LECTURE 1

Introduction, Vertex and Set cover

Since most real world combinatorial optimization problems are **NP**-hard and it is unlikely that they could be solvable in polynomial time, researchers in the past have attempted to find polynomial time algorithms that guarantee a solution with value that is "close" to the value of the optimum solution. The notion of "close" can be defined both in absolute and relative terms. Thus, an algorithm which returns a solution of value within α of the optimum is said to have an *absolute performance guarantee* of α while an algorithm whose solution value is within a multiplicative factor of α of the value of the optimum solution is said to have a *relative performance guarantee* of α. Absolute performance guarantees tend to be very rare and are known for only a few selected problems like edge coloring, vertex coloring in planar graphs *etc.*. In fact for many problems it can be shown that the existence of an algorithm with an additive performance guarantee would imply **P**=**NP**. Henceforth, by an *α-approximation* algorithm we mean (unless stated otherwise) that the algorithm has a *relative* performance guarantee of α. The notion of approximation algorithms can be extended to include all such algorithms which trade the quality of the solution obtained with the amount of resources used (space, time, number of processors etc.).

The search for approximation algorithms has enriched our understanding of **NP**-complete problems. Thus, while it is true that a polynomial time solution to one **NP**-complete problem would imply polynomial time solutions for all others, the problems differ widely in the limits of their approximability. As far as relative measures are concerned there are problems like Knapsack and Bin packing which can be approximated within a factor $(1+\epsilon)$ for any choice of ϵ; problems like vertex cover and maximum satisfiability which can be approximated within a constant factor; problems like set cover which can be approximated only to a logarithmic factor; problems like maximum clique and coloring which cannot be approximated within a factor less than n^ϵ, for some fixed $\epsilon < 1$; and finally problems like the traveling salesman problem for which no algorithm with a polynomial performance guarantee is possible.

1.1 Yes and No certificates

Problems that are **NP**-complete differ from those that are in **P** in one crucial aspect — the absence of a 'NO' certificate. We elaborate on this below.

A *matching* in an undirected graph is a subset of the edges such that no two edges are incident at the same vertex. A vertex is *matched* if there is an edge in the matching that is incident to it. Given a bipartite graph $B = (U, V)$, how can we convince somebody that the graph has a/no perfect matching. The 'Yes' certificate is straightforward — we just demonstrate a perfect matching in the graph. A classical theorem of graph theory says that a bipartite graph has no perfect matching iff there exists a set of vertices, $S \subseteq U$, such that $|S| > |N(S)|$, where $N(S) \subseteq V$ is the vertices in V that are neighbors of the vertices in S. Thus if our graph had no perfect matching then there would exist a set S with the property as claimed in the theorem. Clearly, all the vertices of this set cannot be matched — there are not enough neighbors — and this is a convincing proof of the non-existence of a perfect matching; the 'NO' certificate that we were looking for.

A *vertex cover* in a graph is a subset of the vertices such that every edge has at least one endpoint in this subset. Once again a 'Yes' certificate to the question "Does there exist a vertex cover of size at most k?" is straightforward. What about the 'No' certificate? It turns out that there is no short argument to prove that the size of the vertex cover exceeds k. As a particular instance, consider the Petersen graph which has no vertex cover of size 5. But the only arguments to prove this fact rely on the structure of the graph and cannot in general be applied to other instances. This absence of a short 'No' certificate is a "property" all **NP**-complete problems share. Note however, that there is a short proof that the Petersen graph has no vertex cover of size 4. This is because the graph has a matching of cardinality 5 (a perfect matching). Since any vertex cover should contain one of the two endpoints of each matched edge, no vertex cover can be of size less than 5. From the point of view of approximation algorithms this is an important observation and we elaborate on this below.

1.2 Bounding the Optimum

Consider a **NP**-hard minimization problem whose decision version is **NP**-complete. From our discussion above we know that there is no 'No' certificate to the question "Is there a solution of value at most k?". But maybe, as in the vertex cover example above, we can provide a 'No' certificate for a sufficiently low value of k, say k'. If such is the case then we have proof that the optimum solution is of value at least k'. So although we have no idea what the optimum solution is, we have a lower bound on its value. This fact is crucial to the design of an approximation algorithm. To guarantee an approximation factor of α, our analysis of the algorithm needs to compare the optimum with the quality of the solution found by the algorithm. Since finding the value of the optimum is **NP**-complete, we have to resort to comparing the value of the solution obtained with a lower bound on the value of the optimum. A lower bound is said to be *β-good* if for all instances the value of the lower bound is at least $\frac{1}{\beta}$ times the value of the optimum. If for some instance the lower bound is $\frac{1}{\beta}$ times the optimum then any approximation algorithm relying on this lower bound cannot guarantee an approximation factor better than β. Similar arguments apply for maximization problems except that we now need an upper bound on the value of the optimum. Thus, finding good techniques to bound the optimum value is central to the task of designing good approximation algorithms.

1.3 Vertex Cover

Given an undirected graph, the problem is to find a vertex cover of minimum cardinality. It follows from the discussion before that the size of a matching is a lower bound on the size of the minimum vertex cover. Now, consider a matching that is *maximal*, *ie.* no more edges can be added to the matching. The maximality of the matching implies that every edge not in the matching has a common endpoint with some edge in the matching. Thus the set of matched vertices forms a vertex cover. The size of this vertex cover is twice the size of the matching and so within twice the optimum. This gives a 2-approximation algorithm for the minimum vertex cover problem.

Is a 2-approximation the best possible? With this lower bound on the optimum, Yes!! For odd n, K_n, the complete graph on n vertices has a vertex cover of size $n-1$ and a maximum matching of size $(n-1)/2$. So any algorithm that uses a matching (maximal or maximum) as a lower bound to the size of the minimum vertex cover cannot perform better than 2.

1.4 Set Cover

Given a universe of n elements $U = \{e_1, e_2 \ldots e_n\}$ and sets $S_1, S_2 \ldots S_m \subseteq U$, we wish to find a minimum collection of sets that cover all elements. Once again as in the vertex cover example we search for a good lower bound to the value of this covering problem. Recall that in the case of the vertex cover we took an independent set of edges (a matching) and argued that any vertex cover has to pick at least one end point of each edge in this set. We could argue similarly in this case by picking a collection of elements, C_1, no two of which are covered by the same set. Then the set cover is of size at least $|C_1|$. Or we could pick a collection of elements, C_2, such that no set covers more than two elements in this collection. Then the set cover is of size at least $|C_2|/2$. Generalizing this line of argument would get us to something like the following.

Consider the sets to be boxes of unit volume and the elements to be objects which need to be packed in the boxes so as to occupy the minimum number of them. Each object can only go into a specified set of boxes. Suppose we assign a volume to each object such that the sum of the volumes of the objects that a box *can contain* is less than the volume of the box itself (such a volume assignment is called *feasible*). A feasible volume assignment does not impose any additional constraints on our problem. What it however tell us is that the size of the set cover is at least the sum of the volumes of the objects. For example, the assignment of unit volume to objects in C_1 and zero volume to other objects is feasible and leads to a lower bound of $|C_1|$. Similarly we get a lower bound of $|C_2|/2$ by assigning a volume $1/2$ to objects in C_2 and zero to all others. Thus this way of obtaining a lower bound on the size of the minimum set cover seems to subsume those discussed earlier.

1.4.1 An $\mathcal{H}(n)$-approximation algorithm

It is not immediate from the lower bound developed above what a good strategy for picking sets should be. We need another idea here. Intuitively, we would like to pick the set which covers the most number of elements that are not already covered. We refer to this way of picking sets as the `Greedy` algorithm. How well does `Greedy` perform? We bound the performance of `Greedy` by comparing the number of sets it picks to the lower bound obtained by the packing argument. The way we do this is by assigning volumes to elements when they are first covered by `Greedy`. In particular let S be the first set (picked by `Greedy`) that covers element i and $T \subseteq S$ the elements in S that are not already covered. Then the volume assigned to i, which we denote by y_i, is $1/|T|$. Since each element in T is assigned this volume it follows that the sum of the volumes of the elements is equal to the number of sets picked by `Greedy`.

But is this volume assignment feasible? No! In fact while assigning volumes we made no attempt at ensuring feasibility. We do so now.

What is the "extent of infeasibility" of our volume assignment, *ie.* by what factor does the sum of the volumes of the elements in a set exceed the volume of the set?

Lemma 1.4.1 *For any set S, the sum of the volumes of the elements in S is at most $\mathcal{H}(|S|)$.*

Proof: Let us order the elements of S in the order they were covered by `Greedy`; if two elements were covered by the same set we order them arbitrarily. We claim that the volume assigned to the i^{th} element (in this ordering) is at most $1/(|S| - (i-1))$. This is because when the i^{th} element was covered the set S still had $|S| - (i-1)$ elements that were uncovered. Hence the set which covered the i^{th} element had at least $|S| - (i-1)$ uncovered elements. Therefore the volume assigned to the i^{th} element is at most $1/(|S| - (i-1))$ and summing for all elements in S proves the lemma. ■

Given that the sum of the volumes of the elements in any set S is at most $\mathcal{H}(n)$ (since $|S| \le n$), a natural way to get a feasible volume assignment would be to scale down the volume of each element by $\mathcal{H}(n)$, *ie.* $y_i = y_i/\mathcal{H}(n)$. While this assignment of volumes is feasible the sum of volumes now is $1/\mathcal{H}(n)$ of what it was earlier, which, as mentioned before, was equal to the number of sets picked by `Greedy`. Since the sum of the volumes in any feasible assignment is a lower bound on the size of the minimum set cover we have that $1/\mathcal{H}(n)$ times the number of sets picked by `Greedy` is less than the optimum. Thus the number of sets picked by `Greedy` is at most $\mathcal{H}(n)$ times the optimum giving us an $\mathcal{H}(n)$-approximation algorithm for the set cover problem.

1.4.2 An f-approximation algorithm

We shall now develop another algorithm for the set cover problem with an uncomparable performance guarantee. For analyzing `Greedy` we had an initial infeasible assignment of volumes to the elements which we then scaled to obtain a feasible assignment. We shall now try to assign volumes in a manner so that at each point the assignment is feasible. In particular, when an element is first covered, it is assigned a volume equal to the maximum volume it can have without being infeasible. We achieve this by maintaining the "empty space" in each box/set. Initially, the empty space in a set is just the volume of the set (unit in our case). Every time we assign a volume to an element we decrement the empty space in each set containing this element by the volume of this element.

Thus the maximum volume that we can assign to an element is the minimum empty space that any set containing this element has.

Our algorithm now looks as follows: We pick an uncovered element and set its volume to the maximum possible. We then update the empty space in each set that contains this element. At least one of these sets (the one with the minimum empty space) now has zero empty space. We include this set in our solution and repeat till there are no more uncovered elements.

It is easy to see that the volume assignment to the elements is feasible. Let f be the maximum number of sets any element belongs to. Every time we assign a unit volume to an element the total reduction in the empty space of the sets is at most f. Since a set is included in our solution only when its empty space is down to zero, the total reduction in the empty space of the sets is at least the number of sets in our solution. This implies that the sum of the volumes of the elements is at least $1/f$ times the size of our set cover. Since the sum of the volumes is a lower bound on the size of the minimum set cover, the size of our set cover is at most f times the optimum.

There exists a one-to-one correspondence between vertex covers in a graph and set covers in a set system where the elements are the edges of the graph and the sets correspond to vertices with each set/vertex covering the elements/edges incident to it. In this set system each element belongs to exactly two sets so that $f = 2$. Thus this f-approximation algorithm for the set cover gives us another 2-approximation algorithm for the vertex cover.

1.5 Bibliographic Notes

The 2-approximation algorithm for vertex cover using a maximal matching is due to Gavril (1974). Savage (1982) uses a depth-first search of the graph to obtain a vertex cover with the same performance guarantee. The linear relaxation of an integer programming formulation of the minimum weight vertex cover problem was shown to have half-integral optimum solutions by Nemhauser and Trotter (1975). That this fact could be used to obtain a 2-approximation for the minimum weight vertex cover problem was observed by Hochbaum (1982). The first purely combinatorial approach for computing a 2-approximation to the minimum weight vertex cover was due to Bar-Yehuda and Even (1981). This approximation ratio was later later reduced to $2 - \frac{\log \log n}{\log n}$ by Bar-Yehuda and Even (1985) and Monien and Speckenmeyer (1985).

For set cover, the greedy algorithm was analyzed to have an approximation guarantee of $\mathcal{H}(n)$ independently by Johnson (1974) and Lovász (1975). A similar result was achieved for the weighted case by Chvátal (1979).

Exercises

The minimum vertex cover might be as large as twice the maximum matching. However, for bipartite graphs these two quantities are equal. Prove this. Thus, for the minimum vertex cover in bipartite graphs we have a lower bound that matches the upper bound. This leads to a polynomial time algorithm for finding the minimum vertex cover in bipartite graphs. Suggest one.

Consider the case when the sets have weights associated with them and the objective is to find a set cover of minimum weight (weighted set cover problem). Extend `Greedy` to obtain an $\mathcal{H}(n)$-approximation algorithm for this problem.

Obtain an f-approximation algorithm for the weighted set cover problem by extending the algorithm for the set cover problem. Observe that this yields a 2-approximation algorithm for finding a minimum weight vertex cover in a graph with weights on vertices (weighted vertex cover problem).

Suppose for each element e_i we are also given a "cover requirement", $r_i \in \mathbf{Z}^+$ which is the number of sets in our solution that e_i should occur in (set multi-cover problem). Modify `Greedy` to obtain an $\mathcal{H}(n)$-approximation algorithm.

Now assume that instead of sets we have multisets so that for each multiset S_j and element e_i, c_{ij} denotes the number of times S_j includes/covers e_i. The element e_i also has a cover requirement r_i (multi-set multi-cover problem). Modify `Greedy` to obtain an $\mathcal{H}(n)$-approximation algorithm.

LECTURE 2

Facility location and Bottleneck problems

2.1 Introduction

The k-center problem is a classical problem from location theory: given n cities and the distances between them, we wish to select k cities as centers so that the maximum distance of a city from its closest center is minimized. A closely related problem is the k-supplier problem. The vertices are now partitioned as supplier and customer vertices and the problem is to choose k-supplier vertices such that the maximum distance of any customer to its nearest supplier (from amongst those chosen) is minimum. These two problems fall into a general class of problems known in the literature as *bottleneck problems*. Roughly speaking a bottleneck problem is one in which we are trying to optimize a bottleneck, *ie.* minimizing the maximum or maximizing the minimum.

2.2 The k-center problem

Suppose that the cities are points in the plane and the distance between two cities is the Euclidean distance between the points. In this setting the k-center problem is equivalent to finding the minimum radius such that k disks of this radius can cover all points. Note that the center of a disk should be at one of the n points.

Suppose there exists a set of $k+1$ points such that every pair of points in this set is more than distance x apart. A disk of radius $x/2$ can cover at most one point from this set and hence we need at least $k+1$ disks of radius $x/2$ to cover all points. Hence $x/2$ is a lower bound on the minimum radius.

We would like to have a procedure for obtaining good lower bounds on the minimum radius. Consider the following natural one. We start with k disks of a certain radius, say r. Place the center of the first disk at one of the n points. From amongst the points that are not covered choose one arbitrarily and place the center of the next disk there. Continue in this manner till either all points are covered or there are no disks left. If in the latter case we did not manage to cover all points then the k points at which we placed the centers of the disks and some point that is not covered form a set of $k+1$ points every pair of which is at least r distance apart. Hence $r/2$ is a lower bound on the optimum. We run this procedure for all choices of the radius and let r^* be the minimum radius at which we manage to cover all points with k disks. Then $r^*/2$ is a lower bound on the optimum radius. But we have a solution with radius r^*. This means that the above simple procedure is a 2-approximation algorithm for the euclidean k-center problem.

Note that while searching for r^* the only choices of radius that we need to consider are the distinct interpoint distances which is a small discrete set. This establishes the polynomiality of the algorithm.

To apply this argument to the general setting of graphs we need a few definitions from graph theory.

Definition 2.2.1 *Given a graph $G = (V, E)$ the i^{th}* power of G, *denoted by $G^i = (V, E^i)$ is a graph with the same vertex set as G and an edge between two vertices if they are connected by a path of length at most i in G.*

Definition 2.2.2 *A vertex v* dominates *a vertex u if u is adjacent to v or $u = v$,* ie. *a vertex* dominates *its neighbors and itself. A set of vertices $S \subseteq V$ is a* dominating set *if every vertex in the graph is dominated by some vertex in the set.*

Finding the minimum dominating set in a graph is **NP**-hard. It is equivalent to finding a minimum set cover in a set system where the elements are the vertices of the graph and for each vertex v we have a set S_v whose elements are the vertices that v dominates. This observation gives a $\mathcal{H}(d+1)$-approximation algorithm for the dominating set problem, where d is the maximum degree.

Definition 2.2.3 *A set of vertices $S \subseteq V$ is an* independent set *if no two vertices in the set are adjacent. An independent set is* maximal *if the addition of any vertex to the set yields a set that is not independent.*

Finding the maximum independent set is **NP**-hard, However a maximal independent set can be found in polynomial time by the following simple procedure. To begin with our independent set is empty. A vertex is added to the set if the new set formed is also independent. Continuing in this manner till no more vertices can be added yields a maximal independent set. If the order in which the vertices are considered is their lexicographic ordering then the set found is the *lexicographically first maximal independent set.* This is the maximal independent set that we refer to in the rest of the chapter. Note that a maximal independent set is also a dominating set.

Getting back to the k-center problem, let $G = (V, E)$ be an undirected graph and $w : E \to \mathbf{R}^+$ a weight function on the edges. To derive an algorithm with a bounded performance guarantee we require that the weight function satisfy triangle inequality; this condition is usually no restriction. Let $w_1, w_2, w_3 \ldots$ be the distinct edge weights in increasing order and let G_i be the subgraph induced by edges of weight at most w_i. First observe that the optimum value for the k-center problem is equal to one of the edge weights; in particular it is the minimum edge weight w_i such that G_i has a dominating set of size at most k. While it is easy to generate the subgraphs $G_1, G_2, G_3 \ldots$, the problem of checking if these subgraphs have a dominating set of size at most k is **NP**-complete. However, suppose that G_i has a set, I, such that no vertex in G_i dominates two vertices of I and $|I| > k$. Then the minimum dominating set in G_i is of size more than k. Note that the set I is an independent set in G_i^2.

Thus to argue that G_i has no dominating set of size at most k, we need an independent set in G_i^2 of size larger than k. What if the largest independent set we can find in G_i^2 is of size no more than k? If the independent set we find is also maximal then we have a dominating set in G_i^2 of size at most k!

Let G_j be the first subgraph in the sequence $G_1, G_2, G_3, \ldots$ such that the maximal independent set found in G_j^2 is of size no more than k. Since G_{j-1}^2 has an independent set of size larger than k, every dominating set in G_{j-1} is of size more than k and hence the optimum value is at least w_j. Further, G_j^2 has a dominating set (the maximal independent set found) of size at most k. Since the edge weights satisfy triangle inequality, the longest edge in G_j^2 has weight at most $2w_j$. Thus we have a k-center in which the distance of any vertex to its closest center is at most twice the optimum.

2.3 The Basic Paradigm

We shall now extract the essence of the argument for the k-center problem to develop a paradigm that could be applied to other bottleneck problems. A bottleneck problem typically has the form "Find the minimum w_i such that the subgraph G_i has a certain property $\mathcal{P}$". Suppose we can find another property $\mathcal{P}'$ such that

1. If G_i does not satisfy $\mathcal{P}'$ it does not satisfy $\mathcal{P}$.
2. If G_i satisfies $\mathcal{P}'$ a suitable power of G_i (say G_i^c) satisfies $\mathcal{P}$.

A polynomial time procedure to check if a graph satisfies property $\mathcal{P}'$ yields a c-approximation algorithm for this bottleneck problem as follows. Let G_i be the first subgraph in the sequence $G_1, G_2, G_3 \ldots$ which satisfies property $\mathcal{P}'$. Since G_{i-1} does not satisfy $\mathcal{P}'$, it does not satisfy $\mathcal{P}$ either and so the optimum value is at least w_i. Further, G_i^c satisfies property $\mathcal{P}$ and since the longest edge in G_i^c is of weight at most $c \cdot w_i$, we have a solution of value at most c times the optimum. Thus the first condition on $\mathcal{P}'$ establishes a lower bound on the optimum value while the second gives a solution of value at most c times the lower bound.

For the k-center problem, $\mathcal{P}$ is the property that the graph has a dominating set of size at most k and $\mathcal{P}'$ is the property that the the maximal independent set in the square of the graph is of size at most k.

2.4 Applying the Paradigm

For the remaining problems that we consider in this chapter, we shall only identify the properties $\mathcal{P}$ and $\mathcal{P}'$ and show that $\mathcal{P}'$ satisfies the two conditions for a suitable choice of c.

2.4.1 The k-supplier problem

The vertex set of the graph is now partitioned into sets $\mathcal{S}$ and $\mathcal{C}$, the set of suppliers and customers respectively. We wish to find the minimum i such that a set of at most k suppliers dominates all the customers in G_i (property $\mathcal{P}$). $\mathcal{P}'$ is the property that the maximal independent set in the subgraph induced by the customers in G_i^2 is of cardinality at most k.

If G_i does not satisfy $\mathcal{P}'$ then there are $k+1$ customers that form an independent set in G_i^2. Since no supplier can dominate two vertices from this independent set, any set of suppliers dominating the customers is of size at least $k+1$; hence G_i does not satisfy $\mathcal{P}$. G_i satisfying $\mathcal{P}'$ implies that G_i^2 has a set, I, of at most k customers which dominate all customers. For each vertex $v \in I$ we pick a vertex $u \in \mathcal{S}$ such that u and v are adjacent in G_i. This gives us a set of at most $|I|$ $(\leq k)$ suppliers which dominate all customers in G_i^3. Hence, if G_i satisfies property $\mathcal{P}'$, G_i^3 satisfies $\mathcal{P}$ and this yields a 3-approximation algorithm for the k-supplier problem.

2.4.2 The Bottleneck Traveling Salesman Problem

A traveling salesman wishes to visit n cities stopping at each city for the night. He wants to plan his tour in such a manner that the farthest distance he has to travel on any day is minimized. This variant of the traveling salesman problem known as the *bottleneck traveling salesman problem* is also **NP**-hard. The property that we are now interested in is that G_i be *hamiltonian.*

Definition 2.4.1 *A graph $G = (V, E)$ is* biconnected *or* 2-vertex connected *if on removal of any vertex the graph remains connected. Equivalently, G is biconnected if every pair of vertices belongs to some simple cycle.*

The property $\mathcal{P}'$ that we consider is that G_i be biconnected. Clearly, if G_i violates $\mathcal{P}'$ it cannot be hamiltonian. For the other condition on $\mathcal{P}'$ we need the following result from graph theory: *the square of a biconnected graph is hamiltonian.* Hence if G_i satisfies $\mathcal{P}'$, G_i^2 satisfies $\mathcal{P}$ and we have a 2-approximation algorithm for the bottleneck traveling salesman problem.

2.5 Are better approximations possible?

Surprisingly enough, for the problems considered in this chapter the answer is No! A polynomial time $(2-\epsilon)$-approximation algorithm for the k-center problem would imply **P=NP**. We elaborate on this below.

Let $\mathcal{A}$ be a polynomial time algorithm which given a complete graph and weights on the edges which satisfy triangle inequality, returns a k-center of value at most $(2-\epsilon)$ times the optimum for some $\epsilon > 0$. We show that $\mathcal{A}$ can be used to answer the dominating set problem. An instance of the dominating set problem is a graph $G = (V, E)$ and a number k and we wish to know if G has a dominating set of size at most k. From G we construct an instance of the k-center problem by assigning unit weight to the edges in E and a weight 2 to edges not in E. Clearly, these weights satisfy triangle inequality. The k-center instance has an optimum value of 1 iff G has a dominating set of size k; else the optimum value is 2. Therefore $\mathcal{A}$ returns a solution of value 1 iff G has a dominating set of size k thus solving an **NP**-complete problem.

Bibliographic Notes

The general paradigm for bottleneck problems was developed by Hochbaum and Shmoys (1986). They also gave the 2-approximation for the k-center problem and the 3-approximation for the k-supplier problem besides algorithms for other a host of problems. The 2-approximation for the bottleneck traveling salesman problem is due to Parker and Rardin (1982). They also gave a polynomial time algorithm for finding the hamiltonian cycle in the square of a biconnected graph; this was independently obtained by Lau (1980). The theorem that the square of a biconnected graph is hamiltonian goes back to Chartrand, Hobbs, Jung, Kapoor and Nash-Williams (1974) and Fleischner (1974).

The **NP**-hardness of approximating the k-center is due to Hochbaum and Shmoys (1986) and that of the k-supplier is due to Karloff.

LECTURE 3

Relevance of Linear Programming to Approximation Algorithms

Linear Programming has turned out to be a powerful tool in the design of approximation algorithms. Given an **NP**-hard optimization problem we can formulate it as an integer program and by relaxing the integrality constraints we obtain a linear program, referred to as a **LP**-relaxation of the integer program. Note that there might be many different ways of relaxing the integrality constraints and hence the **LP**-relaxation is not unique.

The linear program that we obtain by relaxing the integrality constraints would, in general, have solutions that are fractional. If the linear program is a minimization program then the value of the optimum fractional solution is only less than that of the optimum (since we only relax some constraints). Thus the optimum fractional solution and, by duality, any feasible solution to the dual program, provide a lower bound on the optimum. Associating a semantics with the dual feasible solution sometimes helps in designing a good approximation algorithm for the primal problem. We illustrate this using the example of set cover.

In some instances the **LP**-relaxation might have even stronger properties. For example, we might be able to argue that the **LP**-relxation has an optimum solution that although not integral is half-integral. In such a case we can "round" the solution suitably to obtain an integral solution of value "close" to the value of the optimum fractional solution and hence close to the optimum. We illustrate this with the example of vertex cover.

3.1 Weighted vertex cover

An Integer Program (IP) for the vertex cover problem can be obtained by assigning for each vertex $v \in V$ a variable $x(v)$ which is constrained to be either 0 or 1 with $x(v) = 1$ denoting that v is in the vertex cover. Thus for each edge $e = (u, v)$ it must be the case that $x(u) + x(v) \geq 1$. If we relax the constraint on x to $\forall v \in V, 0 \leq x(v) \leq 1$, we obtain the linear program

$$\begin{array}{lrcll}
\text{minimize} & \sum_{v \in V} x_v & & & \\
\text{subject to} & & & & \\
& x_u + x_v & \geq & 1 & \text{(for all edges } e = (u,v)) \\
& x_v & \leq & 1 & \text{(for all vertices } v) \\
& x_v & \geq & 0 & \text{(for all vertices } v)
\end{array}$$

Since we are minimizing $\sum_{v \in V} x_v$ we would never set x_v to more than 1 in an optimal solution. Hence the second set of inequalities are redundant. The dual of this linear program is

$$\begin{array}{lrcll}
\text{maximize} & \sum_{e \in E} y_e & & & \\
\text{subject to} & & & & \\
& \sum_{e \in \nabla(v)} y_e & \leq & 1 & \text{(for all vertices } v) \\
& y_e & \geq & 0 & \text{(for all edges } e)
\end{array}$$

An integral solution to this dual linear program is a matching (an edge e is in the matching if $y_e = 1$). We refer to the polyhedron defined by the inequalities in the above two linear programs as the *vertex cover polyhedron* and the *fractional matching polyhedron* respectively.

All vertices of the vertex cover polyhedron are half-integral, *ie.* if x^* is a vertex solution then for all vertices v, $x^*(v) \in \{0, 1/2, 1\}$. Therefore for any choice of vertex weights, there is an optimum solution that is half-integral. Rounding up all the $1/2$'s to 1's we obtain an integral solution of weight at most twice the optimum. The vertices whose variable is at a 1 in this rounded solution form a vertex cover of weight at most twice the optimum.

It is also known that every vertex of the fractional matching polytope is half-integral. By adding another set of inequalities (known as the *odd set inequalities*) we can obtain a polytope every vertex of which is integral and corresponds to a matching in the graph; *the matching polytope.* For bipartite graphs the vertex cover polyhedron and the fractional matching polytope have integral vertices; this again implies that for bipartite graphs the minimum vertex cover can be found in polynomial time.

3.2 Set cover

The set system is given by a universe U of n elements and a collection of subsets $\mathcal{S}$. For each subset S in the collection $\mathcal{S}$ we have a variable x_S; $x_S = 1$ in an integral solution implies that S is included in the set cover. Then for every element we require that the sum of the variables of the sets containing the element should be at least 1. Relaxing the 0/1 constraint on x_S to $x_S \geq 0$ we get the following linear program

$$\begin{array}{lrcll}
\text{minimize} & \sum_{S \in \mathcal{S}} x_S & & & \\
\text{subject to} & & & & \\
& \sum_{S: e \in S} x_S & \geq & 1 & \text{(for all elements } e \text{ in } U) \\
& x_S & \geq & 0 & \text{(for all sets } S \text{ in } \mathcal{S})
\end{array}$$

whose dual is

$$\begin{array}{llll} \text{maximize} & \sum_{e \in U} y_e & & \\ \text{subject to} & & & \\ & \sum_{e \in S} y_e & \leq 1 & \text{(for all sets } S \text{ in } \mathcal{S}) \\ & y_e & \geq 0 & \text{(for all elements } e \text{ in } U) \end{array}$$

where y_e is a variable associated with the element e. Note that this is the same as the volume we associated with the element/object when deriving a lower bound for the size of the set cover. Any feasible solution to the dual **LP** is of value at most that of the primal **LP** and hence the sum of the volumes in any feasible volume assignment is a lower bound on the size of the set cover. Thus the duality theory of linear programming provides a handy tool for bounding the optimum value.

Note that the best lower bound we can obtain by assigning volumes to elements is no more than the optimum value of the primal **LP**– the size of the minimum fractional set cover. Thus for any instance the gap between this lower bound and the optimum value is the same as the gap between the values of the optimum fractional and integral solution. The maximum gap (over all instances) is referred to as the *integrality gap* of the linear program.

The following is an instance of the set cover problem which demonstrates that the integrality gap for our linear program is at least $\frac{1}{4}\log n$.

Consider a set system the elements of which are those m bit strings that have more 1's than 0's. Thus the total number of elements is $n = 2^{m-1}$. There are m sets in this system and the i^{th} set is the set of all those elements that have a 1 in the i^{th} position. Thus each element is included in at least $m/2$ sets and hence assigning $x_S = 2/m$ to each set S yields a feasible solution of value equal to 2. On the other hand no set cover is of size $m/2$ or less since an element whose m bit string has 1's only in positions corresponding to sets not included in the set cover would not be covered. Thus the minimum set cover is of size at least $1 + m/2$ and the fractional set cover of size at most 2 yielding an integrality gap of at least $m/4 \geq \frac{1}{4}\log n$.

LECTURE 4

Steiner Forests

4.1 Introduction

A large class of **NP**-hard optimization problems in graphs involve finding a 'small' subgraph which satisfies certain 'connectivity properties'. 'small' usually refers to the weight of the subgraph when edge weights are specified or just the number of edges in the subgraph. The connectivity requirement could for instance be that all vertices are connected, in which case the problem is just of computing the minimum spanning tree, or the requirement could be that two vertices s, t be connected in which case the problem is that of finding the shortest path between s and t. One of the most general ways of specifying connectivity requirements is to list for each pair of vertices the minimum number of edge disjoint paths that should connect the pair.

In this chapter we consider the case when the number of paths required between a pair of vertices is either 0 or 1. Note that the optimum solution in this case is a forest and we refer to this problem as the *steiner forest* problem. A special case of this problem arises when only a specified subset of vertices (called *terminals*) should be connected *ie.*, every pair of terminals should be connected by a path. The optimum solution is now a tree spanning the terminals, the non-terminal vertices are referred to as *steiner vertices* and the problem is the classical *steiner tree problem*.

The steiner tree problem was originally formulated as the minimum way of connecting a given set of points in the plane. Suppose we had three points sitting on the vertices of an equilateral triangle of side 1. Then the naive way of connecting these points could be to pick two sides of the equilateral triangle yielding a length of 2 while by placing another point at the centroid of this triangle (a steiner point) and connecting this to all the other three we obtain a connection of length only $\sqrt{3}$. The steiner tree problem (and hence also the steiner forest problem) remain **NP**-hard even for this Euclidean case. The simplicity of this setting allows us to develop good lower bounds and this is what we do next.

4.2 Lower Bounding the Optimum

For the steiner tree problem a lower bound on the optimum value can be obtained by placing non-overlapping disks at the points that need to be connected. Any way of connecting the points would have within each disk a segment of length at least the radius of the disk. Since these true for each disk and the disks are non-overlapping, the sum of the radii of the disks is a lower bound on the value of the optimum solution.

This lower bound is usually not close enough to the optimum and hence we need to improve it further. We do this by introducing the notion of a *moat*. A moat is just a ring of uniform width enclosing a region of the plane. If both sides of a moat have at least one point then any tree connecting the points has within the moat a segment of length at least the width of the moat. Thus a non-overlapping collection of disks and moats yields a lower bound equal to the sum of the radii of the disks and the widths of the moats. For the sake of uniformity we view a disks as a moat of width equal to the radius of the disk.

This same lower bound applies to the steiner forest problem. Note however, that the width of a moat can be included in the lower bound only if there exists a pair of vertices that need to be connected and lie on different sides of this moat, otherwise the optimum solution might have no segment inside the moat.

4.3 Building the Lower Bound

We now give a procedure for constructing a lower bound (a collection of moats) and a steiner forest of length within twice the lower bound. We begin by growing a moat around each point; the width of a moat increases continuously with time with each moat growing at the same rate. When two moats 'collide' we grow a moat around the region occupied by these two moats. Furthermore we grow a moat iff some pair of vertices that need to be connected is separated by the moat.

Our procedure ensures that when two moats collide there are two points, one each in the region surrounded by these moats, such that every part of the edge joining these points is contained in some moat or the other. We say that such an edge is 'tight' and we include it into our solution. Thus at any point of the algorithm the edges that have already been included in the solution form a forest. Each tree of this forest spans the points contained in a contiguous region of the plane covered by the moats grown so far. Let F be the forest of edges included in the solution; this is not the final solution since we delete from F all those edges that are inessential. An edge $e \in F$ is inessential if $F - \{e\}$ satisfies all the connectivity requirements. Let F_{sol} denote the forest obtained by removing all inessential edges. We will now show that the total length of all edges in F_{sol} is at most twice the lower bound.

4.4 The Lower Bound

An *event* occurs when two moats collide and the i^{th} iteration is the duration between the $(i-1)^{\text{th}}$ and the i^{th} iteration. Let the i^{th} iteration last for a time t; thus we increase the width of all moats in this iteration by t. The total increase in the lower bound in the i^{th} iteration is therefore equal to t times the number of moats that are growing.

To account for the length of the edges note that all edges in F_{sol} are tight. Hence the length of any such edge is equal to the sum of the widths of the moats that this edge crosses. Thus the length of F_{sol} can be split into the contributions to F_{sol} in the various iterations. In particular the contribution of the i^{th} iteration to F_{sol} is t times the total number of times the edges of F_{sol} cross the moats that are growing in this iteration.

To formalize the above concepts we introduce some notation. Let F_i be the subgraph obtained from F_{sol} by contracting all edges that were picked in iteration 1 through $i-1$. It is easy to see that F_i is also a forest. Each region of the plane covered at the beginning of the i^{th} iteration corresponds to a vertex in F_i; the vertices corresponding to those regions around which we are growing moats are called *active vertices*. Thus the contribution of the i^{th} iteration to the lower bound is t times the number of active vertices and its contribution to F_{sol} is t times the total degree of the active vertices in F_i.

We shall now show that the contribution of the i^{th} iteration to F_{sol} is at most $\left(2-\frac{2}{k}\right)$ times its contribution to the lower bound, where k is the total number of points. This then implies that F_{sol} has length at most $\left(2-\frac{2}{k}\right)$ times the lower bound. To prove this we need to show that the sum of the degrees of the active vertices in F_i is at most $\left(2-\frac{2}{k}\right)$ times the number of active vertices.

If F_i has n non-isolated vertices then it has at most $n-1$ edges so that the sum of the degrees of all these vertices is at most $2(n-1)$. Let $p \leq n$ be the number of active vertices. Note that every vertex which is a leaf in F_i is active. This is because if a leaf vertex was non-active then the edge incident to it is not essential and would not belong to F_{sol} (and hence to F_i). Given that every leaf is an active vertex, all non-active vertices that are not isolated have degree at least two. Thus the sum of the degree of the non-active non-isolated vertices is at least $2(n-p)$ and so the sum of the degrees of the active vertices is at most $2(p-1)$. Since $p \leq k$ the claim follows.

4.5 Steiner forests in graphs

We now extend the ideas developed for solving the steiner forest problem when we were given points in the plane to the more general setting of graphs. An instance of the problem is an indirected graph $G=(V,E)$, a non-negative length function $l: E \to \mathbf{R}^+$ and a requirement function $r: V \times V \to \mathbf{Z}^+$ which is symmetric *ie.* $r(u,v)=r(v,u)$ and denotes the number of edge-disjoint paths required between every pair of vertices. For the steiner forest problem the only requirements we have are that a pair of vertices be connected and hence $r: V \times V \to \{0,1\}$.

We first translate the requirement function which is defined on pairs of vertices to a function defined over sets of vertices $r: 2^V \to \mathbf{Z}^+$. Consider a subset of vertices $S \subset V$ and let there exist $u \in S, v \in \overline{S}$ such that $r(u,v)=1$. Then any steiner forest contains at least one edge from $\bigtriangledown(S)$. $r(S)$ equals 1 for all such sets and 0 for the others; note that $r(S)=r(\overline{S})$. We refer to $r(S)$ as the *requirement* of S since it is the number of edges that should go across $(S,\overline{S})$ in any solution. It is easy to see that a set of edges that which satisfies the requirements of all sets also satisfies the original connectivity requirements over pairs of vertices. Henceforth we view the steiner forest problem as one of picking a set of edges F such that F has at least $r(S)$ edges from $\bigtriangledown(S)$ for every set $S \subset V$.

It is now easy to formulate the steiner forest problem as an integer program. Let x_e be a 0/1 variable associate with edge e; $x_e = 1$ implies that e is in the solution. Then the condition that for a set $S \subset V$ there be at least $r(S)$ edges from $\bigtriangledown(S)$ in the solution can be written as $\sum_{e \in \bigtriangledown(S)} x_e \geq r(S)$. Relaxing the 0/1 requirement on x_e we obtain the following linear program

$$
\begin{array}{rrcll}
\text{minimize} & \sum_{e \in E} l_e x_e & & & \\
\text{subject to} & & & & \\
& \sum_{e \in \bigtriangledown(S)} x_e & \geq & r(S) & \text{(for all sets of vertices } S\text{)} \\
& x_e & \geq & 0 & \text{(for all edges } e\text{)}
\end{array}
$$

For any set S, $r(S) \leq 1$ and since we are minimizing $\sum_{e \in E} l_e x_e$ and for all edges e, $l_e \geq 0$, we will never set x_e to more than 1 in an optimum solution. Therefore we are justified in dropping the constraint $x_e \leq 1$ for all edges e from the linear program above.

The dual of this linear program is an assignment of non-negative values to subsets of vertices $y : 2^V \to \mathbf{R}^+$ such that certain 'packing' constraints are not violated.

$$
\begin{array}{rrcll}
\text{maximize} & \sum_{S \subset V} r(S) y_S & & & \\
\text{subject to} & & & & \\
& sum_{S : e \in \bigtriangledown(S)} y_S & \leq & l_e & \text{(for all edges } e\text{)} \\
& y_S & \geq & 0 & \text{(for all sets of vertices } S\text{)}
\end{array}
$$

An edge $e \in E$ 'feels' the dual variable y_S if $e \in \bigtriangledown(S)$. Thus the assignment of values to sets is such that the total value of the dual variables that an edge feels is no more than its length. e is *tight* if $\sum_{S : e \in \bigtriangledown(S)} y_S = l_e$.

The primal and dual linear programs form a covering/packing pair. The primal program is an assignment of values to edges such that they cover every set to the extent of its requirement. The dual program on the other hand is an assignment of values to sets so that any edge is not packed to more than its length.

Any feasible solution to the dual **LP** yields a lower bound on the value of the optimum steiner forest. In fact the lower bound that we used for the Euclidean case — the sum of the widths of the active moats in a non-overlapping collection of moats — is really the value of a feasible solution to the dual **LP** above. The width of a moat grown around a set of points S is just the value of the dual variable y_S; a moat is active iff $r(S) = 1$. The condition that the moats be non-overlapping is equivalent to the packing condition on the variables y_S.

4.5.1 The Algorithm

As for the geometric case we shall construct a steiner forest and a feasible solution to the dual **LP** and argue that the length of the steiner forest is no more than twice the value of the dual solution. This then yields a 2-approximation algorithm for the steiner forest problem. Note that assigning non-zero values to sets with $r(S) = 0$ does not contribute to the objective function value of the dual **LP**. Hence we shall assign non-zero values to only those sets with $r(S) = 1$.

Our procedure for finding a steiner forest and building a dual solution proceeds in steps. In each step we include an edge into our solution F; F is initially empty and our procedure ensures that at each step it is a forest. We also increase dual values so as to increase the lower bound; the feasibility of the dual solution is maintained at each step. The procedure for the i^{th}step is now the following.

> Let F denote the set of edges picked so far (in steps 1 through $i-1$). Identify all connected components of F. The connected components of F for which $r(\cdot)$ is 1 are called *active components*. Increase the dual variables of the active components, uniformly and simultaneously, till some edge goes tight. Include this edge in F.

The procedure stops when none of the components of F is active. Clearly, at this point F is a steiner forest. However it might contain a whole lot of non-essential edges — edges which can be deleted without violating any connectivity requirement. We delete all such edges to obtain the final solution F_{sol}.

4.5.2 The Analysis

Note that all edges in F (and hence F_{sol}) are tight. Therefore the length of F_{sol} can be rewritten as

$$\sum_{e \in F_{\text{sol}}} l_e = \sum_{e \in F_{\text{sol}}} \sum S : e \in \bigtriangledown(S) y_S = \sum S \subset V y_S \deg_{F_{\text{sol}}}(S)$$

where $\deg_{F_{\text{sol}}}(S)$ denotes the degree of the set S in F_{sol}.

To show that the length of F_{sol} is no more than twice the value of the dual solution we show that the contribution of each step to the length of F_{sol} is at most twice its contribution to the value of the dual solution. Let F_i be the forest whose vertices are the connected components of F at the i^{th} step and the edges are those edges of F_{sol} that were picked in and after the i^{th} iteration. A vertex of F_i is *active* if the component corresponding to it is active; let p be the number of active vertices. If δ was the amount we raised the dual variables by in the i^{th} step then the contribution of this step to the lower bound is δp and its contribution to the length of F_{sol} is equal to δ times the sum of the degrees of the active vertices in F_i. Thus we need to show that the sum of the degrees of the p active vertices in F_i is at most $2p$.

Let v be a leaf in F_i and $e \in F_{\text{sol}}$ the edge incident at it. If v was not active then the edge e would be non-essential and hence could not belong to F_{sol}. Thus every leaf of F_i is active. Now let n be the number of non-isolated vertices in F_i. Then the sum of the degrees of all the non-isolated vertices in F_i is at most $2(n-1)$. Every non-isolated non-active vertex in F_i has degree at least 2. Therefore the total degree of the active vertices in F_i is at most $2(n-1) - 2(n-p) = 2(p-1)$.

LECTURE 5

Uniform Connectivity

In the previous chapter we were only interested in certain pairs of vertices being connected; we had no requirements on the number of edge-disjoint paths between the pairs. Before considering the general problem, where for each vertex-pair we are specified a requirement which is the number of edge-disjoint paths between the pair that the subgraph we choose should contain, we consider the simpler case of uniform connectivity. In this setting the number of edge-disjoint paths required between every pair of vertices is the same, say k. Thus given a graph with lengths on the edges, we wish to pick a k-edge connected spanning subgraph of minimum total length.

5.1 Uniform Lengths

We first consider the even simpler setting of uniform edge-lengths. Thus the problem is now to find the smallest k-edge connected spanning subgraph in a given undirected graph $G = (V, E)$. Note that G is not complete and there is no length function associated with the edges. The problem, however, is still **NP**-hard since checking if G is Hamiltonian is equivalent to checking if the minimum 2-edge connected spanning subgraph in G is of size $|V|$.

Note that the degree of every vertex in a k-edge connected spanning subgraph of G must be at least k. This implies that the total number of edges in any k-edge connected spanning subgraph is at least $kn/2$ thus giving us a lower bound on the optimum value. We now show how to construct a k-edge connected spanning subgraph with only $k(n-1)$ edges; this gives us a $(2-2/n)$-approximation algorithm for this problem.

The procedure for finding a small k-edge connected spanning subgraph is easily described. We pick a maximal forest in G, remove all the edges of the forest and pick another maximal forest in the remaining graph. We continue in this manner till we have picked k forests. We claim that the edges of these k forests form a k-edge connected spanning subgraph.

Let F_i be the forest picked at the i^{th} step and $F = F_1 \cup F_2 \cup \ldots \cup F_k$ be all the edges picked. If $H = (V, F)$ is not k-edge connected then there exists a cut $(S, \overline{S})$ such that at most $k-1$ edges of F go across this cut and hence one of the forests, say F_i, has no edge across this cut. Since $G = (V, E)$ is k-edge connected, it has at least k edges going across $(S, \overline{S})$ and hence there exists an edge $e \in E - F$ which belongs to the cut $(S, \overline{S})$. Now note that $F_i \cup \{e\}$ is also a forest and this contradicts our assumption that F_i was a maximal forest. Hence $H = (V, F)$ is k-edge connected. Since each forest has at most $n-1$ edges the number of edges in F is at most $k(n-1)$.

Note that the lower bound we used for this $(2 - 2/n)$-approximation to the minimum k-edge connected spanning subgraph is independent of the graph G. Thus for any graph on n vertices we know that a k-edge connected spanning subgraph has at least $kn/2$ edges. In fact it is also known that any minimal k-edge connected spanning subgraph has at most $k(n-1)$ edges. Thus instead of the above procedure of finding maximal forests and deleting them we could also have found a minimal k-edge connected subgraph and this too would have yielded a $(2 - 2/n)$-approximation algorithm for this problem.

We now develop a more sophisticated lower bound on the size of the minimum 2-edge connected spanning subgraph. This lower bound which is based on the notion of a *tree carving* yields a $3/2$-approximation algorithm for the problem of finding a minimum 2-edge connected spanning subgraph.

Definition 5.1.1 *Given a graph $G = (V, E)$, a partition of the vertex set $V = [V_1, V_2, \ldots V_p]$ is a* tree carving *iff there exists a tree T over p nodes and a one-to-one correspondence between the nodes of T and the sets in the partition such that for any edge $e = (u, v)$ if u and v belong to different sets of the partition then the nodes in T corresponding to these sets are adjacent. The* size *of a* tree carving *is the number of sets in the partition* ie. *p.*

If G has a tree carving of size p then any 2-edge connected spanning subgraph of G has at least $2(p-1)$ edges. This together with the fact that any 2-edge connected spanning subgraph has at least n edges implies that $\max(n, 2(p-1))$ is a lower bound on the optimum value.

The classical approach to checking if a graph is 2-edge connected is via Depth First Search (`DFS`). We perform a `DFS` of the graph. Let vertex u be the parent of vertex v in the `DFS` tree. When backtracking from v we check to see if there is a back-edge going from some vertex in the subtree rooted at v to an ancestor of v. If there is no such edge then edge $e = (u, v)$ is a *cut-edge* (its removal disconnects the graph) and hence G is not 2-edge connected.

Our procedure for finding a small 2-edge connected spanning subgraph relies on similar ideas. We denote by F the edges in our solution; F is initially empty. We perform a `DFS` of the graph and include all the edges of the `DFS` tree in F. To make $H = (V, F)$ 2-edge connected we need to augment F with some back-edges. This we do as follows.

When backtracking from a vertex v during the `DFS` we check to see if the edge $e = (u, v)$ (u is the parent of v in the `DFS` tree) is a cut-edge in the subgraph $H = (V, F)$ which is the subgraph of edges picked so far. If e is indeed a cut-edge we pick a back-edge going from the subtree rooted at v to an ancestor of v; such an edge is guaranteed to exists since G is 2-edge connected. The back-edge that we pick is the one that goes *deepest*, *ie.* as close to the root as possible. This is how we augment F. Clearly the final set of edges forms a 2-edge connected spanning subgraph and this is our solution.

Note that during the above procedure some edges of the `DFS` tree threatened to be cut edges; we color these edges red. For each of these edges we included a back-edge in F. Let p be the number of subtrees formed by removing the red edges from the `DFS` tree. These p subtrees define a partition of the vertex set; vertices in one subtree form a set of the partition. We claim that this partition of the vertex set is a tree carving of G.

The tree T that we need for the tree carving is obtained from the `DFS` tree by shrinking each of the p subtrees into a single vertex. This also establishes a correspondence between the sets of the partition and the nodes of T. Note that every edge of T corresponds to a red edge of the `DFS` tree.

We now need to show that if for an edge $e = (u, v)$, u and v lie in different sets of the partition then the nodes of T corresponding to these sets are adjacent. This is clearly true if e is an edge of the `DFS` tree. Since we always included the highest going back-edge in F, no back-edge spans two red edges. Thus if e is a back-edge and u and v belong to different sets of the partition then the nodes of T corresponding to these sets must be adjacent.

We now have a tree carving of size p and a 2-edge connected spanning subgraph, $H = (V, F)$ with $(n-1)+(p-1)$ edges. Since $\mathrm{OPT} \geq \max(n, 2(p-1))$, where OPT denotes the size of the minimum 2-edge connected spanning subgraph, we have

$$|F| = n - 1 + p - 1 \leq \mathrm{OPT} + \frac{1}{2}\mathrm{OPT} = \frac{3}{2}\mathrm{OPT}$$

LECTURE 7

Knapsack, Bin Packing and Scheduling

In this lecture and the next we consider **NP**-hard number problems — problems whose hardness arises because of the numbers involved. In particular all the three problems that we consider, `knapsack`, `multiprocessor scheduling` and `bin-packing`, are trivially solvable if the numbers involved are all identical. Another common thread running through these problems is that they are amenable to some kind of *approximation schemes ie.* one can obtain solutions with value very close to the optimum value by investing a sufficiently large time. However, the nature of approximation schemes obtainable differ considerably and illustrate the various different possibilities that can arise in solving number problems.

7.1 Knapsack

Consider a sack which can hold a total weight at most W. We are given n objects each having a certain weight and profit associated with it. We wish to pack the sack with objects whose total weight does not exceed W and the sum of whose profits is maximum.

Let $w : \{1, \ldots, n\} \to \mathbf{R}^+$ be the weights and $p : \{1, \ldots, n\} \to \mathbf{Z}^+$ the profits associated with the objects; thus the i^{th} object has weight w_i and profit p_i. Further let $p_{\max}$ be the maximum profit. A simple dynamic programming scheme allows us to find the collection of objects which fit into the sack and maximize profit. This is done by determining for each i, $1 \le i \le n$, and each value of the total profit, p, a collection of objects from the subset $\{1, \ldots, i\}$ that achieves profit p and whose total weight is minimum. Given this information we can find the collection of objects from the set $\{1, \ldots, n\}$ that has total weight at most W and maximizes the profit.

Before detailing the procedure for computing all this information we need some notation. Let X be a table with n rows and $np_{\max} + 1$ columns whose $(i, j)^{\text{th}}$ entry, $1 \le i \le n$, $0 \le j \le np_{\max}$, denotes the least weight of any collection of objects from the subset $\{1, \ldots, i\}$ that achieves a profit j. We shall compute the entries of X row-by-row. To compute the first row note that since we are now allowed only to pick the first object the only profit values that we may have are 0 and p_1 and these correspond to weights 0 and w_1. Since the other profit values cannot be attained we set the corresponding entries of the first row to infinity. In general, while computing the entries of the $(i+1)^{\text{th}}$ row we are allowed to pick objects from the subset $\{1, \ldots, i+1\}$ as against the subset $\{1, \ldots, i\}$ which was what we were limited to when computing the entries of the i^{th} row. Thus we now have the flexibility of including the $(i+1)^{\text{th}}$ object when determining the minimum weight collection achieving a certain profit, p. If the $(i+1)^{\text{th}}$ object is not included then we are only picking objects from $\{1, \ldots, i\}$ and so the collection with minimum weight has weight $X(i,p)$. On the other hand any collection that picks the $(i+1)^{\text{th}}$ object and has total profit p has a profit $p - p_{i+1}$ from objects in the subset $\{1, \ldots, i\}$. Hence the minimum weight of any such collection is $w_{i+1} + X(i, p - p_{i+1})$. Since these are the only two options

$$X(i+1, p) = \min\{X(i,p), w_{i+1} + X(i, p - p_{i+1})\}$$

As mentioned before the solution to the `knapsack` problem is the rightmost column (maximum profit) for which the entry in the last row ($i = n$) is at most W. Each entry of X can be computed in constant time and hence the total time required to solve the `knapsack` problem is $O(n^2 p_{\max})$. Since a binary encoding of $p_{\max}$ has length only $\log p_{\max}$, the running time of this algorithm is not polynomial in the length of the input. However, if $p_{\max}$ is encoded in unary then the running time is polynomial; such an algorithm whose running time is polynomial in the length of the input when the numbers involved are encoded in unary is called a *pseudo-polynomial time* algorithm.

This pseudo-polynomial time algorithm can, for any choice of $\epsilon > 0$, be converted into a $(1-\epsilon)$-approximation algorithm with a running time that is polynomial in n and $1/\epsilon$[1]. Such a scheme, a collection of algorithms one for each $\epsilon > 0$, is called a *fully polynomial time approximation scheme*(`FPTAS`). Thus the `knapsack` problem admits a `FPTAS`; in fact all problems that have pseudo-polynomial time algorithms also have a `FPTAS` and vice-versa.

Note that the large running time of the dynamic programming algorithm was due to the large number of columns in X which in turn can be attributed to the large profits. Suppose we scale the profits by a suitable factor $K > 1$ to obtain a new profit function $\hat{p} : \{1, \ldots, n\} \to \mathbf{Z}^+$ defined as

$$\hat{p}_i = \lfloor p_i / K \rfloor$$

We now use this profit function in the dynamic programming scheme so that the running time is now $O(n^2 p_{\max}/K)$.

Let $I, \hat{I}$ be the collection of objects of weight at most W which maximizes the profit when profits are given by the function $p, \hat{p}$. The total profit of the objects in the collection $\hat{I}$ can now be bounded as

$$\sum_{i \in \hat{I}} p_i \geq K \sum_{i \in \hat{I}} \hat{p}_i \geq K \sum_{i \in I} \hat{p}_i \geq K \sum_{i \in I} \left(\frac{p_i}{K} - 1 \right) = \sum_{i \in I} p_i - K|I|$$

where the second inequality follows from the fact that the total profit of the collection $\hat{I}$ is more than that of the collection I when the profits are given by the function $\hat{p}$. The other inequalities follow from the observation that $p_i/K - 1 \leq \hat{p}_i \leq p_i/K$.

[1] A binary encoding of ϵ is of length $\log 1/\epsilon$. Since the running time is polynomial in $1/\epsilon$ and not $\log 1/\epsilon$, this algorithm is not polynomial if ϵ is part of the input. For this reason we assume that the constant ϵ is 'hard-wired' in the algorithm so that we have one algorithm for each choice of ϵ.

Since $\sum_{i \in I} p_i$ is the value of the optimum solution, OPT, the solution returned by our algorithm, $\hat{I}$, has value at least $\mathrm{OPT} - K|I| \geq \mathrm{OPT} - Kn$. Since $\mathrm{OPT} \geq p_{\max}$, choosing $K = \epsilon p_{\max}/n$ implies that the value of our solution $\hat{I}$ is at least $\mathrm{OPT} - \epsilon p_{\max} \geq (1-\epsilon)\mathrm{OPT}$. The running time of this $(1-\epsilon)$-approximation algorithm is $\mathrm{O}(n^2 p_{\max}/K) = \mathrm{O}(n^3/\epsilon)$ which is polynomial in n and $1/\epsilon$.

7.2 Bin Packing

Another problem similar to `knapsack` is that of packing objects in bins. Given a collection of n objects with sizes in the range (0,1] we wish to find the smallest number of unit-sized bins that can hold all these objects. Note that if all objects had the same size then any minimal solution is also an optimum solution and the problem is trivially solvable. The **NP**-hardness of this problem stems from the fact that objects have different sizes; let $s_1, s_2, \ldots s_k$ be the k different sizes and let n_i be the number of objects of size s_i so that $\sum_{i=1}^{k} n_i = n$.

An optimum solution to the `bin-packing` problem can be found in time $\mathrm{O}(n^{2k})$ using dynamic programming. A *configuration* of objects is a k-tuple $O = (o_1, o_2, \ldots o_k)$ where $0 \leq o_i \leq n_i$ for $1 \leq i \leq k$. Thus the number of distinct configurations is $\pi_{i=1}^{k}(n_i+1) = \mathrm{O}(n^k)$. The configuration O is *valid* if o_1 objects of size s_1, o_2 objects of size s_2 and so on, can fit into a single bin *ie.* $\sum_{i=1}^{k} o_i s_i \leq 1$. Clearly the number of valid configurations is no more than the number of configurations. A configuration $\hat{O} = (\hat{o}_1, \hat{o}_2, \ldots \hat{o}_k)$ *dominates* a configuration $O = (o_1, o_2, \ldots o_k)$ if for all $1 \leq i \leq k$, $\hat{o}_i \geq o_i$.

Let `bins(`$O = (o_1, o_2, \ldots o_k)$`)` denote the minimum number of bins required to hold the configuration O; clearly `bins(`O`)` is 1 for a valid configuration O. Given the value of `bins(`O`)` for all configurations O that the configuration $\hat{O}$ dominates, $\hat{O} \neq O$, we can determine `bins(`$\hat{O}$`)` as follows

$$\mathtt{bins}(\hat{O}) = 1 + \min_{\substack{O=(o_1,o_2,\ldots o_k)\\ \text{is a valid configuration}}} \mathtt{bins}(\hat{o}_1 - o_1, \hat{o}_2 - o_2, \ldots \hat{o}_k - o_k)$$

Note that the configurations can be totally ordered in a manner so that all configurations that a certain configuration dominates occur before it in this order. This is the order in which we compute the minimum number of bins required for each configuration. Since computing this quantity for each configuration requires $\mathrm{O}(n^k)$ comparisons, the total running time of this algorithm is $\mathrm{O}(n^{2k})$.

7.3 Multiprocessor Scheduling

Given p identical processors and n jobs with running times given by the function $t : \{1, \ldots n\} \to \mathbf{R}^+$ we wish to find a way of assigning jobs to processors such that the time T at which all processors finish processing the jobs assigned to them is minimized. Note that we are only interested in partitioning the n jobs among the p processors; the order in which a processor processes the jobs assigned to it is immaterial. The quantity T is also referred to as the *make-span* of the schedule.

Given an algorithm for the `bin-packing` problem we can compute a schedule with the minimum make-span as follows. We 'guess' a value for the make-span, say T, and scale the running times of all jobs by this factor. This gives us an instance of the `bin-packing` problem where the i^{th} object has size t_i/T. A packing of these objects into p bins yields a multiprocessor schedule with make-span T. Thus finding a schedule minimizing the make-span is the same as finding the least T for which the corresponding `bin-packing` instance requires only p bins.

LECTURE 8

Approximation Schemes

8.1 Multiprocessor Scheduling

In the previous chapter we gave a dynamic programming scheme for finding a schedule with minimum make-span in time $\mathrm{O}(n^{2k})$ where k was the number of distinct running times in the given `multiprocessor scheduling` problem instance. In this section we show a scaling technique to reduce the number of distinct running times. This leads to a faster algorithm; however the make-span of the schedule obtained is not the minimum possible. In fact for any $\epsilon > 0$ we show a $(1+\epsilon)$-approximation algorithm for the `multiprocessor scheduling` problem that has a running time of $\mathrm{O}(n^{1+2\ln_{1+\epsilon} 1/\epsilon})$. Note that this is polynomial in n only when ϵ is a constant and not part of the input. Thus for each choice of $\epsilon > 0$ we have a polynomial time $(1+\epsilon)$-approximation algorithm: we call this collection of algorithms a *polynomial time approximation scheme*(`PTAS`).

It is no loss of generality to assume that the longest processing time of a job is 1 *ie.* $t : \{1, \ldots n\} \rightarrow (0 \ldots 1]$. We employ two techniques to reduce the number of distinct running times. Jobs with sufficiently small running times can be assigned to the processors in a fairly even manner so that all processors have roughly the same load. Therefore we begin by setting aside all jobs with running times less than ϵ; these jobs are assigned to processors after we have taken care of the larger jobs (*elimination of short jobs*).

The jobs with running time larger than ϵ are now grouped together into a small number of groups. In particular all jobs with running time in the range $(\epsilon(1+\epsilon)^{i-1} \ldots \epsilon(1+\epsilon)^i]$ form the i^{th} group. Therefore the number of groups formed is the minimum k such that $\epsilon(1+\epsilon)^k \geq 1$ which implies that $k = \lceil \ln_{1+\epsilon} 1/\epsilon \rceil$.

We now construct another instance of the `multiprocessor scheduling` problem by setting the running time of a job in the i^{th} group $\epsilon(1+\epsilon)^{i-1}$. Note that by doing so we only decrease the running times of the jobs. Let $\hat{t}$ be the function that gives these new running times. Thus, $t_i/(1+\epsilon) \leq \hat{t}_i \leq t_i$ for all jobs i such that $t_i > \epsilon$. Since the number of distinct running times in this new instance is only k we can find a schedule S that minimizes the make-span for this instance in time $\mathrm{O}(n^{2k}) = \mathrm{O}(n^{1+2\ln_{1+\epsilon} 1/\epsilon})$.

While the schedule S has minimum make-span when the running times of the jobs are given by $\hat{t}$, its make-span need not be the minimum possible when the running times are given by t. Let S_{opt} be the schedule with minimum make-span, OPT, when running times are given by t. The the make-span of S_{opt} when running times are given by $\hat{t}$ is at most OPT. Hence the make-span of schedule S when running times are given by $\hat{t}$ is at most OPT. This implies that the make-span of S when running times are given by t is at most $(1+\epsilon)$OPT. We now have a $(1+\epsilon)$-approximation algorithm provided we can find a way of assigning the short jobs (those set aside in the first step) to the processors without increasing the make-span.

The rule for assigning the short jobs to the processors is to assign a job to that processor which is least loaded *ie.* the processor for which the sum of the processing times of the jobs already assigned to it is minimum. Thus starting with schedule S we follow the above rule of assigning the short jobs to obtain a schedule S_{fin}. The make-span of S_{fin} is at least as large as that of S; if it is no more we are done since then the make-span of S_{fin} is within $(1+\epsilon)$OPT. If the make-span of S_{fin} is strictly larger than that of S then we can use a different argument to show that the make-span of S_{fin} is no more than $(1+\epsilon)$OPT.

Consider the processor that takes longest to process all jobs assigned to it and hence determines the make-span. Since the make-span of S_{fin} is strictly larger than that of S, the last job assigned to this processor is a short job. Let T be the total load on this processor before the last job was assigned to it. Hence the make-span of S_{fin} is at most $T+\epsilon$.

From our rule for assigning short jobs it follows that every processor has a load of at least T in the schedule S_{fin}. This implies that no schedule can have a make-span less than T and hence OPT $\geq T$. Further, since the longest job has a processing time 1, OPT ≥ 1. Combining these two lower bounds on OPT gives a $(1+\epsilon)$OPT bound on the make-span of S_{fin}.

8.2 Bin Packing

Recall that the `bin-packing` problem is to find the minimum number of unit-sized bins in which a given collection of objects (with sizes in the range $(0 \ldots 1]$) can fit. Given a collection of objects it is easy to decide if they fit in one bin; we simply sum the sizes of the objects and check if this sum is less than one. Deciding whether the objects can be packed into two bins is also fairly easy; it can be done pseudo-polynomial time as follows. We create an instance of the `knapsack` problem where the weights and profits of the objects are their sizes and the capacity of the sack is one. A solution to this `knapsack` instance gives the maximum extent to which the first bin can be packed. If the sizes of the remaining objects sum to less than one they can fit into the second bin, else we shall need a third one. This yields a pseudo-polynomial time algorithm for the problem of deciding if a collection of objects can be packed into two bins; in fact this problem is *weakly* **NP**-*hard*. However it is **NP**-hard to decide if the objects fit into 3 bins or if they need 4 or more bins. Clearly, any algorithm for the `bin-packing` problem that achieves an approximation ratio strictly less than 4/3 can be used to answer the above decision problem and so no algorithm for the `bin-packing` problem can achieve an approximation ratio strictly less than 4/3 unless **P**=**NP**.

But what about absolute performance guarantees ? The above argument only says that the `bin-packing` problem cannot have a relative performance guarantee less than 4/3. It is completely consistent with the above argument that there is an algorithm which packs the objects into one more bin than the minimum number of bins, *ie.* has an absolute performance guarantee of 1. While we do not know of any algorithm with a constant absolute performance guarantee we shall show an algorithm that for any $\epsilon > o$, packs the objects into $(1+\epsilon)\mathrm{OPT}+1/\epsilon^2+2$ bins in time that is a polynomial in n and $1/\epsilon$. As before we view this as a collection of polynomial time algorithms, one for each $\epsilon > 0$, and we call this an *asymptotic fully polynomial time approximation scheme* (`AFPTAS`) to distinguish it from an `FPTAS` which is a $(1+\epsilon)$-approximation and has no additive terms in the performance guarantee.

We begin by formulating the `bin-packing` problem as an integer program. Recall that an instance of the `bin-packing` problem is specified by the sizes of the n objects; the objects have k different sizes and there are n_i objects of size s_i, $1 \leq i \leq k$. A valid configuration $O = (o_1, o_2, \ldots o_k)$ is a k-tuple such that o_i objects of size s_i $(1 \leq i \leq k)$ fit into a bin *ie.*, $\sum_{i=1}^{k} o_i s_i \leq 1$. Let $\mathcal{O}$ denote the set of valid configurations. We associate a variable x_O with a configuration $O \in \mathcal{O}$ the value of which represents the number of bins which have this configuration. Thus $x_O \in \mathbf{Z}^+$ and we need to minimize $\sum_{O\in\mathcal{O}} x_O$. To ensure that all objects are packed we only need to ensure that for all i, $1 \leq i \leq k$, the total number of times an object of size s_i appears in the bins is at least n_i *ie.* $\sum_{O=(o_1,o_2,\ldots o_k)\in\mathcal{O}} x_O o_i \geq n_i$, $1 \leq i \leq k$. This yields the following integer program

$$
\begin{array}{llll}
\text{minimize} & \sum_{O\in\mathcal{O}} x_O & & \\
\text{subject to} & & & \\
& \sum_{O=(o_1,o_2,\ldots o_k)\in\mathcal{O}} x_O o_i \geq n_i & & (\text{for all } 1 \leq i \leq k) \\
& x_O \in \mathbf{Z}^+ & & (\text{for all } O \in \mathcal{O})
\end{array}
$$

Relaxing the integrality constraint on the variable x_O to $x_O \geq 0$ gives us a linear program. Although this linear program has exponentially many variables it can be solved in polynomial time using the ellipsoid method. An important observation about this **LP** is that it has only k constraints. Therefore any *basis solution*, and hence any optimum solution to this **LP** has non-zero values for at most k variables. If we round up these non-zero values we get an integer solution of value at most k more than the value of the optimum fractional solution. Since the optimum fractional solution is a lower bound on the minimum number of bins required we have a packing of the objects into bins that uses at most k additional bins.

A natural approach to improve the additive k performance guarantee that the above simple rounding scheme provides is to reduce k, *ie.* reduce the number of distinct piece sizes. We achieve this by employing two techniques one of which is the same as that used for `multiprocessor scheduling` *viz.* eliminating objects of small size. We set aside all objects of size less than ϵ; let m be the number of objects remaining. Further let S denote the sum of the sizes of all the objects *ie.*, $S = \sum_{i=1}^{k} s_i n_i$. Clearly $\mathrm{OPT} \geq S \geq \epsilon m$.

The other technique that we use to reduce the number of distinct piece sizes is *linear grouping.* We group the objects into a small number of groups each of which (except possible one) is of cardinality h. To obtain these groups we order the objects by decreasing size (objects of the same size are ordered arbitrarily); the first h objects form the first group, the next h the second group and so on. Note that the last group might contain less than h objects. We now create another instance with a smaller number of distinct piece sizes by increasing the size of an object to that of the largest object in its group. Thus the number of different piece sizes in this instance is the same as the number of groups and is $\lceil m/h \rceil$.

We can now round up the optimum fractional solution for this new instance to obtain a solution with at most $\lceil m/h \rceil$ more bins than the optimum for this instance (say OPT'). Since in obtaining this new instance we only increased the sizes of the objects we can use this solution to pack the large objects (size more than ϵ) of the original instance into $\mathrm{OPT}' + \lceil m/h \rceil$ bins.

How does OPT' relate to OPT or how do the optimum values of the two instances relate ? We now show a way of packing the objects of the new instance into the objects of the original instance. Thus an object that belongs to the i^{th} group of the new instance is packed into an object of the $(i-1)^{\text{th}}$ group of the original instance — this is possible since each object in the $(i-1)^{\text{th}}$ group has size at least as large as that of any object in the i^{th} group. Such a packing is possible for all objects except those in the first group; we pack these h objects into h new bins. We have thus shown a way of packing the objects of the new instance into the objects of the original instance and an additional h bins. This implies that $\mathrm{OPT}' \leq \mathrm{OPT} + h$ and so we have a way of packing the large objects of the original instance into $\mathrm{OPT} + h + \lceil m/h \rceil$ bins.

We now need to choose the parameter h (the size of the groups) so that we have the desired approximation guarantee. This is achieved by choosing $h = \lceil \epsilon^2 m \rceil$ which implies that $h \leq \epsilon^2 m + 1 \leq \epsilon \mathrm{OPT} + 1$. Therefore the number of bins used is at most $(1+\epsilon)\mathrm{OPT} + 1/\epsilon^2 + 2$.