CS880: Approximations Algorithms

Scribe: Michael Kowalczyk **Lecturer:** Shuchi Chawla
Topic: Intro, Vertex Cover, TSP, Steiner Tree **Date:** 1/23/2007

Today we discuss the background and motivation behind studying approximation algorithms and give a few examples of approximations for classic problems.

1.1 Introduction

Approximation algorithms make up a broad topic and are applicable to many areas. In this course a variety of different techniques for crafting approximation algorithms will be covered. Since there is no fixed procedure that will always work when attempting to solve new problems, the aim of this course is to achieve a solid understanding of the most relevant techniques and to get a feel for when they are likely to apply. For those with a particular interest in theory, another goal of this course is to provide enough background in the area so that current research papers on approximation algorithms can be read with relative ease.

1.2 Motivation

In practice, many important problems are NP-hard. If we want to have a good chance at solving them, we need to modify our goal in some way. We could use heuristics, but these are approaches that make no guarantees as to their effectiveness. In some applications, we may only need to solve a special case of the NP-hard problem, and this loss of generality may admit tractibilty. Alternately, average case analysis may be useful; since NP-hardness is a theory of worst case behavior, a problem can be NP-hard and yet very easy to solve on most instances. Finally, if the NP-hard problem deals with optimizing some parameter then we can try to design an approximation algorithm that efficiently produces a sub-optimal solution. It turns out that we can often design these algorithms in such a way that the quality of the output is guaranteed to be, say, within a constant factor of an optimal solution. This is the approach that we will investigate throughout the course.

1.3 Terminology

In order for approximations to make sense, we need to have some quality measure for the solutions. We will use OPT to denote the quality associated with an optimal solution for the problem at hand, and ALG to denote the (worst case) quality produced by the approximation algorithm under consideration. We would like to guarantee $ALG(I) \geq \frac{1}{\alpha} OPT(I)$ on any instance I for maximization problems, where α is known as the approximation factor. Note that $\alpha \geq 1$, and we allow α to be a function of some property of the instance, such as its size. For minimization problems, we write $ALG(I) \leq \alpha OPT(I)$ instead. In other words, we usually state approximations in such a way that $\alpha \geq 1$, although this standard is not universal.

Different problems exhibit different approximation factors. One class of problems have the property

that given any positive constant ϵ, we can give a $(1+\epsilon)$-approximation algorithm for it (although the runtime of the approximation algorithm is efficient for fixed ϵ, it may get much worse as ϵ becomes small). Problems with this property are said to have a polynomial time aproximation scheme, or $PTAS$. This is one of the best scenarios for approximation algorithms. At the other extreme, we can prove in some cases that it is NP-hard to give an approximation algorithm with α any better than, say n^ϵ. We can then categorize optimization problems in terms of how well they can be approximated by efficient algorithms.

Approximation factor (α)	Approximability
$1+\epsilon$ ($PTAS$)	Very approximable
1.1	
2	
constant c	
$\log n$	Somewhat approximable
$\log^2 n$	
$\sqrt{n}$	
n^ϵ	
n	Not very approximable

Figure 1.3.1: Some various degrees to which a problem can be approximated

We will also study the limits of approximability. Hardness of approximation results are proofs that no approximation better than a certain factor exists for a particular problem, under some hardness asumption. For example, set cover cannot be approximated to within an $o(\log n)$ factor, where n is the number of nodes, unless $NP = P$ (in which case we can solve it exactly). For some problems like set cover, we can get tight results, i.e. any improvement over the currently best known factor of approximation would imply $P = NP$ or refute another such complexity assumption. Other problems have a huge gap between the upper and lower approximability bounds currently known, so there is much variation from problem to problem.

1.4 Vertex Cover

One important step in finding approximation algorithms is figuring out what OPT is. But having a method for calculating OPT exactly is often enough to get a full solution for the problem. Vertex cover has this property.

Definition 1.4.1 (Vertex cover) *Given an undirected graph $G = (V, E)$, find a smallest subset $S \subseteq V$ such that every edge in E is incident on at least one of the vertices in S.*

Now assume that we have an algorithm that can tell us how many vertices are in an optimal vertex cover for any given graph. Then we can use this procedure to efficiently find a solution that is optimal. The idea is to observe how removing a vertex effects the size of an optimal vertex cover. If the number of vertices needed to make a vertex cover in the induced subgraph is reduced, then we know that we need to include the removed vertex in S. If the count remains the same, then we know that there is no harm in leaving that vertex out of S. We continue in this way to determine

which vertices to include in the vertex cover. Once our algorithm outputs zero for the remaining induced subgraph, S is an optimal vertex cover. This property of being able to reduce an instance to a smaller instance of the same problem is called *self-reducibility*.

It would be ideal to prove that an algorithm always outputs a solution that is within an α factor of OPT, even if we don't know what OPT is. This is where lower bounds come into play. Given a problem instance I, we can find some property Π such that $\Pi(I) \leq OPT(I)$. Then we get around calculating OPT by showing that $ALG(I) \leq \alpha \cdot \Pi(I) \leq \alpha \cdot OPT(I)$.

For vertex cover, we can look for a collection of edges such that no two of these edges share a vertex. Since any vertex cover would have to choose at least one of the vertices from each of these edegs, this shows that the size of any vertex cover for G is at least the size of any matching for G.

Lemma 1.4.2 *The size of any matching in G is a lower bound on the size of an optimal vertex cover in G.*

Proof: See above. ■

Since we want $\Pi(I)$ to be as large as possible, we choose our lower bound to be the size of a maximal matching for G. Given a maximal matching, a natural approximation to an optimal solution is just to include all vertices incident on the edges of our maximal matching. It is easy to see that this is an admissible solution since the existence of an edge that isn't incident on one of these vertices contradicts the maximality of the matching.

```
Approximate Vertex Cover(G = (V, E) - an undirected graph)
    Let M = any maximal matching on G
    Let S = all vertices incident on M
    return S
```

Theorem 1.4.3 *The above algorithm is a 2-approximation to vertex cover.*

Proof: Let $\Pi(I)$ be the size of the matching found by the above algorithm. Then $ALG(I) \leq 2 \cdot \Pi(I)$ and by lemma 1.4.2, $\Pi(I) \leq OPT(I)$, so $ALG(I) \leq 2 \cdot \Pi(I) \leq 2 \cdot OPT(I)$. ■

This is the best known approximation for vertex cover to date. In future lectures we will see other ways of obtaining the same approximation factor.

1.5 Metric TSP

Now we will consider the traveling salesperson problem.

Definition 1.5.1 (Traveling salesperson problem (TSP)) *Given a graph with weights assigned to the edges, find a minimum weight tour that visits every vertex exactly once.*

It's a good exercise to show that any reasonable approximation for TSP yields a solution for the Hamiltonian cycle problem (even when the graph is required to be the complete graph). Therefore, we will consider a less general version of the problem where we relax the ban on revisiting vertices.

Definition 1.5.2 (Metric TSP) *Given a graph with weights assigned to the edges, find a mini-*

mum weight tour that visits each vertex at least once.

This is called metric TSP because it is equivalent to solving the original TSP (without revisiting vertices) on the "metric completion" of the graph. By metric-completion, we mean that we put an edge between every pair of nodes in the graph with length equal to the length of the shortest path between them. The shortest path function on a connected graph forms a metric. A metric is an assignment of length to every pair of nodes such that the following hold for all u, v, and w.

$$\begin{aligned} d(u,v) &\geq 0 \\ d(u,v) &= 0 \quad \text{if and only if} \quad u = v \\ d(u,v) &= d(v,u) \\ d(u,v) &\leq d(u,w) + d(w,v) \qquad \text{(the triangle inequality)} \end{aligned}$$

This is a minimization problem, so now we look for lower bounds on OPT. Some possibilities include minimum spanning tree, the diameter of the graph, and TSP-path (i.e. a Hamiltonian path of minimum weight).

One desirable property for a lower bound is that it is as close to OPT as possible. Suppose we had a property Π such that there exist arbitrarily large instances I such that $OPT(I) = 100 \cdot \Pi(I)$. Then we can't hope to get α better than 100. In our case, if I is the complete graph with unit edge weights, and $\Pi(I)$ is the diameter of the graph then we see that $OPT(I) = n \cdot \Pi(I)$, so we won't be able to use graph diameter as a lower bound for metric TSP.

Another quality of a good lower bound is that the property is easier to understand or calculate than OPT itself. We may run into this trouble with TSP-path. However, minimal spanning tree can be computed efficiently, so we will try to use this as our lower bound (to see that it is a lower bound, note that an optimal tour with one edge removed is a tree).

If we use a depth first search tour of our tree as an approximation then we see that $ALG = 2 \cdot MST \leq 2 \cdot OPT$ so we have a 2-approximation of metric TSP.

Can this analysis be improved upon? The answer is no, because we can find a family of instances where $OPT(I) = 2 \cdot \Pi(I)$. The line graph suffices as an example of this. That is, let I be a graph on n vertices where each vertex connects only to the "next" and "previous" vertices (there will be a total of $n-1$ edges, each with edge weight 1). Then $OPT(I) = 2(n-1) = 2 \cdot \Pi(I)$. This example shows that we cannot get a better approximation for TSP using only the MST lower bound. But it doesn't preclude the possibility of a different algorithm using a different lower bound.

The crucial observation in improving this algorithm is to observe that our approach was really to convert the tree into an Eulerian graph and then use the fact there is an Eulerian tour. If we can find a more cost-effective way to transform the tree into an Eulerian graph, we could improve the approximation factor. An Eulerian tour exists if and only if every vertex has even degree (given that we want to start and end at the same vertex). Thus it suffices to find a matching between all nodes of odd degree in the minimal spanning tree and add these edges to the tree.

Lemma 1.5.3 *Given a weighted graph $G = (V, E)$ and a subset of vertices $S \subseteq V$ with even cardinality, a minimum cost perfect matching for S has weight at most half of a minimum cost metric TSP tour on G.*

Proof: Let S as in the statement of the lemma and consider a minimum cost TSP tour on S. Because of the metric property, this tour has a cost at most OPT. It also induces 2 matchings (take every other edge), and the smaller matching has cost at most $\frac{1}{2}OPT$. ■

APPROXIMATE METRIC TSP($G = (V, E)$ - a weighted undirected graph)
  Let M = minimal spanning tree in G
  Let U = set of vertices with odd degree in M
  Let P = minimum cost perfect matching on U in G
  return Eulerian tour on $M \cup P$

Theorem 1.5.4 *The above algorithm is a $\frac{3}{2}$-approximation to metric TSP.*

Proof: Since there must be an even number of vertices of odd degree in the minimal spanning tree M, we can use a minimum cost perfect matching beteween these vertices to complete an Eulerian graph. We know the minimum cost perfect matching has weight at most $\frac{1}{2}OPT$, thus $ALG \leq MST + MATCHING \leq \frac{3}{2}OPT$. We have a $\frac{3}{2}$-approximation for metric TSP. ■

Note that we used two lower bounds in our approximation. Each lower bound on its own isn't fantastic but combining them together produced a more impressive result. As an exercise, find a family of instances for which there is a big gap between the matching lower bound and the optimal TSP tour. This simple approximation algorithm has been known for over 30 years [1] and no improvements have been made since its discovery.

1.6 Steiner Tree

Next time we will talk about the Steiner tree problem.

Definition 1.6.1 (Steiner tree problem) *Given a weighted graph $G = (V, E)$ and a subset $T \subseteq V$ of nodes in a graph, find a least cost tree connecting all of the nodes in T (the "helper nodes" that are in the graph but not in in T are known as Steiner nodes).*

Applications of Steiner trees arise in computer networks. We will discuss this problem next lecture.

References

[1] N. Christofides. Worst-case analysis of a new heuristic for the traveling salesman problem. In *Technical report no. 388, Graduate School of Industrial Administration, Carnegie-Mellon University*, 1976.

CS880: Approximations Algorithms	
Scribe: Siddharth Barman	**Lecturer:** Shuchi Chawla
Topic: Steiner Tree; Greedy Approximation Algorithms	**Date:** 01/25/07

In this lecture we give an algorithm for Steiner tree and then discuss greedy algorithms.

2.1 Steiner Tree

Problem Statement: Given a weighted graph $G = (V, E)$ and a set $R \subseteq V$, our goal is to determine the least cost connected subgraph spanning R. Vertices in R are called terminal nodes and those in $V \backslash R$ are called Steiner vertices.

Note that the least cost connected subgraph spanning R is a tree. Also we are free to use non terminal vertices of the graph (the Steiner nodes) in order to determine the Steiner tree.

We first show that we can assume without loss of generality that G is a complete graph. In particular, let G^c denote the metric completion of G: the cost of an edge (u, v) in G^c for all $u, v \in V$ is given by the length of the shortest path between u and v in G. As an exercise verify that costs in G^c form a metric.

Lemma 2.1.1 *Let T be a tree spanning R in G^c, then we can find a subtree in G, T' that spans R and has cost at most the cost of T in G^c.*

Proof: We can get T' by taking a union of the paths represented by edges in T. Cost of edges in G^c is the same as the cost of the paths (shortest) of G they represent. Hence the sum of costs of all the paths making up T' is at most the cost of T. ■

As seen in the previous lecture a lower bound is a good place to start looking for an approximate solution. We will use ideas from the previous lecture relating trees to traveling salesman tours to obtain a lower bound on the optimal Steiner tree.

Lemma 2.1.2 *Let G^c_R be the subgraph of G^c induced by R. Then the cost of the MST in G^c_R is at most the cost of the optimal TSP tour spanning G^c_R.*

Proof: See previous lecture. ■

Lemma 2.1.3 *The cost of the optimal TSP tour spanning G^R_c is at most twice OPT (the optimal Steiner tree in G^c spanning R)*

Proof: As shown in Figure 2.1.1 traversing each edge of the Steiner tree twice gives us a path which covers all the terminal nodes i.e. a tour covering all of R. Hence TSP spanning $R \leq 2OPT$. ■

Corollary 2.1.4 *The cost of the MST of G^c_R is at most twice OPT.*

Proof: Follows from the above mentioned lemmas. ■

The lower bound in itself suggests a 2-approximation to the Steiner tree. The algorithm is to simply determine the MST on R in G^c_R i.e.

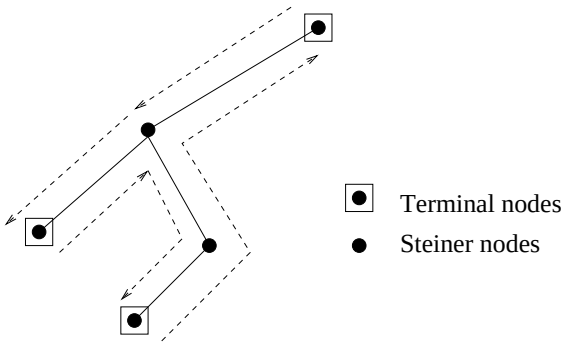


Figure 2.1.1: Constructing tour spanning R from the optimal Steiner tree

- Consider G^c, the metric completion of graph G.
- Get G^c_R, the subgraph induced by set R on G_c.
- Determine MST on G^c_R, translate it back to a tree in G as described in the proof of lemma 2.1.1.
- Output this tree

Theorem 2.1.5 *The algorithm above gives a* 2*-approximation to Steiner tree.*

Proof: Follows from the three lemmas stated above. ■

By a more careful analysis the algorithm can be shown to give a $2\left(1-\frac{1}{|R|}\right)$ approximation. This is left as an exercise.

The best known approximation factor for the Steiner tree problem is 1.55 [5]. Also from the hardness of approximation side it is known that Steiner tree is "$APX - Hard$", i.e. there exists some constant $c > 1$ s.t. Steiner tree is $\mathcal{NP}$- Hard to approximate better than c [1].

2.2 Greedy Approximation Algorithms—the min. multiway cut problem

Next we look at greedy approximation algorithms. The design strategy carries over from greedy algorithms for exact algorithm design i.e. our aim here is to pick the myopic best action at each step and not be concerned about the global picture.

First we consider the **Min Multiway Cut Problem**.

Problem Statement: Given a weighted graph G with a set of terminal node T and costs (weights) on edges our goal is to find the smallest cut separating all the terminals.

Let $k = |T|$ denote the cardinality of the terminal set. When $k = 2$, the problem reduces to simple min cut and hence is polytime solvable. For $k \geq 3$ it is known that the problem is $\mathcal{NP}$-Hard and

also $APX\text{-}Hard$ [3]. Note that multiway cut is different from the multicut problem. We will study the latter later on in the course.

We provide a greedy approximation algorithm for the min multiway cut problem and give a tight analysis to show that it achieves an approximation factor of $2\left(1-\frac{1}{k}\right)$. The algorithm and analysis is due to Dahlhaus et al. [3]

Algorithm: For every terminal $t_i \in T$, find the min-cut C_i separating t_i from $T\backslash\{t_i\}$. A Multiway cut is obtained by taking the union of the $(k-1)$ smallest cuts out of C_1,C_2,...C_k.

The size of multiway cut determined by the greedy algorithm is $\sum_{i\leq k}|C_i|$.

We proceed to analyze the relative goodness of the greedy solution with respect to that of the optimal. The idea behind the analysis is captured by figure 2.2.2.

Lemma 2.2.1 $\sum_i |C_i| \leq 2OPT$, *where* OPT *is the cost of the minimum multiway cut.*

Proof: Consider the components generated by OPT, call them S_1,...,S_k. For each t_i, consider the full cut C_i' consisting of all the edges in OPT with *exactly one* end point in S_i. We know that $|C_i| \leq |C_i'|$, because C_i' separates t_i from $T\backslash\{t_i\}$.

Also (Figure 2.2.2) on summing up the capacities of C_i's each edge of the min multiway cut is counted twice hence we have $\sum_{i=1}^{k}|C_i'| = 2OPT$. Combining the two we get $\sum_{i=1}^{k}|C_i| \leq 2OPT$ ■

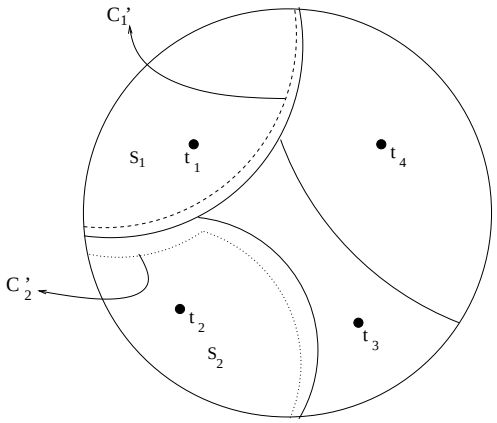


Figure 2.2.2: Min multiway cut

A tighter version of the above may be stated as follows.

Lemma 2.2.2 *Let* $j = argmax_i|C_i|$, *i.e.* j *is the largest of the cuts then*

$$\sum_{i\in[k],i\neq j}|C_i| \leq 2\left(1-\frac{1}{k}\right)OPT$$

Proof:

As $|C_j|$ is greater than the average value of the cuts we have,

$$\sum_{i\in[k],i\neq j} |C_i| \leq \left(1-\frac{1}{k}\right)\sum_{i\in[k]} |C_i|$$

Combining this with the previous lemma we get the desired result. ■

Theorem 2.2.3 *The above algorithm gives a* $2\left(1-\frac{1}{k}\right)$ *approximation to min multiway cut.*

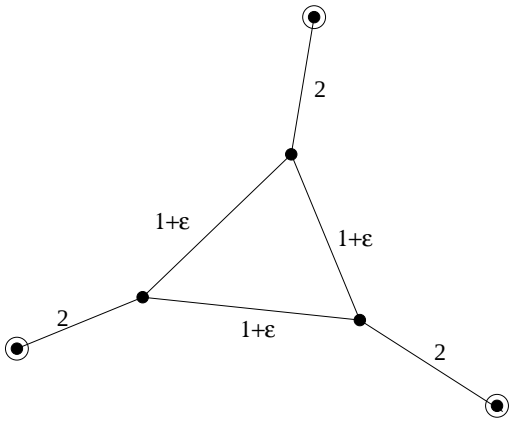


Figure 2.2.3: Bad example for Min multiway cut greedy algorithm

We now give a tight example for the algorithm. First let us examine the case of $k = 3$ as in the figure above. In this case the algo. returns a cut of cost 4, whereas the min. multiway cut has cost $3(1+\epsilon)$. Generalizing the triangle of Figure 2.2.3 to a k cycle we find that the analysis of the algorithm provided above is in fact tight. In particular for the k cycle case, $|C_i| = 2$ for each i and hence the greedy multiway cut is of size $2(k-1)$. The min multiway cut is in fact the k cycle hence $OPT = (1+\epsilon)k$. The example shows that our analysis of the provided greedy algorithm is in fact tight.

Though the analysis provided here is tight, better algorithms exist for the min multiway cut problem. Calinescu et al. [2] gave a $\left(\frac{3}{2}-\frac{1}{k}\right)$ approx. which was subsequently improved to a 1.3438 approx by Karger et al. [4].

References

[1] M. Bern, P. Plassmann. The steiner problem with edge lengths 1 and 2. In *Information Processing Letters* Volume 32-1 (1989), pp: 171-176.

[2] G. Calinescu, H.J. Karloff, Y. Rabani. An Improved Approximation Algorithm for Multiway Cut. In *STOC*(1998), pp: 48-52.

[3] E. Dahlhaus, D.S. Johnson, C.H. Papadimitriou, P.D. Seymour, M. Yannakakis. The Complexity of Multiterminal Cuts. In *SIAM J. Comput.* Volume 23 (1994), pp: 864-894.

[4] D.R. Karger, P.N. Klein, C. Stein, M. Thorup, N.E. Young. Rounding Algorithms for a Geometric Embedding of Minimum Multiway Cut. In *STOC*(1999), pp: 668-678.

[5] G. Robins, A. Zelikovsky. Tighter Bounds for Graph Steiner Tree Approximation. In *SIAM Journal on Discrete Mathematics* Volume 19-1 (2005), pp: 122-134.

CS880: Approximations Algorithms

Scribe: Matt Elder **Lecturer:** Shuchi Chawla
Topic: Greedy Approximations: Set Cover and Min Makespan **Date:** 1/30/06

3.1 Set Cover

The *Set Cover* problem is: Given a set of elements $E = \{e_1, e_2, \ldots, e_n\}$ and a set of m subsets of $E, \mathcal{S} = \{S_1, S_2, \ldots, S_n\}$, find a "least cost" collection C of sets from $\mathcal{S}$ such that C covers all elements in E. That is, $\cup_{S_i \in C} S_i = E$.

Set Cover comes in two flavors, unweighted and weighted. In unweighted Set Cover, the cost of a collection C is number of sets contained in it. In weighted Set Cover, there is a nonnegative weight function $w : \mathcal{S} \to \mathbb{R}$, and the cost of C is defined to be its total weight, i.e., $\sum_{S_i \in C} w(S_i)$.

First, we will deal with the unweighted Set Cover problem. The following algorithm is an extension of the greedy vertex cover algorithm that we discussed in Lecture 1.

Algorithm 3.1.1 Set Cover(E, $\mathcal{S}$):

1. $C \leftarrow \emptyset$.
2. While E contains elements not covered by C:
   (a) Pick an element $e \in E$ not covered by C.
   (b) Add all sets S_i containing e to C.

To analyze Algorithm 3.1.1, we will need the following definition:

Definition 3.1.2 *A set E' of elements in E is* independent *if, for all $e_1, e_2 \in E'$, there is no $S_i \in C$ such that $e_1, e_2 \in S_i$.*

Now, we shall determine how strong an approximation Algorithm 3.1.1 is. Say that the *frequency* of an element is the number of sets that contain that element. Let F denote the maximum frequency across all elements. Thus, F is the largest number of sets from $\mathcal{S}$ that we might add to our cover C at any step in the algorithm. It is clear that the elements selected by the algorithm form an independent set, so the algorithm selects no more than $F|E'|$ elements, where E' is the set of elements picked in Step 2a. That is, ALG $\leq F|E'|$. Because every element is covered by some subset in an optimal set cover, we know that $|E'| \leq$ OPT for any independent set E'. Thus, ALG $\leq F$ OPT, and Algorithm 3.1.1 is therefore an F–approximation.

Theorem 3.1.3 *Algorithm 3.1.1 is an F–approximation to Set Cover.*

Algorithm 3.1.1 is a good approximation if F is guaranteed to be small. In general, however, there could be some element contained in every set of $\mathcal{S}$, and Algorithm 3.1.1 would be a very poor approximation. So, we consider a different unweighted Set Cover approximation algorithm which uses the greedy strategy to yield a $\ln n$–approximation.

Algorithm 3.1.4 Set Cover(E, $\mathcal{S}$):

1. $C \leftarrow \emptyset$.
2. While E contains elements not covered by C:
   (a) Find the set S_i containing the greatest number of uncovered elements.
   (b) Add S_i to C.

Theorem 3.1.5 *Algorithm 3.1.4 is a* $\ln \frac{n}{OPT}$*–approximation.*

Proof: Let $k = \text{OPT}$, and let E_t be the set of elements not yet covered after step i, with $E_0 = E$. OPT covers every E_t with no more than k sets. ALG always picks the largest set over E_t in step $t+1$. The size of this largest set must cover at least $|E_t|/k$ in E_t; if it covered fewer elements, no way of picking sets would be able to cover E_t in k sets, which contradicts the existence of OPT. So, $|E_{t+1}| \leq |E_t| - |E_t|/k$, and, inductively, $|E_t| \leq n\,(1 - 1/k)^t$.

When $|E_t| < 1$, we know we are done, so we solve for this t:

$$\begin{aligned} &\left(1 - \frac{1}{k}\right)^t < \frac{1}{n} \\ \Rightarrow\ & n < \left(\frac{k}{k-1}\right)^t \\ \Rightarrow\ & \ln n \leq t \ln\left(1 + \frac{1}{k-1}\right) \approx \frac{t}{k} \\ \Rightarrow\ & t \leq k \ln n = \text{OPT} \ln n. \end{aligned}$$

Algorithm 3.1.4 finishes within $\text{OPT} \ln n$ steps, so it uses no more than that many sets. We can get a better analysis for this approximation by considering when $|E_t| < k$, as follows:

$$\begin{aligned} &n\left(1 - \frac{1}{k}\right)^t = k \\ \Rightarrow\ & n\frac{1}{e^{t/k}} = k \text{ (because } (1-x)^{1/x} \leq \frac{1}{e} \text{ for all } x). \\ \Rightarrow\ & e^{t/k} = \frac{n}{k} \\ \Rightarrow\ & t = k \ln \frac{n}{k}. \end{aligned}$$

Thus, after $k \ln \frac{n}{k}$ steps there remain only k elements. Each subsequent step removes at least one element, so $\text{ALG} \leq \text{OPT}\left(\ln \frac{n}{OPT} + 1\right)$. ■

Theorem 3.1.6 *If all sets are of size* $\leq B$*, then there exists a* $(\ln B + 1)$*–approximation to unweighted Set Cover.*

Proof: If all sets have size no greater than B, then $k \geq n/B$. So, $B \geq n/k$, and Algorithm 3.1.4 gives a $(\ln B + 1)$–approximation. ■

Now we extend Algorithm 3.1.4 to the weighted case. Here, instead of selecting sets by their number of uncovered elements, we select sets by the "efficiency" of their uncovered elements, or the number of uncovered elements *per unit weight*.

Algorithm 3.1.7 Weighted Set Cover($\mathcal{S}$, C, E, w):

1. $C \leftarrow \emptyset$, and $E' \leftarrow E$.
2. While E contains uncovered elements:
   (a) $s \leftarrow \text{argmax}_{X \in \mathcal{S}} |X \cap E|/w(X)$.
   (b) $C \leftarrow C \cup s$, $\mathcal{S} \leftarrow \mathcal{S} \setminus \{s\}$, and $E' \leftarrow E' \setminus \mathcal{S}$.

Algorithm 3.1.7 was first analyzed in [5].

Theorem 3.1.8 *Algorithm 3.1.7 achieves a* $\ln n$*–approximation to Weighted Set Cover.*

Proof: For every picked set S_j, define θ_j as $|S_j \cap E|/w(S_j)$ at the time that S_j was picked. For each element e, let S'_j be the first picked set that covers it, and define $\text{cost}(e) = 1/\theta_j$. Notice that $\sum_{e \in E} \text{cost}(e) = \text{ALG}$.

Let us order the elments in the order that they were picked, breaking ties arbitrarily. At the time that the i^{th} element (call it e_i) was picked, E contained at least $n - i + 1$ elements. At that point, the "per-element" cost of OPT is at most $\text{OPT}/(n - i + 1)$. Thus, for at least one of the sets in OPT, we know that

$$\frac{|S \cap E|}{w(s)} \geq \frac{n - i + 1}{\text{OPT}}.$$

Therefore, for the set S_j picked by the algorithm, we have $\theta_j \geq (n - i + 1)/\text{OPT}$. So,

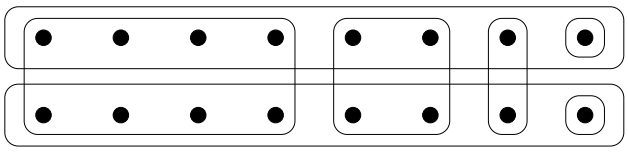
$$\text{cost}(e_i) \leq \frac{\text{OPT}}{n - i + 1}.$$

Over the execution of Algorithm 3.1.7, the value of i goes from n to 1. Thus, the total cost of each element that the algorithm removes is at most

$$\sum_{i=1}^{n} \frac{\text{OPT}}{n - i + 1} \leq \text{OPT} \ln n.$$

Thus, Algorithm 3.1.7 is a $\ln n$–approximation to Weighted Set Cover. ■

The above analysis is tight, which we can see by the following example:



The dots are elements, and the loops represent the sets of S. Each set has weight 1. The optimal solution is to take the two long sets, with a total cost of 2. If Algorithm 3.1.7 instead selects the leftmost thick set at first, then it will take at least 5 sets. This example generalizes to a family of examples each with 2^k elements, and shows that no analysis of Algorithm 3.1.7 will make it better than a $O(\ln n)$–approximation.

A $\ln n$–approximation to Set Cover can also be obtained by other techniques, including LP-rounding. However, Feige showed that no improvement, even by a constant factor, is likely:

Theorem 3.1.9 *There is no* $(1-\epsilon)\ln n$*–approximation to Weighted Set Cover unless* $NP \subseteq DTIME(n^{\log\log n})$. *[1]*

3.2 Min Makespan Scheduling

The *Min Makespan Problem* is: given n jobs to schedule on m machines, where job i has size s_i, schedule the jobs to minimize their makespan.

Definition 3.2.1 *The* makespan *of a schedule is the earliest time when all machines have stopped doing work.*

This problem is NP-hard, as can be seen by a reduction from Partition. The following algorithm due to Ron Graham yields a 2–approximation.

Algorithm 3.2.2 (Graham's List Scheduling) [2] Given a set of n jobs and a set of m empty machine queues,

1. Order the jobs arbitrarily.
2. Until the job list is empty, move the next job in the list to the end of the shortest machine queue.

Theorem 3.2.3 *Graham's List Scheduling is a 2–approximation.*

Proof: Let S_j denote the size of job j. Suppose job i is the last job to finish in a Graham's List schedule, and let t_i be the time it starts. When job i was placed, its queue was no longer than any other queue, so every queue is full until t_i. Thus, $\text{ALG} = S_i + t_i \leq S_i + \frac{(\sum_{j=1}^{n} S_j) - S_i}{m} = \frac{1}{m}\sum_{j=1}^{n} S_j + (1-1/m)S_i$. It's easy to see that $S_i \leq \text{OPT}$ and that $\frac{1}{m}\sum_{j=1}^{n} S_j \leq OPT$. So, we conclude that $\text{ALG} \leq (2-1/m)\text{OPT}$, which yields a 2–approximation. ■

This analysis is tight. Suppose that after the jobs are arbitrarily ordered, the job list contains $m(m-1)$ unit-length jobs, followed by one m-length job. The algorithm yields a schedule completing in $2m-1$ units while the optimal schedule has length m.

This algorithm can be improved. For example, by ordering the job list by increasing duration instead of arbitrarily, we get a (4/3)–approximation, a result proved in [3]. Also, this problem has a poly-time approximation scheme (PTAS), given in [4]. However, a notable property of Algorithm 3.2.2 is that it is an online algorithm, i.e., even if the jobs arrive one after another, and we have no information about what jobs may arrive in the furture, we can still use this algorithm to obtain a 2–approximation.

References

[1] Uriel Feige. A Threshold of $\ln n$ for Approximating Set Cover. In *J. ACM* 45(4), pp 634-652. (1998)

[2] Graham, R. Bounds for Certain Multiprocessing Anomalies. In *Bell System Tech. J.*, 45, pp 1563-1581. (1966)

[3] Ronald L. Graham. Bounds on Multiprocessing Timing Anomalies. In *SIAM Journal of Applied Mathematics*, 17(2), pp 416-429. (1969)

[4] Dorit S. Hochbaum, David B. Shmoys. A Polynomial Approximation Scheme for Scheduling on Uniform Processors: Using the Dual Approximation Approach. In *SIAM J. Comput.* 17(3), pp 539-551. (1988)

[5] D. S. Johnson. Approximation Algorithms for Combinatorial Problems. In *Journal of Computer and System Sciences*, 9, pp 256-278. (1974) Preliminary version in *Proc. of the 5th Ann. ACM Symp. on Theory of Computing*, pp 36-49. (1973)

CS880: Approximations Algorithms

Scribe: Siddharth Barman **Lecturer:** Shuchi Chawla

Topic: Edge Disjoint Paths; Dynamic Programming **Date:** 02/01/07

In this lecture we give an algorithm for Edge disjoint paths problem and then discuss dynamic programming.

4.1 Edge disjoint paths

Problem Statement: Given a **directed graph** G and a set of terminal pairs $\{(s_1, t_1), (s_2, t_2), \cdots, (s_k, t_k)\}$, our goal is to connect as many pairs as possible using non edge intersecting paths.

Edge disjoint paths problem is $\mathcal{NP}$-Complete and is closely related to the multicommodity flow problem. In fact integer multicommodity flow is a generalization of this problem. We describe a greedy approximation algorithm for the edge disjoint path problem due to Jon Kleinberg [4].

Algorithm: Compute shortest path distance between every (s_i, t_i) pair. Route the one with smallest distance along the corresponding shortest path, remove all the used edges from the graph and repeat.

Theorem 4.1.1 *The above algorithm achieves an $O(\sqrt{m})$ approximation, where m is the number of edges in the given graph.*

Before we dwell on the proof of the above theorem we present an instance of the problem (Figure 1) for which the greedy algorithm gives an $\Omega(\sqrt{m})$ approximation. This shows that the analysis is in fact tight.

See Figure 1; The graph is constructed such that the length of the path between terminal vertices s_{l+1}, t_{l+1} is smaller than all other (s_i, t_i) paths. Hence the greedy algorithm picks the path connecting s_{l+1} and t_{l+1} at the first go. This in turn disconnects all other terminal pairs. Thus the greedy algorithm returns a single path, but we can connect (s_i, t_i) pairs for all i between 1 and l by edge disjoint paths.

Note that for the construction to go through length of the path between s_{l+1} and t_{l+1} must be at least l and so the length of the shortest path between s_i and t_i for $1 \leq i \leq l$ must be more than l. So $m = O(l^2)$ and the approximation achieved is $l = \Omega(\sqrt{m})$.

Relation between the optimal solution and our greedy algorithm is achieved by charging each path in OPT to the *first* path in ALG that intersects it. For this we define short and long paths. A *short path* is one which has no more than k edges. Rest of the paths shall be referred to as *long paths*. We will pick an approximate value of k later.

Lemma 4.1.2 *OPT has no more than m/k long paths, where m is the number of edges.*

Proof: The paths in OPT are edge disjoint hence m/k paths of length more than k will cover all the m edges. ■

Lemma 4.1.3 *Each short path in OPT gets charged to some short path in ALG.*

Proof: The greedy algorithm picks the shortest path which is still available. Say P_G is the path

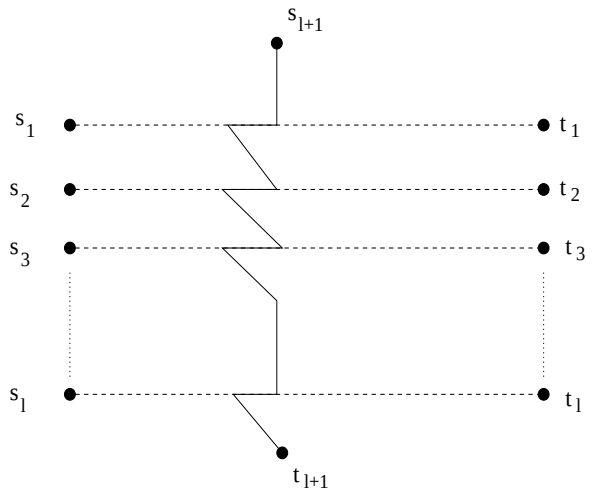


Figure 4.1.1: Bad example for greedy algorithm

picked up ALG that "cuts" P_{OPT} a fixed short path in OPT for the first time. As described earlier P_{OPT} gets charged to P_G. At that point of time all of previously selected paths of ALG are edge disjoint with P_{OPT}, hence P_{OPT} is still available, but ALG decides to choose P_G which implies that length of P_G is less than k, i.e. it is a short path. ■

Lemma 4.1.4 *Each short path in ALG gets charged at most k times.*

Proof: Paths of OPT are edge disjoint themselves hence in the worst case each edge of a short path selected by ALG cuts a different path of OPT. This bound the charge to k. ■

Next we use the above mentioned lemmas to prove Theorem 4.1.1.

Proof of Theorem 4.1.1**:** We partition the optimal solution in long and short paths i.e. $OPT = OPT_{long} + OPT_{short}$. By lemma 4.1.2 we have that OPT_{long} is at most m/k and using the other two lemmas we can bound OPT_{short} by $ALG \times k$. Hence,

$$OPT \leq \frac{m}{k} + ALG \times k$$

Setting $k = \sqrt{m}$ and noting that $ALG \geq 1$ we get

$$OPT \leq 2\sqrt{m} \times ALG$$

Hence the algorithm achieves an approximation factor of $2\sqrt{m}$. ■

The above algorithm and analysis is from Kleinberg's thesis [4]. Chekuri and Khanna [2] gave a better analysis of the same algorithm in terms of the number of vertices of the graph. In particular they showed that the greedy algorithm achieves an approximation of $O(n^{4/5})$ in general and an $O(n^{2/3})$ approximation for undirected graphs.

Surprisingly with complexity theoretic assumptions the greedy algorithm turns out to be the best one could hope for, although the same factor can also be achieved using linear programming.

Hardness results for the edge disjoint paths problem show that unless $\mathcal{P} = \mathcal{NP}$, in directed graphs it is not possible to approximate the edge disjoint path problem better than $\Omega(m^{1/2-\epsilon})$ for any fixed $\epsilon \geq 0$ [3]. For undirected graphs, edge disjoint paths cannot be approximated better than $\Omega(\log^{1/3-\epsilon} m)$ exists unless $\mathcal{NP} \subseteq \mathcal{ZTIME}(n^{\text{polylog}\, n})$ [1].

4.2 Dynamic Programming: Knapsack

The idea behind dynamic programming is to break up the problem into several subproblems, solve these optimally and then combine the solutions to get an optimal solution use them to solve the prob at hand. Generally for approximation we do not use dynamic programming to solve the given instance directly. We use two approaches. First we morph it into an instance with some special property and then apply dynamic programming to solve the special instance exactly. The approximation factor comes from this morphing.

Secondly, dynamic programming can as well be viewed as a clever enumeration technique to search through the entire solution space. With this in mind, approximation algorithms can be designed that restrict the search to only a part of the solution space and not the entire space and apply dynamic programming over this subspace. In this case the approximation factor reflects the gap between the overall optimal solution and the optimal solution over the subspace. Over the next two lectures we will see both kinds of techniques used.

We proceed to design an approximation algorithm for the knapsack problem which uses the morphing idea. Note that knapsack is known to be $\mathcal{NP}$-complete.

Problem Statement: Given a set of n items each with a weight w_i and profit p_i, along with a knapsack of size B our goal is to find a subset of items of total weight less than B and maximum total profit.

Knapsack can be solved exactly using dynamic programming. The exact algorithm proceeds by filling up a $n \times B$ matrix recursively. Each entry (i, b) in the matrix corresponds to the maximal profit that can be achieved using elements 1 through i with total weight less than b. Each entry takes a constant amount of time hence the time complexity is $O(nB)$. We can also employ another exact algorithm. This time we fill up an $n \times P$ matrix M, where $P = \sum_i p_i$ is the total profit. An entry (i, p) of M holds the value of the minimum possible weight required to achieve a profit of p using elements 1 through i. This algorithm takes time $O(nP)$.

These exact algorithms fall in the class of *pseudo polynomial time* algorithms. Formally, a pseudo polynomial time algorithm is one that takes time polynomial in the size of the problem in *unary*. Problems that have pseudo-poly time algorithms and are $\mathcal{NP}$-Hard are called *weakly $\mathcal{NP}$-Hard*.

We now describe how to obtain a polytime approximation algorithm. The main idea is to modify the instance so as to reduce $P = \sum_i p_i$ to some value that is bounded by a polynomial in n. In particular, we pick $K = n/\epsilon$ to be the new maximum profit, for some $\epsilon > 0$. We *scale* the profits uniformly, such that the max. profit equals K., and then we round down these scaled values to the nearest integer to ensure that we have integer profits i.e.

$$p'_i = \left\lfloor p_i \times \frac{K}{p_{max}} \right\rfloor$$

We then solve the knapsack exactly on the new profit values p'_i. We now show that we do not loose much in rounding.

Theorem 4.2.1 *The above mentioned algorithm achieves a $1+\epsilon$ approximation.*

Proof: Let O denote the value of the optimal solution OPT on the original instance and O' denote the value of OPT on new instances, i.e. $O = \sum_{i \in OPT} p_i$ and $O' = \sum_{i \in OPT} p'_i$.

Similarly A and A' be the value of the solution obtained by the algorithm on original and new instances respectively.

Note that $p_i \geq \frac{p_{max}}{K} \left\lfloor p_i \times \frac{K}{p_{max}} \right\rfloor$. Hence

$$\sum_{i \in ALG} p_i \geq \sum_{i \in ALG} \frac{p_{max}}{K} \left\lfloor p_i \times \frac{K}{p_{max}} \right\rfloor$$

Which is equivalent to $A \geq \frac{p_{max}}{K} \times A'$. We solve the problem exactly on the new instances hence A' is the optimal value for p'_is. Hence $A' \geq O'$. Combining the two inequalities we get $A \geq \frac{p_{max}}{K} O'$. Expanding O' we get the following

$$\begin{aligned}
\frac{p_{max}}{K} O' &= \frac{p_{max}}{K} \sum_{i \in OPT} \left\lfloor p_i \times \frac{K}{p_{max}} \right\rfloor \\
&\geq \frac{p_{max}}{K} \sum_{i \in OPT} \left(\frac{p_i K}{p_{max}} - 1 \right) \\
&= \sum_{i \in OPT} p_i - n \frac{p_{max}}{K} \\
&= O - \epsilon p_{max} \\
&\geq O(1-\epsilon)
\end{aligned}$$

Hence $A \geq O(1-\epsilon)$

■

Note that the above mentioned algorithm takes $O(p_{max} n^2)$ time, hence the algorithm runs in time $poly(n, \frac{1}{\epsilon})$. Such algorithms belong to the so called $FPTAS$ (Fully Polynomial Time Approx. Scheme) class. In general we have the following two relevant notions:

Definition 4.2.2 *FPTAS (Fully Polynomial Time Approx. Scheme): An algorithm which achieves an $(1+\epsilon)$ approximation in time $poly(size, 1/\epsilon)$, for any $\epsilon > 0$.*

Definition 4.2.3 *PTAS(Polynomial Time Approx. Scheme): Approximation scheme which achieves an $(1+\epsilon)$ approximation in time $poly(size)$, for any $\epsilon > 0$.*

Note that an $FPTAS$ is the best algorithm possible for an NP-Hard problem. As mentioned before, knapsack also has a pseudo polytime algorithm. In fact, problems with an $FPTAS$ often have pseudo polytime algorithms. The following theorem formalizes this.

Theorem 4.2.4 *Suppose that an $\mathcal{NP}$ Hard optimization problem has an integral objective function, and the value of the function at its optimal solution is bounded by some polynomial in the size of the problem in unary, then an $FPTAS$ for that problem implies an exact pseudo-polytime algorithm.*

Proof: Suppose that $B = poly(size)$, where *size* is the size of the problem in unary, upper bounds the optimal objective function value. Then we pick $\epsilon = \frac{1}{2B}$ an run the $FPTAS$ with this value. Then

$$ALG \leq OPT\left(1 + \frac{1}{2B}\right) < OPT + 1$$

Since the objective function is integral, $ALG = OPT$, and we obtain an optimal solution. The algorithm runs in time $poly(size)$. ■

References

[1] M. Andrews, L. Zhang. Hardness of the undirected edge-disjoint paths problem. In: *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing (STOC)* (2005), pp: 276–283

[2] C. Chekuri, S. Khanna. Edge disjoint paths revisited. In: *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2003), pp: 628–637

[3] V. Guruswami, S. Khanna, R. Rajaraman, B. Shepherd, M. Yannakakis. Near-optimal hardness results and approximation algorithms for edge-disjoint paths and related problems. In: *Proceedings of thirty-first annual ACM symposium on Theory of computing (STOC)* (19993), pp: 19–28

[4] J. Kleinberg. Approximation Algorithms for Disjoint Paths Problems. *Ph.D Thesis, Dept. of EECS, MIT* (1996).

CS880: Approximations Algorithms	
Scribe: Tom Watson	**Lecturer:** Shuchi Chawla
Topic: Bin Packing and Euclidean TSP	**Date:** 2/6/2007

In the previous lecture, we saw how dynamic programming could be employed to obtain an FPTAS for the Knapsack problem. The key idea was to morph the given instance into another instance with additional structure — namely that the item profits weren't too large — that allowed us to solve it exactly, and such that an optimal solution for the morphed instance could be used to construct a near-optimal solution for the original instance. In this lecture, we will further explore this idea by applying it to the Bin Packing and Euclidean TSP problems. For the Bin Packing problem, our morphed instanced will have a solution space that is small enough to search exhaustively. For the Euclidean TSP problem, we will place geometric contraints on the morphed instance that allow us to solve it exactly using dynamic programming.

5.1 Bin Packing

5.1.1 The Problem

The Bin Packing problem is, in a sense, complementary to the Minimum Makespan Scheduling problem, which we studied in a previous lecture. In the latter problem, the goal is to schedule jobs of various lengths on a fixed number of machines while minimizing the makespan, or equivalently to pack items of various sizes into a fixed number of bins while minimizing the largest bin size. We now consider the problem where we swap the roles of constraint and objective: all bins have a fixed size, and we wish to minimize the number of bins needed.

Definition 5.1.1 (Bin Packing) *Given items with sizes $s_1, \ldots, s_n \in (0, 1]$, pack them into the fewest number of bins possible, where each bin is of size 1.*

Note that the assumption that the bins are of size 1 is without loss of generality, since scaling the bin size and all item sizes by the same amount results in an equivalent instance.

It is easy to see that Bin Packing is NP-hard by a reduction from the following problem.

Definition 5.1.2 (2-Partition) *Given items with sizes $s_1, \ldots, s_n$, can they be partitioned into two sets of equal size?*

Clearly, an instance of 2-Parition is a yes-instance if and only if the items can be packed into two bins of size $\frac{1}{2}\sum_{i=1}^{n} s_i$. Thus a polynomial-time algorithm for Bin Packing would yield a polynomial-time algorithm for 2-Partition. In fact, even a $(3/2 - \epsilon)$-approximation algorithm for Bin Packing would yield a polynomial-time algorithm for 2-Partition: on no-instances it would clearly use at least three bins, but on yes-instances it would use at most $(3/2 - \epsilon)2 < 3$ bins.

Theorem 5.1.3 *For all $\epsilon > 0$, Bin Packing is NP-hard to approximate within a factor of $3/2 - \epsilon$.*

Corollary 5.1.4 *There is no PTAS for Bin Packing unless $P = NP$.*

The above result exploited the fact that OPT could be small. Can we do better if OPT is large? The answer is yes. We will obtain an *asymptotic* PTAS, where we only require a $(1+\epsilon)$-approximate solution if OPT is sufficiently large.

Definition 5.1.5 *An* asymptotic PTAS *is an algorithm that, given $\epsilon > 0$, produces a $(1+\epsilon)$-approximate solution provided $OPT > C(\epsilon)$ for some function C, and runs in time polynomial in n for every fixed ϵ.*

In particular, we will obtain the following result.

Theorem 5.1.6 *There is an algorithm for Bin Packing that, given $\epsilon > 0$, produces a solution using at most $(1+\epsilon)OPT + 1$ bins and runs in time polynomial in n for every fixed ϵ.*

Corollary 5.1.7 *There is an asymptotic PTAS for Bin Packing.*

Proof: Given $\epsilon > 0$, running the algorithm from Theorem 5.1.6 with parameter $\epsilon/2$ yields a solution using at most $(1 + \epsilon/2 + 1/OPT)OPT$ bins, which is at most $(1+\epsilon)OPT$ provided $OPT > 2/\epsilon$. ■

The following algorithm is due to W. Fernandez de la Vega and G. Lueker [4].

5.1.2 The Algorithm

We seek to prove Theorem 5.1.6. Given an instance I of Bin Packing, we would like to morph it into a related instance that can be solved optimally, and for which an optimal solution allows us to construct an near-optimal solution for the original instance I. Our strategy will be to reduce the solution space so it is small enough to be searched exhaustively. One idea is to throw out small items, since intuitively the large items seem to be the bottleneck in finding a good solution. Another idea is to make sure there aren't too many different item sizes. The following result confirms that these ideas accomplish our goal.

Theorem 5.1.8 *There is a polynomial-time algorithm that solves Bin Packing on instances where there are at most K different sizes of items and at most L items can fit in a single bin, provided K and L are constants.*

Proof: Call two solutions equivalent if they are the same up to the ordering of the bins, the ordering of the items within each bin, and the distinguishing of items of the same size. We will show that there are polynomially many nonequivalent solutions, and thus an optimal solution can be found by exhaustive search.

The number of configurations for a single bin is at most K^L, even if we distinguished between different orderings of the items, since a configuration can be specified by the size of each of the at most L items in the bin. The important thing is that it is a constant.

If we are not careful about only counting nonequivalent solutions, we might reason that since a solution uses at most n bins, each of which can be in one of at most K^L configurations, there are at most $(K^L)^n$ solutions. This bound is not good enough, and we can do better by remembering that it only matters how many bins of each configuration there are, not what order they're in. If x_i denotes the number of bins with the ith configuration, then nonequivalent solutions correspond

to nonnegative integral solutions to the equation

$$x_1 + x_2 + \cdots + x_{K^L} \leq n,$$

of which there are at most

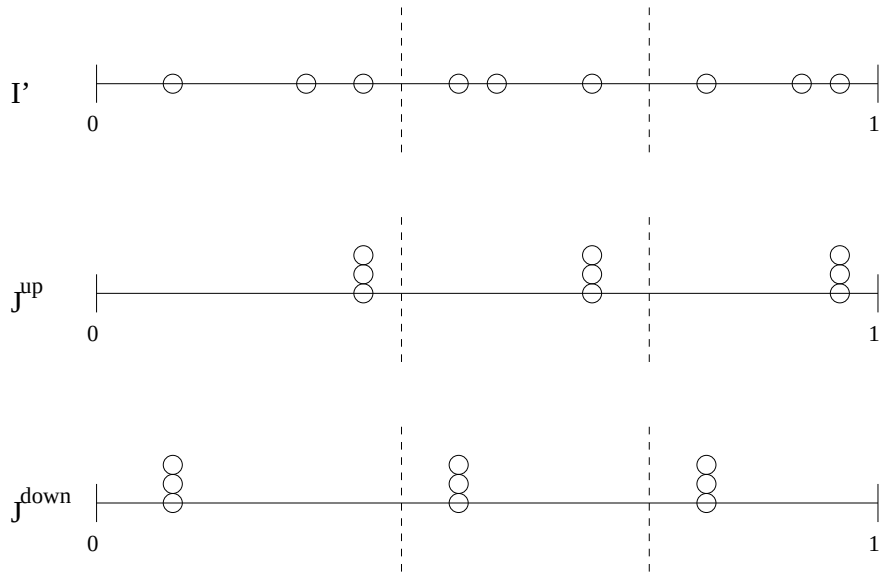
$$\binom{n + K^L}{K^L}$$

by a classic combinatorial argument. This bound is at most a polynomial of degree K^L. ■

In light of the previous theorem, we pause to remark that in contrast to the clever FPTAS for Knapsack we saw in the last lecture, the running time of the algorithm we are developing for Bin Packing is prohibitively expensive. Typically, PTAS's have a bad dependence on $1/\epsilon$. For this reason, PTAS's are not usually very practical and are of primarily theoretical interest.

Given instance I, we seek to morph it into an instance where Theorem 5.1.8 applies. We will first obtain an instance I' by throwing out all items of size less than ϵ. We will worry about packing these items later. Now at most $L = \lceil 1/\epsilon \rceil$ items can fit into any one bin.

There could still be as many as n different item sizes, so we need to morph I' further to get an instance with a constant number K of different item sizes. One way to do this is to consider the items in sorted order, partition them into K groups, and round each item's size up to the largest size in its group, yielding an instance J^{up}. Another way is to round each item's size down to the smallest size in its group, yielding an instance J^{down}.



Neither of these two possibilities seems ideal. We want our new instance to satisfy the following two (informal) properties.

(1) Given a solution to the new instance, it is easy to construct a comparable solution to I'.

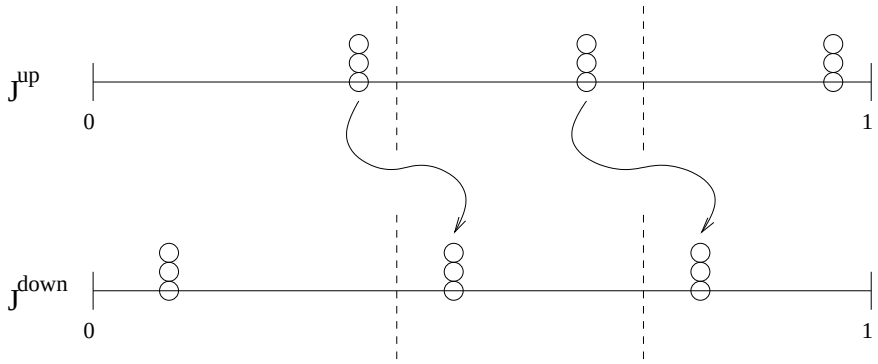
(2) The optimum value of the new instance isn't too much worse than the optimum value of I'.

The instance J^{up} satisfies (1), since when we go back to instance I', the items can only shrink, which means that the same solution is still feasible. However, it doesn't seem to satisfy (2), since if we try to translate the optimal solution for I' into a solution for J^{up}, all the bins could overflow, requiring many new bins to be opened up. The instance J^{down} satisfies (2) since the optimal solution for I' immediately yields a feasibile solution for J^{down} with the same number of bins. However, it doesn't seem to satisfy (1) since a solution to J^{down} can't generally be translated to a solution for I' without lots of bins overflowing.

It turns out that if we select the parameters in the right way, J^{up} *does* satisfy property (2). We can argue this by comparing J^{up} to J^{down}. (However, our final algorithm will apply the algorithm from Theorem 5.1.8 only to J^{up}, not to J^{down}.)

We consider the items in sorted order and partition them into $K = 1/\epsilon^2$ groups of size $Q = n\epsilon^2$ each , breaking ties arbitrarily. (The last group might have fewer items.) We tacitly ignore the pedantic details associated with rounding these quantities to integers, as these details distract from the essense of the algorithm. We obtain J^{up} by rounding each item's size up to the size of the largest item in its group, and similarly obtain J^{down} by rounding each item's size down to the size of the smallest item in its group. For each of these instances, there are at most K different item sizes.

Our algorithm will actually construct J^{up} and apply the algorithm from Theorem 5.1.8 to it. The resulting solution is also a feasible solution for I', as noted above. We would like to show that the number of bins this solution uses is not too much more than $OPT(I')$. As usual, we will need a lower bound on $OPT(I')$ to compare with. Observe that $OPT(J^{\text{down}}) \leq OPT(I')$ since each feasible solution of I' is also a feasible solution of J^{down}. We will use this lower bound. How much worse than $OPT(J^{\text{down}})$ can $OPT(J^{\text{up}})$ be? The critical observation is that a solution to J^{up} can be constructed from a solution to J^{down} by taking each group of items, except the last, and moving them to the locations occupied by the items in the next group, and assigning each item of the last group to its own new bin. Since all groups (except possibly the last) have the same size, this correspondence can be made.



Since the size of every item in a group in J^{up} is at most the size of every item in the next group in J^{down}, it follows that every item of J^{up} is at most the size of the item of J^{down} whose place it's taking. (Note that the locations of the items of the first group of J^{down} aren't filled by any items of J^{up}.) This shows that the resulting solution of J^{up} is feasible. Moreover, it has at most Q additional bins, one for each item in the last group. We conclude that

$$\begin{aligned} OPT(J^{\text{up}}) &\leq OPT(J^{\text{down}}) + Q \\ &\leq OPT(I') + Q \\ &= OPT(I') + n\epsilon^2 \\ &\leq OPT(I') + OPT(I')\epsilon \\ &= (1+\epsilon)OPT(I'). \end{aligned}$$

In going from the first line to the second, we used our lower bound on $OPT(I')$. In going from the third line to the fourth, we use a *second* lower bound on $OPT(I')$, namely that no solution can do better than to pack every bin completely, which would use at least $n\epsilon$ bins (since every item is of size at least ϵ). This reveals why we chose Q the way we did: to make sure that the number of extra bins we used in our comparison of $OPT(J^{\text{up}})$ to $OPT(J^{\text{down}})$ was at most $\epsilon OPT(I')$.

With this result in hand, we can prove the main result of this section.

Proof of Theorem 5.1.6: We are given instance I and $\epsilon > 0$. First, we construct I' by throwing out all items of size less than ϵ, and then we construct J^{up} and solve it optimally using the algorithm of Theorem 5.1.8. The resulting solution, we have argued, is a feasible solution to I' using at most $(1+\epsilon)OPT(I')$ bins. The running time so far is

$$O(n^{K^L}) = O(n^{O(1/\epsilon)^{O(1/\epsilon)}}) = poly(n).$$

But we still have to pack the items of size less than ϵ. For this, we can make use of the empty space in the bins used by our current packing. A natural thing to do is use a greedy strategy: pack each item of size less than ϵ into the first bin it fits in, only opening a new bin when necessary. We next argue that this does the job.

If the greedy phase does not open any new bins, then the number of bins is at most $(1+\epsilon)OPT(I') \leq (1+\epsilon)OPT(I)$ as shown above. Here we used the trivial lower bound $OPT(I') \leq OPT(I)$. If the greedy phase does open a new bin, then at the end, all but the last bin must be more than $1-\epsilon$ full (since otherwise the item that caused the last bin to be opened would have fit into one of them). Denoting by ALG the number of bins used by our algorithm's solution, we conclude that

$$(1-\epsilon)(ALG - 1) \leq \sum_{i=1}^{n} s_i \leq OPT(I).$$

Here we have used a second lower bound on $OPT(I)$. It follows that

$$ALG \leq \frac{1}{1-\epsilon} OPT(I) + 1.$$

We have $\frac{1}{1-\epsilon} = 1 + O(\epsilon)$ provided ϵ is at most some fixed positive constant, which is no loss of generality. Since we may run this algorithm with a smaller ϵ parameter than the one we are given, this suffices to prove the theorem. ■

It's worth noting what prevents this approach from giving a PTAS (instead of an asymptotic PTAS). In the final greedy phase, we can't say anything about how full the last bin to be opened is. This prevents us from applying the $\sum_{i=1}^{n} s_i \leq OPT(I)$ bound to this last bin. Thus we get an extra $+1$ term floating around, which as a fraction of OPT, only goes down as OPT goes up, not as ϵ goes down.

There are a number of heuristics for Bin Packing that give good worst case performance. See [3] for a survey.

Finally, we remark that the practical importance of the Bin Packing problem spawned research into algorithms for generalizations of this problem. For example, one can consider packing higher-dimensional items into higher-dimensional bins. This generalization presents tricky issues not present in the one-dimensional case that we considered. In the 2-dimensional case under certain restrictions on the packing, one can get an asymptotic PTAS [2]. For higher (but still constant) dimensions, constant factor approximations are known. However, we will not explore these results in this course.

5.2 Euclidean TSP

5.2.1 The Problem

In this section we consider the following practically important special case of the traveling salesperson problem (TSP).

Definition 5.2.1 (Euclidean TSP) *Given n points in the d-dimensional Euclidean metric space (for some fixed d), find a minimum length tour that visits all of them.*

For simplicity, we will restrict our attention to the 2-dimensional case $d = 2$. The algorithm we will present generalizes easily to higher dimensions.

The Metric TSP problem, for which we obtained a 3/2-approximation algorithm in a previous lecture, is known to be APX-hard. Euclidean TSP in the plane is NP-hard, but it is conceivable that the special structure of the Euclidean case allows us to overcome the obstacle that prevents us from obtaining a PTAS for Metric TSP. We will show that this is, in fact, the case.

Theorem 5.2.2 *There is a PTAS for the Euclidean TSP problem.*

This result was proved independently by Arora [1] and Mitchell [5]. We will present Arora's algorithm and analysis.

We pause to emphasize the distinction between the problem at hand and the restriction of TSP to planar metrics. A planar metric is one that arises as the shortest path metric of a planar graph. The edge weights on this planar graph can be selected in any way, and need not have anything to do with Euclidean distance. There is also a PTAS for the TSP on planar metrics, but it is quite different from the algorithm we will present.

5.2.2 The Algorithm

Following the theme of the Bin Packing result and the Knapsack result from the previous lecture, our overall strategy will be to morph the given instance into a related instance that has additional structure that allows us to solve it optimally, and which allows us to construct a "good" solution for the original instance from the optimal solution for the morphed instance. In short, we will first modify the instance by moving each point a little bit so that it is in a convenient location, and then we will further modify the instance by imposing some geometric constraints on the tours. We will be able to solve the new instance exactly by dynamic programming, and then construct a tour for the original instance without the cost growing by too much.

Before getting to the details, we first make the simplifying assumption that the smallest bounding square of the given points has side length exactly n^2. This is without loss of generality since we can scale the given instance without affecting its solution set in any significant way. We denote the resulting instance by I. We also observe that the smallest bounding square must have two points on opposite sides (either one on the left side and one on the right side, or one on the bottom and one on the top). Since every tour must traverse the distance from one of these points to the other and back, we get the following lower bound on the optimum, which will be useful later.

Lemma 5.2.3 *$OPT(I) \geq 2n^2$.*

Now we describe the first modification we will make to our instance. We round the coordinates of each input point to integers values, yielding an instance I'. We can argue that this modification doesn't prevent us from getting a good approximation.

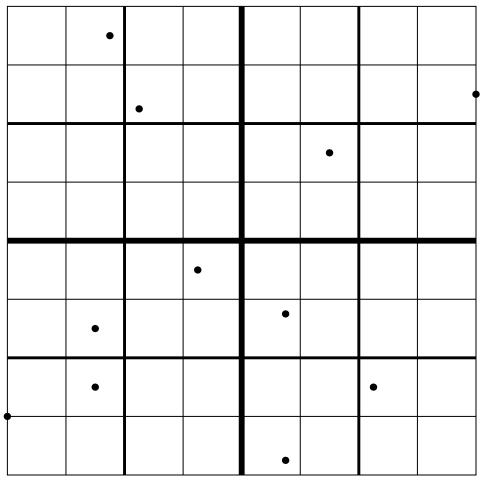
Lemma 5.2.4 *If I' can be approximated within factor $1+\epsilon$, then I can be approximated within factor $1+\epsilon+4/n$.*

Proof: Given a tour of I' of cost $ALG' \leq (1+\epsilon)OPT(I')$, it suffices to show that the corresponding tour of I is of cost $ALG \leq (1+\epsilon+4/n)OPT(I)$. Note that each point in I is at most $\sqrt{2}$ distance from its location in I'. Now given a tour in one of these two instances, the tour in the other instance that follows the same path but "sidesteps" at each input point to visit it's new location and come back has additional total cost at most $2\sqrt{2}n$ and is at least as long as the tour that visits the input points along straight line paths. It follows that corresponding tours in the two instances can differ in cost by at most $2\sqrt{2}n$. Hence, $OPT(I') \leq OPT(I) + 2\sqrt{2}n$ and $ALG \leq ALG' + 2\sqrt{2}n$, and thus

$$\begin{aligned}
ALG &\leq (1+\epsilon)OPT(I') + 2\sqrt{2}n \\
&\leq (1+\epsilon)(OPT(I) + 2\sqrt{2}n) + 2\sqrt{2}n \\
&= (1+\epsilon)OPT(I) + (2+\epsilon)2\sqrt{2}n \\
&\leq (1+\epsilon)OPT(I) + (2+\epsilon)\sqrt{2}\frac{OPT(I)}{n} \\
&\leq (1+\epsilon+4/n)OPT(I).
\end{aligned}$$

We have used Lemma 5.2.3 in going from the third line to the fourth. In going from the fourth line to the fifth, we have assumed that $\epsilon \leq 2\sqrt{2}-2$, which is no loss of generality. Thus given a $(1+\epsilon)$-approximate solution to I', the corresponding solution to I is $(1+\epsilon+4/n)$-approximate. ■

We would like to attempt to solve our morphed instance exactly by dynamic programming. A natural line of attack is to break up the bounding square into four equal parts, and try to recombine solutions to these four subproblems into a solution for the original problem. Each of these subproblems would be broken up into four more subproblems in a similar way, and so on, leading to a 4-ary tree of subproblems.



This motivates the modification we made earlier of rounding all coordinates to integer values. We would like our base case to be when there's just a single point, and this modification ensures that our tree of subproblems won't have to be too deep in order to separate two points that are close to each other.

There seems to be a problem with this naive approach: it's not clear how recombine optimal tours for the four subproblems into an optimal tour for the subproblem at hand. For example, we could have the pathological case where the optimal tour zigzags across one of the dividing lines. Our subproblems need to take into account how they interact with each other across the dividing lines.

On each dividing line we will introduce some number of equidistant *portals* and only consider *portal-proper* tours: ones that only cross dividing lines at portals. Then for each node in the 4-ary tree of subproblems, we will actually have many subproblems, corresponding to different ways of entering and exiting the square at its portals. This increases the number of subproblems, but by setting the parameters properly, we will be able to keep the number under control. This also allows us to form the optimal solution to the subproblem at hand by trying all possible ways of specifying how its four subproblems interact with each other at the portals, and stitching the solutions together. We will require that the number of portals be small enough that we can do this quickly, but large enough that the optimum tour length doesn't deteriorate by too much when we impose this geometric restriction.

More details on this construction will be provided in the next lecture.

References

[1] S. Arora. Polynomial Time Approximation Schemes for Euclidean Traveling Salesman and other Geometric Problems. In *FOCS*, 1996, pp. 2-12.

[2] N. Bansal, A. Lodi, and M. Sviridenko. A Tale of Two Dimensional Bin Packing. In *FOCS*, 2005, pp. 657-666.

[3] E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation Algorithms for Bin Packing: A Survey. In *Approximation Algorithms for NP-Hard Problems*, Dorit S. Hochbaum (editor), PWS Publishing Company, 1997, pp. 46-93.

[4] W. Fernandez de la Vega and G. Lueker. Bin Packing Can Be Solved within $1 + \epsilon$ in Linear Time. In *Combinatorica*, 1(4), 1981, pp. 349-355.

[5] J. S. B. Mitchell. Guillotine Subdivisions Approximate Polygonal Subdivisions: A Simple Polynomial-Time Approximation Scheme for Geometric TSP, k-MST, and Related Problems. In *SIAM Journal on Computing*, 28, 1999, pp. 1298-1309.

CS880: Approximations Algorithms	
Scribe: Dave Andrzejewski	**Lecturer:** Shuchi Chawla
Topic: Euclidean TSP (contd.)	**Date:** 2/8/07

Today we continue the discussion of a dynamic programming (DP) approach to the Euclidean Travelling Salesman Problem (TSP). Finally, a randomized modification in introduced which acheives an expected approximation factor of $1 + 2\epsilon$.

6.1 Euclidean TSP

6.1.1 Intro

As introduced in the previous lecture, the Euclidean TSP invovles finding the minimum cost tour of a set of points in a plane with a Euclidean metric.

The approaches to Bin-Packing and Knapsack discussed in the previous lecture work by first transforming or restricting the original problem, then using DP to obtain an exact solution to the transformed problem. This is the strategy we will apply to Euclidean TSP.

First, we will assume without loss of generality that the points are contained within a minimum bounding box with side length n^2. Second, we will assume all points have integer coordinates. It can be shown that this modification can add no more than $2\sqrt{2}n$ to the total length of an optimal tour in the transformed instance (see previous lecture). Also, note that this rounding may 'collapse' several nearby points to the same coodrdinates. This means that the distance between any two points in our tranformed instance must be either 0 or ≥ 1.

6.1.2 Divide anhd conquer

Our DP approach divides the minimum bounding box into 4 equal subboxes. These boxes are further subdivided, and so on. Following the standard DP strategy, we will find optimal subpaths, join them together, and then move up to the next level until we have a full solution.

Our key problem then is determining how to combine these subpaths. Obviously we cannot simply consider all possible start/end point pairs within each subpath. The solution is to subdivide each box into 4 boxes using 4 lines we will call the "level i lines", where i is the current recursion level. On each of these lines we then place $2m$ equidistant "portal" points. We then restrict our attention to paths which only cross level i lines through these portals.

Definition 6.1.1 *A tour is* portal proper *if it only crosses level i lines through the portal points.*

We now place further restrictions on the paths we will consider.

Definition 6.1.2 *A tour is* proper *if*

1. *it is portal proper*
2. *it crosses each portal ≤ 2 times*

3. it only self-crosses at portals

We need to show that these new restrictions do not affect the length of a portal proper tour.

Lemma 6.1.3 *If T is a portal proper tour, then we can find a proper tour T' that is not longer.*

Proof: First, we must show how any self-crossing can be eliminated without any additional length. A diagram of a simple example shows how any crossover can be removed, resulting in a path that is no longer than the original.

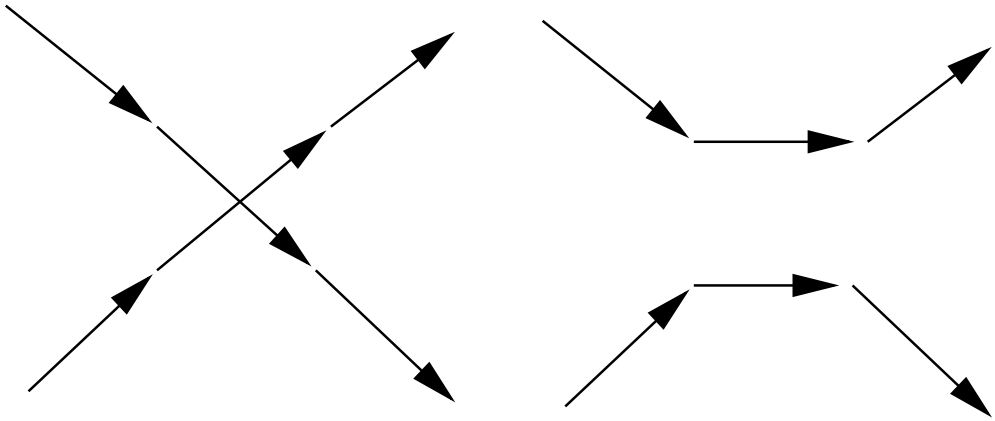


Figure 6.1.1: A self-crossing can be removed without additional length.

Second, we must show how any path which crosses a portal > 2 times can be modified to cross ≤ 2 times without additional length. This can be accomplished by having paths simply "turn back" instead of going through the portal again. Any odd number of portal crossings can be reduced to a single crossing, and any even number of crossings can be reduced to two, as shown below.

■

6.1.3 DP for proper tours

Now we need to analyze the DP approach for finding the optimal proper tour. The key questions here are:

1. how long to solve one subproblem?
2. how many subproblems have we created?

6.1.3.1 Subproblem size

First we will consider the number of possible solutions for a single subproblem. For a given ϵ, let us choose m to be a power of 2 in the range

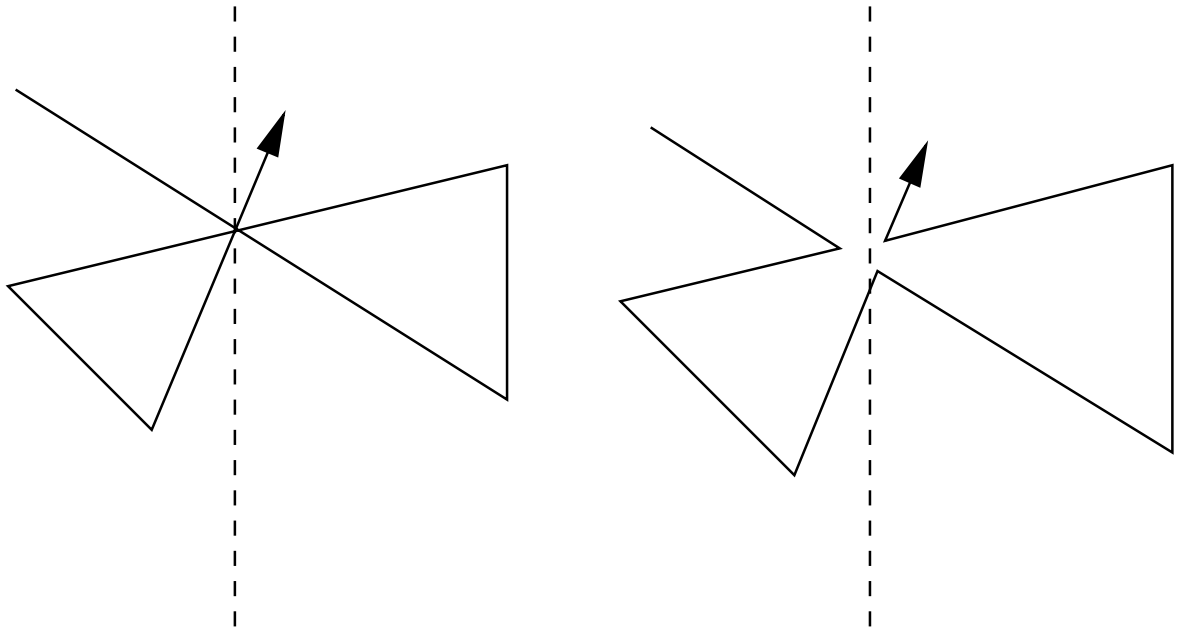


Figure 6.1.2: An odd number of portal crossings reduced to a single crossing.

$$\left[\frac{\log n}{\epsilon}, \frac{2\log n}{\epsilon}\right] \tag{6.1.1}$$

Then for any given box, the maximum number of portals on a side is m. The maximum number of sides with portals is 4, so the total number of portals on the box is then $\leq 4m$. Since we are restricting ourselves to paths that only use each portal 0,1, or 2 times the total number of portal usage assignments $3^{4m} = n^{O(\frac{1}{\epsilon})}$. Since every path which enters the box must also leave it, we can throw out all portal usage assignments which sum to an odd number of portal crossings. Say a given portal usage assignment uses $2r$ portals.

Given the portal usage assignment, how many possible paths are there? The "no self-cross" property of a proper tour allows us to further constrains the number of allowable paths. Within a single box, there is a bijection between portal matchings which satisfy the "no self-cross" constraint and balanced arrangements of parentheses. This is most easily seen by arbitrarily choosing a 'starting' portal on the box and going around the box clockwise to order the portals. Once a path 'enters' through a portal, any paths entering 'afterwards' must exit before the first path does. Portals used twice can be considered equivalent to two adjacent portals for the purposes of this analysis.

Starting from the lower left-hand portal and going clockwise, the diagram below illustrates a portal matching equivalent to the parenthesization "(()) (() ())".

The number of balanced parenthesizations for r pairs of parentheses is equal to the r^{th} Catalan number [2], which is bounded from above by $2^{2r} = n^{O(\frac{1}{\epsilon})}$.

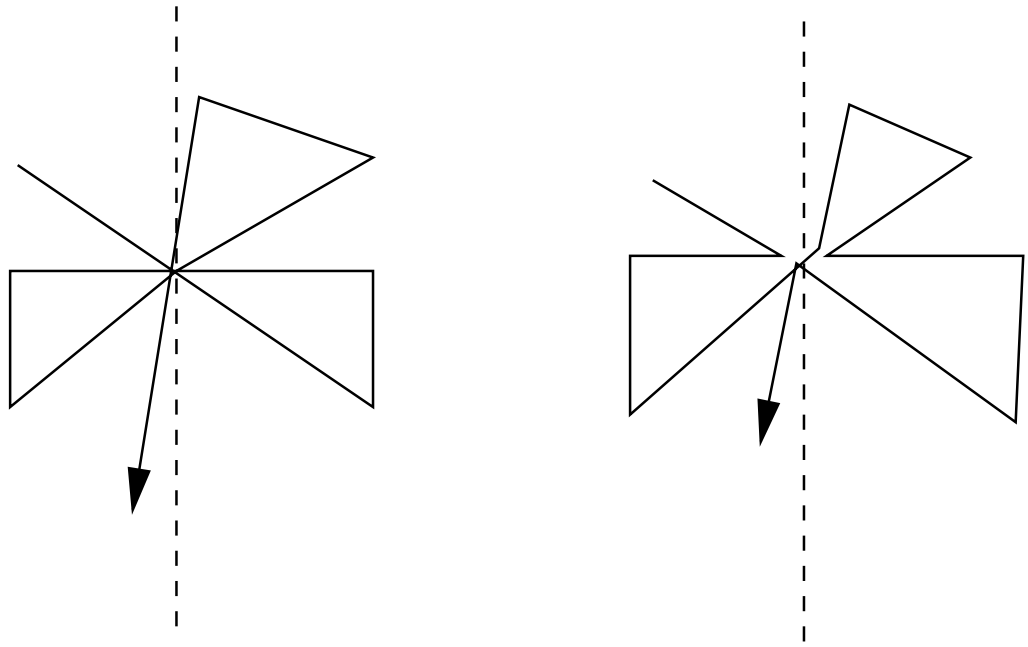


Figure 6.1.3: An even number of portal crossings reduced to two crossings.

This finally tells us that the total number of possible portal proper paths for a given box is $n^{O(\frac{1}{\epsilon})}$.

6.1.3.2 Number of subproblems

We now need to determine how many subproblems we have created. Each division creates 4 boxes, so the total number of problems is given by 4^L where L is the number of recursion levels.

Since we are only allowing integer coordinates, and our minimum bounding box has side length n^2, this means that the depth of our recursion is $L = 2 \log n$. Combining this with the previous result gives us $4^{2 \log n} = n^4$ squares at the lowest level level of recursion. This result also could have been arrived at by noticing that we have restricted the problem the integer coordinates and that our minimum bounding square has side length n^2. Since the smallest boxes cover exactly one point, we get n^4 one-point boxes.

6.1.3.3 Putting it all together

For any given box, we have $n^{O(\frac{1}{\epsilon})}$ valid portal matchings (a 'visit'). For a given visit, the portal usages on the outer portals are fixed, so we then need to consider only the compatible visits for the 4 sub-boxes. Since each sub-box has $n^{O(\frac{1}{\epsilon})}$ valid visits, there are still only $n^{O(\frac{1}{\epsilon})}$ sets of 4 visits to consider, only some of which will be compatible with the larger box portal matching and with each other. The cost of a compatible set of 4 visits is then the sum of the optimal visit costs for each sub-box, which have already been calculated at the lower level of recursion. The cost of the lowest cost path for is then stored as the optimal cost for this particular visit of the larger box.

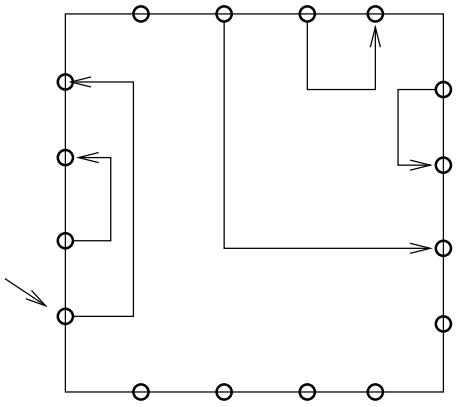


Figure 6.1.4: A portal proper path.

This process is started at the leaves and then proceeds up the tree. Combining the per subproblem cost with the total number of subproblems gives us a final result of $n^{O(\frac{1}{\epsilon})}$ for the cost of our optimal DP algorithm for proper tours.

6.1.4 Proper tour vs OPT

Now that we know we have a polynomial time algorithm for computing optimal proper tours. But how much worse is the optimal proper tour than OPT?

To determine this, we construct a worst case OPT tour which has to go out of its way to go through portals. Assuming that we are making a detour at each of the $2m$ portals and that each detour adds $\leq \frac{n^2}{2m}$, then we are adding $\approx n^2$ total distance beyond OPT. Since we defined n^2 to be the side length of the minimal bounding box, $2n^2$ forms a lower bound on OPT. In our worst-case scenario, saying that $OPT \approx 2n^2$ and our detours add $\approx n^2$ means that we have a constant-factor approximation.

6.1.5 Randomized version

We can see that the worst-case example was very specific to our division line placement. This provides the intuition for a randomized approach, which acheives an expected cost within a $(1+2\epsilon)$ multiple of OPT.

The randomized version makes the top-level divisions at an offset from the $n^2/2$ center lines. The offsets are chosen uniformly. The rest of the algorithm is unchanged, giving virtually identical analysis.

The one difference is that we can now compute the expectation of the number of line corssings.

Lemma 6.1.4 *$E(c_i) = OPT\frac{2^{i+2}}{M}$ where c_i is the number of times the optimal path crosses a level i line, and M is the length of the level i line.*

Proof: Divide the OPT path into segments with length δ. The probability that a segment crosses

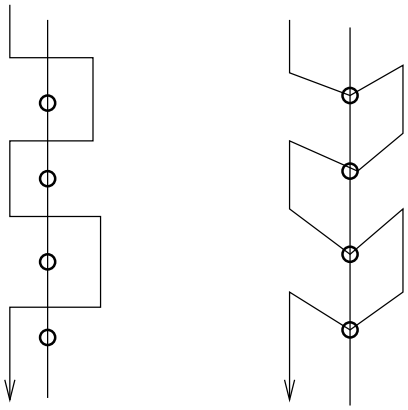


Figure 6.1.5: A worst-case portal proper path.

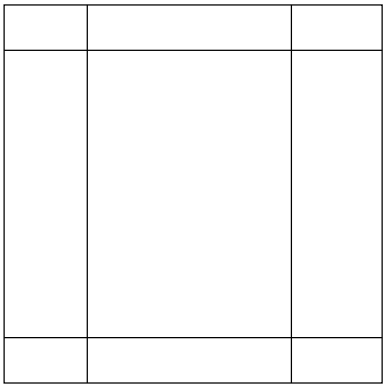


Figure 6.1.6: Randomized level 1 division.

the vertical level i line is then $\leq \frac{2\delta}{M}$. The expected number of segments which cross the vertical level i line is then given by

$$\frac{OPT}{\delta}\frac{2\delta}{M} = \frac{2OPT}{M} \tag{6.1.2}$$

Since the same analysis holds for the horizontal line, the expected number of crossings is then $\frac{4OPT}{M}$.

To take the crossings at each lower level into account, notice that each lower level has twice as many lines as the level above it.

The expected number of crossings at level i is then equal to the our previous calculation times 2^i.

$$2^i(\frac{4OPT}{M}) = \frac{2^{2+i}OPT}{M} \tag{6.1.3}$$

■

Now that we have the expected number of crossings, we can compute the expected additional detour cost. Our definition of m with respect to $\log n$ will now play a crucial role.

Theorem 6.1.5 *An OPT tour can be made proper at an increased length cost* $\leq 2\epsilon OPT$.

Proof: First we need to calculate the worst-case distance from a crossing point to a portal on a level i line. Since on level i we have $2^{i+1}m$ portals per line, the distance must be $\leq \frac{M}{2^{i+1}m}$.

Now we can see that even though the expected number of crossings grows exponentially with i, the distance to the nearest portal shrinks exponentially with i. The expected extra distance d necessary to go through the portals can now be calculated.

$$E(d) \leq \sum_i \frac{M}{2^{i+1}m} \frac{2^{2+i}OPT}{M} = \frac{OPT}{m} 2\log n \tag{6.1.4}$$

We recall that $m \propto \log n$, and substitute in our definition of m.

$$E(d) \leq 2\epsilon OPT \tag{6.1.5}$$

■

It is now clear that our randomized version has expected cost arbitrarily close to OPT.

$$E(ALG) \leq (1 + 2\epsilon)OPT \tag{6.1.6}$$

Now that we have a bound on the expected error, we can use the Markov inequality to get a probability bound on how much worse our solution is than OPT. This probability can then be made arbitrarily small by repeatedly running the randomized algorithm.

A de-randomization of this algorithm is also possible [1].

References

[1] S. Arora. Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems. In *Journal of the ACM*, 1996.

[2] R. A. Brualdi. Introductory Combinatorics, 4th ed. New York: Elsevier, 1997.

CS880: Approximations Algorithms	
Scribe: Chi Man Liu	**Lecturer:** Shuchi Chawla
Topic: Local Search: Max-Cut, Facility Location	**Date:** 2/13/2007

In previous lectures we saw how dynamic programming could be used to obtain PTAS for certain NP-hard problems. In this lecture we will discuss local search and look at approximation algorithms for two problems — Max-Cut and Facility Location.

7.1 Local Search

Local search is a heuristic technique for solving hard optimization problems and is widely used in practice. Suppose that a certain optimization problem can be formulated as finding a solution maximizing an objective function in the solution space. A local search algorithm for this problem would start from an arbitrary or carefully chosen solution, then iteratively move to other solutions by making small, local changes to the solution, each time increasing the objective function value, until a local optimum is found.

We view the solution space as a graph where vertices are the candidate solutions. An edge exists between two solutions if we can move from one to the other by making some small changes. A solution is a global optimum if its objective function value is maximized (or minimized, depending on the context) over all candidate solutions. A solution is a local optimum if none of its neighbor solutions has a greater (or smaller) objective function value.

When we apply local search to solving optimization problems we require that the solution graph has low (polynomially bounded) vertex degrees, so that at each vertex only polynomial time is needed to find a "good" neighbor. In order to show that the algorithm runs in polynomial time, we need to show that starting from any solution we can reach a local optimum in a polynomial number of steps. It doesn't suffice to show that the diameter of the solution graph is polynomial. We will see a few different arguments including a potential function based argument. And lastly, to show that the algorithm is indeed a good approximation algorithm, we must prove that any local optimum is nearly as good as a global optimum.

In the following sections we will see how local search can be used as approximation algorithms for two problems. The first problem we consider is the *Max-Cut* problem. The second problem is the *Facility Location* problem for which the analysis is a bit more involved. We will defer the last part of the analysis to the next lecture.

7.2 Max-Cut

7.2.1 The Problem

We first review the notion of a *cut* in graph theory.

Definition 7.2.1 *Let $G = \{V, E\}$ be a weighted graph where each edge $e \in E$ has a weight w_e. A*

cut *is a partition of* V *into two subsets* S *and* $S' = V \backslash S$*. We simply denote a cut by either one of its subsets. The value of a cut* S *is*

$$c(S) = \sum_{\substack{(s,s') \in E \\ s \in S, s' \in S'}} w_{s,s'}.$$

Now we can define the Max-Cut problem on weighted graphs.

Definition 7.2.2 (The Max-Cut Problem) *Given a weighted graph* $G = \{V, E\}$ *where each edge* $e \in E$ *has a positive integral weight* w_e*, find a cut in* G *with maximum cut value.*

While the Min-Cut problem is polynomial-time solvable by reducing it to Maximum Flow, the Max-Cut problem is known to be NP-hard. It has been shown that approximating Max-Cut to a factor of 17/16 is still NP-hard. In the following we present a simple 2-approximation local search algorithm for Max-Cut.

7.2.2 The Algorithm

The algorithm we are going to describe is a straightforward application of local search. The only thing we need to specify is the small changes we are allowed to make in each step. Since each candidate solution is a just a partition of the vertex set V, the following local step seems natural: Two partitions S_1 and S_2 are joined by an edge if and only if $|S_1 \backslash S_2| \leq 1$ and $|S_2 \backslash S_1| \leq 1$. Intuitively this means moving one vertex from one subset to the other. Thus we obtained the following algorithm:

1. Start with an arbitrary partition (for example, $\emptyset$).
2. Pick a vertex $v \in V$ such that moving it across the partition would yield a greater cut value.
3. Repeat step 2 until no such v exists.

Before we move on to prove that this algorithm is 2-approximate, let us first analyze its running time. Each step involves examining at most $|V|$ vertices and selecting one that increases the cut value when moved. This process takes $O(|V|^2)$ time. Since we assume that edges have positive integral weights, the cut value is increased by at least 1 after each iteration. The maximum possible cut value is $\sum_{e \in E} w_e$, hence there are at most such number of iterations. The overall time complexity of this algorithm is thus $O(|V|^2 \sum_{e \in E} w_e)$. For the special case of unweighted graphs (all weights equal to 1), the time complexity becomes $O(|V|^4)$, which is strongly polynomial in the input size.

Note. There exist modifications to the algorithm that give a strongly polynomial running time, with an additional ϵ term in the approximation coefficient. An example of such modifications will be shown when we discuss the Facility Location problem.

We now proceed to prove that the above algorithm is 2-approximate.

Theorem 7.2.3 *The local search algorithm described above gives a 2-approximation to the Max-Cut problem.*

Proof: First of all, we observe that the maximum cut value cannot be larger than the sum of all edge weights, thus giving

$$\sum_{e \in E} w_e \geq OPT.$$

We say that an edge *contributes* to a cut if its endpoints lie in different subsets of the cut. Let S be a cut produced by our algorithm. Let v be a vertex in S. Consider the set E_v of edges incident to v. If we move v from S to $S' = V \backslash S$, edges in E_v that contributed to S become non-contributing, and vice versa. Edges not in E_v are not affected. Since S is a local optimum, moving v to S' does not increase the cut value. Therefore we have

$$\begin{aligned}
\sum_{\substack{u \in S' \\ (u,v) \in E}} w_{u,v} &\geq \sum_{\substack{u \in S \\ (u,v) \in E}} w_{u,v} \\
2 \sum_{\substack{u \in S' \\ (u,v) \in E}} w_{u,v} &\geq \sum_{\substack{u \in S \\ (u,v) \in E}} w_{u,v} + \sum_{\substack{u \in S' \\ (u,v) \in E}} w_{u,v} \\
\sum_{\substack{u \in S' \\ (u,v) \in E}} w_{u,v} &\geq \frac{1}{2} \sum_{u:(u,v) \in E} w_{u,v}.
\end{aligned}$$

Similarly, for any $v' \in S'$ we get

$$\sum_{\substack{u \in S \\ (u,v') \in E}} w_{u,v'} \geq \frac{1}{2} \sum_{u:(u,v') \in E} w_{u,v}.$$

Summing over all vertices, we obtain the desired result:

$$2c(S) = 2 \sum_{\substack{u \in S, v \in S' \\ (u,v) \in E}} w_{u,v} \geq \sum_{e \in E} w_e \geq OPT.$$

■

There are many other 2-approximate algorithms for Max-Cut. One example is the following simple randomized algorithm: Construct the cut by uniformly assigning each vertex to one of of two subsets. It is trivial to show by linearity of expectation that this algorithm produces a cut with an expected value of $\frac{1}{2} \sum_{e \in E} w_e \geq \frac{1}{2} OPT$. This algorithm can be made deterministic by applying derandomization.

As a sidenote, the best known approximation for Max-Cut is 1.134-approximate [1]. This algorithm is based on semidefinite programming. The approximation factor (1.134) arises from geometric arguments and is suspected to be the best possible. (We will learn more about this in a subsequent lecture.) It is not likely that this approximation result can be improved unless P = NP [2].

7.3 Facility Location

7.3.1 The Problem

Suppose there is a set of customers and a set of facilities serving these customers. Our goal is to open some facilities and assign each customer to the nearest opened facility. Each facility has an opening cost. Each customer has a routing cost, which is proportional to the distance (see the next paragraph) traveled. The Facilitiy Location problem asks for a subset of facilities to be opened such that the total cost (opening costs plus routing costs) is minimized. While the aforementioned "distance" could mean geographic distance, in general any metric would do. Recall the definition of a metric:

Definition 7.3.1 *Let M be a set. A* metric *on M is a function $d : M \times M \to \mathbb{R}$ that satisfies the following three properties:*

1. *$d(x, y) \geq 0$; and $d(x, y) = 0$ iff $x = y$ (non-negativity)*
2. *$d(x, y) = d(y, x)$ (symmetry)*
3. *$d(x, z) \leq d(x, y) + d(y, z)$ (triangle inequality)*

The formal definition of the the Facility Location problem is given below.

Definition 7.3.2 (The Facility Location Problem) *Let X be a set of facilities and Y be a set of customers. Let c be a metric defined on $X \cup Y$. For each $i \in X$, let f_i be the cost of opening facility i. Let S be a subset of X. For each $j \in Y$, the routing cost of customer j with respect to S is $r_j^S = \min_{i \in S} c(i, j)$. The total cost of S is $C(S) = C_f(S) + C_r(S)$, where $C_f(S) = \sum_{i \in S} f_i$ is the facility opening cost of S and $C_r(S) = \sum_{j \in Y} r_j^S$ is the routing cost of S. The* Facility Location *problem asks for a subset $S^* \subseteq X$ that minimizes $C(S^*)$ over all subsets of X.*

The requirement that c is a metric seems arbitrary at first sight. However, if we let c be any general weight function, we will be able to reduce Set Cover to this relaxed Facility Location problem, implying that it is unlikely to be approximable within a constant factor (see lecture 3). By imposing the metric constraints, we are able to get a constant factor approximation algorithm for Facility Location.

The Facility Location problem arises very often in operations research. There are many variants of the problem, such as Capacitated Facility Location, in which each facility can only serve a certain number of customers. Another variant is the k-Median problem, in which facilities have no opening costs but at most k of them can be opened.

We remark that the best known approximation algorithm for Facility Location has an approximation coeffcient of 1.52 [3]. On the other hand, it has been shown that approximating this problem within a factor of 1.46 is NP-hard unless NP $\subseteq$ DTIME($n^{\log \log n}$) [4].

7.3.2 The Algorithm

We present a local search algorithm for approximating Facility Location within a factor of $5 + \epsilon$ for any constant $\epsilon > 0$ [5]. In fact, it can be shown that this algorithm is a $(3 + \epsilon)$-approximation [6],

but the analysis is more involved and we will not go through that in class.

We start by specifying the solution graph. Each vertex in the graph represents a subset of facilities to be opened. At each step, we can add a facility to the subset, remove a facility from a subset, or swap a facility in the subset with one outside the subset. The algorithm is as follows:

1. Start with an arbitrary subset of facilities.
2. Carry out an operation (add, remove or swap) that leads to a decrease in the total cost by a factor of at least $(1 - \epsilon/p(n))$, where $p(n)$ is a polynomial to be determined later.
3. Repeat step 2 until such an operation does not exist.

We first bound the running time of this algorithm. Let $n = |X| + |Y|$. For easier analysis we assume that all distances and costs are integers. Observe that the cost of any solution is bounded from above by some polynomial in n, c and f. Let $q(n, c, f)$ be such a polynomial. The cost of the solution after t iterations is at most $(1 - \epsilon/p(n))^t q(n, c, f)$. After $p(n)/\epsilon$ iterations, the cost is at most $q(n, c, f)/e$ since $(1 - x)^{1/x} \approx 1/e$. Thus the algorithm halts after $O(\log q(n, c, f)p(n)/\epsilon)$ steps, which is polynomial in $1/\epsilon$ and the size of the instance (in binary).

Theorem 7.3.3 says that any local optimum in the solution graph is nearly as good as any global optimum. Note that in the analysis, we will assume that in step 2 of the algorithm we carry out an operation as long as there is some improvement, even if the improvement is not by a factor of at least $(1 - \epsilon/p(n))$. We will deal with this issue later on, and instead of a 5-approximation we will only get a $(5+\epsilon)$-approximation. This theorem follows directly from Lemma 7.3.4 and Lemma 7.3.5 which we will see in a second.

Theorem 7.3.3 *Let S^* be a global optimum and S be a local optimum. Then $C(S) \leq 5C(S^*)$.*

Remark. As mentioned at the beginning of this section, the bound on $C(S)$ in Theorem 7.3.3 can be improved to $3C(S^*)$. We will not go into details.

For the proofs of the two lemmas we need a few notations. We define, for any $j \in Y$,

- $\sigma^*(j)$ = facility that customer j is assigned to in S^*
- $\sigma(j)$ = facility that customer j is assigned to in S
- $r^*(j)$ = routing cost for customer j in S^*
- $r(j)$ = routing cost for customer j in S.

Lemma 7.3.4 *Let S^* be a global optimum and S be a local optimum. Then $C_r(S) \leq C(S^*)$.*

Proof: Let $i^* \in S^*$. Consider adding i^* to S. Since S is a local optimum, this operation does not decrease the total cost, and therefore

$$f_{i^*} + \sum_{j:\sigma^*(j)=i^*} (r^*(j) - r(j)) \geq 0.$$

Summing over all $i^* \in S^*$, we get

$$\begin{aligned} C_f(S^*) + C_r(S^*) - C_r(S) &\geq 0 \\ C(S^*) - C_r(S) &\geq 0. \end{aligned}$$

■

Lemma 7.3.5 *Let S^* be a global optimum and S be a local optimum. Then $C_f(S) \leq 2C(S^*) + 2C_r(S) \leq 4C(S^*)$.*

Proof: To bound $C_f(S)$, we consider swapping facility i in S for the nearest facility i^* in S^*. We reassign all customers of i to i^* in S^*. Since S is a local optimum, this operation does not increase the total cost, and hence we can bound the opening cost of i by the increase in routing cost plus the opening cost of i^*. However, the opening cost of i^* will be charged multiple times in our analysis if there is more than one facility in S whose nearest facility in S^* is i^*. To avoid this, we only charge the opening cost of i^* to the one nearest to i^* among those facilities. We call that facility *primary*, and the rest *secondary*. A formal argument is presented below.

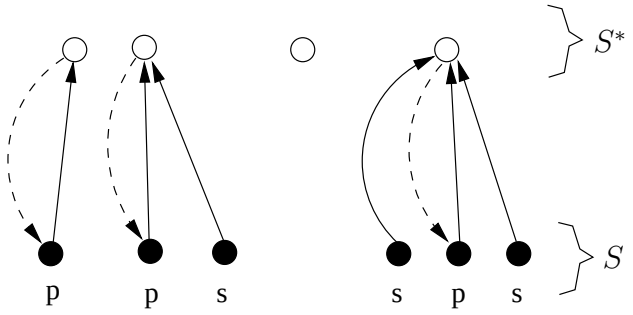


Figure 7.3.1: Primary and secondary facilities. Solid arrows represent the σ function; dotted arrows represent the π function; 'p' means primary and 's' means secondary.

For any $i \in S$, let $\sigma^*(i)$ be the facility in S^* which is nearest to i, i.e. $\sigma^*(i) = \arg\min_{i^* \in S^*} c(i, i^*)$. For any $i^* \in \{\sigma^*(i) \mid i \in S\} \subseteq S^*$, let $\pi(i^*)$ be the primary facility associated to i^*, i.e. $\pi(i^*) = \arg\min_{i \in S : \sigma^*(i) = i^*} c(i, i^*)$.

For any $i \in S$, let R_i be the total routing cost of all customers of i in S, i.e. $R_i = \sum_{j:\sigma(j)=i} r_j$; let R_i^* be the total routing cost of the same set of customers in S^*, i.e. $R_i^* = \sum_{j:\sigma(j)=i} r_j^*$. Note that R_i and R_i^* sum over the same set of customers — R_i^* does not sum over $\{j \mid \sigma^*(j) = i\}$.

Claim 7.3.6 *If $i \in S$ is primary, then $f_i \leq R_i + R_i^* + f_{i^*}$.*

Proof: Consider swapping i for i^*, then assigning all customers of i to i^*. Since S is a local optimum, such an operation would not decrease the total cost, and therefore

$$f_{i^*} - f_i + \sum_{j:\sigma(j)=i} (c(j, i^*) - c(j, i)) \geq 0. \tag{7.3.1}$$

We are going to show that the term $c(j, i^*) - c(j, i)$ in the summation is small. By the triangle inequality, we have $c(j, i^*) - c(j, i) \leq c(i, i^*)$. Now since i^* is the facility in S^* that is closest to i,

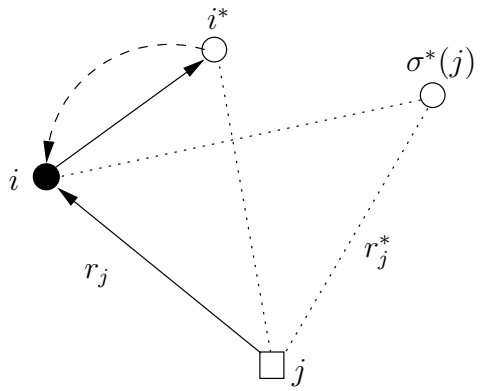


Figure 7.3.2: Entities in Claim 7.3.6.

and noting that $\sigma^*(j) \in S^*$, we get $c(i, i^*) \leq c(i, \sigma^*(j))$. By applying the triangle inequality again, $c(i, \sigma^*(j)) \leq c(j, i) + c(j, \sigma^*(j)) = r_j + r_j^*$. Combining the above, we get the following inequality:

$$c(j, i^*) - c(j, i) \leq r_j + r_j^*.$$

Now we can substitute it into Equation 7.3.1 to obtain the desired result:

$$\begin{aligned} 0 &\leq f_{i^*} - f_i + \sum_{j:\sigma(j)=i} (r_j + r_j^*) \\ &\leq f_{i^*} - f_i + R_i + R_i^*. \end{aligned}$$

■

For a secondary facility i in S, we remove it and assign its customers to a nearby facility in S. Specifically, we assign them to the primary facility associated to $\sigma^*(i)$.

Claim 7.3.7 *If $i \in S$ is secondary, then $f_i \leq 2(R_i + R_i^*)$.*

Proof: Let $i^* = \sigma^*(i)$ and $i' = \pi(i^*)$. Consider removing i from S and assigning its customers to i'. Let j be a customer assigned to i in S, and consider the increase in its routing cost:

$$\begin{aligned} c(i', j) - c(i, j) &\leq c(i, i^*) + c(i^*, i') \\ &\leq 2c(i, i^*) \\ &\leq 2c(i, \sigma^*(j)) \\ &\leq 2(r_j + r_j^*) \end{aligned}$$

The first and last inequalities follow from the triangle inequality; the second inequality follows from the fact that i' is the primary facility associated to i^*; the third inequality follows from the fact that i^* is the facility in S^* closest to i.

Summing over all customers of i, we have shown that the total increase in routing costs is no more than $2(R_i + R_i^*)$. Since the increase in the total cost is non-negative, we get

$$-f_i + 2(R_i + R_i^*) \geq 0.$$

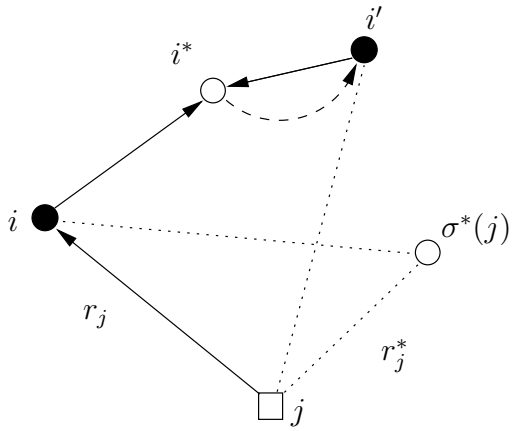


Figure 7.3.3: Entities in Claim 7.3.7.

■

From the above two claims we obtain

$$\begin{aligned} C_f(S) &\leq C_f(S^*) + 2\sum_{i \in S}(R_i^* + R_i) \\ &= C_f(S^*) + 2C_r(S^*) + 2C_r(S) \\ &\leq 2C(S^*) + 2C_r(S). \end{aligned}$$

Finally, by Lemma 7.3.4 we have

$$C_f(S) \leq 2C(S^*) + 2C_r(S) \leq 4C(S^*).$$

■

Finally, we will show that carrying out step 2 only when the cost decreases significantly increases the approximation factor by only a small amount. Our algorithm moves from one solution to another only if the total cost decreases by a factor of at least $(1 - \epsilon/p(n))$, so it may not halt at a local optimum. Nevertheless, we can still bound the cost of the solution by slightly modifying the above proofs. In Lemma 7.3.4 and Lemma 7.3.5, we can replace 0 with $(-\epsilon/p(n))C(S)$ when we set up inequalities for the increases in cost. There are less than n^2 inequalities. Summing them up we get $(1 - \epsilon n^2/p(n))C(S) \leq 5C(S^*)$. By choosing $p(n) = n^2$, we have shown that $C(S)$ is a $(5 + \epsilon)$-approximation for $C(S^*)$.

Recall that our analysis of the algorithm is not tight — the bound can be improved to $C(S) \leq 3C(S^*)$ by reassigning customers more carefully in Lemma 7.3.5. In the following, we present an example showing that this is in fact the best we can get from this local search algorithm.

Suppose that there are $k + 2$ facilities and $k + 1$ customers "connected" in the way shown in Figure 7.3.4. (The distance between any two entities is the shortest path distance between them. Note that shortest path distances on a graph always form a metric.) The opening cost of facility x_{k+2} is $2k$; other facilities have zero opening cost. The global optimum $S^* = \{i_1, \ldots, i_{k+1}\}$ has

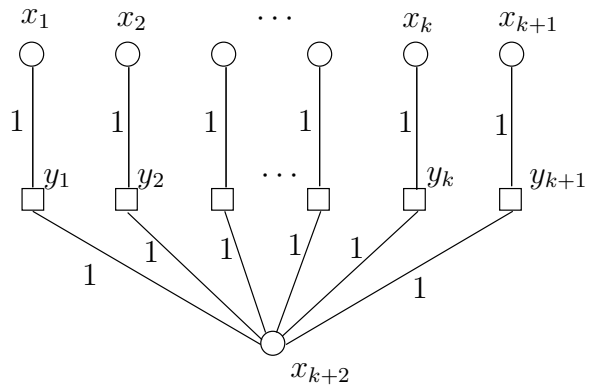


Figure 7.3.4: A worst case example for our local search algorithm.

total cost $k+1$. Let $S = \{x_{k+2}\}$. The total cost of S is $3k+1$. We claim that S is a local optimum: We cannot remove a facility from S since that would result in the empty set. Adding a facility does not help either. Swapping x_{k+2} for another facility, say x_m, does not decrease the total cost, since each customer except y_m is 3 units away from x_1. The total cost of S is roughly 3 times that the total cost of S^*, thus showing the tightness of the bound.

References

[1] M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. In *JACM*, 42, 1115–1145, 1995.

[2] S. Khot, G. Kindler, E. Mossel and R O'Donnell. Optimal inapproximability results for max-cut and other 2-Variable CSPs? In *45th Annual IEEE Symposium on Foundations of Computer Science*, pp. 146–154, 2004.

[3] M. Mahdian, Y. Ye and J. Zhang. Improved approximation algorithms for metric facility location problems. In *Proceedings of the 5th International Workshop on Approximation Algorithms for Combinatorial Optimization*, pp. 229–242, September 17–21, 2002.

[4] S. Guha and S. Khuller. Greedy Strikes Back: Improved Facility Location Algorithms. In *Journal of Algorithms*, Volume 31, Issue 1, pp. 228–248, 1999.

[5] M. R. Korupolu, C. G. Plaxton and R. Rajaraman. Analysis of a local search heuristic for facility location problems. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 1–10, 1998.

[6] V. Arya, N. Garg, R. Khandekar, K. Munagala and V. Pandit. Local search heuristics for k-median and facility location problems. In *Symposium on Theory of Computing*, pp. 21–29, 2001.

CS880: Approximations Algorithms

Scribe: Siddharth Barman **Lecturer:** Shuchi Chawla
Topic: Min Degree Spanning Tree **Date:** 02/15/07

In this lecture we give a local search based algorithm for the Min Degree Spanning Tree problem.

8.1 Min Degree Spanning Tree

Problem Statement: Given an unweighted graph G, find a spanning tree with least possible max degree.

This problem cab be shown to be $\mathcal{NP}$-Hard by reducing Hamiltonian path to it. Essentially existence of a spanning tree of max degree two is equivalent to a having a Hamiltonian path in the graph. This also shows that the best approximation we can hope for (unless $\mathcal{P} = \mathcal{NP}$) is $\Delta^* + 1$ where Δ^* is the max degree of the optimal tree. Note that we usually express approximations in a multiplicative form, but also occasionally in an additive form. In general, for additive approximations to be meaningful the optimal value should be bounded from below. As in this case Δ^* is an integer no less than two.

In this lecture we provide a *local search* approximation algorithm which achieves an approximation of $2\Delta^* + \log n$. The algorithm and its analysis is by Fürer and Raghavachari [1]. They also provide an improved approximation with max degree $\Delta^* + 1$ in the same paper.

The main idea behind local search is to consider a graph over the solution space; nodes of this meta graph represent possible solutions. We then define a set of low cost operations (edges in the meta graph) which morph one solution into another. The strategy is to start from some solution/node and traverse the meta graph, improving the objective function at each step. We stop when we hit a local optimum. The length of the path traversed dictates the time complexity of the algorithm. Also for a good approximation factor we must show that (compared with the global optima) any local minimum reached by the algorithm is not too bad.

For the min degree spanning tree problem the solution space consists of all spanning trees. The connecting operation is to swap edges till the degree goes down. Formally:

Definition 8.1.1 (T-swap:) *Add edge $e_1 \notin T$ and remove edge $e_2 \in T$ s.t. e_2 is on the unique cycle that is formed by adding e_1 to the tree.*

In order to improve the objective function value, we would like our local search algorithm to perform T-swaps as long as one of the following happens:

- the max degree of the current tree decreases or,
- the number of max degree nodes decrease

Unfortunately the *T-swap* operation with the above defined improvement criteria might get stuck with a high max degree tree (as shown in Figure 1). In particular in the figure, adding edge $(2, 3)$

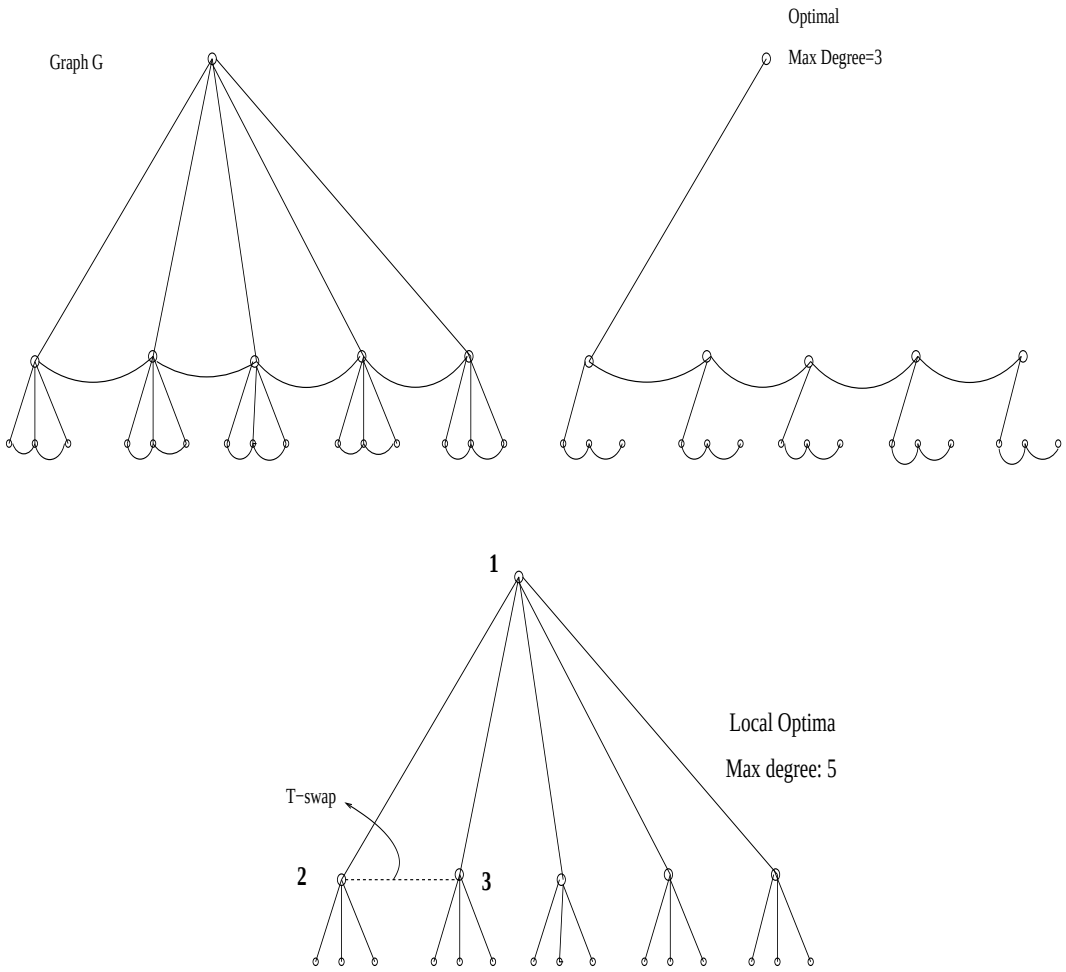


Figure 8.1.1: Allowed T-swaps

introduces cycle $1 \to 2 \to 3 \to 1$, but removing any edge edge from this cycle keeps the max degree at 5. So although the optimal tree has max degree only 3; the local optimal has max degree 5. Extending this example we can show that a local optimum can have max degree $\Omega(n)$ even when the optimal solution has max degree 3.

To overcome this problem, we allow T-swaps even when the above two conditions do not hold. In particular, consider edges $e_1 = (u_1, u_2)$ and $e_2 = (u_3, u_4)$ with $e_1 \notin T$, $e_2 \in T$ and e_2 lying on the unique cycle in $T \cup \{e_1\}$, say the T-swap of e_1 and e_2 transforms T into T'. Finally let δ indicate the max degree of the four end points of edges e_1 and e_2 in T, i.e. $\delta = \max_i\{deg_T(u_i)\}$ and similarly in T', $\delta' = \max_i\{deg_{T'}(u_i)\}$. We perform a T-swap if $\delta' < \delta$, i.e. we perform a T-swap if the max degree over the four end points of the involved edges goes down.

This operation does not necessarily decrease the objective function value, however we note that the degree sequence of the tree decreases lexicographically. In particular, say the string χ_T represents

the multiset of degrees of the vertices in tree T arranged in decreasing order. For example, with a max degree d, χ_T would look something like $d\ d\ d-1\ d-1$ .....221111. By performing a T-swap we replace the degree entry of δ by a lower value, δ', thus decrementing χ_T. The relevant observation here is that each step of the local search decreases χ_T lexicographically.

Let T be a locally optimal tree obtained by the above mentioned algorithm and T^* be the globally optimal tree. We denote by Δ the degree of T and by Δ^* the degree of T^*. Then we have the following,

Theorem 8.1.2 $\Delta \leq 2\Delta^* + \log n + 1$, *i.e. the degree of the local optimal found by the local search algorithm is bounded above by twice the degree of the global optimal with an additive factor of* $\log n + 1$.

We first give a generic lower bound on Δ^* in the following lemma. This along with the fact that T is a local optima will be used to establish a *witness* for Δ^*. We define $c(X)$ as the number of connected components in $G[V \setminus X]$.

Lemma 8.1.3 *For any $X \subseteq V$ we have*

$$\frac{|X| + c(X) - 1}{|X|} \leq \Delta^*$$

Proof:

We consider the average degree of X in any spanning tree of the given graph G. Any spanning tree must contain at least $|X| + c(X) - 1$ edges connecting the $c(X)$ components and the nodes in X to each other. Fewer edges would leave at least one of these disconnected from the other. Also components are not connected amongst themselves so each of these $|X| + c(X) - 1$ edges is incident to a vertex belonging to set X hence the average degree of X is no less than $\frac{|X|+c(X)-1}{|X|}$. This is true for any spanning tree hence $\Delta^* \geq \frac{|X|+c(X)-1}{|X|}$ ■

We now proceed to prove Theorem 8.1.2.

Proof of Theorem 8.1.2: Say the local optimal reached is T, we consider X_t the set of all nodes with degree no less than t in T. We denote the number of connected components in $T[V \setminus X_t]$ by $c_T(X_t)$. Note that the total degree of nodes in X_t is no less than $t|X_t|$.

We can bound this degree sum $t|X_t|$ by counting edges in T: Let us consider each of the $c_T(X_t)$ components as a single vertex and preserve connections of T between connected components and vertices in X_t i.e. edges between a connected component and a node of X_t are preserved, along with edges with both end points in X_t. The number of edges in this tree is $|X_t| + c_T(X_t) - 1$. The degree contribution of edges with both end points in X_t needs to be counted once more. But such edges are at most $|X_t| - 1$. Hence degree sum $t|X_t| \leq |X_t| + c_T(X_t) - 1 + |X_t| - 1$, which gives us

$$c_T(X_t) \quad \geq \quad t|X_t| - 2(|X_t| - 1) \tag{8.1.1}$$

We essentially use $c_T()$ to determine $c()$, T being a local optimum. The observation is that if A and B are two different connected components in $T[V \setminus X_t]$ and there is a graph edge between A and B then the end points of this edge is in X_{t-1} (see Figure 2). Otherwise this edge would have

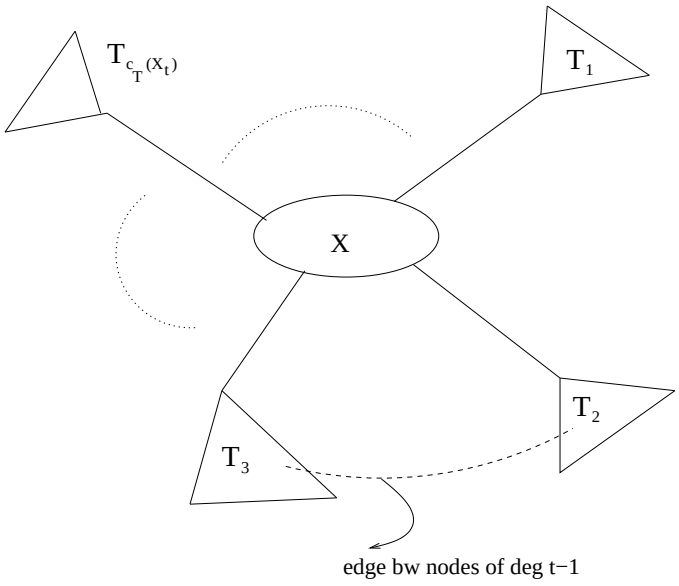


Figure 8.1.2: Connected components in tree

been T-swapped in to reduce the degree of one of the vertices currently in X_t. So if A and B are connected in the graph, it must be via an edge with end points in X_{t-1}. Hence, if we remove the X_{t-1} vertices all the $c_T(X_t)$ components definitely get disconnected in the original graph. This implies that $c(X_{t-1}) \geq c_T(X_t)$ for all t.

By lemma 8.1.3 we have

$$\Delta^* \geq \frac{c(X_{t-1}) + |X_{t-1}| - 1}{|X_{t-1}|}$$

Since $|X_{t-1}| \geq |X_t|$ and $c(X_{t-1}) \geq c_T(X_t)$ the above along with eqn. 8.1.1 reduces to

$$\begin{aligned} \Delta^* &\geq \frac{t|X_t| - 2(|X_t| - 1) + |X_t| - 1}{|X_{t-1}|} \\ &= \frac{(t-1)|X_t| + 1}{|X_{t-1}|} \\ &\geq \frac{(t-1)|X_t|}{|X_{t-1}|} \end{aligned}$$

We now pick a t that is large, and simultaneously for which $|X_t|$ is a large fraction of $|X_{t-1}|$. Note that there is always a t in $[\Delta - \log n, \Delta]$ s.t. $|X_t| \geq \frac{1}{2}|X_{t-1}|$. This holds because $\log n$ jumps of relative size more than $1/2$ will use up all the n vertices.

For this t, $\Delta^* \geq (\Delta - \log n - 1) \times \frac{1}{2}$ hence we have $\Delta \leq 2\Delta^* + \log n + 1$. ■

We now proceed to determine the running time of algorithm. We prove that the number of steps required to reach a local optimal is polynomial n.

Consider the degree sequence χ_T of nodes in the tree $\Delta\ \Delta\ \Delta - 1 \ldots 2\ 2\ 1\ 1$. As stated earlier at every step χ_T moves down lexicographically. But the number of different sequences and therefore, the number of possible steps is exponentially large. Instead we use a potential function argument to show fast convergence. We define the following potential function $\phi(T) = \sum_v 3^{deg(v)}$. The intuition behind this definition is that we assign exponential higher weight to high degree vertices. Also note that $\phi(T_0) \leq n3^{n-1}$.

Define $\Delta\phi$ as the change in potential function value obtained after a T-swap. Say currently we are at tree T_1 and T-swap edge (u_1, u_2) by introducing (v_1, v_2) to obtain new tree T_2, i.e. $T_2 = T_1 \setminus \{(u_1, u_2)\} \cup \{(v_1, v_2)\}$. Say

- $deg_{T_1}(u_1) = x_1$ and $deg_{T_1}(u_2) = x_2$
- $deg_{T_1}(v_1) = y_1$ and $deg_{T_1}(v_2) = y_2$

Assume w.l.o.g. that $x_1 \geq x_2$, since the T-swap was legal we have $y_1 + 1 \leq x_1 - 1$ and $y_2 + 1 \leq x_2$. This implies

$$\Delta\phi = \phi(T_1) - \phi(T_2) = \frac{2}{3}\ 3^{x_1} + \frac{2}{3}\ 3^{x_2} - \frac{4}{3}\ 3^{x_1 - 1}$$

We ignore the second term in the summand to obtain

$$\Delta\phi \geq \frac{2}{3}\ 3^{x_1} - \frac{4}{3}\ 3^{x_1 - 1} \geq \frac{2}{3}\ 3^{x_1 - 1}$$

We modify the algorithm to allow a T-swaps only if the max degree of the four end points of e_1 and e_2 is greater than $\Delta - \log n$ (this does not impact the analysis of Theorem 8.1.2). Hence $\Delta\phi \geq$ const $\times \frac{3^\Delta}{n}$.

As $\phi(T_1) \leq n3^\Delta$, we have $\Delta\phi \geq$ const $\times \frac{\phi(T_1)}{n^2}$. In other words in n^2 steps we shave off a constant factor. Therefore in $n^2 \times \log(Initial\ value) = O(n^2 \log(n3^{n-1})) = O(n^3)$ steps the algorithm would reach a local optimum. This implies that the running time of the algorithm is polynomial in n.

To summarize the local search algorithm is given as follows:

1. Start with an arbitrary spanning tree.
2. Say current max degree of tree is Δ
3. Consider edges $e_1 = (u_1, u_2)$ and $e_2 = (u_3, u_4)$ with $e_1 \notin T$, $e_2 \in T$ and e_2 lying on the unique cycle in $T \cup \{e_1\}$;
   Let $x_i = deg_T(u_i)$ for $i \in [4]$ and $\delta = \max_{i \in [4]}\{x_i\}$;
   Let $\delta' = \max\{x_1 + 1, x_2 + 1, x_3 - 1, x_4 - 1\}$;
4. If ($\delta > \delta'$ and $\delta \geq \Delta - \log n$) perform T-swap of edges e_1 and e_2.
5. Repeat steps 2 through 4 until no edges satisfy the conditions in step 4.

References

[1] M. Fürer, B. Raghvachari. Approximating the minimum degree spanning tree to within one from the optimal degree. *In: Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms(SODA)* (1992), pp: 317-324

CS880: Approximations Algorithms

Scribe: Matt Darnall **Lecturer:** Shuchi Chawla
Topic: Linear Programming **Date:** 2/22/07

9.1 Linear Programming

Linear programming is a method for solving a large number of maximization/minimization problems. Linear programming problems have the property that the constraints and the objective function are all linear functions of the input variables. The existence of a polynomial time algorithm for solving linear programs and the multitude of optimization problems that they can encode makes them particularly useful in practice.

To be precise, a linear programming problem (LP) is one that can be formulated as follows:

$$\text{Minimize} \quad \mathbf{c}^T\mathbf{x} \tag{9.1.1}$$

$$\text{Subject to} \quad A\mathbf{x} \leq \mathbf{b} \tag{9.1.2}$$

Here $\mathbf{x}$ is a vector of real-valued variables (sometimes assumed to be nonnegative), $\mathbf{c}$ and $\mathbf{b}$ are vectors of real constants, and A is a matrix of real constants. A useful geometric interpretation of this problem can be useful for understanding. We view each constraint $\sum_{i=1}^{n} a_{ij}x_i \leq b_j$ in $A\mathbf{x} \leq \mathbf{b}$ as a hyperplane in $\Re^n$, where the vector $\mathbf{x}$ has n entries. The constraint says that the solution vector $\mathbf{x}$ must lie below this hyperplane. The intersection of the constraints will be a polytope in $\Re^n$, with the points inside the polytope called **feasible** solutions. We then look at the planes $\mathbf{c}^T\mathbf{x} = k$ for real k. The solution to the linear program is the largest value of k such that the intersection of the constraint polytope and the hyperplane $\mathbf{c}^T\mathbf{x} = k$ is nonempty.

Using our geometric interpretation, it is easy to see that the solution of a linear program will occur at a vertex of the constraint polytope. These vertices are are called **basic** solutions. A possible method of solving a linear program is to enumerate the basic solutions and find the one with the largest value of the objective function. Unfortunately, there can be an exponential number of basic solutions. An example of this is the cube in $\Re^n$. Here, using only n variables and the $2n$ constraints $x_i \leq 1$ and $x_i \geq 0$ for all i, we can describe a constraint polytope with 2^n basic points.

9.2 Methods of Solving LPs

The first class of algorithms to solve LPs attempt to find the optimal solution by searching the boundary of the constraint polytope. These methods use "pivot rules" to determine the next direction of travel once a basic point is reached. If no direction of travel yields an improvement

to the objective function, then the basic point is the optimal solution. The first algorithm to use this idea was the Simplex Method from George Dantzig in 1947. Though these methods perform well in practice, it is not known if any "pivot rule" algorithm can run in polynomial time. In particular, an example has been given that takes exponential time using the Simplex Method [1].

The first polynomial time algorithm for solving an LP was derived from the Ellipsoid Method of Shor, Nemirovsky, and Yudin. This algorithm gives way of finding a feasible solution to an optimization problem. The idea is to enclose feasible solutions in an ellipse. Then, check to see center of the ellipse is a feasible solution. If not, find a violating constraint using a seperation oracle. Then, enclose the half of the ellipse where the feasible solutions must lie in another ellipse. Since this next ellipse is at least a fixed constant times smaller, by repeating we are able to hone in on a solution exponentially fast. Khachiyan was able to adapt this method of finding a feasible solution to give a polynomial time solution to LPs. Though this result was a breakthrough in the theory, the algorithm usually takes longer than the Simplex Method in practice.

The next class of algorithms for solving LPs are called "interior point" methods. As the name suggests, these algorithms start by finding an interior point of the constraint polytope and then proceeds to the optimal solution by moving inside the polytope. The first interior point method was given by Karmarkar in 1984. His method is not only polynomial time like the Ellipsoid Method, but it also gave good running times in practice like the Simplex Method.

9.3 Integer Linear Programming

To recall from last time, a linear programming problem is given by

$$\text{Minimize} \qquad \mathbf{c}^T\mathbf{x} \tag{9.3.3}$$

$$\text{Subject to} \qquad A\mathbf{x} \leq \mathbf{b} \tag{9.3.4}$$

where $\mathbf{x}$ is a vector of real-valued variables (sometimes assumed to be nonnegative), $\mathbf{c}$ and $\mathbf{b}$ are vectors of real constants, and A is a matrix of real constants. We saw that there exists a polynomial time algorithm for solving an LP. If we add the additional condition that the variables in $\mathbf{x}$ be integers, we get what is called an integer linear programming problem, or IP. Unfortunately, there is no polynomial time algorithm for solving an IP. In fact, the existence of one would imply that P = NP!

We can relax the conditions of an IP make it an LP. It is obvious that the optimal solution to the LP, $\mathbf{x_L}$ is a lower bound on the optimal solution to the IP, $\mathbf{x_I}$. To approximate an IP, we can solve the "relaxed" LP and then find an integer solution close to the optimal solution of the LP. If we do it correctly, the approximate solution for the IP will be close to the solution for the LP, and we will have a good approximation algorithm. The ratio the LP solution and IP solution is called the integrality gap. We will attack many of the previous problems we looked at using this

technique.

9.4 Vertex Cover

The first problem we will approximate using this technique is vertex cover. In vertex cover we are given a graph $G = (V, E)$ and a cost function $c : V \to \Re^+$. We want to the subset $S \subset V$ of least cost such that every edge is attached to at least one of the vertices in S. To start, we must phrase the problem as an IP. For each vertex $v \in V$, we shall have a variable x_v in $\mathbf{x}$ corresponding to v. x_v shall be 1 if $v \in S$ and 0 otherwise. Our objective function to minimize is then

$$Ob(\mathbf{x}) = \sum_{v \in V} c(v) x_v$$

Since every edge must be incident to a vertex in S, our constraints shall be that, for every edge (u, v),

$$x_v + x_u \geq 1$$

Now, we relax the conditions to say that x_v can be a positive real number less than or equal to 1 for every v. This gives us an LP that we can solve in polynomial time to get an optimal solution $\mathbf{x}_L$. We shall now find a solution to the IP, $\mathbf{x}_I$, such that $Ob(\mathbf{x}_I)$ is close to $Ob(\mathbf{x}_L)$. To get $\mathbf{x}_I$, we shall round all the entries in $\mathbf{x}_L$ to the nearest integer. First, we notice that each entry at most doubles in value, so we have $Ob(\mathbf{x}_I) \leq 2Ob(\mathbf{x}_L)$. The rounded solution still satisfies the constraints because, for $x_u + x_v \geq 1$ in $\mathbf{x}_L$ we must have that at least one of x_u and x_v is greater than $1/2$. Thus, this variable will be rounded to 1 in $\mathbf{x}_I$ and the condition shall still be satisfied. This technique gives us a 2-approximation to vertex cover.

It turns out that the basic solutions of the Vertex Cover LP have the property that every variable x_v is either 0, $1/2$ or 1. We prove this by showing that for every point in the feasible polytope that doesn't have this property, there exists a line through this point that contains feasible solutions on both sides of the point. Since every line through a basic point has feasible solutions on at most one side of the line, we know that the basic solutions have $x_v = 0$, $1/2$, or 1. To show the line property, we fix an $\epsilon > 0$. Now, in a feasible solution $\mathbf{x}$, we consider the set, S, of $x_v \in (\epsilon, 1/2 - \epsilon)$ and the set T of $x_v \in (1/2 + \epsilon, 1 - \epsilon)$. If we lower the x_v in S by ϵ and raise the x_v in T by ϵ, we still have a feasible solution. This is because for each edge, if we lowered one of the x_v coresponding to it, we either raised the other by the same amount, or the other was already large enough to satisify the edge's requirement. Likewise, we can lower the x_v in T by ϵ and raise the x_v in S by ϵ and we shall still have a feasible solution. The only way that this doesn't give three colinear points with our original point in the middle is if S and T are empty. By taking ϵ small enough, this only happens if x_v is 0, $1/2$, or 1.

9.5 Set Cover

We shall obtain an f approximation to set cover using algorithm similar to the one for vertex cover. Let X be a finite set and let C be collection of subsets of X. Let $c : C \to \Re^+$ be a cost function for the elements of C. Our goal is to find a collection of subsets in C of minimal cost such that every element of X is in at least one chosen subset. Here f is the maximum number of subsets in C that an element of X belongs to. As with vertex cover, for any subset $S \in C$ we define an variable x_S that is 1 if S is chosen in our covering collection and 0 otherwise. Our objective function is

$$\sum_{S \in C} c(s) x_S$$

For an element $e \in X$, let C_e be the sets of C that contain e. As constraints, we have that for every e,

$$\sum_{S \in C_e} x_S \geq 1$$

As before, we relax the conditions on $\mathbf{x}$ to allow real values between 0 and 1. Now, we get an optimal solution $\mathbf{x}_L$ to this LP. To get $\mathbf{x}_I$, we round an entry in $\mathbf{x}_L$ up to 1 if it is greater than or equal to $1/f$ and set it to 0 otherwise. For every $e \in X$ at least one of the x_S in $\mathbf{x}_L$ for $S \in C_e$ must be larger than $1/f$. This means that in $\mathbf{x}_I$ at least one of the sets chosen contains e for all $e \in S$. Thus, $\mathbf{x}_I$ is a feasible solution to the IP. It is an f-approximation since each entry in $\mathbf{x}_L$ was raised by at most a factor of f.

9.6 Max Flow

As a final example for this lecture, we shall look at the max flow problem on a graph. We are given a directed graph $G = (V, E)$, a capacity function $c : E \to \Re^+$, as well as a pair of vertices (s, t). Our goal is to route the maximum amount from s to t subject to the constraint that any edge, E, must have at most $c(E)$ routed through it. To phrase this problem as a linear program, we make a variable x_e for every edge in the graph. This variable represents the amount routed on this edge. Our goal is to maximize:

$$\sum_{e \to t} x_e$$

For a vertex, v, and an edge, e; $e \to v$ denotes that e goes to v and $e \leftarrow v$ means e goes from v. As constraints, we need that the amount flowing out of a particular vertex is the same as the amount flowing in. So for every vertex, v, that is not s or t, we must have

$$\sum_{e \to v} x_e = \sum_{e \leftarrow v} x_e$$

We also must have that the amount flowing out of s is equal to the amount flowing into t, so

$$\sum_{e \to t} x_e = \sum_{e \leftarrow s} x_e$$

Finally, each edge must not exceed its capacity, so the final constraints are that for each edge e,

$$x_e \leq c(e)$$

If the capacity function only takes integer values, it turns out that the optimal solution will have an integer amount of flow on each edge. We did not go over the proof of this statement, though it has to do with the fact that the incidence matrix of a directed graph is totally unimodular (all determinants of submatrices are 0, 1, or -1). Thus the solution to the IP where we require all the flow on an edge to be an integer can be solved in polynomial time, since it is the same as the solution of the LP where all the capacities have been rounded down to the nearest integer.

References

[1] *Klee-Minty Polytope Shows Exponential Time Complexity of the Simplex Method*. University of Colorado at Denver, 1997.

CS880: Approximations Algorithms

Scribe: Matt Elder **Lecturer:** Shuchi Chawla
Topic: LP Rounding and Randomized Rounding **Date:** 2/22/2007

In out last lecture, we discussed the LP Rounding technique for producing approximation algorithms. The idea behind LP Rounding is to write the problem as an integer linear program, relax its integrality restraints to efficiently solve the general linear program, and then move the LP solution to a nearby integral point in the feasible solution space. The difficulty of this process lies in the rounding step, which demands that a bound on its suboptimality.

We discussed how to apply this method to vertex cover, set cover, and network flow. Here, we give somewhat more complicated rounding methods for facility location, and introduce the technique of randomized rounding in application to set cover and min-congestion rounding.

11.1 Facility Location

Again, the facility location problem gives a collection of facilities and a collection of customers, and asks which facilities we should open to minimize the total cost. We accept a facility cost of f_i if we decide to open facility i, and we accept a routing cost of $c(i,j)$ if we decide to route customer j to facility i. Furthermore, we know that the routing costs form a metric.

First, we design a linear program to answer a "relaxed" version of this problem. We let the variable x_i denote the extent to which facility i is open, and let y_{ij} denote the extent to which customer j is assigned to facility i. The following linear program then expresses the problem:

$$\begin{aligned} \text{minimize} \quad & \sum_i f_i x_i + \sum_{i,j} c(i,j) y_{ij}, \\ \text{where} \quad & 0 \le y_{ij} \le x_i \le 1 \qquad \forall i, j \end{aligned}$$

For convenience, let $C_f(x)$ denote the total factory cost induced by x, i.e., $\sum_i f_i x_i$. Similarly, let $C_r(y)$ denote the total routing cost induced by y, $\sum_{i,j} c(i,j) y_{ij}$.

If this LP were modified to require that these variables each equal 0 or 1, this system would be precisely the ILP we need to solve. But, again, solving general ILPs is NP-hard problem, so we solve this related real-valued LP instead.

Let x^*, y^* be the optimal solution to this linear program. Since every feasible solution to the original ILP lies in the feasible region of this LP, the cost $C(x^*, y^*)$ is less than the optimal solution to the ILP. Since x^* and y^* are almost certainly non-integral, we need a way to round this solution to a feasible, integral solution without increasing the cost function much.

To do so, we first employ the filtering technique of Lin and Vitter [1] to produce $\tilde{x}, \tilde{y}$. This filtering will later allow us to put upper bounds on the routing cost that we accept.

1. For each customer j, compute the average cost $\tilde{c}_j = \sum_i c(i,j) y^*_{ij}$.

2. For each customer j, let the S_j denote the set $\{i \mid c(i,j) \leq 2\tilde{c}_j\}$.

3. For all i and j: if $i \notin S_j$, then set $\tilde{y}_{ij} = 0$; else, set $\tilde{y}_{ij} = y^*_{ij} / \sum_{i \in S_j} y^*_{ij}$.

4. For each facility i, let $\tilde{x}_i = \min(2x^*_i, 1)$.

Lemma 11.1.1 *For all i and j, $\tilde{y}_{ij} \leq 2y^*_{ij}$.*

Proof: If we fix j and treat y^*_{ij} as a probability distribution, then we can show this by Markov's inequality. However, the proof of Markov's Inequality is simple enough to show precisely how it applies here:

$$\tilde{c}_j = \sum_i c(i,j) y^*_{ij} \geq \sum_{i \notin S_j} c(i,j) y^*_{ij} \geq \sum_{i \notin S_j} 2\tilde{c}_j y^*_{ij} \geq 2\tilde{c}_j \sum_{i \notin S_j} y^*_{ij}.$$

So, $1/2 \geq \sum_{i \notin S_j} y^*_{ij}$. For any fixed j, y^*_{ij} is a probability distribution, so $\sum_{i \in S_j} y^*_{ij} \geq 1/2$. Therefore, $\tilde{y}_{ij} = y^*_{ij} / \left(\sum_{i \in S_j} y^*_{ij}\right) \leq 2y^*_{ij}$. ■

Lemma 11.1.2 *$\tilde{x}, \tilde{y}$ is feasible, and $C(\tilde{x}, \tilde{y}) \leq 2C(x^*, y^*)$.*

Proof: For any fixed j, the elements $\tilde{y}_{ij}$ form a probability distribution. For every i and j, $\tilde{y}_{ij} \leq 2y^*_{ij}$ and thus $\tilde{x}_i \geq \sum_i \tilde{y}_{ij}$. It is clear that $0 \leq x_i, y_{ij} \leq 1$ for all i and j, so $\tilde{x}$ and $\tilde{y}$ are feasible solutions to the LP. ■

Now, given $\tilde{x}$ and $\tilde{y}$, we perform the following algorithm:

1. Pick the unassigned j that minimizes $\tilde{c}_j$.

2. Open factory i, where $i = \text{argmin}_{i \in S_j}(f_i)$.

3. Assign customer j to factory i.

4. For all j' such that $S_j \cap S_{j'} \neq \emptyset$, assign customer j' to factory i.

5. Repeat steps 1-4 until all customers have been assigned to a factory.

Let L be the set of facilities that we open in this way. We now show that the solution that this algorithm picks has reasonably limited cost.

Lemma 11.1.3 *$C_f(L) \leq C_f(x^*)$ and $C_r(L) \leq 6C_r(y^*)$.*

Proof: For any two customers j_1 and j_2 that were picked in Step 1, $S_{j_1} \cap S_{j_2} = \emptyset$.

Consider the facility cost incurred by one execution of Steps 1 through 4. Let j be the customer chosen in Step 1, and let i be the facility chosen in Step 2. Since $\tilde{x}$ is part of a feasible solution, $1 \leq \sum_{k \in S_j} \tilde{x}_k$. So, $f_i \leq f_i \sum_{k \in S_j} \tilde{x}_k$; and since f_i is chosen to be minimal, $f_i \leq \sum_{k \in S_j} f_k \tilde{x}_k$. Facility i is the only member of S_j that the algorithm can open.

Let J be the set of all customers selected in Step 1. Considering the above across the algorithm's whole execution yields

$$C_f(L) \leq \sum_{j \in J} \sum_{k \in S_j} f_k \tilde{x}_k = \sum_i f_i \tilde{x}_i = C_f(\tilde{x}) \leq C_f(x^*).$$

Consider now the routing cost C_r. If j was picked in Step 1, then its routing cost is $c(i,j)$ for some facility i; so $C_r(j) \leq 2\tilde{c}_j$.

Now, suppose instead that j' was not picked in Step 1. By the algorithm, there is some j that picked in Step 1 such that $S_j \cap S_{j'} \neq \emptyset$. Suppose that facility i' is in this intersection, and say that facility i is the facility to which customers j and j' are routed. Now, at long last, we use the fact that $c(i,j)$ forms a metric: we know that $C_r(j') \leq c(i',j') + c(i',j) + c(i,j)$. Because i is in both S_j and $S_{j'}$, we know by their definition that $c(i',j') \leq 2\tilde{c}_{j'}$ and that $c(i',j) + c(i,j) \leq 4\tilde{c}_j$. The customer j' was not picked in Step 1, and customer j was, so $\tilde{c}_j \leq \tilde{c}_{j'}$, and thus, $C_r(j') \leq 6\tilde{c}_{j'}$.

Now, $\tilde{c}_j$ was the routing cost of customer j in the y^* LP solution. So, $C_r(L) \leq 6C_r(y^*)$. ■

This lemma yields the following as a corollary:

Theorem 11.1.4 *This algorithm is a 6-approximation to Facility Location.*

Notice that, in the preceding construction, we picked S_j to be all i such that the cost $c(i,j) \leq 2\tilde{c}_j$. This 2 is actually fairly arbitrary. Suppose we replace it with α, some parameter of the construction. If you redo the above arithemetic, you find that $C_f(L) \leq (1/(1-\alpha))C_f(x^*)$ and that $C_r(L) \leq (3/\alpha)C_r(y^*)$. Thus, if we let $\alpha = 3/4$ instead of $1/2$, this method yields a 4-approximation. If we let α be a variable in the actual computed values of $C_f(x^*)$ and $C_r(y^*)$, we would get a somewhat better approximation.

11.2 Set Cover

Again, the set cover problem is: given a set of elements E, a collection of sets $\mathcal{S} \in \mathcal{P}(E)$, and a cost for each set c: $\mathcal{S} \to \mathbb{R}$, find a collection of sets $C \subseteq \mathcal{S}$ such that every element in E is contained in a set in C, and the total cost of C is minimized.

As an integer linear program, we can state this problem as follows:

$$\begin{aligned} \text{Minimize} \quad & \sum_{S \in C} c(S) x_S, \\ \text{where} \quad & \sum_{\substack{S \in C \\ S \ni e}} x_S \geq 1 \ \ \forall e, \\ \text{and} \quad & x_S \in \{0,1\} \text{ is 1 iff } S \in C \end{aligned}$$

To find an approximate solution to this ILP, we relax the condition $x_S \in \{0,1\}$ to $x_S \in [0,1]$, getting an LP. Then, we perform the following:

1. Solve the LP to get x^*.
2. For each set S, pick S with probability x^*_S.
3. Repeat Step 2 until all elements are covered.

Lemma 11.2.1 *The expected cost of Step 2 is the cost of x^*.*

Proof: Let Z_S be an indicator variable, which is 1 iff we pick set S in this run of Step 2. We compute:

$$\mathbf{E}[\text{cost of Step 2}] = \mathbf{E}\left[\sum_S Z_S c(S)\right] = \sum_S \mathbf{E}[Z_S] = c(x^*).$$

■

We now need to estimate the number of times that Step 2 is executed. To do so, we estimate the probability that any one element is covered in a particular execution of Step 2. Fix some element a. We know that $\sum_{S \ni a} x^*_S \geq 1$. This gives us the following reasoning:

$$\mathbf{Pr}[a \text{ is picked}] = 1 - \prod_{S \ni a}(1 - x^*_S) \geq 1 - \prod_{S \ni a} \exp\left(-x^*_S\right) = 1 - \exp\left(-\sum_{S \ni a} x^*_S\right) \geq 1 - e^{-1}.$$

So, the probability that e is unpicked after k steps is no more than e^{-k}, because each execution of Step 2 is independent. So, the probability that any particular element is unpicked after, say, $2 \ln n$ steps is no more than $(1/n^2)$. By the union bound, the probability that there exists an unpicked element after $2 \ln n$ steps is at most $n(1/n^2) = 1/n$.

Thus, with high probability, the number of executions of Step 2 is $O(\log n)$. So the expected total cost of the algorithm is $c(x^*)O(\log n)$, and this algorithms is a $O(\log n)$-approximation in expectation. Standard methods can convert this to an arbitrarily high-probability result.

References

[1] JH Lin and JS Vitter. Approximation Algorithms for Geometric Median Problems. In *Information Processing Letters*, 1992.

CS880: Approximations Algorithms	
Scribe: Dave Andrzejewski	**Lecturer:** Shuchi Chawla
Topic: Randomized routing, LP-Duality	**Date:** 3/1/07

The first part of the lecture shows how a randomized rounding scheme can be used to transform an optimal LP solution to a valid solution of the original problem. The specific problem used for this example is the min-congestion routing problem.

The second part of the lecture introduces the concept of LP-Duality, and the primal-dual interpretation of the maxflow-mincut problem.

12.1 Randomized rounding

12.1.1 Min congestion routing problem

GIVEN: a graph $G = (V, E)$ and k pairs of demand vertices (s_i, t_i).
DO: find a single path p_i from s_i to t_i for every i, trying to minimize the congestion C. The congestion is defined as the number of paths going through the most-used edge in the graph.

$$C = \max_{e \in E} |\{i | e \in p_i\}|$$

We then cast the problem as an LP where x_{ie} is defined as the amount of path i flow being sent through edge e.

$$\begin{aligned}
\min t & & \text{obj fcn} \\
\sum_{e \in \delta^+(v)} x_{ie} = \sum_{e \in \delta^-(v)} x_{ie} & & \forall i, \forall v \neq s_i, t_i \\
\sum_{e \in \delta^-(s_i)} x_{ie} = \sum_{e \in \delta^+(t_i)} x_{ie} = 1 & & \forall i \\
\sum_i x_{ie} \leq t & & \forall e \\
x_{ie} \geq 0 & & \forall i, \forall e
\end{aligned}$$

The sets $\delta^+(v)$ and $\delta^-(v)$ correspond to the incoming and outgoing flow, respectively, for vertex v. The summation constraints then enforce flow conservation, and source/sink assignments.

Since the objective function is to minimize t, which is constrained to be an upper bound for the flow across any edge, t will give us the (possibly fractional) congestion for a solution point of this LP.

In order to recover an integral/unsplittable flow solution from the LP solution, we will consider an equivalent formulation of the LP.

In this alternative formulation x_p refers to the flow along path p, and P_i is the set of all possible paths from s_i to t_i.

$$\begin{aligned} \min t & & \text{obj fcn} \\ \sum_{p \in P_i} x_p = 1 & & \forall i \\ \sum_{\{p | e \in p\}} x_p \leq t & & \forall e \\ x_p \geq 0 & & \forall p \end{aligned}$$

These formulations are equivalent in the sense that any solution to one can be converted into a solution of the other.

To convert from the first to the second, find all non-zero x_{ie} for a given i. Then, considering only these edges, perform DFS from s_i to find a path to t_i. For this path p, set x_p to the minimum x_{ie} value on the path, then subtract that amount from all x_{ie} on the path. That will effectively remove the minimum edge from our edge set. Repeat this procedure until there are no edges remaining, building a set of x_p values as you go.

To convert from the second LP to the first, find all non-zero x_p for a given P_i, then simply add that much flow to the edge flow x_{ie} for each $e \in p$.

It is important to note that the second formulation is impractical to solve directly, as the number of possible paths in the P_i sets will lead to exponentially many constraints. However the x_p quantities will come in useful for our randomized rounding scheme, as we will see.

12.1.2 Randomized rounding transformation

As on previous problems, the optimal objective function value of our LP forms a lower bound on the true optimal solution of the original problem. That is, $LP^* \leq OPT$. However, we need a way to do rounding from the flows in our LP solution to legal unsplittable flows for the original problem.

Our approach will be to solve the LP in the first formulation, convert the solution to the second formulation, and then treat the x_p values as path selection probabilities. For a given path set P_i, the constraints that all x_p must be ≥ 0 and sum to 1 ensure that this is a valid probability distribution. The algorithm is then relatively simple.

1. solve original formulation LP,
2. convert solution LP^* to second formulation
3. for each i, pick a $p \in P_i$ with probability x_p

Obviously, this algorithm will select exactly one path for each (s_i, t_i) pair, yielding a valid solution. What will the congestion be?

For every edge, total traffic in LP^* is $\leq t$. The traffic corresponds to the x_p values, which also then correspond to the path selection probabilities.

For each edge define a set of indicator random variables X_i.

$$X_i = \begin{cases} 1 & \text{if algo picks edge } e \text{ for commodity } i \\ 0 & \text{else} \end{cases}$$

Then define the expectation of X_i.

$$E(X_i) = \mu_i = \sum_{\{p \in P_i | e \in p\}} x_p$$

Then for any edge e and set of random variable values $X_i = x_i$, define its congestion to be $C_e = \sum_i x_i$. The expectation of the edge congestion can then be computed using our μ_i values and the linearity of expectation.

$$E(C_e) = E(\sum_i X_i) = \sum_i E(X_i) = \sum_i \mu_i \leq t$$

For our approximation factor, we want the sum of all X_i for every edge to be small. As shown above, we know the expectation for each edge e to be $\leq t$. More specifically, we want to show that for an appropriate value of λ, $Pr[C_e \geq \lambda E(C_e)]$ is small, for all e.

This can be accomplished through the use of a concentration bound result, specifically Chernoff's bound [1]. This bound assumes that the individual X_i are independent variables which can take on the values $\{0, 1\}$, and then uses Markov's inequality applied to a certain function to get the bound. For $\lambda \in [0, 1]$, the bound is:

$$Pr[X \notin (1 \pm \lambda)E(X)] \leq \exp \frac{-\lambda^2 E(X)}{3}$$

We customize the bound to our specific situation, using the fact that $\sum_i \mu_i \leq t$ to get the following inequality.

$$Pr[C_e \geq (1 + \lambda)t] \leq \exp \frac{-\lambda^2 \sum_i \mu_i}{3}$$

Note that $\sum_i \mu_i$ may be quite small, meaning that no value of $\lambda \leq 1$ will give a small probability of error. So in order to obtain a smaller bound we must use a more general formulation of Chernoff's bound that also holds for $\lambda > 1$.

$$P(C_e \geq (1 + \lambda)t) \leq \left(\frac{\exp \lambda}{(1 + \lambda)^{1+\lambda}} \right)^{\sum_i \mu_i}$$

We then manipulate the right-hand side.

$$\begin{aligned}\left(\frac{e^\lambda}{(1+\lambda)^{1+\lambda}}\right)^{\sum_i \mu_i} &\leq \left(\frac{e^\lambda}{\lambda^\lambda}\right)^{\sum_i \mu_i} \\ &\leq \left(\frac{1}{(\lambda/e)^\lambda}\right)^{\sum_i \mu_i}\end{aligned}$$

We now pick a value of λ for which the term λ^λ in the denominator becomes $n^O(1)$. Specifically we set $\lambda = O(\frac{\log n}{\log\log n})$. We then substitute in our definition of λ.

$$\begin{aligned}\lambda^\lambda &= \exp(\lambda \log \lambda) \\ &= \exp\left(\frac{\log n}{\log\log n}(\log\log n - \log\log\log n\right) \\ &= \Theta(n^c)\end{aligned}$$

where c is a factor determined by the actual terms in the $O()$ equation used to calculate λ. Setting λ appropriately to get $c = 3$, we plug this back into the original bound and end up with the result that for any edge e:

$$Pr[C_e > (1 + \left(\frac{\log n}{\log\log n}\right))t] < 1/n^3$$

We take the union of this bound over all $\leq n^2$ edges to get a total probability bound.

$$P(\exists e | C_e > (1 + \left(\frac{\log n}{\log\log n}\right))t) < 1/n$$

By repeatedly applying our algorithm, we can then achieve an arbitrarily low probability of exceeding an $(1 + \frac{\log n}{\log\log n})$-approximation to OPT.

12.2 LP-Duality

12.2.1 Definition/Derivation of LP-Duality

Consider the following example linear program. (For more discussion of this example, see [1].)

$$\begin{aligned}\min\, & x + 4y \\ & x + 2y \geq 5 \\ & 2x + y \geq 4 \\ & x, y \geq 0\end{aligned}$$

How could we obtain a lower bound on the true optimal objective function value for this LP (without actually solving it, that is)?

We can take a non-negative linear combination of the constraint equations. Since $x, y \geq 0$, if the x coefficient in our combination is $\leq$ the x coefficient in the objective function, and the same holds true for the y coefficients, our linear combination of constraints must also be $\leq$ the objective function for any legal x, y.

When taking linear combinations of the constraints, we will also end up a linear combination of the right-hand side of the constraints. Since it is a lower bound on the left-hand side, the linear combination of the right-hand sides of the constraints is also a lower bound on the objective function.

To see what we mean, call the linear combination coefficients u, v.

$$\begin{aligned} \min\, & x + 4y \\ & (x + 2y)u \geq 5u \\ & (2x + y)v \geq 4v \\ & x, y, u, v \geq 0 \end{aligned}$$

Enforcing that the x, y coefficients in the linear combination are $\leq$ than the x, y coefficients in the original objective function gives us some constraints on u, v. Since we want the tightest lower bound possible, we then want to maximize the right-hand size of the constraint linear combination. This mix of constraints and objective function give us a new LP.

$$\begin{aligned} \max\, & 5u + 4v \\ & u + 2v \leq 1 \\ & 2u + v \leq 4 \\ & u, v \geq 0 \end{aligned}$$

This LP is known as the dual of the original LP, which is called the primal. It is interesting to note that the objective function coefficients of the primal have become the constraint bounds in the dual, while the constraint bounds of the primal have become the objective function coefficients of the dual. Also, the u, v constraint coefficient matrix is the transpose of the x, y constraint coefficient matrix.

The relationship between the primal and dual LPs is very special and useful. The specifics will be spelled out in series of lemmas. For notation, $Val_P(x, y)$ and $Val_D(u, v)$ are the objective function values of the primal and dual LPs, respectively, with an $*$ denoting the optimal objective function value.

Theorem 12.2.1 *Let (x, y) be any feasible primal solution. Let (u, v) be any feasible dual solution. Then $Val_P(x, y) \geq Val_D(u, v)$.*

This result follows from manipulation of the constraints in the definition of the dual presented above, and is known as the Weak LP-Duality Theorem.

Theorem 12.2.2 *Val_P^* is finite iff Val_D^* is finite.*

Theorem 12.2.3 *If both the primal and the dual have feasible solutions, then $Val_P^* = Val_D^*$.*

These results are known as the Strong LP-Duality Theorem.

What if there are no feasible solutions for one of the versions of the problem, or if one of the problems is unbounded? It turns out that the dual has no feasible solutions iff the primal is unbounded below. Likewise, the primal has no feasible solution iff the dual is unbounded above.

Finally, it is interesting to note what happens when we take the dual of the dual.

Lemma 12.2.4 *The dual of the dual is the original primal.*

12.2.2 Applications of LP-Duality

12.2.2.1 General motivation

Why do we care about LP-Duality? For one thing, some of the optimization algorithms for actually finding LP solutions rely heavily on LP-Duality and its consequences.

Aside from that, it may be that the primal formulation is unwieldy, or that the rounding transformation for the dual solution may be more favorable. The dual formulation also may yield useful combinatorial insight into problem structure.

Finally, we can use the structure of the primal and dual to guide a purely combinatorial approximation algorithm for the underlying optimization problem. This technique is known as the primal-dual method and will be sketched out more fully in the next lecture.

12.2.2.2 Mincut-Maxflow

Consider the following LP formulation of the standard maxflow problem. Let x_p be the flow on path p, which is a path from source s to sink t.

$$\begin{aligned} \max \sum_p x_p & & \text{obj fcn} \\ \sum_{\{p|e\in p\}} x_p \le c_e & & \forall e \\ x_p \ge 0 & & \forall p \end{aligned}$$

Then take the dual.

$$\begin{aligned} \min \sum_{e\in E} y_e c_e & & \text{obj fcn} \\ \sum_{e\in p} y_e \ge 1 & & \forall p \\ y_e \ge 0 & & \forall p \end{aligned}$$

Note that since all $c_e \geq 0$ and we are trying to minimize the sum over e, in a solution no y_e will be assigned a value greater than 1.

What is the intuitive interpretation of the dual of our maxflow LP? Each y_e corresponds to 'choosing' an edge. The constraints state that we must choose ≥ 1 edge from every path, and the objective function tells us to minimize the sum of $y_e c_e$ over all e.

Ignoring fractional y_e, this means that our optimal LP solution to this dual needs to choose edges so that every path contains one of the chosen edges, and also choose the edges with the smallest total capacity. The smallest capacity set of edges which interesects every path from s to t is, by definition, the minimum weight separating cut between s and t.

As it turns out, all basic points of the primal and dual LPs are integral. This, along with the Strong LP Duality theorem (Theorem 12.2.3), implies the Max-Flow Min-Cut Theorem. The following lemma and theorem formalize this notion.

Lemma 12.2.5 *The Mincut LP has integral basic points.*

Corollary 12.2.6 *Maxflow-Mincut Theorem: The maximum possible flow between s and t is equal to the capacity of the minimum cut separating s and t.*

In this way LP-Duality allows us to clearly see the duality of the maxflow and mincut problems.

References

[1] V. Vazirani. Approximation Algorithms. Springer, 2001.

CS880: Approximations Algorithms

Scribe: Tom Watson **Lecturer:** Shuchi Chawla
Topic: Primal-Dual Method: Vertex Cover and Steiner Forest **Date:** 3/6/2007

In the previous few lectures we have seen examples of LP-rounding, a method for obtaining approximation algorithms that involves solving a linear programming relaxation of the problem at hand and rounding the solution. In the last lecture we also discussed the basic theory of LP-duality. Today we will apply this theory to obtain a second LP-based technique for obtaining approximation algorithms — the *primal-dual method*. This technique has the advantage that it circumvents the need to actually solve an LP relaxation, leading to efficient algorithms that are purely combinatorial. We will apply this technique to the Vertex Cover and Steiner Forest problems. The primal-dual method in the context of approximation algorithms was first used by Goemans and Williamson [1].

13.1 Linear Programming Duality

Consider the general linear program

$$\begin{array}{rl} \text{minimize} & c^T x \\ \text{subject to} & Ax \geq b \\ & x \geq 0 \end{array}$$

where $x = (x_1, \ldots, x_n)^T$ is vector of variables, $A = (A_{ij})$ is an $m \times n$ matrix, and $c = (c_1, \ldots, c_n)^T$ and $b = (b_1, \ldots, b_m)^T$. Recall that the dual of this linear program is given by

$$\begin{array}{rl} \text{maximize} & b^T y \\ \text{subject to} & A^T y \leq c \\ & y \geq 0 \end{array}$$

where $y = (y_1, \ldots, y_m)^T$ is a vector of variables. The variables in y are in one-to-one correspondence with the constraints of the primal LP, and the variables in x are in one-to-one correspondence with the constraints of the dual LP. In the last lecture we showed that the dual of the dual of an LP is again the original LP. We also proved the following result, stating that the objective value of every feasible solution to the dual lower bounds the objective value of every feasible solution to the primal. We reiterate the proof since it will come in handy later.

Theorem 13.1.1 (Weak Duality Theorem) *If x and y are feasible solutions to the primal and dual respectively, then $c^T x \geq b^T y$.*

Proof: We have

$$c^T x = \sum_{i=1}^{n} c_i x_i \geq \sum_{i=1}^{n} \Big(\sum_{j=1}^{m} A_{ji} y_j \Big) x_i = \sum_{j=1}^{m} \Big(\sum_{i=1}^{n} A_{ji} x_i \Big) y_j \geq \sum_{j=1}^{m} b_j y_j = b^T y,$$

where the two inequalities hold by feasibility of x and y. ■

We also stated, but did not prove, the following result, which establishes an intimate connection between the primal and dual LPs.

Theorem 13.1.2 (Strong Duality Theorem) *The optimal objective value of the primal is finite if and only if the optimal objective value of the dual is finite, and in this case the optimal objective values are equal.*

If x and y are optimal solutions to the primal and dual LPs, then Theorem 13.1.2 tells us that $c^T x = b^T y$, and it follows that both inequalities in the proof of Theorem 13.1.1 must hold with equality. That is,

$$\sum_{i=1}^{n} c_i x_i = \sum_{i=1}^{n} \Big(\sum_{j=1}^{m} A_{ji} y_j \Big) x_i$$

and

$$\sum_{j=1}^{m} b_j y_j = \sum_{j=1}^{m} \Big(\sum_{i=1}^{n} A_{ji} x_i \Big) y_j.$$

Let us consider the first equation. Since $\sum_{j=1}^{m} A_{ji} y_j \leq c_i$ for all i, the only way the first equation can hold is if $c_i x_i = (\sum_{j=1}^{m} A_{ji} y_j) x_i$ for all i. This is certainly true if $c_i = \sum_{j=1}^{m} A_{ji} y_j$ for all i, i.e. all dual constraints are tight. However, this is not necessary; even if $c_i < (\sum_{j=1}^{m} A_{ji} y_j)$ for some i, we can still have $c_i x_i = (\sum_{j=1}^{m} A_{ji} y_j) x_i$ provided $x_i = 0$. Similarly, for all j, either $y_j = 0$ or $b_j = \sum_{i=1}^{n} A_{ji} x_i$. Conversely, if x and y are feasible solutions to the primal and dual respectively such that these conditions hold, then equality holds throughout the proof of Theorem 13.1.1, implying that x and y have the same objective value and are thus both optimal. This proves the following result.

Lemma 13.1.3 *Let x and y be feasible solutions to the primal and dual respectively. Then x and y are both optimal if and only the following two conditions hold.*

Primal complementary slackness conditions: *For $i = 1, \ldots, n$, either $x_i = 0$ or $\sum_{j=1}^{m} A_{ji} y_j = c_i$.*

Dual complementary slackness conditions: *For $j = 1, \ldots, m$, either $y_j = 0$ or $\sum_{i=1}^{n} A_{ji} x_i = b_j$.*

This suggests an approach for finding optimal solutions to the primal and dual LPs: search for feasible solutions satisfying both complementary slackness conditions. We can use this idea to obtain approximation algorithms by searching for feasible solutions satisfying a relaxed version of the complementary slackness conditions. We say that x and y satisfy the *α-approximate dual complementary slackness conditions* if for $j = 1, \ldots, m$, either $y_j = 0$ or $\sum_{i=1}^{n} A_{ji} x_i \leq \alpha b_j$. That is, when $y_j \neq 0$, we don't require the corresponding primal constraint to be tight, but it shouldn't be too far from being tight. The following lemma is useful for proving approximation guarantees.

Lemma 13.1.4 *Suppose x and y are feasible solutions to the primal and dual respectively, satisfying the primal complementary slackness conditions and the α-approximate dual complementary slackness conditions. Then x is an α-approximate solution to the primal LP.*

Proof: We have

$$c^T x = (A^T y)^T x = y^T A x \leq \alpha y^T b,$$

where the first equality follows from the primal complementary slackness conditions and the inequality follows from the α-approximate dual complementary slackness conditions. The lemma follows since $y^T b$ is at most the optimal objective value of the primal LP, by Theorem 13.1.1. ■

Our strategy is to construct feasible solutions x and y such that x is integral and the primal complementary slackness conditions and α-approximate dual complementary slackness conditions are satisfied. We do so *without actually solving the LP*, which makes this approach appealing from a practical standpoint. Lemma 13.1.4 then guarantees that x is an α-approximate solution to the LP relaxation and hence an α-approximate solution to the problem at hand. This is the main idea behind the primal-dual method.

This is not the only way to design primal-dual algorithms, however. For example, our primal-dual algorithm for the Steiner Forest problem does not satisfy the 2-approximate complementary slackness conditions for every j, yet we can show that nevertheless, we obtain a 2-approximate solution.

13.2 Vertex Cover

13.2.1 Linear Programming Formulation

Our first example of a primal-dual algorithm is for the weighted version of the Vertex Cover problem.

Definition 13.2.1 (Vertex Cover) *Given a graph $G = (V, E)$ and vertex weights $w : V \to \mathbb{R}^+$, find a minimum weight subset of the vertices such that every edge is covered.*

We can capture this problem with an ILP that is very similar to the ILP we saw in a previous lecture for the unweighted version. We introduce a variable x_v for each vertex v to indicate whether v is in the chosen vertex cover. We seek to minimize the total weight of the picked vertices subject to the constraint that for every edge, at least one of its endpoints is picked.

$$\begin{array}{rll} \text{minimize} & \sum_{v \in V} w_v x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \forall \{u, v\} \in E \\ & x_v \in \{0, 1\} & \forall v \in V \end{array}$$

We relax this to the following LP.

$$\begin{array}{rll} \text{minimize} & \sum_{v \in V} w_v x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \forall \{u, v\} \in E \\ & x_v \geq 0 & \forall v \in V \end{array}$$

The dual of this LP can be written by inspection. Since the constraints of the primal correspond to the edges of G, we have a dual variable y_e for each edge.

$$\begin{array}{rll} \text{maximize} & \sum_{e \in E} y_e & \\ \text{subject to} & \sum_{e : v \in e} y_e \leq w_v & \forall v \in V \\ & y_e \geq 0 & \forall e \in E \end{array}$$

Incidentally, if all vertex weights are 1, then the ILP corresponding to this dual LP exactly captures the Maximum Matching problem. Theorem 13.1.1 then tells us that the cardinality of every matching is a lower bound on the size of every vertex cover, which is exactly the lower bound we used to design a 2-approximation algorithm for the unweighted version of Vertex Cover in the first lecture. That algorithm can be viewed as a primal-dual algorithm and the algorithm we are about to describe can be viewed as a generalization.

13.2.2 Primal-Dual Algorithm

We begin by giving a high-level overview of the primal-dual method. We assume throughout this discussion that the problem at hand is a minimization problem where both c and b are positive. We start with a dual feasible solution and a primal infeasible solution, typically $y = (0, \ldots, 0)^T$ and $x = (0, \ldots, 0)^T$. These solutions certainly satisfy the primal and dual complementary slackness conditions. We iteratively modify x and y, making x closer to being feasible and y closer to being optimal, while maintaining the dual feasibility of y, the primal complementary slackness conditions, and the α-approximate dual complementary slackness conditions. If we can get to a point where x is feasible, then we can use Lemma 13.1.4 to conclude that x is an α-approximate solution.

A more detailed plan is as follows.

- Start with $x = (0, \ldots, 0)^T$ and $y = (0, \ldots, 0)^T$.
- Intuitively, we want our dual solution y to have as high an objective value as possible, since this is our lower bound. So begin by raising some y variables, improving the dual objective value, until some dual constraint goes tight. The crucial design aspect is deciding how to raise the variables — this depends on the structure of the problem at hand.
- When some dual constraint goes tight, we are then free to raise the x variable corresponding to that constraint while maintaining the primal complementary slackness condition. This makes the primal solution x closer to being feasible. In particular, we usually raise the x variable in such a way that all the primal constraints involving this variable are satisfied.
- At this point, raising any of the y variables involved in the constraint that just went tight would violate that constraint, so "freeze" all of these variables and repeat the whole process, alternating between raising y and raising x, and only selecting unfrozen y variables to raise.
- Finally, show that when all y variables become frozen, the primal solution x becomes feasible, and that the α-approximate dual complementary slackness conditions are satisfied.

We now show how to employ this method to obtain a 2-approximation algorithm for Vertex Cover. We start out with $x = (0, \ldots, 0)^T$, i.e. no vertex is picked, and $y = (0, \ldots, 0)^T$, i.e. every edge has a label of 0. We seek to improve the feasibility of the primal, so we pick an arbitrary unsatisfied edge e and raise its label y_e until some dual constraint, say the one corresponding to vertex v, goes tight, which is to say the sum of the labels of edges incident with v is exactly the weight of v. The fact that the primal complementary slackness conditions would still be satisfied if we set $x_v = 1$ is a hint that we should pick v to be in the cover. So we set $x_v = 1$, and since increasing the label

of any edge incident with v would violate the dual constraint corresponding to v, we freeze the dual variables associated with these edges. If the constraints corresponding to both endpoints of e go tight at the same time, then we put both of them in the cover and freeze all edges incident with them. Then we repeat the process by picking an unfrozen edge, raising its label until some dual constraint goes tight, putting the corresponding vertex in the cover, and freezing the edges incident with it. We continue until all edges are frozen, and then output the set of vertices v such that $x_v = 1$.

We remark that although we think of the dual variable y_e as being raised in continuous time, an implementation of this algorithm just needs to set $y_e = \min_{v \in e} (w_v - \sum_{e' \ni v} y_{e'})$.

Lemma 13.2.2 *At termination, y is dual feasible.*

Proof: When a dual constraint goes tight, we freeze all dual variables appearing in it, so this constraint cannot be violated. ■

Lemma 13.2.3 *At termination, the primal complementary slackness conditions are satisfied.*

Proof: We only set $x_i = 1$ when the corresponding dual constraint is tight. ■

Lemma 13.2.4 *At termination, x is primal feasible.*

Proof: Suppose for contradiction that some edge e is not covered by the corresponding vertex cover. Since all frozen edges are incident with picked vertices, it must be the case that e is not frozen. But the algorithm does not terminate with unfrozen edges. ■

Lemma 13.2.5 *At termination, the 2-approximate dual complementary slackness conditions are satisfied.*

Proof: This follows from the fact that for each edge $\{u, v\}$, $x_u \leq 1$ and $x_v \leq 1$ and therefore $x_u + x_v \leq 2$. ■

Although trivial to prove, the above lemma dictates our final approximation guarantee.

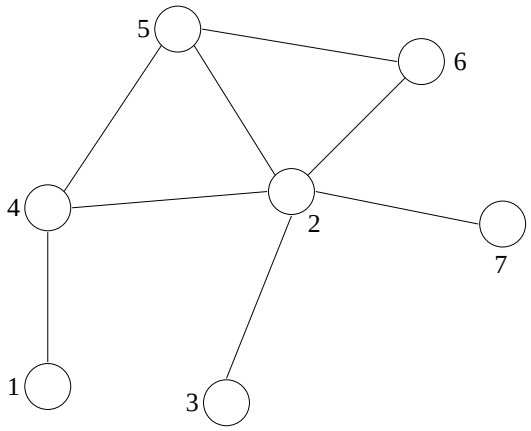
Theorem 13.2.6 *The primal-dual algorithm described above is a 2-approximation algorithm for Vertex Cover.*

Proof: This follows from Lemma 13.1.4 and the previous four lemmas. ■

13.2.3 Examples

We illustrate the behavior of our algorithm on the example in the following figure.

The algorithm may begin by choosing to raise the edge between the vertices of weight 4 and 5. Once the label of this edge hits 4, the constraint corresponding to the vertex of weight 4 goes tight, since the sum of the labels of the edges incident with it is $4 + 0 + 0 = 4$. This vertex is picked to be in the cover, and the three edges incident with it become frozen. The algorithm may then pick the edge between the vertices of weight 5 and 6 to raise. Once the label of this edge hits 1, the constraint corresponding to the vertex of weight 5 goes tight, since the sum of the labels of the edges incident with it is $4 + 0 + 1 = 5$. This vertex is picked to be in the cover, and the two edges incident with it that weren't already frozen become frozen. There are now three unfrozen edges; the algorithm may pick the edge between the vertices of weight 2 and 7 to raise. Once the label of



this edge hits 2, the constraint corresponding to the vertex of weight 2 goes tight, so this vertex is picked to be in the cover, and the remaining three edges become frozen. Now each edge is frozen, so each edge is incident on a picked vertex. The algorithm outputs the vertices of weight 2, 4, and 5, which form a vertex cover of total weight 11. Note that the output is not optimal; the vertex of weight 4 could be replaced with the vertex of weight 1 to yield a vertex cover of total weight 8.

Finally, we observe how this algorithm behaves when all vertex weights are 1. It selects an edge and raises the corresponding variable to 1, at which point the constraints corresponding to both endpoints go tight, so both endpoints are picked to be in the cover. All edges incident with these two endpoints become frozen, so every edge raised in the future will not share an endpoint with the edge raised in this iteration. In other words, the algorithm finds a maximal matching in G and outputs all endpoints of edges in the matching. This is exactly the 2-approximation algorithm for the unweighted version of Vertex Cover that was described in the first lecture.

13.3 Steiner Forest

13.3.1 Linear Programming Formulation

The primal-dual algorithm we just described for Vertex Cover worked by raising dual variables one at a time. Our next example is a primal-dual algorithm that operates differently; it raises many dual variables simultaneously until one of the corresponding primal constraints goes tight. This algorithm also has the feature that Lemma 13.1.4 does not apply directly; a more ad-hoc argument is needed. We choose to study the following problem instead of the Steiner Tree problem since the extra generality doesn't end up complicating the algorithm or analysis significantly.

Definition 13.3.1 (Steiner Forest) *Given a graph $G = (V, E)$, edge costs $c : E \to \mathbb{R}^+$, and sets $S_i \subseteq V$, find a minimum-cost forest F such that for all i and all $u, v \in S_i$, there is a path from u to v in F.*

Our first task is to fomulate this problem as an ILP. There are several ways of doing this. One natural approach would be to have a variable for every edge indicating whether it is in the forest,

as well as a variable for each path connecting two vertices in the same S_i, and constraints enforcing that every pair of vertices in the same S_i is connected by some path and that every edge on this path is indeed picked. This yields an ILP with exponentially many constraints. A polynomial-sized ILP can be obtained by expressing the problem as a flow problem, with a different commodity for each pair of vertices in the same S_i. We will use an equivalent but more structured formulation with a dual that is simpler to state.

We still have a variable x_e for each edge indicating whether it is in the forest, but instead of thinking of pairs of vertices in the same S_i as being *connected*, we think of them as being *not disconnected*. That is, every cut separating them must contain some picked edge. More formally, for every $S \subseteq V$ such that $S \cap S_i \notin \{\emptyset, S_i\}$ for some i, we require that $x_e = 1$ for at least one edge $e \in \delta(S)$ have . Letting $\mathcal{S}$ denote the set of all such S, we have the following ILP formulation.

$$\begin{array}{rll} \text{minimize} & \sum_{e \in E} c_e x_e & \\ \text{subject to} & \sum_{e \in \delta(S)} x_e \geq 1 & \forall S \in \mathcal{S} \\ & x_e \in \{0, 1\} & \forall e \in E \end{array}$$

An immediate concern is that the above ILP has exponentially many constraints. However, this serves to illustrate a key point about the primal-dual method: the linear programming formulation of a problem is merely a conceptual tool; the algorithms are purely combinatorial. In this case, our algorithm only ever needs to deal with polynomially many of the primal constraints and dual variables, so the exponential size of the LP formulation is not prohibitive.

We obtain the LP relaxtion for this problem.

$$\begin{array}{rll} \text{minimize} & \sum_{e \in E} c_e x_e & \\ \text{subject to} & \sum_{e \in \delta(S)} x_e \geq 1 & \forall S \in \mathcal{S} \\ & x_e \geq 0 & \forall e \in E \end{array}$$

For the dual, we introduce a variable y_S for each $S \in \mathcal{S}$. The dual LP can be written by inspection.

$$\begin{array}{rll} \text{maximize} & \sum_{s \in \mathcal{S}} y_S & \\ \text{subject to} & \sum_{S : e \in \delta(S)} y_S \leq c_e & \forall e \in E \\ & y_S \geq 0 & \forall S \in \mathcal{S} \end{array}$$

13.3.2 Primal-Dual Algorithm

Our primal-dual algorithm starts with the primal infeasible solution $x = (0, \ldots, 0)^T$ and the dual feasible solution $y = (0, \ldots, 0)^T$. We would like to raise some dual variables, thereby increasing the dual objective, until some dual constraint goes tight, at which time the primal complementary slackness conditions would indicate that it is safe to put the corresponding edge in our forest. For the Vertex Cover problem, we picked a single unsatisfied primal constraint and raised the corresponding dual variable until some dual constraint went tight. The order in which the unsatisfied constraints were picked didn't affect the final cost much. In the present settting, it turns out that raising one dual variable at a time does not work. The costs for satisfying different constraints can vary

wildly, so ordering matters more. Instead, we raise many dual variables simultaneously until some dual constraint goes tight. This way, we are not focusing on making progress toward satisfying any particular primal constraint, but rather taking a more global perspective.

Another difference between our Vertex Cover algorithm and the current example is that we are not able to guarantee that the α-approximate dual complementary slackness conditions are satisfied. But nevertheless we can show that our algorithm yields a 2-approximate solution, by a similar argument to the one used to show that if the 2-approximate dual complementary slackness conditions are satisfied then the resulting solution is 2-approximate. In particular, we will show that these constraints hold "on average".

We will see more details about this algorithm in the next lecture. We will also see a primal-dual algorithm for the Facility Location problem.

References

[1] M. X. Goemans and D. P. Williamson. A General Approximation Technique for Constrained Forest Problems. In *SIAM Journal on Computing*, 24, 1995, pp. 296-317.

CS880: Approximations Algorithms

Scribe: Chi Man Liu **Lecturer:** Shuchi Chawla

Topic: Primal-Dual Method: Steiner Forest and Facility Location **Date:** 3/8/2007

Last time we discussed how LP-duality can be applied to approximation. We introduced the primal-dual method for approximating the optimal solution to an LP in a combinatorial way. We applied this technique to give another 2-approximation for the Vertex Cover problem. In this lecture, we present and analyze a primal-dual algorithm that gives a 2-approximation for the Steiner Forest problem. We also briefly talk about a primal-dual algorithm for Facility Location.

14.1 Steiner Forest

We restate the Steiner Forest problem here.

Definition 14.1.1 (Steiner Forest) *Given an undirected graph $G = (V, E)$, edge costs $c : E \to \mathbb{R}^+$, and disjoint subsets $S_i \subseteq V$, find a minimum-cost forest F such that for all i and $u, v \in S_i$, there exists a path connecting u to v in F.*

To formulate Steiner Forest as an ILP, we introduce an indicator variable x_e for each edge e. Let $\mathcal{S}$ be the collection of all sets $S \subseteq V$ that cut some S_i into two parts, i.e. $\mathcal{S} = \{S \in V \mid S \cap S_i \notin \{\emptyset, S_i\} \text{ for some } i\}$. Then from last lecture we have the following ILP formulation for Steiner Forest.

$$\begin{array}{rll} \text{minimize} & \sum_{e \in E} c_e x_e & \\ \text{subject to} & \sum_{e \in \delta(S)} x_e \geq 1 & \forall S \in \mathcal{S} \\ & x_e \in \{0, 1\} & \forall e \in E \end{array}$$

where $\delta(S)$ denotes the boundary of S, i.e. the set of edges in E with exactly one endpoint in S. The LP relaxation for this problem is as follows. Note that we do not restrict x_e to be at most 1 since any optimal solution x^* to the LP must satisfy $x_e^* \leq 1$ for all $e \in E$.

$$\begin{array}{rll} \text{minimize} & \sum_{e \in E} c_e x_e & \\ \text{subject to} & \sum_{e \in \delta(S)} x_e \geq 1 & \forall S \in \mathcal{S} \\ & x_e \geq 0 & \forall e \in E \end{array}$$

We introduce a variable y_S for each $S \in \mathcal{S}$ and obtain the dual LP.

$$\begin{array}{rll} \text{maximize} & \sum_{S \in \mathcal{S}} y_S & \\ \text{subject to} & \sum_{S : e \in \delta(S)} y_S \leq c_e & \forall e \in E \\ & y_S \geq 0 & \forall S \in \mathcal{S} \end{array}$$

If $c_e = 1$ for all e and integer constraints are imposed, the dual LP can be interpreted as the problem of picking a largest collection $\mathcal{C} \subseteq \mathcal{S}$ such that no two sets in $\mathcal{C}$ have a common edge on their boundaries.

We start with the primal infeasible solution $x = (0, \ldots, 0)^T$ and the dual feasible solution $y = (0, \ldots, 0)^T$. We then raise some of the dual variables until some dual constraint goes tight. At this

point, we put the edge corresponding to the constraint into F, and freeze all dual variables in the constraint. We repeat this process, all the time keeping the dual solution feasible, until all dual variables are frozen. A detailed description of our primal-dual algorithm is as follows.

1. Set $x_e = 0$ for all $e \in E$, and $y_S = 0$ for all $S \in \mathcal{S}$.
2. Raise uniformly the dual variables y_S's corresponding to all **minimal** unsatisfied sets.
3. When a dual constraint goes tight, assign the value 1 to the primal variable x_e corresponding to the constaint, and freeze all dual variables y_S in the constraint. Without loss of generality, we assume that at most one dual constraint goes tight at any point in time.
4. Repeat the above two steps until all y_S are frozen.

After the above primal-dual steps, we get a collection of edges F. Finally, we need a pruning step to remove extra edges from F.

5. For all edges $e \in F$ such that $F \backslash \{e\}$ is still feasible, remove e from F. We denote the resulting collection of edges by F'.

Intuitively, we can think of the algorithm as expanding minimal unsatisfied sets (*active sets*) until some of them touch on some edge, at which time we merge the active sets to form a larger set. The newly formed set may be active or inactive. If it is active, we start expanding it with other existing active sets. Initially, each terminal node forms an active set. The above process continues until all sets become inactive. Also note that when we expand an active set, it may touch a non-Steiner node. In such a case, a new set is formed by adding that node into the active set.

Lemma 14.1.2 *At termination, y is dual feasible.*

Proof: In our algorithm, whenever a dual constraint goes tight, all its variables are frozen, therefore no dual constraints can be violated throughout the algorithm. Since all dual constraints are satisfied at the beginning, they remain satisfied. ■

By weak duality, we have the following corollary.

Corollary 14.1.3 *Let OPT be the cost of an optimal Steiner forest. Then $\sum_{S \in \mathcal{S}} y_S \leq OPT$.*

Lemma 14.1.4 *At the end of the primal-dual steps, F is a forest and is primal feasible.*

Proof: Since we consider dual constraints one by one, and always add edges between two minimal unsatisfied sets, we never form a cycle, and thus F is a forest. If F was not feasible, then some $S \in \mathcal{S}$ remained unsatisfied upon termination. That means none of the edges in $\delta(S)$ were in F; equivalently, y_S had not been frozen. This contradicts step 4 of our algorithm. ■

Lemma 14.1.5 *F and y satisfy the primary complementary slackness conditions.*

Proof: We only raise the value of a primal variable in step 3 of our algorithm. Such an event happens only when the corresponding dual constraint goes tight. Hence, $x_e > 0$ implies that $\sum_{S: e \in \delta(S)} y_S = c_e$. ■

In view of Corollary 14.1.3 and Lemma 14.1.5, we may want to show that $y_S > 0$ implies $\sum_{e \in \delta(S)} x_e \leq 2$. Then this would give

$$\sum_{e \in F} c_e = \sum_{e \in F} \sum_{S: e \in \delta(S)} y_S = \sum_{S \in \mathcal{S}} \sum_{e \in F \cap \delta(S)} y_S = \sum_{S \in \mathcal{S}} y_S \sum_{e \in \delta(S)} x_e \leq 2 \sum_{S \in \mathcal{S}} y_S \leq 2OPT.$$

However, this is not true in general, because average degree of the sets can be high when the sets become inactive. That is why we need the pruning step to remove extra degrees. We now shift our attention to the pruned solution F'.

Lemma 14.1.6 *F' is a forest and is primal feasible.*

Proof: F' is a forest follows from the fact that its superset F is forest. To show that F' is feasible, pick an i and $u, v \in S_i$. Since F is feasible and is a forest, there exists a unique path in F connecting u to v. Removing any edge in the path would make the solution infeasible, therefore the whole path must be contained in F'. ■

Lemma 14.1.7 *F' and y satisfy the primary complementary slackness conditions.*

Proof: This is a direct corollary of Lemma 14.1.5. ■

Lemma 14.1.8 *Let OPT be the cost of an optimal Steiner forest. Then*

$$\sum_{e \in F'} c_e \leq 2 \sum_{S \in \mathcal{S}} y_S \leq 2OPT.$$

Proof: By Lemma 14.1.7,

$$\sum_{e \in F'} c_e = \sum_{e \in F'} \sum_{S: e \in \delta(S)} y_S.$$

Rewriting the summations on the right-hand side, we get

$$\begin{aligned} \sum_{e \in F'} c_e &= \sum_{S \in \mathcal{S}} \sum_{e \in F' \cap \delta(S)} y_S \\ &= \sum_{S \in \mathcal{S}} y_S \cdot deg_{F'}(S) \end{aligned}$$

where $deg_{F'}(S)$ is the number of edges in F' crossing the boundary of S, i.e. $|F' \cap \delta(S)|$. Our goal is to show that $\sum_{S \in \mathcal{S}} y_S \cdot deg_{F'}(S) \leq 2 \sum_{S \in \mathcal{S}} y_S$. We use an inductive argument to show that the inequality holds at any point in time. The base case ($t = 0$) is trivial since $\sum_{S \in \mathcal{S}} y_S = 0$. For the inductive part, we assume that the inequality is satisfied at time t. Let $\Delta > 0$ be an amount such that no dual constraint goes tight in the time interval $[t, t + \Delta)$. Let $\mathcal{A}(t)$ be the collection of all active sets at time t. Then, the increase in the left-hand side of the inequality within $[t, t + \Delta]$ is

$$\Delta \cdot \sum_{S \in \mathcal{A}(t)} deg_{F'}(S),$$

whereas the increase in the right-hand side is

$$2\Delta \cdot |\mathcal{A}(t)|.$$

We claim that any time t, the average degree of all active sets is at most 2, i.e. $\sum_{S \in \mathcal{A}(t)} deg_{F'}(S) \leq 2 \cdot |\mathcal{A}(t)|$. If that holds, then the increase in LHS is at most the increase in RHS, thus completing our induction. To prove the claim, we consider all components of F at time t. The average degree of these components under the final F is at most 2, implying that the average degree of these components under F' is at most 2. Some of these components are active sets while others are inactive. If we can show that all the inactive ones have degree (under F') not equal to 1, then we are done. We prove this by contradiction. Suppose that for some inactive component S, $deg_{F'}(S) = 1$. Let e be the only edge in $F' \cap \delta(S)$. Then e must lie on some path in F' connecting $u, v \in S_i$ for some i, otherwise it would have been removed from F'. But then $u \in S$ and $v \notin S$ (or vice versa), implying that $S \in \mathcal{S}$, which is a contradiction since we have assumed that S is inactive. ■

Remark. The above 2-approximation is due to Agarwal, Klein and Ravi [1]. Later, Goemans and Williamson [2] started to use the primal-dual method in solving many other approximation problems.

14.2 Facility Location

We restate the Facility Location problem here.

Definition 14.2.1 (Facility Location) *Given a set of facilities I, a set of customers J, facility opening costs $f : I \to \mathbb{R}^+$ and a metric c on $I \cup J$, find a subset $S \subseteq I$ that minimizes the total cost $C(S) = C_f(S) + C_r(S)$, where $C_f(S)$ is the total facility openning cost defined as $C_f(S) = \sum_{i \in S} f(i)$, and $C_r(S)$ is the total routing cost defined as $C_r(S) = \sum_{j \in J} \min_{i \in S} c(i, j)$.*

To formulate Facility Location as an ILP, we introduce an indicator variable x_i for each facility i. For each facility i and customer j, we introduce a variable y_{ij} which is set to 1 if and only if j is assigned to i. For notational convenience, we write f_i and c_{ij} instead of $f(i)$ and $c(i, j)$. Then we have the following ILP formulation for Facility Location.

$$\begin{array}{rll} \text{minimize} & \sum_{i \in I} f_i x_i + \sum_{i \in I, j \in J} c_{ij} y_{ij} & \\ \text{subject to} & \sum_{i \in I} y_{ij} \geq 1 & \forall j \in J \\ & x_i - y_{ij} \geq 0 & \forall i \in I, j \in J \\ & x_i, y_{ij} \text{ are non-negative integers} & \forall i \in I, j \in J \end{array}$$

By introducing a variable α_j for each customer j and β_{ij} for each facility-customer pair (i, j), we get the following dual of the LP relaxation.

$$\begin{array}{rll} \text{maximize} & \sum_j \alpha_j & \\ \text{subject to} & \alpha_j - \beta_{ij} \leq c_{ij} & \forall i \in I, j \in J \\ & \sum_j \beta_{ij} \leq f_i & \forall i \in I \\ & \alpha_i, \beta_{ij} \geq 0 & \forall i \in I, j \in J \end{array}$$

We can interpret the dual LP as follows. We are the owners of the facilities and are going to collect money from the customers in order to open some of the facilities. For each customer j, α_j is the amount paid by j. For each facility i and customer j, β_{ij} is the portion of α_j that pays for facility i. The constraint $\alpha_j - \beta_{ij} \leq c_{ij}$ states that the amount paid by customer j should not exceed

the portion that goes to facility i plus the routing cost c_{ij}, for every facility i. The constraint $\sum_j \beta_{ij} \leq f_i$ states that no facility should overcharge its customers, i.e. the customers should not pay more than enough to open the facility. Under these constraints, we want to maximize $\sum_j \alpha_j$, the total amount collected from the customers. Note that each customer may be paying for several facilities. To deal with this, we close some of the facilities in our primal solution such that every customer pays for at most one facility, and reassign all unassigned customers to the remaining facilities. We will look at the primal-dual algorithm next lecture.

References

[1] A. Agrawal, P. Klein and R. Ravi. When Trees Collide: An Approximation Algorithm for the Generalized Steiner Problem on Networks. In *SIAM Journal on Computing*, 24, pp. 440–456, 1995.

[2] M. X. Goemans and D. P. Williamson. A General Approximation Technique for Constrained Forest Problems. In *SIAM Journal on Computing*, 24, pp. 296–317, 1995.

CS880: Approximations Algorithms

Scribe: Matthew Darnall **Lecturer:** Shuchi Chawla
Topic: Primal/Dual Method: Facility Location/Cut **Date:** 3/08

17.1 Facility Location

Recall from last time the formulation of the facility location problem as an LP with primal and dual. We have costs c_{ij} of routing a person j to facility i. There is a cost f_i of opening facility i. We have a variable x_i for each facility i that is the amount that the facility is open. We have a variable y_{ij} for the amount that person j is routed to facility i. We wish to minimize

$$\sum_i f_i x_i + \sum_{i,j} y_{ij} c_{ij}$$

suject to the constraints that for each j

$$y_{ij} \leq x_i$$

and for each j

$$\sum_i bij \geq 1$$

The dual problem then has variables α_j for every person j and β_{ij} for every customer/facility pair. The dual problem is to maximize

$$\sum_j \alpha_j$$

subject to the constraints that for any i, j

$$\alpha_j - \beta_{ij} \leq c_{ij}$$

and for any i

$$\sum_j \beta_{ij} \leq f_i$$

as well as the usual condition that all variables are non-negative.

17.2 Algorithm

We shall construct an integral solution to the primal LP as well as a feasible solution to the dual LP such that the integral primal is within a factor of three of the dual solution. Thus, our final solution shall be a three approximation for the facility location problem, since any dual feasible solution is an upper bound on a primal solution. As we usually do, we shall ensure that one of the complementary slackness conditions remains tight, while relaxing the other. This time, we shall relax the primal slackness condition. We shall do the following to get a set I of possible facilities to open. We continue until all the people are assigned a facility.

For all unassigned customers, raise the corresponding α_j uniformly. If an equaltiy of the form $\alpha_j = c_{ij}$ is reached the corresponding β_{ij} must be raised also to ensure that the constraint $\alpha_j - \beta_{ij} \leq c_{ij}$ isn't violated. Remember that we want this to be a feasible dual solution. For such customers, $\alpha_j - \beta_{ij}$ is fixed to be c_{ij} and that constraint is tight. Once a constraint of the form $\sum_j \beta_{ij} \leq f_i$ is reached for some i, we include this i in our set I of possible facilities to open. Also, we consider the j that have nonzero b_{ij} to be assigned to i and freeze the values of β_{ij} and α_i. Then we continue the process.

At this point, we have a set of facilities I that we wish to open. We call a pair i, j tight if $\alpha_j - \beta_{ij} = c_{ij}$ at the time i was included in the set I. We shall decide the final facilities to open by dong the following. There is a natural ordering on the facilities by the time at which they were included in I. We call a pair i, j tight if $\alpha_j = c_{ij}$. Now, we recursively do the following. Select the first facility in I that is not thrown out or already selected. Now, throw out all the facilities that share a tight customer with the selected facility. Continue until all facilities are selected or thrown out. The selected facilities shall be the ones we open, call this set of facilities S. We shall route each customer completely to a tight facility if one exists. If one doesn't exist, we route the customer to a facility that caused one of the tight facilities to the customer to be thrown out. By the metric propery, we know that the cost of routing to the open facility shall be bounded appropriately.

17.3 Analysis

For each customer, we break down the α_j into two portions, the portion payng for facility opening α_j^f and a portion for routing α_j^r. If j is assigned to a facility with which it is tight, then $\alpha_j = \beta_{ij} + c_{ij}$ for that facility. We call $\alpha_j^f = \beta_{ij}$ and $\alpha_j^r = c_{ij}$. Else, $\alpha_j^r = \alpha_j$ and the facility portion for this α_j is zero. Let S_i be the set of customers assigned to facility i. We notice that for any facility in S, $\sum_{j \in S_i} \alpha_j^f = \sum_{j \in S_i} \beta_{ij} = f_i$. Since these S_i are disjoint, $\sum_j \alpha_j^f$ is precisely the opening cost of the facilities in S. Now, for the routing costs we have that for j with a tight facility α_j^r is exactly the cost of routing j to that facility. For the other j, we don't have a tight facility to route to. But, the facility that j is routed to must share a customer, j', with a tight facility, i', for j. By the order in which we chose the facilities to include, the routing cost from i to j', from i' to j' and from i' to j are all bounded by α_j, since each of these pairs is tight. The metric property gives us that the cost paid to route j is then at most $3\alpha_j$. So, we have that:

$$3 \sum_j \alpha_j \geq \sum_{i \in S} f_i + \sum_{i,j} c_{ij}$$

Since the RHS is the objective function of the original problem and the optimal value for facility location is bounded by the sum of the α_j, we get a 3 approximation to facility location. This idea was first seen in [1].

17.4 Min Cut/Max Flow

Recall the Min Cut problem. We are given two nodes, s and t, in a graph G. We wish to find the cheapest set of edges to remove such that s and t are in different components of the graph. We shall consider the Min Cut problem in our primal dual setting. First, we must state the Min Cut problem as an ILP. To do this, we can give a variable x_e to each edge in the graph that stands for whether or not we remove e. Let c_e be the cost of removing e. We seek to minimize:

$$\sum_e c_e x_e$$

The constraints we have say that there should be no path from s to t that doesn't cross a selected edge. Thus, for every path P from s to t:

$$\sum_{e \in P} x_e \geq 1$$

As we saw in lecture 12, the dual of this LP describes the Max Flow problem. Using the Duality Theorem and the fact that the basic solutions of these LPs are integral, we get the famous Max Flow = Min Cut Theorem.

17.5 Metric LPs

Though the Min Cut problem is simple enough not to need another formulation, it is useful to use the Min Cut to expalin Metric LPs. Notice that any cut, C, defines a metric, d_C on the set of vertices. Namely, the distance $d_C(x, y)$ from x to y is length of the shortest path from x to y where the length traveled by moving across edge e is x_e. x_e is 1 if e is cut, zero otherwise. Now, our goal is to:

$$\text{Minimize} \qquad \sum_e c_e x_e \tag{17.5.1}$$

$$\text{Subject to} \quad d_C(s, t) \geq 1 \tag{17.5.2}$$

Unfortunately, the constraint $d_C(s, t) \geq 1$ is not linear in the variables x_e. To get over this, we change how we phrase the problem. Instead of assigning a varible x_e for each edge, we have a variable $d(u, v)$ for each pair of vertices. We shall want d to form a metric and shall think of $d(u, v)$ as being the amount we select the edge (if it exists) between u and v. Now we seek to minimize:

$$\sum_{(u,v) \in E} c_{(u,v)} d(u, v)$$

subject to the constraints that for any u, v, w:

$$d(u,v) + d(v,w) - d(u,w) \geq 0 \tag{17.5.3}$$
$$d(u,v) - d(v,u) = 0 \tag{17.5.4}$$
$$d(u,u) \geq 0 \tag{17.5.5}$$
$$d(s,t) \geq 1 \tag{17.5.6}$$

Notice that the first three of these constraints are perfectly encapsulates what it means for d to be a metric! Also, notice that the metric induced by any feasible cut satisfies the constraints. If the cut is optimal, then $\sum_{(u,v)\in E} c_{(u,v)} d(u,v) = \sum_e c_e x_e$ since we can lower the value of $x_{(u,v)}$ down to $d(u,v)$ without violating any of the contraints. Thus, finding the solution to Min Cut is equivalent to finding the best integral solution to the LP above.

As we will see in future lectures, phrasing some problems as a metric LP and using metric embedding techniques can lead to good approximation factors.

References

[1] K. Jain, V. Vazarani. Primal-Dual Algorithms for Mteric Facility Location and$k$-Median Problems. In *40th annual Symposium on the Foundations of Computer Science*, 1999.

CS880: Approximations Algorithms	
Scribe: Dave Andrzejewski	**Lecturer:** Shuchi Chawla
Topic: Metric multiway cut and multicut, integrality gap	**Date:** 3/15/07

The lecture further explores the use of cut metrics, with applications to the multiway cut and multicut problems. Also, the idea of an expander graph is introduced and applied to deriving the integrality gap between an optimal LP solution and the optimal corresponding integral solution.

16.1 Metric multiway cut

16.1.1 Problem setup and LP

GIVEN: a graph $G = (V, E)$ and a set $T \subset V$ of terminals.
DO: find the minimum weight cut separating every pair of terminals $t_i, t_j \in T$ from one another.

Just as in the previous lecture, we will formulate this cut problem as a metric LP. We enforce the separation of all pairs of terminals by requiring that our metric assign them distance ≥ 1.

$$\min \sum_{(u,v)\in E} c_{uv} d(u,v) \qquad \text{obj fcn}$$

$$d \text{ is a metric}$$

$$d(t_i, t_j) \geq 1 \qquad \forall t_i, t_j \in T$$

where c_{uv} is the cost of the edge between vertices u and v.

16.1.2 Rounding

Once we've found an optimal solution to the metric LP above, we need to transform our metric to a cut metric, which will yield a valid multiway cut. As usual, we use the fact that $LP^* \leq OPT$.

To do this analysis, it will be useful to consider a set of interesting physical analogies for the quantities in our problem.

- edges $\rightarrow$ pipes
- $c_e \rightarrow$ cross-sectional area of pipe e
- $d_e \rightarrow =$ length of pipe e
- $B(t_i, r) \rightarrow =$ ball of radius r, centered at t_i
- $f(t_i, r, e) \rightarrow =$ fraction of edge e covered by ball $B(t_i, r)$
- $Vol(B(t_i, r)) \rightarrow =$ total pipe volume enclosed by ball $B(t_i, r)$

- $Area'(B(t_i, r)) \rightarrow =$ the total pipe cross section area on the surface of ball $B(t_i, r)$
- $Area(B(t_i, r)) \rightarrow =$ the total cost (c_e) of the edges crossing the ball $B(t_i, r)$

Note that $Area \neq Area'$ in general, because the surface of the ball may not be perpendicular to the pipe (Figure 16.1.1). In fact, $Area' \geq Area$. The expressions for f,Vol and $Area$ are as below. For f, edge $e = (u, v)$, and f is simply 1 or 0 if both or neither of (u, v) are contained in the ball, with the expression below covering the more interesting case where $u \in B$ and $v \notin B$.

$$f(t_i, r, e) = \frac{r - d(t_i, u)}{d(t_i, v) - d(t_i, u)} \tag{16.1.1}$$

$$Vol(B(t_i, r)) = \sum_{e \in E} f(t_i, r, e) d_e c_e \tag{16.1.2}$$

$$E' = \{(u, v) \text{ s.t. } |B \cap u, v| = 1\} \tag{16.1.3}$$

$$Area(B(t_i, r)) = \sum_{(u,v) \in E'} c_{uv} \tag{16.1.4}$$

$$Area'(B(t_i, r)) = \frac{d}{dr} Vol(B(t_i, r)) = \sum_{(u,v) \in E'} c_{uv} \frac{d(u, v)}{d(t_i, v) - d(t_i, u)} \tag{16.1.5}$$

These quantities will provide an intuitive framework with which to analyze our rounding scheme.

Algorithm

- $\forall i$, pick $r_i = \arg\min_{r \in [0,1/2]} Area(B(t_i, r))$
- let C_i be the cut associated with that radius
- pick the $k - 1$ minimum weight cuts from $C_1, C_2, ..., C_k$

Since each pair of terminals (t_i, t_j) must satisfy $d(t_i, t_j) \geq 1$ in the original LP solution, making our cuts with at a radius $r \leq 1/2$ around each terminal will clearly yield a valid set of separating cuts.

We now derive the approximation factor of our resulting scheme.

Lemma 16.1.1 $Area(t_i, r_i) \leq 2Vol(t_i, 1/2)\ \forall i$

Proof: As the radius of a ball increases, the volume enclosed by the ball will grow proportionally to the surface areas of all pipes currently cut by the surface of the ball.

If all cut pipes were perpendicular to the surface of the expanding ball the volume growth would be exactly equal to the current surface area, but since this is not neccessarily the case (see 16.1.1) $Area$ lower bounds the rate of volume growth ($Area'$).

$$Area(t_i, r) \leq \frac{d}{dr} Vol(t_i, r)$$

Recall that we have chosen r_i to minimize $Area(t_i, r)$ on the interval $r \in [0, 1/2]$. We use this fact to substitute the constant term $Area(t_i, r_i)$ in for $Area(t_i, r)$ above, and then integrate both sides with respect to r with bounds from $r = 0$ to $r = 1/2$. This yields our final result.

$$1/2 Area(t_i, r_i) \leq Vol(t_i, 1/2)$$

■

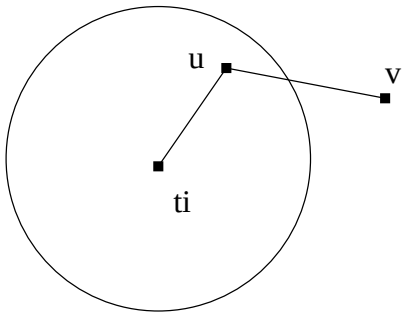


Figure 16.1.1: An edge cut by a ball may not be perpendicular to the ball surface.

Lemma 16.1.2 $\sum_i Vol(t_i, 1/2) \leq LP^*$

Proof: Using the fact that $d(t_i, t_j) \geq 1$ for all terminal pairs, it is clear that all $B(t_i, 1/2)$ will be disjoint. Since the cost of LP^* is equal to the volume of the entire graph, and the set of $B(t_i, 1/2)$ form a disjoint subset of the graph, it must be true that the total ball volume is $\leq LP^*$. ■

The cut produced by our algorithm chooses all edges which are partially cut by the surface of each ball $B(t_i, r_i)$. Thus the total cost is equal to $\sum_i^{k-1} Area(t_i, r_i)$. Recall that we simply omit the most expensive cut, since that terminal is already isolated by the other $k - 1$ cuts. Finally we combine these lemmas to get an approximation factor of $2(1 - 1/k)$.

$$\sum_{i=1}^{k-1} Area(t_i, r_i) \leq 2 \sum_{i=1}^{k-1} Vol(t_i, 1/2) \leq 2(1 - 1/k) LP^* \leq 2(1 - 1/k) OPT$$

16.2 Metric multicut

16.2.1 Problem setup and LP

Multicut is very similar to multiway cut, except we now have k pairs of terminals (s_i, t_i) and our cut only needs to separate each s_i from its correpsonding t_i for all i. The cut does not need to separate different pairs from each other.

The metric LP formulation is identical, except our separation constraint is now $d(s_i, t_i) \geq 1 \ \forall i$.

(As an aside, it is intresting to note that this LP could be reformulated in the 'path-style', where we would require that the path distance x_p between each terminal pair is ≥ 1, for all paths p between the

pair. As in previous lectures, this formulation would have the disadvantage of exponentially many constraints, but could be approached using the ellipsoid method. The ellipsoid method requires an efficient 'separation oracle' to reveal which constraint is violated by any proposed solution. For this problem we could use the results of the polytime all-pairs shortest path algorithm for this purpose.)

16.2.2 Rounding

Given an optimal LP solution, how can we round to a valid cut, and how can we analyze this scheme? The method applied to the multiway cut problem is no longer directly applicable, because the fact that we now only separate *pairs* of terminal nodes means that the B centered at each terminal are no longer guaranteed to be disjoint.

The key idea to overcome this is to only charge the area to the sub-ball entirely enclosed by a given cut. Once the edges are charged to that cut, they are then removed from the graph, potentially changing area and volume calculations for subsequent steps. Crucially, these modifications ensure that the volumes remain disjoint.

Our derivation will involve dividing by $Vol(s_i, 0)$ at some point, which would be an undefined divide-by-zero operation. We avoid this by redefining volume slightly. Call F the total volume of the graph. Then redefine volume by assigning volume F/k to each terminal s_i.

$$Vol'(s_i, r) = F/k + \sum_e f_e c_e d_e$$

Now the total volume of the graph has doubled, so all previous volume lemmas still hold, with both sides multiplied by 2. The initial volume of a ball is then F/k, and its final volume must be $\leq F/k + F$.

Algorithm

- for each i, pick minimum r such that $Area(s_i, r) \leq \alpha Vol(s_i, r)$
- make the cut C_i at that r, remove those edges from the graph
- repeat for next i

We pick the α value to be $2\ln(k+1)$.

Lemma 16.2.1 $r_i \leq 1/2 \ \forall i$

Proof: As before, we know that $Area(s_i, r) \leq \frac{d}{dr} Vol(s_i, r)$.

Now assume $r_i \geq 1/2$. Then we must have that

$$\frac{d}{dr} Vol(s_i, r) > \alpha Vol(s_i, r) \ \forall r < 1/2$$

because otherwise we would have picked one of those smaller r.

We then manipulate the equation, integrating from $r = 0$ to $r = 1/2$.

$$
\begin{aligned}
dVol(s_i, r) &> \alpha Vol(s_i, r)\, dr \\
\frac{1}{Vol(s_i, r)}\, dVol(s_i, r) &> \alpha\, dr \\
\int_0^{1/2} \frac{dVol(s_i, r)}{Vol(s_i, r)} &> \int_0^{1/2} \alpha\, dr \\
\ln\left(\frac{Vol(s_i, 1/2)}{Vol(s_i, 0)}\right) &> \alpha/2
\end{aligned}
$$

Recall that the maximum possible value of Vol is now $F + F/k$, and that $Vol(s_i, 0)$ is now defined to be F/k. Substitute these values in to get another inequality.

$$\ln(1+k) = \ln\left(\frac{F + F/k}{F/k}\right) > \ln\left(\frac{Vol(s_i, 1/2)}{Vol(s_i, 0)}\right) > \alpha/2$$

This gives us $\alpha < 2\ln(1+k)$. Since we have chosen $\alpha = 2\ln(1+k)$, we have derived a contradiction.

■

Lemma 16.2.2 $\sum_i Vol(s_i, r_i) \leq 2LP^* = 2F$

Proof: This is acheived by construction. Our modified disjoint volumes now sum to no more than $2F$. Since LP^* is equal to the volume of the full graph F, this is clearly true. ■

These lemmas will be used in our full derivation of the algoirthm approximation factor.

Theorem 16.2.3 $\sum_i |C_i| = \sum_i Area(s_i, r_i) \leq 2\alpha LP^* \leq 2\alpha OPT$

This theorem simply follows from the combination of the previous lemmas and definitions. Plugging our chosen $\alpha = 2\ln(1+k)$ in shows that this algorithm achieves an $4\log(1+k)$-approximation [4].

Multicut is known to be APX-hard [1], meaning that it cannot be approximated within every constant factor.

Furthermore, if Unique Games Conjecture is true, it cannot even be approximated within any constant factor. A stronger version the Unique Games Conjecture further implies that it cannot be approximated with a factor $\Omega(\log\log n)$ either [3].

16.3 Integrality gap analysis

To derive an integrality gap for a given problem, we find a problem instance in which the optimal integral solution is significantly worse than the optimal fractional solution. The factor by which the integral solution is worse is known as the integrality gap. The existence such a gap bounds the performance of approximation algorithms based on LP rounding, since we cannot approximate LP^* at a factor better than the integrality gap. Our specific approach will be to use a special mathematical stucture known as an expander graph to construct a problem instance with this property.

Definition 16.3.1 *An* α-expander graph $G = (V, E)$ *is a graph with the special property that for any subset* $S \subset V$ *such that* $|S| \leq |V|/2$*, we have* $|E(S, \bar{S})| \geq \alpha|S|$.

The study of expander graphs constitutes a very active research area, and expander graphs have been applied to many different problems in multiple fields [2]. For our purposes it is enough to know the basic definition property, and that there exist explicit methods for constructing graphs that have this, and other, properties.

We will now consider a multicut problem instance. Say that we have a degree 3 expander graph with constant α. Let (s_i, t_i) be the set of all pairs with distance $\geq \beta \log n$.

A feasible fractional solution is then given by

$$x_e = \frac{1}{\beta \log n} \ \forall e$$

By our problem definition, this is a feasible solution. The total cost is then given the by number of edges, divided by $\beta \log n$, which is $O(\frac{n}{\log n})$.

To get an integral solution, we can break the graph into components, which each must have size $\leq n/2$. The size of the components can be bounded in this way by the specification of the diameter of the expander graph, and our definition of β in the problem.

This can be reasoned by observing that no 2 terminals can be in the same component, thus each diameter must be $\leq \beta \log n$. Since every node has degree 3, this bounds the number of nodes in a component by $\leq 3^{\beta \log n} \leq n/2$.

The expander property then shows that we must cut $O(n)$ edges.

$$\text{num edges in cut} \geq \frac{1}{2} \sum_i \alpha |C_i| = \frac{\alpha}{2} n$$

Thus the optimal integral solution is $O(n)$, while we have found a feasible fractional solution of $O(\frac{n}{\log n})$. Therefore we have shown an integrality gap of $\log n$.

References

[1] The Complexity of Multiterminal Cuts Elias Dahlhaus, David S. Johnson, Christos H. Papadimitriou, P. D. Seymour, Mihalis Yannakakis. SIAM J. Comput. 23 (1994), 864-894.

[2] Expander graphs and their applications Shlomo Hoory; Nathan Linial; Avi Wigderson. Bull. Amer. Math. Soc. 43 (2006), 439-561.

[3] On the Hardness of Approximating Multicut and Sparsest-Cut. Shuchi Chawla, Robert Krauthgamer, Ravi Kumar, Yuval Rabani, D. Sivakumar. IEEE Conference on Computational Complexity 2005: 144-153

[4] Approximate max-flow min-(multi)cut theorems and their applications. Naveen Garg, Vijay V. Vazirani, Mihalis Yannakakis. STOC 1993: 698-707

CS880: Approximations Algorithms

Scribe: Matt Elder **Lecturer:** Shuchi Chawla
Topic: Sparsest Cut and Balanced Cut **Date:** 3/20/07

17.1 Multicut

First, consider the multicut problem. Given a graph $G = (V, E)$, K pairs of terminal vertices $\{s_i, t_i\}$, and a cost function on the edges $c\colon E \to \mathbb{R}$, the multicut problem asks for a minimum-cost cut of G that separates s_i and t_i for all i. Last time, we gave a $O(\log k)$ approximation for this problem.

The (relaxed) linear program for this problem is as follows; call it "Primal 1".

$$\begin{aligned} \text{minimize} \quad & \sum_{e \in E} c_e d_e \\ \text{where} \quad & d(s_i, t_i) \geq 1 \quad \forall i \\ & d \text{ is a metric} \end{aligned}$$

We can rewrite as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{e \in E} c_e d_e \\ \text{where} \quad & \mathcal{P}_i = \{\text{All paths from } s_i \text{ to } t_i\} \\ & \sum_{e \in P} d_e \geq 1 \qquad \forall i \, \forall P \in \mathcal{P}_i \\ & d_e \geq 0 \qquad \forall e \end{aligned}$$

The dual of this LP, which we'll call "Dual 1", is as follows:

$$\begin{aligned} \text{maximize} \quad & \sum_i \sum_{P \in \mathcal{P}_i} f_{i,P} \\ \text{where} \quad & \mathcal{P}_i = \{\text{All paths from } s_i \text{ to } t_i\} \\ & \sum_i \sum_{\substack{P \in \mathcal{P}_i \\ P \ni e}} f_{i,P} \leq c_e \qquad \forall e \\ & f_{i,P} \geq 0 \end{aligned}$$

Dual 1 solves the max-sum multi-commodity flow problem: c_e represents the capacity of an edge, and $f_{i,P}$ is the amount of flow directed from s_i to t_i along the path P. The LP tries to maximize the total amount of commodity flow.

Lemma 17.1.1 *Multicut is always larger than the corresponding max-sum multi-commodity flow.*

Lemma 17.1.2 *Multicut is at most $O(\log K)$ times the corresponding max-sum multi-commodity flow.*

Theorem 17.1.3 *When $k = 2$, multicut equals max-sum multi-commodity flow.*

17.2 Maximum Concurrent Multicommodity Flow

A solution to Dual 1 may starve some commodities while routing others. In contrast, max-concurrent multicommodity flow routes equal fractions of all commodities while respecting capacities. Thus, we devise the following LP for max-concurrent multicommodity flow, which we call Dual 2:

$$\begin{aligned} \text{maximize} \quad & t \\ \text{where} \quad & \sum_i \sum_{\substack{P \in \mathcal{P}_i \\ P \ni e}} f_{i,P} \leq c_e && \forall e \\ & \sum_{P \in \mathcal{P}_i} f_{i,P} \geq r_i t && \forall i \\ & f_{i,P} \geq 0 && \forall i, \forall P \in \mathcal{P} \end{aligned}$$

Intuitively, the difference between Dual 1 and Dual 2 is that Dual 1 seeks to maximize the total flow across independent commodities, while Dual 2 seeks to maximize the minimum of a set of weighted flows. This is the *maximum concurrent multicommodity flow* problem.

Primal 2, the dual of the maximum concurrent multicommodity flow problem, is as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_e d_e c_e \\ \text{where} \quad & \sum_{e \in P} d_e \geq y_i && \forall i, \forall P \in \mathcal{P}_i \\ & \sum_i r_i y_i \geq 1 \\ & d_e \geq 0 && \forall e \\ & y_i \geq 0 && \forall i \end{aligned}$$

The costs c_e are constants of the problem instance, so Primal 2 will seek to minimize the values for d_e. Thus, they will be no larger than they are constrained to be, so $y_i = d(s_i, t_i)$, the shortest distance from s_i to t_i where each edge e has length d_e. So, we can devise the following linear program, equivalent to Primal 2:

$$\begin{aligned} \text{minimize} \quad & \sum_e c_e d_e \\ \text{where} \quad & \sum_i r_i d(s_i, t_i) \geq 1 \\ & d \text{ is a metric} \end{aligned}$$

Up to scaling d, this is the same as the following program:

$$\begin{aligned} \text{minimize} \quad & \frac{\sum_e c_e d_e}{\sum_i r_i d(s_i, t_i)} \\ \text{where} \quad & d \text{ is a metric,} \end{aligned}$$

which will tie directly to the sparsest cut problem.

17.3 Sparsest Cut

Let $G = (E, V)$ be some graph. Let T, the set of terminals, be a set of pairs of vertices. For any cut S, a subset of V, let $\alpha(S)$ denote the *sparsity* of S, with

$$\alpha(S) = \frac{\left|E(S, \bar{S})\right|}{\left|(S \times \bar{S}) \cap T\right|}.$$

Thus, $\alpha(S)$ is the size of the cut S divided by the number of terminals that S separates. The *sparsest cut problem* takes G and T, and finds the cut S that minimizes $\alpha(S)$. We can generalize this further, by introducing a cost c on the edges and a weight r_i for each terminal i. Then, our more general $\alpha(S)$ looks like this:

$$\alpha(S) = \frac{c(E(S, \bar{S}))}{\sum_{\{i | s_i \in S \Leftrightarrow t_i \notin S\}} r_i}.$$

Consider the *uniform* version of sparsest cut, where we let the set of terminals be all possible pairs. That is, we let $T = V \times V$ and $r_{u,v} = 1$ for all $u \neq v$. Then, the sparsity is

$$\alpha(S) = \frac{c(E(S, \bar{S}))}{|S||\bar{S}|},$$

which is quite similar to the expansion of a set, from the context of expander graphs. Here, we give a $O(\log K \log D)$-approximation for this problem, where $D = \sum_i r_i$ and $K = |T|$. Next time, we'll find a $O(\log K)$-approximation. The best known efficient approximation is a $O(\sqrt{\log K \log\log K})$-approximation.

Suppose we solve Primal 2, yielding the metric d. Let $y_i \stackrel{\text{def}}{=} d(s_i, t_i)$. We know that $\sum_i r_i y_i \geq 1$ and that $\forall e, d_e \leq 1$. We need to round d into a cut metric — a metric with only ones and zeroes — without much increasing the sparsity.

Consider the special case where $\forall i, y_i \geq \frac{1}{2} y_{\max}$. Let $y_{\max} \stackrel{\text{def}}{=} \max_i y_i$, $d' \stackrel{\text{def}}{=} d/y_{\min}$, and $y'_i \stackrel{\text{def}}{=} d'(s_i, t_i)$ for all i. Then, $\forall i, y' \geq 1$, and d' is a feasible solution to the multicut LP. So, we can feed d' to the multicut approximation algorithm we saw last lecture. That algorithm will yield a cut of value $O(\log K) \sum_e c_e d'_e$. We deduce:

$$\begin{aligned} O(\log K) \sum_e c_e d'_e &\leq O(\log K) \frac{1}{y_{\min}} \sum_e c_e d_e \\ &\leq O(\log K) \frac{1}{y_{\max}} \sum_e c_e d_e. \end{aligned}$$

The demand this separates is thus $\sum_i r_i \geq (1/y_{\max}) \sum_i r_i y_i$. So, the sparsity of this algorithm, in this special case, is at most

$$O(\log K) \frac{\sum c_e d_e}{\sum r_i y_i}.$$

So, now consider the general case, with an arbitrarily large ratio between $y_{\max}$ and $y_{\min}$. Define I_x, the set of all y_i in a conveniently-defined interval, as:

$$I_x = \left\{ i | y_i \in \left(\frac{y_{\max}}{2^{x+1}}, \frac{y_{\max}}{2^x} \right] \right\}.$$

When x is constrained to the integers, it's clear that every y_i is contained in exactly one I_x. For each I_x, our algorithm will construct a multicut instance as in the special case, but it will scale d by $2^{x+1}/y_{\max}$ instead of $1/y_{\max}$. The sparsity for each of these instances is not too large:

$$\alpha(I_x) \leq O(\log K) \frac{\sum_e c_e d_e}{\sum_{i \in I_x} r_i y_i}.$$

If there exists a constant β and an x such that $\sum_{i \in I_x} r_i y_i \geq \beta^{-1} \sum_i r_i y_i$, then the sparsity of this instance is at least $O(\log K) \beta \sum c_e d_e / (\sum r_i y_i)$.

We claim that we can ignore all i such that $y_i < y_{\max}/D^2$. Again, D is the total demand $\sum r_i$. Define the set W containing wee values of y_i, $W = \left\{ i | y_i < y_{\max}/D^2 \right\}$. Ignoring W can result in the loss of at most $\sum_{i \in W} r_i y_i < \sum_{i \in W} r_i y_{\max}/D^2 \leq y_{\max}/D \leq 1/D$ from the denominator of our algorithm's sparsity. Assuming $D \geq 2$, ignoring W has only a small constant approximation cost. (If $D < 2$, this is an easy boundary case, which we can effeciently handle in an ad-hoc way.)

Thus, we need to consider only those I_x where $2^x < D^2$. There are at most $2 \log D$ such sets, so $\beta \geq 1/(2 \log D)$. This yields a $O(\log K \log D)$-approximation.

17.4 Balanced Cut

Given a graph $G = (V, E)$, the balanced cut problem demands the min-cost cut such that each side has at least αn nodes, for some constant value of $\alpha \leq \frac{1}{2}$. It is known that this problem is inapproximable to $n^{2-\epsilon}/\text{OPT}$ if P $\neq$ NP. This is an absurdly poor approximation.

So, we consider instead a *pseudo-approximation algorithm*, in which we approximate both the objective function of the optimal solution and the parameters of its instance. So, in this case, when asked for a cut with a balance of α, we instead output a cut with a balance α', such that $\alpha' < \alpha$ and $\alpha' \leq 1/3$. If the optimal cut of balance α has cost C_α, our cut will have cost no greater than $O(\log n) C_\alpha / (\alpha - \alpha')$. Notice that, though we have a reasonable bound on the ratio between the size of our cut and C_α, the ratio between the size of our cut and C'_α may be unbounded.

The algorithm employs a direct reduction to the sparsest cut problem, letting $T = V \times V$ and $r_i = 1$. Then, the sparsity of a cut S is, as before, $c(E(S, \bar{S}))/(|S||\bar{S}|)$. We discuss further details next time.

Even though this algorithm is far from optimal, it is actually useful. This pseudo-approximation has applications in divide-and-conquer algorithms on graphs. It ensures that we can always divide a graph into two pieces, each with size roughly linear in the size of the original graph, such that the cut between the pieces isn't too large. This yields log-depth recursion, which divide-and-conquer algorithms demand, while bounding the cost of recombining pieces.

CS880: Approximations Algorithms	
Scribe: Tom Watson	**Lecturer:** Shuchi Chawla
Topic: Balanced Cut, Sparsest Cut, and Metric Embeddings	**Date:** 3/21/2007

In the last lecture, we described an $O(\log k \log D)$-approximation algorithm for Sparsest Cut, where k is the number of terminal pairs and D is the total requirement. Today we will describe an application of Sparsest Cut to the Balanced Cut problem. We will then develop an $O(\log k)$-approximation algorithm for Sparsest Cut, due to Linial, London, and Rabinovich [4], using metric embeddings techniques of Bourgain [3].

18.1 Balanced Cut

Recall the Sparsest Cut problem.

Definition 18.1.1 (Sparsest Cut) *Given a graph $G = (V, E)$, edge capacities c_e, and requirements $r_{u,v} \geq 0$ for all $(u, v) \in V \times V$, find a set $S \subseteq V$ minimizing*

$$sparsity(S) = \frac{c(E(S, \overline{S}))}{\sum_{(u,v)\in(S\times\overline{S})\cup(\overline{S}\times S)} r_{u,v}}.$$

The sparsity of a cut is just its capacity divided by the total requirement separated by it. We denote by k the number of *terminal pairs* — pairs (u, v) such that $r_{u,v} > 0$.

We give an application of Sparsest Cut to the following problem.

Definition 18.1.2 (Balanced Cut) *Given a graph $G = (V, E)$, edge capacities c_e, and a balance requirement $\beta \leq 1/2$, find a minimum-capacity cut $(S, \overline{S})$ subject to the constraint that $(S, \overline{S})$ is β-balanced, i.e. $|S|, |\overline{S}| \geq \beta n$ where $n = |V|$.*

In the following, we use C_β to refer both to the minimum capacity of a β-balanced cut and to an optimal cut itself.

The key link between the two problems is that given an instance of Balanced Cut, we can give every pair of nodes a requirement of 1, and then for a given balance β, the capacity of a β-balanced cut is $\Theta(n^2)$ times its sparsity.

We use this link to obtain a *pseudo-approximation algorithm* for Balanced Cut. That is, for a given β and $\beta' < \beta$, we find a cut of balance β' of capacity within some guaranteed factor of C_β. (Note that C_β may be much higher than $C_{\beta'}$.) Specifically, we obtain the following result.

Theorem 18.1.3 *If there exists a ρ-approximation algorithm for Sparsest Cut, then for all $\beta \leq 1/2$ and all $\beta' < \beta$, there exists an algorithm for Balanced Cut that finds β'-balanced cut of capacity at most*

$$\frac{\rho}{(\beta - \beta')(1 - \beta + \beta')} C_\beta,$$

provided $\beta' \leq 1/3$.

Proof: We would like to use our Sparsest Cut algorithm to get information about balanced cuts in G, so we declare every pair of nodes to have requirement 1. Then the total requirement separated by a β-balanced cut is at least $\beta(1-\beta)n^2$, and so

$$sparsity(C_\beta) \leq \frac{C_\beta}{\beta(1-\beta)n^2}.$$

Thus if the hypothesized ρ-approximation algorithm for Sparsest Cut happens to return a β'-balanced cut, then its sparsity is at most

$$\frac{\rho}{\beta(1-\beta)n^2}C_\beta$$

and it separates less than n^2 requirement, so it has capacity at most

$$\frac{\rho}{\beta(1-\beta)}C_\beta$$

and we can output this cut.

If the algorithm doesn't return a β'-balanced cut, then we can repeat this process on the larger side of the cut, taking the smaller side of the cut obtained and combining it with the smaller side of our first cut. One key observation is that since the first cut isn't β'-balanced, the larger side must be a large fraction of the original graph G, and so its optimal sparsity can't be too much larger than that of G. This allows us to argue that the capacity of the final cut we obtain isn't too much larger than C_β. The other key observation is that by iterating this process on the larger side, we end up with a cut that is relatively balanced. These ideas motivate the following algorithm, which we analyze formally next.

1) Set $r_{u,v} = 1$ for all $(u, v) \in V \times V$.
2) Set $\overline{S_0} = V$, $i = 0$.
3) While $|S_1 \cup \cdots \cup S_i| < \beta' n$,
4) Increment i.
5) Apply the hypothesized Sparsest Cut algorithm to the subgraph $G_{\overline{S_{i-1}}}$ induced on $\overline{S_{i-1}}$ to obtain a cut $(S_i, \overline{S_i})$ where $|S_i| \leq |\overline{S_i}|$.
6) Output $S_1 \cup \cdots \cup S_\ell$, where ℓ is the final value of i.

We argue that the final cut has capacity at most

$$\frac{\rho}{(\beta-\beta')(1-\beta+\beta')}C_\beta.$$

The intuition is that in each iteration, if $|S_i|$ is small, then the cut $(S_i, \overline{S_i})$ does not separate much requirement, but since it has low sparsity, it must have small capacity and so it is safe to cut those edges. More formally, the amount of requirement separated by $(S_i, \overline{S_i})$ is clearly at most $|S_i| \cdot n$, so the capacity of $(S_i, \overline{S_i})$ is at most $\rho \cdot |S_i| \cdot n$ times the optimal sparsity of a cut in $G_{\overline{S_{i-1}}}$. What is that optimal sparsity? It's at most the sparsity of the cut C_β restricted to $G_{\overline{S_{i-1}}}$. Clearly, C_β

cannot have larger capacity in $G_{\overline{S_{i-1}}}$ than it does in G. Since each side of C_β contains at most $(1-\beta)n$ nodes of G, and $|\overline{S_{i-1}}| > (1-\beta')n$, it follows that each side contains at least

$$(1-\beta')n - (1-\beta)n = (\beta - \beta')n$$

nodes of $G_{\overline{S_{i-1}}}$ and thus C_β separates at least

$$(\beta - \beta')n(1 - \beta + \beta')n$$

requirement in $G_{\overline{S_{i-1}}}$. Thus the sparsity of C_β in $G_{\overline{S_{i-1}}}$ is at most

$$\frac{C_\beta}{(\beta - \beta')(1 - \beta + \beta')n^2}.$$

We conclude that the capacity of $(S_i, \overline{S_i})$ is at most

$$\rho \cdot |S_i| \cdot n \cdot \frac{C_\beta}{(\beta - \beta')(1 - \beta + \beta')n^2} = \frac{\rho}{(\beta - \beta')(1 - \beta + \beta')n} \cdot C_\beta \cdot |S_i|.$$

The capacity of the final cut $(S_1 \cup \cdots \cup S_\ell, \overline{S_\ell})$ is at most the sum of the capacities of the cuts found in all the iterations (it could be less since an edge crossing from S_i to $\overline{S_i}$ could have an endpoint in e.g. S_{i+1} and thus not cross the final cut). Furthermore, $|S_1 \cup \cdots \cup S_\ell| = |S_1| + \cdots + |S_\ell| < n$ since the S_i's are disjoint, and so the capacity of the final cut is at most

$$\sum_{i=1}^{\ell} \frac{\rho}{(\beta - \beta')(1 - \beta + \beta')n} \cdot C_\beta \cdot |S_i| < \frac{\rho}{(\beta - \beta')(1 - \beta + \beta')} \cdot C_\beta.$$

All that's left to argue is that this algorithm gives a β'-balanced cut. We know that $|S_1 \cup \cdots \cup S_\ell| \geq \beta' n$ since otherwise the algorithm wouldn't have terminated. We also know that $|S_1 \cup \cdots \cup S_{\ell-1}| < \beta' n$ and so $|\overline{S_{\ell-1}}| > (1-\beta')n$, which implies that $|\overline{S_\ell}| \geq \frac{1-\beta'}{2}n$. In order for this side of the final cut to meet the balance requirement, we just need $\frac{1-\beta'}{2} \geq \beta'$, i.e. $\beta' \leq 1/3$. ■

18.2 Sparsest Cut

18.2.1 Results

In the last lecture, we used an LP relaxation of Sparsest Cut to obtain the following result.

Theorem 18.2.1 *There is an $O(\log k \log D)$-approximation algorithm for Sparsest Cut, where $D = \sum_{u,v} r_{u,v}$.*

Today, we embark on proving the following stronger result. Our algorithm uses metric embeddings technology.

Theorem 18.2.2 *There is an $O(\log k)$-approximation algorithm for Sparsest Cut.*

We only exhibit an $O(\log n)$-approximation algorithm. With a little more work, it can be extended to obtain an $O(\log k)$-approximation.

The following result, due to Arora, Lee, and Naor [1], is the state-of-the-art for Sparsest Cut.

Theorem 18.2.3 *There is an $O(\sqrt{\log k \log\log k})$-approximation algorithm for Sparsest Cut.*

The oldest result on the Sparsest Cut problem is due Leighton and Rao [5]. They obtained an $O(\log n)$-approximation for the uniform version, where every requirement $r_{u,v} = 1$ (as we used in the reduction from Balanced Cut). The state-of-the-art for Uniform Sparsest Cut is the following result of Arora, Rao, and Vazirani [2].

Theorem 18.2.4 *There is an $O(\sqrt{\log n})$-approximation algorithm for Uniform Sparsest Cut.*

We will not explore these results in depth in this course.

18.2.2 Relaxation of Sparsest Cut

We prove Theorem 18.2.2 in two steps. Today, we argue that the Sparsest Cut problem reduces to the problem of finding a good embedding of a metric into Euclidean space under the ℓ_1 norm. In the next lecture, we will show how to find such an embedding.

We start out by recalling the Maximum Concurrent Multicommodity Flow problem.

Definition 18.2.5 (Maximum Concurrent Multicommodity Flow) *For a given graph $G = (V, E)$, edge capacities c_e, and requirements $r_{u,v} \geq 0$ for all $(u, v) \in V \times V$, simultaneously route $f \cdot r_{u,v}$ units of flow from u to v for all (u, v), for f as large as possible.*

A separate commodity is defined for each (u, v) such that $r_{u,v} > 0$, and all commodities must be routed simultaneously while satisfying the capacity constraints. The Maximum Concurrent Multicommodity Flow problem is similar to Maximum Sum Multicommodity Flow, except that the flow values for different commodities must satisfy proportionality requirements. The Maximum Concurrent Multicommodity Flow problem can be exactly captured by an LP, and the dual of this LP turns out to be the following, where $\mathcal{P}_{u,v}$ is the set of all paths from u to v.

$$\begin{array}{rll} \text{minimize} & \sum_e c_e d_e & \\ \text{subject to} & \sum_{e \in p} d_e \geq y_{u,v} & \forall (u,v) \in V \times V,\ \forall p \in \mathcal{P}_{u,v} \\ & \sum_{u,v} r_{u,v} y_{u,v} \geq 1 & \\ & d_e \geq 0 & \forall e \in E \\ & y_{u,v} \geq 0 & \forall (u,v) \in V \times V \end{array} \tag{18.2.1}$$

In what follows, we prove many equivalences of various programs. In these programs, we have a set of variables $d_{u,v}$ for $(u, v) \in V \times V$ constrained to form some sort of metric on V, and we use the notation d_e to refer to the distance between the endpoints of edge e. When we refer to equivalences (or relaxations), we do not necessarily refer to the feasible set of one program being equal to (or a subset of) the feasible set of another program, but rather the ability to take a feasible solution to one and generate a feasible solution to the other with at least as good of an objective value.

Lemma 18.2.6 *Program (18.2.1) is equivalent to the following program.*

$$\begin{array}{rl} \text{minimize} & \sum_e c_e d_e \\ \text{subject to} & \sum_{u,v} r_{u,v} d_{u,v} \geq 1 \\ & d \text{ is a metric} \end{array} \tag{18.2.2}$$

Proof: A feasible solution to Program (18.2.2) immediately yields a feasible solution to Program (18.2.1) with the same objective value by setting $y_{u,v} = d_{u,v}$. Given a feasible solution to Program (18.2.1), the $y_{u,v}$'s might not form a metric, but for $\{u, v\} \notin E$ we can change $y_{u,v}$ to be the shortest path distance between u and v under edge lengths d_e, without changing the feasibility or objective value. Setting $d_{u,v} = y_{u,v}$ then gives a feasible solution to Program (18.2.2) with no worse objective value. ■

Lemma 18.2.7 *Program (18.2.2) is a relaxation of the Sparsest Cut problem.*

Proof: Given a cut $S \subseteq V$, let $d'_{u,v} = 1$ if u and v are on opposite sides of the cut, and $d'_{u,v} = 0$ otherwise (i.e. d' is the cut metric associated with S). The total requirement separated by the cut is $\sum_{u,v} r_{u,v} d'_{u,v}$ and hence

$$sparsity(S) = \frac{\sum_e c_e d'_e}{\sum_{u,v} r_{u,v} d'_{u,v}}.$$

It follows that if we set $d_{u,v} = d'_{u,v} / \sum_{u,v} r_{u,v} d'_{u,v}$ for all u, v, then we get a feasible solution to Program (18.2.2) having objective value

$$\sum_e c_e d_e = sparsity(S).$$

■

Corollary 18.2.8 *The optimum objective value for Program (18.2.2) is a lower bound on the minimum sparsity of a cut in G.*

Corollary 18.2.9 *The sparsity of every cut in G upper bounds the value f of every concurrent multicommodity flow in G.*

Proof: This follows from Corollary 18.2.8, Lemma 18.2.6, and weak duality, but it is also easy to see directly. For every cut and every concurrent multicommodity flow, all the flow for the commodities separated by the cut must flow through the cut, and the total amount of such flow is bounded by the capacity of the cut. Since this flow would have to be proportioned according to the $r_{u,v}$'s, the value f is bounded by the sparsity of the cut. ■

Our proof of Theorem 18.2.2 is an LP-rounding algorithm — we solve relaxation (18.2.2) via LP (18.2.1) and round the solution to obtain a cut. We next describe some equivalent characterizations of Sparsest Cut that help us accomplish this.

18.2.3 Characterizations of Sparsest Cut

Recall that a cut metric is one obtained by choosing a cut $S \subseteq V$ and setting the distance between two nodes to be 1 if they are on opposite sides of the cut and 0 otherwise. In the next result we show that if in Program (18.2.2) we require d to be a scalar multiple of a cut metric (note that $d_{u,v}$

refers to the cut metric distance between u and v, *not* to the shortest path distance when the edge lengths are d_e), then we actually get an exact characterization of the Sparsest Cut problem.

Lemma 18.2.10 *Sparsest Cut is equivalent to the following program.*

$$\begin{array}{rl} \text{minimize} & \sum_e c_e d_e \\ \text{subject to} & \sum_{u,v} r_{u,v} d_{u,v} \geq 1 \\ & d \text{ is a nonnegative scalar multiple of a cut metric} \end{array} \tag{18.2.3}$$

Proof: We already argued in Lemma 18.2.7 that a cut gives a feasible solution with objective value at most the sparsity of the cut. Conversely, a feasible solution can be assumed to satisfy $\sum_{u,v} r_{u,v} d_{u,v} \geq 1$ with equality, which implies that edges crossing the corresponding cut have length $1/\sum_{u,v} r_{u,v} d_{u,v}$ and thus the cut has sparsity equal to the objective value of the feasible solution. ■

Lemma 18.2.11 *Program (18.2.3) is equivalent to the following program.*

$$\begin{array}{rl} \text{minimize} & \sum_e c_e d_e \\ \text{subject to} & \sum_{u,v} r_{u,v} d_{u,v} \geq 1 \\ & d \text{ is linear combination of cut metrics, with nonnegative coefficients} \end{array} \tag{18.2.4}$$

Proof: A feasible solution to Program (18.2.3) is trivially a feasible solution to this program. Now consider a feasible solution to Program (18.2.4), corresponding to some sets S and nonnegative real coefficients β_S, and let d_S denote the cut metric corresponding to S. The solution can be assumed to satisfy

$$\sum_{u,v} r_{u,v} d_{u,v} = \sum_S \beta_S \sum_{u,v} r_{u,v} (d_S)_{u,v} = 1,$$

and so the objective value is

$$\frac{\sum_S \beta_S \sum_e c_e (d_S)_e}{\sum_S \beta_S \sum_{u,v} r_{u,v} (d_S)_{u,v}}.$$

We leave it as an exercise to show that the smallest ratio

$$\frac{\sum_e c_e (d_S)_e}{\sum_{u,v} r_{u,v} (d_S)_{u,v}}$$

is at most that objective value. Thus, selecting the sparsest cut S^* among those used to define d and rescaling so that $\sum_{u,v} r_{u,v} (d_{S^*})_{u,v} = 1$ gives a feasible solution to Program (18.2.3) with at least as good of an objective value.

For our application, we are given the linear combination of cut metrics that comprises d. ■

Before giving our last characterization, we need to define some standard normed metrics.

Definition 18.2.12 *For a positive integer p, the ℓ_p distance between $(x_1, \ldots, x_m) \in \mathbb{R}^m$ and $(y_1, \ldots, y_m) \in \mathbb{R}^m$ is defined to be*

$$\Big(|x_1 - y_1|^p + \cdots + |x_m - y_m|^p \Big)^{1/p}.$$

The ℓ_∞ distance is defined to be

$$\max_i |x_i - y_i|.$$

A metric is said to be an ℓ_p metric if its points can be mapped to $\mathbb{R}^m$ for some m in such a way that the ℓ_p distance between each pair of points is the same as their distance in the original metric.

The ℓ_1 metric is also known as the Manhattan metric, and the ℓ_2 metric is also known as the Euclidean metric. Interestingly, the ℓ_2 metric is the only one of the above metrics where the distance between two points does not depend on the particular coordinate system used. While it is possible to solve linear programs subject to the constraint that the variables specify a metric, it is NP-hard to optimize over ℓ_p metrics for certain values of p, e.g. $p = 1$. Indeed, the following result implies that being able to optimize over ℓ_1 metrics would allow us to solve Sparsest Cut exactly.

Lemma 18.2.13 *Program (18.2.4) is equivalent to the following program.*

$$\begin{array}{ll} \text{minimize} & \sum_e c_e d_e \\ \text{subject to} & \sum_{u,v} r_{u,v} d_{u,v} \geq 1 \\ & d \text{ is an } \ell_1 \text{ metric} \end{array} \tag{18.2.5}$$

Proof: Left as a homework problem. ■

Our strategy for approximating Sparsest Cut is to solve LP (18.2.1), or equivalently Program (18.2.2), and then "round" the solution to an ℓ_1 metric. Then the characterizations given by Lemmas 18.2.10, 18.2.11, and 18.2.13 allow us to construct a sparse cut. The precise notion of rounding is what we discuss next.

18.2.4 Metric Embeddings

Definition 18.2.14 *Given metric spaces (V, μ) and (V', μ') where μ and μ' are metrics on V and V' respectively, an embedding from (V, μ) to (V', μ') is a mapping $f : V \to V'$. The embedding is said to have expansion α if for all $x, y \in V$,*

$$\mu'(f(x), f(y)) \leq \alpha \mu(x, y).$$

The embedding is said to have contraction β if for all $x, y \in V$,

$$\mu'(f(x), f(y)) \geq \frac{1}{\beta} \mu(x, y).$$

If the embedding has expansion α and contraction β, then it is said to have distortion $\alpha \cdot \beta$.

Observe that rescaling either of the metrics by a fixed factor changes the expansion and contraction by that same factor, but the distortion remains constant.

We can now formalize the connection between Sparsest Cut and metric embeddings.

Theorem 18.2.15 *If there exists an efficiently computable ρ-distortion embedding from arbitrary metrics into ℓ_1, then there exists a ρ-approximation algorithm for Sparsest Cut.*

Proof: Suppose we have a ρ-distortion ℓ_1 embedding algorithm. We can solve LP (18.2.1) and obtain a metric d on V as in Lemma 18.2.6. We know from Lemma 18.2.7 that $\sum_e c_e d_e$ is a lower

bound on the optimal sparsity of a cut in G. We apply the ρ-distortion embedding algorithm to obtain an ℓ_1 metric. By rescaling, we may assume this embedding has contraction 1 and distortion ρ. Now we have a feasible solution to Program (18.2.5) with objective value at most a factor of ρ higher than our lower bound on the optimal sparsity. Lemmas 18.2.13, 18.2.11, and 18.2.10 then allow us to construct a cut of sparsity within a factor ρ of the optimum. ■

A partial converse to Theorem 18.2.15 is known. We omit the proof, which is involved.

Theorem 18.2.16 *If the integrality gap of LP (18.2.1) is ρ, then for every metric there exists a ρ-distortion embedding into ℓ_1.*

We now just need to obtain a low-distortion ℓ_1 embedding algorithm. We will show the following result in the next lecture.

Theorem 18.2.17 *There exists an efficiently computable $O(\log n)$-distortion embedding for arbitrary metrics into ℓ_1.*

Corollary 18.2.18 *There exists an $O(\log n)$-approximation algorithm for Sparsest Cut.*

As we mentioned before, with a little more work one can improve Corollary 18.2.18 to obtain Theorem 18.2.2.

It can be shown using the expander graph example from a previous lecture that Theorem 18.2.17 is tight (up to constant factors), i.e. there exist metrics such that no ℓ_1 embedding achieves distortion $o(\log n)$.

Note that this Sparsest Cut algorithm uses an embedding where the distortion guarantee applies to *every* distance, which is more than enough to prove the approximation guarantee. It is conceivable that we can do better using an embedding that only has low distortion *on average*. It turns out that for the uniform version, where every requirement is exactly 1, this approach works. The seminal result of Leighton and Rao [5] uses a low-average-distortion embedding to achieve an $O(\log n)$ approximation for Uniform Sparsest Cut. While their result is superceded by the present result of Linial, London, and Rabinovich [4], the best known result for Uniform Sparsest Cut, due to Arora, Rao, and Vazirani [2], uses low-average-distortion embeddings and achieves an $O(\sqrt{\log n})$ approximation, which is better than the best known approximation for Sparsest Cut, which is $O(\sqrt{\log n \log\log n})$. The $O(\sqrt{\log n})$ result uses a different relaxation, optimizing over ℓ_2^2 metrics, which can be done efficiently using semi-definite programming. It then uses a low-average-distortion embedding of ℓ_2^2 metrics into ℓ_1 metrics. This embedding can be made to have low distortion for *every* pair of points at the cost of an $O(\sqrt{\log\log n})$ factor loss in the approximation factor.

In the next lecture, we will describe how to obtain an $O(\log n)$-distortion embedding of an arbitrary metric on n points into ℓ_1. We will also start discussing the use of semi-definite programming in approximation algorithm design.

References

[1] S. Arora, J. Lee, A. Naor. Euclidean Distortion and the Sparsest Cut. In *STOC*, 2005, pp. 553-562.

[2] S. Arora, S. Rao, and U. Vazirani. Expander Flows, Geometric Embeddings and Graph Partitioning. In *STOC*, 2004, pp. 222-231.

[3] J. Bourgain. On Lipschitz Embedding of Finite Metric Spaces in Hilbert Spaces. In *Israeli J. Math.*, 52, 1985, pp. 46-52.

[4] N. Linial, E. London, and Y. Rabinovich. The Geometry of Graphs and Some of Its Algorithmic Applications. In *Combinatorica*, 15, 1995, pp. 215-245.

[5] T. Leighton and S. Rao. Multicommodity Max-Flow Min-Cut Theorems and Their Use in Designing Approximation Algorithms. In *Journal of the ACM*, 46, 1990, pp. 259-271.

CS880: Approximations Algorithms

Scribe: Chi Man Liu **Lecturer:** Shuchi Chawla
Topic: Bourgain's Embedding into ℓ_1, Semi-Definite Programming **Date:** 3/22/2007

In the previous lecture, we saw how to approximate Sparsest Cut given a low distortion embedding from arbitrary metrics into ℓ_1. In the first part of this lecture, we present such an embedding with $O(\log n)$ distortion [1], and hence a $O(\log n)$-approximation for Sparsest Cut [2]. In the second part of this lecture, we introduce a generalization of linear programming known as semi-definite programming (SDP), and give an SDP relaxation for the Max-Cut problem [3].

19.1 Bourgain's Embedding into ℓ_1

Last time we showed that Sparsest Cut could be approximated by interpreting the problem as an LP over ℓ_1-metrics. In this section, we show how to embed an arbitrary metric over a set of n points into an ℓ_1 metric with $O(\log n)$ distortion [1]. An immediate result would be a $O(\log n)$-approximation for Sparsest Cut [2].

Let V be a set of n points and d be a metric over V. We want to find an embedding of d into $\mathbb{R}^k$ for some k with distortion $O(\log n)$, i.e. we want to find an embedding $f : V \to \mathbb{R}^k$ such that there exists $\alpha, \beta > 1$ with $\alpha\beta = O(\log n)$ and for all $x, y \in V$,

$$\begin{aligned} \ell_1(f(x), f(y)) &\leq \alpha \cdot d(x, y) \text{ and} \\ \ell_1(f(x), f(y)) &\geq \frac{1}{\beta} \cdot d(x, y). \end{aligned}$$

19.1.1 The Algorithm

In our construction, we use *Fréchet embeddings*, which map general metric spaces to normed metric spaces as follows. Let (V, d) be a metric space. Suppose that we want to embed this metric space into ℓ_1 over $|V|$ points in $\mathbb{R}_k$. Then, for the i^{th} coordinate ($1 \leq i \leq k$), we pick a subset $A_i \subseteq V$, and set $f_i(x) = d(x, A_i)$ for all $x \in V$, where $d(x, A_i) = \min_{y \in A_i} d(x, y)$.

We are going to map V to n points in $\mathbb{R}^{M_1 M_2}$, where $M_1 = \lceil \log_2 n \rceil$ and M_2 is a constant multiple of $\log n$ to be determined later. The algorithm is as follows.

(1) For $i = 1, \ldots, M_1$,

(2) For $j = 1, \ldots, M_2$,

(3) Form the set A_{ij} by picking every $x \in V$ with probability 2^{-i}.

(4) For all $x \in V$, set $f(x) = (f_{11}(x), f_{12}(x), \ldots, f_{M_1 M_2}(x))$, where $f_{ij}(x) = d(x, A_{ij})$.

We give an intuition for the above algorithm. Consider a particular coordinate. Arrange the points in V on the real number line according to their coordinates. For any two points, we do not want

their distance on the number line to be too large, otherwise the expansion of the embedding would be large. In fact, using the triangle inequality, we can conclude that the distance will never be too large: $|f_{ij}(x) - f_{ij}(y)| = |d(x, A_{ij}) - d(y, A_{ij})| \leq d(x, y)$. Likewise, we do not want them too close, since we want to keep the contraction small. One way to achieve this is to ensure that $d(x, A_{ij})$ is quite large and $d(y, A_{ij})$ is relatively small (or vice versa). That is, we want A_{ij} to include at least one point close to y, but no point close to x. If the set A_{ij} is too large, it may include points close to x, and if it is too small, it may not include any point close to y. This is why we use randomness in our algorithm: we hope that the overall distortion will be small by randomly picking sets of varying sizes (depending on the value of i).

19.2 Analysis

We need to show that our algorithm gives a $O(\log n)$-distortion embedding with reasonbly high probability. The following lemma bounds the expansion of f.

Lemma 19.2.1 *The expansion of f is at most $M_1 M_2$.*

Proof: Pick any $x, y \in V$ and look at one coordinate. By the definition of f and the triangle inequality, we have

$$|f_{ij}(x) - f_{ij}(y)| = |d(x, A_{ij}) - d(y, A_{ij})| \leq d(x, y).$$

Thus, the ℓ_1 distance between $f(x)$ and $f(y)$ is bounded by

$$\sum_{i=1}^{M_1} \sum_{j=1}^{M_2} |f_{ij}(x) - f_{ij}(y)| \leq M_1 M_2 \cdot d(x, y).$$

■

The following lemma bounds the contraction of f. The proof of this lemma is deferred.

Lemma 19.2.2 *The contraction of f is $O(\frac{1}{\log n})$, i.e. for all $x, y \in V$, $\ell_1(f(x), f(y)) = \Omega(\log n) \cdot d(x, y)$.*

Our main theorem follows from Lemma 19.2.1 and Lemma 19.2.2 directly.

Theorem 19.2.3 *There exists an efficiently computable $O(\log n)$-distortion embedding from (V, d) into $(\mathbb{R}^k, \ell_1)$, where $k = O(\log^2 n)$.*

Before we prove Lemma 19.2.2, we need a few definitions.

Definition 19.2.4 *For any $x \in V$ and $r \geq 0$, we denote by $B(x, r)$ the (closed) ball centered at x with radius r, i.e. $B(x, r) = \{y \in V \mid d(x, y) \leq r\}$. For any $x \in V$ and non-negative integer i, let $r_i(x)$ be the smallest r such that $|B(x, r)| \geq 2^i$.*

We now proceed to the proof of Lemma 19.2.2.

Proof: [Proof of Lemma 19.2.2] Fix $x, y \in V$. For any i, let $\rho_i = \max\{r_i(x), r_i(y)\}$. Note that $|B(x, \rho_i)| \geq 2^i$ and $|B(y, \rho_i)| \geq 2^i$. Let t be the smallest index such that $\rho_t \geq \frac{1}{2} d(x, y)$. Consider $\rho_0, \rho_1, \ldots, \rho_t$. If $\rho_t + \rho_{t-1} > d(x, y)$, we redefine $\rho_t = d(x, y) - \rho_{t-1}$. By doing this, we ensure that $B(x, \rho_{i-1})$ and $B(y, \rho_i)$ are disjoint (but they can touch) for $i = 1, \ldots, t$. We want to show that

for any i and j, there is a good chance that A_{ij} contains a point near x but no point near y. This effectively bounds the contraction of f between x and y. This is formalized in the following claim.

Claim 19.2.5 *Let $1 \leq i \leq t$. Let $S_i = \{j \mid |f_{ij}(x) - f_{ij}(y)| \geq \rho_i - \rho_{i-1}\}$. Then there exists a constant $c > 0$ such that $\mathbf{Pr}[|S_i| \geq c \log n] \geq 1 - n^{-3}$.*

Proof: [Proof of Claim 19.2.5] Without loss of generality, suppose that $\rho_i = r_i(y)$. Let the i^{th} *good ball* G_i be $B(x, \rho_{i-1})$, and the i^{th} *bad ball* B_i be the **open** ball centered at y with radius ρ_i, i.e. $B_i = \{v \in V \mid d(y, v) < \rho_i\}$. Note that $|G_i| \geq 2^{i-1}$ and $|B_i| \leq 2^{i-1}$ (since B_i is open).

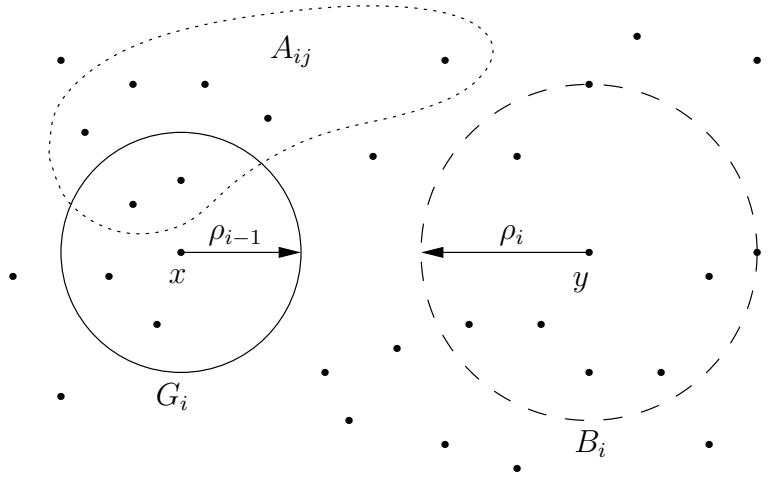


Figure 19.2.1: The i^{th} good ball G_i, the i^{th} bad ball B_i, and the set A_{ij}.

Fix j. We bound the probability that some point in A_{ij} lies in G_i but none lies in B_i as follows.

$$\begin{aligned}
& \mathbf{Pr}[A_{ij} \cap G_i \neq \emptyset \text{ and } A_{ij} \cap B_i = \emptyset] \\
= \; & \mathbf{Pr}[A_{ij} \cap G_i \neq \emptyset] \cdot \mathbf{Pr}[A_{ij} \cap B_i = \emptyset] && \text{(by independence, because } G_i \text{ and } B_i \text{ are disjoint)} \\
\geq \; & \left(1 - \left(1 - \frac{1}{2^i}\right)^{2^i/2}\right)\left(1 - \frac{1}{2^i}\right)^{2^i} \\
\geq \; & (1 - e^{-1/2}) \cdot \frac{1}{4}.
\end{aligned}$$

Thus, the above probability is at least some constant. Note that $A_{ij} \cap G_i \neq \emptyset$ implies that $f_{ij}(x) \leq \rho_{i-1}$, and $A_{ij} \cap B_i = \emptyset$ implies that $f_{ij}(y) \geq \rho_i$. Hence, it follows from the above that $\mathbf{Pr}[|f_{ij}(x) - f_{ij}(y)| \geq \rho_i - \rho_{i-1}]$ is at least some constant.

For each j, let Z_j be an indicator random variable taking the value 1 if and only if $|f_{ij}(x) - f_{ij}(y)| \geq \rho_i - \rho_{i-1}$. Then by linearity of expectation, we have $\mathbf{E}\left[\sum_{j=1}^{M_2} Z_j\right] \geq c' \cdot M_2$ for some constant $c' > 0$. By choosing M_2 (recall that M_2 is a constant times $\log n$ and we have not picked the constant yet)

and a suitable constant $c'' > 0$ and applying the Chernoff bound[1], we have

$$\mathbf{Pr}\left[\sum_{j=1}^{M_2} Z_j < c \cdot M_2\right] < e^{-\frac{c'' \cdot M_2}{3}} = \frac{1}{n^3},$$

where c depends on c', c'' and M_2. Recall that $M_2 = \Theta(\log n)$ and $\sum_{j=1}^{M_2} Z_j = |S_i|$. This proves our claim. ∎

Let c be the constant in Claim 19.2.5. For every i, we call it *good* if $|S_i| \geq c \cdot \log n$, and *bad* otherwise. If all i's are good, then

$$\begin{aligned}\sum_{i=1}^{t}\sum_{j=1}^{M_2} |f_{ij}(x) - f_{ij}(y)| &\geq \sum_{i=1}^{t} \Omega(\log n)(\rho_i - \rho_{i-1}) \\ &\geq \Omega(\log n)\rho_t \\ &\geq \Omega(\log n) d(x,y).\end{aligned}$$

Thus, the contraction of f is $O(\frac{1}{\log n})$. It remains to show the probability that all i's are good is not too small. By union bound, we have

$$\mathbf{Pr}[\text{there exists a bad } i] \leq \frac{1}{n^3}\log n < \frac{1}{n^2}.$$

Hence, for fixed $x, y \in V$, all i's are good with probability at least $1 - n^{-2}$. We say that a pair (x, y) *fails* if not all i's are good. By union bound, we get

$$\mathbf{Pr}[\text{there exists a pair } (x,y) \text{ which fails}] \leq \frac{1}{2}.$$

We can repeat the algorithm until the contraction of f is $O(\frac{1}{\log n})$. The expected number of times we need to run the algorithm is constant. ∎

Note. It can be shown that f is a $O(\log n)$-distortion embedding into ℓ_p for general p using a similar analysis. Also, there are metrics which cannot be embedded into ℓ_1 with distortion $o(\log n)$, e.g. expander graph metrics.

19.3 Semi-Definite Programming

One downside of linear programming is that it cannot capture nonlinear constraints. In this section, we introduce semi-definite programming, which is capable of capturing a specific form of nonlinear constraints that turns out to be useful in some formulations.

19.3.1 Definitions

A *semi-definite program* (SDP) can be thought of as a linear program with an additional "semi-definiteness" constraint. More specifically, an SDP is a mathematical program with the following elements:

[1]The Chernoff bound we use here is $\mathbf{Pr}[\sum Z_j < (1-\epsilon)\mathbf{E}[\sum Z_j]] < \exp(-\epsilon^2\mathbf{E}[\sum Z_j]/3)$.

- a set of variables;
- a linear objective function to be minimized or maximized;
- a set of linear constraints;
- a special constraint saying that some matrix of variables is positive semi-definite (see below).

The special constraint is the only thing that is not seen in linear programs. This constraint has the form "A has to be positive semi-definite, where A is a square matrix whose entries are taken from the variables in the program". In the following, we define positive semi-definite matrices and give some related results in linear algebra.

Definition 19.3.1 (Positive semi-definite matrix) *An $n \times n$ matrix A is* positive semi-definite *(denoted as $A \succeq 0$) if and only if*

1. *A is real symmetric; and*
2. *for all $x \in \mathbb{R}^n$, $x^T A x = \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij} x_i x_j \geq 0$.*

The following proposition implies that convex combinations of feasible solutions to an SDP are still feasible.

Proposition 19.3.2 *Let A, B be positive semi-definite matrices of the same size. Then*

1. *$A + B \succeq 0$;*
2. *$\lambda A \succeq 0$ for all real $\lambda \geq 0$.*

The following proposition gives different characterizations of positive semi-definite matrices.

Proposition 19.3.3 *Let A be an $n \times n$ matrix. The following statements are equivalent:*

1. *$A \succeq 0$.*
2. *All eigenvalues of A are real and non-negative.*
3. *There exists a matrix $C \in \mathbb{R}^{m \times n}$ ($m \leq n$) such that $A = C^T C$.*

Given a positive semi-definite matrix A, the matrix C in the third characterization can be computed in polynomial time using an algorithm called *Cholesky decomposition*. Note that a positive semi-definite matrix can be viewed as a *matrix of dot products*. If we think of $C = [C_1 \cdots C_n]$ as a bunch of column vectors in $\mathbb{R}^m$, the $(i,j)^{\text{th}}$ entry of A is the dot product $C_i \cdot C_j$. This leads us to viewing SDP as vector programming. A *vector program* is a mathematical program with the following elements:

- a set of variables v_i's in $\mathbb{R}^n$, for some $n > 0$;
- an objective function linear in all $(v_i \cdot v_j)$'s, to be minimized or maximized;
- a set of linear constraints over all $(v_i \cdot v_j)$'s

We will see some vector program formulations in later lectures.

19.3.2 Solving SDPs

Similar to an LP, the feasible region (polytope) of an SDP is convex, except that one of its faces (the one corresponding to the nonlinear constraint) might not be "flat". As a result, the optimal solution to an SDP could be in the interior of a face of the feasible region and could be irrational. Hence, having a polynomial-time algorithm for solving SDPs exactly is impossible. However, we can achieve a $(1+\epsilon)$-approximation in time $poly(n, \log \frac{1}{\epsilon})$, where n is the input size. Typical algorithms for solving SDPs include interior point methods and the ellipsoid method. Recall that the ellipsoid method starts by enclosing the whole feasible region in a large ellipsoid. It then checks if the center of the ellipsoid lies in the feasible region. If not, it picks a violated constraint and computes the intersection of the ellipsoid with the hyperplane corresponding to that constraint. A new, smaller ellipsoid is then used to enclose the intersection. This process is repeated until we have found a point lying in the feasible region, or the ellipsoid has become so small that we can conclude the feasible region to be empty. One nice thing about the ellipsoid method is that even if the SDP has a super-polynomial number of constraints, it still runs in polynomial time, provided that there exists an efficient algorithm for testing feasibility of a given solution, and finding a violated constraint if the solution is infeasible. Such an algorithm is known as a *separation oracle* for the SDP.

19.3.3 Example: Max-Cut

Recall the (Unweighted) Max-Cut problem.

Definition 19.3.4 (Max-Cut) *Given an undirected graph $G = (V, E)$, find a cut in G with maximum cut value, i.e. find a subset $S \subseteq V$ that maximizes*

$$c(S) = \left|\{(v, v') \mid v \in S, v' \in V \backslash S, (v, v') \in E\}\right|.$$

We are going to formulate Max-Cut as an SDP. For each vertex $v \in V$, we have a variable x_v that takes a value in $\{1, -1\}$ depending on the partition to which v is assigned. It is clear that the value of the cut is $\sum_{(u,v)\in E} \frac{|x_u - x_v|}{2}$. We are going to maximize the value. Thus, we have the following nonlinear program.

$$\begin{array}{lll} \text{maximize} & \sum_{(u,v)\in E} \frac{|x_u - x_v|}{2} & \\ \text{subject to} & x_v \in \{-1, 1\} & \forall v \in V \end{array}$$

Note that the objective function in not linear in the x_v's. We can convert the above program to the following equivalent quadratic integer program.

$$\begin{array}{lll} \text{maximize} & \sum_{(u,v)\in E} \frac{(x_u - x_v)^2}{4} & \\ \text{subject to} & x_v^2 = 1 & \forall v \in V \end{array}$$

Solving quadratic programs is NP-hard in general. By introducing new variables, the objective function can be converted to a linear one. Specifically, we have a new variable y_{uv} for each pair $(u, v) \in V \times V$. We add (nonlinear) constraints to enforce the equality $y_{uv} = x_u x_v$ for every u and v. By observing that $(x_u - x_v)^2 = x_u^2 + x_v^2 - 2x_u x_v = 2 - 2y_{uv}$, we obtain the following program.

$$\begin{array}{lll} \text{maximize} & \sum_{(u,v)\in E} \frac{1 - y_{uv}}{2} & \\ \text{subject to} & y_{uv} = x_u x_v & \forall (u, v) \in V \times V \\ & y_{vv} = 1 & \forall v \in V \end{array}$$

Let $Y = [y_{uv}]_{u,v \in V}$ be a matrix of variables. Then $Y = X^T X$ where $X = [x_v]_{v \in V}$ is a row vector containing the variables x_v's. We rewrite the above program in matrix form.

$$\begin{array}{rll} \text{maximize} & \sum_{(u,v) \in E} \frac{1 - y_{uv}}{2} & \\ \text{subject to} & y_{vv} = 1 & \forall v \in V \\ & Y = X^T X & \\ & X \text{ is a row vector} & \end{array}$$

The above program can be relaxed into an SDP. By Proposition 19.3.3, $Y \succeq 0$. The resulting SDP relaxation is as follows. Note that x_v's no longer appear in the program.

$$\begin{array}{rll} \text{maximize} & \sum_{(u,v) \in E} \frac{1 - y_{uv}}{2} & \\ \text{subject to} & y_{vv} = 1 & \forall v \in V \\ & Y \succeq 0 & \end{array}$$

Since the SDP is a relaxation of the original program, its optimal value must be at least the optimal cut value. To get an approximation, we still need to show how to convert the SDP solution to a cut whose value is not much smaller than the optimal value of the SDP. Let Y^* be an optimal solution to the above SDP. Using Cholesky decomposition, we can find in polynomial time a matrix X^* such that $Y^* = X^{*T} X^*$. Suppose that X^* has m rows, where $m \leq n$. Then each vertex v is represented by a vector in $\mathbb{R}^m$. Note that these vectors are unit vectors because $y_{vv} = 1$ for all v, and hence they all lie on the unit sphere in $\mathbb{R}^m$. In order to obtain a cut, we need separate these vectors into two sets. Let (u, v) be an edge in G. Let X_u^* and X_v^* be a unit vectors corresponding to u and v. If X_u^* and X_v^* are far apart, their dot product $X_u^* \cdot X_v^* = y_{uv}$ is small, and so the value $\frac{1 - y_{uv}}{2}$ is large. Thus, to maximize $\sum_{(u,v) \in E} \frac{1 - y_{uv}}{2}$, our objective is to separate the vectors into two sets such that "long" edges get cut. If we pick a random hyperplane through the origin, the probability that an edge gets cut is roughly proportional to its length. This randomized approach seems to be a nice and easy way to meet our objective. We will continue our discussion next time. The algorithm is due to Goemans and Williamson [3].

References

[1] J. Bourgain. On Lipschitz embedding of finite metric spaces in Hilbert space. *Israel J. Math.*, 52(1–2), pp. 46–52, 1985.

[2] N. Linial, E. London and Y .Rabinovich. The geometry of graphs and some of its algorithmic applications. *Combinatorica*, 15, pp. 215–245, 1995.

[3] M.X. Goemans and D.P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM (JACM)*, 42(6), pp. 1115–1145, 1995.

CS880: Approximations Algorithms

Scribe: Siddharth Barman **Lecturer:** Shuchi Chawla
Topic: SDP: Max-cut, Max-2-SAT **Date:** 03/27/07

In this lecture we give SDP (semi definite programming) based algorithms for the Max-cut and Max-2-SAT problem.

A semi definite program is a set of linear inequalities along with the constraint that the matrix of variables is positive semi definite. The general form of a SDP is as follows:

$$\begin{aligned} & \min AC \\ \text{subject to} \quad & A \cdot B_k \geq b_k \quad \forall k \\ & A \succeq 0 \end{aligned}$$

Where $A \succeq 0$ indicates that the variable matrix A is positive semi definite. Also note that the first constraint, $A \cdot B_k \geq b_k$, is a constraint over the linear combination of elements A_{ij} of matrix A i.e. $\sum_{ij} A_{ij}(B_k)_{ij} \geq b_k$.

An important observation is that a semi definite program is equivalent to a *vector program* where matrix A is composed of inner products of variable vectors. That is the above mentioned SDP is equivalent to the following vector program

$$\begin{aligned} & \min \text{linear funct. of } \langle v_i, v_j \rangle \\ \text{subject to} \quad & \text{linear constraints } over \langle v_i, v_j \rangle \\ & v_i \in \Re^n \text{ and} \\ & A_{ij} = v_i \cdot v_j \end{aligned}$$

We shall formulate vector programs for the Max-cut and Max-2-SAT problem, which essentially are semi definite programs, the general strategy then is to solve the SDP to obtain vector solutions for the variables. We then "extract" an integral solution from these vectors which preserve the objective function value to a good extent.

20.1 Max-cut

Problem Statement: Given a graph G with weights c_e on edges, find a cut $C \subseteq E$ of maximum weight which separates the vertices of G in two connected components. Here the weight of the cut C is defined as $\sum_{e \in C} c_e$.

First we mention that the direct linear programming based approach that we have applied for other cut problems does not go through for Max-cut. An LP relaxation for Max-cut is as follows:

$$\begin{aligned} & \max \sum c_e d_e \\ \text{subject to} \quad & \text{d is a metric} \\ & d(u,v) \leq 1 \quad \forall u, v \end{aligned}$$

But this program can be maximized by setting $d(u,v) = 1\ \forall u \neq v$ (which is a legitimate metric); resulting in objective function value equal to $\sum_{e \in E} c_e$. But this implies that all the edges are in the cut and the requirement that the graph is separated into only two components S and $V \setminus S$ is not met.

The important insight here is to restrict the form of the metric d to ensure that a sound cut is obtained. Specifically we wish to determine a metric which places the vertices of the graph either at 1 or at -1. That is for all vertices v we must have $x_v^2 = 1$. Thus d_e is either 0 or 2 and the objective function becomes $\max \sum \frac{c_e d_e}{2}$.

For the above mentioned metric we have $d_{uv}^2 = (x_u - x_v)^2 = x_u^2 + x_v^2 - 2x_u x_v$. But x_u^2 and x_v^2 are equal to 1 hence $d_{uv}^2 = 2(1 - x_u x_v)$. Also note that $d_{uv} \in \{0, 2\}$ thus $d_{uv} = d_{uv}^2/2$. Hence the Max-cut problem can be concisely stated as the following *quadratic program*:

$$\begin{aligned} & \max \textstyle\sum_{(u,v) \in E} \frac{c_e}{2}(1 - x_u x_v) \\ \text{s.t.} \quad & x_u^2 = 1\ \forall u \\ & x_u \in \Re \end{aligned}$$

We relax this to a vector program. In particular,

$$\begin{aligned} & \max \textstyle\sum_{(u,v) \in E} \frac{c_e}{2}(1 - x_u \cdot x_v) \\ \text{s.t.} \quad & x_u \cdot x_u = 1\ \forall u \\ & x_u \in \Re^n \end{aligned} \tag{20.1.1}$$

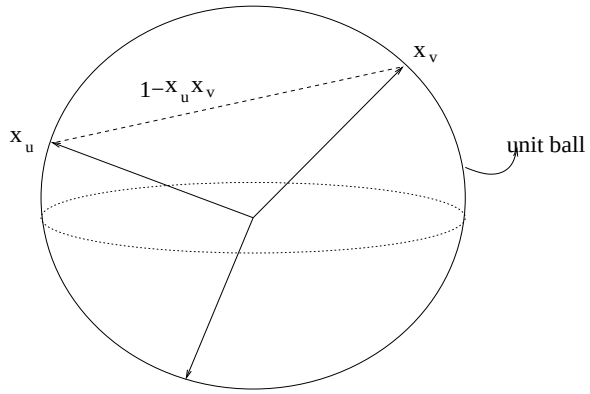


Figure 20.1.1: Unit ball in $\Re^n$

After solving this relaxed formulation we get vectors x_u on the unit ball (see Figure 20.1.1) and we wish to determine a cut in $\Re^n$ such that long edges are very likely in it. The idea is to consider a random hyperplane in $\Re^n$ passing through the origin; long edges which contribute more to the SDP solution have high probability of being cut by such a plane. Our cut is the set of edges that cross this hyperplane. Next we prove that this in fact achieves an approximation factor of 0.878.

Formally the Max-cut algorithm is as follows

1. Solve SDP 20.1.1 to obtain vectors $\{x_u\}$
2. Pick a random unit vector α in $\Re^n$
3. Output $\{v \mid x_v^T \alpha > 0\}$

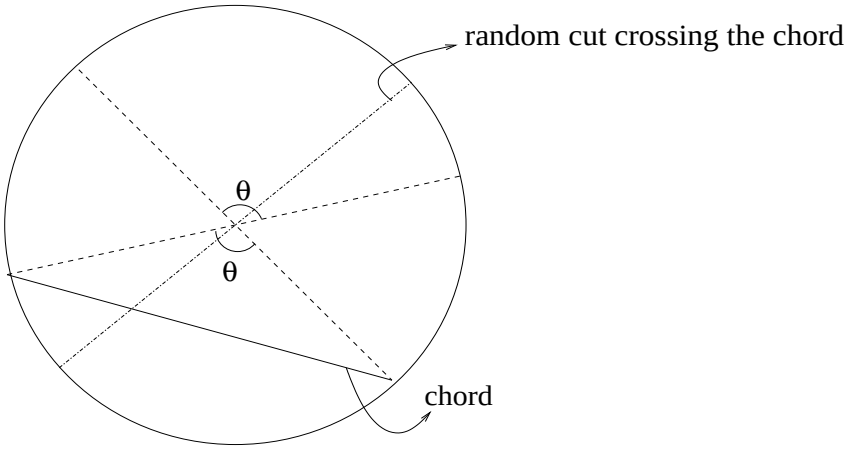


Figure 20.1.2: Chord with random cut

First we consider vectors in $\Re^2$ and then generalize to cuts in $\Re^n$. Consider a chord in a unit circle that subtends an angle of θ at the origin (see Figure 20.1.2). Note that in the present context selecting an angle between 0 and 2π, uniformly at random is equivalent to selecting a cut uniformly at random.

The probability that the chord (representing an edge) is cut is θ/π. As any angle selected in the marked region of Figure 20.1.2 would define a cut that intersects the chord and the total angle contained by the marked region is 2θ the probability is $2\theta/2\pi = \theta/\pi$.

As vectors x_u lie on the unit ball the square of the length of the chord/edge is $2(1-\cos\theta)$. Hence its contribution to SDP is $2c_e(1-\cos\theta)$. The expected contribution of the edge to our solution is $\frac{\theta}{\pi}c_e$. The maximal difference between the two could be $\gamma = \max_\theta \frac{\pi(1-\cos\theta)}{2\theta} = \frac{1}{0.878} \approx 1.12$. That is the contribution of any edge to the SDP can not be more than γ times its expected contribution to our solution.

The following lemma ensures that the above mentioned Max-cut algorithm achieves a 0.878 approximation.

Lemma 20.1.1 *For every edge $e = (u, v)$ in the graph, let θ_e be the angle between x_u and x_v, then the probability that we cut edge e is θ_e/π.*

Proof: Consider the plane defined by vectors x_u, x_v and the origin. The intersection of the hyperplane defined by random vector α and this plane is a line passing through the origin. Say this line is α'. Note that α' is picked from a spherically symmetric distribution hence as above the probability that (u, v) is cut is θ_e/π. ■

Note that step 2 of the algorithm is equivalent to picking a point uniformly at random from the surface of a unit sphere. This is accomplished by picking a point from an n dimensional Gaussian and then renormalizing it to be of unit length.

Feige and Schechtman [2] have shown that the integrality gap of this SDP is in fact 1/0.878. Also hardness of 0.878 for Max-cut has been shown under the unique games conjecture [3].

Integral solutions of the Max-cut problem satisfy an interesting property. In particular any integral solution sets $x_u \in \{-1, 1\}$ hence it must satisfy $d_{uv}^2 = 2d_{uv}$. This relation implies a triangle inequality for distance squares, that is $d_{uv}^2 \leq d_{uw}^2 + d_{wv}^2$. Such a relation is not necessary for all metrics but it holds for integral solutions. With this constraint we can formulate another stronger SDP for Max-cut:

$$\begin{aligned} & \max \textstyle\sum_{(u,v)\in E} \frac{c_e}{2}(1 - x_u x_v) \\ \text{s.t.} \quad & x_u^T x_u = 1 \ \forall u \\ & x_u \in \Re^n \\ & 1 - x_u x_v \leq 1 - x_u x_w + 1 - x_w x_v \ \ \forall u, v, w \end{aligned} \tag{20.1.2}$$

The last constraint can be simplified to $x_u x_w + x_w x_v \leq 1 + x_u x_v$. The solutions to such an SDP are special metrics called l_2^2 metrics:

Definition 20.1.2 *Metrics for which not only the distances but the square root of distances satisfy the triangle inequality are called* ***squared euclidian metrics*** *(negative type) and are denoted as* l_2^2.

Note that l_2^2 metric embed with distortion $O(\sqrt{\log n} \log \log n)$ in l_1 [1], which implies better approximation factor for some problems with such metric requirement. Though for Max-cut formulation 20.1.2 does not reduce the integrality gap but it does for some other problems, in particular for sparsest cut integrality gap is reduced to $\tilde{O}(\sqrt{\log n})$ [1].

20.2 Max-2-SAT

Problem Statement: Given a 2-CNF boolean formula we need to determine an assignment which maximizes the number of satisfied clauses. In the weighted version of the problem each clause is associated with a weight v_i and we need to determine an assignment which maximizes the sum of the values v_i for the satisfied clauses.

We formulate a SDP by considering vector v_x for every variable x, also $v_{\overline{x}}$ (for $\overline{x}$) is set to $-v_x$. For reference a true vector v_T is also defined, thus informally $v_x v_T$ is the extent to which x is true.

Now, clause $x \vee y$ contributes a value of 1 to the objective function if either x or y is true else it contributes a value of 0. For $v_x \in \{-1, 1\}$ this can be equivalently expressed as $\frac{3+v_x v_T + v_y v_T - v_x v_y}{4}$. As the expression is set to one when either $v_x v_T$ or $v_y v_T$ is set to one, and it is set to zero when v_x and v_T are of different signs. We relax v_x to vectors in $\Re^n$ to obtain the SDP:

$$\begin{aligned} & \max \textstyle\sum_{clauses \quad x \vee y} \frac{3+v_x v_T+v_y v_T-v_x v_y}{4} \\ \text{s.t.} \quad & v_x v_x = 1 \ \forall x \\ & v_x \in \Re^n \\ & v_T \text{ is the truth vector} \end{aligned} \tag{20.2.3}$$

The algorithm $\mathcal{A}$, for Max-2-SAT is as follows:

1. Pick a random hyperplane
2. Set everything on the side of v_T to true and rest to false

The analysis of Max-2-SAT follows closely from Max-cut. The objective function in this case can be rewritten as $\frac{1}{4}\sum((1+v_x v_T)+(1+v_y v_T)+(1-v_x v_y))$, thus the expression consists of terms of the form $1+v_i v_j$ and $1-v_i v_j$. Next we analyze the two terms separately.

As in the case of Max-cut the probability that the edge between i and j is cut is θ/π. The SDP contribution of the term is directly $1-v_i v_j$. As before with probability θ/π the plane separates the two vectors in which case the contribution of the term is 2 else the contribution is 0. Hence the expected contribution is $\frac{2\theta}{\pi}+0$. As before the ratio of the expected contribution to the SDP contribution is no less than $\gamma = \min \frac{2\theta}{\pi(1-\cos\theta)} = 0.878$.

Note that $v_i v_j = \cos\theta$, where θ is the angle between v_i and v_j. Hence for the term $1+v_i v_j$ the SDP contribution is $1+\cos\theta$. The expected contribution to the integral solution on the other hand is $0 \times \frac{\theta}{\pi} + 2\left(1-\frac{\theta}{\pi}\right)$. Hence the ratio in this case is $\frac{2(\pi-\theta)}{\pi(1+\cos\theta)}$. By setting $\theta' = \pi - \theta$ the ratio reduces to $\frac{2\theta'}{\pi(1-\cos\theta')}$, as before this is no less than 0.878. Hence the total expected contribution is no less than 0.878. So we have the following theorem:

Theorem 20.2.1 *The above mentioned algorithm $\mathcal{A}$ achieves an approximation factor of* 0.878 *for Max-2-SAT.*

References

[1] S. Arora, J.R. Lee, A. Naor Euclidean distortion and the sparsest cut. *In Proceedings on 37th Annual ACM Symposium on Theory of Computing (STOC)* (2005), pp: 553-562.

[2] U. Feige, G. Schechtman: On the integrality ratio of semidefinite relaxations of MAX CUT. *In Proceedings on 33rd Annual ACM Symposium on Theory of Computing (STOC)* (2001), pp: 433-442.

[3] S. Khot, G. Kindler, E. Mossel, R. O'Donnell Optimal Inapproximability Results for Max-Cut and Other 2-Variable CSPs? *In 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS)* (2004), pp: 146-154.

CS880: Approximations Algorithms

Scribe: Chi Man Liu **Lecturer:** Shuchi Chawla
Topic: Semi-Definite Programming: Graph Coloring **Date:** 3/29/2007

In the previous lecture , we saw how semi-definite programming (SDP) could be employed to approximate Max-Cut and Max-2-SAT. In this lecture, we look at another problem which can be approximated using SDP: Graph Coloring.

21.1 Graph Coloring: Overview

Let $G = (V, E)$ be a graph. A *k-coloring* for G is a function $f : V \to [k]$ such that $f(u) \neq f(v)$ for all $(u, v) \in E$. In other words, a k-coloring is an assignment of vertices to k colors such that no edge is monochromatic. We say that a graph G is *k-colorable* if there exists a k-coloring for G. The *chromatic number* of G is the least k such that G is k-colorable. Given a k-colorable graph G, finding a k-coloring for G is solvable in polynomial time for $k = 2$, but NP-hard for $k \geq 3$.

In Homework 1, we showed how to find a $O(\sqrt{n})$-coloring for any 3-colorable graph in polynomial time, where n is the number of vertices in the graph. That algorithm is due to Wigderson [1] and can be extended to find $O(n^{2/3})$-colorings for $k = 4$, and $O(kn^{1-1/(k-1)})$-colorings for general k. Blum [2] gave a combinatorial algorithm for coloring 3-colorable graphs using $\tilde{O}(n^{3/8})$ colors[1]. Karger, Motwani and Sudan [3] presented a semi-definite programming based $\tilde{O}(n^{1/4})$-coloring for 3-colorable graphs. Combining the techniques in [2] and [3], Blum and Karger [4] improved the bound to $\tilde{O}(n^{3/14})$. The best known approximation for 3-coloring uses $O(n^{0.2111})$ colors. This algorithm, due to Arora, Chlamtac and Charikar [5], is based on semi-definite programming and the triangle inequality.

On the hardness of Graph Coloring, it is known that coloring a 3-colorable graph using only 4 colors is NP-hard. In fact, the Unique Games Conjecture implies that there is no constant factor approximation for 3-coloring. It is also known that approximating k-coloring better than a factor of $n^{1-\epsilon}$ for arbitrary k is NP-hard.

21.2 SDP Formulation

We need to divide the vertices into k sets such that every edge gets cut. This is different from the Max-Cut and Max-2-SAT problems for which only two sets are sufficient. Our strategy is to map the vertices to unit vectors in $\mathbb{R}^n$, and we want to maximize distances between the edges. Finally, a randomized procedure is used to partition the vertices into k sets according to the optimal solution vectors.

We introduce a vector variable $v_i \in \mathbb{R}^n$ for each $i \in V$. Recall that the dot product between two vectors increases as they get closer to each other. Therefore, to maximize distances between the edges, we want to minimize the quantity $\max_{(i,j)\in E} v_i \cdot v_j$. We formulate Graph Coloring as the

[1]The notation $\tilde{O}$ is sometimes used to hide low-order multiplicative terms, such as $\log n$.

following SDP (equivalently, vector program).

$$\begin{array}{rll} \text{minimize} & t & \\ \text{subject to} & v_i \cdot v_j \leq t & \forall (i,j) \in E \\ & v_i \cdot v_i = 1 & \forall i \in V \\ & v_i \in \mathbb{R}^n & \forall i \in V \end{array}$$

Unfortunately, the relation between an feasible solution to the SDP and a legal coloring for the graph is not obvious — given a feasible solution to the SDP, how to construct a legal k-coloring? Furthermore, how does k depend on t? The following two lemmas answer the latter question.

Lemma 21.2.1 *Let t^* be the optimal value of the SDP. If G is k-colorable, then $t^* \leq \frac{-1}{k-1}$.*

Lemma 21.2.2 *Let t^* be the optimal value of the SDP. If G contains a k-clique, then $t^* \geq \frac{-1}{k-1}$.*

Remark. We define $\theta(G) = 1 - \frac{1}{t^*}$ for any graph G. This function is known as the *Lovász theta function*. Note that if $t^* = \frac{-1}{k-1}$, then $\theta(G) = k$. In this case, the graph is called vector-k-colorable. For a graph G, the *clique number* of G is the size of the largest clique in G. By Lemma 21.2.1, $\theta(G)$ is at most the chromatic number of G. By Lemma 21.2.2, $\theta(G)$ is at least the clique number of G. A graph with its chromatic number equal to its clique number is known as a *perfect graph*. For example, complete graphs are perfect graphs. Note that for perfect graphs, their chromatic numbers and clique numbers can be computed in polynomial time, namely by computing the Lovász theta function.

Proof: [Proof of Lemma 21.2.1] Suppose that we have a partition of V into k subsets. We will define explicitly k unit vectors $v_1, \ldots, v_k$ in $\mathbb{R}^k$ such that $v_i \cdot v_j \leq \frac{-1}{k-1}$ for any $i \neq j$. Assigning vertices in different subsets to different vectors, we have found a feasible solution to the SDP with value at most $\frac{-1}{k-1}$, thus proving the lemma. We demonstrate how to find such vectors by induction on k.

The base case ($k = 2$) is trivial. Assume that for some $k \geq 2$ we have unit vectors $v'_1, \ldots, v'_k \in \mathbb{R}^k$ such that $v'_i \cdot v'_j \leq \frac{-1}{k-1}$ for $i \neq j$. We construct $k+1$ unit vectors $v_1, \ldots, v_{k+1} \in \mathbb{R}^{k+1}$ as follows.

- $v_{k+1} = (0, \ldots, 0, 1)$;
- for $1 \leq i \leq k$, $v_i = (\alpha v'_i, -1/k)$, i.e. the $k+1$-vector obtained by adding one coordinate $(-1/k)$ to the k-vector $\alpha v'_i$, where $\alpha = \sqrt{1 - 1/k^2}$.

There are a few things to show. First, the v_i's are unit vectors:

$$v_i \cdot v_i = \alpha^2 (v'_i \cdot v'_i) + \frac{1}{k^2} = 1.$$

Second, $v_i \cdot v_j \leq \frac{-1}{k}$ for $1 \leq i, j \leq k$:

$$\begin{aligned} v_i \cdot v_j &= \alpha^2 (v'_i \cdot v'_j) + \frac{1}{k^2} \\ &\leq \left(1 - \frac{1}{k^2}\right)\left(\frac{-1}{k-1}\right) + \frac{1}{k^2} \\ &= -\frac{k+1}{k^2} + \frac{1}{k^2} \\ &= \frac{-1}{k}. \end{aligned}$$

Third, $v_i \cdot v_{k+1} = \frac{-1}{k}$ for $1 \leq i \leq k$, which is obvious. ■

Proof: [Proof of Lemma 21.2.2] Let $v_1, \ldots, v_k \in \mathbb{R}^k$ be k unit vectors. Consider the dot product of $\sum_{i=1}^{k} v_i$ with itself:

$$\begin{aligned} \left\langle \sum_{i=1}^{k} v_i \, , \sum_{i=1}^{k} v_i \right\rangle &= \sum_{i=1}^{k} (v_i \cdot v_i) + \sum_{\substack{i \neq j \\ 1 \leq i,j \leq k}} (v_i \cdot v_j) \\ &= k + k(k-1)\overline{r}, \end{aligned}$$

where $\overline{r}$ is the average dot product. Noticing that the above dot product is non-negative, we get

$$\begin{aligned} k + k(k-1)\overline{r} &\geq 0 \\ \overline{r} &\geq \frac{-1}{k-1} \\ \max_{\substack{i \neq j \\ 1 \leq i,j \leq k}} (v_i \cdot v_j) &\geq \frac{-1}{k-1} \end{aligned}$$

Suppose that G contains a k-clique. Consider the k unit vectors in the optimal solution corresponding to the vertices in this clique. These vectors satisfy the above inequality. Since there is an edge between every pair of these vertices, t^* must be at least the maximum dot product among these vectors, which is at least $\frac{-1}{k-1}$. ■

In view of Lemma 21.2.1 and Lemma 21.2.2, we interpret our SDP as the following program.

$$\begin{array}{lll} \text{minimize} & k & \\ \text{subject to} & v_i \cdot v_j \leq \frac{-1}{k-1} & \forall (i,j) \in E \\ & v_i \cdot v_i = 1 & \forall i \in V \\ & v_i \in \mathbb{R}^n & \forall i \in V \end{array}$$

21.3 Converting from SDP Solution to Coloring

Next we give a randomized algorithm for transforming a solution to the above SDP into a feasible coloring. For simplicity, we restrict our attention to 3-colorings. The following 3-coloring algorithm can be generalized to handle more colors with some more effort.

1. Pick t (whose value to be decided) "directions" (unit vectors) $\alpha_1, \ldots, \alpha_t$ uniformly at random.
2. Divide the graph into 2^t parts according to $sgn(\alpha_j \cdot v_i)$ for every j. Color each part with a unique color.
3. Recurse on monochromatic edges.

If G is 3-colorable, then by Lemma 21.2.1 we get an optimal solution to the SDP with $k \leq 3$ and $v_i \cdot v_j \leq -1/2$ for all $(i,j) \in E$. This implies that for any $(i,j) \in E$, the angle between v_i and v_j is at least $2\pi/3$.

Let Δ be the maximum degree of the graph. We pick $t = \log_3 \Delta + 1$.

Claim 21.3.1 *Suppose that we use the above algorithm to color a 3-colorable graph $G = (V, E)$. Then, the expected number of monochromatic edges after any iteration is at most $n/4$, where $n = |V|$.*

Proof: Consider any $(x, y) \in E$. Fix a j $(1 \leq j \leq t)$. Then, since the angle between v_x and v_y is at least $2\pi/3$ as discussed above, we have

$$\mathbf{Pr}[sgn(\alpha_j, v_x) = sgn(\alpha_j, v_y)] \leq \frac{1}{3}.$$

Hence,

$$\begin{aligned} \mathbf{Pr}[(x,y) \text{ is monochromatic}] &= \mathbf{Pr}[\forall j, sgn(\alpha_j, v_x) = sgn(\alpha_j, v_y)] \\ &\leq \frac{1}{3^t} \\ &= \frac{1}{3\Delta}. \end{aligned}$$

Thus, the expected number of monochromatic edges is at most $\frac{n\Delta}{2} \cdot \frac{1}{3\Delta} \leq n/4$. ■

We know from Claim 21.3.1 that the expected number of vertices need to be considered after any iteration is at most $n/2$ (since there are $n/4$ monochromatic edges). Thus, we have the following corollary.

Corollary 21.3.2 *With high probability, the above algorithm terminates after $O(\log n)$ iterations.*

We now analyze the number of colors used in the coloring. In each iteration, at most 2^t new colors are used. Supposing there are $\log n$ iterations, the total number of colors used is

$$\begin{aligned} 2^t \log n &\approx 2^{\log_3 \Delta} \cdot \log n \\ &= \Delta^{\log_3 2} \cdot \log n \\ &= \Delta^{0.63} \log n \\ &= O(n^{0.63} \log n). \end{aligned}$$

This is worse than the $O(\sqrt{n})$-approximation we got in Homework 1. We can improve the approximation factor by combining the above algorithm with the idea in Homework 1. Note that in the above analysis we simply bound the value of Δ by n. As in Homework 1, we can set a threshold for the value of Δ, say Δ^*. The combined algorithm is as follows.

1. Pick a vertex v with degree greater than Δ^*. 3-color the neighbors of v (including v), and remove them from the graph.

2. Repeat step 1 until all vertices in the graph have degree at most Δ^*.

3. Run the above 3-coloring algorithm on the pruned graph.

The number of colors used in the pruning phase (steps 1 and 2) is at most $\frac{3n}{\Delta^*}$. The (expected) number of colors used in step 3 is $(\Delta^*)^{0.63} \log n$. We can minimize their sum by setting $\Delta^* = n^{1/1.63}$, which leads to a $\tilde{O}(n^{0.39})$-coloring. A smarter preprocessing gives a $\tilde{O}(n^{0.25})$-coloring algorithm for 3-colorable graphs, but we will not discuss it in class.

References

[1] A. Wigderson. Improving the performance guarantee for approximate graph coloring. *Journal of the ACM*, 30(4):729–735, October 1983.

[2] A. Blum. New approximation algorithms for graph coloring. *Journal of the ACM*, 31(3):470–516, 1994.

[3] D. Karger, R. Motwani and M. Sudan. Approximate graph coloring by semidefinite programming. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, November 1994.

[4] A. Blum and D. Karger. An $\tilde{O}(n^{3/14})$-coloring algorithm for 3-colorable graphs. *Information Processing Letters*, 61(1):49–53, 1997.

[5] S. Arora, E. Chlamtac and M. Charikar. New approximation guarantee for chromatic number. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing*, 2006.

CS880: Approximations Algorithms	
Scribe: Dave Andrzejewski	**Lecturer:** Shuchi Chawla
Topic: Embedding metrics into trees	**Date:** 3/30/07

This lecture introduces the idea of embedding a metric into a tree, and applies this technique to the development of an approximation algorithm for the Multicommodity "Buy at Bulk" Network Design problem.

22.1 Multicommodity Buy-At-Bulk Network Design

22.1.1 Problem formulation

GIVEN:

- a graph $G = (V, E)$
- edge lengths ℓ_e
- pairs of demand vertices (s_i, t_i)
- quantities q_i to be sent $s_i \rightarrow t_i$
- a concave cost function $f(c_e)$ for "buying" capacity c_e on edge e

DO: find

- a set of paths P_i from s_i to t_i such that $\sum_{p \in P_i} p \geq q_i$ for each i
- a set of edge capacity purchases c_e such that $\sum_{\{p|e \in p\}} p \leq c_e$

such that the total cost $\sum_e f(c_e)\ell_e$ is minimized.

The cost of purchasing capcity c_e on edge e is defined as $f(c_e)\ell_e$. This means that the cost of purchasing edge capacity is linearly related to the length of that edge, which will be important for our analysis. Also note that the cost function f is concave, and shared by all edges. This makes our formulation the "uniform" case. If each edge were allowed to have a different cost function f_e, it would be the non-uniform case, which is much harder, and was not known to have any sub-polynomial approximation until recently, when a poly-log approximation was discovered [1].

22.1.2 Algorithm design

Our first observation is that this problem would be greatly simplified for the special case where G is a tree, because each $s_i \to t_i$ path would be unique.

Furthermore, we recall that the total cost is linear in the edge lengths ℓ_e. This means that if we can find a low-distortion embedding from our graph G to some tree T, it will be relatively simple to analyze the impact of the distortion on our cost function.

22.1.3 Tree embeddings

To analyze potential embeddings, we must first ask which graph structures would be most difficult to embed into a tree. A natural first thought is to consider a complete graph. However, note that we could simply place any single node as the hub, and have all other nodes only be connected to the root as spokes. In the simplified case of a a graph with uniform edge costs, this clearly achieves an expansion factor of $\rho = 2$ (Figure 22.1.1).

The actual worst-case would be a graph which is simply a single large cycle of all n nodes (n-cycle). In this case, we can create a tree by simply removing any single edge. However, the distance between the 2 endpoints of that edge has now expanded by a factor of $\rho = \Omega(n)$ (Figure 22.1.1). It can be shown, in fact, that no embedding of the n-cycle into trees has distortion $o(n)$.

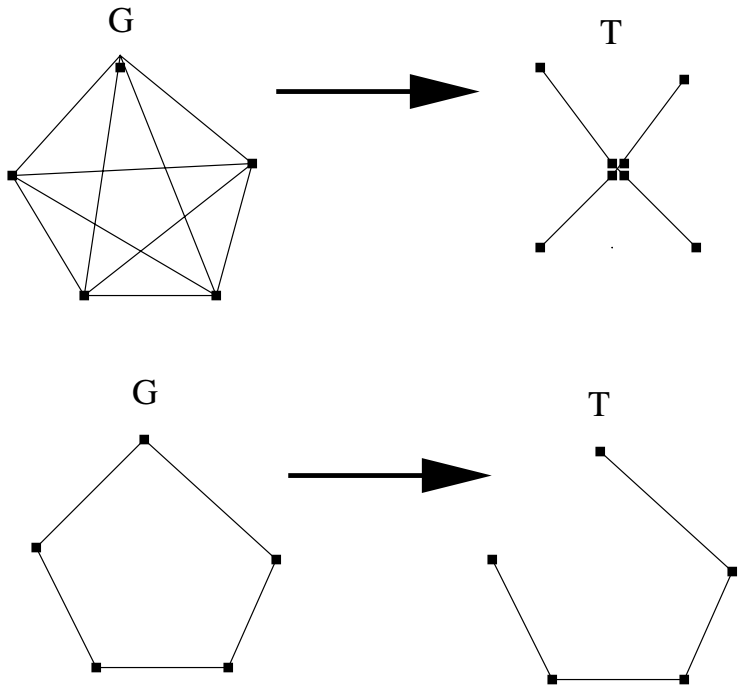


Figure 22.1.1: Embedding example graphs into trees.

22.1.4 Probabilistic tree embeddings

To avoid this worst-case scenario, we note that while embedding into a *single* tree may suffer large worst-case distortion, embedding into a *distribution* over trees can still achieve low expected distortion.

For our n-cycle graph example, define α as the uniform distribution over τ, the set of all trees T created by removing a single edge of the original graph. The expected distance between 2 vertices in an embedding drawn from α is then given by

$$d_\alpha(x,y) = E_{\alpha_T}[d_T(x,y)] = \sum_{T\in\tau} \alpha_T d_T(x,y)$$

where α_T is the probability of drawing tree T, and $d_T(x,y)$ is the distance between vertices x and y in T.

Theorem 22.1.1 *An n-cycle embeds into our distribution α with distortion $\rho = 2$.*

Proof: Each edge of the n-cycle is included in $n-1$ of the trees in τ. Therefore

$$E_\alpha[d_T(A,B)] = \frac{n-1}{n}(1) + \frac{1}{n}(n-1) \leq 2$$

■

We now formalize the our probabilistic embedding idea with the following definition.

Definition 22.1.2 *Given G, a* β-probabilistic embedding *into a distribution over trees is a distribution β over τ such that*

- $V[T_i] \supseteq V[G]\ \forall T_i \in \tau$
- $d_{T_i}(x,y) \geq d_G(x,y)\ \forall x,y\ \forall T_i \in \tau$
- $E_\alpha[d_T(x,y)] = \sum_T \alpha_T d_T(x,y) \leq \beta d_G(x,y)\ \forall x,y$

This definition sets up the main result from this lecture, which is that for all G there exists an $O(\log n)$-probabilistic embedding into trees.

22.1.5 Approximation of network design with a probabilistic embedding

Theorem 22.1.3 *Given a β-probabilistic embedding of G into trees, there exists a β-approximation for multicommodity uniform buy-at-bulk network design on G.*

Proof: Consider the following algorithm:

1. Given a β-probabilistic distribution α, pick $T \sim \alpha$

2. Solve uniform buy-at-bulk network design on T

3. Foreach $e = (u, v) \in T$, find shortest (u, v) path p in G and install capacity c_e^T on all edges in p

4. Map all p_i^T to their corresponding paths in G

This procedure clearly recovers a feasible solution on G, since all paths and capacities in the valid T solution are feasibly mapped to G.

What is the cost of our converted solution?

Claim 22.1.4 $E[cost_T] \leq \beta OPT$

Proof: Translate OPT_G to some solution in T by mapping each edge (u, v) in G to the unique path between u and v in T. Then

$$\begin{aligned} E[OPT_T] \leq E[cost_T] &\leq \sum_{(u,v)\in E_g} f(c_e^{OPT}) d_T(u, v) && (22.1.1)\\ &= \sum_{(u,v)\in E_g} f(c_e^{OPT}) \beta d_G(u, v) && (22.1.2)\\ &= \beta \sum_{(u,v)\in E_g} f(c_e^{OPT}) \ell_e && (22.1.3)\\ &= \beta OPT && (22.1.4) \end{aligned}$$

■

Claim 22.1.5 *Given a solution of cost X in T, our solution in G will have cost $\leq X$.*

Proof:

$$cost_G = \sum_{(u,v)\in E_T} f(c_e^T) d_G(u, v) \leq \sum_{(u,v)\in E_T} f(c_e^T) d_T(u, v) = X$$

■

These claims prove the theorem. ■

Note that this analysis relied on the fact that our objective function is linear in lengths $\ell_e = d_G(u, v)$.

These strategies were originally developed by Bartal, who derived $O(\log^2 n)$ and $O(\log n \log \log n)$ probabilistic embeddings of graphs into distributions over trees [4] [5]. Fakcharoenphol, Rao, and Talwar later improved these results to a $O(\log n)$ probabilistic embedding, which was also shown to be tight [3].

22.1.6 $O(\log n)$-probabilistic embedding

How can be get an $O(\log n)$-probabilistic embedding of general graphs into trees? The basic idea is to do a hierarchical clustering on all vertices in G (Figure 22.1.2).

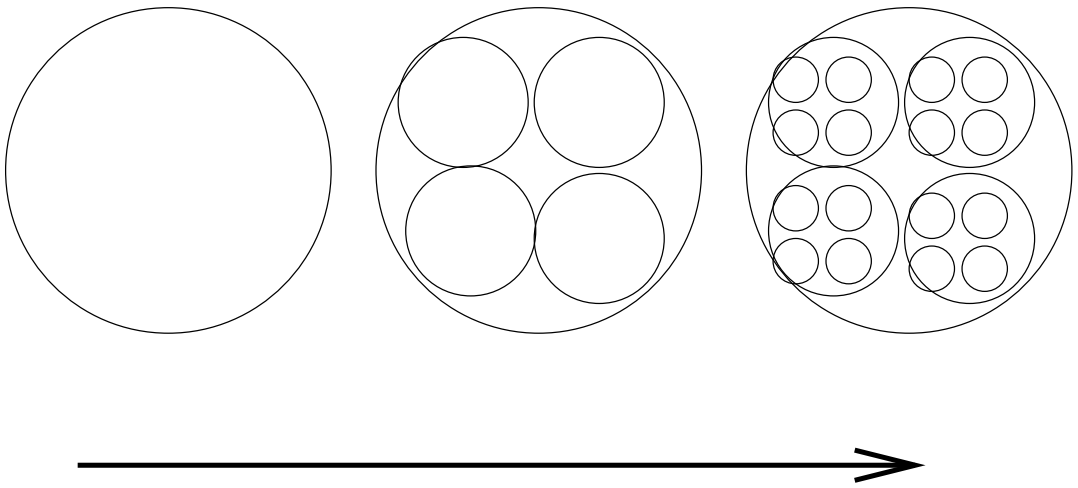


Figure 22.1.2: Hierarchical clustering by partitioning.

Under this scheme, all vertices of the original graph G are leaf nodes in our tree T. Each cluster in our hierarchical clustering then corresponds to a sub-tree in T. That is, all interior nodes of T are artifacts of our clustering scheme and were not originally present in G.

Starting from a graph with diameter Δ, we want our probabilistic embedding to have the property that $d_T(x, y) \geq d_G(x, y) \forall x, y$.

We can achieve this by building our clusters such that the diameter of the initial root cluster is Δ, the diameter of each child cluster is $\Delta/2$, and so on.

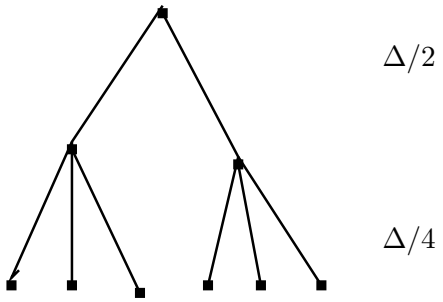


Figure 22.1.3: Tree representation of our hierarchical clustering.

Next, we need a partitioning scheme in order to build each level of our hierarchical clustering.

Definition 22.1.6 *A* β low-diameter low-distortion partitioning *with parameter* δ *is a partition of* V *into* $\{V_1, V_2, ..., V_k\}$ *such that*

1. $diam(V_i) \leq \delta \ \forall i$

2. $Pr[ecut] \leq \frac{d_e}{\delta}\beta \ \forall e \in E$

In this context, edge $e = (u, v)$ being "cut" by the partitioning means that $u \in V_i$ and $v \in V_j$ such that $i \neq j$.

Lemma 22.1.7 *Given a β low-diameter low-distortion partitioning scheme, there exists an $4\beta \log \Delta$-distortion probabilistic embedding of a graph into trees.*

Proof: Begin with a Δ-diameter graph. Partition with $\delta = \Delta/2$. Recursively embed $V_1, V_2, ..., V_k \hookrightarrow T_1, T_2, ..., T_k$.

Obviously, this approach satisfies properties 1 and 2 of Definition 22.1.6.

To see that it satisfies property 3, fix x, y and suppose that $e = (x, y)$ is an edge in E. Then suppose that they are separated at the top-level partitioning. Then $d_T(x, y) \leq 2\Delta$. If they are in the same subtree, then by induction

$$E_s[d_s(x, y)] \leq 4\beta \log \left(\frac{\Delta}{2}\right) d_G(x, y)$$

Since we must have 1 of these 2 cases (they are in the same subtree, or not), we can then calculate the expected distance between them in T. (Remember that for this first partition, $\delta = \Delta/2$.)

$$\begin{aligned}
E[d_T(x, y)] &= 2\Delta \left(\frac{d_e}{\delta}\beta\right) + 4\beta \log \left(\frac{\Delta}{2}\right) d_G(x, y) && (22.1.5)\\
&= 4\beta d_G(x, y) + 4\beta \log \left(\frac{\Delta}{2}\right) d_G(x, y) && (22.1.6)\\
&= 4\beta d_G(x, y)[1 + \log \left(\frac{\Delta}{2}\right)] && (22.1.7)\\
&= 4\beta d_G(x, y) \log \Delta && (22.1.8)\\
&= O(\beta \log \Delta) d_G(x, y) && (22.1.9)
\end{aligned}$$

Finally, it is easy to see that the worst distortion happens on edges present in the original graph G, so this bound will also hold for other pairs x, y.

■

The resulting tree is *k-hierarchically well-separated* (k-HST), with $k = 2$. This means that the edge weights between a parent and and all children are equal, and that the edge weights on any path from root to leaf decrease by a factor of at least k on each edge [4].

Our tree is also *ultrametric*, meaning that the root to leaf distances are the same for all leaves. Ultrametric trees have especially interesting applications in the construction of phylogenetic trees, which model the evolutionary relationships between the genomic sequences of different organisms [2].

The only missing component of our approach is now the β low-distortion low-diameter partition scheme, which will be introduced in the next lecture.

References

[1] C. Chekuri, M. T. Hajiaghayi, G. Kortsarz, M. R. Salavatipour. Approximation Algorithms for Non-Uniform Buy-at-Bulk Network Design. FOCS 2006, 677-686.

[2] Richard Durbin, Sean Eddy, Anders Krogh, and Graeme Mitchison. Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids. Cambridge University Press, 1998.

[3] Jittat Fakcharoenphol, Satish Rao, Kunal Talwar. A Tight Bound on Approximating Arbitrary Metrics by Tree Metrics. STOC 2003, 448-455.

[4] Yair Bartal. Probabilistic Approximations of Metric Spaces and Its Algorithmic Applications. FOCS 1996, 184-193.

[5] Yair Bartal. On approximating arbitrary metrices by tree metrics. STOC 1998, 161-168.

[6] V. Vazirani. Approximation Algorithms. Springer, 2001.

CS880: Approximations Algorithms

Scribe: Matt Darnall **Lecturer:** Shuchi Chawla
Topic: Tree Embeddings **Date:** 4/10/07

23.1 Problem Statement and Results

REcall that in the previous lecture, we discussed the use of probabilistic tree embeddings for approximations algorithms. We now present an algorithm for obtaining low distortion tree embeddings.

Let G be a graph and let $\mathcal{T}$ be a collection of trees on the same vertices as G. We say the $\mathcal{T}$ is a γ distortion if, for any vertices x and y and any $T \in \mathcal{T}$:

$$d_G(x, y) \leq d_T(x, y)$$

and

$$E[d_T(x, y)] \leq \gamma d_G(x, y)$$

The expected value is taken over a distribution given to the elements in $\mathcal{T}$. It has been shown in [2] that for an arbitrary graph we can have $\gamma = O(\log n \log \log n)$. This result has been improved to a $O(\log n)$ distortion in [1], which is essentially optimal.

23.2 Construction

Let G be a graph with n nodes and diameter $\Delta = 2^\delta$. The constructions for good tree embeddings have the set $\mathcal{T}$ made of trees of the following form. They are calle Hierarchically well seperated trees, which means the distance from each parent to its children is the same, and at each level the distance decrease by a constant factor. A tree will have a root that corresponds to the entire graph. Then, considering each node in the tree as a subset of the graph, the children nodes of each node, v, will be a partition of the vertices of the graph in the subset corresponding to v. We shall make it so that a node at a depth of i shall correspond to a subset of the graph with diameter less than $2^{\delta-i}$. Notice that at depth δ we are left with the vertices of the original graph.

We set the cost of traveling on an edge of depth i to be $2^{\delta-i}$. Thus, the cost of getting from a vertex u to a vertex v in one of our trees is at least the cost of getting from u to v in the graph since in a tree you have to travel up to a subset that contains both u and v and then back down. This cost will be large enough from the cost put on edges of depth i. Thus, we have that a set of trees with this property satisfies the first requirement to be a γ distotion embedding of G. The key to the argument, is that if the graph is partitioned well then the expected distance between two points won't change much. As seen by Bartal, if a metric graph can be probabilistically partitioned into pieces where the diameter has decreased by a constant factor and the chance an edge is cut is about its length times λ over the diameter, then, by recursively using these partitions for our

tree, we get a final probabilistic embedding into trees with distortion $O(\lambda log(\Delta)$. The existance of a partition with $\lambda = log(n)$ was given by Calinescu et al, see [3].

In [1] a method for partitioning the graph G randomly into a tree of the above form is given. By a better analysis, they were able to achieve $O(log(n))$ as the final distortion, removing the dependence on Δ. The basic idea is as follows. The vertices are ordered in an arbitrary manner. Then, a random $\beta \in [1, 2]$ is chosen. The set G is partitioned by moving through the vertices in the order given and including all points at a radius of $2^{\delta-1}\beta$ from the current vertex. The points in the ball are made into a cluster that form a node at the next level. We do this for each vertex, only including a new vertex in a cluster if it hasn't been assigned yet. We then treat each cluster as a new graph and repeat, adjusting the radius by a factor of 2.

Now, we look at the expected distance between vertices u and v. As shown in [1], the expected distance is raised by at most a factor of $O(log(n))$. Let e be the edge between u and v. The expected distance between u and v is less than:

$$\sum_j Pr[e \text{ is cut at time } t]2^t$$

.

Let $B(e, r)$ be the balls of radius r around u and v. By proving that the probability that e is cut at time t is less than $4d(u, v)log(\frac{|B(e,2^{\delta-t)}|}{|B(e,2^{\delta-t+1)}|}2^{-t}$ and noticing that this is a telescoping sum with first term $4d(u, v)log(n)$, we get the $O(log(n))$ factor.

This result is tight because tree metrics are contained in L^1 metrics, which we know have a distortion of $O(log(n))$. For more details on the proof, please see the references below.

References

[1] Fakcharoenphal et. al., *A Tight Bound on Approximating Arbitrary Metrics by Tree Metrics*. STOC 2003, San Diego, CA.

[2] Bartal, *On Approximating Arbitrary Metrics by Tree Metrics*. http://www.cs.huji.ac.il/ yair/pubs/B-prob-approx2.ps

[3] Calinescu, et. al.,*Approximation Algorithms for the 0-Extension Problem (2000)*. SODA 2001, Washington D.C., USA

CS880: Approximations Algorithms

Scribe: Matt Darnall **Lecturer:** Shuchi Chawla
Topic: Apporximate Counting **Date:** 4/12/07

24.1 Definitions

We will attempt to approximate a function that counts something. Typically, we are interested in finding the number of solutions to an NP-Complete problem, which is of course harder than solving the NP-Complete problem itself. So many such problems are # P-complete. An example of this is the number of ways to satisfy a statement in disjunctive normal form. A good algorithm for approximating a function will be an FPRAS or Fully Polynomial Randomized Approximation Scheme.

An algorithm A is a FPRAS for a function f if given an instance X of the problem, for any ϵ,

$$Pr\left[\frac{|A(X) - f(X)|}{f(X)} < \epsilon\right] < 1/4$$

and the algorithm runs in time polynomial in the length of X and $1/\epsilon$. The algorithm is just a PRAS if it runs in time polynomial in the length of X, but not in $1/\epsilon$.

What we are looking for is an algorithm A such that:

$$Pr\left[A(X) \in ((1-\epsilon)f(X), (1+\epsilon)f(X))\right] < 1 - \delta$$

where the runtime is polynomial in the size of X, $1/\epsilon$, and $log(1/\delta)$. By iterating, an FPRAS will give this to us.

24.2 Disjuntive Normal Form Counting

The problem we shall look at is counting the number of truth assignments to variables x_i such that the statement D is satisfied. D is forced to look like $D_1 \vee \cdots \vee D_m$, where each D_i is a conjunction of variables $x_{i_1} \wedge \cdots \wedge x_{i_k}$.

Let U be the set of all assigments, S be the set of satisfying assignments. We shall approximate $|S|$ by randomizing selecting elements of U and testing for membership in S. Then, if we test t samples from U and t' are from S, we know that:

$$\frac{|S|}{|U|} \approx \frac{t'}{t}$$

so our estimate is $|S| = t'|U|/t$. If $p = |S|/|U|$ is our probability of pulling a member of S, we get by Chernoff bounds that:

$$Pr\left[|t' - pt| > \epsilon pt\right] < e^{\frac{\epsilon^2 pt}{3}}$$

Thus, taking $t = \log(1-\delta)/(\epsilon^2 p)$ will get us within $1 \pm \epsilon$ of $|S|$ with probability greater than $1 - \delta$. The only thing that can make this not polynomial time is if p is small. This means that $|S|$ is small relative to $|U|$.

If we use Chebyshev's inequality instead of Chernoff, we get that:

$$Pr\left[|t' - pt| > \epsilon pt\right] < \frac{Var(t')}{\epsilon^2 p^2 t^2}$$

Since $Var(t') = p(1-p)t$, we need to take $t = (1-p)/(\epsilon^2 p\delta)$. In order to reduce the dependence on δ, we use the "'median of means"' method. Now, if we let $\delta = 1/4$, and we run the experiment with $t = 4(1-p)/(\epsilon^2 p\ 2\Delta + 1$ times, we expect about $(2\Delta + 1)/4$ of our trials to fall out of the $(1 \pm \epsilon)$ range of the actual solution. Specifically, the probability that the median of the $2\Delta + 1$ trials with $t = 4(1-p)/(\epsilon^2 p$ falls outside the $(1 \pm \epsilon)$ range of the actual solution is less than the probability that $\Delta + 1$ of the trials falls out of the $(1 \pm \epsilon)$ range of the actual solution. This probability, by Chernoff, is less than $(3/4)^s$. Thus, by running t trials $O(log(1/\delta))$ times, and taking the median of these answers, we are within $(1 \pm \epsilon)$ of the acutal solution with probability $1 - \delta$.

24.3 Reducing U

It is evident when analyzing the value of t that making the ratio $p = |S|/|U|$ small is crucial to limiting the runtime. We shall look at a method for reducing the universe U for our example of solutions to a DNF statement. Let the statement D be made up of m clauses D_i that are conjuntions of the literals or their negation. Let the solutions of D_i be S_i. Notice that S is just the union of the S_i and each S_i has size 2^{n-k_i}, where there are k_i literals in D_i and n literals in total. Although we can easily estimate $|S_i|$, it is hard to get an estimate of their union because of repitition.

Our new universe, U' shall be the set of pairs (i, a_i), $i \leq m$ where a_i is an element of S_i. Notice that sampling from U' at random is easy, we just pick a random i and then pick a random assignment to the variables not in D_i. In order for this to be uniform, we pick i with probability proportional to $|S_i|$, we we can do since cacluting $|S_i|$ is easy (just 2 to the power of the number of literals in S_i). The variables in D_i are fixed. We need a way to imbed the set S inside our new universe U' and a quick way to test membership in S. To do this, let every assignment, s, of the variables that satisfies D be associated with the first D_i that is satisfies. In other words, our embedded S' is the set of (i, s), where $s \in S$ and $i \leq m$ is the first D_i that s satisfies. Since it satisfies at least one of the D_i, we know that this assignment is counted exactly once in the set U', so S' is of the same size as S. Importantly, since there are only m clauses, and each element in S' is associated with a clause, $|S'|/|U'|$ is greater than $1/m$. Thus, we have made p larger than the inverse of the size of the instance, and our number of trials t can be polynomial in the size of the instance.

24.4 Counting = Sampling

In this section, we will give an overview of the proof that being able to approximately count the elements in S is equivalent to being able to uniformly sample from S. Here S is the solution set of a finite problem. The problem must be self reducible.

We define a problem Π to be *self reducible* if, given an instance P of size n:

1. The solutions that satisfy P can be written as binary strings in $poly(n)$. S_i is the set of solutions with the first bit of the string i.
2. There is a problem $P' \in \Pi$ of length less than n such that there is a bijection of the solutions in P' to the solutions S_0 (and so also S_1.

Theorem 24.4.1 *If Π is self reducible, then there is a near uniform way to sample solutions iff there is an FPRAS for Π.*

Near uniform way to sample solutions means the probability of selecting a particular solution is in between $(1-\epsilon)/t$ and $(1+\epsilon)/t$, where there are t solutions total.

If you can sample near uniformly, then we can use the fact that:

$$|S| = \frac{|S|}{|S_0|}\frac{|S|_0}{|S_{00}|}\cdots$$

to estimate $|S|$ using our uniform sampling and self reducibility to estimate the ratios on the right hand side. We will have a good estimate on $|S|$ if we had a good way of sampling.

If we have an FPRAS for Π, then we estimate the size of S_0 and S_1 using our FPRAS. We then select the first bit of our random solution to be 0 with probability $\frac{|S_0|}{|S_1|}$. We then recursively pick the next bits of our random solution in the same way. The fact that our FPRAS is able to approximate the number of solutions with a certain starting sequence of bits well means that we get a close to uniformly picked solution.

24.5 Further Reading

A great set of lecture slides on this material can be found at:

http://www3.math.tu-berlin.de/ipco05/Pages/Download/IPCO05_Dyer_LectureNotes.pdf

CS880: Approximations Algorithms	
Scribe: Dave Andrzejewski	**Lecturer:** Shuchi Chawla
Topic: Approx counting/sampling, MCMC methods	**Date:** 4/24/07

The previous lecture showed that, for self-reducible problems, the problem of estimating the size of the set of feasible solutions is equivalent to the problem of sampling nearly uniformly from that set. This lecture explores the applications of that result by developing techniques for sampling from a uniform distribution. Specifically, this lecture introduces the concept of Markov Chain Monte Carlo (MCMC) sampling approaches.

25.1 Markov Chain Monte Carlo (MCMC)

25.1.1 Problem motivation

There are many situations where we wish to sample from a given distribution, but it is not immediately clear how to do so. For example, we may have a function that gives the probability of an event within a normalization factor, but no way to calculate that normalization factor. Or the sample space of possible outcomes may be exponentially large, or even infinite.

25.1.2 Random walks and their properties

We approach this problem by considering our state space to be a graph, with individual events as nodes on this graph. If we can define this graph in a way such that the stationary distribution of a random walk over this graph is equal to our target distribution that we wish to draw samples from, then samples from a random walk on this graph can be used to approximate samples from the target distribution. We introduce the following definitions:

- Ω = the state space
- $n = |\Omega|$
- P = the transition matrix, $P_{ij} = Pr[i \rightarrow j]$
- π = a distribution on the nodes in Ω

Then, if we start from a node chosen from the distribution π and take a single step according to our transition matrix P, we get the distribution πP. Note that a random walk obeying these definitions is memoryless. That is, the next step in the walk depends only on the current state, and not on any history beyond that. This is also known as the *Markov property*, and our random walk is an example of a *Markov chain*.

For our random walk, there are two quantities of particular interest. The first is the *stationary distribution* π^*. This is a distribution over all states with the special property that $\pi^* P = \pi^*$. If we follow a random walk on a Markov chain, after a while we expect our position to be distributed

according to π^*. This is the definition of a stationary distribution. It is the limit distribution of the location of a random walk as the number of steps taken goes to infinity. There are special properties of the chain which are required to guarantee the existence and uniqueness of π^*, and these will be introduced shortly.

The second quantity of interest is the *mixing time* τ_ϵ, which is a measure of how long a random walk on the graph will take to converge to π^*. This will be defined more formally.

To illustrate these concepts, we will study the simple example of a d-regular undirected graph (all vertices have degree d). We define the transition probabilities from vertex to be uniform over all outgoing edges. That is, each edge is taken with probability $1/d$. We then make the following claims.

Claim 25.1.1 *For a uniform random walk over our d-regular graph, π^* is uniform.*

Claim 25.1.2 *For the directed version of our graph with $d(in) = d(out) = d$, π^* is uniform.*

Claim 25.1.3 *If G is undirected, $\pi^*(v) = \frac{d(v)}{2m}$.*

To see this last claim, consider that we can also define our random walk as a distribution over all edges. Let $Q(u, v)$ be the probability of taking the edge (u, v). Then:

$$\begin{aligned} Q(u,v) &= \pi^*(u)P_{uv} && (25.1.1)\\ &= \frac{d(u)}{2m}\frac{1}{d(u)} && (25.1.2)\\ &= \frac{1}{2m} && (25.1.3) \end{aligned}$$

Summing this over all d of v's neighbors then shows us that this π^* satisfies $\pi^*P = \pi^*$ for any all v and is therefore a valid stationary distribution. The first two claims follow by a similar argument.

But when can we know that a unique π^* exists? Consider the 2 example graphs shown in Figure 25.1.2. For the first graph, if we say that you start in the left node with probability 1, then your probability of being in that node is 1 at the start of iteration 1,3,5,... and 0 at the start of iteration 2,4,6,... Although the uniform distribution satisfies $\pi P = \pi$, it is not guaranteed that all random walks on the graph will converge to this distribution, as shown by this example. Therefore this graph can not have a stationary distribution, because it is periodic. For the second graph, you can reach 2 different stationary distributions starting from the center node, depending on which direction is taken on the first step. In this case there is no unique stationary distribution. These ideas are formalized in the following theorem:

Theorem 25.1.4 *An aperiodic irreducible finite Markov chain is ergodic and has a unique stationary distribution.*

A chain is aperiodic if for every state, there is no number > 1 which can divide the index of every future step which has non-zero probability of returning to that state. That is, given that you are in the state, there is no periodic pattern to when you can return (every second step, third step etc). This can be a somewhat tricky notion to prove about a graph, but adding self-edges to all nodes

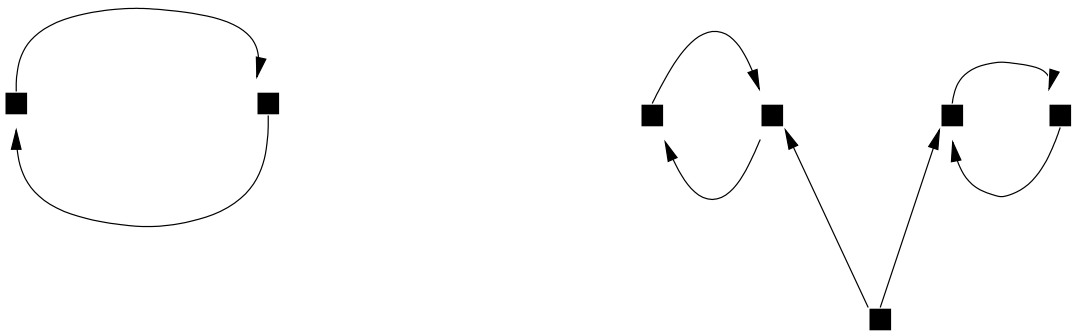


Figure 25.1.1: Two Markov chains which do not have unique stationary distributions.

automatically makes the chain aperiodic, so it is often easiest to simply define the graph in this way.

A chain is irreducible if it is possible to get from any state to any other state. This notion of 'reachability' can be considered an equivalence relation, with a chain being irreducible if all states are in the same equivalence class.

Now we formally define the mixing time τ_ϵ, first defining the mixing time of a random walk starting from state x.

$$\tau_\epsilon(x) = \min\{t \text{ s.t.} |\pi_x P^{t'} - \pi^*|_1 < \epsilon \ \forall t' \geq t\} \tag{25.1.4}$$

Where $\pi_x = 1$ for state x, and 0 for all other states. The mixing time for the chain itself is then:

$$\tau_\epsilon = \max_x \tau_\epsilon(x) \tag{25.1.5}$$

Different chains can have widely varying mixing times. For example, a walk over a regular expander can never get 'trapped' in any subset of the graph, because of the expander property. In the 'lollipop' graph (Figure 25.1.2) which consists of a clique with $n/2$ nodes and a long 'stem' with $n/2$ nodes, a random walk will take $O(n^3)$ steps to ever reach the end of the stem starting from a point inside the clique.

25.1.3 Mixing time analysis

For a given chain then, we would like to bound τ_ϵ by some polynomial in n. How can we do this? There are 3 general analysis approaches.

1. Conductance
2. Canonical paths
3. Coupling

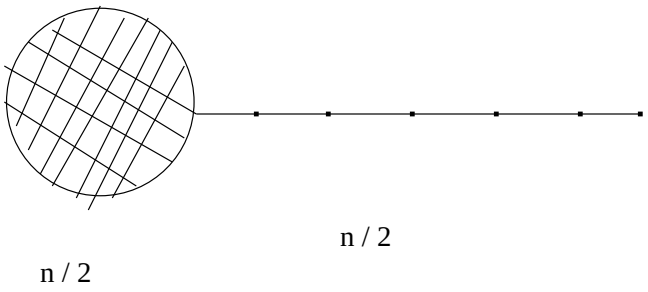


Figure 25.1.2: A graph with a slow mixing time.

The key property of the transition matrix which determines mixing time is the eigenvalue gap γ between the principal and second eigenvalues. Assume that all eigenvalues λ are real (which is the case for undirected graphs anyways). Then order the eigenvalues $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_n$. Then call the eigenvalue gap $\gamma = \lambda_1 - \lambda_2$.

We know that at least one of the λ_i's is equal to 1, because by definition π^* is an eigenvector or P whenever a stationary distribution exists ($\pi^* P = \pi^*$). Since P is a stochastic matrix, it is also easy to see that $|\lambda_i| \leq 1 \forall i$, therefore $\lambda_1 = 1$ and π^* is the first eigenvector. It is also interesting to note than if $|\lambda_i| < 1$, that means that v_i must have mixed-sign components, and cannot be normalized to sum to 1. Also, λ_2 cannot be 1, unless the graph has a more than one strongly connected component, and the stationary distribution is therefore non-unique.

Also, the eigenvectors of P are guaranteed to be orthogonal if P is real and symmetric, which corresponds to a time-reversible Markov chain, where time-reversible means that $\pi^*(u)P_{uv} = \pi^*(v)P_{vu} \forall u, v$.

Theorem 25.1.5 *$\tau_\epsilon \leq O(\frac{1}{\gamma}\log(\frac{n}{\epsilon})$ for a time-reversible Markov chain.*

Proof: Consider the representation of a distribution over states π in the basis of the eigenvectors $\pi = \sum_i c_i v_i$.

$$\pi P = (\sum_i c_i v_i)P = \sum_i c_i \lambda_i v_i \tag{25.1.6}$$

$$\pi P^2 = \sum_i c_i \lambda_i^2 v_i \tag{25.1.7}$$

$$\pi P^t = \sum_i c_i \lambda_i^t v_i \tag{25.1.8}$$

Observe that for $|\lambda_i| < 1$, $\lambda_i^t \to 0$ as $t \to \infty$. The lone exception if for $i = 1$, since $|\lambda_1| = 1$. Thus we can see that $\pi P^t \to \pi^* = c_1 v_1$ as $t \to \infty$, where $c_1 = 1$ as $\pi^* = v_1$. We can express the distribution at time t in terms of the stationary distribution and an error term.

$$\pi P^t = \pi^* + \sum_{i>1} c_i \lambda_i^t v_i \tag{25.1.9}$$

$$||\pi P^t - \pi^*||_2^2 = ||\sum_i c_i \lambda_i^t v_i||_2^2 \tag{25.1.10}$$

$$= \sum_i c_i^2 \lambda_i^{2t} ||v_i||_2^2 \tag{25.1.11}$$

(the previous step uses the orthogonality of the v_i's)

$$\leq \lambda_2^{2t} \sum_{i>1} c_i^2 ||v_i||_2^2 \tag{25.1.12}$$

$$\leq \lambda_2^{2t} \tag{25.1.13}$$

The final step can be seen by noting that $\sum_{i>1} c_i^2 ||v_i||_2^2 \leq \sum_i c_i^2 ||v_i||_2^2 = ||\pi||_2^2 \leq 1$, because our original π is a probability distribution.

Now we wish to use this ℓ_2 bound to get an ℓ_1 bound. It is a general result that, in n-dimensional space $||x||_1 \leq \sqrt{n}||x||_2$ (Figure 25.1.3). This result is essentially a restatement of the Cauchy-Schwarz inequality. Plugging this result into our bound from above, we get:

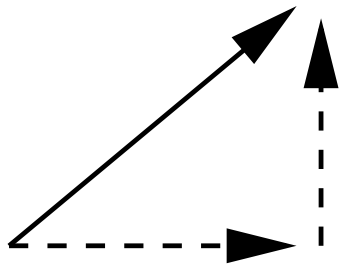


Figure 25.1.3: Visual representation of bounding the ℓ_1-norm in terms of the ℓ_2-norm.

$$||\pi P^t - \pi^*||_1 \leq \lambda_2^t \sqrt{n} \tag{25.1.14}$$

We want to pick t such that $\lambda_2^t \sqrt{n} \leq \epsilon$.

$$(1-\gamma)^t \sqrt{n} \leq \epsilon \tag{25.1.15}$$

$$e^{-t\gamma} \sqrt{n} \leq \epsilon \tag{25.1.16}$$

$$\frac{\sqrt{n}}{\epsilon} \leq e^{t\gamma} \tag{25.1.17}$$

$$t \geq \frac{1}{\gamma} \log(\frac{\sqrt{n}}{\epsilon}) \tag{25.1.18}$$

■

25.1.4 Conductance

Definition 25.1.6 *The* conductance $\phi(G)$ *of a distribution* π *over graph* G *is defined as*

$$\phi(G) = \min_{S \subset V, \pi(S) \leq 1/2} \left[\frac{\sum_{u \in S, v \notin S} \pi(u) P_{uv}}{\pi(S)} \right] \tag{25.1.19}$$

The quantity in the brackets can be roughly understood as the probability of 'escaping' set S. The conductance $\phi(G)$ then finds the 'stickiest' set in G, and is closely related to the notion of graph sparsity discussed earlier in the course. We can bound the conductance in terms of the eigenvalue gap γ.

Theorem 25.1.7

$$\frac{\phi^2(G)}{2} \leq \gamma \leq 2\phi(G) \tag{25.1.20}$$

This inequality can be used to bound the sparsity, but since $\phi(G)$ can be small, $\phi^2(G)$ can be very small, giving a loose and unhelpful bound. If we can show that $\phi(G)$ is large however, this bound will be relatively tight. A tight bound on the eigenvalue gap γ can then be plugged into the previous theorem to bound τ_ϵ. This is the basic approach of the conductance method.

Corollary 25.1.8

$$\tau_\epsilon \leq O(\frac{1}{\phi^2(G)} \log(\frac{n}{\epsilon})) \tag{25.1.21}$$

25.1.5 Canonical paths

The basic concept of the canonical paths technique is to find paths between all pairs of points (canonical paths), and then argue that no edge is "overloaded" with respect to its probability, which plays a role analogous to its capacity.

If the canonical paths can be shown to have low congestion, this can be shown to be equivalent to saying the graph has high conductance.

Let T_{xy} be the canonical path between x and y. Then the congestion ρ of a set of canonical paths T be defined as:

$$\rho(T) = \max_e \sum_{e \in T_{xy}, e=(u,v)} \frac{\pi^*(x)\pi^*(y)}{Q(u,v)} \tag{25.1.22}$$

The congestion $\rho(T)$ can then be used to bound the mixing time τ_ϵ.

Theorem 25.1.9

$$\tau_\epsilon \leq \rho(T) \log(\frac{n}{\epsilon}) \tag{25.1.23}$$

25.1.6 Coupling

The coupling technique works by running two Markov chains X and Y in parallel. The chain X is started from some initial distribution π, while the chain Y is started from the stationary distribution π^*. The evolution of the chains im time is then coupled or linked. If it can be shown that for some t we get $X^t = Y^t$, then by the Markov property the chain X will have reached the stationary distribution π^*.

Consider the example of the random walk on the graph shown in Figure 25.1.6. From our earlier claims, we can immediately see the stationary distribution π^* for this walk is uniform. So now consider our two random walks X and Y, where their evolution is defined such that for each step:

- X chooses a neighbor uniformly
- Y go in the same 'direction' as X

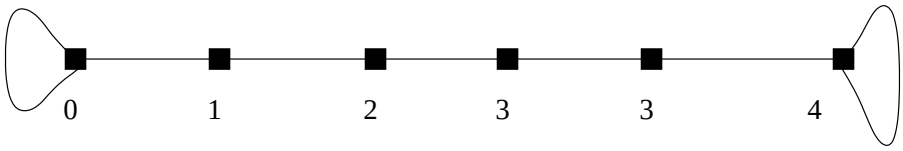


Figure 25.1.4: Coupling method example.

The key consequence of this scheme is that whenever one of the random walks is at an endpoint, the distance between the two points will be reduced by 1 if that walk takes the self-loop edge of that endpoint, which will happend with probability 1/2. This means that, no matter where the two chains started, we can guarantee that $X = Y$ after a self-loop edge has been taken n times. The following lemma formalizes the relationship between mixing (small $||X^t - \pi^t||_1$) and coupling ($X^t = Y^t$).

Lemma 25.1.10 $||X^t - \pi^*||_1 \leq 2Pr[X^t \neq Y^t]$

Proof:

$$
\begin{aligned}
Pr[X^t = Y^t] &\leq \sum_i \min\{X_i^t, Y_i^t\} && (25.1.24)\\
||X^t - Y^t||_1 &= \sum_i \max\{X_i^t, Y_i^t\} - \min\{X_i^t, Y_i^t\} && (25.1.25)\\
&= \sum_i (\max\{X_i^t, Y_i^t\} + \min\{X_i^t, Y_i^t\}) - 2\min\{X_i^t, Y_i^t\} && (25.1.26)\\
&= 2 - 2\sum_i \min\{X_i^t, Y_i^t\} && (25.1.27)\\
&\leq 2(1 - Pr[X^t = Y^t]) && (25.1.28)\\
&\leq 2(Pr[X^t \neq Y^t]) && (25.1.29)
\end{aligned}
$$

■

In order to complete the argument, we need to determine a t such that $Pr[X^t \neq Y^t] \leq \epsilon$. The crux of this method is determining the hitting time $h(u, v)$ for each pair of points (u, v), where $h(u, v)$ is the expected time to reach v, starting from u. This will give us the time it takes to go to an end point on the line starting from an arbitrary point in the middle.

For this example, the maximum $h(u, v)$ occurs when u and v are the two endpoints on the opposite sides of the graph, A and B. This can be shown to be n^2 by solving the following set of equations.

$$h(i,0) = 1 + \frac{1}{2}h(i-1,0) + \frac{1}{2}h(i+1,0) \ \forall 1 < i < n \tag{25.1.30}$$

$$h(1,0) = 1 + \frac{1}{2}h(2,0) \tag{25.1.31}$$

This $O(n^2)$ hitting time in turn implies an $O(1/\epsilon, n^3)$ mixing time for this random walk. We omit the details of this step, which involves the use of Markov's inequality.

The coupling analysis technique requires typically requires the underlying graph to have a nice, known structure. It can be applied to a number of Markov chains, such as electrical networks, card shuffling, and random graph colorings.

References

[1] V. Vazirani. Approximation Algorithms. Springer, 2001.

CS880: Approximations Algorithms	
Scribe: Dave Andrzejewski	**Lecturer:** Shuchi Chawla
Topic: Metropolis method, volume estimation	**Date:** 4/26/07

The previous lecture discussed they some of the key concepts of Markov Chain Monte Carlo (MCMC) methods, including the stationary distribution π^* and the mixing time τ_ϵ. This lecture introduces the Metropolis method for constructing Markov chains in order to sample from some distribution. The use of sampling methods for volume estimation is also introduced.

26.1 Metropolis method

26.1.1 MCMC review

Recall from last time the key properties of a random walk Markov chain.

- Ω = the state space
- $n = |\Omega|$
- P = the transition matrix, $P_{ij} = Pr[i \rightarrow j]$
- π^* = the stationary distribution such that $\pi^* P = \pi^*$
- τ_ϵ = the mixing time, after which the ℓ_1-norm of the difference between the chain distribution and the stationary distribution is guaranteed to be $< \epsilon$.

Also recall this important theorem concerning the existence and uniqueness of the stationary distribution π^*.

Theorem 26.1.1 *An aperiodic irreducible finite Markov chain is ergodic and has a unique stationary distribution.*

We can easily guarantee the aperiodicity of our chain by simply adding self-loops to all vertices. This will increase the mixing time by no more than a factor of 2.

26.1.2 Metropolis filter

But how do we actually construct a Markov chain with a stationary distribution equal to our target distribution? Also, we want this method to have a good (that is, small) mixing time. The Metropolis method allows us achieve these goals by defining our Markov chain as a random walk over a suitably defined graph.

We define the approach as follows. Say we which to sample values $i \in \Omega$ from a distribution $Q(i)$. Then we define an undirected d-regular graph G on Ω, picking this graph in such a way that it has high conductance. Then from node v, pick the next node u uniformly from the d neighbors. Then:

- If $Q(u) \geq Q(v)$, move to node u
- Else move to node u with probability $\frac{Q(u)}{Q(v)}$, stay with probability $(1 - \frac{Q(u)}{Q(v)})$.

First we examine the graph itself. Since it is fully connected and undirected, it is irreducible. Since all nodes have self-edges, it is aperiodic. Therefore this random walk is guaranteed to have a unique stationary distribution π^*. Now we must show that this stationary distribution is equal to our target distribution Q.

Claim 26.1.2 $\pi^* = Q$

Proof: Say that our initial $\pi = Q$, then take one step. Consider any node v, and calculate the probability of arriving at node v after this one step. If it is equal to $Q(v)$, then we have shown that $QP = Q$, and therefore $\pi^* = Q$.

We need to calculate the probability of starting at distribution Q, taking one step, and then ending up in state v. This can be decomposed into three cases: we move from a neighbor u into v where $Q(u) \geq Q(v)$, we move from a neighbor u into v where $Q(u) < Q(v)$, or we are already in v and we choose a neighbor u such that $Q(u) < Q(v)$ but we end up staying at v. Let n be the number of neighbors u such that $Q(u) \geq Q(v)$.

$$
\begin{aligned}
Q'(v) &= \sum_{\substack{u|(u,v)\in G,\\ Q(u)\geq Q(v)}} \frac{1}{d} Q(u) \frac{Q(v)}{Q(u)} + \sum_{\substack{u|(u,v)\in G,\\ Q(u)<Q(v)}} \frac{1}{d} Q(u) + \sum_{\substack{u|(u,v)\in G,\\ Q(u)<Q(v)}} \frac{1}{d} Q(v)(1 - \frac{Q(u)}{Q(v)}) && (26.1.1)\\
&= \frac{n}{d} Q(v) + \frac{d-n}{d} Q(u) + \frac{d-n}{d} Q(v) - \frac{d-n}{d} Q(u) && (26.1.2)\\
&= Q(v) && (26.1.3)
\end{aligned}
$$

■

This shows that a random walk using the Metropolis method is guaranteed to converge to our target distribution Q. It is worth noting that our scheme of uniformly choosing a neighbor is a special case of the general Metropolis-Hastings sampler [3]. In the more general case, a proposal distribution is used to select the next candidate state conditioned on the current state. This proposal distribution need not be uniform over neighbors, and in fact need not even be symmetric.

26.1.3 Volume estimation

An interesting application of sampling techniques is the problem of estimating the volume of a convex shape $K \in \mathbb{R}^n$ using an inclusion oracle which reveals whether a given point is contained in the shape or not. We are also given two balls, one completely enclosing K and one completely enclosed by K. Call these $K \subseteq B(0, R)$ and $K \supseteq B(0, r)$. This technique that we use has interesting parallels to the concept of self-reducibility.

What is the probability that a uniformly chosen point in the larger ball will be in K? We can use the smaller ball to bound this probability as $\geq \frac{vol(B(0,r))}{vol(B(0,R))}$. However, for large n we will suffer the 'curse of dimensionality' [4], and this lower bound will be very small, in particular $(\frac{r}{R})^n$.

The key insight of our approach is that we can use a sample from a *series* of regions K_i in order to make the probability of finding a point in the region as large as possible.

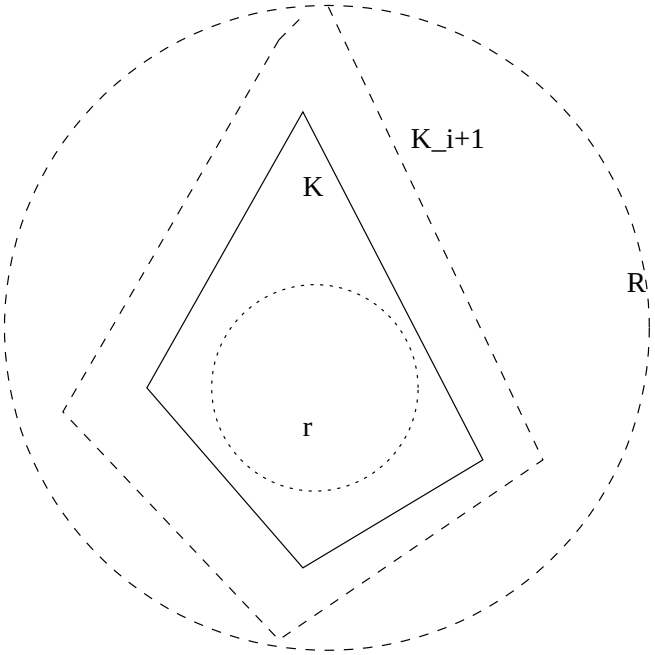


Figure 26.1.1: An example of our volume estimation problem setup.

$$K = K_o \subseteq K_1 \subseteq ... \subseteq K_\ell = B(0, R) \tag{26.1.4}$$

These regions are constructed such that $\frac{vol(K_{i+1})}{vol(K_i)}$ is small for all i. Furthermore we choose a small $\ell \approx n \log(\frac{R}{r})$, so we do not need to iterate too many times. We can estimate the volume of K using estimates of the ratios of adjacent K_i in the following way.

$$vol(K) = \frac{vol(K_o)}{vol(K_1)} \frac{vol(K_1)}{vol(K_2)} ... \frac{vol(K_{\ell-1})}{vol(K_\ell)} vol(B(0, R)) \tag{26.1.5}$$

So now we simply need to figure out how to sample from K_i. We define K_i as a re-scaling of K, subject to containment by $B(0, R)$.

$$K_i = (1 + 1/n)^i K \cap B(0, R) \tag{26.1.6}$$

This gives us a nice bound on the volume ratio of adjacent K_i.

$$\frac{vol(K_{i+1})}{vol(K_i)} \leq (1 + 1/n)^n = e \tag{26.1.7}$$

Note that $(1+\frac{1}{n})^{n \log R/r} K$ contains $B(0,r)$, therefore e is indeed $O(n \log R/r)$.

To sample from K_i, we then simply sample uniformly from K and then re-scale. But how to sample from K itself? To approach this problem, we employ the MCMC methods we have been discussing.

We define our random walk, known as the Ball-walk, as follows. From any point $u \in K$, sample a point randomly from the ball centered at u with radius δ, $B(u,\delta)$, and move to the new point if it is inside K. If the point is outside K, stay at u.

Note that the graph defined by this rule allows us to reach any point from any other point, and also allows self-loops. Therefore it is irreducible and aperiodic, and must have a unique stationary distribution π^*. The resulting Markov chain is time-reversible. Therefore, the stationary distribution is uniform.

For the practicality of this scheme, it is important to choose a good value for δ in order to get good samples from K. Taken to the extreme, a huge δ value would result in constantly picking points outside K, and therefore remaining at the current point. Likewise, a very small δ would result in taking very small steps, making it very slow to explore all of K. Also, if something is known about the geometry of K, it may be helpful to rescale the proposal ball to an ellipse, for example. This is accomplished by putting the body in an "isotropic" position via an affice transformation, so as to remove all sharp corners.

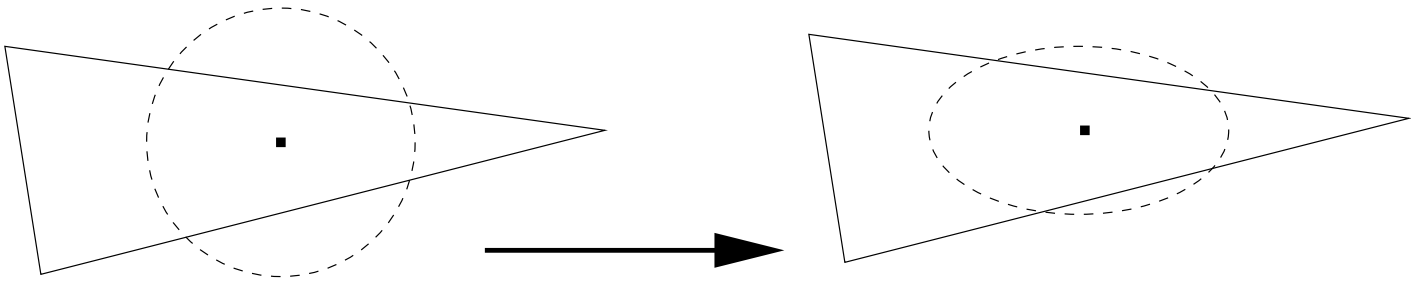


Figure 26.1.2: Rescaling the proposal ball to an ellipse based on the geometry of K.

The first approach based on this technique was polynomial in n, but with an unfortunate order $O(n^{23})$ [1]. Newer approaches, dubbed 'hit and run', first choose a direction, and then sample uniformly from the line segment along that direction contained in K. This approach drastically improves mixing time, achieving $\tilde{O}(n^4)$ [2].

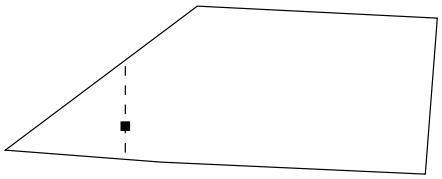


Figure 26.1.3: The 'hit and run' technique.

The inapproximability result is that one cannot estimate volume within a constant factor in $\Omega(n^2)$ time.

References

[1] Martin Dyer, Alan Frieze, Ravi Kannan. A random polynomial-time algorithm for approximating the volume of convex bodies. JACM 1991.

[2] Laszlo Lovasz, Santosh Vempala. Simulated Annealing in Convex Bodies and an $O(n^4)$ Volume Algorithm FOCS 2003.

[3] D. MacKay. Information Theory, Inference, and Learning Algorithms. Cambridge University Press, 2003.

[4] Trevor Hastie, Robert Tibshirani, Jerome Friedman. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer-Verlag, 2001.

[5] V. Vazirani. Approximation Algorithms. Springer, 2001.

CS880: Approximation Algorithms

Scribe: Matt Elder | **Lecturer:** Shuchi Chawla
Topic: Lagrangian Relaxation | **Date:** 4/26 and 4/27, 2007

We have seen many examples of the utility of linear programming. In some cases, to round an LP solution to an integer solution demands that we relax a constraint that we prefer to maintain. The Lagrangian technique will yield a method to maintain such constraints. This technique is especially useful for bicriteria optimization problems, that is, problems with two objectives where we have a fixed bound on one objective and want to optimize the other.

For example, recall the k-median problem: We are given a set of customers, a set of facilities, and a routing cost from each customer to each facility. We want to open no more than k facilities, while minimizing the total routing cost. Using standard LP techniques, it is difficult to round a relaxed LP solution to an integer LP solution without using more than k facilities. Lagrangian relaxation provides a workaround for this problem, so that we can guarantee that the final, integer solution obeys the k-facility constraint.

To demonstrate the technique of Lagrangian relaxation, we consider a solution to the k-minimum spanning tree problem. An approximation to k-median can be obtained in a similar way.

26.1 k-Minimum Spanning Trees

In an instance of the k-minimum spanning tree problem, we have a graph $G = (V, E)$, a cost $c_e > 0$ for each edge in E, and a root vertex $r \in V$. We want to find a tree that connects at least k nodes to the root while minimizing the total cost for the tree's edges. We are free to choose the set of k vertices that we will connect to the root r; call this set S.

Note that the relationship between the k-MST problem and the prize-collecting Steiner tree problem is analogous to the relationship between the k-median problem and the facility location problem. Both k-MST and k-median contain a constant-size-set restriction, which performs the task of an extra cost parameter in prize-collecting Steiner tree and facility location. As we will see, this relationship is central to the idea of the Lagrangian relaxation technique.

The integer LP for k-MST is as follows:

$$y_v = \begin{cases} 1, & v \text{ is not in the tree.} \\ 0, & \text{otherwise.} \end{cases}$$

$$x_e = \begin{cases} 1, & e \text{ is in the tree.} \\ 0, & \text{otherwise.} \end{cases}$$

$$\sum_{e \in \delta(S)} x_e \geq 1 - y_v, \ \forall S \subseteq V \setminus \{r\}, \forall v \in S.$$

$$\sum_{v \in V} y_v \leq n - k. \qquad (*)$$

$$\text{minimize} \sum_{e \in E} c_e x_e.$$

To use LP techniques, we need to relax this integer LP to a real-valued LP, and somehow still be able to respect Constraint $*$ when we round real values back to integers. We shall do this by introducing a family of LPs, parameterized by the Lagrange multiplier λ. We can think of varying λ as varying the cost of omitting vertices from our tree. It's important to note that λ is not, itself, a variable of the LP. It is a parameter of the LP, and is constant with respect to any routine that produces LP solutions. So, define the linear program LR_λ as follows:

$$y_v \in [0, 1]$$

$$x_e \in [0, 1]$$

$$\sum_{e \in \delta(S)} x_e \geq 1 - y_v, \ \forall S \subseteq V \setminus \{r\}, \forall v \in S.$$

$$\text{minimize} \sum_{e \in E} c_e x_e + \lambda \left(\sum_v y_v - (n - k) \right).$$

The term $\lambda \left(\sum_v y_v - (n - k) \right)$, above, replaces Constraint $*$ in the original LP. For any λ, the LP LR_λ has the same optimal solution as the following prize-collecting Steiner tree LP, PCST_λ:

$$y_v \in [0, 1]$$

$$x_e \in [0, 1]$$

$$\sum_{e \in \delta(S)} x_e \geq 1 - y_v, \ \forall S \subseteq V \setminus \{r\}, \forall v \in S.$$

$$\text{minimize} \sum_{e \in E} c_e x_e + \lambda \left(\sum_v y_v \right).$$

Allowing LP names to stand for the optimal values of their objective functions, it's clear that $\text{PCST}_\lambda - \lambda(n-k) = \text{LR}_\lambda$. Furthermore, any solution to the k-MST problem is a feasible solution to LR_λ; when Constraint $*$ is tight, as it is for all solutions to the k-MST problem, then the objective value of this solution is the same in both problems. So, $\text{LR}_\lambda \leq \text{OPT}$. (OPT is the optimal solution to k-MST. Remember, that's the problem that we're (still) trying to approximate.)

Let PCST'_λ be the integer solution to PCST_λ yielded by the LP-dual algorithm. If we let $\lambda = 0$, then PCST'_λ is a tree containing only the root because there is no penalty for leaving unused vertices. Similarly, if we let $\lambda = \max_e c_e$, then PCST'_λ will contain all vertices because the penalty for unused vertices dominates the cost of expanding the tree. So, it seems like there should be some moderate value of λ for which PCST'_λ contains nearly k vertices. This need not quite be the case, but we can use binary search to find two values of lambda, $\lambda_1 \approx \lambda_2$, for which we get two trees T_1 and T_2 such that $|T_1| < k < |T_2|$. From these trees, we can interpolate a solution using exactly k vertices. However, with luck, this interpolation may not be necessary.

Theorem 26.1.1 *If $PCST'_\lambda$ has k vertices, then it gives a 2-approximation to k-MST.*

Proof: Let $x, y \stackrel{\text{def}}{=} \text{PCST}'_\lambda$. Since PCST'_λ has k vertices, we know that $\sum_v y_v = n - k$. Then, by our analysis of Problem 4 in Homework 3,

$$\sum_e c_e x_e + 2\lambda \sum_v y_v \leq 2\text{PCST}_\lambda, \text{ so}$$

$$\sum_e c_e x_e \leq 2\left(\text{PCST}_\lambda - \lambda \sum_v y_v\right) = 2\left(\text{PCST}_\lambda - \lambda(n-k)\right) = 2\text{LR}_\lambda \leq 2\text{OPT}.$$

■

If we are unable to find a λ such that PCST'_λ has exactly k vertices, then we need to find a way to combine T_1 and T_2 into a single tree, which does not cost much more than OPT. Let $\lambda_1 = \lambda_2$; except that they generate two different trees, we assume that the difference between λ_1 and λ_2 is negligible.

Let μ_1 and $\mu_2 \stackrel{\text{def}}{=} 1 - \mu_1$ satisfy $\mu_1 k_1 + \mu_2 k_2 = k$, where $k_1 = |T_1|$ and $k_2 = |T_2|$. Then:

$$\mu_1 = \frac{k_2 - k}{k_2 - k_1}$$
$$\mu_2 = \frac{k - k_1}{k_2 - k_1}$$

Now, letting $c(T)$ denote the cost of tree T, we know the following

$$\begin{aligned}
c(T_1) + 2\lambda(n - k_1) &\leq 2\text{PCST}_\lambda, \text{ and} \\
c(T_2) + 2\lambda(n - k_2) &\leq 2\text{PCST}_\lambda, \text{ so} \\
\mu_1 c(T_1) + \mu_2 c(T_2) + 2\lambda(n - \mu_1 k_1 - \mu_2 k_2) &\leq 2\text{PCST}_\lambda, \text{ which yields} \\
\mu_1 c(T_1) + \mu_2 c(T_2) &\leq 2\left(\text{PCST}_\lambda - \lambda(n-k)\right) \leq 2\text{OPT}.
\end{aligned}$$

If $\mu_2 \geq \frac{1}{2}$, then $c(T_2) \leq 2\mu_2 c(T_2) \leq 4\text{OPT}$. Since $|T_2| > k$, we can simply use T_2 as our solution.

Otherwise, $\mu_1 \geq \frac{1}{2}$. Let $T_2' \stackrel{\text{def}}{=} T_2 \setminus T_1$. The following subroutine, Find-Subtree, will find a subtree of T_2 of size at least $(k - k_1)$.

Find-Subtree:

1. Exchange each undirected edge of T_2 for two directed edges of the same cost, one pointing each way. These edges form an Euler tour containing all vertices of T_2'. Note that each vertex appears twice in the tour.
2. From each vertex in T_2', start following the Euler tour in a clockwise direction until $2(k - k_1)$ nodes of T_2' are encountered, including repeats. This gives us at least $2(k_2 - k_1)$ different subpaths of the Euler tour, two for each vertex in T_2'.
3. Return the shortest such subtour.

Each edge of the Euler tour belongs to exactly $2(k - k_1)$ subpaths and there are at least $2(k_2 - k_1)$ subpaths in all. Therefore, since the cost of the entire Euler tour is $2c(T_2)$, one of the subpaths has length at most $\frac{2(k-k_1)}{2(k_2-k_1)} 2c(T_2)$.

So, suppose that Find-Subtree outputs the tree S. S contains at least $(k - k_1)$ distinct nodes of T_2, and costs at most $\frac{2(k-k_1)}{k_2-k_1} c(T_2) = 2\mu_2 c(T_2)$.

We build the interpolated tree by starting with T_1, adding S, and adding the shortest path from T_1 to S. The first piece has cost $c(T_1)$ and the second has cost $c(S) \leq 2\mu_2 c(T_2)$. If we have preprocessed the graph to throw away all nodes whose distance to the root is greater than OPT, we can ensure that this last path has cost no more than OPT. We don't know what OPT is, so we'll need to run this entire algorithm n times; on run i we remove the i vertices farthest from the root.

Thus:

$$\begin{aligned}
\text{total cost} &= c(T_1) + c(S) + \text{ cost of shortest path} && (26.1.1)\\
&\leq 2\mu_1 c(T_1) + 2\mu_2 c(T_2) + \text{OPT} && (26.1.2)\\
&\leq 4\text{OPT} + \text{OPT} && (26.1.3)\\
&= 5\text{OPT}. && (26.1.4)
\end{aligned}$$

Thus, the technique of Lagrangian relaxation gives us this algorithm, a 5-approximation to the k-minimum spanning tree problem.

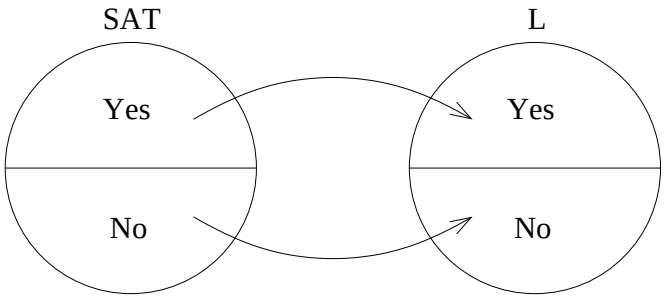
CS880: Approximations Algorithms	
Scribe: Tom Watson	**Lecturer:** Shuchi Chawla
Topic: Inapproximability	**Date:** 4/27/2007

So far in this course, we have been proving upper bounds on the approximation factors achievable for certain NP-hard problems by giving approximation algorithms for them. In this lecture, we shift gears and prove lower bounds for some of these problems, under the assumption that $P \neq NP$. For some problems, essentially matching upper and lower bounds are known, indicating that the approximability properties of these problems are well-understood. We show that (nonmetric) TSP cannot be approximated within any factor. We also show that the Edge Disjoint Paths problem is NP-hard to approximate within a factor of $O(m^{1/2-\epsilon})$ for all $\epsilon > 0$, essentially matching the $O(\sqrt{m})$ algorithm obtained in a previous lecture. Finally, we give a bootstrapping argument showing that if the Clique problem cannot be approximated within some constant factor, then it cannot be approximated within any constant factor.

27.1 Framework

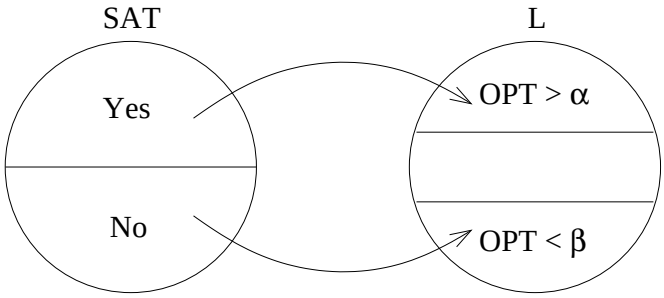
The study of approximation algorithms is mostly focused on NP-hard problems. These problems cannot be solved in polynomial time, assuming $P \neq NP$. Our general goal is to show that under the same assumption, not only can these problems not be solved exactly, but they cannot even be approximated within certain factors in polynomial time. While the complexities of solving NP-complete problems exactly are all polynomially related, we have already seen evidence that these problems can exhibit wildly different approximability properties. For example, we have seen that the Knapsack problem has an FPTAS, indicating that it has a loose grip on its intractability, in some sense. We will shortly see that TSP cannot be approximated within any factor, indicating that it has a very strong grip on its intractability.

We begin by recalling the standard method for showing that a language L is NP-hard. One begins with another NP-hard language, say SAT, and exhibits a polynomial-time mapping reduction from SAT to L. This reduction takes an instance x and produces an instance y such that $x \in$ SAT iff $y \in L$. This can be viewed pictorially as follows.



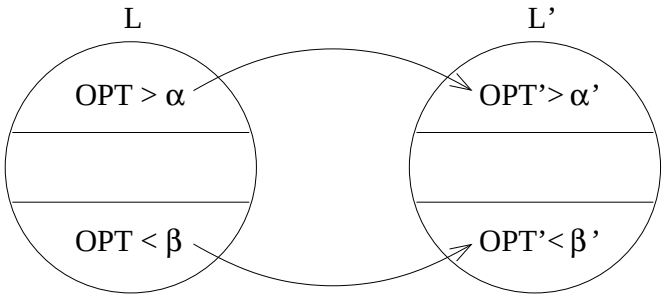
Thus if we had a polynomial-time (exact) algorithm for L then we could compose it with the

reduction to obtain a polynomial-time algorithm for SAT. For inapproximability results, L is an optimization problem, say a maximization problem, rather than a language, and we would like it to be the case that composing a polynomial-time *approximation* algorithm for L with the reduction yields a polynomial time-algorithm for SAT. This goal is achieved with a *gap-introducing reduction*, illustrated as follows.



This type of reduction maps the "yes" instances of SAT to instances of L with optimal objective value greater than α, and it maps the "no" instances of SAT to instances of L with optimal objective value less than β, for some parameters α and β. An α/β-approximation algorithm for L has the property that on instances with $OPT < \beta$ it produces a solution of value less than β (obviously) and on instances with $OPT > \alpha$ it produces a solution of value greater than $\frac{\alpha}{\alpha/\beta} = \beta$. Thus an exact algorithm for SAT can be obtained by applying the mapping reduction followed by the purported approximation algorithm, and testing whether the output has value greater than or less than β. Thus exhibiting such a gap-introducing reduction shows that it is NP-hard to approximate L within a factor of α/β.

With NP-hardness proofs, one doesn't always reduce from SAT; after proving that a problem L is NP-hard, it gets added to the arsenal of problems one can reduce from. Similarly, once we have established an inapproximability result for L, we would like to be able to establish inapproximability results for other problems by means of a reduction from L. However, in this case we don't need to do the work of introducing a gap with our reduction; we merely need to preserve the gap. This is achieved with a *gap-preserving reduction*, illustrated as follows.



It is clear from the picture that if there is a gap-introducing reduction from SAT to L as in the previous figure then composing it with this gap-preserving reduction yields a gap-introducing

reduction from SAT to L', proving that L' is NP-hard to approximate with a factor of α'/β'. We will see examples where L and L' are the same problem but $\alpha'/\beta' > \alpha/\beta$; these reductions can appropriately be called *gap-amplifying reductions*.

We have been assuming that the problems we are dealing with are maximization problems, but the same concepts apply to minimization problems. We now look at some concrete examples of how this framework is employed to prove inapproximability results.

27.2 Traveling Salesperson Problem

Our first example is for the Traveling Salesperson Problem (TSP), in which we are given a set of n nodes and a distance between each pair of nodes, and we seek a minimum-length tour that visits every node exactly once. Recall that in a previous lecture, we obtained a 3/2-approximation algorithm for Metric TSP. How much does the approximability deteriorate when we eliminate the requirement that the distances form a metric? We show that TSP in its full generality is actually inapproximable in a very strong sense.

Theorem 27.2.1 *TSP cannot be approximated within any factor unless $P = NP$.*

Proof: Recall the standard NP-hardness proof for TSP, which is a reduction from the Hamiltonian Tour problem. Given an unweighted graph G, we construct a TSP instance on the same set of nodes, where nodes adjacent in G are at distance 1 and nodes not adjacent in G are at distance ℓ for some value $\ell > 1$. Now if G has a Hamiltonian tour then the TSP instance has a tour of length n (the number of nodes) and otherwise every tour in the TSP instance has length at least $n - 1 + \ell$. Thus this reduction is, in fact, a gap-introducing reduction, proving that TSP is hard to approximate within any factor less than $(n-1+\ell)/n$. Since we can choose ℓ to be as large as we like, this proves the theorem. ■

Note that this result critically uses the fact that the distances are not required to form a metric. The largest we can make ℓ and still be guaranteed that the distances form a metric is $\ell = 2$. This implies that Metric TSP is NP-hard to approximate within any factor less than $(n - 1 + 2)/n = 1 + 1/n$. However, it is known that Metric TSP is NP-hard to approximate within a factor of $1 + \epsilon$ for some constant $\epsilon > 0$.

27.3 Edge Disjoint Paths

Our second example is for the Edge Disjoint Paths (EDP) problem, in which we are given an unweighted graph and k terminal pairs (s_i, t_i) and seek to connect s_i to t_i with edge disjoint paths for as many i as possible. In a previous lecture, we gave an $O(\sqrt{m})$-approximation algorithm for this problem, where m is the number of edges. We now show that that result is essentially tight.

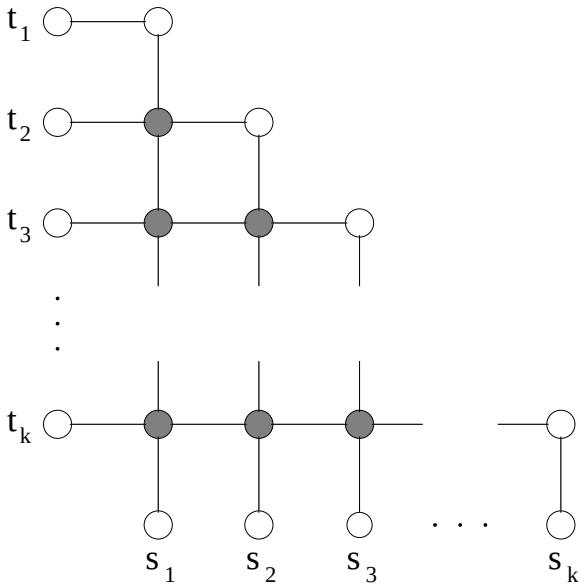
There is an NP-hardness proof for EDP where the hard EDP instances only have $k = 2$ terminal pairs. Since for such instances, the optimum objective value is either 2 or at most 1, we immediately get the following result.

Theorem 27.3.1 *For $k = 2$, EDP is NP-hard to approximate within any factor less than 2.*

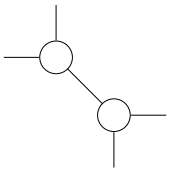
We employ a bootstrapping argument to show that Theorem 27.3.1 implies the following result.

Theorem 27.3.2 *EDP is NP-hard to approximate within a factor of* $O(m^{1/2-\epsilon})$ *for all* $\epsilon > 0$.

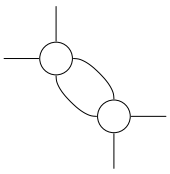
Proof: We give a gap-amplifying reduction from EDP with 2 source-sink pairs to EDP. Suppose we are given a graph H with two terminal pairs (x_1, y_1) and (x_2, y_2). We construct an EDP instance with k terminal pairs, for some k to be determined later, in a graph G having the following general structure.



If we expand each of the filled-in nodes into a single edge like this:

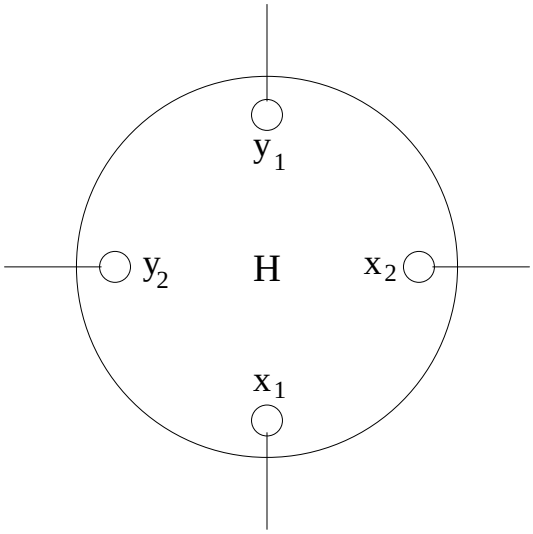


then the optimal objective value is always 1 since for every two (s_i, t_i) pairs, the unique paths connecting s_i to t_i intersect at some filled-in node and would thus have to share the edge there. If we expand each of the filled-in nodes into a pair of edges like this:



then the optimal objective value is always k, since each filled-in node can accomodate both of the (s_i, t_i) pairs whose paths intersect there.

Instead, we expand each filled-in node into a copy of H, as follows.



It follows that if H has edge disjoint paths from x_1 to y_1 and from x_2 to y_2, then each of the filled-in nodes can accommodate both (s_i, t_i) pairs that would like to use it, implying that the optimal objective value for G is k, and otherwise each filled-in node can accomodate at most one (s_i, t_i) pair, implying that the optimal objective value for G is 1. We have succeeded in amplifying the trivial factor 2 gap for H into a factor k gap for G.

Let h denote the number of edges in H. For any given $\epsilon > 0$, we can take $k = h^{1/\epsilon}$ and the reduction still runs in polynomial time. Since the number of edges in G is $m = O(k^2 h) = O(k^{2+\epsilon})$, the inapproximability factor we achieve is $k = \Omega(m^{1/(2+\epsilon)})$, which suffices to prove the theorem. ■

Interestingly, the above proof shows a large gap between the objective values in an EDP instance and the same instance of the fractional version of EDP, which is just a multicommodity flow problem. If H is connected but does not have edge disjoint paths connecting both terminal pairs, then the optimal integral objective value in G is 1, but the optimal fractional objective value is $k/2$ since we can route $1/2$ unit of each commodity without sending more than 1 unit of flow along any edge in G.

27.4 Clique

Our third example is for the Clique problem, in which we are given an unweighted graph and seek a maximum size clique in it. The best known approximation algorithm for this problem achieves an approximation guarantee of $O(n/\log^2 n)$, so even when there is a clique of size $\Omega(n)$, the algorithm only guarantees that it will find a clique of size $\Omega(\log^2 n)$. Indeed, the known inapproximability results come close to matching this bound; the best such result to date shows that for all $\epsilon > 0$, Clique cannot be approximated within a factor of $n^{1-\epsilon}$ unless $NP = ZPP$ [1].

We now prove the following result, which is another example of a bootstrapping argument.

Theorem 27.4.1 *If Clique is NP-hard to approximate within a factor of α, then it is also NP-hard to approximate with a factor of α^2.*

Proof: We give a gap-amplifying reduction from Clique to itself. Suppose we are given a graph $G = (V, E)$. We construct the graph $G' = (V', E')$ as follows. We take $V' = V \times V$, and let $\{(u, v), (w, x)\} \in E'$ iff both of the following conditions hold:

1) $\{u, w\} \in E$ or $u = w$

2) $\{v, x\} \in E$ or $v = x$.

We claim that if the maximum clique size in G is k, then the maximum clique size in G' is k^2. We first show that the maximum clique size in G' is at least k^2 by taking an arbitrary clique $Q \subseteq V$ in G and showing that $Q \times Q \subseteq V'$ is a clique in G'. Let (u, v) and (w, x) be nodes in $Q \times Q$. Since $u, w \in Q$, condition 1 follows immediately. Since $v, x \in Q$, condition 2 follows immediately. Thus $\{(u, v), (w, x)\} \in E'$ and $Q \times Q$ is a clique of size k^2 in G'.

Now we show that the maximum clique size in G' is at most k^2. Consider an arbitrary clique $S \subseteq V \times V$ in G', and let $S_\ell = \{u : (u, v) \in S \text{ for some } v \in V\}$ and $S_r = \{v : (u, v) \in S \text{ for some } u \in V\}$. Now S_ℓ is a clique in G since if $u \neq w$ are nodes in S_ℓ then there exist $v, x \in V$ such that $(u, v), (w, x) \in S$ and thus $\{(u, v), (w, x)\} \in E'$, which implies that $\{u, w\} \in E$ by condition 1. Similarly, S_r is a clique in G. We have $|S_\ell| \cdot |S_r| \geq |S|$, and since $|S_\ell| \leq k$ and $|S_r| \leq k$, we get that $|S| \leq k^2$, as desired. This finishes the proof of our claim.

It follows that if the maximum clique size in G is either at least s or less than s/α, for some value s, then the maximum clique size in G' is either at least s^2 or less than $(s/\alpha)^2 = s^2/\alpha^2$. Thus we have succeeded in amplifying a gap of α to a gap of α^2. ■

In the next lecture, we will see inapproximability results based the PCP Theorem, a deep result in complexity theory.

References

[1] J. Hastad. Clique is Hard to Approximate within n to the power 1-epsilon. In *Acta Mathematica*, 182, 1999, pp. 105-142.

CS880: Approximation Algorithms

Scribe: Tom Watson **Lecturer:** Shuchi Chawla
Topic: Inapproximability **Date:** 5/1/2007

In this lecture we continue our discussion of inapproximability by discussing results based on probabilistically checkable proofs (PCPs). In particular, we show how the PCP Theorem, a deep result in computational complexity, is essentially equivalent to the inapproximability of MAX-3SAT. We also show how a strengthening of the PCP Theorem leads to a tight inapproximability result for MAX-3SAT, and we discuss other issues related to PCPs.

28.1 Probabilistically Checkable Proofs

Recall the template developed in the last lecture for proving inapproximability results. One exhibits a reduction from an NP-complete problem such as SAT to an optimization (say, maximization) problem L in such a way that Yes instances are mapped to instances with optimal objective value at least some α and No instances are mapped to instances with optimal objective value at most some β. The existence of such a reduction proves that L is NP-hard to approximate within any factor better than α/β. In the last lecture we saw examples where standard NP-hardness reductions were used to obtain weak inapproximability results in the above way, and self-reductions were used to magnify the hardness factor achieved. These reductions primarily exploited properties of the problem for which an inapproximability result was desired, i.e. the problem being reduced to. In this lecture, we instead focus on properties of the problem being reduced *from*.

The problem being reduced from is an NP-complete problem, or more generally an arbitrary problem in NP. We now revisit the definition of NP and see how different characterizations of NP can afford additional structure that we can exploit in our gap-introducing reductions. Ideally, we would like the Yes instances to look very different from the No instances in some sense, so that it's easier to introduce a gap in the reduction.

Definition 28.1.1 *A language L is in NP if there exists a polynomial-time verifier V and a constant c such that for an input x of length n, the following two properties are satisfied.*

1) *(Completeness) If $x \in L$ then there exists a witness y of length $O(n^c)$ such that V accepts the pair $\langle x, y \rangle$.*
2) *(Soundness) If $x \notin L$ then for all witnesses y of length $O(n^c)$, V rejects $\langle x, y \rangle$.*

The terminology *completenss* and *soundness* is a carry-over from logic, where completenss refers to the property that all true statements are provable in a given proof system, and soundness refers to the property that no false statements are provable.

Intuitively, there isn't much of a gap between Yes instances and No instances according to this definition, since out of the exponentially many candidate witnesses, a single witness can make or break the membership of x to L. We now alter our notion of proof verification from that of

explicitly verifying the correctness of a proof to that of just randomly spot-checking a few locations of the proof. This is the notion of *probabilistically checkable proofs*, or PCPs for short. This notion leads to a surprising and robust characterization of NP, and the restricted structure of this new type of verifier for NP languages can be exploited in our reductions to prove a wide variety of inapproximability results.

Definition 28.1.2 *A language L is in $PCP_{c,s}(r,q)$ if there exists a polynomial-time verifier V that, given an input x of length n and a purported proof y, runs in time $poly(n)$, uses $r(n)$ bits of randomness, makes $q(n)$ queries to y, and satisfies the following two properties.*

1) *(Completeness) If $x \in L$ then there exists a y such that V accepts the pair $\langle x, y\rangle$ with probability at least c.*

2) *(Soundness) If $x \notin L$ then for all y, V accepts $\langle x, y\rangle$ with probability at most s.*

Note that in principle there is no limit on the size of y. Of course, y should be at most exponentially long, since otherwise the verifier would not have time to write down an index into y. However, we will soon see that for NP, y only needs to be polynomially long. Also note that the size of the alphabet that y is presented in is not specified. Frequently, we assume the binary alphabet is used, but sometimes it is more convenient to work with larger alphabets.

Let us see how PCPs relate to the standard definition of NP presented above. Clearly, a standard NP verifier can be thought of as a PCP verifier that uses no randomness, queries polynomially many bits of the proof, and accepts with probability 1 or 0. Thus, the standard definition of NP can be viewed as follows.

Observation 28.1.3 $NP = PCP_{1,0}(0, poly(n))$.

This characterization has excellent completeness and soundness parameters, but it uses many queries. At the other extreme, we have the following characterization.

Observation 28.1.4 $NP = PCP_{1/2^{O(n)},0}(poly(n), 0)$.

Proof: An NP statement can be verified by randomly guessing a witness and checking that it causes the standard verifier to accept. Conversely, a $PCP_{1/2^{O(n)},0}(poly(n), 0)$ language can be solved in NP by interpreting the witness as a random string that causes the PCP verifier to accept. ■

This characterization is not terribly useful either since the completeness parameter is terrible. Is there some way to interpolate between these two characterizations? As a step in this direction, we can obtain the following characterization, which exploits the NP-completeness property of 3SAT.

Observation 28.1.5 $NP = PCP_{1,1-1/n^{O(1)}}(O(\log n), 3)$.

Proof: The inclusion from right to left follows from Lemma 28.1.7, which we argue below. For the inclusion from left to right, it suffices to demonstrate a $PCP_{1,1-1/m}(O(\log m), 3)$ verifier for 3SAT, where m is the number of clauses in the given formula φ. The verifier can interpret the proof as an assignment the variables of φ. It can select one clause uniformly at random, query the three bits of the proof containing the values of the variables in that clause, and accept if and only if the clause is satisfied by the assignment. If φ is satisfiable then a satisfying assignment causes

the verifier to accept with probability 1. If φ is not satisfiable then all purported proofs encode a non-satisfying assignment, and so with probability at least $1/m$, the three queried bits will not satisfy that clause. ■

The 3SAT verifier described above has very nice r and q parameters, but its soundness is rather high. This is because an unsatisfiable formula may be very close to being satisfiable, in the sense that some assignment satisfies all but one of its clauses. Indeed, recalling the standard mapping reduction from an arbitrary NP language to 3SAT, one can see that the formulas produced are encodings of nondeterministic computations, and there is a single clause expressing that the computation must be *accepting*. It is always possible to satisfy all clauses except this last one simply by encoding a valid nonaccepting computation.

Driving the soundness down to a constant value without spoiling the r and q parameters requires working with encodings that are robust in the sense that any error in the proof must be "spread out" so that it is caught with constant probability. In fact, this can be achieved; this is the content of the famous PCP Theorem, due to [1] and [2]. The proof of this celebrated result is very involved, so we omit it.

Theorem 28.1.6 (The PCP Theorem) $NP = PCP_{1,1/2}(O(\log n), O(1))$.

The difficult direction of the PCP Theorem is the inclusion from left right. The other inclusion follows from the following lemma. This lemma also finishes the proof of Observation 28.1.5. Finally, this lemma justifies the claim we made earlier that restricting our attention to polynomial-size proofs for NP is no loss of generality.

Lemma 28.1.7 $PCP_{c,s}(r(n), q(n)) \subseteq NTIME(2^{O(r(n)+q(n))}n^{O(1)})$.

Proof: Consider a PCP where the verifier uses $r(n)$ random bits and makes $q(n)$ queries. For each random string, the number of locations of the proof that could ever get queried is at most $2^{q(n)}$ (due to the possibly adaptive nature of the verifier). Thus over all proofs and all random strings, there are at most $2^{r(n)+q(n)}$ locations that ever get queried (for a given input). A nondeterministic machine can guess the addresses and answers to these queries in time $2^{r(n)+q(n)}n^{O(1)}$. It can then run over all random strings, simulate the verifier for each one (looking up the answers to queries in the guessed table), and explicitly compute the probability of acceptance of the verifier for the guessed proof. This takes time $2^{r(n)}n^{O(1)}$. If the probability of acceptance is at least c then the machine should accept, and otherwise it should reject. ■

28.2 Inapproximability of MAX-3SAT

While the PCP Theorem is quite interesting and profound in its own right, some of its major customers are inapproximability results. We now show that the PCP Theorem is actually equivalent in some sense to the inapproximability of MAX-3SAT.

Theorem 28.2.1 *The following are equivalent.*

1) $NP \subseteq PCP_{1,1/2}(O(\log n), O(1))$.

2) There is a mapping reduction from 3SAT to 3SAT that maps satisfiable formulas to satisfiable

formulas and maps unsatisfiable formulas to formulas for which no assignment satisfies more than a $1-\epsilon$ fraction of the clauses, for some constant $\epsilon > 0$.

Proof: We first prove that statement 2 implies statement 1. Suppose the reduction specified in statement 2 exists. We can design a $PCP_{1,1/2}(O(\log n), O(1))$ verifier for 3SAT by having the verifier perform the reduction on its input formula φ to obtain a formula ψ and assume the proof is an assignment to ψ. The verifier can randomly pick a clause of ψ and query the 3 bits of the proof specifying the values of the variables in that clause, as in Observation 28.1.5.

If φ is satisfiable then so is ψ and thus the verifier's test passes with probability 1 in this case. If φ is not satisfiable then every assignment to the variables of ψ, and in particular the assignment given by any proof, violates at least an ϵ fraction of the clauses. Thus the verifier's test passes with probability at most $1-\epsilon$. The soundness is not yet $1/2$, but we can pick another clause uniformly at random, independently of the first clause, and it will be satisfied by the proof's assignment with probability at most $1-\epsilon$, and so if the verifier only accepts if both tests pass, then the soundness is driven down to at most $(1-\epsilon)^2$. Repeating this a constant number of times drives the soundness down to $1/2$. The reason this soundness reduction strategy doesn't work in Observation 28.1.5 is that there, the soundness is $1-o(1)$ to begin with and so a super-constant number of repetitions would be needed to drive it down to $1/2$, and this would result in a super-logarithmic amount of randomness and a super-constant number of queries to the proof.

We now prove that statement 1 implies statement 2. Suppose that $NP \subseteq PCP_{1,1/2}(O(\log n), O(1))$; then in particular, there is a $PCP_{1,1/2}(r, q)$ verifier V for 3SAT where $r = O(\log n)$ and q is a constant. We assume that this verifier chooses its query locations nonadaptively; this is no loss of generality since an adaptive verifier that queries q bits of the proof can query at most $2^q - 1$ different locations for a given choice of random string, over all possible answers to those queries, and can thus be simulated by a nonadaptive verifier that makes at most $2^q - 1$ queries, which is still a constant number.

We design a mapping reduction from 3SAT to 3SAT satisfying the property of statement 2. Given a formula φ, we generate a formula ψ with a variable for each of the polynomially many bits of the proof V could ever query on input φ. We run over all random strings R of length r (of which there are only polynomially many), determine which q bits of the proof (variables of ψ) V queries for that choice of R, and determine which of the 2^q settings of those variables cause V to accept. This function on q variables can be expressed in CNF using at most 2^q clauses of size q. The conjunction of these clauses over all choices of R is a CNF formula with at most $2^q 2^r$ clauses. Now a clause $(\ell_1 \vee \cdots \vee \ell_q)$ where the ℓ_i's are literals can be expressed as

$$(\ell_1 \vee \ell_2 \vee z_2) \wedge (\overline{z_2} \vee \ell_3 \vee z_3) \wedge (\overline{z_3} \vee \ell_4 \vee z_4) \wedge \cdots \wedge (\overline{z_{q-3}} \vee \ell_{q-2} \vee z_{q-2}) \wedge (\overline{z_{q-2}} \vee \ell_{q-1} \vee \ell_q),$$

where $z_2, \ldots, z_{q-2}$ are new variables unique to that clause. It can be easily verified that a satisfying assignment to this conjunction yields a satisfying assignment to $(\ell_1 \vee \cdots \vee \ell_q)$ and vice versa. Expanding each clause in our CNF formula in this manner yields a 3CNF formula with at most $q2^q 2^r = poly(n)$ clauses. If φ is satisfiable then ψ is satisfiable by setting the variables according to a proof that makes V accept with probability 1. If φ is not satisfiable, then each assignment to the variables of ψ yields a proof that causes V to accept for at most half of the random strings

R. Since the block of clauses corresponding to R can only satisfied by the assignment if V accepts this proof for that choice of R, it follows that at least half of these blocks must have at least one unsatisfied clause under this arbitrary assignment. Therefore the fraction of unsatisfied clauses is always at least $1/2q2^q$. We can take ϵ to be this value, and the theorem is proved. ■

Corollary 28.2.2 *There is no PTAS for MAX-3SAT unless $P = NP$.*

Proof: Statement 1 in Theorem 28.2.1 holds by the PCP Theorem and thus the mapping reduction and constant $\epsilon > 0$ given by statement 2 exists. A $(1 - \epsilon')$-approximation algorithm for MAX-3SAT for any $\epsilon' < \epsilon$ could be combined with the reduction in the standard way to arrive at a polynomial-time (exact) algorithm for 3SAT. Given a satisfiable formula, the reduction would produce a satisfiable formula, and then the MAX-3SAT algorithm would find an assignment that satisfies at least a $1-\epsilon' > 1-\epsilon$ fraction of the clauses. Given an unsatisfiable formula, the reduction would produce a formula for which no assignment satisfies more than a $1-\epsilon$ fraction of the clauses. Thus testing whether the assignment produced by the MAX-3SAT algorithm satisfies more than a $1 - \epsilon$ fraction of the clauses would allow us to distinguish between the Yes and No instances of 3SAT. ■

28.3 Other PCP Theorems

The inapproximability factor $1 - \epsilon$ given in Theorem 28.2.1 is very close to 1 and is thus not very strong. In this section we consider other results related to PCPs that can be used to prove stronger inapproximability results.

28.3.1 Repetition of PCPs

We begin by considering the soundness parameter. Typically, the lower the soundness and the lower the number of queries, the stronger the inapproximability results one can prove. Having the prover provide k separate copies of the proof and running the verifier independently on each proof drives the soundness down to s^k, since if the input is not in the language, then each of the k sections of the proof defines some proof that makes the original verifier accept with probability at most s. This is called *sequential repetition*. While it has the desired effect on the soundness parameter, it increases the number of queries by a factor of k.

Another approach for reducing the soundness is called *parallel repetition*. In this approach the new verifier runs k copies (repetitions) of the old verifier simultaneously, using k independent random strings, but it waits for each repetition to formulate its first query and then sends all the queries to the prover at once. The prover responds to these k queries at once (and thus requires a larger alphabet), and the simulation of the k repetitions continues until they form their second queries, all of which are sent to the prover at once, and so on. If the original verifier makes q queries and the proof is written in the alphabet R, then the verifier for k parallel repetitions makes q queries and the proof is writen in the alphabet $R^{(k)} = R \times \cdots \times R$. Often, the increase in alphabet size is an acceptable tradeoff for keeping the number of queries at its original value.

But what happens to the soundness parameter? Unfortunately, it does not decrease to s^k in general. The basic reason for this can be intuited by considering the case $k = 2$. The first repetition assumes

it is being run with some proof y_1 written in alphabet R and the second repetition assumes it is being run with some proof y_2 written in alphabet R. The actual proof y has of an entry for each pair consisting of an entry in y_1 and an entry in y_2. That is, if the ith entry of y_1 is supposed to be some symbol $a \in R$ and the jth entry of y_2 is supposed to be $b \in R$, then the (i, j) entry of y is supposed to be the symbol $(a, b) \in R \times R$. However, the entries of y might not be consistent with each other, in the sense that the (i, j) entry of y may contain (a, b) but the (i, j') entry for some $j' \neq j$ may contain (c, d) where $c \neq a$. Thus no unique proof y_1 can be extracted from the proof y in this case; the proof as seen by the first repetition can "adapt" depending on what the second repetition is doing. The proof can exploit this to get an overall probability of acceptance greater than s^2. We give a concrete example of this in the next subsection, in a slightly different PCP setting.

28.3.2 2-Prover 1-Round Games

In the 2-Prover 1-Round Game model, the verifier interacts with two provers who may agree on any strategy before the protocol begins but may not communicate during the protocol. The verifier flips some coins, asks one question (that depends on the outcome of the coin flips) to each prover, and decides whether to accept based on their responses. Each prover's response is a function only of its query and the common input, and not the other prover's query. Equivalently, we can think of this model as a PCP where the proof is divided into two parts — the first part is written in some alphabet R_1 and corresponds to the first prover's responses to the different queries the verifier could ask it, and the second part is written in some alphabet R_2 and corresponds to the second prover's responses to the different queries the verifier could ask it.

Definition 28.3.1 *A language L is in $2P1R_{c,s}(r)$ if there exists a polynomial-time verifier V that, given an input x of length n and access to two noncommunicating provers, runs in time $poly(n)$, uses $r(n)$ bits of randomness, makes one query to each prover, and satisfies the following two properties.*

1) *(Completeness) If $x \in L$ then there exist provers such that V accepts x with probability at least c.*

2) *(Soundness) If $x \notin L$ then for all provers, V accepts x with probability at most s.*

Although this model may seem quite restricted since the verifier may only ask one question of each prover, it turns out that the verifier's ability to cross-check the noncommunicating provers' answers restricts their ability to cheat and allows this model to capture all of NP. This is illustrated in the following result.

Theorem 28.3.2 *$NP = 2P1R_{1,1-\epsilon'}(O(\log n))$ for some constant $\epsilon' > 0$.*

Proof: The inclusion from right to left is similar to Lemma 28.1.7 and is left as an exercise. For the other inclusion, by NP-completeness it suffices to show that $3\text{SAT} \in 2P1R_{1,1-\epsilon'}(O(\log n))$. Let the input formula be φ. By the PCP Theorem and Theorem 28.2.1, there is reduction that maps φ to a 3CNF formula ψ such that if φ is satisfiable then so is ψ, and if φ is unsatisfiable then no assignment satisfies more than a $1 - \epsilon$ fraction of the clauses of ψ, for some constant $\epsilon > 0$. The verifier expects prover 1 to have an assignment to the variables of ψ, and we expect prover 2 to have

for each clause a satisfying assignment to the variables in that clause. Thus if ψ has ℓ variables and m clauses then the first proof is written in the binary alphabet and is of length ℓ, and the second proof is written in an alphabet of size 7 and is of length m. The verifier selects a clause of ψ and one of the variables in that clause uniformly at random, queries that variable and that clause to provers 1 and 2 respectively, and accepts if and only if the two responses agree on the value of the chosen variable.

Now if φ is satisfiable then so is ψ, and the provers can give answers consistent with some satisfying assignment to ψ, and the verifier will accept with probability 1. If φ is not satisfiable then with probability at least ϵ, the verifier will choose a clause of ψ that is not satisfied by the assignment that prover 1 has written down. Since prover 2's response for this clause satisfies it, there must be at least one variable on which prover 2's response disagrees with prover 1's assignment. Conditioned on picking a clause that is not satisfied by prover 1's assignment, the verifier catches the inconsistency with probability at least 1/3. It follows that the soundness of this $2P1R$ system is at most $1 - \epsilon/3$. ■

The characterization given by Theorem 28.3.2 is frequently used in inapproximability results.

We now return to the question of parallel repetition. In the 2P1R setting, we can perform k parallel repetitions in the same way as with PCPs. This increases prover 1's alphabet from R_1 to $R_1^{(k)}$ and prover 2's alphabet from R_2 to $R_2^{(k)}$ but does not increase the number of queries. In this setting, we can give a simple concrete example to show that k parallel repetitions does not drive the soundness down to s^k in general.

Consider the (silly) protocol where the verifier selects two bits r_1, r_2 uniformly at random and sends r_1 to prover 1 and r_2 to prover 2. Each prover is supposed to respond with a pair (i, r) indicating that prover i was sent bit r. The verifier accepts if and only if the two responses agree and are both correct. Then for any outcome that causes the verifier to accept, say both provers respond with $(1, r_1)$, since flipping r_1 cannot change prover 2's response, the latter outcome leads to rejectance. Thus at most half of the outcomes lead to acceptance and the soundness of this protocol is at most 1/2. (It is also straightforward to see that the soundness is at least 1/2.)

However, when we take $k = 2$ parallel repetitions, the soundness stays at 1/2. In this case, the verifier picks four bits r_1, r_2, s_1, s_2 uniformly at random and sends (r_1, s_1) to prover 1 and (r_2, s_2) to prover 2. Each prover responds with a tuple (i_1, r, i_2, s) and the verifier accepts if and only if the two responses agree and $r_{i_1} = r$ and $s_{i_2} = s$. However, the provers can get the verifier to accept with probability 1/2 in the following way. Prover 1 responds with $(1, r_1, 2, r_1)$ and prover 2 responds with $(1, s_2, 2, s_2)$. They agree and are both correct if and only if $r_1 = s_2$, which happens with probability 1/2. In this case we can see that prover 1 is using information about repetition 1 in its response to repetition 2, and prover 2 is using information about repetition 2 in its response to prover 1. In this way, they are able to cheat the verifier.

Despite this difficulty, [4] showed that the soundness does go down exponentially with k. The proof of this result is involved, so we omit it.

Theorem 28.3.3 *For every 2P1R system with soundness s, k parallel repetitions have soundness at most $(s')^k$ where s' is a constant depending only on s and the sizes of the alphabets used by the provers.*

28.3.3 Hastad's 3-Bit PCP

We now move from the issue of reducing the soundness parameter of a PCP to that of reducing the number of queries it makes. The following result is due to [3]. Again, we omit the proof.

Theorem 28.3.4 (Hastad's 3-Bit PCP) *$NP = PCP_{1-\epsilon,1/2+\epsilon}(O(\log n), 3)$ for all constants $\epsilon > 0$. Moreover, the PCP verifier for NP operates by choosing three positions i_1, i_2, i_3 of the proof and a bit b according to some distribution, reading the three bit values $y_{i_1}, y_{i_2}, y_{i_3}$, and accepting if and only if $y_{i_1} \oplus y_{i_2} \oplus y_{i_3} = b$.*

Recall that we obtained a 3-bit PCP characterization of NP in Observation 28.1.5. However, the present result has a soundness parameter that is close to 1/2, which is much better than the $1 - 1/n^{O(1)}$ soundness parameter in that simple protocol.

The linear nature of the tests run by Hastad's 3-bit PCP intimately connect it to the following constraint satisfaction problem.

Definition 28.3.5 (MAX-3LIN) *Given a system of linear equality constraints over $GF(2)$ where each constraint involves exactly three variables, find a solution that maximizes the number of satisfied constraints.*

Theorem 28.3.6 *The following are equivalent.*

1) *Theorem 28.3.4.*
2) *For all constants $\epsilon > 0$ there is a reduction from 3SAT (or any NP-complete problem) to MAX-3LIN that maps satisfiable formulas to MAX-3LIN instances where a $1 - \epsilon$ fraction of the constraints can be satisfied simultaneously, and maps unsatisfiable formulas to MAX-3LIN instances where no assignment satisfies more than a $1/2 + \epsilon$ fraction of the constraints.*

Proof: The proof is similar to Theorem 28.2.1. We first prove that statement 2 implies statement 1. Fix $\epsilon > 0$, and suppose the reduction specified in statement 2 exists. We can design a $PCP_{1-\epsilon,1/2+\epsilon}(O(\log n), 3)$ verifier for 3SAT by having the verifier perform the reduction on its input formula φ to obtain a system of equations over $GF(2)$. The proof is assumed to contain an assignment to the variables of this system. The verifier picks one of the constraints uniformly at random, queries the 3 bits of the proof specifying the values of the variables in that constraint, and accepts if and only if the constraint is satisfied.

If φ is satisfiable then there is an assignment for the system of equations that satisfies at least a $1 - \epsilon$ fraction of them. The prover can provide this assignment, and the verifier will accept with probability at least $1 - \epsilon$. If φ is not satisfiable then every assignment to the variables of the system, and in particular the assignment given by any proof, violates at least a $1/2 - \epsilon$ fraction of the constraints. Thus the verifier's test passes with probability at most $1/2 + \epsilon$.

We now prove that statement 1 implies statement 2. Fix $\epsilon > 0$, and suppose that the characterization of NP given in Theorem 28.3.4 holds. Then in particular, there is a $PCP_{1-\epsilon,1/2+\epsilon}(O(\log n), 3)$ verifier V for 3SAT. This verifier chooses its three query locations nonadaptively.

We design a reduction from 3SAT to MAX-3LIN satisfying the property of statement 2. Given a formula φ, we generate a system of linear equations over $GF(2)$ with a variable for each of the

polynomially many bits of the proof that V could ever query on input φ. We run over all random strings R of length $O(\log n)$, determine which three bits of the proof V queries for that choice of R, and determine which of the eight settings of those variables cause V to accept. Since V's test is linear, this function can be expressed as a MAX-3LIN constraint. The collection of these constraints over all random strings R forms our MAX-3LIN instance.

If φ is satisfiable then there is a proof such that at least a $1-\epsilon$ fraction of the choices of R lead to acceptance. Thus setting the variables of our system according to this proof satisfies at least a $1-\epsilon$ fraction of the constraints. If φ is not satisfiable, then each assignment to the variables of the system yields a proof that causes V to accept for at most a $1/2+\epsilon$ fraction of the random strings R. Thus no assignment to the variables of the system satisfies more than a $1/2+\epsilon$ fraction of the constraints. ■

Theorem 28.3.6 reveals why Hastad's 3-bit PCP does not have a completeness parameter of 1 — if it did, then we could reduce all of NP to the problem of determining whether a system of linear equations over $GF(2)$ is consistent. This can be solved in polynomial time by Gaussian elimination, so we would have $P = NP$.

With Hastad's 3-bit PCP and Theorem 28.3.6 in hand, we can now obtain a strong inapproximability result for MAX-3SAT. Since there exists a 7/8-approximation algorithm for MAX-3SAT, the following result essentially closes the case on the approximability of MAX-3SAT.

Corollary 28.3.7 *For all constants $\epsilon' > 0$, MAX-3SAT is NP-hard to approximate within a factor of $7/8+\epsilon'$.*

Proof: By Hastad's 3-bit PCP and Theorem 28.3.6, a reduction of the type given by statement 2 exists for all $\epsilon > 0$. We give a gap-preserving reduction from MAX-3LIN to MAX-3SAT. A linear constraint on three variables over $GF(2)$ is a function that happens to evaluate to 1 for exactly four of the eight settings of its variables, so the CNF representation of this function is a collection of four clauses with three variables each. The conjunction of all these clauses over all the constraints of the given MAX-3LIN instance forms our MAX-3SAT instance. Note that the two instances have the same set of variables.

Assignments that satisfy at least a $1-\epsilon$ fraction of the linear constraints satisfy at least a $1-\epsilon$ fraction of the clauses of our 3CNF formula, since they satisfy all the clauses in blocks corresponding to satisfied constraints. Assignments that violate at least a $1/2-\epsilon$ fraction of the linear constraints violate at least a $(1/2-\epsilon)/4 = 1/8-\epsilon/4$ fraction of the clauses of our 3CNF formula, since they violate at least one of the four clauses in each of the blocks corresponding to violated constraints. Given a 3CNF formula in which at least a $1-\epsilon$ fraction of the clauses can be satisfied, a $(7/8+\epsilon')$-approximation algorithm for MAX-3SAT would find an assignment satisfying at least a $(1-\epsilon)(7/8+\epsilon') = 7/8 + (\epsilon' - \epsilon(7/8+\epsilon'))$ fraction of the clauses. This fraction is greater than $1-(1/8-\epsilon/4)$ provided we choose ϵ small enough. Thus a $(7/8+\epsilon')$-approximation algorithm for MAX-3SAT would allow us to distinguish between instances of MAX-3LIN where a $1-\epsilon$ fraction of the constraints can be satisfied and those where at most a $1/2+\epsilon$ fraction can be satisfied, for some constant $\epsilon > 0$. ■

In the next lecture, we will see how to use Hastad's 3-bit PCP to show that it is NP-hard to approximate Vertex Cover within any constant factor less than 7/6. We will also see that Set

Cover cannot be approximated within a factor of $b \cdot \log n$ for some constant b, under an assumption slightly stronger than $P \neq NP$.

References

[1] S. Arora, C. Lund, R. Motwani, M. Sudan, M. Szegedy. Proof Verification and the Hardness of Approximation Problems. In *Journal of the ACM*, 45(2), pp. 501-555, 1998.

[2] A. Arora, S. Safra. Probabilistic Checking of Proofs: A New Characterization of NP. In *Journal of the ACM*, 45(1), pp. 70-122, 1998.

[3] J. Hastad. Some Optimal Inapproximability Results. In *Journal of the ACM*, 48(4), pp. 798-859, 2001.

[4] R. Raz. A Parallel Repetition Theorem. In *SIAM Journal on Computing*, 27(3), pp. 763-803, 1998.

CS880: Approximations Algorithms

Scribe: Chi Man Liu **Lecturer:** Shuchi Chawla

Topic: Inapproximability: Vertex Cover and Set Cover **Date:** 5/3/2007

In the previous lecture we introduced probabilistically checkable proofs (PCPs) and saw how they could be used to obtain a tight inapproximability result for MAX-3SAT. We also introduced Hastad's 3-bit PCP, a very useful tool for proving inapproximability results. In this lecture we apply Hastad's 3-bit PCP to show that approximating Vertex Cover within any constant factor less than 7/6 is NP-hard. We also show that Set Cover cannot be approximated within a factor of $\beta \cdot \log n$ for some constant β, unless NP $\subseteq$ DTIME($n^{\log \log n}$).

29.1 Vertex Cover

Last time we introduced Hastad's 3-bit PCP and out of it we obtained the following result.

Theorem 29.1.1 *For all constants $\epsilon > 0$ there is a reduction from* 3SAT *(or any* NP*-complete problem) to* MAX-3LIN *that maps satisfiable formulas to* MAX-3LIN *instances where a $1-\epsilon$ fraction of the constraints can be satisfied simultaneously, and maps unsatisfiable formulas to* MAX-3LIN *instances where no assignment satisfies more than a $1/2+\epsilon$ fraction of the constraints.*

We used Theorem 29.1.1 to show that it is NP-hard to approximate MAX-3SAT within any constant factor greater than 7/8. In the following, we show that approximating Vertex Cover within any constant factor less than 7/6 is NP-hard using a similar technique.

Theorem 29.1.2 *For all constants $\epsilon > 0$, Vertex Cover is* NP*-hard to approximate within a factor of $7/6 - \epsilon$.*

Proof: Our goal is to construct a gap-preserving reduction from MAX-3LIN to Vertex Cover. Given a MAX-3LIN instance with m constraints, we want to construct a graph G. For each linear constraint $x_p \oplus x_q \oplus x_r = b$, we add to G a clique with 4 vertices, where each vertex represents a setting of (x_p, x_q, x_r) satisfying the constraint. Then, for any two vertices, if their corresponding settings of variables conflict with each other, we add an edge between them. We now show that this polynomial-time reduction is gap-preserving. Suppose that the MAX-3LIN instance has an assignment satisfying at least a $1-\epsilon$ fraction of the constraints. For each satisfied constraint, pick the vertex in G that corresponds to its setting of variables in this assignment. These vertices form an independent set since they all come from different cliques, and all settings of variables agree. This independent set has size at least $(1-\epsilon)m$. There are $4m$ vertices in G, hence G has a vertex cover of size at most $(3+\epsilon)m$. Now suppose that no assignment satisfies more than a $1/2+\epsilon$ fraction of constraints. We claim that any independent set of G has at most $(1/2+\epsilon)m$ vertices. From this it follows that G has minimum vertex cover of size at least $(7/2-\epsilon)m$, and we have achieved a gap of $7/6-\epsilon$. It still remains to prove the claim. Let S be an independent set of G with k vertices. The vertices in S correspond to distinct linear constraints as they must lie in different cliques. Moreover, these vertices represent settings of variables that have no conflict. Thus, we can augment the settings to a total assignment that satisfies those k constraints. We finish the

argument by setting $k = (1/2 + \epsilon)m$. ■

29.2 Set Cover

In this section, we show that Set Cover is unlikely to have a sublogarithmic approximation. Formally, we have the following theorem.

Theorem 29.2.1 *For some constant β, approximating Set Cover within a factor of $\beta \cdot \log n$ is* NP*-hard unless* NP $\subseteq$ DTIME($n^{\log \log n}$).

To prove this theorem, we consider the 2-Prover 1-Round system for 3SAT in the previous lecture. Given a 3CNF with m clauses, the verifier selects a clause and one of the variables in that clause uniformly at random, queries that clause and that variable to provers 1 and 2 respectively, and accepts if and only if the two responses agree on the value of the chosen variable. This system uses $O(\log m)$ random bits, has completeness 1 and soundness $1 - \epsilon'$ for some cosntant $\epsilon' > 0$. We take k parallel repetitions to reduce the soundness to α^k for some constant $\alpha < 1$, while increasing the amount of randomness to $O(k \log m)$. We will determine the value of k later. Now, a query to prover 1 is a k-tuple of clauses, while a query to prover 2 is a k-tuple of variables. The numbers of different queries to provers 1 and 2 are m^k and n^k respectively.

We introduce the Label Cover problem in the next section. It will soon be clear that finding optimal provers (ones that maximize the accepting probability) in 2P1R systems can be reduced to solving Label Cover instances.

29.2.1 Label Cover

Consider a bipartite graph $G = (Q_1, Q_2; E)$ with $|Q_1| = m^k$ and $|Q_2| = n^k$. Each vertex in Q_1 (resp. Q_2) represents a possible query to prover 1 (resp. prover 2). There is an edge between $u \in Q_1$ and $v \in Q_2$ if and only if the verifier can ask prover 1 query u and prover 2 query v simultaneously for some sequence of random bits. To each vertex $w \in Q_1 \cup Q_2$ we associate a *label set* L_w containing all correct answers to the query w. For every edge $e = (u, v)$, let R_e be the relation containing all pairs (a, b) ($a \in L_u$, $b \in L_v$) such that (a, b) is an accepting answer pair to query pair (u, v). Our goal is to pick an answer from each label set so that the probability of acceptance is maximized. In other words, we want to assign to each vertex w a label $a_w \in L_w$ such that the following quantity is maximized:

$$\frac{|\{e \mid e = (u, v) \in E, (a_u, a_v) \in R_e\}|}{|E|}.$$

We can generalize the above problem in the following way. Let L be a set of labels. For each vertex w, the label set L_w is a nonempty subset of L. For each edge e, the relation R_e is a subset of $L \times L$. This generalized problem is known as *Label Cover*.

The 2P1R system for 3SAT discussed above has completeness 1 and soundness α^k. Hence, G either has an assignment satisfying all edges, or has no assignment satisfying more than an α^k fraction of the edges. Note that $|L| = 2^{O(k)}$ and G has $O(n^k)$ vertices, where n is the number of literals in the formula. Moreover, we can assume that G satisfies certain properties that we will make use of in the reduction to Set Cover. This is formalized in the following lemma.

Lemma 29.2.2 *There exists a constant $\alpha < 1$, such that for every integer $k > 0$, there exists an infinite family of instances of the Label Cover problem indexed by n, such that the following holds:*

1. *The bipartite Label Cover graph is regular and has an equal number of vertices on both sides. Moreover, each side has $O(n^k)$ vertices.*
2. *For each edge $e = (u, v)$, the relation R_e is a projection from L_u to L_v. That is, for any $a \in L_u$, there exists a unique $b \in L_v$ satisfying $(a, b) \in R_e$.*
3. *Either there is an assignment satisfying all the edges (YES instances), or no assignment satisfies more than an α^k fraction of the edges (NO instances).*
4. *It is not possible to distinguish between YES and NO instances in polynomial time, unless* NP $\subseteq$ DTIME(n^k).

Proof: We form a Label Cover instance based on 3SAT(5) instead of a MAX-3SAT instance. (3SAT(5) is a special case of 3SAT where each variable in the formula occurs in exactly 5 clauses.) We use the fact that 3SAT(d) has gap instances just like 3SAT, for any $d \geq 5$. Following the above construction, we can reduce any 3SAT(5) instance with n variables to a Label Cover instance G satisfying property (3). For property (1), observe that $|Q_1| = (5n/3)^k$ and $|Q_2| = n^k$. Moreover, every vertex in Q_1 has degree 3^k while every vertex in Q_2 has degree 5^k. We create 3^k copies of Q_1 and 5^k copies of Q_2, and add edges appropriately. This gives us a (15^k)-regular graph with $(5n)^k$ vertices on each side which is essentially equivalent to G. Finally, property (2) follows from that any correct answer given by prover 1 induces a unique accepting answer for prover 2, namely the setting of variables that agrees with prover 1's answer.

Suppose that this problem can be solved in time poly(n^k). Then, by our reduction, 3SAT(5) can also be solved in time poly(n^k). Since 3SAT(5) is NP-hard, we have NP $\subseteq$ DTIME(n^k). ■

29.2.2 Reduction from Label Cover to Set Cover

Given a 3SAT(5) instance with n variables, we can reduce it to a regular Label Cover gap instance G with $O(n^k)$ vertices. We are going to show how to reduce regular Label Cover instances with edge relations being projections to Set Cover instances. Our strategy is to associate each edge $e = (u, v)$ in G with a Set Cover instance I_e such that picking an accepting pair $(a, b) \in L_u \times L_v$ would correspond to covering I_e with just a few sets, and vice versa. In other words, we would like to have a Set Cover instance in which the "coordinated" solutions have very few sets, whereas "uncoordinated" solutions use a much larger number of sets. For our purpose, we want this gap to be at least logarithmic.

Let U be the universe of elements and t be some integer parameter. For $i = 1, \ldots, t$, we construct a set S_i by picking each element of U independently with probability $1/2$. The collection of sets in the instance is $\mathcal{C} = \{S_1, \ldots, S_t, \bar{S}_1, \ldots, \bar{S}_t\}$, where $\bar{S}$ denotes the complement of S, i.e. $\bar{S} = U \backslash S$. We claim that the instance $I = (U, \mathcal{C})$ has the property we want. Let $\mathcal{S} \subseteq \mathcal{C}$ be a set cover. It is clear that if we pick both S_i and $\bar{S}_i$ for some i, we get a set cover with size 2. Otherwise, the (expected) size of $\mathcal{S}$ would be at least logarithmic in $|U|$: The expected number of elements covered by the first set in $\mathcal{S}$ is $|U|/2$. The second set covers $|U|/4$ of the uncovered elements on average

since the sets are constructed by picking elements independently and uniformly at random. This rough argument suggests that covering all elements in U would require $\Theta(\log |U|)$ sets.

We can now use I as a building block to transform our Label Cover instance G to a Set Cover instance $\tilde{I}$. For each edge $e = (u, v)$, we construct an instance $I_e = (U_e, \mathcal{C}_e)$ equivalent to I with $|U_e| = n^k$. We take $t = |L_v| = 2^k$, so that each label $b \in L_v$ gets mapped to a unique set $S_{e,b} \in \mathcal{C}_e$. Then, $\mathcal{C}_e = \{S_{e,b} \mid b \in L_v\} \cup \{\bar{S}_{e,b} \mid b \in L_v\}$ has size 2^{k+1}. Next, we merge all the "edge" instances to get a large instance $\tilde{I} = (\tilde{U}, \tilde{\mathcal{C}})$ for the whole graph. Let $\tilde{U} = \bigcup_{e \in E} U_e$. For each $v \in Q_2$ and $b \in L_v$, let

$$\tilde{S}_{v,b} = \bigcup_{e \text{ incident on } v} S_{e,b}.$$

For each $u \in Q_1$ and $a \in L_u$, let

$$\tilde{S}_{u,a} = \bigcup_{e \text{ incident on } u} \bar{S}_{e,R_e(a)},$$

where $R_e(a)$ denotes the unique $b \in L_v$ satisfying $(a, b) \in R_e$. Let $\tilde{\mathcal{C}}$ be the collection of all these sets. We have $|\tilde{U}| = n^{O(k)}$ and $|\tilde{\mathcal{C}}| = n^{O(k)}$. Suppose that uncoordinated solutions to "edge" instance I_e have size at least $\ell = \Theta(\log |U_e|) = \Theta(k \cdot \log n)$. We will see that our reduction works if we pick $k = O(\log \log n)$. Then, intuitively, if we had a "good" approximation algorithm for Set Cover, we could use it to solve our Label Cover problem (and thus 3SAT(5)) exactly in $n^{O(\log \log n)}$ time. Hence, Theorem 29.2.1 follows. In the following, we give a formal proof for the correctness of our reduction.

Suppose that G has an assignment satisfying all edges. By picking the sets in $\tilde{\mathcal{C}}$ corresponding to this assignment, we are able to cover $\tilde{U}$ with only $|Q_1| + |Q_2| = O(n^k)$ sets. It remains to show that if G does not have any good solution, covering $\tilde{U}$ would require $\Omega(\ell n^k)$ sets.

Claim 29.2.3 *If every assignment to G satisfies at most an α^k fraction of edges, then every set cover of $\tilde{U}$ has size at least $\Omega(\ell n^k)$.*

Proof: We start with a minimum set cover $\mathcal{C}^*$. Let G_1 (resp. G_2) be the set of vertices in Q_1 (resp. Q_2) that have less than $\ell/2$ sets in $\mathcal{C}^*$. Let $B_1 = Q_1 \backslash G_1$ and $B_2 = Q_2 \backslash G_2$. Let $e = (u, v)$ be an edge such that $u \in G_1$ and $v \in G_2$. We pick an assignment to G in the following way. For every $u \in Q_1$, let A_u denote the set of labels a in L_u such that $\tilde{S}_{u,a} \in \mathcal{C}^*$. Pick an answer uniformly at random from A_u. Likewise, for every $v \in Q_2$, pick an answer uniformly at random from A_v. Consider some edge $e = (u, v)$ with $u \in G_1$ and $v \in G_2$. Then, $|A_u| + |A_v| < \ell$, so $\mathcal{C}^*$ induces a coordinated solution to I_e, and so A_u and A_v contain a pair of "corresponding" answers, i.e. there exist $a \in A_u$ such that $R_e(a) \in A_v$. Thus, the probability of this assignment satisfying e is at least $\frac{1}{\ell/2} \cdot \frac{1}{\ell/2} = 4/\ell^2$. The expected number of edges satisfied by the assignment is at least $\#e(G_1, G_2) \cdot 4/\ell^2$, where $\#e(G_1, G_2) = |E \cap (G_1 \times G_2)|$. Using the fact that the expectation never exceeds the maximum achievable value, we have $\#e(G_1, G_2) \cdot 4/\ell^2 \leq \alpha^k \cdot |E|$. Therefore,

$$\frac{\#e(G_1, G_2)}{|E|} \quad \leq \quad \alpha^k \cdot \frac{\ell^2}{4}.$$

Since $\ell^2 = O(k^2 \log^2 n)$ and $\alpha < 1$, it suffices to pick $k = O(\log \log n)$ to make $\alpha^k \ell^2/4 < 1/2$. Recall that G is a d-regular bipartite graph for some d depending on k. If $|B_1| < |Q_1|/4$ and $|B_2| < |Q_2|/4$,

the number of edges with at least one endpoint in $B_1 \cup B_2$ would be less than $d(|Q_1| + |Q_2|)/4$, implying that $\#e(G_1, G_2)/|E| > 1/2$ since $|E| = d(|Q_1| + |Q_2|)/2$. Hence, either $|B_1| \geq |Q_1|/4$ or $|B_2| \geq |Q_2|/4$. Since $|Q_1|, |Q_2| > n^k$, the number of sets in $\mathcal{C}^*$ is at least $\frac{\ell}{2} \cdot \frac{n^k}{4} = \Theta(\ell n^k)$. ■

Let $N = O(n^{\log \log n})$ denote the size of the Set Cover instance. If Set Cover could be approximated to within a factor of $\beta \cdot \log N$ for every $\beta > 0$, we could easily tell whether our Label Cover instance has an assignment satisfying all edges in time poly(N). By Lemma 29.2.2, NP $\subseteq$ DTIME(N). This proves Theorem 29.2.1.