized α-approximation algorithm* if the algorithm runs in randomized polynomial time, and if it produces a solution that is always within a factor of α of an optimal solution

- with high probability (that is, at least $1 - \frac{1}{n^c}$ for some constant $c > 1$)
- OR in expectation,

over the random choices of the algorithm.

We begin by introducing the maximum satisfiability problem (MAX SAT) and a "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).

What is the dumbest possible use of randomization for this problem?

DumbRandom

```
For i ← 1 to n
    If random(1/2) = 1
        x_i ← TRUE
    else
        x_i ← 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. $\square$

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 $P = NP$.

5.1.2 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.

We can think of this sort of derandomization as a walk down a tree, with each choice of a variable value corresponds to a choice of a branch of the tree. The expectation at each node is the average of the expected values of the nodes below it. Thus we can assure ourselves that as we make these choices, our expected value is staying as least as large as it was initially. Thus the factor of $\frac{1}{2}$ still holds in this derandomized case.

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.

5.1.3 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 ← 1 to n
    If random(p) = 1
        x_i ← TRUE
    else
        x_i ← FALSE.
```

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

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

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

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

□

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:

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

□

We can actually extend this result to the general case when the length 1 clauses can be negated or non-negated. First note that if only a variable's negation appears as a length 1 clause, then we can redefine our variables to have a non-negated variable in that clause instead. The only case we really have to worry about, then, is the case where both a variable and its negation appear as length 1 clauses. In this case we can get the result above by making a stronger statement about the optimal value, OPT.

WLOG, assume that the weight of clause x_i is greater than weight of clause $\bar{x}_i$. Let v_i = wt. of $\bar{x}_i$. This allows a better bound on OPT:

$$OPT \leq \sum_j w_j - \sum_i v_i,$$

since the optimal solution can only satisfy either x_i or $\bar{x}_i$.

Letting C_j denote the j^{th} clause, we have:

$$\begin{aligned} E[W] = \sum_j w_j Pr[\text{clause j satisfied}] &\geq \sum_{j:\forall i\ C_j \neq \bar{x}_i} w_j Pr[C_j \text{ satisfied}] \\ &\geq p \cdot \sum_{j:\forall i\ C_j \neq \bar{x}_i} w_j \\ &\geq p \cdot [\sum_j w_j - \sum_i v_i] \\ &\geq p \cdot OPT \end{aligned}$$

5.1.4 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, Thompson '87)

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

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

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:

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

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

$$0 \leq z_j \leq 1.$$

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

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

Random Round

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

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

for nonnegative a_i.

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

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

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 $\sum_{i=1}^k y_i^* \ge z_j^*$.

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

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

$$z_j^*=0 \;\Rightarrow\; 1-(1-z_j^*/k)^k = 0$$
$$z_j^*=1 \;\Rightarrow\; 1-(1-z_j^*/k)^k = 1-\left(1-\frac{1}{k}\right)^k$$

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

Therefore,

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

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

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

5.1.5 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 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 ≥ 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:

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

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:

$$\begin{aligned}
l_j=1 &\Rightarrow \frac{1}{2}\cdot\frac{1}{2}+\frac{1}{2}z_j^*\geq\frac{3}{4}z_j^*\\
l_j=2 &\Rightarrow \frac{1}{2}\cdot\frac{3}{4}+\frac{1}{2}\cdot\frac{3}{4}z_j^*\geq\frac{3}{4}z_j^*
\end{aligned}$$

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

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

□

5.1.6 Non-linear Randomized Rounding

Nothing says that we had to take the output of the linear program directly into our randomized rounding scheme. We could have just as easily used some function of the output to generate rounding probabilities. This will result in a so-called non-linear randomized rounding approach.

Consider the following algorithm:

Nonlinear-Round

```
Solve LP, get (y*, z*).
Pick any function g(y) such that 1 − 4^(−y) ≤ g(y) ≤ 4^(y−1) for y ∈ [0,1].
For i ← 1 to n
    If random(g(y_i*)) = 1
        x_i ← TRUE
    else
        x_i ← FALSE.
```

Theorem 5.7 (Goemans, W '94) Nonlinear-Round is a 3/4-approximation algorithm for MAX SAT.

Proof: Recall Fact 5.2 in previous section: If $f(x)$ is concave in $[l,u]$, and $f(l)\geq al+b, f(u)\geq au+b$, then $f(x)\geq ax+b$ on $[l,u]$.

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

$$\begin{aligned}
\Pr[C_j \text{ satisfied}] &= 1-\prod_{i=1}^{k}(1-g(y_i^*))\\
&\geq 1-\prod_{i=1}^{k}4^{-y_i^*}\\
&= 1-4^{-\sum_{i=1}^{k}y_i^*}\\
&\geq 1-4^{-z_j^*} \qquad (5.5)\\
&\geq \frac{3}{4}z_j^*, \qquad (5.6)
\end{aligned}$$

where (5.5) follows from the LP constraint $\sum_{i=1}^{k}y_i^*\geq z_j^*$, and (5.6) follows from Fact 5.2.

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

$$\begin{aligned}
\Pr[C_j \text{ satisfied}] &= 1-\prod_{i=1}^{k-1}(1-g(y_i^*))\times g(y_k^*)\\
&\geq 1-\prod_{i=1}^{k-1}4^{-y_i^*}\times 4^{-(1-y_k^*)}\\
&\geq 1-4^{-z_j^*} \qquad (5.7)\\
&\geq \frac{3}{4}z_j^*, \qquad (5.8)
\end{aligned}$$

where again (5.7) follows from the LP constraint $\sum_{i=1}^{k-1}y_i^*+(1-y_k^*)\geq z_j^*$ and (5.8) follows from Fact 5.2. Clauses of other forms are similar.

Hence,

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

□

So where do you go from here? It turns out that this is the end of the line insofar as comparing against the linear programming bound. To see this, consider the instance $x_1 \vee x_2$,$x_1 \vee \bar{x}_2$, $\bar{x}_1 \vee x_2$, $\bar{x}_1 \vee \bar{x}_2$, where each clause has weight 1. An optimal LP solution for this instance sets $y_i = \frac{1}{2}$ for all i and $z_j = 1$ for all clauses (indeed, this is true for any instance which has no length 1 clauses). Thus $Z_{LP} = 4$, and the best bound we can get comparing our solution against Z_{LP} is $\frac{3}{4}$. Observe that in the case there are no length 1 clauses, the optimal solution of $y_i = \frac{1}{2}$ for all i gives no information about how to set the variables; essentially we are back to Johnson's algorithm in this case!

The best known approximation algorithm for MAX SAT so far is ≈ 0.78-approximation algorithm, and makes use of techniques we will see in the near future.

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

ORIE 634 Approximation Algorithms	October 1, 1998
Lecture 6	
Lecturer: David P. Williamson	*Scribe: Tim Roughgarden*

6.1 Randomization: MAX CUT

Today, with the goal of showing that the use of randomization can get quite sophisticated, we turn to the maximum cut problem.

MAX CUT

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

6.1.1 A Dumb Randomized Algorithm for MAX CUT

We begin as we did with the MAX SAT problem, by considering the simplest possible use of randomization.

DumbRandom

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

For notational simplicity, assume that $V = \{1, \ldots, n\}$.

Theorem 6.1 ($\approx$ Sahni, Gonzalez '76) DumbRandom is a 1/2-approximation algorithm.

Proof: For all $i, j \in V$ define a random variable X_{ij} such that

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

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

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

since $\sum_{i<j} w_{ij}$ must certainly be an upper bound on the value of a maximum cut (using the fact that all weights are non-negative). □

Remark 6.1 DumbRandom can be derandomized using the method of conditional expectations, yielding a deterministic 1/2-approximation algorithm.

6.1.2 MAX CUT in Dense Graphs

Unlike the case of MAX SAT, we can't do better than the dumb random algorithm with the tools that we have seen so far. In the second half of the lecture we will introduce a new tool that will help us do better. For the moment, we will show how the tools we already have help us to do better for a particular subcase of the MAX CUT problem. For the time being, we consider 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 DumbRandom 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:

$$\text{Max} \quad \sum_{i \in V} Z_i y_i$$

subject to:

$$Z_i - \epsilon n \leq \sum_{(i,j)\in E} (1 - y_j) \leq Z_i + \epsilon n \qquad \forall i$$

$$0 \leq y_i \leq 1$$

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:

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

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

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

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

A slight generalization of this bound is also useful.

Theorem 6.3 (Hoeffding) Let $X_1, \ldots, X_n$ be n independent random variables (not necessarily from the same distribution), such that each X_i takes either the value 0 or a_i for some $a_i \le 1$. 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 $\delta_i = \sqrt{\frac{2c\ln n}{\max\{ZN(y^*,i),2c\ln n\}}}$. Then

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

Then by the Union Bound, with probability at least $1 - 1/n^{c-1}$,

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

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 the Hoeffding bound with $Z = \max_i Z_i$ and $\delta = \sqrt{\frac{2c\ln n}{\max\{2c\ln n, \sum_i y^*_i \frac{Z_i}{Z}\}}}$

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

So w.h.p.,

$$\begin{aligned} \sum_i x'_i Z_i &\ge (1-\delta)\sum_i Z_i y^*_i \\ &= \left(1 - \sqrt{\frac{2c\ln n}{\max\{2c\ln n, \sum_i y^*_i \frac{Z_i}{Z}\}}}\right)\sum_i Z_i y^*_i \\ &\ge \sum_i Z_i y^*_i - \sqrt{2Zc\ln n \sum_i y^*_i Z_i} \\ &\ge \sum_i Z_i y^*_i - n\sqrt{2cn\ln n} \end{aligned}$$

Using this result to continue, we have

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

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 $\ge (1-\epsilon')OPT$ with high probability for sufficiently large n.

6.1.3 Degenie-izing the algorithm

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

Theorem 6.4 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.

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

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.

6.2 Semidefinite Programming

We now turn to a new tool which gives substantially improved approximation algorithms in some cases. We saw that in the case of MAX SAT, we could do no better than a $\frac{3}{4}$-approximation algorithm using the linear programming relaxation we introduced. In the case of MAX CUT, no known linear programming relaxation can lead to anything better than a $\frac{1}{2}$-approximation algorithm by bounding against the value of the LP objective function. So we turn to *convex* programming relaxations of various problems; in particular, to something called *semidefinite programming.*

First we define a positive semidefinite (psd) matrix X; we sometimes write $X \succeq 0$ to denote that X is psd.

Definition 6.1 A matrix $X \in \Re^{n\times n}$ is positive semidefinite (psd) iff $\forall a \in \Re^n$, $a^T X a \geq 0$.

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

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

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 only on $\log(1/\epsilon)$. This additive error of ϵ is necessary because sometimes the solutions to SDP's may be irrational numbers. SDPs also need to meet some additional specialized conditions (e.g., having an interior point in both the primal and the dual) to be solvable in this way, but all the SDPs we consider will be well-behaved, and so we will ignore these additional conditions.

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

$$\begin{array}{lll} \text{Max or Min} & \sum c_{ij}(\vec{v_i}\cdot\vec{v_j}) & \\ \text{subject to:} & & \\ & \sum_{i,j} a_{ijk}(\vec{v_i}\cdot\vec{v_j}) = b_k & \forall k \\ & \vec{v_i} \in \Re^n & \forall i \end{array}$$

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.

6.2.1 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).

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

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

Claim 6.5 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).

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

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

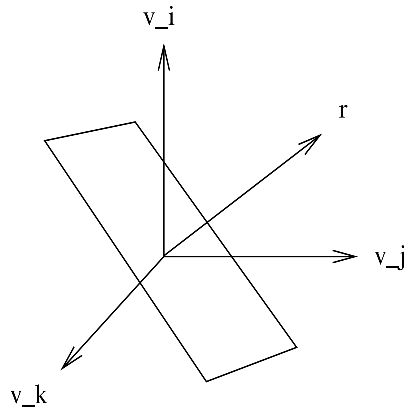


Figure 6.1: A random hyperplane example

that has a negative dot product with $\vec{r}$ will be on the other side (see Figure 6.1). 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 6.6 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 6.1 $\frac{\vec{r}}{||\vec{r}||}$ (i.e., normalization of $\vec{r}$) is uniformly distributed over a unit sphere.

Fact 6.2 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.

Corollary 6.7 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 6.2).

Lemma 6.8 $\Pr[i \in S, j \notin S \text{ or } i \notin S, j \in S] = \frac{1}{\pi}\arccos(\vec{v_i^*} \cdot \vec{v_j^*})$.

Proof:

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

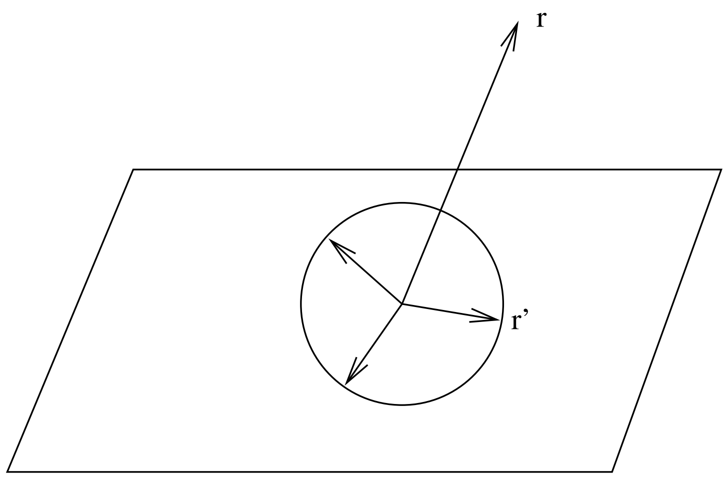


Figure 6.2: Projection of r to r'

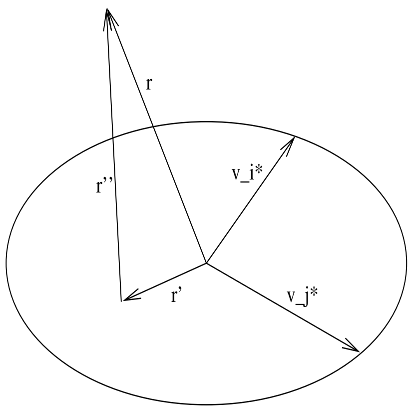


Figure 6.3: Projection of r into plane defined by v_i^* and v_j^*

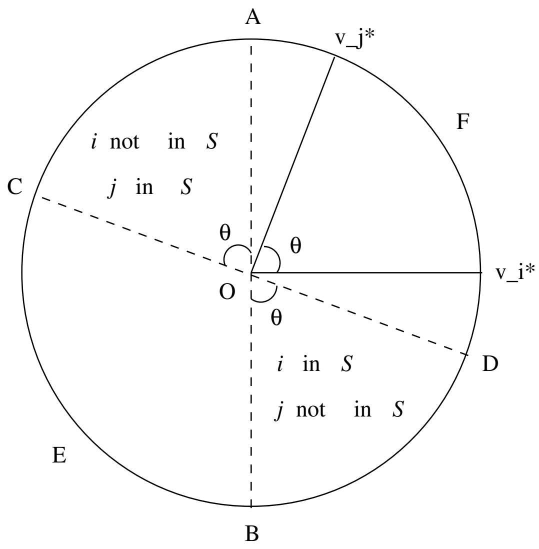


Figure 6.4: Determining the probability

If $\vec{r} = \vec{r'} + \vec{r''}$ (Figure 6.3), 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 6.4) 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. □

Lemma 6.9

$$\min_{-1 \leq x \leq 1} \frac{\frac{1}{\pi}\arccos(x)}{\frac{1}{2}(1-x)} \geq 0.878.$$

Proof: Using Mathematica! □

So now we can prove the theorem.

Theorem 6.10 (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

$$\begin{aligned} E[W] &= \sum_{i<j} w_{ij} \cdot \Pr[i \in S, j \notin S \text{ or } i \notin S, j \in S] \\ &= \sum_{i<j} w_{ij} \frac{1}{\pi} \arccos(\vec{v_i^*} \cdot \vec{v_j^*}) \qquad (6.1) \\ &\geq 0.878 \cdot \frac{1}{2} \sum_{i<j} w_{ij} (1 - \vec{v_i^*} \cdot \vec{v_j^*}) \qquad (6.2) \\ &= 0.878 \cdot Z_{SDP} \\ &\geq 0.878 \cdot OPT, \end{aligned}$$

where (6.1) follows by Lemma 6.8 and (6.2) follows by Lemma 6.9. □

ORIE 634 Approximation Algorithms	October 8, 1998
Lecture 7	
Lecturer: David P. Williamson	*Scribe: Kathryn Nyman*

7.1 Semidefinite Programming

7.1.1 MAX CUT, continued

In the last lecture we saw that a semidefinite program (SDP) is equivalent to *vector programming* (VP) which can be formally stated as

$$\begin{aligned} &\text{Max or Min} \quad \sum c_{ij} (\vec{v_i} \cdot \vec{v_j}) \\ &\text{subject to:} \\ &\qquad \sum_{i,j} a_{ijk} (\vec{v_i} \cdot \vec{v_j}) = b_k \qquad \forall k \\ &\qquad \vec{v_i} \in \Re^n \qquad \forall i, \end{aligned}$$

where n is the number of vectors in the vector program.

Last time, we considered the following formulation of the MAX CUT:

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

We showed that it could be relaxed to the following vector program.

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

We proved the following two lemmas:

Lemma 7.1 $\Pr[i \in S, j \notin S$ or $i \notin S, j \in S] = \frac{1}{\pi}\arccos(\vec{v_i^*} \cdot \vec{v_j^*})$.

Lemma 7.2 For $-1 \leq x \leq 1$, $\frac{1}{\pi}\arccos(x) \geq .878\frac{1}{2}(1-x)$

These two lemmas imply that the Vector Round algorithm from last time gives a .878-approximation algorithm for MAX CUT. Can we do better than this? It turns out that we will have to do something quite different, as the following theorems attest.

Corollary 7.3 For any graph with non-negative weights,

$$\frac{OPT}{Z_{SDP}} \geq 0.878.$$

Theorem 7.4 (Delorme, Poljak '93) For the 5-cycle,

$$\frac{OPT}{Z_{SDP}} = \frac{32}{25+5\sqrt{5}} \approx 0.884.$$

Theorem 7.5 (Karloff '95) There exists graphs G such that,

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

This is true even if any valid inequality is added to the 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. So far, no better approximation algorithm is known. However, some very recent work shows that the worst case of the algorithm is quite confined: in many cases, we can get an approximation algorithm with a performance guarantee better than .878.

Theorem 7.6 (Zwick, 10/7/98) If for a graph G

$$\frac{Z_{SDP}}{\sum_{i<j} w_{ij}} < .84 - \epsilon$$

then there is a $(.878 + f(\epsilon))$-approximation algorithm, where f is an increasing function.

Compare this with previously known work:

Theorem 7.7 (Goemans, W '95) If for a graph G

$$\frac{Z_{SDP}}{\sum_{i<j} w_{ij}} > .84 + \epsilon$$

then VectorRound is a $(.878 + g(\epsilon))$-approximation algorithm, where g is an increasing function.

It turns out that there is a limit on how well we can do in any case.

Theorem 7.8 (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

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

We get into some conceptual trouble if $OPT < 0$, since then our notion of coming within a factor of $\alpha < 1$ for a maximization problem implies that we would be doing better than optimal! We could redefine the notion of an approximation algorithm for this case, but for now we restrict our attention to a case in which the objective function is non-negative. Assume $A \succeq 0$. This solves the problem since the objective function is $x^t Ax$, and we know that since A is psd, then $x^t Ax \geq 0$.

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

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

Consider the same vector rounding algorithm.

VectorRound2

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

We will now give two lemmas similar to Lemmas 7.1 and 7.2. Dimitris Bertisimas has observed that the first lemma was shown by Shepherd in 1900.

Lemma 7.9 $E[\bar{x}_i \cdot \bar{x}_j] = \frac{2}{\pi}\arcsin(\vec{v_i^*} \cdot \vec{v_j^*})$

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 7.1, and the fourth equality follows since $\arcsin(x) + \arccos(x) = \frac{\pi}{2}$. $\square$

We would like our proof to proceed as follows. We would like to prove an equivalent of Lemma 7.2; suppose there is some α such that the following is true:

Lemma 7.10

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

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

$$\begin{aligned}
E[\sum_{i,j} a_{ij}(\bar{x}_i \cdot \bar{x}_j)] &= \sum_{i,j} a_{ij} E[\bar{x}_i \cdot \bar{x}_j] \\
&= \sum_{i,j} a_{ij} \frac{2}{\pi}\arcsin(\vec{v_i^*} \cdot \vec{v_j^*}) \\
&\ge \alpha \sum_{i,j} a_{ij}(\vec{v_i^*} \cdot \vec{v_j^*}) \qquad (7.1) \\
&\ge \alpha \cdot OPT. \qquad (7.2)
\end{aligned}$$

But we cannot do this because the inequality (7.1) 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 $a_{ij} < 0$.

Thus to analyze this algorithm, we will have to do something different. We will have to consider comparing the expected value of the solution to the SDP value on a global basis, rather than a term-by-term basis.

Fact 7.1 If $A \succeq 0$, $B \succeq 0$, then $\sum_{ij} a_{ij} b_{ij} \ge 0$.

Fact 7.2 If $X \succeq 0$, $|x_{ij}| \le 1$, and $Z = (z_{ij})$ such that $z_{ij} = \arcsin(x_{ij}) - x_{ij}$, then $Z \succeq 0$.

Theorem 7.11 (Nesterov '97) VectorRound2 is a $\frac{2}{\pi}$-approximation algorithm.

Proof: We want to show

$$E[\sum_{ij} a_{ij}(\bar{x}_i \cdot \bar{x}_j)] \ge \frac{2}{\pi}\sum_{ij} a_{ij}(\vec{v_i^*} \cdot \vec{v_j^*}) \ge \frac{2}{\pi} \cdot OPT$$

We know

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

So we want to show

$$\frac{2}{\pi}\sum_{ij} a_{ij}\arcsin(\vec{v_i^*} \cdot \vec{v_j^*}) - \frac{2}{\pi}\sum_{ij} a_{ij}(\vec{v_i^*} \cdot \vec{v_j^*}) \ge 0$$

Setting $x_{ij} = \vec{v_i^*} \cdot \vec{v_j^*}$, we obtain that $X = (x_{ij}) \succeq 0$ and $|x_{ij}| \le 1$, since

$$\vec{v_i^*} \cdot \vec{v_j^*} = \|\vec{v_i^*}\|\|\vec{v_j^*}\|\cos\theta_{ij} = \cos\theta_{ij}.$$

Thus the left-hand side is equal to

$$\frac{2}{\pi}\sum_{ij} a_{ij}(\arcsin(x_{ij}) - x_{ij}).$$

Setting $z_{ij} = \arcsin(x_{ij}) - x_{ij}$, this is equal to

$$\frac{2}{\pi}\sum_{ij} a_{ij} z_{ij} \ge 0,$$

since $Z = (z_{ij})$ is psd by Fact 7.2 and thus $\sum_{ij} a_{ij} z_{ij} \ge 0$ by Fact 7.1. Hence the result. $\square$

7.1.3 Graph Coloring

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

Theorem 7.12 (Karger, Motwani, Sudan '94) There is a polynomial-time algorithm to color a 3-colorable graph with $\tilde{O}(n^{1/4})$ colors.

The previous best algorithms used $\tilde{O}(n^{3/8})$ colors (Blum '94) and $O(\sqrt{n})$ colors (Wigderson '83).

Definition 7.1 A function $f(n) = \tilde{O}(g(n))$ when the following is valid:

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

The following facts are known about coloring graphs.

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

Proof: Color greedily (color with color not used by neighbors yet). □

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

Color1

While $\exists v \in G$ s.t. $\deg(v) \geq \sqrt{n}$
  Color v with color #1
  Color neighbors of v in polynomial time with 2 new colors
  Remove v and its neighbors from graph
Color remaining G with $\sqrt{n}$ new colors

Theorem 7.13 (Widgerson '83) Color1 colors 3-colorable graphs in polynomial time 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 $3\sqrt{n} + 1$. □

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

$$\text{Min} \quad \lambda$$

subject to:

$$\begin{array}{ll} v_i \cdot v_j \leq \lambda & \forall \quad (i,j) \in E \\ v_i \cdot v_i = 1 & \forall \quad i \\ v_i \in R^n & \end{array}$$

Claim 7.14 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 this solution is a feasible solution to the vector program with $\lambda = -1/2$. Thus in the optimal solution, $\lambda \leq -1/2$. □

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 $R_1 = \{i : r_1 \cdot v_i \geq 0, r_2 \cdot v_i \geq 0, \ldots, r_t \cdot v_i \geq 0\}$
  $R_2 = \{i : r_1 \cdot v_i < 0, r_2 \cdot v_i \geq 0, \ldots, r_t \cdot v_i \geq 0\}$
  ⋮
  $R_{2^t} = \{i : r_1 \cdot v_i < 0, r_2 \cdot v_i < 0, \ldots, r_t \cdot v_i < 0\}$
Color vertices in R_i with color i

Theorem 7.15 (Karger, Motwani, Sudan '94) 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 $m \le n\Delta/2$. 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 polynomial time 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. To balance these two parts, we set σ such that $\frac{n}{\sigma} = \sigma^{\log_3 2}$, which gives $\sigma = n^{0.613}$. This gives 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 7.16 (Karger, Motwani, Sudan)

$$\Pr[i \text{ and } j \text{ get same color for edge } (i,j)] = \tilde{O}(t^{-3}) = \tilde{O}(\Delta^{-1})$$

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

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

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.

ORIE 634 Approximation Algorithms — October 15, 1998

Lecture 8

Lecturer: *David P. Williamson* — Scribe: *Vardges Melkonian*

8.1 The Primal-Dual Method

8.1.1 The Basic 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. $A \cap T_i \neq \emptyset \quad \forall i$.

Note that this problem is equivalent to the Set Cover problem. Namely, the ground element e_i of the Hitting Set corresponds to the subset S_i of the Set Cover, and the subset T_i of the Hitting Set corresponds to the ground element t_i of the Set Cover.

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

- Step 1

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

- 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:

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

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.

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.

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.

Application: Feedback Vertex Set Problem

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

Proof: Do breadth-first search of the graph. By the properties of the graph, if we do not close a cycle by revisiting a previously explored node, then at least in every other level the number of explored nodes increases by a factor of 2. Thus at depth i, we will have explored $2^{i/2}$ nodes. By depth $2\log_2 n$, we will have found a cycle. $\square$

Thus in our algorithm, we always choose as our violated set any unhit cycle of length no longer than $4\log_2 n$.

Theorem 8.2 (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. $\Box$

8.1.2 Deleting Unnecessary Elements

The above algorithm (Primal-Dual1) 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 elements increase the cost of A. Consider the following refinement to remove the unnecessary elements 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 element 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 8.1 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 8.3 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 $|A_f \cap T_{k_j}| \leq \max |B \cap T_{k_j}| \leq \beta$. $\Box$

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.

Application: 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 $\delta(S) = \{(u, v) \in E : u \in S \text{ and } v \notin S\}$. 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

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

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.

Theorem 8.4 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 8.3. Let A be an infeasible solution, and let B be a minimal augmentation of A. Now let $s, v_1, v_2, \dots, v_l, t$ 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 $s, w_1, w_2, \dots, w_j, v_i$, where $w_\ell \in S_k$. Thus $s, w_1, w_2, \dots, w_j, v_i, v_{i+1}, \dots, v_l, t$ is an s-t path, and if we let $B' = \{(v_i, v_{i+1}), (v_{i+1}, v_{i+2}), \dots, (v_l, t)\}$, 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, $\beta = 1$. $\square$

Application: 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 $(s_i, t_i), i = 1 \dots l$
  - 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 : $T_i = \delta(S_i)$ iff $|S_i \cap \{s_j, t_j\}| = 1$ 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.

ORIE 634 Approximation Algorithms	October 22, 1998
Lecture 9	
Lecturer: David P. Williamson	*Scribe: Christina Ahrens*

9.1 The Primal-Dual Method

9.1.1 Generalized Steiner Trees, continued

We will now continue our discussion of the Generalized Steiner Tree problem. Recall first the definition of a minimal augmentation:

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.

And also the Generalized Steiner Tree problem itself:

Generalized Steiner Tree Problem

- **Input:**
  - An undirected graph $G = (V, E)$
  - l pairs of vertices $(s_i, t_i), i = 1 \dots l$
  - 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).

Recall that this can be modeled 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 $\{(s, t_j)\}, j = 1 \dots n$ is a minimum augmentation that crosses the cut $\delta(\{s\})$ l times, which (using Theorem 8.3) would imply a β-approximation algorithm, for $\beta \geq l$. This is not very good.

Perhaps we picked the wrong infeasible solution to augment; that is, maybe our previous algorithm will work if we change the way that we choose the violated set. It turns out that this approach does not work either because it still leads to a l-approximation in the worst case scenario (where all the s's are in one group, unconnected to any of the t's).

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 $\frac{l+l}{l+1} < 2$. Therefore it might be wise to pick several violated sets and increase the associated dual variables all at the same time. This observation leads 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 $\sum_{i: e_l \in T_i} y_i = c_{e_l}$
  $A \leftarrow A \cup \{e_l\}$
For $j \leftarrow l$ down to 1
  If $A - \{e_j\}$ is still feasible
    $A \leftarrow A - \{e_j\}$
Return A.

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

Theorem 9.1 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

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

Proof: Homework problem, problem set 4. Note that this theorem is a generalization of Theorem 8.3. $\square$

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

Theorem 9.2 (Agrawal, Klein, Ravi '95, Goemans, W '95) For this subroutine Violated, **Primal-Dual3** is a 2-approximation algorithm for the Generalized Steiner Tree Problem.

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 9.3 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. $\square$

Note that we can actually improve one of the previous results slightly to

$$\sum_{v \in Red} deg(v) \leq 2|Red| - 2c,$$

where $c =$ the number of components in $G' = (V', B)$.

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.

9.1.2 Feedback Vertex Set Problem, revisited

Recall the Feedback Vertex Set Problem:

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).

By using the primal-dual method we were able to obtain a $4 \log_2 n$ approximation algorithm.

Recall also the general IP formulation of the problem that we used:

$$\begin{aligned}
\text{Min} \quad & \sum_{v \in V} w_v x_v \\
\text{subject to:} \quad & \\
& \sum_{v \in C} x_v \geq 1 \qquad \forall \text{ cycles } C \\
& x_v \in \{0, 1\}
\end{aligned}$$

Definition 9.2 The *integrality gap* of an integer program is the worst-case ratio of the optimum value of the integer program to the optimum value of its linear programming relaxation.

Because our approximation algorithm for the FVS problem was based on a dual feasible solution, we know that the integrality gap for the above formulation of the FVS problem is at most $4\log_2 n$. To see this, observe that in any primal-dual algorithm that gives a performance guarantee of α we find a solution A and a dual feasible solution y such that

$$IP_{OPT} \leq \sum_{e\in A} c_e \leq \alpha \sum_i y_i \leq \alpha LP_{OPT}.$$

We would like to improve our approximation algorithm, but we are in trouble because it is known that for the above formulation of the FVS, the integrality gap is $\Omega(\log_2 n)$. Therefore, to obtain a better bound using the primal-dual method we must use a different IP formulation of the problem.

Some facts about biconnected components

In order to develop our new integer programming formulation of the feedback vertex set problem, we will need to define the *biconnected components* of a graph, and state some simple facts about them. A biconnected (or 2-connected or 2-vertex-connected) graph is one such that for all distinct $x, z \in V$ and for all $a \in V$ there exists a path in G from x to z not containing a.

Any graph (biconnected or not) can be decomposed into "biconnected components"; that is, subgraphs which behave like biconnected graphs, just like the connected components of a graph are subgraphs that are connected. To be more precise, the edges of an undirected graph can be partitioned into biconnected components $E_1, E_2, \ldots, E_k$ using the following equivalence relationship: $e_1 \approx e_2$ iff $e_1 = e_2$ or e_1 and e_2 are in some common simple cycle. Let $V_i =$ vertices of edges in E_i. Then the following are true:

- $G_i = (V_i, E_i)$ is biconnected;
- $|V_i \cap V_j| \leq 1$ for all $i \neq j$.

Now let $d(v) =$ degree of V, $b(v) = |\{i : v \in V_i\}|$, $c(G) =$ the number of connected components in G, and $k =$ number of biconnected components in G. As a warm up, we prove the following fact that we will need later.

Lemma 9.4
$$\sum_{v\in V} b(v) \leq |V| + k - c(G).$$

Proof: We prove the statement by contradiction. Note that the inequality holds for the empty graph. Now let H by the smallest graph (in terms of number of vertices) for which the inequality does not hold. For any graph, there must exist some biconnected component such that there is at most one vertex of the component with $b(v) \geq 2$; this follows since otherwise there would exist a cycle whose edges were in different biconnected components, which violates the equivalence relation. Pick such a biconnected component E_i of H. If $b(v) = 1$ for all $v \in V_i$, then V_i is a connected component of H; deleting V_i and E_i from H decreases both the left-hand side of the inequality and the right-hand side of the inequality by $|V_i|$ (note that on the right-hand side the number of connected and biconnected components both decrease by 1). Thus if the inequality does not hold for H, it does not hold for H with V_i and E_i deleted, which contradicts the minimality of H.

Now suppose that $b(v) \geq 2$ for exactly one $v \in V_i$. Deleting E_i and $V_i - v$ from H decreases both the left-hand side and right-hand side of the inequality by $|V_i|$, the left-hand side since $b(v)$ decreases by 1 for $|V_i|$ vertices, and the right-hand side since $|V_i| - 1$ vertices are removed and one biconnected component is removed. This again contradicts the minimality of H. $\square$

A new integer programming formulation for FVS

To get a new integer programming formulation for the feedback vertex set problem, we prove the following lemma.

Lemma 9.5 For any FVS F,

$$\sum_{v\in F}(d(v) - b(v)) \geq |E| - |V| + c(G).$$

Proof: We first start by proving a simpler inequality. We know that if we remove F and adjacent edges from the graph, the remaining set of edges is acyclic. Since $\sum_{v\in F} d(v)$ is a bound on the number of edges removed, and $|V| - |F| - c(G)$ is a bound on the number of edges left over, we have that

$$|E| \leq \sum_{v\in F} d(v) + |V| - |F| - c(G).$$

Rearranging terms gives us

$$\sum_{v\in F}(d(v) - 1) \geq |E| - |V| + c(G),$$

which is a start on what we want.

Now define $d_S(v) \equiv$ degree of vertex v in graph $(S, E[S])$, where $E[S]$ is the subset of E that has both endpoints in S. Now assume that we have partitioned our graph into biconnected components $E_1, E_2, \ldots, E_k$. Now we apply the inequality above to each of the biconnected components of the graph. This gives us

$$|E| \le \sum_{i=1}^{k} \left(\sum_{v \in V_i \cap F} d_{V_i}(v) + |V_i| - |V_i \cap F| - 1 \right) = \sum_{v \in F} d(v) + \sum_{v \in V} b(v) - \sum_{v \in F} b(v) - k.$$

Applying the lemma above gives

$$|E| \le \sum_{v \in F} d(v) + |V| + k - c(G) - \sum_{v \in F} b(v) - k.$$

Rearranging terms we arrive at the statement of the lemma. □

Now consider $S \subseteq V$ and $G[S] = (S, E[S])$. Set $f(S) = |E[S]| - |S| + c(G[S])$. Then by observing that $F \cap S$ is an fvs for $G[S]$, we have the following corollary.

Corollary 9.6 For any subset $S \subseteq V$,

$$\sum_{v \in F \cap S} (d_S(v) - b_S(v)) \ge f(S).$$

Using this we can formulate the following integer program:

$$\begin{array}{lll} \text{Min} & \sum_{v \in V} w_v x_v & \\ \text{subject to:} & & \\ & \sum_{v \in S} (d_S(v) - b_S(v)) x_v \ge f(S) & \forall S \subseteq V \\ & x_v \in \{0, 1\}. & \end{array}$$

Note that every FVS is feasible for the IP because of the corollary above. If x is not a FVS, we need to show some constraint is violated. We know that if x is not an FVS, there must exist a cycle C such that $x_v = 0 \ \forall v \in C$. So now look at the constraint corresponding to C; the left-hand side will be zero, but the right-hand side will be at least one because $f(C) \ge 1$; this follows since the number of edges in C must be at least $|C|$ (in order for there to be a cycle) and $c(G[C]) \ge 1$. Therefore, the integer program above exactly models the feedback vertex set problem in undirected graphs.

Next time we will take the dual of this new IP and apply the primal-dual method to get a 2-approximation algorithm.

Lecture 10

Lecturer: David P. Williamson *Scribe: dan brown*

10.1 The Primal-Dual Method

10.1.1 The Feedback Vertex Set Problem, cont.

From last lecture, we remember what the feedback vertex set problem is:

Feedback Vertex Set in Undirected Graphs

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

We've shown a primal-dual method which gave a $4 \log n$-approximation for this problem. However, since the LP which gave rise to this algorithm has an integrality gap which is $\Omega(\log n)$, this is essentially the best approximation algorithm possible from this particular LP (to within a constant factor).

Last time, we came up with a different IP formulation for the problem–this time we show that it can give a 2-approximation algorithm via the primal-dual method.

First, some notation:

- Let $d(v)$ be the degree in the graph G of node v.
- Let $b(v)$ be the number of biconnected components of G that include the vertex v.
- Let $c(G)$ the the number of connected components of the graph G.
- Let $G[S]$, for a graph $G = (V, E)$ and $S \subset V$, be the induced subgraph of G on S. That is, $G[S]$ is a graph whose vertex set is S, and whose edges are those edges of E with both endpoints in S.
- Let $E[S]$ be the edge set of $G[S]$ (those edges whose endpoints are both in S).

- Let $d_S(v)$ be the degree of the node v in the graph $G[S]$.
- Let $b_S(v)$ be the number of biconnected components of $G[S]$ which include v.
- Let $f(S) = |E[S]| - |S| + c(G[S])$.

Last time, we showed that the following integer program is a correct formulation for the feedback vertex set problem (FVS):

$$
\begin{array}{lll}
\text{Min} & \displaystyle\sum_{v \in V} w_v x_v & \\
\text{subject to:} & & \\
& \displaystyle\sum_{v \in S} (d_S(v) - b_S(v)) x_v \geq f(S) & \forall S \subseteq V \\
& x_v \in \{0, 1\} & \forall v \in V.
\end{array}
$$

Our goal for today is to use the relaxation of this IP to get a 2-approximation algorithm for FVS. This idea is from [Becker, Geiger '94] and [Bafna, Berman, Fujito '95], and was made into a primal-dual algorithm by [Chudak, Goemans, Hochbaum, W]. The approach we use today is from Fujito.

The dual of the linear programming relaxation of the above IP is:

$$
\begin{array}{lll}
\text{Max} & \displaystyle\sum_{S} f(S) y_S & \\
\text{subject to:} & & \\
& \displaystyle\sum_{S: v \in S} (d_S(v) - b_S(v)) \leq w_v & \forall v \in V \\
& y_S \geq 0 & \forall S \subseteq V.
\end{array}
$$

Using our previous primal-dual algorithms as a guide, we devise the following primal-dual algorithm.

FVSPrimalDual

> $y \leftarrow 0$
> $F \leftarrow \emptyset$
> $S \leftarrow V$
> $i \leftarrow 0$ (i is a counter)
> While F is not feasible ($G[S]$ isn't acyclic)
> $i \leftarrow i + 1$
>   Increase y_S until $\sum_{S: v_i \in S} (d_S(v_i) - b_S(v_i)) y_S = w_i$ for some $v_i \in S$.
> $F \leftarrow F \cup \{v_i\}$
> $T \leftarrow \{v \in S : v \text{ not on some cycle in } G[S - v_i]\}$
> $S \leftarrow S - \{v_i\} - T$
> For $l \leftarrow i$ down to 1
>   If $F - \{v_l\}$ is still a feedback vertex set
> $F \leftarrow F - \{v_l\}$
> Return F.

This is a very standard primal-dual method; let's try the analysis mode we've been using for a while. Let F_f be the final feedback vertex set returned by the algorithm. We know that the following is true, by simple reversal of sums:

$$
\begin{aligned}
\sum_{v \in F_f} w_v &= \sum_{v \in F_f} \sum_{S: v \in S} (d_S(v) - b_S(v)) y_S \\
&= \sum_{S} \sum_{v \in F_f \cap S} y_S (d_S(v) - b_S(v))
\end{aligned}
$$

If we can show that this sum is less than or equal to $2 \sum_S f(S) y_S$, twice the feasible dual solution's objective function value, then we have a 2-approximation algorithm, since the primal objective will be less than two times a lower bound on the problem's objective. So we'd like to show that, if $y_S > 0$, then

$$\sum_{v \in F_f \cap S} (d_S(v) - b_S(v)) \leq 2 f(S).$$

As in previous proofs about the primal-dual method, we would like to frame this inequality as one about minimal augmentations or minimal feedback vertex sets, so that the inequality reduces to a general statement about graphs, rather than one about the workings of this particular algorithm. The following claim will allow us to do this.

Claim 10.1 *$F_f \cap S$ is a minimal FVS for $G[S]$, for all S with $y_S > 0$.*

Proof: Pick an S for which $y_S > 0$. We raised y_S in, say, the ith iteration of the while loop. We prove the statement by contradiction: suppose there exists an element

$v_j \in F_f \cap S$ such that $F_f \cap S - v_j$ is a feedback vertex set for $G[S]$. By the construction of the algorithm, we know that it must be the case that $\{v_1, v_2, \ldots, v_{j-1}\} \cup F_f - v_j$ must not be a feedback vertex set for G, otherwise we would have removed v_j from the solution in the reverse delete step of the algorithm. Thus if $\{v_1, v_2, \ldots, v_{j-1}\} \cup F_f - v_j$ is not a feedback vertex set for G, it follows that v_j must lie on some cycle C such that v_j is the only vertex of $\{v_1, \ldots, v_{j-1}\} \cup F_f$ in C. Also, $C \subseteq S$ since S consists only of vertices lying on cycles remaining after the removal of $\{v_1, \ldots, v_i\}$ and $i < j$ since $v_j \in S$. Thus since C is in $G[S]$ and $C \cap F_f = \{v_j\}$, it cannot be the case that $F_f \cap S - v_j$ is a feedback vertex set for $G[S]$. □

So we know now that for any S such that $y_S > 0$, $F_f \cap S$ is a minimal feedback vertex set. If we combine this fact with the following theorem, we can show that we have a 2-approximation algorithm by our previous reasoning; i.e., it implies that $\sum_{v \in F_f} w_v \le 2 \sum_S f(S) y_S$.

Theorem 10.2 For any graph G such that every $v \in V$ is contained in some cycle, and for any minimal feedback vertex set F of G,

$$\sum_{v \in F} (d(v) - b(v)) \le 2f(V) = 2(|E| - |V| + c(G)).$$

Proof: We know that $\sum_{v \in V} d(v) = 2|E|$, so removing $2|E|$ from both sides of the inequality, and doing some rearranging, we are left with trying to prove:

$$\sum_{v \notin F} d(v) \ge 2|V| - \sum_{v \in F} b(v) - 2c(G).$$

Notice that $\sum_{v \notin F} d(v) = \sum_{v \notin F} d_{V-F}(v) + |\delta(F)|$. Consider the first term on the right-hand side. Let $l = c(G[V-F])$ be the number of connected components of $G[V-F]$. $G[V-F]$ is a forest, since it's the acyclic graph we get when we remove F from V, so we know that $\sum_{v \notin F} d_{V-F}(v) = 2(|V| - |F| - l)$. Subtracting that from both sides, we are left with wanting to show that:

$$|\delta(F)| \ge 2|F| + 2l - \sum_{v \in F} b(v) - 2c(G),$$

or

$$2|F| + 2l \le |\delta(F)| + \sum_{v \in F} b(v) + 2c(G).$$

Since F is a minimal feedback vertex set for G, each $v \in F$ is the unique vertex of F which is in some cycle C_v (otherwise we could delete v and still have a feedback vertex set). Hence, each v in F is adjacent to two "cycle" edges from F to $V - F$, and we can "charge" the quantity $2|F|$ on the left-hand side to these edges in $|\delta(F)|$ on the right-hand side. Observe that these two cycle edges for any vertex $v \in F$ must both be adjacent to the same component of $G[V-F]$.

Now, wish to show that for each component of $G[V-F]$, we can "charge" 2 against the right-hand side of the above inequality; either to the number of connected components of G (which counts double because of their coefficient), to the non-cycle edges of $\delta(F)$, or to $b(v)$ for some $v \in F$. If we can successfully show that we can charge 2 to the right-hand side for each component in $G[V-F]$, then we have shown that the left-hand side is, in fact, no more than the right-hand side.

We will do this by a large case analysis based on the components C of $G[V-F]$. Notice that since every vertex in G lies on some cycle, every cycle contains some vertex of F, it must be the case that every component C of $G[V-F]$ is adjacent to at least two edges from $\delta(F)$.

1. If there are any number of cycle edges and at least 2 non-cycle edges connecting C to F, then charge 2 for the non-cycle edges to $|\delta(F)|$.

2. If there is exactly one pair of cycle edges which connect C to $v \in F$, and no other edges in G between F and C, then either $v \cup C$ is a connected component of G, and we can charge 2 to $2c(G)$, or $v \cup C$ is a biconnected component of G with $b(v) \ge 2$, and we can charge 2 to $b(v)$.

3. If there is exactly one pair of cycle edges between C and $v \in F$, and exactly one other edge between $u \in C$ and $w \in F$, then charge 1 to $b(v)$ and 1 to the edge $(u, w) \in \delta(F)$.

4. If there are at least 2 pairs of cycle edges connecting C to $v, w \in F$, then we can charge 1 each to $b(v)$ and $b(w)$.

□

Through this theorem, then, the desired 2-approximation is shown.

10.2 Metric Methods

10.2.1 The Minimum Multicut Problem

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 $(s_i, t_i), i = 1 \dots k$
  - 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 $G' = (V, E - F)$.

This problem is 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 (admittedly exponentially large) integer programming formulation of the problem is:

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

subject to:

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

$$x_e \in \{0, 1\}. \qquad \forall e \in E.$$

We then relax the integer program to a linear program by replacing the constraints $x_e \in \{0, 1\}$ by $x_e \geq 0$.

This linear program can be solved in polynomial time by the *ellipsoid method*, even though it is exponential-sized, if we can provide a polynomial-time *separation oracle* for the problem. A separation oracle for a problem is a subroutine which, when given a possibly feasible vector x either certifies that x is feasible for the LP, or supplies a violated LP inequality which the x fails to satisfy.

In this case, a separation oracle is simple, since we can just use a subroutine which solves the shortest path problem. That is, let x_e be the length of the edge e. The LP constraints, $\sum_{e \in P} x_e \geq 1$ for all $s_i - t_i$ paths P, will be satisfied if the shortest $s_i - t_i$ path is at least 1 in length. If the shortest $s_i - t_i$ path for some i is less than 1 in length, then that path provides a violated primal inequality. On the other hand, if, for all i the shortest $s_i - t_i$ path is at least 1 in length, then all paths are at least 1 in length, and x is a feasible vector. So a separation oracle is trivial to construct.

For that matter, we can also construct a polynomial-sized linear program which is equivalent to this LP and use more practical linear programming algorithms than the ellipsoid method.

We can gain some insight into the linear program by considering a physical interpretation of it. We think of the LP as a pipe system, where $e = (i, j) \in E$ means there's a pipe between i and j, x_e is the length of the pipe, and c_e is the cross-sectional area of the pipe. Thus the volume of pipe e is $c_e x_e$, its cross-sectional area times its

length. For an example, see Figure 10.1. Note that the LP is to compute the pipe system with minimum total volume, where all $s_i - t_i$ pairs are at least 1 unit apart.

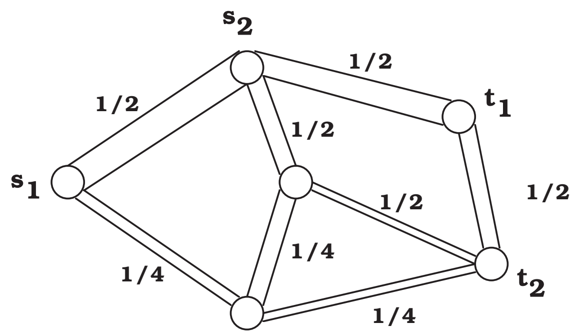


Figure 10.1: LP solution as a pipe system.

Given a feasible LP solution, x, let $dist_x(u, v)$ be the shortest path length between u and v with edge lengths x. Let $B_x(u, r) = \{v : dist_x(u, v) \leq r\}$ be the ball of radius r around u with the edge lengths x. See Figure 10.2 for an example.

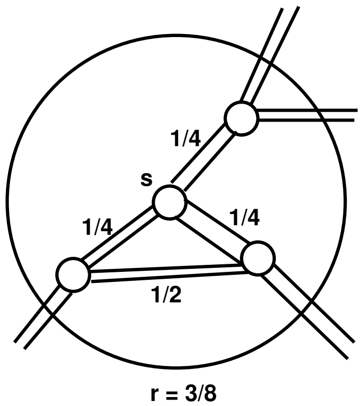


Figure 10.2: Example of a ball of radius 3/8.

This gives rise to a very simple approximation algorithm for the problem:

GVY

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

Two simple lemmas are immediate.

Lemma 10.3 The algorithm terminates.

Proof: The algorithm terminates because we disconnect at least one pair per iteration of the while loop; s_i and t_i are separated. □

Lemma 10.4 The algoritm returns a multicut.

Proof: The only difficulty in proving this statement is that in some iteration in which we separated s_i and t_i, we ended up with both s_j and t_j in S for some j. Then the algorithm would remove S from the graph and s_j and t_j might not be separated in the solution returned by the algorithm. Suppose this happens. Then by construction of the algorithm it follows that $dist_x(s_i, s_j) \leq r$ and $dist_x(s_i, t_j) \leq r$. This means that a path from s_j to s_i to t_j exists of length less than $2r$. But, since $r < 1/2$, that means there's a path between s_j and t_j of length less than 1, which violates the LP feasibility of x. Hence, this can't happen, and the algorithm returns a feasible solution to the LP. □

Next time, we'll prove that this algorithm is a good approximation algorithm.

Lecture 11

Lecturer: David P. Williamson *Scribe: Amit Kumar*

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 $(s_i, t_i), i = 1 \ldots k$
  - 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 $G' = (V, E - F)$.

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 can give a physical interpretation to a solution to the LP as a "pipe system" as follows.

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

- c_e = cross section area of pipe

With this interpretation, $\sum_e c_e x_e$ is the total volume of the pipe system. So, the objective is to minimize the total volume such that for all i, the distance between s_i and t_i is at least 1.

We defined $dist_x(u,v)$ be the distance from u to v given edge lengths x, $B_x(u,r) = \{v \in V : dist_x(u,v) \leq r\}$ for a solution to the LP.

Definition 11.1 For an optimal solution x to the above LP, define V^* as $\sum_{e \in E} c_e x_e$.

Clearly, $V^* \leq OPT$, where OPT denotes the value of optimal minimum multicut. We gave the following approximation algorithm for minimum multicut :

GVY

```
F ← ∅
Solve LP and get optimal solution x
While ∃ 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 δ(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. Fix some iteration of the algorithm in which the pair (s_i, t_i) is chosen. To prove the performance guarantee of the algorithm, define the following two quantities

Definition 11.2

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

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.

Let us observe some properties of these two functions. Clearly, $V_x(s_i,r)$ is an increasing function of r (as we increase r, the ball of radius r contains more volume). In fact this is piece-wise linear function, with possible discontinuities at those values of r for which there exists a vertex v such that $dist_x(s_i,v) = r$. So, if there doesn't exist a vertex $v \in V$ such that $dist_x(s_i,v) = r$, then $V_x(s_i,r)$ is differentiable at r. Moreover, from the definition of $V_x(s_i,r)$ and $C_x(s_i,r)$, it is clear that

$$\frac{d(V_x(s_i,r))}{dr} = C_x(s_i,r)$$

We will prove the following theorem.

Theorem 11.1

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

Furthermore, we can find such an r in polynomial time.

Intuitively, this theorem says that if we choose this particular value of r, then we can charge the cost of the cut $C_x(s_i,r)$, i.e., the set of edges removed, to the volume of the pipe system we remove. Let us make this formal by showing how the theorem leads to a good approximation algorithm for the minimum multicut problem.

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 our solution 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. □

Proof of Theorem 11.1: Relabel the vertices according to their distance from s_i, i.e., the vertices are relabeled as $s_i = v_1, v_2, \ldots, v_l$. Define $r_j = dist(s_i, v_j)$. So, $0 = r_1 \leq r_2 \leq \ldots \leq r_l = \frac{1}{2}$.

Suppose the statement of the theorem is not true. Then, for $r \in (r_j, r_{j+1})$ (recall that $V_x(s_i,r)$ is differentaible for this r), the fact that $C_x(s_i,r)$ equals the derivative of $V_x(s_i,r)$ implies that

$$\frac{1}{V_x(s_i,r)}\frac{d(V_x(s_i,r))}{dr} > 2\ln 2k$$

We now integrate both sides from r_j to r_{j+1} (actually it should be from r_j^+ to r_{j+1}^-). we get

$$
\begin{aligned}
\int_{r_j}^{r_{j+1}} \frac{1}{V_x(s_i,r)}\frac{d(V_x(s_i,r))}{dr}dr &> \int_{r_j}^{r_{j+1}} 2\ln 2k dr \\
\text{i.e.,}\quad \ln(V_x(s_i,r_{j+1})) - \ln(V_x(s_i,r_j)) &> 2(r_{j+1} - r_j)\ln 2k
\end{aligned}
$$

Now, we would like to add these terms for j varying from 1 to $l-1$ so that all but two terms on the LHS cancel out. But there is a caveat. We are ignoring the discontinuities of $V_x(s_i, r)$ at r_j for all j. But the fact that $V_x(s_i, r)$ is an increasing function implies that by doing this we will only underestimate the LHS of this telescopic sum. So, we do not get the wrong result if we sum the above equation from $j = 1$ to $l-1$. By summing the above equation from $j = 1$ to $l-1$ and cancelling terms, we get

$$\ln V_x(s_i, r_l) - \ln V_x(s_i, 0) > 2(r_l - 0) \ln 2k$$

We know that $r_l = \frac{1}{2}$ and $V_x(s_i, 0) = \frac{V^*}{k}$. Substituting these facts, we get

$$\ln V_x(s_i, \frac{1}{2}) > \ln\left(\frac{V^*}{k}\right) + \ln 2k$$

Removing logarithms on both sides, we get

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

which is a contradiction because $V_x(s_i, \frac{1}{2})$ is only a part of the total volume. Thus, the theorem must be true.

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 $v_1, v_2, \ldots, v_l$ 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. □

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 which 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 11.3 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 11.4 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).

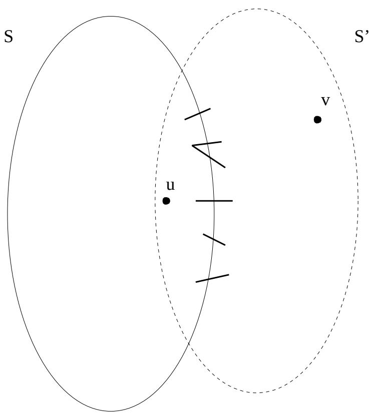


Figure 11.1: Dark lines represent edges whose variable is set to one.

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

When $u \notin S$, the argument is essentially the same (because there are at least $(\frac{2}{3} - \frac{1}{2})\, n$ vertices in $S' \cap S$). So, the solution given by $\bar{x}$ is in fact feasible and hence, $Z_{LP} \leq OPT(\frac{1}{2})$. $\square$

There is a problem we have ignored in all this – the LP could be quite large. If we can find a 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 $|S| \geq \frac{2}{3}n$. But note that $S_0 \subset S_1 \subset \ldots \subset S_{\frac{1}{3}n}$. So, it is sufficient to verify the constraint for S_0 only.

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
    $dist_x(u,v) \geq \left(\frac{2}{3} - \frac{1}{2}\right) = \frac{1}{6}$
  $C \leftarrow B_x(u, r)$, 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 $\sum_{v \in \bar{S}} \sum_{e \in P_{uv}} x_e \geq (\frac{2}{3} - \frac{1}{2})n = \frac{n}{6}, \forall\, u \in \bar{S},\ P_{uv} \in \mathcal{P}_{uv}$. But then the average path length from u $barS$ 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 $dist_x(u,v) \geq \frac{1}{6}$. $\square$

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 $dist_x(u,v) \geq \frac{1}{6}$. Since $C = B_x(u,r)$ 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,

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

$\square$

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, analogous to those we have seen before:

Definition 11.5

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

Definition 11.6

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

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 $f : V \to \{1, 2, \ldots, n\}$ which minimizes

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

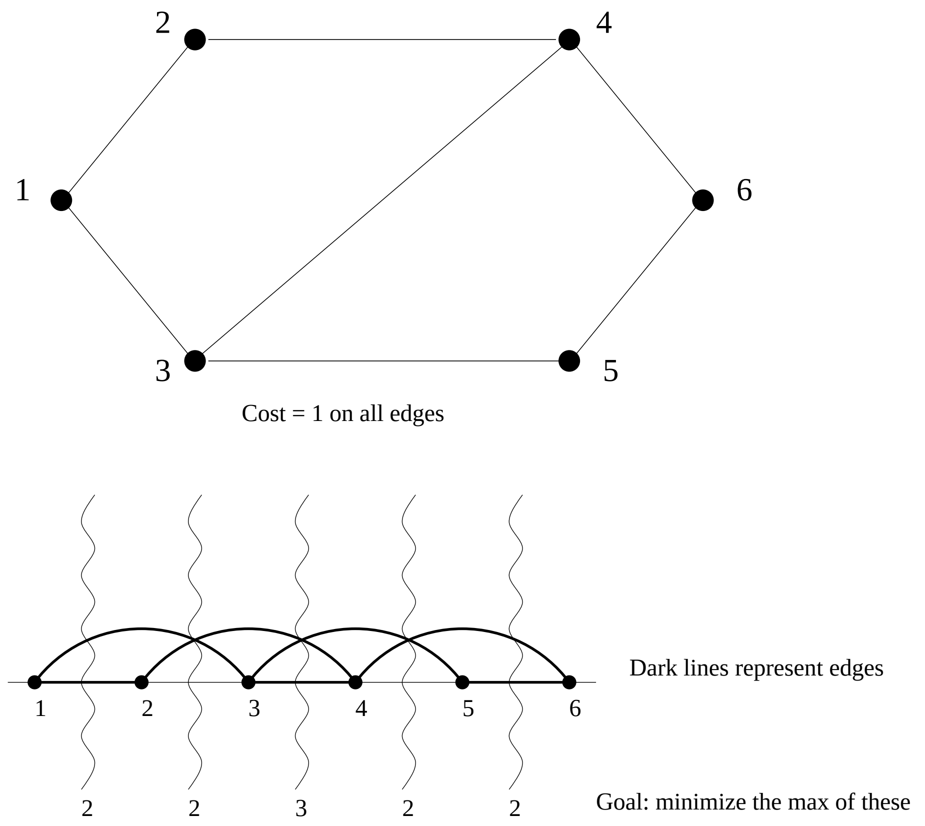


Figure 11.2: A graph and a representation of a corresponding linear arrangement.

As an example, consider Figure 11.2. The curly lines represent "$f(u) \leq i, f(v) > i$". 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 $OPT(\frac{1}{2}) \leq OPT_{LA}$, 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 $\frac{n}{2}$. Given the linear arrangement, let S be the set of vertices assigned to the numbers 1 through $n/2$. Then certainly $OPT(\frac{1}{2}) \leq \sum_{e \in \delta(S)} c_e$. Furthermore, by the definition of the linear arrangement problem, $OPT_{LA} \geq \sum_{e \in \delta(S)} c_e$. See Figure 11.3 for a "proof-by-picture".

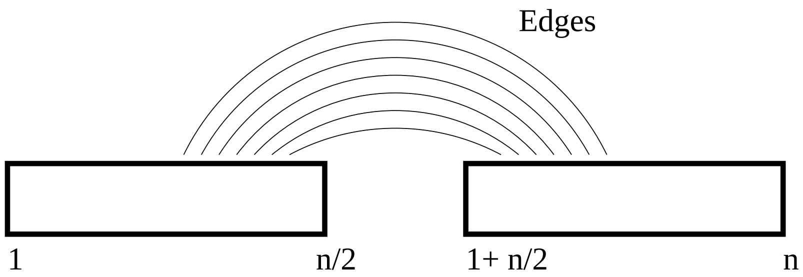


Figure 11.3: Getting a bisection from a linear arrangment

So then $OPT(\frac{1}{2}) \leq LA$, 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 algorithm below. The algorithm **LAYOUT** finds an $\frac{1}{3}$-balanced cut S, then recursively lays out vertices in S on the numbers 1 through $|S|$ and recursively lays out vertices in $V - S$ on the numbers $|S| + 1$ through $|V|$.

LAYOUT$(V, E, [i, i + |V| - 1])$

- **if** $V = \{v\}$ (i.e. singleton set)
  - $f(v) \leftarrow i_1$;
- **else**
  - Find a $\frac{1}{3}$-balanced cut S of (V, E)
  - **LAYOUT**$(S, E[S], [i, i + |S| - 1])$
  - **LAYOUT**$(V - S, E_[V - S], [i + |S|, i + |V| - 1])$

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.

Note : The best known algorithm for the linear arrangement problem is an $O(\log n \ \log\log n)-$approximation algorithm.

Proof:

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

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

We shall prove that

$$L(V, E) \leq c \log^2_{\frac{3}{2}} n \ OPT_{LA}$$

for a suitable constant c that we choose later. The proof is by induction on the size of G, i.e., n.

So, assume that the fact is true for all graphs of size less than n.

Let G be a graph on n vertices. Observe that

$$\begin{aligned}
(11.1) \quad L(V, E) &\leq \max\{L(S, E[S]), L(V - S, E[V - S])\} + B(V, E) \\
(11.2) \quad &\leq \max\{L(S, E[S]), L(V - S, E[V - S])\} + d \log_{\frac{3}{2}} n \ OPT_{LA}
\end{aligned}$$

where (11.1) follows from the fact that in the worst case, all the edges of the balanced cut appear in all the divisions of the layout. (11.2) follows from the fact that **ENRS** gives a $\frac{1}{3}-$balanced cut of value at most $(d \log_{\frac{3}{2}} n) OPT\left(\frac{1}{2}\right)$, which is at most $(d \log_{\frac{3}{2}} n) OPT_{LA}$ (using Lemma 11.10), for some value of d.

Since the number of vertices in the graphs $(S, E[S])$ and $(V - S, E[V - S])$ are at most $\frac{2n}{3}$ (follows from the definition of a $\frac{1}{3}-$balanced cut), our induction hypothesis implies that $L(S, E[S])$ and $L(V - S, E[V - S])$ are at most $c \log^2_{\frac{3}{2}} \left(\frac{2n}{3}\right) \ OPT_{LA}$. So, equation (11.2) becomes

$$\begin{aligned}
L(V, E) &\leq c \log^2_{\frac{3}{2}} \left(\frac{2n}{3}\right) \ OPT_{LA} + (d \log_{\frac{3}{2}} n) OPT_{LA} \\
&= c \left(\log_{\frac{3}{2}} n - 1\right)^2 OPT_{LA} + (d \log_{\frac{3}{2}} n) OPT_{LA}
\end{aligned}$$

If we choose $c > d$ and simplify, we get $L(V, E) \leq (c \log^2_{\frac{3}{2}} n) OPT_{LA}$, which proves the induction hypothesis. $\square$

ORIE 634 Approximation Algorithms	November 12, 1998
Lecture 12	
Lecturer: David B. Shmoys	*Scribe: Tim Roughgarden*

12.1 Uncapacitated Facility Location

Today, we will consider the uncapacitated facility location problem.

Uncapacitated Facility Location Problem (UFL)

- **Input:**
  - A set F of potential facility sites, with non-negative costs f_i $\forall i \in F$.
  - A set D of demand points, with cost c_{ij} of assigning demand point j to facility i (for all $i \in F, j \in D$).
  - Costs c_{ij} obey the triangle inequality.
- **Goal:** Find $S \subseteq F$ and assignment of demand points to facilities in S that minimizes the total cost.

12.1.1 An LP Lower Bound

Before proceeding, we make one observation. Suppose S is chosen to be the set of open facilities. Then the assignment of demand points to facilities is easy: simply assign each demand point to the nearest open facility. Thus, one way to model the problem is to choose a subset $S \subseteq F$ that minimizes $\sum_{i\in S} f_i + \sum_{j\in D} \min_{i\in S}\{c_{ij}\}$.

While this intuition will be useful later on, we now formalize the UFL more traditionally, by an integer program with a linear objective function and a polynomial number of variables:

$$\text{Min} \quad \sum_{i\in F} f_i y_i + \sum_{i\in F}\sum_{j\in D} c_{ij} x_{ij}$$

subject to:

$$\begin{array}{ll} \sum_{i\in F} x_{ij} = 1 & \forall j \in D \\ x_{ij} \le y_i & \forall i \in F, \forall j \in D \\ x_{ij} \in \{0,1\} & \forall i \in F, \forall j \in D \\ y_i \in \{0,1\} & \forall i \in F \end{array}$$

In this formulation, we have

$$y_i = \begin{cases} 1 & \text{if facility } i \text{ is opened} \\ 0 & \text{otherwise} \end{cases}$$

and

$$x_{ij} = \begin{cases} 1 & \text{if demand point } j \text{ is assigned to facility } i \\ 0 & \text{otherwise} \end{cases}$$

The first constraint of the integer program requires that every demand point be assigned, and the second requires that demand points are only assigned to opened facilities. We denote this integer program by (IP). We can relax this integer program to a linear program in the usual way, by replacing the last three constraints by $0 \le x_{ij} \le y_i$ for all $i \in F$ and $j \in D$. We denote this linear program by $(LP1)$. Since UFL is a minimization problem, we know that any optimal solution to $(LP1)$ gives a lower bound on the optimal integer solution.

12.1.2 A Second LP Lower Bound

Consider an optimal solution to $(LP1)$, say (x^*, y^*). As noted above, (x^*, y^*) gives us a lower bound on the optimal integer solution. In this section we investigate a second lower bound.

First, suppose that we ignore facility building costs entirely. Surely an optimal solution to the facility location problem with $f_i = 0$ for all $i \in F$ yields a lower bound on the original problem. In addition, the optimal solution is easy to state and compute: simply build all of the facilities and assign each demand point to the closest one, thus incurring a cost of $\sum_{j\in D} \min_{i\in F}\{c_{ij}\}$. However, this lower bound is too weak to be of any use. Can we do better?

Suppose that we eliminate our facility costs in a more clever fashion. More precisely, for each facility i, we define $|D|$ nonnegative costs $w_{i1}, \ldots, w_{i|D|}$ such that $\sum_j w_{ij} = f_i$. Intuitively, we wish to charge a portion of the facility building cost to each demand point. Of course, a demand point should only incur this extra cost if it actually uses the facility. Thus, in this new problem, there are no facility costs and assigning demand point j to facility i incurs a cost of $c_{ij} + w_{ij}$. We denote this problem by (LB).

We claim that, for any legal assignment of values to the w's, the value of an optimal solution to (LB) lower bounds the value of the optimal solution to UFL. Intuitively, in the UFL problem an entire facility must be paid for before it can be used, while in (LB) some demand points can be assigned to a facility while only paying for a fraction of it. More formally, consider an optimal solution to an instance of UFL, incurring a cost of $\sum_{i\in S} f_i + \sum_{j\in D} \min_{i\in S} c_{ij}$ for some $S \subseteq F$. Now consider the same assignments of demand points to facilities in corresponding instance of (LB).

As there are no facility costs in (LB), we can open all facilities in S at zero cost and achieve a feasible solution. Since $\sum_j w_{ij} = f_i$ for all i, we have

$$\begin{aligned}(IP)\text{ soln} &= \sum_{i\in S} f_i + \sum_{j\in D} \min_{i\in S} c_{ij} \\ &= \sum_{i\in S}\sum_{j\in D} w_{ij} + \sum_{j\in D} \min_{i\in S} c_{ij} \\ &\geq \sum_{i\in S}\sum_{j\in N(i)} w_{ij} + \sum_{j\in D} \min_{i\in S} c_{ij} \\ &= (LB)\text{ soln}\end{aligned}$$

where $N(i)$ denotes the set of demand points assigned to facility i in the solution to (IP). Hence, an optimal solution to (LB) will be a lower bound for (IP).

Notice that (LB) yields a lower bound for any legal choice of w's. We are of course interested in the sharpest possible lower bound, a problem that can be modeled by the following non-linear program:

$$\text{Max} \quad \sum_{j\in D} v_j$$

subject to:

$$\begin{aligned} v_j &= \min_{i\in F}\{c_{ij} + w_{ij}\} && \forall j \in D \\ \sum_{j\in D} w_{ij} &= f_i && \forall i \in F \\ w_{ij} &\geq 0 && \forall i \in F, j \in D \end{aligned}$$

Since we have a maximization problem, the following linear program is equivalent:

$$\text{Max} \quad \sum_{j\in D} v_j$$

subject to:

$$\begin{aligned} v_j &\leq c_{ij} + w_{ij} && \forall i \in F, \forall j \in D \\ \sum_{j\in D} w_{ij} &= f_i && \forall i \in F \\ w_{ij} &\geq 0 && \forall i \in F, j \in D \end{aligned}$$

We denote this linear program by $(LP2)$. Note that both $(LP1)$ and $(LP2)$ (one minimization problem, one maximization problem) yield lower bounds on UFL. Thus, the following facts may not come as a surprise.

Fact 12.1 $(LP1)$ and $(LP2)$ are duals of each other.

Fact 12.2 $(LP1)$ and $(LP2)$ have equal optimal values.

Fact 12.3 At optimality, both primal and dual complementary slackness conditions hold. That is, for optimal (x^*, y^*) and (v^*, w^*), $x^*_{ij} > 0$ implies that $v^*_j = c_{ij} + w^*_{ij}$ and $w^*_{ij} > 0$ implies that $x^*_{ij} = y^*_i$.

12.1.3 A General Decomposition Algorithm

In this section we present the main algorithmic tool on which all known approximation algorithm for UFL with a constant performance guarantee are based. Before introducing this decomposition algorithm (due to (Shmoys, Tardos, & Aardal, '97)), we first develop some motivation for it.

Lemma 12.1 If (x^*, y^*) and (v^*, w^*) are optimal primal and dual solutions, then $x^*_{ij} > 0$ implies that $c_{ij} \leq v^*_j$.

Proof: By complementary slackness $x^*_{ij} > 0$ implies that $v^*_j = c_{ij} + w^*_{ij}$. The lemma then follows from the fact that $w^*_{ij} \geq 0$. □

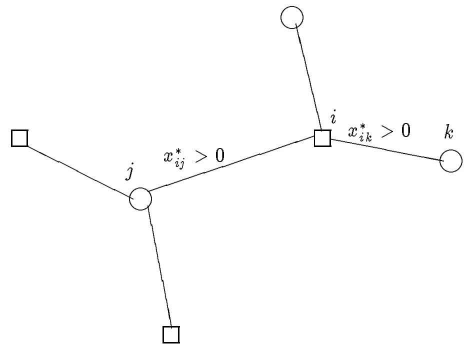
Consider optimal solutions to $(LP1)$ and $(LP2)$, say (x^*, y^*) and (v^*, w^*). We construct a bipartite graph G_{x^*} from x^* as follows. Let $G_{x^*} = (F \cup D, E)$, where $E = \{(i,j) : x^*_{ij} > 0\}$. We associate cost c_{ij} with each $(i,j) \in E$. Intuitively, G_{x^*} encodes a "neighboring relation" among demand points and facilities, as determined by the optimal LP solution. We make this relation more precise with the following definition.

Definition 12.1 Facility i is a *neighbor* of demand point j if $(i,j) \in E$. Similarly, demand point j is a *neighbor* of facility i if $(i,j) \in E$.

We will also make use of one additional definition.

Definition 12.2 A *cluster centered at* j is a set consisting of a demand point j, the facilities neighboring j, and the demand points neighboring j's neighbors.

Pictorially, the cluster centered at j might look as follows:



Observe that, by Lemma 12.1, edge (i,j) has cost at most v_j^*. Thus, if we open a set of facilities S such that each demand point has a neighbor in S, our total assignment costs are upper bounded by $\sum_j v_j^* \le OPT$. Unfortunately, if we restrict ourselves to sets of facilities with this property, we cannot guarantee a small total facility cost. However, we can design a good approximation algorithm by opening a set of facilities such that each demand point has some facility "close by" (though not necessarily a neighbor):

UFL-Decomp

> Solve $(LP1)$ to obtain optimal solution (x^*, y^*).
> $S \leftarrow D$.
> while $S \neq \emptyset$
>   Choose some $j \in S$.
>   Let $X = \{i \in F : x_{ij}^* > 0\}$.
>   Let $Y = \{j' \in S - \{j\} : \exists i \in X \text{ s.t. } x_{ij'}^* > 0\}$.
>   Choose some $i \in X$ and open i.
>   Assign all demand points in $\{j\} \cup Y$ to i.
> $S \leftarrow S - (\{j\} \cup Y)$.

Intuitively, the algorithm decomposes the demand points and facilities into clusters, opens one facility for each cluster, and assigns every demand point to the facility in its cluster. Observe that the clusters are disjoint, every demand point is contained in exactly one cluster, and every facility is contained in at most one cluster.

Two actions of the algorithm are unspecified: how to choose cluster centers and how to choose which facility to open in each cluster. By our earlier discussion, once we have decided the facilities to open, one merely assigns each demand point to its nearest open facility. We will obtain two approximation algorithms by considering different implementations of these actions.

12.1.4 A Deterministic 4-Approximation Algorithm

In this section, we present a 4-approximation algorithm for the uncapacitated facility problem, which is a simplification, due to (Chudak '98), of the original result of (Shmoys, Tardos, & Aardal '97). This algorithm relies on the following implementation rules for UFL-Decomp:

1. When selecting a new cluster center, choose the demand point j such that v_j^* is minimized.

2. When opening a facility in a cluster, open the facility i such that f_i is minimized.

We will call the first rule *center rule #1* and the second rule *facility rule #1*.

Let $\tilde{x}$ and $\tilde{y}$ denote the rounded (integral) solution produced by the decomposition algorithm with the above two rules. We analyze the worst-case facility and assignment costs separately, in the following two lemmas.

Lemma 12.2 $\sum_{i\in F} f_i \tilde{y}_i \le \sum_{i \in F} f_i y_i^*$.

Proof: Since the algorithm opens exactly one facility per cluster and the clusters are disjoint, it suffices to prove the lemma for a single cluster (summing over all clusters yields the more general result). Consider the cluster centered at j. Let X denote the facilities contained in this cluster. Using the first LP constraint and the fact that all of j's neighbors are contained in the cluster, we have $\sum_{i\in X} x_{ij}^* = 1$. Then, using our facility selection rule and the second LP constraint we have:

$$\begin{aligned}\sum_{i\in X} f_i \tilde{y}_i &= \min_{i \in X} f_i \\ &\le \sum_{i\in X} f_i x_{ij}^* \\ &\le \sum_{i \in X} f_i y_i^*\end{aligned}$$

□

Lemma 12.3 $\sum_{i\in F}\sum_{j\in D} c_{ij}\tilde{x}_{ij} \le 3\sum_{j\in D} v_j^*$.

Proof: We consider two types of demand points separately: cluster centers and non-centers. Since each cluster center j is assigned to a neighbor, by Lemma 12.1 we have $\sum_{i\in F} c_{ij}\tilde{x}_{ij} \le v_j^*$ (for each cluster center j). Now consider a non-center k, contained in a cluster centered at j. Let the facility opened in this cluster be facility i (and thus k was assigned to i in the rounded solution). By our definition of a cluster, two important properties hold. First, i and j are neighbors. Second, there is some facility ℓ such that j and ℓ are neighbors, and k and ℓ are neighbors (see Figure below). From the discussion of section 12.1.3, it follows that $c_{ij} \le v_j^*$, $c_{\ell j} \le v_j^*$, and $c_{\ell k} \le v_k^*$. Recalling that the costs obey the triangle inequality, we have $c_{ik} \le 2v_j^* + v_k^*$. Finally, since the algorithm chose j as a center instead of k we conclude that $v_j^* \le v_k^*$ and hence $c_{ik} \le 3v_k^*$.

Combining the bounds for centers and non-centers and summing over all demand points yields the lemma. □

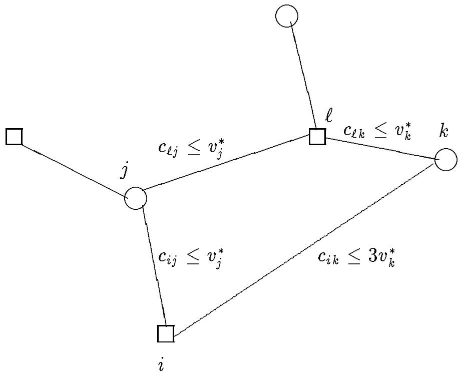


Figure: Bounding the assignment costs for non-centers.

Combining Lemmas 12.2 and 12.3 gives the desired result.

Theorem 12.4 (Chudak '98) UFL-Decomp with center rule #1 and facility rule #1 is a 4-approximation algorithm for UFL.

Proof: By Lemmas 12.2 and 12.3 we have:

$$\begin{aligned}\sum_{i\in F} f_i\tilde{y}_i + \sum_{i\in F}\sum_{j\in D} c_{ij}\tilde{x}_{ij} &\leq \sum_{i\in F} f_i y_i^* + 3\sum_{j\in D} v_j^* \\ &\leq 4\cdot OPT\end{aligned}$$

using the lower bounds provided by $(LP1)$ and $(LP2)$. □

12.1.5 A Randomized 3-Approximation Algorithm

In this section, we analyze another implementation of the general decomposition algorithm. Before proceeding, we define $C_j^* = \sum_{i\in F} c_{ij}x_{ij}^*$. That is, C_j^* is the total cost "paid by" the LP solution to (fractionally) assign demand point j. We then use the following two rules:

1. When selecting a new cluster center, choose the demand point j such that $v_j^* + C_j^*$ is minimized.
2. When opening a facility in a cluster centered at j, open a facility at random, choosing from the distribution defined by the the x_{ij}^*'s (i.e., open exactly one facility such that facility i is opened with probability x_{ij}^*).

We will call the first rule *center rule #2* and the second rule *facility rule #2*.

The following lemma is necessary to show that the algorithm is well-defined.

Lemma 12.5 In the cluster centered at j, the x_{ij}^*'s define a probability distribution.

Proof: From the LP constraints, $x_{ij}^* \geq 0$ for all $i \in F$ and $j \in D$. Letting X denote the set of j's neighbors, we have from the proof of Lemma 12.2 $\sum_{i\in X} x_{ij}^* = 1$. □

Let $\tilde{x}$ and $\tilde{y}$ denote the rounded (integral) solution produced by the decomposition algorithm with the above two rules. As before, we analyze the expected facility and assignment costs separately, in the following two lemmas.

Lemma 12.6 $E[\sum_{i\in F} f_i\tilde{y}_i] \leq \sum_{i\in F} f_i y_i^*$.

Proof: As in the proof of Lemma 12.6, it suffices to prove the result for an arbitrary cluster. Consider the cluster centered at j, and let X denote the facilities contained in this cluster. Then, using linearity of expectation, our facility selection rule, and the second LP constraint we have:

$$\begin{aligned}E[\sum_{i\in X} f_i\tilde{y}_i] &= \sum_{i\in X} f_i E[\tilde{y}_i] \\ &= \sum_{i\in X} f_i x_{ij}^* \\ &\leq \sum_{i\in X} f_i y_i^*\end{aligned}$$

□

Lemma 12.7 $E[\sum_{i\in F}\sum_{j\in D} c_{ij}\tilde{x}_{ij}] \leq 2\sum_{j\in D} v_j^* + \sum_{j\in D} C_j^*$.

Proof: As before, we consider cluster centers and non-centers separately. For a cluster center j, we have:

$$\begin{aligned}E[\sum_{i\in F} c_{ij}\tilde{x}_{ij}] &= \sum_{i\in F} c_{ij}E[\tilde{x}_{ij}] \\ &\leq \sum_{i\in F} c_{ij}x_{ij}^* \\ &= C_j^*\end{aligned}$$

For a non-center k in the cluster centered at j, we first observe that, as before, there exists a facility ℓ such that both j and k are neighbors of ℓ. Thus, $c_{\ell j} \leq v_j^*$ and $c_{k\ell} \leq v_k^*$. By the analysis above, we also know that the expected cost of assigning center j to facility i is at most C_j^*. Thus, by triangle inequality, the expected cost of assigning k to facility i is at most $C_j^* + v_j^* + v_k^*$. Finally, since the algorithm chose j as a center instead of k we conclude that $C_j^* + v_j^* \leq C_k^* + v_k^*$ and hence $E[\sum_{i \in F} c_{ik}\tilde{x}_{ik}] \leq 2v_k^* + C_j^*$.

Combining the bounds for centers and non-centers and summing over all demand points yields the lemma. □

Combining Lemmas 12.6 and 12.7 gives the desired result.

Theorem 12.8 UFL-Decomp with center rule #2 and facility rule #2 is a randomized 3-approximation algorithm for UFL.

Proof: By Lemmas 12.6 and 12.7 we have:

$$\begin{aligned} E[\sum_{i \in F} f_i \tilde{y}_i + \sum_{i \in F}\sum_{j \in D} c_{ij}\tilde{x}_{ij}] &\leq \sum_{i \in F} f_i y_i^* + \sum_j C_j^* + 2\sum_{j \in D} v_j^* \\ &\leq 3 \cdot OPT \end{aligned}$$

using the lower bounds provided by $(LP1)$ and $(LP2)$. □

As an aside, it may be surprising that we are designing and analyzing relatively complicated decomposition algorithms when the following simple scheme immediately springs to mind: solve the LP to obtain an optimal solution (x^*, y^*), employ traditional randomized rounding (i.e., build each facility i independently with probability y_i^*), and assign each demand point to the nearest open facility. In addition to its simplicity, this scheme has the advantage of assigning a constant fraction of the demand points to neighbors (note that the previous two algorithms cannot make this guarantee):

$$\begin{aligned} \Pr[\text{no neighbor of } j \text{ is open}] &= \prod_{i \in X}(1 - y_i^*) \\ &\leq \prod_{i \in X} e^{-y_i^*} \\ &= e^{-\sum_{i \in X} y_i^*} \\ &\leq \frac{1}{e} \end{aligned}$$

However, at the time of this writing, it is not known how to obtain good bounds on the expected cost incurred by demand points not assigned to neighbors. Thus, proving a good approximation ratio for this simple scheme remains an interesting open question.

12.1.6 A Randomized $1 + 3/e$-Approximation Algorithm

In this section we develop an algorithm that attempts to leverage as much of the analysis of the simple randomized rounding scheme as is possible while keeping the good worst-case assignment bounds enjoyed by the first two algorithms. In what follows, we shall argue that the optimal LP solution can be assumed to have an additional structural property; this additional property simplifies the proofs, but is not essential to the guarantees obtained.

Definition 12.3 An optimal solution to $(LP1)$, (x^*, y^*), is said to be *complete* if whenever $x_{ij}^* > 0$ we have $x_{ij}^* = y_i^*$.

Lemma 12.9 For every instance of UFL, there exists an equivalent, polynomial-size instance for which the optimal solution is complete. Moreover, such an instance can be constructed in polynomial time.

Proof: We omit all but the following hint. Suppose that there is exactly one i, j pair such that $0 < x_{ij}^* < y_i^*$, replace facility i by two new facilities, i_1 and i_2. Let $y_{i_1}^* = x_{ij}^*$, $y_{i_2}^* = y_i^* - x_{ij}^*$, $x_{i_1 j}^* = x_{ij}^*$, $x_{i_2 j}^* = 0$ and for all k originally assigned to i let $x_{i_1 k}^* = y_{i_1}^*$ and $x_{i_2 k}^* = y_{i_2}^*$. In the general case (when there are more pairs i, j, for which $0 < x_{ij}^* < y_i^*$) one can repeat this argument in much the same way to obtain an input for which the resulting LP optimum is complete. □

We now present the new algorithm. Recall that in our general decomposition algorithm, while each demand point is contained in exactly one cluster, each facility is contained in *at most* one cluster. An important change in the new algorithm is that, in addition to building one facility per cluster, we may also open facilities not contained in any cluster (thus improving the probability that a demand point gets assigned to a neighbor).

UFL-Round2

> Solve $(LP1)$ to obtain optimal solution (x^*, y^*).
> Modify (x^*, y^*) so that it is complete.
> $S \leftarrow D$.
> $T \leftarrow F$.
> while $S \neq \emptyset$
>   Choose some $j \in S$.
>   Let $X = \{i \in F : x_{ij}^* > 0\}$.
>   Let $Y = \{j' \in S - \{j\} : \exists i \in X \text{ s.t. } x_{ij'}^* > 0\}$.
>   Choose some $i \in X$ and open i.
> $S \leftarrow S - (\{j\} \cup Y)$.
> $T \leftarrow T - X$.
> For each $i \in T$, open i (independently) with probability y_i^*.
> Assign each demand point to the nearest open facility.

In this section, we will use center rule #1 (see Section 12.1.4) and facility rule #2 (see Section 12.1.5).

Our analysis of the algorithm differs from that of the last two sections. In the last two algorithms, our worst-case analysis assumed that possibly all demand points were not assigned to neighbors, and we focused on bounding the worst-case assignment cost for non-center demand points. Here, we will argue that a significant portion of the demand points are in fact assigned to neighbors (much as we did in our aside on simple randomized rounding), and that those not assigned to neighbors will not incur too much assignment cost (which we were unable to claim in the case of simple randomized rounding). We now proceed more formally, letting $(\tilde{x}, \tilde{y})$ denote the (rounded) output of the algorithm.

Lemma 12.10 For every facility i, $\Pr[\tilde{y}_i = 1] = y_i^*$.

Proof: Consider an arbitrary facility, i. If i is not in any cluster then i is opened with probability y_i^* by definition of the algorithm. Otherwise, i is in exactly one cluster, say the cluster centered at j. Then, i is opened with probability x_{ij}^*. The lemma follows from the fact that (x^*, y^*) is complete. □

Lemma 12.11 $E[\sum_{i \in F} f_i \tilde{y}_i] = \sum_{i \in F} f_i y_i^*$.

Proof: By linearity of expectation and Lemma 12.10 we have:

$$\begin{aligned} E[\sum_{i \in X} f_i \tilde{y}_i] &= \sum_{i \in X} f_i E[\tilde{y}_i] \\ &= \sum_{i \in X} f_i y_i^* \end{aligned}$$

□

In analyzing the assignment cost incurred, we would like to mimic the analysis for the case of simple randomized rounding. However, in UFL-Round2 facilities in the same cluster are not opened independently of each other, and our analysis will thus be a bit more complicated. To gain some intuition, consider one demand point j, and focus on the nature of the dependency that exists between the events that two neighbors of j, i_1 and i_2, have facilities opened. If i_1 and i_2 are in different clusters of the decomposition, these are independent events. However, if they are in the same cluster, then they are dependent. But now for the good news: the conditioning between these two events only works to our advantage, since if i_1 is not opened, that only increases the probability that i_2 has been opened.

Next we introduce some notation to facilitate the analysis of the expected assignment cost incurred by demand point j. Let X denote the neighbors of j, and group them according to the cluster in which they belong; that is, partition X into $X_1 \cup ... \cup X_r$, where each facility i not belonging to any cluster is one part of this partition (i.e., a singleton set), and each other part X_i consists of the intersection of X with one cluster of the decomposition.

Lemma 12.12 For any demand point j, the probability that a facility is opened at none of its neighbors is at most $1/e$.

Proof: Let O_ℓ denote the event that some facility in X_ℓ is opened by the algorithm, $\ell = 1, \ldots, r$. By the discussion above, these are independent events. Furthermore, $Pr[O_\ell] = \sum_{i \in X_\ell} y_i^*$; let Y_ℓ^* denote this value. Clearly, $\sum_{\ell=1}^r Y_\ell^* = 1$. Hence, by the same estimate as was used for the simple randomized rounding algorithm, the probability that no neighbor is opened is $\prod_{\ell=1}^r (1 - Y_\ell^*) \leq e^{-1}$. □

Lemma 12.13 For each demand point $j \in D$, its expected assignment cost, $E[\sum_{j \in D} c_{ij} \tilde{x}_{ij}] \leq C_j^* + \frac{3}{e} v_j^*$.

Proof: For any integer solution $\tilde{y}$, we can upper bound the corresponding assignment cost for j in the following somewhat unnatural way. For *each* neighbor i of j for which $\tilde{y}_i = 1$, we incur a cost of c_{ij} and, if no neighbors of j are opened (that is, $\tilde{y}_i = 0$ for each $i \in X$), then j can be assigned to the open facility its cluster, and so we incur an assignment cost of at most $3v_j^*$ (by the same argument as in Lemma 12.3). So, if $\tilde{Y}$ is an indicator variable that denotes whether a neighbor of j is opened (i.e., $\tilde{Y} = \max_{i \in X} \tilde{y}_i$), then

$$\sum_{i \in X} c_{ij} \tilde{y}_i + (1 - \tilde{Y}) 3 v_j^*$$

an upper bound on the assignment cost of j. Computing the expectation of this upper bound is simple: using linearity of expectation, the facts that $E[\tilde{y}_i] = y_i^*$ and $C_j^* = \sum_{i \in X} c_{ij} y_i^*$, as well as Lemma 12.12, we get the bound claimed in the lemma. □

Combining Lemmas 12.11 and 12.13 gives the desired result.

Theorem 12.14 (Chudak '98, Chudak & Shmoys '98) UFL-Round2 with center rule #1 and facility rule #2 is a randomized $1 + 3/e$ (≈ 2.104)-approximation algorithm for UFL.

Proof: By linearity of expectation together with Lemmas 12.11 and 12.13 we have:

$$\begin{aligned} E[\sum_{i \in F} f_i \tilde{y}_i + \sum_{i \in F} \sum_{j \in D} c_{ij} \tilde{x}_{ij}] &\leq \sum_{i \in F} f_i y_i^* + \sum_{j \in D} C_j^* + \frac{3}{e} \sum_{j \in D} v_j^* \\ &\leq \left(1 + \frac{3}{e}\right) \cdot OPT \end{aligned}$$

using the lower bounds provided by $(LP1)$ and $(LP2)$. □

Remark 12.1 Much as our 4-approximation algorithm could be improved to a 3-approximation algorithm, it can be shown that UFL-Round2 with center rule #2 is a $1 + \frac{2}{e}$ (≈ 1.736)-approximation algorithm for UFL (currently the best published result).

Remark 12.2 All of the randomized algorithms presented in this lecture can be derandomized using the method of conditional expectations. In fact, the 3-approximation algorithm can be derandomized by a more direct (greedy-like) method.

Finally, the following theorem provides the sharpest hardness of approximation result currently known for UFL.

Theorem 12.15 (Guha, Khuller '98, Sviridenko '98) There is no 1.427-approximation algorithm for UFL unless $P = NP$.

ORIE 634 Approximation Algorithms	November 19, 1998
Lecture 13	
Lecturer: David P. Williamson	*Scribe: Nathan Edwards*

13.1 Jain's Technique

We shall not cease from exploration
And the end of all our exploring
Will be to arrive where we started
And know the place for the first time.
– T. S. Eliot, "Four Quartets"

For our last full lecture, we in essence return to a technique that we saw in our first full lecture; namely, approximation algorithms via the deterministic rounding of linear programming relaxations. We saw with the algorithm **Round** for the Set Cover problem that we could set up a linear programming relaxation for the problem, and round up every variable greater than a certain value $1/f$ to obtain an f-approximation algorithm for the problem. Today we return to a much more sophisticated use of the same technique, due to Jain.

13.1.1 The Survivable Network Design Problem

To illustrate this technique, we will consider the following problem, which is a generalization of the generalized Steiner tree problem we saw earlier.

Survivable Network Design Problem (SNDP)

- **Input:**
  - Undirected graph $G = (V, E)$.
  - Edge costs $c_e \geq 0$, for each $e \in E$.
  - Request counts $r(u, v)$, for each $u, v \in V, u \neq v$.
- **Goal:** Find $F \subseteq E$ of minimum cost, such that $G' = (V, F)$ contains at least $r(u, v)$ edge-disjoint paths between u and v, for each $u, v \in V, u \neq v$.

There had been substantial work on this problem prior to Jain's result. Define $k = \max_{u,v} r(u, v)$. A $2k$-approximation algorithm due to W, Goemans, Mihail and Vazirani appeared in 1993 (conference version) and 1995 (journal version). This was improved to a $2H_k$- approximation algorithm due to Goemans, Goldberg, Plotkin, Shmoys, Tardos and W in 1994. We will see today a 2-approximation algorithm due to Jain, a graduate student at Georgia Tech, that appeared in January 1998.

The first two results use the primal-dual method in stages, taking care of one of the edge-disjoint paths required between each u and v per stage. Jain's result uses an entirely different proof strategy.

13.1.2 The Model

All three results use a relaxation of a certain integer programming formulation of the problem. First, we define $f(S) = \max_{u \in S, v \notin S} r(u, v)$ and $\delta(S)$ to be the edges across the cut S. Then we can formulate the survivable network design problem as the following integer program:

$$\begin{array}{lll} \text{Min} & \sum_{e \in E} c_e x_e & \\ \text{subject to:} & & \\ & \sum_{e \in \delta(S)} x_e \geq f(S) & \text{for each } S \subseteq V, \\ & x_e \in \{0, 1\} & \text{for each } e \in E. \end{array}$$

A maximum flow-minimum cut argument shows that this models the surviviable network design problem.

As usual, we will relax this integer program to a linear program, replacing $x_e \in \{0, 1\}$ with $0 \leq x_e \leq 1$. The resulting linear program has exponentially many constraints, but we can construct a polynomial time separation oracle for it. The oracle computes the maximum flow between each u, v pair in turn using the current solution, $x_e, e \in E$, as arc capacities. If the maximum flow between some pair u, v is less than $r(u, v)$, then the corresponding minimum cut S defines a violated constraint in our linear program.

We'll use the shorthand notation $x(F) = \sum_{e \in F} x_e$ throughout, in particular, notice that the constraints above become $x(\delta(S)) \geq f(S)$.

13.1.3 Weak Supermodularity

The function f we defined has some particularly useful properties.

Definition 13.1 A function $g : 2^V \longrightarrow \mathbf{N}$ is weakly supermodular if $g(V) = 0$ and for each $A, B \subseteq V$, one of the following holds:

(i) $g(A)+g(B) \le g(A \cup B)+g(A \cap B)$,

(ii) $g(A)+g(B) \le g(A-B)+g(B-A)$.

Claim 13.1 f is weakly supermodular.

The following result is the cornerstone of Jain's 2-approximation algorithm.

Theorem 13.2 [Jain 1998] For any weakly supermodular f and any extreme point x of our linear program, there exists $e \in E$ such that $x_e \ge 1/2$.

We first show how this theorem implies a 2-approximation algorithm for the survivable network design problem, and then turn to the proof of the theorem.

13.1.4 Jain's Algorithm

Given this theorem, we can obtain a 2-approximation algorithm for our problem in only a slightly more complicated fashion than in the **Round** algorithm for set cover. The basic idea is that we can solve the LP, obtain an extreme point, find some edge of value at least 1/2, round it up to 1 (thereby losing a factor of 2 for that edge), then recurse on a subproblem with a modified function f and the edge removed from the graph. We formalize this idea in the following algorithm.

Jain

> $f_0(S) \leftarrow f(S)$ for each $S \subseteq V$
> $i \leftarrow 0$
> **while** $\cup_{j=1}^{i} F_j$ is not a feasible solution
>   Solve the linear program on the graph $(V, E - \bigcup_{j=1}^{i} F_j)$ using the
>   function f_i to obtain an extreme point solution x^*
>   **for each** $e \in E$
>   **if** $x_e^* \ge 1/2$
> $F_{i+1} \leftarrow F_i \cup \{e\}$
> $f_{i+1}(S) \leftarrow f(S) - |\delta(S) \cap \bigcup_{j=1}^{i+1} F_j|$ for each $S \subseteq V$
> $i \leftarrow i+1$

To show that the algorithm works and that we can apply Jain's theorem, we need to show two things: first, that we can solve the LP in every iteration, and secondly, that the function f_i is in fact weakly supermodular every iteration. For the survivable network design problem the first is easy, via the following separation oracle: for every edge in $\bigcup_{j=1}^{i} F$ we install an edge of capacity 1, and again do a maxflow computation between every u, v pair in the graph G. We now show the second.

Lemma 13.3 For any $F \subseteq E$, $f_i(S) = f(S) - |\delta(S) \cap F|$ for each $S \subseteq V$, is weakly supermodular.

Proof: Define $z_e = \begin{cases} 1 & \text{if } e \in F, \\ 0 & \text{otherwise;} \end{cases}$ and $z(F') = \sum_{e \in F'} z_e$ for $F' \subseteq E$. We will show that for any such z,

$$z(\delta(A)) + z(\delta(B)) \ge z(\delta(A \cup B)) + z(\delta(A \cap B))$$

and

$$z(\delta(A)) + z(\delta(B)) \ge z(\delta(A - B)) + z(\delta(B - A)),$$

for all $A, B \subseteq V$.

We claim that this implies the lemma. Notice that $f_i(S) = f(S) - z(\delta(S))$ for each $S \subseteq V$. Then suppose that for the pair $A, B \subseteq V$, condition (i) of weak supermodularity holds for f. Then

$$\begin{aligned} f_i(A) + f_i(B) &= f(A) - z(\delta(A)) + f(B) - z(\delta(B)) \\ &\le f(A \cup B) - z(\delta(A \cup B)) + f(A \cap B) - z(\delta(A \cap B)) \\ &= f_i(A \cup B) + f_i(A \cap B). \end{aligned}$$

which shows that condition (i) holds for f_i. On the other hand, if condition (ii) of weak supermodularity holds for f and the pair A, B, a similar argument shows condition (ii) holds for f_i. Hence f_i is weakly supermodular.

To show these conditions on z we use a simple case analysis on the edges of F. It is possible to show that every edge that contributes z_e to the right-hand side of the inequalities will also contribute z_e to the left-hand side. For example, if edge $e \in F$ has one end point in $A - B$ and the other end point in $\overline{A \cup B}$, then e contributes z_e to $z(\delta(A))$ and $z(\delta(A \cup B))$; and to $z(\delta(A))$ and $z(\delta(A - B))$. Notice that only one type of edge contributes unequally to each side of these conditions on z: an edge e with one end point in $A - B$ and the other in $B - A$ contributes $2z_e$ to the left-hand side of first condition, but zero to the right-hand side. Similarly, an edge e from $A \cap B$ to $\overline{A \cup B}$ contributes $2z_e$ to the left-hand side of the second condition, but zero to the right-hand side. $\square$

We can now show the following.

Theorem 13.4 The algorithm Jain is a 2-approximation algorithm for any such integer program with weakly supermodular functions f_i such that we can solve the associated linear programs.

Proof: The proof proceeds by induction on the number of iterations in the while loop of the algorithm Jain.

We want to show that the cost of the edges we choose in all iterations is no more than twice the linear program optimal value. That is:

$$\sum_{\substack{e \in F_i \\ i \geq 1}} c_e \leq 2 \sum_{e \in E} c_e x_e^0$$

where x^0 is the solution of the linear program - the solution at the zeroth iteration. More generally, define x^i as the solution of the LP with function f_i.

Base case: Suppose that exactly one iteration of the while loop is necessary to achieve feasibility. Then

$$\sum_{e \in F_1} c_e \leq 2 \sum_{e \in E} c_e x_e^0$$

since the solution F_1 involves rounding up exactly those $x_e^0 \geq 1/2$.

Inductive step: The inductive assumption is:

$$\sum_{\substack{e \in F_i \\ i \geq 2}} c_e \leq 2 \sum_{e \in E - F_1} c_e x_e^1.$$

By similar reasoning as in the base case above, we know that

$$\sum_{e \in F_1} c_e \leq 2 \sum_{e \in F_1} c_e x_e^0.$$

The crucial observation is that x^0 is a feasible solution to our linear program with weakly supermodular function f_1 on the graph $(V, E - F_1)$. Notice that

$$\sum_{e \in \delta(S)} x_e^0 \geq f(S).$$

Therefore

$$\sum_{e \in \delta(S) - F_1} x_e^0 \geq f(S) - |\delta(S) \cap F_1| = f_1(S).$$

Since x^0 satisfies the constraints for our LP with function f_1 on the graph $(V, E - F_1)$ and x^1 is the optimal solution to this LP, we have

$$\sum_{e \in E - F_1} c_e x_e^1 \leq \sum_{e \in E - F_1} c_e x_e^0.$$

Putting this all together, we have

$$\sum_{\substack{e \in F_i \\ i \geq 1}} c_e \leq 2 \sum_{e \in F_1} c_e x_e^0 + 2 \sum_{e \in E - F_1} c_e x_e^1 \leq 2 \sum_{e \in F_1} c_e x_e^0 + 2 \sum_{e \in E - F_1} c_e x_e^0 \leq 2 \sum_{e \in E} c_e x_e^0.$$

$\square$

13.1.5 Proof of Theorem 13.2

Throughout this section we'll assume that an extreme point solution x satisfies $0 < x_e < 1$ for each $e \in E$, since if $x_e = 1$ for some $e \in E$, then we are done, and if $x_e = 0$ we can remove e from E without loss of generality.

Define m to be the number of edges in the graph, or equivalently, the number of fractional variables in our linear programming extreme point solution. We need the following definitions for our proof.

Definition 13.2 The sets $A, B \subseteq V$ are said to cross if $A - B$, $B - A$ and $A \cap B$ are non-empty.

Definition 13.3 A collection $\mathcal{L}$ of sets is laminar if no pair of sets in $\mathcal{L}$ cross.

Definition 13.4 A set $S \subseteq V$ is tight if $x(\delta(S)) = f(S)$.

Definition 13.5 The edge incidence vector $\chi_F \in \{0,1\}^m$ for $F \subseteq E$ is defined componentwise as $\chi_F(e) = \begin{cases} 1 & \text{if } e \in F, \\ 0 & \text{otherwise.} \end{cases}$

Theorem 13.5 For an extreme point solution x to our linear program, there exists a collection $\mathcal{L}$ of m sets such that

1. S is tight for each $S \in \mathcal{L}$.
2. The set of vectors $\{\chi_{\delta(S)} : S \in \mathcal{L}\}$ is linearly independent.
3. $\mathcal{L}$ is laminar.

Note that our linear program has $\chi_{\delta(S)}^T x \geq f(S)$ as constraints, that is, $\chi_{\delta(S)}$ is a row of our constraint matrix for each $S \subseteq V$. Clearly, then, since x is an extreme point of our linear program, it follows that there exists a collection $\mathcal{L}$ of m sets such that Conditions 1 and 2 of Theorem 13.5 hold. To show that there exists $\mathcal{L}$ such that Condition 3 holds, we'll need the following lemma.

Lemma 13.6 If A, B cross and are tight, then either

(i) $A \cup B$ and $A \cap B$ are tight and $\chi_{\delta(A)} + \chi_{\delta(B)} = \chi_{\delta(A \cup B)} + \chi_{\delta(A \cap B)}$,

or (ii) $A - B$ and $B - A$ are tight and $\chi_{\delta(A)} + \chi_{\delta(B)} = \chi_{\delta(A - B)} + \chi_{\delta(B - A)}$.

Proof: f is weakly supermodular, and suppose that for A, B, we have that

$$f(A) + f(B) \leq f(A \cup B) + f(A \cap B).$$

However,

$$\begin{aligned}
&(13.1) \qquad & f(A) + f(B) &= x(\delta(A)) + x(\delta(B)) \\
&(13.2) \qquad & &\geq x(\delta(A \cup B)) + x(\delta(A \cap B)) \\
&(13.3) \qquad & &\geq f(A \cup B) + f(A \cap B),
\end{aligned}$$

where (13.1) follows since A and B are tight, (13.2) follows from the proof of Lemma 13.3, and (13.3) follows since x is feasible. Therefore, by the feasibility of x, $x(\delta(A \cup B)) = f(A \cup B)$ and $x(\delta(A \cap B)) = f(A \cap B)$.

To show the second part of condition (i), we recall the proof of Lemma 13.3. In that analysis, there was only one type of edge that didn't contribute to both sides of the inequality equally. However, we just showed that

$$x(\delta(A)) + x(\delta(B)) = x(\delta(A \cup B)) + x(\delta(A \cap B))$$

which indicates that any of these exceptional edges e must have $x_e = 0$. Since we discarded all edges with $x_e = 0$, we must have no exceptional edges in E. Therefore, the second part of condition (i) holds.

The proof is similar if the other condition of weak supermodularity holds for f and A, B. □

Proof of Theorem 13.5: By the properties of basic (feasible) solutions there exists a family $\mathcal{T}$ of m sets satisfying conditions 1 and 2 of Theorem 13.5. Let span($\mathcal{T}$) be the span of the vectors $\{\chi_{\delta(S)} : S \in \mathcal{T}\}$. Note that span$(\mathcal{T}) = \Re^m$. Let $\mathcal{L}$ be a maximal collection of the sets in $\mathcal{T}$ that satisfy Conditions 1, 2 and 3 of Theorem 13.5.

If $|\mathcal{L}| = m$, we are done. Otherwise, we can choose a tight set S such that $\chi_{\delta(S)} \in$ span$(\mathcal{T})$ but $\chi_{\delta(S)} \notin$ span$(\mathcal{L})$ and such that no other such tight set crosses fewer members of $\mathcal{L}$. Note that S must cross at least one set in $\mathcal{L}$, or it could be added to $\mathcal{T}$ without violating Conditions 1, 2 and 3 of Theorem 13.5. We obtain a proof by contradiction by obtaining another set U such that $\chi_{\delta(U)} \in$ span$(\mathcal{T})$, $\chi_{\delta(U)} \notin$ span$(\mathcal{L})$, and such that U crosses fewer members of $\mathcal{L}$ than T.

Pick $T \in \mathcal{L}$ such that S and T cross, and apply Lemma 13.6. Suppose that for the pair S, T we have that $S - T$ and $T - S$ are tight and

$$\chi_{\delta(S)} + \chi_{\delta(T)} = \chi_{\delta(S-T)} + \chi_{\delta(T-S)}.$$

Then it can't be the case that both $\chi_{\delta(S-T)}$ and $\chi_{\delta(T-S)}$ are in span($\mathcal{L}$) since $\chi_{\delta(S)} \notin$ span$(\mathcal{L})$. On the other hand, for the pair S, T, we may have the other alternative from Lemma 13.6, with $S \cup T$ and $S \cap T$ tight and

$$\chi_{\delta(S)} + \chi_{\delta(T)} = \chi_{\delta(S \cup T)} + \chi_{\delta(S \cap T)}.$$

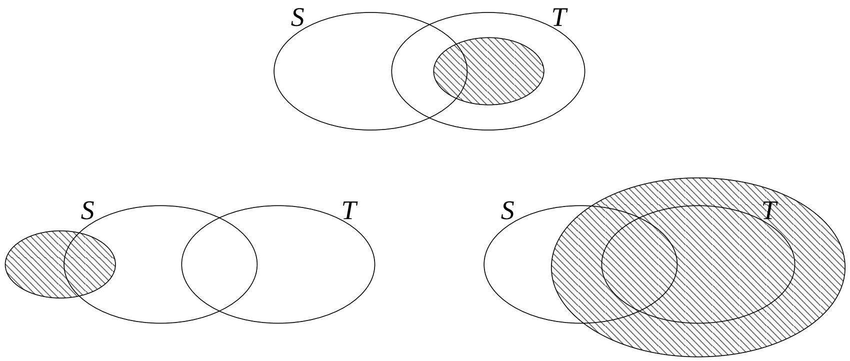


Figure 13.1: Sets of $\mathcal{L}$ that don't cross T but do cross one of $S - T$, $T - S$, $S \cup T$ or $S \cap T$.

In this case, at least one of $\chi_{\delta(S \cup T)}$ and $\chi_{\delta(S \cap T)}$ are not in span($\mathcal{L}$). Therefore, at least one of $S - T$, $T - S$, $S \cup T$ and $S \cap T$ is not in span($\mathcal{L}$).

We'll show that $S - T$, $T - S$, $S \cup T$ and $S \cap T$ all cross fewer sets in $\mathcal{L}$ than S, and hence we will have a set U such that $\chi_{\delta(U)} \in$ span$(\mathcal{T})$, $\chi_{\delta(U)} \notin$ span$(\mathcal{L})$, with U crossing fewer members of $\mathcal{L}$ than T.

Any set crossing $S - T$, $T - S$, $S \cup T$ or $S \cap T$ but not T must also cross S. See Figure 13.1 for a schematic of the possible sets in $\mathcal{L}$ that must be considered. Note too that S crosses T, but none of $S - T$, $T - S$, $S \cup T$ or $S \cap T$ do. Therefore, each of $S - T$, $T - S$, $S \cup T$ and $S \cap T$ crosses strictly fewer sets of $\mathcal{L}$ than S. □

Observation: $f(S) \geq 1$ for each $S \in \mathcal{L}$ since S is tight, $\chi_{\delta(S)} \neq 0$, all edges such that $x_e = 0$ have been discarded, and $f(S)$ is an integer.

Finally, we can state the theorem that will complete our proof of Theorem 13.2. Note however, that instead of showing that there exists an edge e such that $x_e \geq 1/2$, we'll show there exists an edge e such that $x_e \geq 1/3$. This will make the proof of Theorem 13.7 simpler, but won't change anything else of substance.

Theorem 13.7 There exists $S \in \mathcal{L}$ such that $|\delta(S)| \leq 3$.

Note that, in combination with the observation above, this proves (our slightly weaker version of) Theorem 13.2. We will need the following definition regarding the edges and vertices of G.

Definition 13.6 A socket is an edge-vertex pair (e, v) such that v is one of the two endpoints of e.

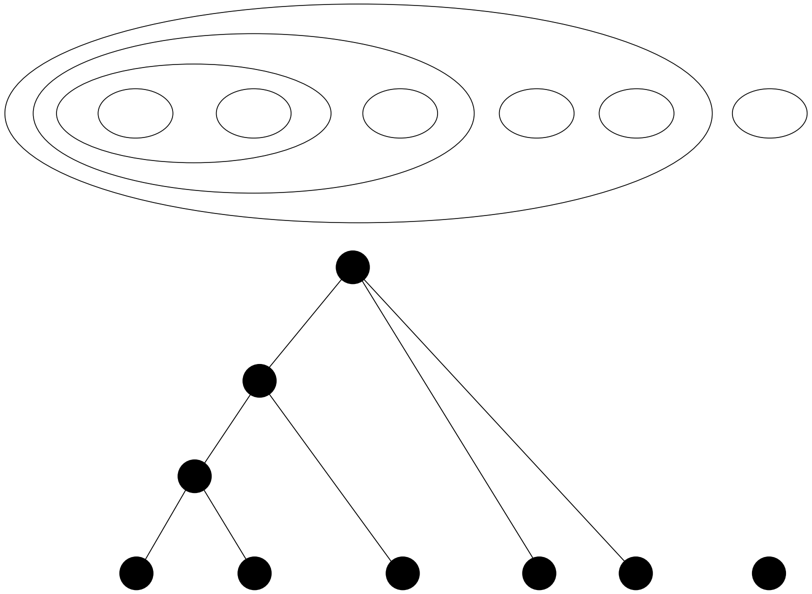


Figure 13.2: Example tree structure for a collection of laminar sets.

Observe that since the graph has m edges, it has $2m$ distinct sockets.

We'll consider the tree defined on the elements on $\mathcal{L}$ by the parent relation: for each $S \in \mathcal{L}$, Parent(S) = the smallest set $T \in \mathcal{L}$ that strictly contains S. See Figure 13.2 for an example.

Proof of Theorem 13.7: Suppose that the theorem is false. Then all $S \in \mathcal{L}$ has $|\delta(S)| \geq 4$. We'll use this to charge sockets uniquely to the elements of $\mathcal{L}$ so that each element of $\mathcal{L}$ is charged 2 sockets except for the root of the tree, which is charged 4 sockets. Since $\mathcal{L}$ has m elements, we have $2m + 2$ sockets, a contradiction, and the theorem is proved.

We'll show that we can charge sockets as specified above by induction on the tree from the leaves on up.

Base case: We charge 4 sockets to each of the leaves of the tree. We can do this because of our assumption that the theorem is false, and each $S \in \mathcal{L}$ has $|\delta(S)| \geq 4$. Thus each such S is associated with four sockets for the edges in $\delta(S)$ and the endpoint of each edge in $\delta(S)$ that S contains.

Inductive step: For any parent, assume inductively that in each child's subtree, 2

sockets are charged per non-root node and 4 sockets are charged to the root.

If there are 2 or more children, then each child can pass a charge of 2 to the parent.

If there is exactly one child, but the parent has at least two additional sockets then the child can pass a charge of 2 to the parent and the inductive assumption will be satisfied.

If the parent P has exactly one child C and no additional sockets, then $\delta(P) = \delta(C)$ and hence $\chi_{\delta(P)} = \chi_{\delta(C)}$ and the elements of $\mathcal{L}$ are no longer linearly independent. Therefore, this case cannot happen.

If the parent P has exactly one child C and exactly one additional socket representing edge e, then since P and C are tight either $f(P) = f(C) - x_e$ or $f(P) = f(C) + x_e$. In either case, we have a contradiction since $0 < x_e < 1$ and f is integer-valued. Therefore, this case cannot happen.

Thus we can, by induction, charge the sockets to the elements of the tree to satisfy the condition that 2 sockets are charged to each non-root node and 4 sockets to the root.

Considering where the falsehood of the theorem was used in constructing the charging scheme, we notice that it was only used in charging 4 to each of the leaves of the tree. Therefore, one of the leaves S of the tree must be such that $|\delta(S)| \leq 3$. $\square$

It would be nice to have a simple, intuitive proof that there exists e such that $x_e \geq 1/2$. It would also be nice to have a combinatorial 2-approximation algorithm for the problem, rather than one that needs to solve a linear program.

ORIE 634 Approximation Algorithms	December 3, 1998
Lecture 14	
Lecturer: David P. Williamson	*Scribe: David P. Williamson*

14.1 Research Problems

For our final class, we will discuss open problems in the area of approximation algorithms, covering some things that are known about them along the way. We've now covered most of the major techniques that are used in approximation algorithms; from here on you'll be able to make your own contributions.

In discussing research problems, we will give five hard problems and ten easy problems. The hard problems might be considered the frontier of the field: they are problems which many people have spent a good deal of time on and for which any progress would be significant. The easy problems are ones that I think should be solvable without investing several years of one's life. But of course it is hard to tell what is hard and what is easy; perhaps in retrospect some of the hard problems, when solved, will appear easy, and vice versa!

14.1.1 Five Hard(?) Problems

Problem 1: Vertex Cover

The first hard problem is the vertex cover problem, which we discussed in Lectures 1 and 2. The best known performance guarantee for the problem is $2 - o(1)$, while the best known hardness bound is $7/6$ (due to Håstad).

One idea one might consider for this problem is to use semidefinite programming to get an improved approximation algorithm. Consider the following vector programming relaxation of the problem:

$$\text{Min} \quad \frac{1}{2}\sum_{i \in V} w_i (1 + v_i \cdot v_0)$$

subject to:

$$\begin{aligned} (v_0 - v_i) \cdot (v_0 - v_j) &= 0 && \forall (i,j) \in E \\ v_i \cdot v_i &= 1 && \forall i \in V \\ v_0 \cdot v_0 &= 1 \end{aligned}$$

If $S \subseteq V$ is an optimal vertex cover, we obtain a feasible solution of value $\sum_{i \in S} w_i$ by setting v_0 to an arbitrary unit vector, $v_i = v_0$ for $i \in S$, and $v_i = -v_0$ for $i \notin S$.

Notice that since for any $(i,j) \in E$, either $i \in S$ or $j \in S$, this implies that the constraints are obeyed.

Unfortunately, the following theorem shows that this approach does not help.

Theorem 14.1 [Goemans, Kleinberg '98] For any $\epsilon > 0$, there exists a graph G such that the ratio of OPT to the value of the SDP relaxation is at least $2 - \epsilon$.

Goemans and Kleinberg considered adding the constraints $(v_0 - v_i) \cdot (v_0 - v_j) \geq 0$ for all i, j. These constraints are valid given our previous feasible solution: whenever either $i \in S$ or $j \in S$ the left-hand side is 0, while if neither $i \in S$ or $j \in S$, then the left-hand side is $(2v_0) \cdot (2v_0) = 4$. Lagergren and Russell claimed they had a proof that the same theorem as above holds for this SDP relaxation as well, but their proof had a bug in it; the proof may or may not be fixable.

Problem 2: Symmetric TSP with triangle inequality

In the lecture notes we discussed the Traveling Salesman Problem, given that edge costs c_{ij} obey the triangle inequality. The best known approximation algorithm for this problem has a performance guarantee of 3/2 (Christofides '76), while to my knowledge the best known hardness bound is relatively insignificant (e.g. 1.000001).

Many people have studied the following linear programming relaxation of the problem:

$$Z_{LP} = \text{Min} \quad \sum_e c_e x_e$$

subject to:

$$\begin{aligned} \sum_{e \in \delta(S)} x_e &\geq 2 && \forall S \subseteq V \\ \sum_{e \in \delta(\{v\})} x_e &= 2 && \forall v \in V \\ x_e &\geq 0 && \forall e \in E. \end{aligned}$$

If Z_C is the value of the solution produced by Christofides' algorithm, then one can show that in fact $Z_C \leq \frac{3}{2} Z_{LP}$.

One reason there has been intense study of this LP relaxation is that in considering the integrality gap of this relaxation, the worst-known example has ratio $OPT/Z_{LP} = 4/3$. So it seems possible that this relaxation is within a factor of 4/3 of optimal.

Problem 3: Asymmetric TSP with triangle inequality

In the asymmetric TSP, we have the case that c_{ij} is not necessarily equal to c_{ji}. Here, the best known approximation algorithm for this problem has a performance guarantee of $O(\log n)$ (Frieze, Galbiati, Maffioli '82). I am not aware of a result showing a stronger hardness bound for asymmetric TSP than for symmetric TSP.

Here we present a new $O(\log n)$-approximation algorithm for the ATSP. (Kleinberg, W '98). First we need the following definition, theorem, and lemma.

Definition 14.1 An *Eulerian tour* of a digraph (V, A) is a tour that visits each arc exactly once.

Theorem 14.2 [Euler] The digraph (V, A) has an Eulerian tour if and only if it is connected and the indegree of every vertex is the same as its outdegree.

Lemma 14.3 Let $G = (V, A)$ be a complete digraph, with edge costs obeying the triangle inequality. If (V, A'), $A' \subseteq A$, has an Eulerian tour, then G has a TSP tour of no greater cost.

Proof: Traverse the Eulerian tour and shortcut vertices that have been previously visited. □

For our algorithm, we will need the following concept.

Definition 14.2 A *minimum-mean cycle* C of a digraph is a cycle C that minimizes $\frac{\sum_{e\in C} c_e}{|C|}$ over all cycles in the graph.

Fortunately for us, this problem is solvable in polynomial time.

Theorem 14.4 [Karp '78] The minimum-mean cycle problem is solvable in $O(mn)$ time.

Now consider the following algorithm that produces an Eulerian tour.

KW

$G_0 \leftarrow G$
$i \leftarrow 0$
While $|V_i| > 1$
  Find min-mean cycle C_i in G_i
  Pick any node $j \in C_i$
  $V_{i+1} \leftarrow V_i - C_i + j$
  $G_{i+1} \leftarrow (V_i, A[V_{i+1}])$
Return $\bigcup_k C_k$

It is not difficult to see that $(V, \bigcup_k C_k)$ has an Eulerian tour. It is also not difficult to see that for any i, $OPT(G_i) \leq OPT(G_0)$, where $OPT(G_i)$ is the length of the optimal TSP tour on G_i: since G_i contains a subset of the nodes and arcs of G_0, given the optimal tour on G_0, we can simply shortcut the nodes in $V_0 - V_i$ to obtain a tour of no greater cost on G_i; thus the statement follows.

We can now show that KW is an $O(\log n)$-approximation algorithm.

Theorem 14.5 KW is an $O(\log n)$-approximation algorithm for the asymmetric TSP with triangle inequality.

Proof: Since we know that C_i is the minimum-mean cycle of G_i, it follows that for any i

$$\frac{\sum_{e\in C_i} c_e}{|C_i|} \leq \frac{OPT(G_i)}{|V_i|},$$

or

$$\sum_{e\in C_i} c_e \leq \frac{|C_i|}{|V_i|} OPT(G_i).$$

By construction we know that $|V_{i+1}| = |V_i| - |C_i| + 1$. The cost of the Eulerian tour constructed by KW is the cost of all the edges in all the cycles found by KW, which is $\sum_i \sum_{e\in C_i} c_e$. We can bound this as follows:

$$\begin{aligned}
\sum_i \sum_{e\in C_i} c_e &\leq \sum_i \frac{|C_i|}{|V_i|} OPT(G_i) \\
&\leq \sum_i \left(\frac{1}{|V_i|} + \frac{1}{|V_i|-1} + \cdots + \frac{1}{|V_i| - |C_i| + 1} \right) OPT \\
&= \sum_i \left(\frac{1}{|V_i|} + \frac{1}{|V_i|-1} + \cdots + \frac{1}{|V_{i+1}|+1} + \frac{1}{|V_{i+1}|} \right) OPT \\
&\leq \left(\sum_{j=1}^{n} \frac{1}{j} + \sum_i \frac{1}{|V_i|} \right) OPT \\
&\leq (2 \ln n) OPT.
\end{aligned}$$

□

Problem 4: Bin packing

As we saw earlier in the term, the best known approximation algorithm produces a solution with $OPT + O(\log^2 OPT)$ bins (Karmarkar, Karp). There is no known hardness bound for this problem. It seems possible that better LP rounding schemes will lead to a better algorithm, however. For the LP we discussed earlier, it is known that there exist instances such that $OPT - Z_{LP} > 1$, but there are no known instances such that $OPT - Z_{LP} > 2$.

Problem 5: $\frac{1}{3}$-balanced cuts

As we discussed when we covered finding balanced cuts, there are essentially no known approximation algorithms for this problem, and no known hardness bounds. It is entirely possible, given our state of knowledge, that an approximation scheme exists for these problems.

In the case of planar graphs, there is a 2-approximation algorithm for 1/3-balanced cuts due to Garg, Saran, and Vazirani (1994).

14.1.2 Ten Easy(??) Problems

In this section, I list ten problems that I view as being eminently solvable (with the exception of problems 8 and 9). To provide further incentive for tackling these problems, I also list a dollar amount that I will provide for the answer to any of these problems that is given by 1/1/1.

Problem 1: ($30) Asymmetric p-center

Consider the following problem:

Asymmetric p-center

- **Input:**
  - Set V of vertices
  - Values d_{ij} for all $i, j \in V$ obeying triangle inequality
  - Parameter p
- **Goal:** Find $S \subseteq V$ with $|S| = p$ such that

$$\max_{i \in V} \min_{j \in S} d_{ij}$$

  is minimized.

If the distances d_{ij} are symmetric, (that is, $d_{ij} = d_{ji}$ for all $i, j \in V$) then a 2-approximation algorithms is known. This turns out to be best possible unless $P = NP$. For the asymmetric version, the best approximation algorithm known is an $O(\log^* n)$-approximation algorithm (Viswanathan, SODA '96).

For $30, give a constant approximation algorithm for this problem.

Problem 2: ($30) Combinatorial survivable network design

For $30, give a combinatorial version of Jain's 2-approximation algorithm for survivable network design. If it makes the problem easier, consider the version in which both the algorithm and the optimal solution may use multiple copies of any edge.

Problem 3: ($40) Scheduling related machines

Consider the scheduling problem known in the literature as $Q|prec|C_{\max}$.

$Q|prec|C_{\max}$

- **Input:**
  - m machines, of speeds $s_1 \geq s_2 \geq \cdots \geq s_m$
  - n jobs, with processing requirements $p_1, p_2, \ldots, p_n$
  - Partial order $\prec$ on jobs
- **Goal:** Find a schedule that minimizes the overall completion time, where the running time of job j on machine i is p_j/s_i, and whenever $j \prec j'$, then j must complete before j' starts.

The best currently known algorithm is an $O(\log m)$-approximation algorithm due to Chudak and Shmoys (SODA '97) (see also a simplified version by Chekuri (IPCO '97)).

For $40, give a constant approximation algorithm.

Problem 4: ($30) Subset feedback vertex set problem

Subset FVS

- **Input:**
  - Undirected graph $G = (V, E)$
  - Weights w_i for all $i \in V$
  - Set $S \subseteq V$
- **Goal:** Find a set $F \subseteq V$ that minimizes $\sum_{i \in F} w_i$ such that for all cycles C in G with $C \cap S \neq \emptyset$ it is also the case that $C \cap F \neq \emptyset$.

There is an 8-approximation algorithm known for this problem (Even, Naor, Zosin, FOCS '96). For $30, give a 2-approximation algorithm.

Problem 5: ($40) Small distortion embeddings

In problem set 5, we claimed that any semimetric has an $O(\log n)$-distortion embedding; this was proven by Linial, London, and Rabinovich (*Combinatorica* 15, 1995). In problem set 3, we saw that one could use semidefinite programming to obtain an embedding with the lowest possible distortion; in particular, the objective function of the semidefinite programming was c^2, where c is the distortion of the embedding. Thus another possible proof of the theorem of Linial, London, and Rabinovich is to give a feasible solution to the dual of the semidefinite program with objective function value $O(\log^2 n)$.

For $40, give such a proof.

Problem 6: ($40) A bidirected relaxation of the Steiner tree problem

The following linear programming relaxation of the Steiner tree problem is known to have an integrality gap of 2:

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

subject to:

$$\sum_{e \in \delta(S)} x_e \geq 1 \qquad \forall S \subseteq V : \emptyset \neq S \cap T \neq T$$
$$x_e \geq 0 \qquad \forall e \in E,$$

where T is the set of terminals. This is true even if $T = V$, and the linear program is a relaxation of the minimum spanning tree problem.

There are linear programming relaxations, however, which are known to produce the optimal solution in the case $T = V$. Here is one of them: suppose we replace every undirected edge $(u, v) \in E$ with two directed arcs (u, v) and (v, u) in arcset A, each having the same cost c_e as the undirected edge. Let r be an arbitrary vertex in T, and let $\delta^-(S) = \{(u, v) \in A : u \notin S, v \in S\}$. The consider the following relaxation:

$$\text{Min} \sum_{a \in A} c_a x_a$$

subject to:

$$\sum_{a \in \delta^-(S)} x_a \geq 1 \qquad \forall S \subseteq V - r : \emptyset \neq S \cap T \neq T - r$$
$$x_a \geq 0.$$

It is easy to see that the optimal Steiner tree is a feasible solution of value OPT for this linear program: given the optimal tree, we can direct all the edges of the tree away from the chosen root r, and this gives a feasible solution for the same value.

The question is whether this relaxation gives an integrality gap of some constant smaller than 2. Recently, Rajagopalan and Vazirani (SODA '99) have shown that if the input graph is such that no non-terminal is adjacent to any other non-terminal, then a primal-dual algorithm using this relaxation gives a $\frac{3}{2}$-approximation algorithm. There is some speculation that the integrality gap is a factor of $\frac{3}{2}$ in general. For $40, give an approximation algorithm with constant performance guarantee better than 2 that uses this linear programming relaxation as a lower bound, or give an example that shows that a factor of 2 is the best achievable using this formulation.

Problem 7: ($40) Tree metrics

Although we did not discuss them in this class, the tool of "probabilistically approximating" a metric by tree metrics has proven to be very useful in designing approximation algorithms. For definitions, see the FOCS '96 paper by Bartal.

For $40, show that every metric can be $O(\log n)$-probabilistically approximated by tree metrics.

Problem 8: Sparsest cut problem

Find an improved approximation algorithm to the sparsest cut problem which was covered in problem set 5. I will give $50 for a performance guarantee of $o(\log n)$, and $100 for a constant performance guarantee. It seems like semidefinite programming should be useful for this problem.

Problem 9: MAX CUT

It has been observed that one could validly add the following "triangle inequality" constraints to the vector programming relaxation of MAX CUT:

$$(v_i - v_j) \cdot (v_i - v_k) \geq 0$$

for all distinct i, j, k. This should give an improved semidefinite programming relaxation. I will give $100 for a tight analysis of the integrality gap of this semidefinite program, otherwise a sliding scale for improvement from .878 up to 16/17 (the known hardness bound for MAX CUT).

Problem 10: ($50) Combinatorial MAX CUT

For $50, give a combinatorial .878-approximation algorithm for MAX CUT.