6.854 Advanced Algorithms

Lecture 11: 10/18/2004 Lecturer: David Karger

Scribe: David Schultz

11.1 Algorithms for Linear Programming

11.1.1 Last Time: The Ellipsoid Algorithm

Last time, we touched on the ellipsoid method, which was the first polynomial-time algorithm for linear programming. A neat property of the ellipsoid method is that you don't have to be able to write down all of the constraints in a polynomial amount of space in order to use it. You only need one violated constraint in every iteration, so the algorithm still works if you only have a *separation oracle* that gives you a separating hyperplane in a polynomial amount of time. This makes the ellipsoid algorithm powerful for theoretical purposes, but isn't so great in practice; when you work out the details of the implementation, the running time winds up being $O(n^6 \log nU)$.

11.1.2 Interior Point Algorithms

We will finish our discussion of algorithms for linear programming with a class of polynomial-time algorithms known as *interior point algorithms*. These have begun to be used in practice recently; some of the available LP solvers now allow you to use them instead of Simplex.

The idea behind interior point algorithms is to avoid turning corners, since this was what led to combinatorial complexity in the Simplex algorithm. They do this by staying in the interior of the polytope, rather than walking along its edges. A given constrained linear optimization problem is transformed into an unconstrained gradient descent problem. To avoid venturing outside of the polytope when descending the gradient, these algorithms use a potential function that is small at the optimal value and huge outside of the feasible region.

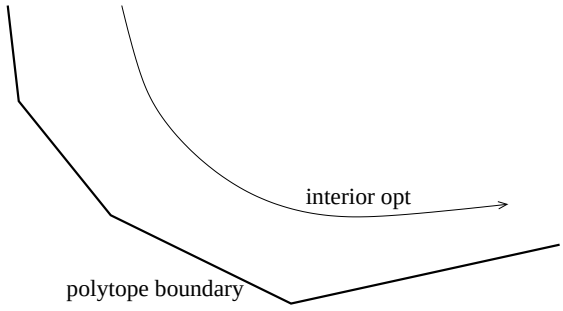

Figure 11.1: An interior point search.

To make this idea more concrete, let us consider a specific interior point algorithm due to Ye. Consider the following linear program:

$$\begin{array}{rll} \text{minimize} & cx & \\ \text{subject to} & Ax & = b \\ & x & \geq 0 \end{array}$$

The dual of this program can be written with slack variable s as follows:

$$\begin{array}{rll} \text{maximize} & yb & \\ \text{subject to} & yA + s & = c \\ & s & \geq 0 \end{array}$$

Ye's algorithm solves the primal and the dual problem simultaneously; it approaches optimality as the duality gap gets smaller. The algorithm uses the following potential function:

$$\phi = A \ln xs - \sum \ln x_i - \sum \ln s_i$$

The intent of the $A \ln xs$ term is to make the optimal solution appealing. Note that $\ln xs$ approaches $-\infty$ as the duality gap approaches 0. The two subtracted terms are *logarithmic barrier functions*, so called because their purpose is to make infeasible points unappealing. When one of the x_i or s_i is small, they add a large amount to the potential function. When one of the x_i or s_i is negative, these functions are infinitely unappealing. The constant A needs to be made large enough that the pull towards the optimal solution outweighs the tight constraints at the optimal point.

To minimize ϕ, we use a gradient descent. We need to argue that the number of gradient steps is bounded by a polynomial in the size of the input. It turns out that this can be done by arguing that in each step, we get a large improvement. Finally, when we reach a point that is very close to the optimum, we use the trick from last time to move to the closest vertex of the polyhedron.

11.1.3 Conclusion to Linear Programming

Linear programming is an incredibly powerful sledgehammer. Just about any combinatorial problem that can be solved can be solved with linear programming (although many of these special cases of linear programming can be solved more quickly by other methods). For the rest of the lecture, however, we will look at some of the problems that cannot be solved by linear programming.

11.2 Polynomial-Time Approximation Algorithms

NP-hard problems are a vast family of problems that (to the best of our knowledge) cannot be solved in polynomial time. It is important to be able to identify such problems because it tells you where to direct your efforts. In this class, we will not prove any of these problems to be NP-hard; you should take 6.840 to learn how to do that. [1]

Instead, the question we concern ourselves with is how to cope once it has been proved that a problem is NP-hard. Here are three possible strategies:

- Assume that the input is random, and prove that an algorithm will perform well in the average case. For example, the NP-hard problem of finding the maximum clique in a graph can be solved efficiently by this argument because the maximum clique in a randomly-chosen graph is small. This argument is often used in practice, but it has the problem that not everyone will agree that the input distribution is random.
- Run a super-polynomial algorithm anyway. Techniques such as branch-and-bound (known as A^* search in the AI world) allow you to enumerate options in a way that lets you skip most of the problem space. However, the desired complexity bounds on these algorithms are usually not provable, and even "slightly" super-polynomial is often too much to be practical.
- Settle for a suboptimal solution (an approximation) that can be found in polynomial time, and prove that this solution is "good enough". This is the technique that we will look at in the rest of the lecture.

11.2.1 Preliminaries

Definition 1 *An **optimization problem** consists of:*

A set of instances I

A set of solutions $S(I)$

An objective function[2] *$f : S(I) \to \boldsymbol{R}$*

The problem is to find $s \in S(I)$ that maximizes (or minimizes) $f(s)$.

Some technicalities: When we come to analyze the complexity of such problems, we will assume that all inputs and the range of f are in the rational numbers. We will also assume that if σ is an optimal solution, then $f(\sigma)$ is polynomial in the number of bits in the input. Our goal will be to find algorithms that take a polynomial amount of time in the representation of the input in bits.

[1] Briefly, P is the set of problems solvable in polynomial time, and NP is a particular superset of P. A problem is NP-hard if it is at least as hard as all problems in NP, and NP-complete if it is in NP and it is also NP-hard. An unproven but generally accepted conjecture, which we will assume is true, is that $P \neq NP$.

[2] The restriction that the range of the objective function is $\boldsymbol{R}$ might be limiting in some situations, but most problems can be formulated this way.

Example 1 ***Graph Coloring*** *is an optimization problem with the following form:*

I: graphs

$S(I)$: assignments of colors to each vertex such that no neighbors have the same color

$f(s)$: number of colors used in s

Often when people talk about NP-hard problems, they are referring to decision problems, which are algorithms for which the output is `yes` or `no`. For instance, a decision version of Graph Coloring is the problem of determining whether a graph is 3-colorable. We need a notion of NP-hardness that applies to optimization problems as well.

Definition 2 ***NP-hardness:*** *An optimization problem is NP-hard if it can be used as a subroutine to solve an NP-hard decision problem in polynomial time, with the optimization problem used as a black box.*

Definition 3 *An* ***approximation algorithm*** *is any algorithm that gives a feasible solution to an optimization problem.*

The above definition sounds a bit silly. For instance, a valid approximation algorithm for Graph Coloring is to color each vertex with its own color. In order to formulate *useful* approximation algorithms, we need a way to be able to differentiate them based on the "goodness" of the approxmiation. In the rest of the lecture, we will consider two ways to measure "goodness."

11.2.2 Absolute Approximations

One way to compare approximation algorithms is to measure the solutions they find relative to the optimal solution, $OPT(I)$.

Definition 4 *A is a* ***k-absolute approximation algorithm*** *if $|A(I) - OPT(I)| \leq k$.*

The absolute value in the above definition allows us to cover both maximization and minimization problems with the same definition. $A(I) \leq OPT(I)$ in a maximization problem, and $A(I) \geq OPT(I)$ in a minimization problem.

Our next example will involve planar graphs. Recall the following definition:

Definition 5 *A* ***planar graph*** *is a graph that can be drawn on the blackboard without any edges crossing.*

Euler's formula for planar graphs tells us that $m \leq 3n - 6$ for $n \geq 3$, where n is the number of vertices and m is the number of edges. As a particular corollary, every planar graph has at least one vertex with degree less than 6, since the sum of the degrees of the vertices is $2m$, which by Euler's formula is less than $6n$.

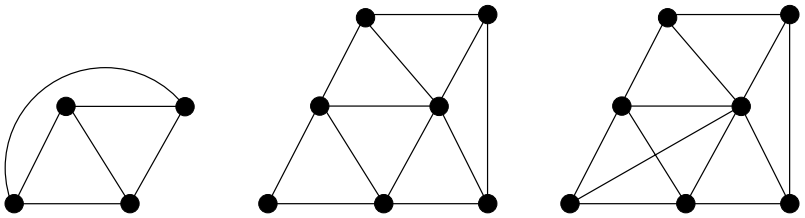

Figure 11.2: Examples of planar and non-planar graphs.

Definition 6 ***Planar Graph Coloring*** *is a restricted version of Graph Coloring where the inputs are constrained to be planar.*

Theorem 1 *There exists a 3-absolute approximation algorithm for Planar Graph Coloring.*

Proof: We will exhibit such an algorithm. First, note that if a graph is 1-colorable, then it has no edges and we can trivially color it. Similarly, a 2-colorable graph is bipartite, and all other graphs require at least 3 colors. We can color any other planar graph with 6 colors with the following greedy algorithm:

- Remove a vertex with degree less than 6. Such a vertex must exist, as we showed above.
- The remainder of the graph is still planar, so recursively color it with 6 colors.
- Re-insert the missing vertex and color it with a color that is not used by any of its (up to 5) neighbors.

The complete algorithm for an arbitrary planar graph uses the above algorithm as a subroutine, and works as follows:

- If the graph is 1-colorable, then color it optimally.
- If the graph is bipartite, color it with 2 colors.
- Otherwise, the graph requires at least 3 colors. Color it with 6 colors.

This algorithm achieves a 3-absolute approximation.[3] ■

Unfortunately, the set of NP-hard problems with absolute approximations is very small. This is because in general, we can transform an absolute approximation into an exact solution by a scaling technique, which would imply that $P = NP$. Assuming that $P \neq NP$, we can prove that no absolute approximation exists for problems that have this property. We will do this for Knapsack, one of the first problems that was shown to be NP-complete, and for Maximum Independent Set, another NP-complete problem.

Definition 7 *In the* ***Knapsack*** *problem, we are given a knapsack of size B and items i with size s_i and profit p_i. We can think of this as a kind of shoplifting problem; the goal is to find the subset of the items with maximum total profit that fits into the knapsack.*

[3] With a little bit more effort, we can obtain a 5-coloring of any planar graph, and hence, a 2-absolute approximation. Of course, the famous Four Color Theorem states that any planar graph can be 4-colored.

Theorem 2 *There is no k-absolute approximation algorithm for Knapsack for any number k.*

Proof: Suppose for the sake of contradiction that we have a k-absolute approximation algorithm. First we will consider the case where the prices p_i are integers. We multiply all the prices by $k+1$ and run our k-absolute approximation algorithm on the resulting instance. The difference between any two solutions in the original instance is at least 1, so the difference between any two solutions in the scaled instance is at least $k+1$. As a result of scaling, the optimum solution increases by a factor of $k+1$, but our k-approximate solution is within k of this. Therefore, the approximation algorithm gives an exact solution for the scaled graph. We divide by $k+1$ to obtain an optimal solution for the original graph. This would constitute a polynomial-time solution for a problem that is believed to be super-polynomial, so we have a contradiction.

Note that if the prices are rational numbers rather than integers, we can simply multiply all of the prices by their common denominator, which has a polynomial number of bits. Then the problem reduces to the integer case above. ■

Definition 8 *In the **Maximum Independent Set** problem, we are given a graph G and we must find the largest subset of the vertices in G such that there is no edge between any two vertices in the set.*

Note that a graph's maximum independent set is identical to the maximum clique in the complementary graph. (The complement of a graph is the graph obtained by taking the same set of vertices as in the original graph, and placing edges only between vertices that had no edges between them in the original graph.)

Theorem 3 *There is no k-absolute approximation for Maximum Independent Set, for any number k.*

Proof: Suppose for the sake of contradiction that we have a k-absolute approximation algorithm for Maximum Independent Set. Although there are no numbers to scale up in this problem, there is still a scaling trick we can use.

For an instance G, we make a new instance G' out of $k+1$ copies of G that are not connected to each other. Then a maximum independent set in G' is composed of one independent set in each copy of G, and in particular, $OPT(G')$ is $k+1$ times as large as $OPT(G)$. A k-absolute approximation to a maximum independent set of G' has size at least $(k+1)|OPT(G')| - k$. This approximation consists of independent sets in each copy of G. At least one of the copies must contain a maximum independent set (i.e., an independent set of size $|OPT(G)|$), because otherwise the total number of elements in the approximation would be at most $(k+1)|OPT(G)| - (k+1)$, contradicting the assumption that we have a k-absolute approximation. Hence, we can obtain an exact solution to Maximum Independent Set in polynomial time, which we presume is impossible. ■

11.2.3 Relative Approximations

Since *absolute* approximations are often not achievable, we need a new technique—a new way to measure approximations. Since we can't handle additive factors, we will try multiplicative ones.

Definition 9 *An* ***α-approximate solution*** $S(I)$ *has value* $\leq \alpha|OPT(I)|$ *if the problem is a minimization problem and value* $\geq |OPT(I)|/\alpha$ *if the problem is a maximization problem.*

Definition 10 *An algorithm has an* ***approximation ratio*** α *if it always yields an* α*-approximate solution.*

Of course, α must be at least 1 in order for these definitions to make sense. We often call an algorithm with an approximation ratio α an α-approximate algorithm. Since absolute approximations are so rare, people will assume that a relative approximation is intended when you say this.

Once we have formulated an algorithm, how can we prove that it is α-approximate? In general, it is hard to describe the optimal solution, since it is our inability to talk about OPT that prevents us from formulating an exact polynomial-time algorithm for the problem. Nevertheless, we can establish that an algorithm is α-approximate by comparing its output to some upper or lower bound on the optimal solution.

Greedy approximation algorithms often wind up providing relative approximations. These greedy methods are algorithms in which we take the best local step in each iteration and hope that we don't accumulate too much error by the time we are finished. To illustrate this technique, we will consider Maximum Cut, which is the obvious complement to the Minimum Cut problem we studied earlier.

Definition 11 *In the* ***Maximum Cut*** *problem (for undirected graphs), we are given an undirected graph and wish to partition the vertices into two sets such that the number of edges crossing the partition is maximized.*

Example 2 ***A Greedy Algorithm for Maximum Cut:*** *In each step, we pick any vertex and place it on one side of the cut or the other. In particular, we place each vertex on the side opposite most of its previously placed neighbors. (If the vertex has no previously placed neighbors, or an equal number of neighbors on each side, then we place it on either side.) The algorithm terminates when all vertices have been placed.*

We can say very little about the structure of the optimum solution to Maximum Cut for an arbitrary graph, but there's an obvious bound on the number of edges in the cut, namely m, the total number of edges in the graph. We can measure the goodness of our greedy algorithm's approximation with respect to this bound.

Theorem 4 *The aforementioned greedy algorithm for Maximum Cut is 2-approximate.*

Proof: We say that an edge is "fixed" if both of its endpoints have been placed. Every time we place a vertex, some number of edges get fixed. At least half of these edges are cut by virtue of the

heuristic used to choose a side for the vertex. The algorithm completes when all edges have been fixed, at which point $m/2$ edges have been cut, whereas we know that optimum solution has at most m edges. ■

The approach we just described was the best known approximation algorithm for Maximum Cut for a long time. It was improved upon in the last decade using a technique that we will discuss in the next lecture.

Example 3 ***A Clustering Problem:*** *Suppose we are given a set of points, along with distances satisfying the triangle inequality. Our goal is to partition the points into k groups such that the maximum diameter of the groups is minimized. The diameter of a group is defined to be the maximum distance between any two points in the group.*

An approximation algorithm for this problem involves first picking k cluster "centers" greedily. That is, we repeatedly pick the point at the greatest distance from the existing centers to be a new center. Then we assign the remaining points to the closest centers. The analysis of this problem is left as a problem set exercise.

As we demonstrated, Maximum Cut has a constant-factor approximation. We will now consider an algorithm that has no constant-factor approximation, i.e., α depends on the size of the input.

Definition 12 *In the* ***Set Cover*** *problem, we are given a collection of n items and a collection of (possibly overlapping) sets, each of which contains some of the items. A cover is a collection of sets whose union contains all of the items. Our goal is to produce a cover using the minimum number of sets.*

A natural local step for Set Cover is to add a new set to the cover. An obvious greedy strategy is to always choose the set that covers the most new items.

Theorem 5 *A greedy algorithm for Set Cover that always chooses the set that covers the most new items is $O(\log n)$-approximate.*

Proof: Suppose that the optimum cover has k sets. Then at a stage where r items remain to be covered, some set covers r/k of them. Therefore, choosing the set that covers the most new points reduces the number of uncovered points to $r - r/k = r(1 - 1/k)$. If we start with n points and repeat this process n times, $r \leq n(1 - 1/k)^j$. Since r is an integer, we are done when $n(1 - 1/k)^j < 1$. This happens when $j = O(k \log n)$. Hence, $\alpha = j/k = O(\log n)$. ■

This result is much better than an $O(n)$-approximate solution, but you have to wonder if we could do beter with a cleverer algorithm. People spent a long time trying to find a constant approximation. Unfortunately, it was recently proven that $O(\log n)$ is the best possible polynomial-time approximation.

However, we can do better for a special case of Set Cover called Vertex Cover. Both problems are NP-hard, but it turns out that the latter does indeed have a constant factor approximation.

Definition 13 *In the **Vertex Cover** problem, we are given a graph G, and we wish to find a minimal set of vertices containing at least one endpoint of each edge.*

After seeing a greedy algorithm for Set Cover, it might seem natural to devise a scheme for Vertex Cover in which a local step consists of picking the vertex with the highest uncovered indegree. This does indeed give a $O(\log n)$ approximation, but we can do better with another heuristic.

As a starting point, imagine that instead of looking at the number of covered edges on each vertex, we simply picked *any* uncovered edge and covered it. Clearly one of the two endpoints for the edge is in the optimum cover. Unfortunately, we might get unlucky and pick the wrong one. Consider the graph in Figure 11.3. If we chose the peripheral vertices first, our cover would contain $n - 1$ vertices.

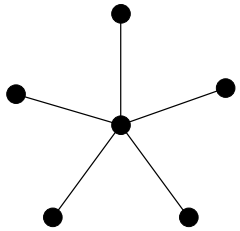

Figure 11.3: A pathological case for naïve greedy Set Cover.

To solve this problem, when we choose an edge to cover, we add both of its endpoints to the cover. This gives us a 2-approximate cover because one of the two endpoints is in the optimum cover. This is a beautiful case of bouding an approximation against a *bound* of the optimum solution without actually knowing the optimum solution.

Relatively recently, someone improved the vertex cover bound to a $2 - O(\log \log n / \log n)$ approximation with a significant amount of work. It is known to be impossible to do better than 4/3, but nobody knows how to close the gap.