CS787: Advanced Algorithms	
Scribe: Shuchi Chawla	**Lecturer:** Shuchi Chawla
Topic: Introduction and Greedy Algorithms	**Date:** Sept 5, 2007

1.1 Introduction and Course Overview

In this course we will study techniques for designing and analyzing algorithms. Undergraduate algorithms courses typically cover techniques for designing exact, efficient (polynomial time) algorithms. The focus of this course is different. We will consider problems for which polynomial time exact algorithms are not known, problems under stringent resource constraints, as well as problems for which the notion of optimality is not well defined. In each case, our emphasis will be on designing efficient algorithms with provable guarantees on their performance. Some topics that we will cover are as follows:

- **Approximation algorithms for NP-hard problems**. NP-hard problems are those for which there are no polynomial time exact algorithms unless P = NP. Our focus will be on finding near-optimal solutions in polynomial time.
- **Online algorithms**. In these problems, the input to the problem is not known apriori, but arrives over time, in an "online" fashion. The goal is to design an algorithm that performs nearly as well as one that has the full information before-hand.
- **Learning algorithms**. These are special kinds of online algorithms that "learn" or determine a function based on "examples" of the function value at various inputs. The output of the algorithm is a concise representation of the function.
- **Streaming algorithms**. These algorithms solve problems on huge datasets under severe storage constraints—the extra space used for running the algorithm should be no more than a constant, or logarithmic in the length of the input. Such constraints arise, for example, in high-speed networking environments.

We begin with a quick revision of basic algorithmic techniques including greedy algorithms, divide & conquer, dynamic programming, network flow and basic randomized algorithms. Students are expected to have seen this material before in a basic algorithms course.

Note that some times we will not explicitly analyze the running times of the algorithms we discuss. However, this is an important part of algorithm analysis, and readers are highly encouraged to work out the asymptotic running times themselves.

1.2 Greedy Algorithms

As the name suggests, greedy algorithms solve problems by making a series of myopic decisions, each of which by itself solves some subproblem optimally, but that altogether may or may not be

optimal for the problem as a whole. As a result these algorithms are usually very easy to design but may be tricky to analyze, and don't always lead to the optimal solution. Nevertheless there are a few broad arguments that can be utilized to argue their correctness. We will demonstrate two such techniques through a few examples.

1.2.1 Interval Scheduling

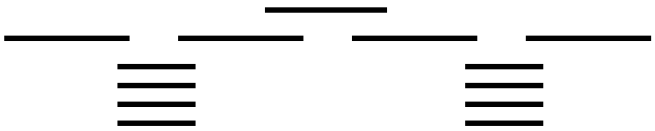
Given: n jobs, each with a start and finish time (s_i, f_i).

Goal: Schedule the maximum number of (non-overlapping) jobs on a single machine.

To apply the greedy approach to this problem, we will schedule jobs successively, while ensuring that no picked job overlaps with those previously scheduled. The key design element is to decide the order in which we consider jobs. There are several ways of doing so. Suppose for example, that we pick jobs in increasing order of size. It is easy to see that this does not necessarily lead to the optimal solution (see the figure below for a counter-example). Likewise, scheduling jobs in order of their arrivals (start times), or in increasing order of the number of conflicts that they have, also does not work.

(a) Bad example for the shortest job first algorithm

(b) Bad example for the earliest start first algorithm



(c) Bad example for the fewest conflicts first algorithm

We will now show that picking jobs in increasing order of finish times gives the optimal solution. At a high level, our proof will employ induction to show that at any point of time the greedy solution is no worse than any *partial* optimal solution up to that point of time. In short, we will show that *greedy always stays ahead.*

Theorem 1.2.1 *The "earliest finish time first" algorithm described above generates an optimal schedule for the interval scheduling problem.*

Proof: Consider any solution S with at least k jobs. We claim by induction on k that the greedy algorithm schedules at least k jobs and that the first k jobs in the greedy schedule finish no later

than the first k jobs in the chosen solution. This immediately implies the result because it shows that the greedy algorithm schedules at least as many jobs as the optimal solution.

We now prove the claim. The base case, $k = 0$ is trivial. For the inductive step, consider the $(k+1)$th job, say J, in the solution S. Then this job begins after the kth job in S ends, which happens after the kth job in the greedy schedule ends (by the induction hypothesis). Therefore, it is possible to augment the greedy schedule with the job J without introducing any conflicts. The greedy algorithm finds a candidate to augment its solution and in particular, picks one that finishes no later than the time at which J ends. This completes the proof of the claim. ■

1.2.2 Minimum Spanning Tree

Our second problem is a network design problem.

Given: A graph G with n vertices or machines, and m edges or potential cables. Each edge has a specified length—ℓ_e for edge e.

Goal: Form a network connecting all the machines using the minimum amount of cable.

Our goal is to select a subset of the edges of minimum total length such that all the vertices are connected. It is immediate that the resulting set of edges forms a spanning tree—every vertex must be included; Cycles do not improve connectivity and only increase the total length. Therefore, the problem is to find a spanning tree of minimum total length.

The greedy approach to this problem involves picking edges one at a time while avoiding forming cycles. Again, the order in which we consider edges to be added to the tree forms a crucial component of the algorithm. We mention two variants of this approach below, both of which lead to optimal tree constructions:

1. **Kruskal's algorithm**: Consider edges in increasing order of length, and pick each edge that does not form a cycle with previously included edges.
2. **Prim's algorithm**: Start with an arbitrary node and call it the root component; at every step, grow the root component by adding to it the shortest edge that has exactly one end-point in the component.

A third greedy algorithm, called **reverse-delete**, also produces an optimal spanning tree. This algorithm starts with the entire set of edges and deletes edges one at a time in decreasing order of length unless the deletion disconnects the graph.

We will now analyze Kruskal's algorithm and show that it produces an optimal solution. The other two algorithms can be analyzed in a similar manner. (Readers are encouraged to work out the details themselves.) This time we use a different proof technique—an *exchange argument*. We will show that we can transform any optimal solution into the greedy solution via local "exchange" steps, without increasing the cost (length) of the solution. This will then imply that the cost of the greedy solution is no more than that of an optimal solution.

Theorem 1.2.2 *Kruskal's algorithm finds the minimum spanning tree of a given graph.*

Proof: Consider any optimal solution, T^*, to the problem. As described above, we will transform this solution into the greedy solution T produced by Kruskal's algorithm, without increasing its length. Consider the first edge in increasing order of length, say e, that is in one of the trees T and T^* but not in the other. Then $e \in T \setminus T^*$ (convince yourself that the other case, $e \in T^* \setminus T$, is not possible). Now consider adding e to the tree T^*, forming a unique cycle C. Naturally T does not contain C, so consider the most expensive edge $e' \in C$ that is not in T. It is immediate that $\ell_{e'} \leq \ell_e$, by our choice of e, and because e' belongs to one of the trees and not the other. Let T_1^* be the tree T^* minus the edge e' plus the edge e. Then T_1^* has total length no more than T^*, and is closer (in hamming distance[1]) to T than T^* is. Continuing in this manner, we can obtain a sequence of trees that are increasingly closer to T in hamming distance, and no worse than T^* in terms of length; the last tree on this sequence is T itself. ■

1.2.3 Set Cover

As we mentioned earlier, greedy algorithms don't always lead to globally optimal solutions. In the following lecture, we will discuss one such example, namely the set cover problem. Following the techniques introduced above we will show that it nevertheless produces a near-optimal solution. The set cover problem is defined as follows:

Given: A universe U of n elements. A collection of subsets $S_1, \cdots, S_k$ of U.

Goal: Find the smallest collection $\mathcal{C}$ of subsets that covers U, that is, $\cup_{S \in \mathcal{C}} S = U$.

[1]We define the hamming distance between two trees to be the number of edges that are contained in one of the trees and not the other.

CS787: Advanced Algorithms

Scribe: David Hinkemeyer and Dalibor Zelený **Lecturer:** Shuchi Chawla
Topic: Greedy Algorithms, Divide and Conquer, and DP **Date:** September 7, 2007

Today we conclude the discussion of greedy algorithms by showing that certain greedy algorithms do not give an optimum solution. We use set cover as an example. We argue that a particular greedy approach to set cover yields a good approximate solution. Afterwards, we discuss the divide and conquer technique of algorithm design. We give two examples of divide and conquer algorithms, and discuss lower bounds on the time complexity of sorting. Finally, we introduce dynamic programming using the problem of weighted interval scheduling as an example.

2.1 Greedy Set Cover

Previously, we had seen instances where utilizing a greedy algorithm results in the optimal solution. However, in many instances this may not be the case. Here we examine an example problem in which the greedy algorithm does not result in the optimal solution and compare the size of the solution set found by the greedy algorithm relative to the optimal solution.

The Set Cover Problem provides us with an example in which a greedy algorithm may not result in an optimal solution. Recall that a greedy algorithm is one that makes the "best" choice at each stage. We saw in the previous lecture that a choice that looks best locally does not need to be the best choice globally. The following is an example of just such a case.

2.1.1 Set Cover Problem

In the set cover problem, we are given a universe U, such that $|U| = n$, and sets $S_1, \ldots, S_k \subseteq U$. A *set cover* is a collection C of some of the sets from $S_1, \ldots, S_k$ whose union is the entire universe U. Formally, C is a set cover if $\bigcup_{S_i \in C} S_i = U$. We would like to minimize $|C|$.

Suppose we adopt the following greedy approach: In each step, choose the set S_i containing the most uncovered points. Repeat until all points are covered.

Suppose $S_1 = \{1, 2, 3, 8, 9, 10\}$, $S_2 = \{1, 2, 3, 4, 5\}$, $S_3 = \{4, 5, 7\}$, $S_4 = \{5, 6, 7\}$, and $S_5 = \{6, 7, 8, 9, 10\}$. Figure 2.1.1 depicts this case. We wish to see how close a greedy algorithm comes to the optimal result. We can see that the optimal solution has size 2 because the union of sets S_2 and S_5 contains all points.

Our Greedy algorithm selects the largest set, $S_1 = \{1, 2, 3, 8, 9, 10\}$. Excluding the points from the sets already selected, we are left with the sets $\{4, 5\}$, $\{4, 5, 7\}$, $\{5, 6, 7\}$, and $\{6, 7\}$. At best, we must select two of these remaining sets for their union to encompass all possible points, resulting in a total of 3 sets. The greedy algorithm could now pick the set $\{4, 5, 7\}$, followed by the set $\{6\}$.

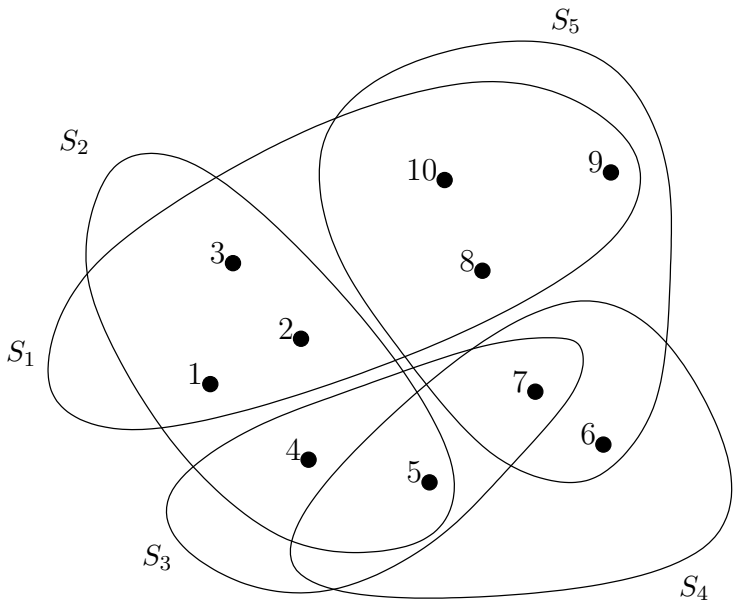


Figure 2.1.1: An instance of a set cover problem.

2.1.2 Upper bound on Greedy Set Cover Problem

In the previous example we saw a case where the greedy algorithm did not produce the optimal solution. In the following theorem we show that size of the set cover found by the greedy algorithm is bounded above by a function of the size of the optimal solution and the number of elements in the universe U.

Theorem 2.1.1 *Suppose an optimal solution contained m sets. Our greedy algorithm finds a set cover with at most $m \log_e n$ sets.*

Proof: Let the universe U contain n points, and suppose that the optimal solution has size m. The first set picked by the greedy algorithm has size at least n/m. Therefore, the number of elements of U we still have to cover after the first set is picked is

$$n_1 \leq n - n/m = n(1 - 1/m).$$

Now we are left with n_1 elements that we have to cover. At least one of the remaining sets S_i must contain at least $n_1/(m-1)$ of such points them because otherwise the optimum solution would have to contain more than m sets. After our greedy algorithm picks the set that contains the largest number of uncovered points, it is left with $n_2 \leq n_1 - n_1/(m-1)$ uncovered nodes. Notice that $n_2 \leq n_1(1 - 1/(m-1)) \leq n_1(1 - 1/m) \leq n(1 - 1/m)^2$. In general, we then have

$$n_{i+1} \leq n_i(1 - 1/m) \leq n(1 - 1/m)^{i+1}. \tag{2.1.1}$$

We would like to determine the number of stages after which our greedy algorithm will have covered all elements of U. This will correspond to the maximum number of sets the greedy algorithm has

to pick in order to cover the entire universe U. Suppose that it takes k stages to cover U. By 2.1.1, we have $n_k \leq n(1-1/m)^k$, and we need this to be less than one.

$$\begin{aligned} n(1-1/m)^k &< 1 \\ n(1-1/m)^{m\frac{k}{m}} &< 1 \\ (1-1/m)^{m\frac{k}{m}} &< 1/n \\ e^{-\frac{k}{m}} &< 1/n \ldots \text{using relation} (1-x)^{\frac{1}{x}} \approx 1/e \\ k/m &> \log_e n \\ k &< m \log_e n \end{aligned}$$

From this we can see that the size of the set cover picked by our greedy algorithm is bounded above by $m \log_e n$. ■

We have just shown that the greedy algorithm gives a $\mathcal{O}(\log_e n)$ approximation to optimal solution of the set cover problem. In fact, no polynomial time algorithm can give a better approximation unless P = NP.

2.2 Divide and Conquer

Whenever we use the divide and conquer method to design an algorithm, we seek to formulate a problem in terms of smaller versions of itself, until the smaller versions are trivial. Therefore, divide and conquer algorithms have a recursive nature. The analysis of the time complexity of these algorithms usually consists of deriving a recurrence relation for the time complexity and then solving it.

We give two examples of algorithms devised using the divide and conquer technique, and analyze their running times. Both algorithms deal with unordered lists.

Sorting We describe mergsort—an efficient recursive sorting algorithm. We also argue that the time complexity of this algorithm, $\mathcal{O}(n \log n)$, is the best possible worst case running time a sorting algorithm can have.

Finding the k-th largest element Given an unordered list, we describe how to find the k-th largest element in that list in linear time.

2.2.1 Mergesort

The goal of sorting is to turn an unordered list into an ordered list. Mergesort is a recursive sorting algorithm that sorts an unordered list. It consists of two steps, none of which are done if the length of the list, n, is equal to one because sorting a list with one element is trivial.

1. *Split* the list in two halves that differ in length by at most one, and sort each half recursively.

2. *Merge* the two sorted lists back together into one big sorted list. We maintain pointers into each of the two sorted lists. At the beginning, the pointers point to the first elements of the lists. In each step, we compare the two list elements being pointed to. We make the smaller one be the next element of the big sorted list, and advance the pointer that points to that element (i.e. we make it point to the next element of the list). We continue in this fashion until both pointers are past the ends of the lists which they point into. The process is illustrated in Figure 2.2.2.

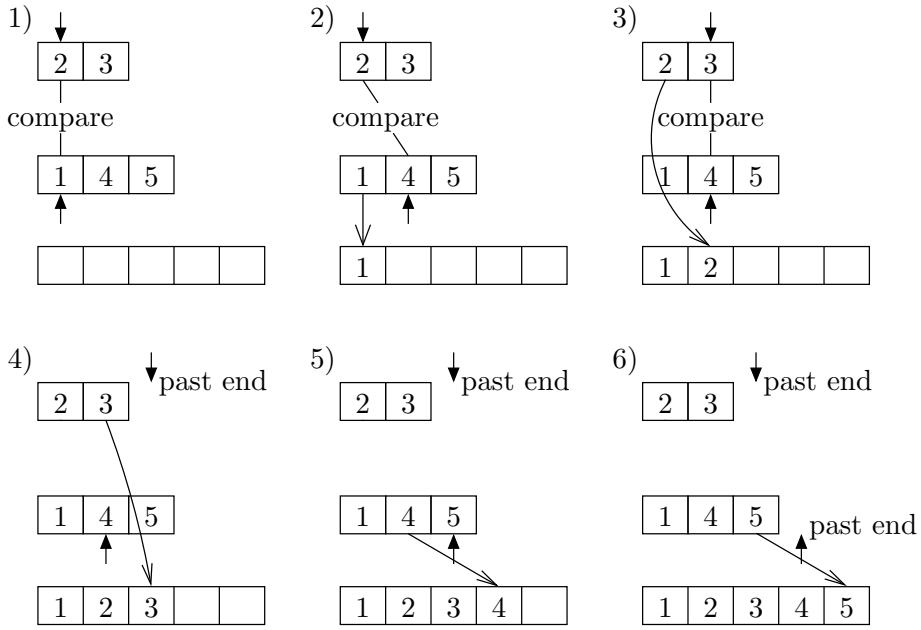


Figure 2.2.2: Merging two sorted lists into one sorted list in linear time.

Let $T(n)$ denote the time it takes mergesort to sort a list of n elements. The split step in the algorithm takes no time because we don't need to take the list and generate two smaller lists. It takes mergesort time $T(n/2)$ to sort each of the two lists, and it takes linear time to merge two smaller sorted lists into one big sorted list. Therefore, we get a recursive definition of $T(n)$ of the form

$$T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n).$$

Solving this recurrence relation yields

$$T(n) = \mathcal{O}(n \log n).$$

We describe one way of solving the recurrence above. We can view the computation of mergesort as a tree (see Figure 2.2.3). Each node in the tree corresponds to one recursive application of mergesort. The amount of time each recursive call contributes to the total running time is shown in the node, and any recursive invocations made from some invocation of the mergesort algorithm are represented by children of the node representing that invocation. The former represents the $\mathcal{O}(n)$

part of the recurrence relation for $T(n)$, the latter represents the $2T\left(\frac{n}{2}\right)$ part of that recurrence. Notice that the tree has at most $\log n$ levels because the smallest list we ever sort has size 1, and that the values in the nodes in Figure 2.2.3 add up to $\mathcal{O}(n)$ at each level of the tree. Hence, $T(n) = \mathcal{O}(n \log n)$.

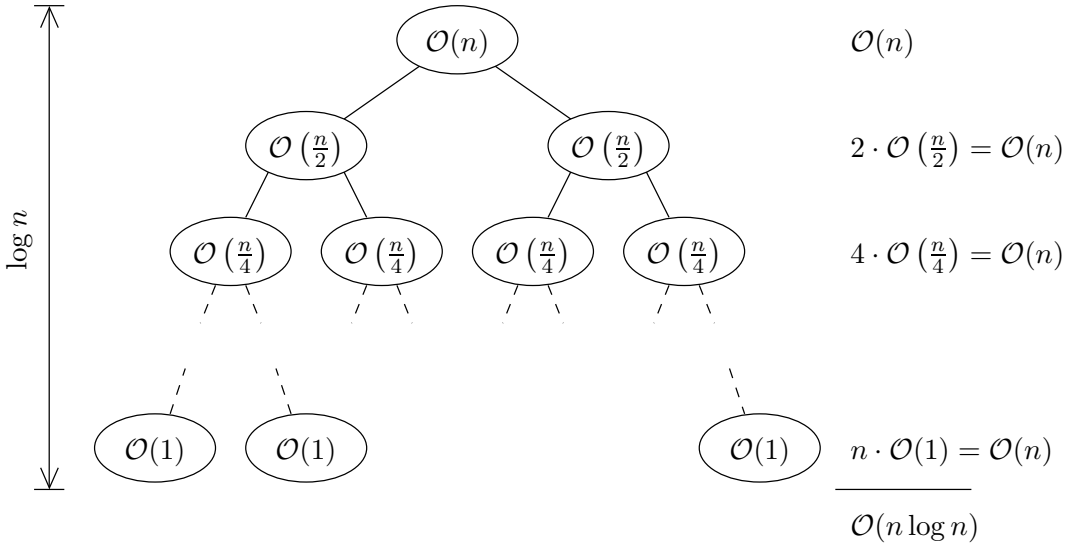


Figure 2.2.3: Computation of the time complexity of mergesort.

2.2.2 Lower Bound on the Complexity of Sorting

We define a special setting called the comparison-based model, for which we demonstrate a lower bound on the worst case complexity of any sorting algorithm.

In the *comparison-based model*, comparisons of pairs of list elements are the driving force of a sorting algorithm. In other words, the sorting algorithm branches only after comparing a pair of list elements. We can view a sorting algorithm in the comparison-based model as a decision tree where nodes correspond to comparisons of elements. Leaves of the decision tree correspond to permutations of the list elements. Each leaf represents a permutation of list elements that turns some unordered lists into ordered lists.

Without loss of generality, assume that no two elements of any list are equal. Then every comparison of list elements a and b has two possible outcomes, namely $a < b$ or $a > b$. Thus, the decision tree is a binary tree.

Theorem 2.2.1 *In the comparison-based model, any sorting algorithm takes* $\Omega(n \log n)$ *time in the worst case.*

Proof: Suppose the sorting algorithm is given an unordered list of length n. There are $n!$ possible orderings of the list elements. Assuming that no two elements of the list are equal, only one of these orderings represents a sorted list. The algorithm must permute the elements of each ordering (including the one that represents a sorted list) in a different way in order to have a sorted list when it finishes. This means that the decision tree representing the sorting algorithm must have at

least $n!$ leaves. The minimum height of the tree is $\log(n!) = \mathcal{O}(\log(n^n)) = \mathcal{O}(n \log n)$. Therefore, the sorting algorithm must make at least $n \log n$ comparisons in order to turn some ordering of list elements into a sorted one. ■

2.2.3 Finding the k-th Largest Element

Suppose that we have an unordered list, and once again assume that no two elements of the list are equal. We would like to find the k-th largest element in the list. We present some simple, but less efficient algorithms, and then give a linear time algorithm for finding the k-th largest element.

We could sort the list before we search. We can easily find the k-th largest element of a sorted list in constant time. However, sorting the list takes at least $\mathcal{O}(n \log n)$ time, which will cause this approach to finding the k-th largest element of a list run in $\mathcal{O}(n \log n)$ time. We can do better than that.

We could maintain the largest k elements in a heap and then go through the list elements one by one, updating the heap as necessary. After seeing all list elements, we use the heap to find the k-th largest element by removing $k-1$ elements from the root of the heap and then looking at the root of the heap. This approach takes $\mathcal{O}(n \log(k))$ time. We now describe an algorithm that find the k-th largest element in linear time.

Suppose that we can find the median of the list elements in linear time (we will show that this can be done afterwards). If the k-th largest element of the original list is the median, we are done. Otherwise we recursively search for the k-th largest element of the original list either in the list of elements that are greater than the median or in the list of elements that are less than the median. We need to search in only one of those, and we can forget about the other. The k-th largest element of the original list is the k'-th largest element of the new smaller list, where k' need not be the same as k.

We demonstrate how the algorithm works by using it to find the 5th largest element in some 15-element list. The process is illustrated in Figure 2.2.4. First of all, the algorithm finds the median of the original list and realizes that the median is not the 5th largest element of the list. Since the median is the 8th largest element of the list, the algorithm looks at elements of the list that are greater than the median. The 5th largest element of the original list is also the 5th largest element in the list of elements greater than the median. The algorithm now searches for the 5th largest element in the new list (second row in Figure 2.2.4). The median of the new list is its 4th largest element. Therefore, the algorithm looks for the 5th largest element of the original list in the list of elements that are smaller than the median it found. In that list, the 5th largest element from the original list is the 1st largest element. The algorithm continues in this fashion until it ends with a single-element list whose median is the 5th largest element of the original list.

In each step, the algorithm searches for the k-th largest element in a list that's half the size of the list from the previous step. The time complexity of this search is, then,

$$T(n) = \mathcal{O}(n) + T\left(\frac{n}{2}\right).$$

Solving this recurrence yields

$$T(n) = \mathcal{O}(n),$$

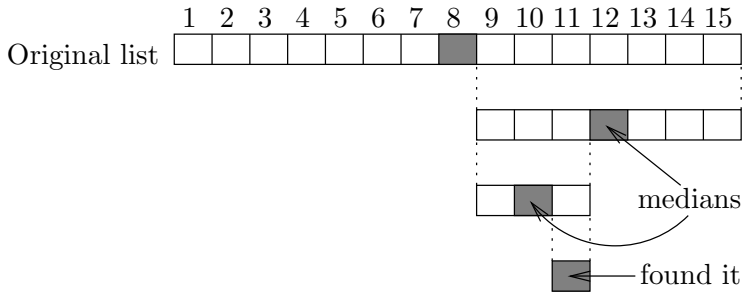


Figure 2.2.4: Finding the 5th largest element in a list of 15 elements. For simplicity of presentation, the list elements are already ordered, but this isn't necessary for the algorithm to work. The 5th largest element has index 11. Each "row" in the picture represents one recursive invocation of the algorithm.

so we have a linear time algorithm for finding the k-th largest element.

Finally, we show that it is possible to find the median of any list in linear time. We start by finding a near-median of the list. For our purposes, a near-median is an element of the list that is greater than at least 3/10 and less than at least 3/10 of the list elements. This happens in three steps

1. Split the list into blocks of five elements (and possibly one block with fewer elements).
2. Find the median in each of these blocks.
3. Recursively find the median of the medians found in the previous step.

The median found by the recursive call is greater than at least 3/10 of the list elements becaue it is greater than one half of the medians of the five-element blocks, and all those medians are greater than two elements in their respective blocks. By similar reasoning, this median is smaller than at least 3/10 of the list elements.

If the median of the original list is the near-median of the list we are currently looking at, we are done. Otherwise we recursively search for the median of the original list either in the list of elements that are greater than the near-median or in the list of elements that are less than the near-median. We need to search in only one of those, and we can forget about the other. Thus, in each step, we search for the median of the original list in a list that is at most 7/10 the size of the list from the previous step. The time complexity of this search is, then,

$$T(n) = \mathcal{O}(n) + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right).$$

The term $T\left(\frac{n}{5}\right)$ appears because we need to find a median of a list with $n/5$ elements when we want to find a median of a list of n elements. The $T\left(\frac{7n}{10}\right)$ appears because we subsequently search for the median of the list of length n in a list of length at most $7n/10$. The $\mathcal{O}(n)$ term appears because it takes linear time to find the medians of all the 5-element blocks the algorithm creates. If we solve the recurrence, we get

$$T(n) = \mathcal{O}(n),$$

which is what we wanted.

2.3 Dynamic Programming

Dynamic programming utilizes recursion while maintaining information about subproblems that have already been solved (this is called *memoization*), allowing us to to efficiently find a solution to a subproblem. We must break up the problem into a polynomial number of subproblems if we want to use memoization.

We investigate the weighted interval scheduling problem and the difficulties associated with breaking it down into subproblems, and show how to use dynamic programming to find an optimal solution in polynomial time.

2.3.1 Weighted Interval Scheduling Problem

In the weighted interval scheduling problem, we want to find the maximum-weight subset o non-overlapping jobs, given a set J of jobs that have weights associated with them. Job $i \in J$ has a start time, an end time, and a weight which we will call w_i. We seek to find an optimum schedule, which is a subset S of non-verlapping jobs in J such that the sum of the weights of the jobs in S is greater than the sum of the weights of jobs in all other subsets of J that contaqin non-overlapping jobs. The weight of the optimum schedule is defined as

$$\min_{S \subseteq J} \sum_{i \in S} w_i.$$

In Figure 2.3.5, the optimum schedule is one that contains only job 2, and the weight of the optimum schedule is 100.

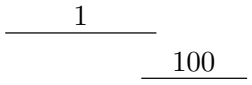


Figure 2.3.5: An instance of the weighted interval scheduling problem with two overlapping jobs. Job weights are shown above the lines representing them. Job 1 has weight $w_1 = 1$, job 2 has weight $w_2 = 100$. This instance can be used to spite the greedy approach that solves the non-weighted case of this problem.

We could try to use the greedy algorithm for job scheduling where jobs don't have weights. However, Figure 2.3.5 shows an instance where this algorithm fails to pick the optimum schedule because it would pick the first job with weight 1 and not the second job of weight 100. We can easily spite other greedy approaches (shortest job first, job overlapping with the least number of jobs first, earliest starting time first) using same counterexamples as in the non-weighted version and giving each job a weight of 1. We need to resort to a different method of algorithm design.

Consider the following algorithm: We look at the first job, consider the two possibilities listed below, and pick the better one. This approach yields the optimum solution because it tries all possible schedules.

1. Schedule the first job and repeat the algorithm on the remaining jobs that don't overlap with it.

2. Drop the first job and apply the algorithm to all jobs except for this first job.

The weights of the two schedules are as follows.

1. w_1+(max weight that can be scheduled after picking job 1)

2. max weight that can be scheduled after dropping job 1

Unfortunately, this approach yields an exponential time algorithm because in the worst case, each recursive application of this algorithm reduces the problem size by one job, which gives us

$$T(n) = \mathcal{O}(1) + 2T(n-1)$$

as a recurrence for the running time $T(n)$ of this algorithm. Solving this recurrence yields

$$T(n) = \mathcal{O}(2^n),$$

which is bad. This time complexity stems from the fact that the algorithm tries all possible schedules. We get the same running time if, instead of considering whether to include the first job in the schedule, we consider whether to include a randomly picked job in the schedule.

Suppose that the jobs are sorted in the order of starting time. Now if we apply our recursive algorithm that considers whether to include the first job in the optimal schedule or whether to exclude it, we notice that it only looks at n subproblems where n is the number of jobs. Each subproblem consists of finding the optimum schedule for jobs i through n, with $1 \leq i \leq n$, and the solution to each subproblem depends only on subproblems of smaller size. Therefore, if we remember solutions to subproblems of smaller sizes that have already been solved, we can eliminate many redundant calculations the divide and conquer algorithm performs.

For $1 \leq i \leq n$, let value(i) be the maximum total weight of non-overlapping jobs with indices at least i. From our earlier discussion, we see that value(i) = $\max\{w_i + \text{value}(j), \text{value}(i+1)\}$ where j is equal to the index of the first job that starts after job i ends.

We now use the following algorithm to find the weight of the optimum schedule.

1. Sort jobs in ascending order of starting time.

2. For $1 \leq i \leq n$, find the smallest $j > i$ such that job j doesn't overlap with job i.

3. For $i = n$ down to 1, compute value(i) and store it in memory.

4. The weight of the optimum schedule is value(1).

If we want to recover the optimum schedule, we would remember the values of value(i) as well as the option that led to it.

The reader shall convince himself that once the list of jobs is sorted in ascending order of starting time, it takes time $\mathcal{O}(n \log n)$ to carry out step 2 of our algorithm. Hence, the running time of the algorithm is

$$T(n) = \mathcal{O}(n \log n) + \mathcal{O}(n \log n) + \mathcal{O}(n)\mathcal{O}(1),$$

where the two $\mathcal{O}(n \log n)$ terms come from sorting and from finding the first job that starts after job i for all jobs, and $\mathcal{O}(n)\mathcal{O}(1)$ appears because there are $\mathcal{O}(n)$ subproblems, each of which takes constant time to solve. Simplifying the expression for $T(n)$ yields

$$T(n) = \mathcal{O}(n \log n).$$

CS787: Advanced Algorithms	
Scribe: Evan Driscoll and David Malec	**Lecturer:** Shuchi Chawla
Topic: Dynamic Programming II	**Date:** September 10, 2007

Today's lecture covered two more dynamic programming problems. The first is Sequence Alignment, which attempts to find the distance between two strings. The second is Shortest Path in a graph. We discussed three algorithms for this; one that solves the single-source shortest path, and two that solve the all-pairs shortest path.

3.1 Sequence Alignment

The Sequence Alignment problem is motivated in part by computational biology. One of the goals of scientists in this field is to determine an evolutionary tree of species by examining how close the DNA is between two potential evolutionary relatives. Doing so involves answering questions of the form "how hard would it be for a species with DNA 'AATCAGCTT' to mutate into a species with DNA 'ATCTGCCAT'?"

Formally stated, the Sequence Alignment problem is as follows:

Given: two sequences over some alphabet and costs for addition of a new element to a sequence, deletion of an element, or exchanging one element for another.

Goal: find the minimum cost to transform one sequence into the other.

(There is a disclaimer here. We assume that only one transformation is done at each position. In other words, we assume that 'A' doesn't mutate to 'C' before mutating again to 'T'. This can be assured either by explicit edict or by arranging the costs so that the transformation from 'A' to 'T' is cheaper than the above chain; this is a triangle inequality for the cost function.)

As an example, we will use the following sequences:

$$\begin{aligned} S &= \text{ATCAGCT} \\ T &= \text{TCTGCCA} \end{aligned}$$

Our goal is to find how to reduce this problem into a simpler version of itself.

The main insight into how to produce an algorithm comes when we notice that we can start with the first letter of S (in our example, 'A'). There are three things that we might do:

- We could remove 'A' from the sequence. In this case, we would need to transform 'TCAGCT' into 'TCTGCCA'.
- We could assume that the 'T' at the beginning of sequence T is in some sense "new", so we add a 'T'. We would need to transform 'ATCAGCT' into 'CTGCCA', then prepend the 'T'.
- We could just assume that the 'A' mutated *into* the 'T' that starts the second string, and exchange 'A' for 'T'. We would then need to transform 'TCAGCT' into 'CTGCCA'.

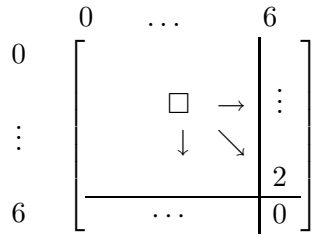
Once we have the above, it becomes a simple matter to produce a recurrence that describes the solution. Given two sequences S and T, we can define a function $D(i_s, j_s, i_t, j_t)$ that describes the distance between the subsequences $S[i_s \dots j_s]$ and $T[i_t \dots j_t]$.

$$D(i_s, j_s, i_t, j_t) = \min \begin{cases} D(i_s + 1, j_s, i_t, j_t) + c(\text{delete } S[i_s]) \\ D(i_s, j_s, i_t + 1, j_t) + c(\text{add } T[i_t]) \\ D(i_s + 1, j_s, i_t + 1, j_t) + c(\text{exchange } S[i_s] \text{ to } T[i_t]) \end{cases}$$

There are two ways to implement the above using dynamic programming. The first approach, top-down, creates a recursive function as above but adds memoization so that if a call is made with the same arguments as a previous invocation, the function merely looks up the previous result instead of recomputing it. The second approach, bottom-up, creates a four dimensional array corresponding to the values of D. This array is then populated in a particular traversal, looking up the recursive values in the table.

However, we can do better at the bottom-up approach than $O(n^4)$. Note that only the starting indices (i_s and i_t) are changed. In other words, we only ever compute $D(\dots)$ with $j_s = |S|$ and $j_t = |T|$. Thus we don't need to store those values in the table, and we only need a two-dimensional array with coordinates i_s, i_t. First, let $m = |S|$ and $n = |T|$ be the length of the strings.

For our example above, this is an 7x7 matrix:



Each element D_{ij} in this matrix denotes the minimum cost of transforming $S[i \dots n]$ to $T[j \dots m]$. Thus the final solution to the problem will be in D_{00}.

When the algorithm is computing the boxed element of the matrix, it need only look at the squares adjacent to it in each of the three directions marked with arrows, taking the minimum of each of the three as adjusted for the cost of the transformation.

The matrix can be traversed in any fashion that ensures that the three squares the box is dependent on are filled in before the boxed square is. (Keeping in mind whether the array is stored in column- or row-major format could help cache performance should the whole array not fit.)

Each square takes constant time to compute, and there are $|S| \cdot |T|$ squares, so the overall running time of the algorithm is $O(|S| \cdot |T|)$.

3.2 Shortest Path and the Bellman-Ford Algorithm

There are two forms of the shortest path algorithm that we will discuss here: single-source multi-target and all-pairs.

We will first look at the single-source problem. The algorithm we will develop to solve this is due to Bellman and Ford. The single-source multi-target shortest path problem is as follows:

Given: Weighted graph $G = (V, E)$ with cost function $c : E \to \mathbb{R}$, and a distinguished vertex $s \in V$.

Goal: Find the shortest path from a source node s to all other vertices.

G must not have cycles of negative weight (or a shortest path wouldn't exist, as it would traverse the cycle an infinite number of times), though it may have edges of negative weight. Section 3.5 discusses this matter further.

Aside: There is a greedy algorithm, Dijkstra's algorithm, that solves single-source shortest path. It expands outward from s, having computed the shortest path to each node within a boundary. However, Dijkstra's algorithm will only produce correct results for graphs that do not have edges with negative weight; our problem statement does not wish to make such an exclusion.

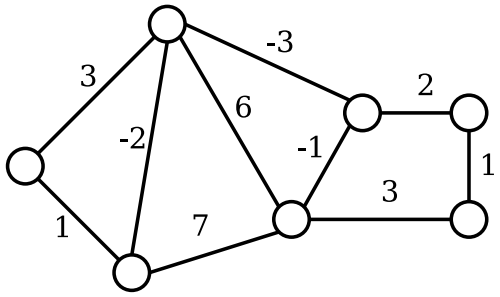
There are two possible approaches to solving this problem with dynamic programming:

- Create a smaller problem by restricting paths to a set number of edges
- Create a smaller problem by restricting the set of nodes that may be visited on intermediate paths

We will start with the former approach, which leads to a single-source solution as well as an all-pairs solution; the latter leads to a solution to the all-pairs problem, and will be discussed later.

The key observation for the first approach comes as a result of noticing that any shortest path in G will have at most $|V| - 1$ edges. Thus if we can answer the question "what's the shortest path from s to each other vertex t that uses at most $|V| - 1$ edges," we will have solved the problem. We can answer this question if we know the neighbors of t and the shortest path to each node that uses at most $|V| - 2$ edges, and so on.

For a concrete illustration, consider the following graph:



Computing the shortest path from s to t with at most 1 edge is easy: if there is an edge from s to

t, that's the shortest path; otherwise, there isn't one. (In the recurrence below, such weights are recorded as ∞.)

Computing the shortest path from s to t with at most 2 edges is not much harder. If u is a neighbor of s and t, there is a path from s to t with two edges; all we have to do is take the minimum sum. If we consider missing edges to have weight ∞ (which we are), then we needn't figure out exactly what us are neighbors of s and can just take the minimum over all nodes adjacent to t. (Or even over all of V.) We must also consider the possibility that the shortest path is still just of length one.

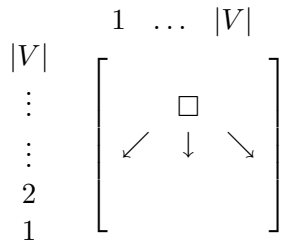
A similar process is repeated for 3 edges; we look at each neighbor u of t, add the weight of going from s to u with at most 2 hops to the weight of going from u to t, then taking the minimum such weight. Again, we must consider the possibility that the best path doesn't change when adding the 3rd edge.

Formally, we define $A(i,k)$ to be the length of the shortest path from the source node to node i that uses at most k edges, or ∞ if such a path does not exist. Then the following recurrence will compute the weights of the shortest paths, where $N(i)$ is the set of nodes adjacent to i and min returns ∞ if $N(i)$ is empty:

$$A(i,k) = \min \begin{cases} \min_{j \in N(i)} \left(A(j,k-1) + c(j,i) \right) \\ A(i,k-1) \end{cases}$$

The first option corresponds to the case where the path described by $A(i,k)$ actually involves all k edges; the second option is if allowing a kth edge doesn't change the solution. The second option can be removed if we consider an implicit, zero-weight loop at each node (that is, $c(v,v) = 0$ for all v).

The bottom-up dynamic programming approach to this problem uses an $|V| \times |V|$ matrix.



Rows represent the maximum number of edges that are allowed for a given instance; the rows are ordered so that the final answer (with a maximum of $|V|$ edges) is at the top of the graph. Columns represent vertices in the graph.

When computing $A(i,k)$, represented by the boxed square, the algorithm looks at all of the boxes in the next row down for nodes that are adjacent to i, as well as i itself. If the degree of the nodes in the graph are not bounded other than by $|V|$, then each square takes $O(|V|)$ time. Since there are $|V|^2$ squares, this immediately gives us an upper bound of $O(|V|^3)$ for the running time. However, we can do better.

Consider a traversal of one entire row in the above matrix. During this pass, each edge will be considered twice: once from node s to node t, and once from t to s. (To see this, note that the

diagonal arrows in the schematic above correspond to edges.) Thus the work done in each *row* is $O(|E|)$. There are $|V|$ rows, so this gives a total bound of $O(|V| \cdot |E|)$ for the algorithm.

This presentation of these shortest-path algorithms only explains how to determine the weight of the shortest path, not the path itself, but it is easy to adapt the algorithm to find the path as well. This is left as an exercise for the reader.

3.3 All Pairs Shortest Path

The all pairs shortest path problem constitutes a natural extension of the single source shortest path problem. Unsurprisingly, the only change required is that rather than fixing a particular start node and looking for shortest paths to all possible end nodes, we instead look for shortest paths between all possible pairs of a start node and end node. Formally, we may define the problem as follows:

Given: A graph $G = (V, E)$. A cost function $c : E \to \mathbb{R}$.

Goal: For every possible pair of nodes $s, t \in V$, find the shortest path from s to t.

Just as the problem itself can be phrased as a simple extension of the single source shortest paths problem, we may construct a solution to it with a simple extension to the Bellman-Ford algorithm we used for that problem. Specifically, we need to add another dimension (corresponding to the start node) to our table. This will change the definition of our recurrence to

$$A[i, j, k] = \min \begin{cases} A[i, j, k-1] \\ \min\limits_{l}\{A[i, l, k-1] + c(l, j)\} \end{cases} .$$

If we take $c(l, l) = 0$ for all $l \in V$, we may write this even more simply as

$$A[i, j, k] = \min_{l}\{A[i, l, k-1] + c(l, j)\}.$$

It is easy to see how this modification will impact the running time of our algorithm — since we have added a dimension of size $|V|$ to our table, our running time will increase from being $O(|V| \cdot |E|)$ to being $O(|V|^2 \cdot |E|)$.

3.4 Time Bound Improvement

If we want to gain some intuition as to how we might go about improving the time bound of our algorithm when applied to dense graphs, it is helpful to alter how we think of the process. Rather than considering a 3-dimensional table A, we will consider a sequence of 2-dimensional tables A_k. Our original table had dimensions for the start node, the end node, and the number of hops; we will use the number of hops to index the sequence A_k, leaving each table in the sequence with

dimensions for the start node and the end node. We may then produce these tables sequentially as

$$\begin{aligned} A_1(i,j) &= c(i,j) \\ A_2(i,j) &= \min_l\{A_1(i,l) + c(l,j)\} \\ A_3(i,j) &= \min_l\{A_2(i,l) + c(l,j)\} \\ &\vdots \end{aligned}$$

If we consider how we define A_1 in the above sequence, we can see that we may write general A_k as

$$A_k(i,j) = \min_l\{A_{k-1}(i,l) + A_1(l,j)\}.$$

Of special interest to us is the close resemblance this definition holds to matrix multiplication, when we write that operation in the form $\sum_l[A_{k-1}(i,l) \times A_1(l,j)]$. If we define an operator on matrices $(\cdot)$ which has the same form as matrix multiplication, but substituting the minimum function for summation and addition for the cross product, then we may rewrite our sequence of matrices as

$$\begin{aligned} A_1(i,j) &= c(i,j) \\ A_2 &= A_1 \cdot A_1 \\ A_3 &= A_2 \cdot A_1 \\ A_4 &= A_3 \cdot A_1 \\ &\vdots \end{aligned}$$

While the change in the underlying operations means that we can not employ efficient matrix multiplication algorithms to speed up our own algorithm, if we look at the definition of A_4 we can see that the similarities to multiplication will still provide an opportunity for improvement. Specifically, expanding out the definition of A_4 gives us

$$A_4 = A_3 \cdot A_1 = A_2 \cdot A_1 \cdot A_1 = A_2 \cdot A_2,$$

which demonstrates how we may speed up our algorithm by mimicking the process which allows exponentiation by a power of 2 to be sped up by replacing it with repeated squaring.

Phrased in terms of our original statement of the problem, where before we would have split a path of length k into a path of length $k-1$ and a path of length 1, we now split such a path into two paths of length $k/2$. Essentially, we are leveraging the fact that while accommodating any possible start for our path increases the complexity of our algorithm, it also gives us more information to work with at each step. Previously, at step k, only paths of length less than k starting at s and paths of length 1 were inexpensive to check; now, at step k, we have that any path of length less than k is inexpensive to check. So, rather than splitting our new paths one step away from the destination, we may split them halfway between the start and the destination. Our recurrence relation becomes

$$A[i,j,k] = \min_l\{A[i,l,k/2] + A[l,j,k/2]\}, \text{ where } A[i,j,1] = c(i,j).$$

The running time for this version of the algorithm may be seen to be $O(|V|^3 \log |V|)$. In each iteration, in order to compute each of the $|V|^2$ elements in our array, we need to check each possible midpoint, which leads to each iteration taking $O(|V|^3)$ time. Since we double the value of k at each step, we need only $\lceil \log |V| \rceil$ iterations to reach our final array. Thus, it is clear that the overall running time must be $O(|V|^3 \log |V|)$, a definite improvement over our previous running time of $O(|V|^2 \cdot |E|)$ when run upon a dense graph.

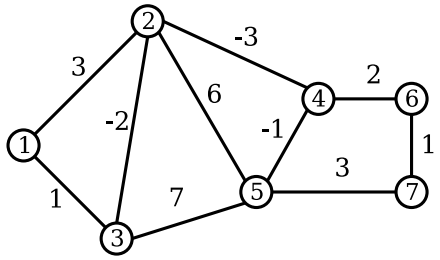
3.5 Negative Cycles

Throughout the preceding discussion on how to solve the shortest path problem — whether the single source variant or the all pairs variant — we have assumed that the graph we are given will not include any negative cost cycles. This assumption is critical to the problem itself, since the notion of a shortest path becomes ill-defined when a negative cost cycle is introduced into the graph. After all, once we have a negative cost cycle in the graph, we may better any proposed minimum cost for traveling between a given start node and end node simply by including enough trips around the negative cost cycle in the path we choose. While the shortest path problem is only well-defined in the absence of negative cost cycles, it would still be desirable for our algorithm to remain robust in the face of such issues.

In fact, the Bellman-Ford algorithm does behave predictably when given a graph containing one or more negative cycles as input. Once we have run the algorithm for long enough to determine the the best cost for all paths of length less than or equal $|V| - 1$, running it for one more iteration will tell us whether or not the graph has a negative cost cycle: our table will change if and only if a negative cost cycle is present in our input graph. In fact, we only need to look at the costs for a particular source to determine whether or not we have a negative cycle, and hence may make use of this test in the single source version of the algorithm as well. If we wish to find such a cycle after determining that one exists, we need only consider any pair of a start node and an end node which saw a decrease in minimum cost when paths of length $|V|$ were allowed. If we look at the first duplicate appearance of a node, the portion of the path from its first occurrence to its second occurrence will constitute a negative cost cycle.

3.6 Floyd-Warshall

A second way of decomposing a shortest path problem into subproblems is to reduce along the intermediate nodes used in the path. In order to do so, we must first augment the graph we are given by labeling its nodes from 1 to n; for example, we might change the graph we used as an example in section 3.2 to



When we reduce the shortest paths problem in this fashion, we end up with the table

$$A[i,j,k] = \text{ shortest path from } i \text{ to } j \text{ using only intermediate nodes from } 1,\ldots,k,$$

which is governed by the recurrence relation

$$A[i,j,k] = \min \left\{ \begin{array}{l} A[i,j,k-1] \\ A[i,k,k-1] + A[k,j,k-1] \end{array} \right. , \text{ where } A[i,j,0] = c(i,j).$$

It is reasonably straightforward to see that this will indeed yield a correct solution to the all pairs shortest path problem. Assuming once again that our graph is free of negative cost cycles, we can see that a minimum path using the first k nodes will use node k either 0 or 1 times; these cases correspond directly to the elements we take the minimum over in our definition of $A[i,j,k]$. An inductive proof of the correctness of the Floyd-Warshall algorithm follows directly from this observation.

If we wish to know the running time of this algorithm, we may once again consider the size of the table used, and the time required to compute each entry in it. In this case, we need a table that is of size $|V|^3$, and each entry in the table can be computed in $O(1)$ time; hence, we can see that the Floyd-Warshall algorithm will require $O(|V|^3)$ time in total.

Given that this improves upon the bound we found for the running time of the all pairs variant of the Bellman-Ford algorithm in (3.4), a natural question is why we might still wish to use the Bellman-Ford algorithm. First, within the context of the single source version of the problem, the Bellman-Ford algorithm actually provides the more attractive time bound, since the Floyd-Warshall algorithm doesn't scale down to this problem nicely. Second, considered within the context of the all pairs version of the problem, the main advantage it offers us is the way it handles negative cycles. While the Bellman-Ford algorithm allows us to test for the presence of negative cycles simply by continuing for one additional iteration, the Floyd-Warshall algorithm provides no such mechanism. Thus, unless we can guarantee that our graph will be free of negative cycles, we may actually prefer to use the Bellman-Ford algorithm, despite its less attractive time bound.

CS787: Advanced Algorithms

Scribe: Blayne Field, Dan Rosendorf
Lecturer: Shuchi Chawla
Topic: Network Flows
Date: Sept. 12 2007

In this lecture, we discussed the problem of maximum networks flows, which is a general problem to which many other problems can be reduced.

4.1 Problem Definition

Given: A graph $G = (V, E)$, which can be either directed or undirected, a capacity function $c : E \mapsto \Re^+$, along with a source s and a sink t. This can be thought of as a set of pipes, with the capacities being analogous to the cross-section of the pipe. The water is inserted into the network at the source s and ends up at the sink t. Our goal will be to find out what the maximum throughput of our network can be. Where by throughput in our analogy we mean the maximum amount of water per unit of time that can flow from s to t.

Let us now explain more precisely what we mean by our notation. In our problem we obviously need not consider loops in the graph (i.e. edges from v to v) since they add nothing to the throughput abilities of the system and we can also WLOG assume there are no multiple edges since if e_1, e_2 are two edges from u to v with capacities c_1, c_2 then we can represent them by one edge e with capacity $c_1 + c_2$.

This means that if we have a directed graph we can consider it to be in the form of some set V and a set E such that $E \subset V \times V$. For an undirected graph we shall consider E to be a subset of $[V]^2$ (the two element subsets of V) by abuse of notation we shall still write (u, v) for an edge instead of the more correct $\{u, v\}$. Very soon we will see that we can actually restrict ourselves to only the case of the directed graph but first let us give a definition.

Definition 4.1.1 *Let $G = (V, E)$ be a directed or undirected graph, c be a capacity function on G and $f : V \times V \mapsto R^+$. We say that f is a flow if the following holds:*

$$\forall v \in V (v \neq s \wedge v \neq t) \rightarrow (\sum_{u \in V} f(u, v) = \sum_{u \in V} f(v, u))$$

In words for all nodes that are not the sink or source the amount of incoming flow is equal to the amount of outgoing flow, (i.e. there are no points of acummulation of flow) this is usually called the conservation property.

We say that a flow is feasible for G (or just feasible if the graph and capacity function in question are obvious) if

$$\forall u, v \in V \ ((u, v) \in E) \rightarrow (f(u, v) \leq c((u, v))) \wedge ((u, v) \notin E) \rightarrow (f(u, v) = 0)$$

In words the flow along any valid edge is smaller then that edge's capacity.

We define the size of a flow f denoted $|f|$ to be $|f| = \sum_{u \in V} f(s, u) - \sum_{u \in V} f(u, s)$ or all outgoing flow from s.

Goal: The goal is to find a feasible flow with maximum size.

Let us now notice a few things so we can from now on only consider directed graphs. Suppose we have an undirected graph $G = (V, E)$ with a capacity function c. We then define G' to be the directed graph with vertices V and edges $E' = \{(u, v); \{u, v\} \in E\}$. In other words for each edge in our original graph we put in two edges going each way. We define $c' : E' \mapsto \mathbb{R}^+$ by $c'((u, v)) = c(\{u, v\})$, which just means we set each of our two edges we created to have the same capacity as the one original edge.

Now we claim that the maximum flow in our new oriented graph has the same size as the maximum flow in our original graph and that it is easy to get the flow in the original graph from the flow in the new graph.

Lemma 4.1.2 *Let $G = (V, E)$ be an undirected graph and define G' as above. Let f_1 be a feasible flow in G and f_2 be a feasible flow in G' then there exist feasible flows f_1' in G' and f_2' in G such that $|f_1| = |f_1'|$ and $|f_2| = |f_2'|$.*

Proof: For f_1 define f_1' to just be f_1. Then obviously the conservation properties still hold and since we made both directions of our new edge have the capacity of the original edge f_1 is still feasible. Since we have not changed anything $|f_1| = |f_1'|$.

On the other hand for f_2 define $f_2'(u, v) = max\{0, f_2(u, v) - f_2(v, u)\}$ then th conservation property is still satisfied as one easilly check. Feasibility is satisfied thanks to the fact that f_2 is nonnegative and thus $|f_2(u, v) - f_2(v, u)|$ is at most $c((u, v))$. The equality of size can be checked in a straightforward manner. ■

Definition 4.1.3 *An s, t cut is a set of edges C such that the graph $G' = (V, E \setminus C)$ contains no s, t path. The capacity of the cut is $cap(C) = \sum_{e \in C} c(e)$*

Note that an s, t cut defines a partition on the set of edges V into $(S, \bar{S})$, such that $s \in S$ and $t \in \bar{S}$

Lemma 4.1.4 *If we take F to be the set of all feasible flows, then for any s, t cut C, $cap(C) \geq max_{f \in F}|f|$.*

Proof: Since C is a cut, any s, t path must contain at least one edge in C, thus any flow must have value less than $cap(C)$.

Corollary 4.1.5 *Let C^* be a minimum s, t cut of G and f^* be a max flow of G, then $cap(C^*) \geq |f^*|$.*

Using this corollary, we see that if we can exhibit a min-cut that equals the value of our flow, it serves as a certificate of the optimality of our flow. This is illustrated in our earlier example by the the cut below, which has a capacity of 6, the same as the value of our flow (see Figure 4.2.2).

4.2 Ford-Fulkerson Algorithm

Since greedy approaches often lead to an optimal solution, it seems natural to start trying to solve this problem with such an approach. We know that if we can find an s, t path with some positive capacity, we can increase the flow along that edge while maintaining feasability. This leads us to the following greedy approach:

- Find an s, t path with some remaining (positive) capacity.
- Saturate the path
- Repeat until no such paths exist

Unfortunately, this approach does not give us an optimal flow. As illustrated below in Figure 4.2.1, where the numbers in parantheses are the capacities and the other ones are the size of the flow) we chose the path through the middle and saturated it with 3 units of flow. This is a reasonable path to choose, since it has the maximum capacity of all s, t paths. However, this leaves us with one remaining s, t path which has a capacity of 1. Saturating this path will result in the termination of the algorithm, giving a flow of value 4. However, as we saw before, the maximum value of this network is 6. Thus we need to improve on this approach. We do this by using the "residual graph" of G, in what is known as the Ford-Fulkerson algorithm.

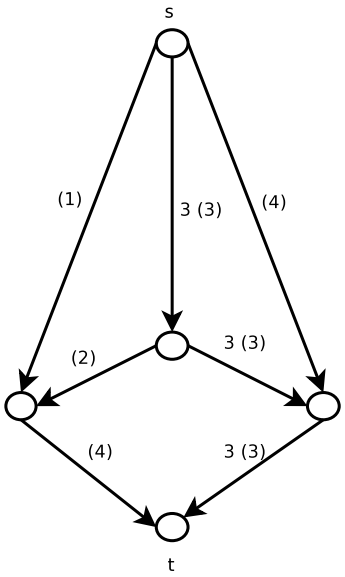


Figure 4.2.1: 3-3-3 flow

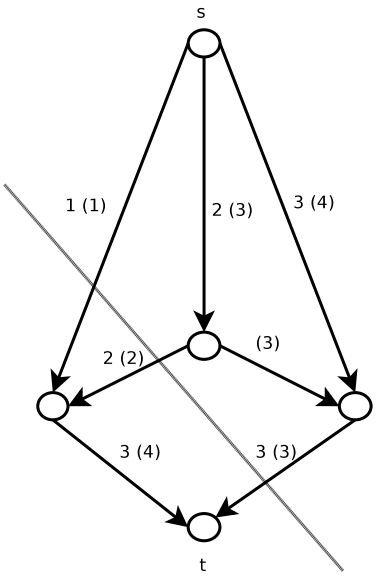


Figure 4.2.2: Cut with flow

4.2.1 Ford-Fulkerson Algorithm

The general idea of the Ford-Fulkerson algorithm is to choose an $s - t$ path through our graph saturate it, and create a new residual graph which has possibly different edges representing the flow and thus allowing us to sort of go back on our original flow if it is not maximal. We iterate this process until there is no $s - t$ path in our residual graph.

First let us define the residual graph for a graph with a given flow.

Definition 4.2.1 *Let $G = (V, E)$ be an directed graph with a capacity function c. Let f be a feasible flow in this graph. Then we define the residual graph, $G_f = (V_f, E_f)$ and c_f in the following manner.*

$V_f = V$ *and* $E_f = \{(u,v)|u,v \in V \wedge (c(u,v) > f(u,v))\} \cup \{(u,v)|u,v \in V \wedge (f(v,u) > 0)\}$. *We define* $c_f(u,v) = c(u,v) - f(u,v) + f(v,u)$.

This means that for every edge in our original graph if it is saturated we remove it and we either add a new edge in the oposite direction with capacity equal to that of the flow or if there exists an edge in that direction we increase it's capacity by the amount of the flow. For a non saturated edge we decrease it's capacity by the amount of the flow and add the capacity of the flow to the edge going in the other direction. Figures 4.2.3 and 4.2.4 show a graph with a flow and it's corresponding residual graph.

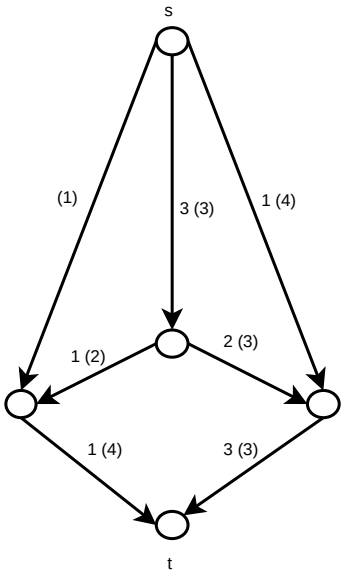


Figure 4.2.3: A graph with a feasible flow

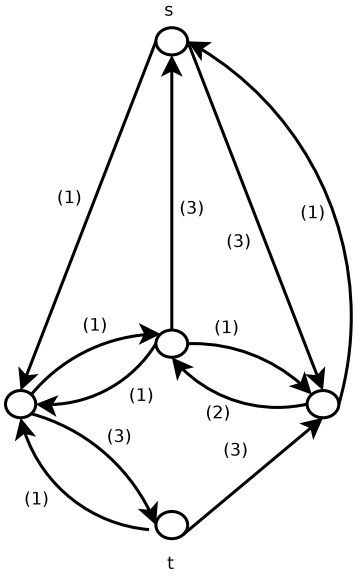


Figure 4.2.4: Residual graph of the flow in the previous figure

We also define the addition of two flows in the obvious fashion.

Definition 4.2.2 *Let* $f : V \times V \mapsto \mathbb{R}^+$ *and* $f' : V \times V \mapsto \mathbb{R}^+$ *then we define* $g = f + f'$ *in the following manner.* $g(u,v) = max\{0, f(u,v) - f(v,u) + f'(u,v) - f'(v,u)\}$. *It is obvious that the conservation property holds for* g.

Let us now proceed to the **Ford-Fulkerson** algorithm.

1. Start with $f : V \times V \mapsto \{0\}$ and $G = (V,E)$ an directed graph and a capacity function c.
2. Create G_f and c_f.
3. Find a simple $s-t$ path P in G_f and let c' be the minimum capacity along this path define let f' be the flow of capacity c' along P and 0 everywhere else. Let $f = f + f'$.
4. if no s,t path is found terminate.
5. Repeat

The paths chosen in step 3 are called "augmenting paths".

Theorem 4.2.3 *The Ford-Fulkerson algorithm described above finds the maximum flow if the capacities are integers.*

Proof: First note that at each step of the algorithm $|f|$ increases by at least a unit thus the algorithm must terminate in a finite number of steps, since we have only a finite amount of capacity leaving the source.

Next we note that the flow created at every step is actually a feasible flow in the graph G. We show this by induction. It is obvious for the empty flow. Let f_i is our flow at stage i which is feasible by hypothesis and suppose P is our s,t path and $c_{f_i,P}$ is the minimum capacity along that path. Suppose (u,v) is any edge on the path P (obviously for edges outside the path feasibility is no problem).

We have $c_{f_i}(u,v) = c(u,v) - f_i(u,v) + f_i(v,u)$. By the definition of the addition of flows we can see that any flow that is a sum of two flows has the property that if $f(u,v) > 0$ then $f(v,u) = 0$. If $f_i(u,v) > 0$ then we have $0 < c_{f_i}(u,v) = c(u,v) - f_i(u,v)$ from feasibility of f_i and since $c_{f_i,P} \leq c_{f_i}(u,v)$ we get by simple algebra $f_{i+1} \leq c(u,v)$. If we have $f_i(v,u) > 0$ then $0 < c_{f_i}(u,v) = c(u,v) + f_i(v,u)$ and $f_{i+1}(u,v)$ is by definition either 0, or $f_{i+1}(u,v) = c_{f_i,P} - f_i(v,u)$ which then by simple algebra yields $f_{i+1}(u,v) \leq c(u,v)$.

To show that our resulting flow is a maximal flow first notice that $S = \{v; v \in V \wedge (\text{ exists an } s,v \text{ path in } G_f)\}$ along with it's complement gives us an s,t cut C in G using the second definition. Otherwise we would have an s,t path in G_f which is a contradiction with the fact that the algorithm terminated. Next we wish to claim that the capacity of this cut is exactly equal to the size of our resulting flow f_r.

By the definition of S we must have for any $(u,v) \in C$, $f_r(u,v) = c(u,v)$ since otherwise $c_{f_r}(u,v) > 0$ and thus we can get from s to v via some path. We also have for any $(u,v) \in E$ where $v \in S$ and $u \notin S$ that $f_r(u,v) = 0$ since otherwise $c_{f_r}(v,u) = c(v,u) + f_r(u,v) > 0$ (since we know that only one of $f_r(u,v)$, $f_r(v,u)$ is non zero). Now this means that we have that the total flow exiting S is the capacity of C. Now

$$\sum_{u \in S}(\sum_{v \in V} f_r(u,v) - \sum_{v \in V} f_r(v,u)) = \sum_{v \in V} f_r(s,v)$$

this follows from the fact that conservation holds for all $u \in S$ other then s, but since $f_r(v,u) = 0$ for $u \in S$ and $v \notin S$ the second sum degenerates into

$$\sum_{v \in S} f_r(v,u)$$

giving us

$$\sum_{u \in S}(\sum_{v \in V} f_r(u,v) - \sum_{v \in S} f_r(v,u))$$

and by regrouping we get $\sum_{u \in S}(\sum_{v \notin S} f_r(u,v))$ which is exactly the total flow out of S. So we see that $|f|$ is actually equal to the capacity of C. ■

Corollary 4.2.4 Max-flow Min-cut theorem.
The size of the maximum flow is exactly equal to the capacity of the minimum cut.

Before we move on let us notice that even though we only proved that the Ford-Fulkerson algorithm works with integer capacities, it is easy to see that if we have rational capacities we can just change them by multiplying all of them by their smallest common denominator, and a maximum flow in such a graph will easily yield a maximum flow in our original graph.

Now let us turn to examining the running time of our algorithm. Every step in an iteration takes at most $O(m)$ time where m is the number of edges. An easy bound on the number of iterations could be for example $F = |f_r|$ the size of the maximal flow since in each iteration we increase the size of our flow by at least one. This gives us a bound of $O(mF)$. Now if we set $C = max_{e \in E} c(e)$ we get a bound of $O(m^2 C)$ or better yet if Δ_s is the number of edges exiting s we get $O(m \Delta_s c)$. This runing time is still only pseudo polynomial in our input though since we input the capacities as binary numbers and as such the size of the input is actually $\log_2(c)$.

The following example illustrates that if we get an unfortunate graph and choose our s, t paths in a bad fashion this worst case time could actually be reached. If we take the graph in Figure 4.2.5 and choose our paths to always include the middle edge of size 1 we actually have to iterate 2^{101} times.

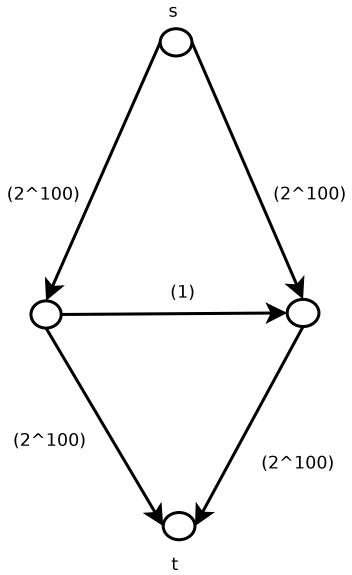


Figure 4.2.5: Bad runtime graph

To handle this situation we can modify our algorithm in the following way. Let us define G_f^{Δ} to be the graph to be G_f without edges of capacity less then Δ. Then start by setting Δ to be half the maximum capacity in the graph. Let the Ford-Fulkerson algorithm run on G_f^{Δ} and after it's termination let $\Delta = \Delta/2$ and repeat until $\Delta = 1$. We can see that in each stage we will make at most $2m$ repetitions since each iteration increases flow by at least Δ and we have the total outgoing

capacity from s at most degree of s times $\Delta * 2$ since all capacities in our graph are at most $\Delta * 2$. There will be at most $\log_2(c) + 1$ repetitions where c is the maximum capacity in our graph and we get a bound of $O(m^2 \log(c))$.

It actually turns out that if we run the Ford-Fulkerson algorithm making sure to choose the least path in every iteration we can get a bound of $O(m^2 n)$, but we shall not show that here.

CS787: Advanced Algorithms

Scribe: Daniel Wong and Priyananda Shenoy | **Lecturer:** Shuchi Chawla
Topic: Applications of Network Flow | **Date:** 9/14/2007

5.1 Introduction and Recap

In the last lecture, we analyzed the problem of finding the maximum flow in a graph, and how it can be efficiently solved using the Ford-Fulkerson algorithm. We also came across the Min Cut-Max Flow Theorem which relates the size of the maximum flow to the size of the minimal cut in the graph.

In this lecture, we will be seeing how various different problems can be solved by reducing the problem to an instance of the network flow problem. The first application we will be seeing is the Bipartite Matching problem.

5.2 Bipartite Matching

A Bipartite Graph $G(V, E)$ is a graph in which the vertex set V can be divided into two disjoint subsets X and Y such that every edge $e \epsilon E$ has one end in X and the other end in Y. A matching M is a subset of edges such that each node in V appears in at most one edge in M. A Maximum Matching is a matching of the largest size.

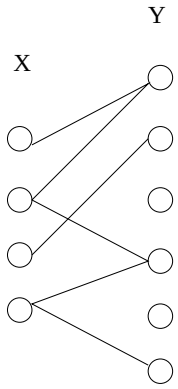


Fig 1 : Bipartite Graph

A clarification about the terminology:

Definition 5.2.1 (Maximal Matching) *A maximal matching is a matching to which no more edges can be added without increasing the degree of one of the nodes to two; it is a local maximum.*

Definition 5.2.2 (Maximum Matching) *A maximum matching is one which is the largest possible; it is globally optimal.*

It can be shown that for any maximal matching $M_i, |M_i| \geq \frac{1}{2}|M|$ where M is the maximum matching. Given this, it is straightforward to obtain a 2-approximation to the maximum matching.

The problem of finding the maximum matching can be reduced to maximum flow in the following manner. Let $G(V, E)$ be the bipartite graph where V is divided into X, Y. We will construct a directed graph $G'(V', E')$, in which V' which contains all the nodes of V along with a source node s and a sink node t. For every edge in E, we add a directed edge in E' from X to Y. Finally we add a directed edge from s to all nodes in X and from all nodes of Y to t. Each edge is given unit weight.

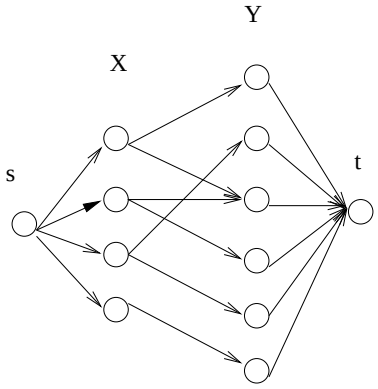


Fig 2: The Bipartite Graph converted into a flow graph

Let f be the maximum integral flow of G', of value k. Then we can make the following observations:

1. There is no node in X which has more than one outgoing edge where there is a flow.
2. There is no node in Y which has more than one incoming edge where there is a flow.
3. The number of edges between X and Y which carry flow is k.

By these observations, it is straightforward to conclude that the set of edges carrying flow in f forms the maximum matching for the graph G.

The running time for the Ford-Fulkerson algorithm is $O(mF)$ where m is the number of edges in E and $F = \sum_{e \in \delta(s)}(c_e)$. In case of bipartite matching problem, $F \leq |V|$ since there can be only $|V|$ possible edges coming out from source node. So the running time is $O(mn)$ where n is the number of nodes.

An interesting thing to note is that any $s - t$ path in the residual graph of the problem will have alternating matched and unmatched edges. Such paths are called **alternating paths**. This property can be used to find matchings even in general graphs.

5.2.1 Perfect Matching

A perfect matching is a matching in which each node has exactly one edge incident on it. One possible way of finding out if a given bipartite graph has a perfect matching is to use the above

algorithm to find the maximum matching and checking if the size of the matching equals the number of nodes in each partition. There is another way of determining this, using Hall's Theorem.

Theorem 5.2.3 *A Bipartite graph G(V,E) has a Perfect Matching iff for every subset $S \subseteq X or S \subseteq Y$, the size of the neighbors of S is at least as large as S, i.e $|\Gamma(S)| \geq |S|$*

We will not discuss the proof of this theorem in the class.

5.3 Scheduling Problems

We will now see how the Max Flow algorithm can be used to solve certain kinds of scheduling problems. The first example we will take will be of scheduling jobs on a machine.

Let $J = \{J_1, J_2, ..., J_n\}$ be the set of jobs, and $T = \{T_1, T_2, .., T_k\}$ be slots available on a machine where these jobs can be performed. Each job J has a set of valid slots $S_j \subseteq T$ when it can be scheduled. The constraint is that no two jobs can be scheduled at the same time. The problem is to find the largest set of jobs which can be scheduled.

This problem can be solved by reducing it to a bipartite matching problem. For every job, create a node in X, and for every timeslot create a node in Y. For every timeslot T in S_j, create an edge between J and T. The maximum matching of this bipartite graph is the maximum set of jobs that can be scheduled.

We can also solve scheduling problems with more constraints by having intermediate nodes in the graph. Let us consider a more complex problem: There are k vacation periods each spanning multiple contiguous days. Let D_j be the set of days included in the j^{th} vacation period. There are n doctors in the hospital, each with a set of vacation days when he or she is available. We need to maximize the assignment of doctors to days under the following constraints:

1. Each doctor has a capacity c_i which is the maximum total number of days he or she can be scheduled.
2. For every vacation period, any given doctor is scheduled at most once.

We will solve this problem by network flow. As was done earlier, for every doctor i we create a node u_i and for every vacation day j we create a node v_j. We add a directed edge with unit capacity from start node s to u_i and from v_j to sink t. To include the constraints, the way the graph is constructed in the following way:

- The doctor capacities are modeled as capacities of the edge from the source to the vertex corresponding to the doctor.
- To prevent the doctor to be scheduled more than once in a vacation period, we introduce intermediate nodes. For any doctor i and a vacation period j, we create an intermediate node w_{ij}. We create an edge with unit capacity from u_i to w_{ij}. For every day in the vacation period that the doctor is available, we create an edge from w_{ij} to the node corresponding to that day with unit capacity.

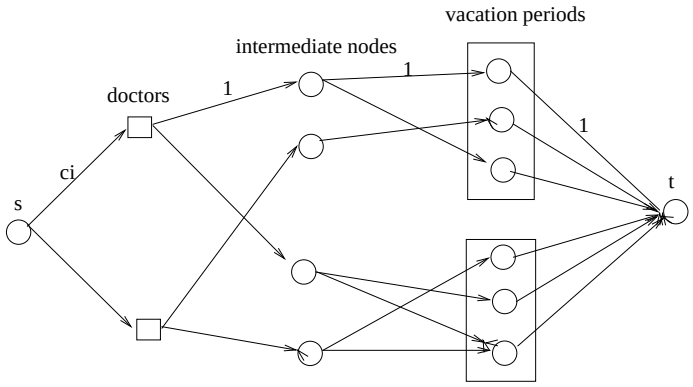


Fig 3: Scheduling Problem with constraints

Let us see if the an integral flow through the graph produces a valid schedule. Since the edge connecting s to the node corresponding to the doctor has the capacity equal to the total availability of the doctor, the flow through the doctor node cannot exceed it. So the first constraint is satisfied. From any doctor to any vacation period the flow is atmost one, since the intermediate node has only one incoming edge of unit capacity. This makes sure that the second criteria is met. So the flow produced satisfies all the constraints.

If k is the size of an integral flow through the graph, then the total number of vacation days which have been assigned is also k, since the edges which connect the nodes corresponding to the vacation days to the sink node have unit capacity each. So the size of the scheduling is equal to the size of the flow. From this we can conclude that the largest valid scheduling is produced by the maximum flow.

5.4 Partitioning Problems

Now we will see how certain kinds of paritioning problems can be solved using the network flow algorithm. The general idea is to reduce it to min-cut rather than max-flow problem. The first example we will see is the Image segmentation problem.

5.4.1 Image Segmentation

Consider the following simplification of the image segmentation problem. Assume every pixel in an image belongs to either the foreground of an image or the background. Using image analysis techniques based on the values of the pixels its possible to assign probabilites that an individual pixel belongs to the foreground or the background. These probabilites can then be translated to "costs" for assigning a pixel to either foreground or background. In addition, there are costs for segmenting pixels such that pixels from differant regions are adjacent. The goal then is to find an assignment of all pixels such that the costs are minimized.

Let f_i be the cost of assigning pixel i to the foreground.
Let b_i be the cost of assigning pixel i to the background.

Let s_{ij} be the cost of separating neighboring pixels i and j into different regions.

Problem: Find a partition of pixels into, $p \in S$, the set of foreground pixels and $\bar{S}$, the set of background pixels,such that the global cost of this assignment is minimized.

Let $(S, \bar{S})$ be the partition of pixels into foreground and background respectively. Then the cost of this partition is defined as:

$$cost(S) = \sum_{i \in S} b_i + \sum_{i \notin S} f_i + \sum_{\substack{i \in S, j \notin S \\ i \text{ and } j \text{ are neighbours}}} s_{ij} \tag{5.4.1}$$

We need to find an S which minimizes *cost*. This problem can be reduced to a min-cut max-flow problem. The natural reduction is to a min-cut problem.

Let each pixel be a node, with neighboring pixels connected to each other by undirected edges with capacity s_{ij}. Create additional source and sink nodes, s and t, respectively which have edges to every pixel. The edges from s to each pixel have capacity b_i. The edges from each pixel to t have capacity f_i.

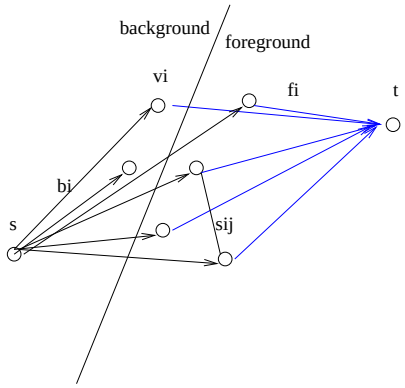


Fig 3: Image segmentation flow graph

Any cut in the graph will then naturally separate the pixels into two groups. With the arrangement the capacity of the cut is the cost associated with that partition. The min-cut max-flow algorithm will find such a partition of minimum cost.

5.4.2 Image Segmentation cont.

Consider a modification of the original problem. Replace f_i and b_i with values representing the "benefit/value" for assigning the pixel i to either the foreground or background.

Problem: Find an assigment such that for all pixels, $p \in S$, the set of foreground pixels, or $\bar{S}$, the set of background pixels,such that the global value is maximized. Where value is:

$$val(S) = \sum_{i \in S} f_i + \sum_{i \notin S} b_i - \sum_{\substack{i \in S, j \notin S \\ i \text{ and } j \text{ are neighbours}}} s_{ij} \tag{5.4.2}$$

Since the min-cut approach is a minimization problem, we need to convert it into a minimization problem in order to reduce it to min-cost. Let $(S, \bar{S})$ be the partition of pixels into foreground and background respectively.

We can relate *cost* (defined as per 5.4.1) and *val* by assuming cost to mean "lost benefit":

$$cost(S) = \sum_{i} (b_i + f_i) - val(S) \tag{5.4.3}$$

Since $\sum_i (b_i + f_i)$ is a constant for any given instance, the problem of maximising val reduces to minimizing the cost.

We can reformulate cost as

$$cost(S) = \sum_{i \in S} b_i + \sum_{i \notin S} f_i + \sum_{\substack{i \in S, j \notin S \\ i \text{ and } j \text{ are neighbours}}} s_{ij} \tag{5.4.4}$$

Comparing this to the previous problem, we see that this is the same as interchanging b_i and f_i in the graph we used to solve the first problem; i.e. the weight of the nodes from s to v_i is set as f_i and weight of v_i to t is set as b_i. Solving this using min-cut will give us the partition which maximizes the benefit.

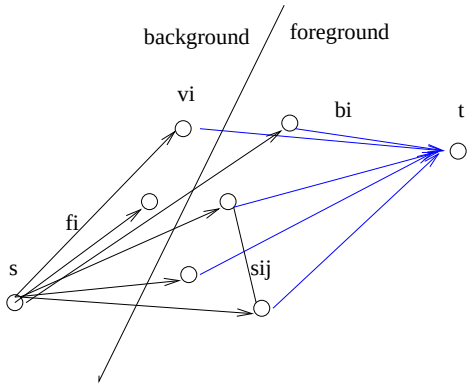


Fig 4: Flow graph for maximizing benefits

5.5 Max-Flow with Costs

Let us consider an extension of the perfect matching problem where different edges have different costs. The naturally leads to the min-cost max-flow problem. In this case, not only do we have to find a max flow through the graph, but also have to find one with the least cost.

Likewise, in the min-cost perfect matching problem, the cost of a perfect matching M is $\sum_{e \epsilon M} c(e)f(e)$ where $f(e)$ is the cost of including the edge e in the matching.

It can be shown that by directing the flow through the minimum cost augementing path first,we will find the maximum flow with the least cost.

References

[1] Eva Tardos, Jon Kleinberg Algorithm Design. *Pearson Education*, 2006.

CS787: Advanced Algorithms

Scribe: Yudong Tang and Baris Aydinlioglu **Lecturer:** Shuchi Chawla
Topic: Randomized Algorithms **Date:** September 17, 2007

In this lecture and the next we cover randomized algorithms. Today we begin by motivating the use of randomized algorithms. Next we recall some related definitions and results from probability theory. We end today's lecture with studying Karger's randomized algorithm for computing a min-cut of a graph.

6.1 Introduction

We define a *randomized algorithm* as an algorithm that can toss coins and act depending on the outcome of those tosses. There are two kinds of randomized algorithms that we will be studying:

- Las Vegas Algorithms: These refer to the randomized algorithms that always come up with a/the correct answer. Their "expected" running time is polynomial in the size of their input, which means that the average running time over all possible coin tosses is polynomial. In the worst case, however, a Las Vegas algorithm may take exponentially long. One example of a Las Vegas algorithm is Quicksort: it makes some of its decisions based on coin-tosses, it always produces the correct result, and its expected running and worst-case running times are $n \log n$ and n^2, respectively.
- Monte Carlo Algorithms: These refer to the randomized algorithms that sometimes come up with an incorrect answer. Like Las Vegas algorithms their expected running time is polynomial but may be exponential.

  By performing independent runs of a Monte Carlo algorithm, we can decrease the chances of obtaining an incorrect result to a value as small as we like. For example, if a particular Monte Carlo algorithm produces an incorrect result with probability 1/4, and we have the ability to detect an incorrect result, then the probability of not obtaining a correct answer after t independent runs is $(1/4)^t$. In some contexts, such as in optimization problems, we do not have the ability to detect an incorrect answer, but can compare different answers and pick the best one. The same analysis would also apply in these cases, because if picking the best answer does not give the optimal answer, then it means the algorithm produced an incorrect answer in all of its independent runs.

  Sometimes we may not have an algorithm with an error probability of a small constant, but we may have one with error probability a function of the input size. In such a case, we aim for an error probability from which we can obtain desired low values with a small number (i.e., polynomial in the size of the input) of independent runs.

6.2 Motivation

Randomized algorithms have several advantages over deterministic ones. We discuss them here:

- Randomized algorithms tend to bring **simplicity**. For example, recall from Lecture 2 the problem of finding the kth smallest element in an unordered list, which had a rather involved deterministic algorithm. In contrast, the strategy of picking a random element to partition the problem into subproblems and recursing on one of the partitions is much simpler.

- Another advantage randomized algorithms can sometimes provide is **runtime efficiency**. Indeed, there are problems for which the best known deterministic algorithms run in exponential time. A current example is Polynomial Identity Testing, where given a n-variate polynomial f of degree d over a field F, the goal is to determine whether it is identically zero. If f were univariate then merely evaluating f at $d+1$ points in F would give us a correct algorithm, as f could not have more that d roots in F. With n variables, however, there may be n^d roots, and just picking n^d+1 points in F gives us exponential time. While we don't know of a deterministic polytime algorithm for this problem, there is a simple algorithm that evaluates f at only $2d$ points, based on a theorem by Zippel and Schwarz.

  A prevalent example where randomization helps with runtime efficiency used to be Primality testing, where given an n-bit number, the goal is to determine if it is prime. While straightforward randomized algorithms for this problem existed since the 70's, which ran in polynomial time and erred only if the number was prime, it was not known until two years ago whether a deterministic polytime algorithm was possible—in fact, this problem was being viewed as evidence that the class of polytime randomized algorithms with one-sided error is more powerful than the class of polytime deterministic algorithms (the so called RP vs P question). With the discovery of a polytime algorithm for Primality, however, the question still remains open and can go either way.

- There are problems in which randomized algorithms help us where deterministic algorithms can provably not work. A common feature of such problems is **lack of information**. A classical example involves two parties, named A (for Alice) and B (for Bob), each in possession of an n bit number (say x and y resp.) that the other does not know, and each can exchange information with the other through an expensive communication channel. The goal is to find out whether they have the same number while incurring minimal communication cost. Note that the measure of efficiency here is the number of bits exchanged, and not the running time. It can be shown that a deterministic protocol has to exchange n bits in order to achieve guaranteed correctness. On the other hand, the following randomized protocol, based on the idea of *fingerprinting*, works correctly with almost certainty: A picks a prime p in the range $[2n, 4n]$ at random, computes $x \bmod p$, and sends both x and p to B, to which B responds with 1 or 0 indicating whether there was a match. With only $O(\log n)$ bits exchanged, the probability of error in this protocol–i.e., $\Pr[x \neq y \text{ and } x \bmod p = y \bmod p]$– can be shown to be at most $1/n$. By repeating the protocol, we can reduce the probability of an error to less than that of an asteroid colliding with either of A or B's residences, which should be a practically sufficient threshold.

We now list several more examples of tasks where randomization is useful:

Load Balancing: In this problem there are m machines and incoming jobs, and the goal is to assign the jobs to the machines such that the machines are equally utilized. One simple way that

turns out very effective here is to simply pick at random, given a job, one of the m machines and assign the job to it. We will study load balancing in a later lecture.

Symmetry Breaking: Randomization comes handy also in designing contention resolution mechanisms for distributed protocols, where a common channel is shared among the communicating entities of a network. The Ethernet protocol is a typical example, where a node that has something to send instantly attempts to send it. If two or more nodes happen to be sending at overlapping windows then they all abort, each pick at random a back-off duration, and reattempt after its picked time elapses. This turns out to be a very effective mechanism with good performance and without any prior communication between sending nodes.

Sampling: Another scenario where a randomized approach works really well is when we want to compute the integral of a multivariate function over a region. Similar to the polynomial identity testing example above, in the case of a univariate polynomial this problem can be solved rather easily, by piecewise linear approximations. But the same method does not work well in the general case as there are exponentially many pieces to compute. The only known efficient method for solving this problem uses randomization, and works as follows. It computes a bounding box for the function, samples points from the box, and determines what portion of the points sampled lies below the function, from which an approximate value for the integral is then computed. We may get to this problem later in the course.

The sampling method applies generally to a number of problems that involve counting. For example, the problem of finding the number of satisfying assignments of a boolean formula involving n variables, which is NP-hard as we will see a later lecture, can be reasonably approximated by sampling points from $\{0,1\}^n$.

Searching for witnesses: A recurring theme among the above examples is the task of searching for witnesses for verifying whether a certain property holds. Randomization yields effective techniques in these cases if the density of witnesses are high. For example, in Polynomial Identity Testing, it can be shown through algebra that if a polynomial is not identically zero, then the points where it evaluates to non-zero has high density relative to all points where it is defined.

The reader should be convinced by now that flipping coins is a powerful tool in algorithm design.

6.3 Elementary Probability Theory

In the following, let X_i be random variables with x_i their possible values, and let $\mathcal{E}_i$ be events.

We have the following definitions for a random variable X:

- Expectation: $\mathbf{E}[X] = \mu(X) = \sum_x x \cdot \mathbf{Pr}[X = x]$.
- Variance: $Var\,(X) = \mathbf{E}\big[(X - \mathbf{E}[X])^2\big] = \mathbf{E}\big[X^2\big] - \mathbf{E}[X]^2$
- Standard Deviation: $\sigma(X) = \sqrt{Var(X)}$

We have the following rules:

- Linearity of Expectation: $\mathbf{E}[\sum_i \alpha_i \cdot X_i] = \alpha_i \cdot \sum_i \mathbf{E}[X_i]$, where α_i's are scalars and no independence assumptions are made on X_i's. It follows from this rule that the above two definitions of variance are identical.
- Multiplication Rule: $\mathbf{E}[X_1 \cdot X_2] = \mathbf{E}[X_1] \cdot \mathbf{E}[X_2]$ if X_1, X_2 are independent.
- Bayes' Rule: If $\mathcal{E}_i$ are disjoint events and $\mathbf{Pr}[\bigcup_i \mathcal{E}_i] = 1$, then $\mathbf{E}[X] = \sum_i \mathbf{E}[X|\mathcal{E}_i] \cdot \mathbf{Pr}[\mathcal{E}_i]$.
- Union Bound: $\mathbf{Pr}[\bigcup_i \mathcal{E}_i] \leq \sum_i \mathbf{Pr}[\mathcal{E}_i]$.
- Markov's Inequality: If $X \geq 0$ with probability 1, then $\mathbf{Pr}[X > \alpha] \leq \frac{\mathbf{E}[X]}{\alpha}$

We only give a proof of the last statement:

Proof: This is a proof of Markov's inequality.

$$\begin{aligned} \mathbf{E}[X] &= \mathbf{E}[X|0 \leq X \leq \alpha] \cdot \mathbf{Pr}[0 \leq X \leq \alpha] + \mathbf{E}[X|\alpha < X] \cdot \mathbf{Pr}[\alpha < X] && (6.3.1) \\ &\geq \mathbf{E}[X|\alpha < X] \cdot \mathbf{Pr}[\alpha < X] && (6.3.2) \\ &\geq \alpha \cdot \mathbf{Pr}[\alpha < X] && (6.3.3) \end{aligned}$$

■

6.4 Global Min Cut

6.4.1 Description of the problem

Let $G = (V, E)$ be a graph. For simplicity, we assume G is undirected. Also assume the capacity of each edge is 1, i.e., $C_e = 1, \forall e \in E$. A *cut* is a partition $(S, \bar{S})$ of V such that $S \neq \emptyset, V$. We define the capacity of $(S, \bar{S})$ to be the sum of the capacities of the edges between S and $\bar{S}$. Since $C_e = 1, \forall e \in E$, it is clear that the capacity of $(S, \bar{S})$ is equal to the number of edges from S to $\bar{S}$, $|E(S, \bar{S})| = |(S \times \bar{S}) \cap E|$.

Goal: Find a cut of globally minimal capacity in the above setup.

6.4.2 Deterministic method

This problem can be solved by running max-flow computation (Ford-Fulkerson algorithm) $n - 1$ times, $n = |V|$. One fixes a node s to be the source, and let the sink run through all other nodes. In each case, calculate the min cut using the Ford-Fulkerson algorithm, and compare these min cuts. The minimum of the $n - 1$ min cuts will be the global min cut.

To see the above statement, consider the following: Let $(S, \bar{S})$ be the global min cut. For the node s we fixed, either $s \in S$ or $s \in \bar{S}$. If $s \in S$, then pick a node t in $\bar{S}$ as the sink. The max-flow computation with s as source and t as sink gives a flow f. f must be no greater than the capacity of $(S, \bar{S})$ since f has to flow from S to $\bar{S}$. So the min cut when s is source and t is sink must be no

greater than $(S, \bar{S})$. Therefore, when $t \in \bar{S}$ is chosen as sink, the min cut has the same capacity as the global min cut. In the case $s \in \bar{S}$, notice that G is undirected, so flow can be reversed and it goes back to the first case.

6.4.3 Karger's algorithm: a randomized method

Description of the algorithm:

1. Set $S(v) = \{v\}, \forall v \in V$.
2. If G has only two nodes v_1, v_2, output cut $(S(v_1), S(v_2))$.
3. Otherwise, pick a random edge $(u, v) \in E$, merge u and v into a single node uv, and set $S(uv) = S(u) \cup S(v)$. Remove any self-loops from G.

Theorem 6.4.1 *Karger's algorithm returns a global min cut with probability* $p \geq \frac{1}{\binom{n}{2}}$.

Proof: Let $(S, \bar{S})$ be a global min cut of G, and k be the capacity of $(S, \bar{S})$. Let $\mathcal{E}$ be the event that the Karger's algorithm produces the cut $(S, \bar{S})$.

First, we show that $\mathcal{E}$ occurs if and only if the algorithm never picks up any edge from S to $\bar{S}$. The "only if" part is clear, so we show the "if" part: Assume the algorithm doesn't pick up any edge of cut $(S, \bar{S})$. Let e be the first edge of cut $(S, \bar{S})$ which is eliminated by the algorithm, if there is any. The algorithm can only eliminate edges in two ways: Pick it up or remove it as a self loop. Since e can't be picked up by assumption, it must be eliminated as a self-loop. e becomes a self-loop if and only if its two end points u, v are merged, which means an edge e' connects u, v is selected in some iteration. In that iteration, u, v may be the combination of many nodes in original graph G, ie, $u = u_1u_2...u_i, v = v_1v_2...v_j$, but since none of the previous picked up edges is in cut $(S, \bar{S})$, the nodes merged together must belong to the same set of S or $\bar{S}$. So if e connects S to $\bar{S}$, the e' will also connects S to $\bar{S}$, which contradicts to the assumption that the algorithm doesn't pick up edges of cut $(S, \bar{S})$.

Second, notice that $(S, \bar{S})$ remains to be global min cut in the updated graph G as long as no edge of $(S, \bar{S})$ has been picked up. This is because any cut of the updated graph is also a cut in the original graph and the edges removed are only internal edges, which don't affect the capacity of the cut.

Now let $\mathcal{E}_i$ be the event that the algorithm doesn't pick any edge of $(S, \bar{S})$ in i-th iteration. Then,

$$\mathbf{Pr}[\mathcal{E}] = \mathbf{Pr}[\mathcal{E}_1] \cdot \mathbf{Pr}[\mathcal{E}_2|\mathcal{E}_1] \cdot \mathbf{Pr}[\mathcal{E}_3|\mathcal{E}_1 \cap \mathcal{E}_2]... \tag{6.4.4}$$

Observe that, in the i-th iteration, assuming $\bigcap_{1 \leq j \leq i-1} \mathcal{E}_j$ has occurred, every node has at least k edges. For otherwise the node with fewer than k edges and the rest of the graph would form a partition with capacity less than k. It follows that there are at least $(n - i + 1)k/2$ edges in i-th

iteration if no edge of $(S, \bar{S})$ has been selected. Therefore,

$$\mathbf{Pr}\left[\mathcal{E}_i | \bigcap_{1 \le j \le i-1} \mathcal{E}_j\right] \geq 1 - \frac{k}{(n-i+1)k/2} = \frac{n-i-1}{n-i+1} \tag{6.4.5}$$

$$\mathbf{Pr}[\mathcal{E}] \geq \frac{n-2}{n}\frac{n-3}{n-1}\cdots\frac{2}{4}\frac{1}{3} = \frac{2}{n(n-1)} \tag{6.4.6}$$

■

To decrease the chances of Karger's algorithm returning a non-minimal cut, as mentioned earlier in the lecture we can perform independent runs of the algorithm and take the smallest of the cuts produced. Then if anyone of the runs produces a min-cut, we get a correct solution. If the algorithm is independently run t times, then the probability that none of the runs result in the min-cut would be

$$\mathbf{Pr}[\texttt{Karger's algorithm is wrong in one run}]^t \leq \left(1 - \frac{2}{n(n-1)}\right)^t \tag{6.4.7}$$

In particular, if $t = k\frac{n(n-1)}{2}$, then roughly, $\left(1 - \frac{2}{n(n-1)}\right)^t \leq e^{-k}$.

One interesting implication of Karger's algorithm is the following.

Theorem 6.4.2 *There are at most $\binom{n}{2}$ global min-cuts in any graph G.*

Proof: For each global min-cut, the algorithm has probability at least $\frac{2}{n(n-1)}$ to produce that min-cut. Let $C_i, 1 \le i \le m$ be all the global min-cuts. Run the algorithm once and obtain the cut C. The probability that C is a global min-cut is

$$\mathbf{Pr}[C \texttt{ is global min cut}] = \sum_i \mathbf{Pr}[C = C_i] \geq m\frac{2}{n(n-1)} \tag{6.4.8}$$

Now since probability is always no greater than 1, we obtain $m \leq \frac{n(n-1)}{2}$. ■

CS787: Advanced Algorithms	
Scribe: Shijin Kong and Yiying Zhang	**Lecturer:** Shuchi Chawla
Topic: Randomized load balancing and hashing	**Date:** September 19, 2007

Today we give two applications of randomized Algorithms. The first one is load balancing, where the goal is to spread tasks evenly among resources. In order to analyze this problem, we first look at a related problem, the Balls and Bins problem. The second one is hashing. We give a basic randomized algorithm and an improved design.

7.1 Randomized Load Balancing and Balls and Bins Problem

In the last lecture, we talked about the load balancing problem as a motivation for randomized algorithms. Here we examine this problem more closely. In general, load balancing is a problem to distribute tasks among multiple resources. One example is assigning students to advisors. On any given day, m students will visit n professors. The subset of students visiting professors on any day is not known prior to making the assignment. The goal is to ensure that every advisor gets even load of students. An easy randomized method is to assign each student uniformly at random to any professor.

We can model this as tossing balls into bins. Here, balls can be considered as tasks and bins as resources. We discuss this problem first, since it is a simple way of describing load balancing and is a nice and more comprehensible example to study.

7.1.1 Balls and Bins

Consider the process of tossing m balls into n bins. The tosses are uniformly at random and independent of each other, which implies that the probability that a ball falls into any given bin is $1/n$. Based on this process, we can ask a variety of questions. Here, we describe several questions of interest and give answers to them.

Before we look at the questions, we define some notation. Let η_i^p denote the event of ball i fall into bin p; ε_{ij} be the event that ball i and ball j collide. Also, note that the collisions mentioned in the questions mean distinct and unordered pairs (e.g. if three balls fall into one bin, three collisions are counted).

Question 1: *What is the probability of any two balls falling into one bin?*

Alternatively, we can view this question as after the first ball falls into a particular bin p, what is the probability of the second ball falling into bin p. Since balls fall into any bin equally likely and tosses are independent of each other, the probability of the second ball falling into bin p is simply $1/n$. Thus the probability of any two balls falling into one bin is $1/n$.

There are two ways to formally prove the correctness of this probability. First, we can use Bayes Rule, which gives

$$\begin{aligned}\mathbf{Pr}[\mathcal{E}_{12}] &= \sum_{p=1}^{n} \mathbf{Pr}[\eta_2^p | \eta_1^p] \mathbf{Pr}[\eta_1^p] \\ &= \sum_{p=1}^{n} \frac{1}{n} \mathbf{Pr}[\eta_1^p] \\ &= \frac{1}{n} \end{aligned} \tag{7.1.1}$$

We can also sum up over p all the probability of the event that both balls fall into bin p

$$\begin{aligned}\mathbf{Pr}[\mathcal{E}_{12}] &= \sum_{p=1}^{n} \mathbf{Pr}\left[\eta_1^p \bigcap \eta_2^p\right] \\ &= \sum_{p=1}^{n} \frac{1}{n^2} \\ &= \frac{1}{n}\end{aligned}$$

Question 2: *What is the expected number of collisions when we toss m balls?*

To answer this question, we use a indicator random variable X_{ij} to indicate if an event happens. $X_{ij} = 1$ if $\mathcal{E}_{ij}$ happens, $X_{ij} = 0$ if $\mathcal{E}_{ij}$ doesn't happen. Note that since X_{ij} only takes zero or one as its value, they are Bernoulli variables, and tosses can be considered as a sequence of Bernoulli trials with a probability 1/n of success. We also define X to be the number of collisions, i.e. $X = \sum_{i \neq j} X_{ij}$. Then we calculate $\mathbf{E}[X]$ as follow

$$\begin{aligned}\mathbf{E}[X] &= \sum_{i \neq j} \mathbf{E}[X_{ij}] \\ &= \sum_{i \neq j} \mathbf{Pr}[X_{ij} = 1] \\ &= \frac{1}{n} \binom{m}{2} \end{aligned} \tag{7.1.2}$$

The answer to this question demonstrates an interesting phenomenon, **Birthday Paradox**. How many people must have there be in a room before there is at least 50% chance that two of them were born on the same day of the year? If we look at this problem as a balls and bins problem, people can be viewed as balls and days as bins. A related question is how many balls there need to be in order to get one collision in expectation in 365 bins, i.e. $\mathbf{E}[X] = \frac{1}{365}\binom{m}{2} = 1$. Solving this equation, we get $m \geq 23$. The answer is surprisingly few.

From now on, we assume $m = n$ for simplicity. All the analysis can be generalized to $m \neq n$ easily.

Question 3: *What is the probability of a particular bin being empty? What is the expected number of empty bins?*

The probability of a ball not fall into a particular bin is $1 - \frac{1}{n}$. Thus, we have

$$\mathbf{Pr}[\textit{bin } i \textit{ is empty}] = \left(1 - \frac{1}{n}\right)^n \to \frac{1}{e} \tag{7.1.3}$$

Using indicator random variables, we can also show that the expected number of empty bins is approximately $\frac{n}{e}$.

Question 4: *What is the probability of a particular bin having k balls?*

$$\mathbf{Pr}[\textit{bin } i \textit{ has } k \textit{ balls}] = \binom{n}{k}\left(\frac{1}{n}\right)^k\left(1 - \frac{1}{n}\right)^{n-k}$$

$$\mathbf{Pr}[\textit{bin } i \textit{ has } k \textit{ balls}] \leqslant \frac{n^k}{k!}\frac{1}{n^k} = \frac{1}{k!} \tag{7.1.4}$$

Question 5: *What is the probability of a particular bin having at least k balls?*

If we look at any subset of balls of size k, then the probability that the subset of balls fall into bin i is $\left(\frac{1}{n}\right)^k$. Note that we no longer have the $\left(1 - \frac{1}{n}\right)^{n-k}$ factor, because we don't care about where the rest of the balls fall. We then take a union bound of these probabilities over all $\binom{n}{k}$ subsets of size k. The events we are summing over, though, are not disjoint. Therefore, we can only show that the probability of a bin having at least k balls is at most $\binom{n}{k}\left(\frac{1}{n}\right)^k$.

$$\mathbf{Pr}[\textit{bin } i \textit{ has at least } k \textit{ balls}] \leqslant \binom{n}{k}\left(\frac{1}{n}\right)^k$$

Using Stirling's approximation, we have

$$\mathbf{Pr}[\textit{bin } i \textit{ has at least } k \textit{ balls}] \leqslant \left(\frac{e}{k}\right)^k \tag{7.1.5}$$

7.1.2 Randomized Load Balancing

After examining basic properties in the balls and bins problem, we now move on to the load balancing problem. In this section, we still use notations from the last section for easy understanding.

If we look at Question 5 in the last section, we need to find some k so that $\left(\frac{e}{k}\right)^k$ to be very small. What should k be to have $\frac{1}{k^k} \approx \frac{1}{n^2}$? The solution is $k = O\left(\frac{\ln n}{\ln \ln n}\right)$.

Therefore, we present the following theorem

Theorem 7.1.1 *With high probability, i.e. $1 - \frac{1}{n}$, all bins have at most $\frac{3 \ln n}{\ln \ln n}$ balls.*

Proof: Let $k = \frac{3 \ln n}{\ln \ln n}$.

From 7.1.5, we have

$$\begin{aligned}
\mathbf{Pr}[\textit{bin } i \textit{ has at least } k \textit{ balls}] &\leqslant \left(\frac{e}{k}\right)^k = \left(\frac{e \ln \ln n}{3 \ln n}\right)^{\frac{3 \ln n}{\ln \ln n}} \\
&\leqslant \exp\left(\frac{3 \ln n}{\ln \ln n}(\ln \ln \ln n - \ln \ln n)\right) \\
&= \exp\left(-3 \ln n + \frac{3 \ln n \ln \ln \ln n}{\ln \ln n}\right)
\end{aligned}$$

When n is large enough

$$\mathbf{Pr}[\textit{bin } i \textit{ has at least } k \textit{ balls}] \leqslant \exp\{-2 \ln n\} = \frac{1}{n^2}$$

Using Union Bound, we have

$$\mathbf{Pr}[\textit{any bin has at least } k \textit{ balls}] \leqslant n \frac{1}{n^2} = \frac{1}{n}$$

which implies

$$\mathbf{Pr}[\textit{all bins have at most } k \textit{ balls}] \geqslant 1 - \frac{1}{n}$$

■

Note that if we throw balls uniformly at random, balls are not uniformly distributed. But every bin has equal probability of having k balls.

Theorem 7.1.1 shows that if we distribute load or balls independently and uniformly at random, we can get maximum load or largest number of balls in any bin of approximately $\frac{3 \ln n}{\ln \ln n}$ with high probability. We can also show that

$$\mathbf{E}[\textit{maximum load}] = \frac{\ln n}{\ln \ln n}(1 + O(1))$$

Can we improve this number and distribute things more evenly? In the previous process, each time we pick one bin independently and uniformly at random, and then throw a ball. Instead suppose every time we pick two bins independently and uniformly at random, and put a ball into the bin with less balls. This increase the choices into *two random choices*. Now the expected maximum load is

$$\mathbf{E}[\textit{maximum load}] = \frac{\ln \ln n}{\ln 2} + O(1)$$

We see that this is a huge improvement over one random choice. We can also increase to t *random choices*, where every time we pick t bins independently and uniformly at random, and put a ball into the bin with least balls. We then get an expected max load of $\frac{\ln \ln n}{\ln t}$, which doesn't have much improvement over *two random choices*. This phenomenon is called *"Power of two choices"*.

7.2 Hashing

Another load balancing related problem is hashing. It is used to maintain a data structure for a set of elements for fast look-up. Two important applications of hashing are maintaining a dictionary and maintaining a list of easy-breaking passwords. Basically we have elements which are small subsets from a large universe and try to map those elements to an array. This mapping relation is defined as a hash function. The problem is formally described as this:

Given:

A set of elements S with size $|S| = m$ from universe U ($|U| = u$).

An array with storage for n elements.

Goal:

For $n = O(m)$, design a mapping h from S to array position to achieve $O(1)$ look-up time.

Two simple options to resolve this problem are:

1. Store only S in storage for m elements, the look-up time is $O(\log m)$ using sorting or binary search trees. The drawback is too slow in worst case.

2. Allocate storage for the whole universe U, the look-up time is $O(1)$, but the drawback is too large storage.

7.2.1 Ideal Situation - Truly Random

A simple solution for mapping is to use a truly random function. We have total n^u such functions for mapping: $U \to [n]$. A truly random hash function is picked from this universe uniformly at random. It also implies that any element will be uniformly mapped to any of the n array positions. There can be distinct elements $i, j \in S$ for which $h(i) = h(j)$. We say those two elements *collide*, since they are mapped to same place. A intuitive way to deal with collision is to build a link list under each array position. Then the look-up process for any element i will look like this:

1. Get the array position by calculating $h(i)$.

2. Scan the linked list under $h(i)$ to see if i exists in this linked list.

This hashing structure (in Fig 7.2.1) is called *chain-hashing*. As we know before, the probability for two elements i and j to collide is:

$$\mathbf{Pr}[elements\ i, j \in S\ collide] = \frac{1}{n} \tag{7.2.6}$$

If we fix i, the expected number of collisions for i is:

$$\mathbf{E}[number\ of\ collisions\ with\ i] = \frac{m-1}{n} \tag{7.2.7}$$

There are two key problems with such truly random function h. First, it's very difficult to simulate such perfect uniformly distributed hash function. Second, the worst case for look-up is not $O(1)$. The worst case look-up time is $\frac{\ln n}{\ln \ln n}$ (from Theorem 7.1.1), which is only slightly better than binary search.

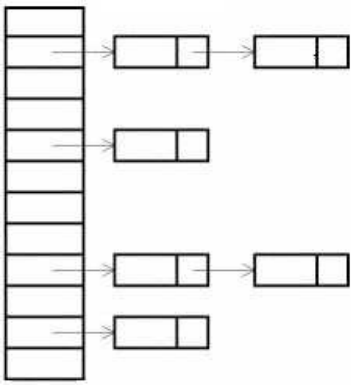


Figure 7.2.1: Chain-hashing

7.2.2 2-Universal Family of Hash Functions

To resolve the problem, we introduce a hash function class called *universal family* of hash functions, which is a family H of functions from U to $[n]$. These functions have three properties:

1. $\forall x \in U, i \in [n]$

$$\mathbf{Pr}[h(x) = i, h \in H] = \frac{1}{n} \tag{7.2.8}$$

2. $\forall x, y \in U$

$$\mathbf{Pr}[h(x) = h(y), h \in H] = \frac{1}{n} \tag{7.2.9}$$

3. (Desideratum) h is easy to compute and specify

Moreover, we can extend our definition of 2-universal hash functions to *strongly d-universal* hash functions:

$\forall x_1, x_2, \ldots, x_d \in U,\ i_1, i_2, \ldots, i_d \in [n]$

$$\mathbf{Pr}[h(x_1) = i_1, h(x_2) = i_2, \ldots, h(x_d) = i_d, h \in H] = \frac{1}{n^d} \tag{7.2.10}$$

For our bounds on the number of collisions, we only require 2-way independence between the mappings of the hash functions. Therefore 2-universal family of hash functions is enough.

7.2.3 An Example of 2-Universal Hash Functions

If we consider all the parameters as bit arrays:

$U = \{0,1\}^{\lg u} \setminus \{0^{\lg u}\}$

Array: $\{0,1\}^{\lg n}$

$H = \{0,1\}^{\lg n \times \lg u}$ (The set of all $\lg n \times \lg u$ 0-1 matrices)

Then, this hash function is like mapping a long vector into a short vector by multiplying matrix $h \in H$.

For example, for $u = 32$, $n = 8$, we can have a h mapping vector $x = (1,1,0,0,1)$ to $i = (0,1,1)$:

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

From this matrix formation, we can get two claims on hash functions:

Claim 1:

Fix input $x \in S$, then the probability that any given position in $h(x)$ is 1 is $\frac{1}{2}$.

Proof: Every position in h has $\frac{1}{2}$ probability to be 1. Consider the product of the j_{th} row of h and the vector x, any position of i after multiplication has $\frac{1}{2}$ probability to be 1.

Claim 2:

Fix input $x, y \in S, (x \neq y)$, the probability that the j_{th} position in $h(x)$ is the same as the j_{th} position in $h(y)$ is $\frac{1}{2}$.

Proof: For any position differs in x and y, if the corresponding position in h is 1, then $h(x)$ is different from $h(y)$ at position j. Otherwise, $h(x) = h(y)$ at position j. Therefore the claim.

Theorem 7.2.1 *The hash functions H described above is a 2-universal family of hash functions.*

Proof: Suppose $n = 2^t$, $t \in \mathbb{Z}$. From claim 1 we know that the probability any position in $h(x)$ is 1 is $\frac{1}{2}$. There are totally t positions in $h(x)$. Therefore the probability that $h(x)$ is a particular value is $\left(\frac{1}{2}\right)^t = \frac{1}{n}$, which satifies the first property of 2-univeral hash functions family (7.2.8).

From claim 2 we know that the probability that any position in $h(x)$ and $h(y)$ are same is $\frac{1}{2}$. Therefore the probability that $h(x) = h(y)$ is $\left(\frac{1}{2}\right)^t = \frac{1}{n}$, which is described as the second property of 2-univeral hash functions family (7.2.9).

Therefore H is 2-universal hash functions family. ■:

Finally, if mapping integer values: $[1, \ldots, u] \rightarrow [1, \ldots, n]$, a widely used hash function is to pick a prime $p(p \geq u)$, and $a, b \in [u], a \neq 0$, and $h_{a,b}(x) = ((ax + b) \; mod \; p) \; mod \; n$.

CS787: Advanced Algorithms

Scribe: Mayank Maheshwari, Chris Hinrichs **Lecturer:** Shuchi Chawla
Topic: Hashing and NP-Completeness **Date:** September 21 2007

Previously we looked at applications of randomized algorithms, and began to cover randomized hashes, deriving several results which will be used in today's lecture notes. Today we will finish covering Randomized hashing and move on to the definition of the classes P and NP, and NP-Completeness.

8.1 Hashing Algorithms

Sometimes it is desireable to store a set of items S taken from a universe U using an amount of storage space n and having average lookup time $O(1)$. One system which has these properties is a hash table with a completely random hash function. Last time we saw how to implement this using a 2-universal hash family. Unfortunately these hash families can have a large lookup time in the worst case. Sometimes we want the worst case lookup time to also be $O(1)$. We will now see how to achieve this *perfect hashing*.

8.1.1 Perfect hashing – reducing worst case loopkup time

Given a 2-uniform hashing function h (informally, a function which has a $1/n$ probability of assigning 2 different independently chosen elements of U to the same value i, also called "bin i") and that $|S| = m$, $|U| = u$ and $S \subseteq U$ we can construct an array $[n]$ of n bits such that if an element $i \in S$ maps to $h(i)$ then the array answers that i is in S. Note that there is a certain probability that this will be false because elements not in S can be mapped to the same location by the hash function. If there are more than one element in S mapped to the same bit i then the single bit in the array is replaced by a linked list of elements, which can be searched until the desired element is found. Under this arrangement, the worst case lookup time is determined by the longest expected linked list in $[n]$, which is equivalent to the expected number of assignments of elements in S to a single bin, i.e. collisions in bin i. The expectation of the number of collisions at bin i is given by

$$\mathbf{E}[X_i] = 1/n \binom{m}{2} \tag{8.1.1}$$

where X_i is a random variable representing the number of elements in S mapped to bin i. In order for the expected lookup time to be $O(1)$ the quantity on the right hand side of (8.1.1) must be $O(1)$ which means that n must be $O(m^2)$.

In order to ensure that $\mathbf{E}[X_i] = O(1)$ we can repeatedly choose new hash functions h from a 2-Universal family of functions $H : U \to [n^2]$ until one is found which produces no more than $O(1)$ collisions in any bin.

8.1.2 Perfect hashing with linear space

An improvement on this method of hashing elements of S which reduces its space complexity is to choose $n \approx m$ and instead of using a linked list of elements which map to the same bin, we can use another hash of size n to store the elements in the list. Let b_i be the number of elements in bin i. We note that the expected maximum number of elements in a single bin, is roughly $\sqrt{n}$ because we know from theorem 7.1.1 that $\mathbf{Pr}[\max_i b_i \geq k] \approx 1/k^2$ and thus $\mathbf{Pr}[\text{any } b_i \geq \sqrt{n}] \approx 1/n$, so we accpet with high confidence that no bin has more than $\sqrt{n}$ items in it. The size of this hash is roughly the square of the number of elements to be stored in it, making its expected lookup time $O(1)$ as discussed above.

If b_i is the number of elements placed in bin i, i.e. the number elements in the sub-hash in bin i, (if $b_i \neq 1$), then the total amount of space used by this method, T_{sp} is given by:

$$T_{sp} = m + \mathbf{E}\left[\sum_i b_i^2\right] \tag{8.1.2}$$

The first term is for the array itself, and the second is for the sub-hashes, each of which stores b_i items, and requires b_i^2 space. We can note that

$$\mathbf{E}\left[\sum_i b_i^2\right] = \sum_i 2 * \binom{b_i}{2} + \sum_i b_i \tag{8.1.3}$$

because

$$\binom{b_i}{2} = \frac{b_i * (b_i - 1)}{2}$$

The $\sum_i b_i$ term is equal to m, the number of items stored in the main hash array. We can approximate the summation term my saying that

$$\sum_i \binom{b_i}{2} \approx \frac{1}{m}\binom{m}{2}$$

T_{sp} then becomes

$$T_{sp} \approx 2m + \frac{2}{m}\frac{m(m-1)}{2} \approx 3m \tag{8.1.4}$$

8.2 Bloom Filters

Before we go on to the idea of bloom filters, let us define the concept of *false positive.*

Definition 8.2.1 (False Positive) *A given element $e \in U$ and $|S| = m$ and the elements of S are mapped to an array of size n. If the element $e \notin S$ but the hashing algorithm returns a 1 or "yes" as an answer i.e. the element e is mapped to the array, it is called a false positive.*

Now the probability of a false positive using perfect hashing is:

$$\begin{aligned} \mathbf{Pr}[\text{a false positive}] &= 1 - \mathbf{Pr}[0 \text{ in that position}] \\ &= 1 - (1 - \frac{1}{n})^m \\ (n \gg m) & \\ &\approx 1 - (1 - \frac{m}{n}) = \frac{m}{n} \end{aligned}$$

This probability of getting a false positive is considerably high. So how can we decrease this?

By using 2-hash functions, we can reduce this probability to $(\frac{m}{n})^2$. So, in general, by using k-hash functions, we can decrease the probability of getting a false postive to a significantly low value. So, a fail-safe mechanism to get the hashing to return a correct value by introducing redundancy in the form of k-hash functions is the basic idea of a bloom filter. *Problem:*Given an $e \in U$, we have to map S to $[n]$ using k hash functions.

So to solve this problem, we

- Find $h_1(e), h_2(e), \ldots, h_k(e)$.
- If any of them is 0, output $e \notin S$, else output $e \in S$.

Now the $\mathbf{Pr}$[a false positive] $= 1 - \mathbf{Pr}$[any one position is 0].

So let's say the

$$\mathbf{Pr}[\text{some given position is 0}] = (1 - \frac{1}{n})^{mk} \approx \exp(\frac{-mk}{n}) = (\text{say})p. \tag{8.2.5}$$

So $\mathbf{Pr}$[any given position is 1]$= 1-p$. And $\mathbf{Pr}$[all positions that e maps to are 1 given $e \notin S$]$=(1-p)^k$.

This is correct if all the hash functions are independent and k is small.

*Problem:*The probability of getting a false positive using k-hash functions is $(1-p)^k$. So how can we minimize it?

Let us say the function to be minimized is $f(k) = (1 - \exp(\frac{-km}{n}))^k$.

By first taking the log of both sides and then finding the first derivative, we get

$$\begin{aligned}\frac{d}{dk}\log(f(k)) &= \frac{d}{dk}k\log(1-\exp(\frac{-km}{n})) \\ &= \log(1-\exp(\frac{-km}{n})) + \frac{k}{1-\exp(\frac{-km}{n})}\exp(\frac{-km}{n})\frac{m}{n} \\ &= \log(1-p) - \frac{p}{1-p}\log(p)\end{aligned}$$

By substituting $p = \exp(\frac{-km}{n})$ and $\frac{km}{n} = -\log(p)$

To minimize this function, we need to solve the equation with its first derivative being equal to 0.

So to solve this equation:

$$\begin{aligned}&\log(1-p) - \frac{p}{1-p}\log p = 0 \\ \Rightarrow\ &(1-p)\log(1-p) = p\log p\end{aligned}$$

Solving this equation gives $p = \frac{1}{2}$.

So we get

$$\exp(\frac{-km}{n}) = \frac{1}{2} \Rightarrow k = \frac{n}{m}\ln 2$$

So with that value of k, we can deduce the value of $f(k)$ as:

$$\begin{aligned}f(k) &= \mathbf{Pr}[\text{positive}] \\ &= (1-p)^k \\ &= \frac{1}{2}^{\frac{n}{m}\ln 2} \\ &= c^{\frac{n}{m}}.\end{aligned}$$

If we choose $n = m\log m$ where $c < 1$ is a constant, this gives $f = \frac{1}{m}$.

So this way, we can reduce the probability of getting a false positive by using k-hash functions.

8.3 NP-Completeness

8.3.1 P and NP

When analyzing the complexity of algorithms, it is often useful to recast the problem into a decision problem. By doing so, the problem can be thought of as a problem of verifying the membership of a given string in a language, rather than the problem of generating strings in a language. The

complexity classes P and NP differ on whether a witness is given along with the string to be verified. P is the class of algorithms which terminate in an amount of time which is $O(n)$ where n is the size of the input to the algorithm, while NP is the class of algorithms which will terminate in an amount of time which is $O(n)$ if given a witness w which corresponds to the solution being verified. More formally,

$L \in$ NP iff $\exists$ P-time verifier V $\in$ P s.t.

$\forall$ x $\in$ L, $\exists$ w, $|w| = \text{poly}(|x|)$, V(x, w) accepts

$\forall$ x $\notin$ L, $\forall$ w, $|w| = \text{poly}(|x|)$, V(x, w) rejects

The class Co-NP is defined similarly:

$L \in$ Co-NP iff $\exists$ P-time verifier V $\in$ P s.t.

$\forall$ x $\notin$ L, $\exists$ w, $|w| = \text{poly}(|x|)$, V(x, w) accepts

$\forall$ x $\in$ L, $\forall$ w, $|w| = \text{poly}(|x|)$, V(x, w) rejects

An example of a problem which is in the class NP is Vertex Cover. The problem states, given a graph $G = (V, E)$ find a set $S \subseteq V$, then $\forall e \in E$, e is incident on a vertex in S, and $|S| \leq k$ for some number k. There exists a verifier V by construction; V takes as input a graph G, and as a witness a set $S \subseteq V$ and verifies that all edges $e \in E$ are incident on at least one vertex in S and that $|S| \leq k$. If G has no vertex cover of size less than or equal to k then there is no witness w which can be given to the verifier which will make it accept.

8.3.2 P-time reducibility

There exist some problems which can be used to solve other problems, as long as a way of solving them exists, and a way of converting instances of other problems into instances of the problem with a known solution also exists. When talking about decision problems, a problem A is said to reduce to problem B if there exists an algorithm which takes as input an instance of problem A, and outputs an instance of problem B which is guaranteed to have the same result as the instance in problem A, i.e. if L_A is the language of problem A, and L_B is the language of problem B, and if there is an algorithm which translates all $l \in L_A$ into $l_B \in L_B$ and which translates all $l' \notin L_A$ into $l'_B \notin L_B$ then problem A reduces to problem B.

The practical implication of this is that if an efficient algorithm exists for problem B, then problem A can be solved by converting instances of problem A into instances of problem B, and applying the efficient solver to them. However, the process of translating instances between problems must also be efficient, or the benefit of doing so is lost. We therefore define P-time reducibility as the process of translating instances of problem A into instances of problem B in time bounded by a polynomial in the size of the instance to be translated, written $A \leq_P B$. This reduction is also called Cook reduction.

8.3.3 NP-Completeness

If it is possible to translate instances of one problem into instances of another problem, then if there exists a problem L such that

$$\forall L' \in \text{NP},\ L' \leq_P L$$

then an algorithm which decides L decides every problem in NP. Such problems are called NP-Hard. If a problem is in NP-Hard and NP, then it is called NP-Complete. If there exists a P-time algorithm which decides an NP-Complete problem, then all NP problems can be solved in P-time, which would mean that $NP \subseteq P$. We already know that $P \subseteq NP$ because every P-time algoritm can be thought of as an NP algorithm which takes a 0-length witness. Therefore, $P = NP$ iff there exists a P-time algorithm which decides any NP-Complete problem. This result was proved independently by Cook and Levin, and is called the Cook-Levin theorem. The first problem proved to be in NP-Complete was Boolean SAT, which asks, given a Boolean expression, is there a setting of variables which allows the entire expression to evaluate to True?

CS787: Advanced Algorithms

Scribe: Shijin Kong and David Malec **Lecturer:** Shuchi Chawla
Topic: NP-Completeness, Approximation Algorithms **Date:** 10/1/2007

As we've already seen in the preceding lecture, two important classes of problems we can consider are P and NP. One very prominent open question is whether or not these two classes of problems are in fact equal. One tool that proves useful when considering this question is the concept of a problem being complete for a class. In the first two portions of this lecture, we show two problems to be complete for the class NP, and demonstrate two techniques for proving completeness of a problem for a class. We then go on to consider approximation algorithms, which have as a goal providing efficient near-optimal solutions to problems for which no known efficient algorithm for finding an optimal solution exists.

9.1 Cook-Levin Theorem

The Cook-Levin Theorem gives a proof that the problem SAT is NP-Complete, via the technique of showing that any problem in NP may be reduced to it. Before proceeding to the theorem itself, we revisit some basic definitions relating to NP-Completeness.

First, we need to understand what problems belong to the classes P and NP. If we restrict ourselves to decision problems, every problem has a set of accepted inputs; we call this set of inputs a language, and can think of the original problem as reducing to the question of whether or not a given input is in this language. Then, P is the class of languages for which membership can be decided in polynomial time, and NP is the class of languages such that $x \in L$ if and only if there's a certificate of its membership that can be verified in polynomial time.

Second, we need the notion of reducibility of one problem to another. We say that a that a problem B can be reduced to A, or $B \leqslant_p A$ if, given access to a solution for A, we can solve B in polynomial time using polynomially many calls to this solution for A. This idea of reducibility is crucial to completeness of a problem for a class. If a problem $L \in NP$ satifies that for any $L' \in NP$, $L' \leqslant_p L$, then we say L is NP-Complete. It is worth noting that for proving NP-Completeness, typically a more restrictive form of reduction is used, namely a Karp reduction. It follows the same basic structure, but requires that only one query be made to the solution for the problem being reduced to, and that the solution be directly computable from the result of the query.

The Cook-Levin Theorem shows that SAT is NP-Complete, by showing that a reduction exists to SAT for any problem in NP. Before proving the theorem, we give a formal definition of the SAT problem (Satisfiability Problem):

Given a boolean formula ϕ in CNF (Conjuctive Normal Form), is the formula satisfiable? In other words, we are given some boolean formula ϕ with n boolean variables $x_1, x_2, \ldots, x_n$, $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$, where each of the C_i is a clause of the form $(t_{i1} \vee t_{i2} \vee \ldots \vee t_{il})$, with each l_{ij} is a literal drawn from the set $\{x_1, x_2, \ldots, x_n, \bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n\}$. We need to decided whether or not the exists some setting of the $x_1, x_2, \ldots, x_n$ that cause the formula ϕ to be satisfied (take on a boolean

value of true).

Theorem 9.1.1 *Cook-Levin Theorem: SAT is NP-complete.*

Proof: Suppose L is a NP problem; then L has a polynomial time verifier V:

1. If $x \in L$, $\exists$ witness y, $V(x, y) = 1$
2. If $x \notin L$, $\forall$ witness y, $V(x, y) = 0$

We can build a circuit with polynomial size for the verifier V, since the verifier runs in polynomial time (note that this fact is nontrivial; however, it is left to the reader to verify that it is true). The circuit contains AND, OR and NOT gates. The circuit has $|x| + |y|$ sources, where $|x|$ of them are hardcoded to the values of the bits in x and the rest $|y|$ are variables.

Now to solve problem L, we only need to find a setting of $|y|$ variables in the input which causes the circuit to output a 1. That means problem the problem L has been reduced to determining whether or not the circuit can be caused to output a 1. The following shows how the circuit satisfication problem can be reduced to an instance SAT.

Each gate in the circuit can be represented by a 3CNF (each clause has exactly three terms). For example:

1. The functionality of an OR gate with input a, b and output Z_i is represented as:
   $(a \vee b \vee \bar{Z}_i) \wedge (Z_i \vee \bar{a}) \wedge (Z_i \vee \bar{b})$
2. The functionality of a NOT gate with input a and output Z_i is represented as:
   $(a \vee Z_i) \wedge (\bar{a} \vee \bar{Z}_i)$

   Notice although some of the clauses have fewer than 3 terms, it is quite easy to pad them with independant literals to form clauses in 3CNF. The value of an independent literal won't affect the overall boolean value of the clauses. We essentially include clauses for it taking the values 0 or 1; regardless of the actual value, our clauses have the same value as the simpler two-term clause.

Suppose we have q gates in V marked as $Z_1, Z_2, \ldots, Z_q$ with Z_q representing the final output of V. Each of them either takes some of the sources or some intermediate output Z_i as input. Therefore the whole circuit can be represented as a formula in CNF:

$\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_q \wedge Z_q$

where

$C_i = (t_1 \vee t_2 \vee t_3), t_1, t_2, t_3 \in (x, y, Z_1, Z_2, \ldots\ Z_q, \bar{Z}_1, \bar{Z}_2, \ldots, \bar{Z}_q)$

As we said before, even if the last clause of ϕ has only one term Z_q, we may extend ϕ to an equivalent formula in 3CNF by adding independent variables. Thus, we have shown that the circuit can be reduced to ϕ, a formula in 3CNF which is satisfiable if and only if the original circuit could be made to output a value of 1. Hence, $L \leqslant_p$ SAT. Since SAT can be trivially shown to be in NP (any

satisfying assignmant may be used as a certificate of membership), we may conclude that SAT is NP-Complete. ■

Furthermore, it should be noted that throughout the proof, we restricted ourselves to formulas that were in 3CNF. In general, if we restrict the formulas under consideration to be in k-CNF for some $k \in \mathbb{Z}$, we get a subproblem of SAT which we may refer to as k-SAT. Since the above proof ensured that ϕ was in 3CNF, it actually demonstrated not only that SAT is NP-Complete, but that 3-SAT is NP-Complete as well. In fact, one may prove in general that k-SAT is NP-Complete for $k \geq 3$; for $k = 2$, however, this problem is known to be in P.

9.2 Vertex Cover

One example of an NP-Complete problem is the vertex cover problem. Given a graph, it essentially asks for a subset of the vertices which cover the edges of the graph. It can be phrased both as a search problem and as a decision problem. Formally stated, the search version of the vertex cover problem is as follows:

Given: A graph $G = (V, E)$

Goal: A subset $V' \subseteq V$ such that for all $(u, v) \in E$, $u \in V'$ or $v \in V'$, where $|V'|$ is minimized.

For a concrete example of this problem, consider the following graph:

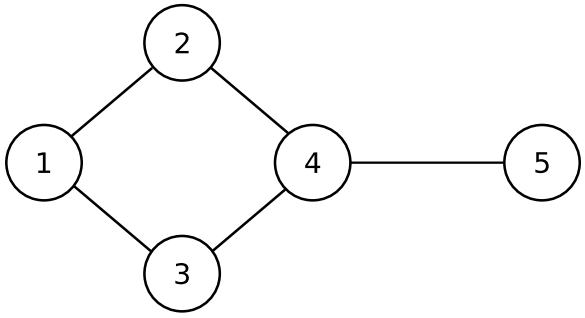


Figure 9.2.1: The nodes 1 and 4 form a minimal vertext cover

The decision variant of this problem makes only two changes to its specification: a positive integer k is added to the givens; and rather than searching for a V' of minimal size, the goal is to decide whether a V' exists with $|V'| = k$.

Furthermore, we can reduce the search version of this problem to the decision version. We know that the size of a minimal vertex cover must be between 0 and V, so if we are given access to a solution for the decision version we can just do a binary search over this range to find the exact value for the minimum possible size for a vertex cover. Once we know this, we can just check vertices one by one to see if they're in a minimal vertex cover. Let k be the size we found for a minimal vertex cover. For each vertex, remove it (and its incident edges) from the graph, and check whether it is possible to find a vertex cover of size $k - 1$ in the resultant graph. If so, the removed vertex is in a minimal cover, so reduce k by 1 and continue the process on the resulting graph. If

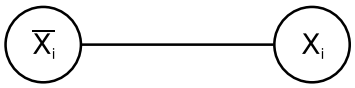


Figure 9.2.2: The gadget X_i

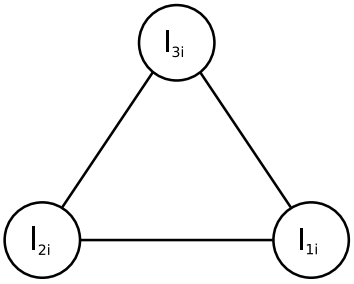


Figure 9.2.3: The gadget C_i

not, add the vertex back into the graph, and try the next one. When k reaches 0, we have our vertex cover. This property of being able to solve a search problem via queries to an oracle for its decision variant is called self-reducibility.

Theorem 9.2.1 *The decision version of Vertex Cover is NP-Complete.*

Proof: It is immediate that Vertex Cover $\in$ NP. We can verify a proposed vertex cover of the appropriate size in polynomial time, and so can use these as certificates of membership in the language.

All we need to show then is that VC is NP-Hard. We do so by reducing 3-SAT to it. Given some formula ϕ in 3-CNF, with n variables $x_1, \ldots, x_n$ and m clauses $c_1, \ldots, c_m$, we will build a graph, starting with the following gadgets.

For each variable x_i, we will build a gadget X_i. X_i will have two vertices, one corresponding to each of x_i and $\bar{x_i}$, and these two vertices will be connected by an edge (see figure 9.2.2). Notice that by construction, any vertex cover must pick one of these vertices because of the edge between them.

For each clause c_i, we will build a gadget C_i. Since ϕ is in 3-CNF, we may assume that $c_i = (l_{i1} \vee l_{i2} \vee l_{i3})$ for some set of literals l_{i1}, l_{i2}, l_{i3}. The gadget C_i will consist of a complete graph of size 3, with a vertex corresponding to each of the literals in the clause c_i (see figure 9.2.3). Notice that since all of the vertices are connected by edges, any vertex cover must include at least two of them.

Given the preceding gadgets for each of the variables x_i and each of the clauses c_i, we describe how the gadgets are connected. Consider the gadget C_i, representing the clause $c_i = (l_{1i} \vee l_{2i} \vee l_{3i})$. For each of the literals in c_i, we add an edge from the corresponding vertex to the vertex for the appropriate boolean value of the literal. So if $l_{i1} = x_k$, we will add an edge from the vertex for l_{i1} in the clause gadget C_i to the vertex for x_k in the variable gadget X_k; similarly, if $l_{i1} = \bar{x_k}$, we add and edge to the vertex for $\bar{x_k}$ in X_k. Figure 9.2.4 gives the connections for an example where $c_i = (x_1 \vee \bar{x_2} \vee x_4)$.

Now that we have our graph, we consider what we can say about its minimum vertex cover. As previously noted, at least one vertex from any variable gadget must be included in a vertex cover; similarly, at least 2 vertices from any clause gadget must be included. Since we have n variables and m clauses, we can thus see that no vertex cover can have size strictly less than $n + 2m$.

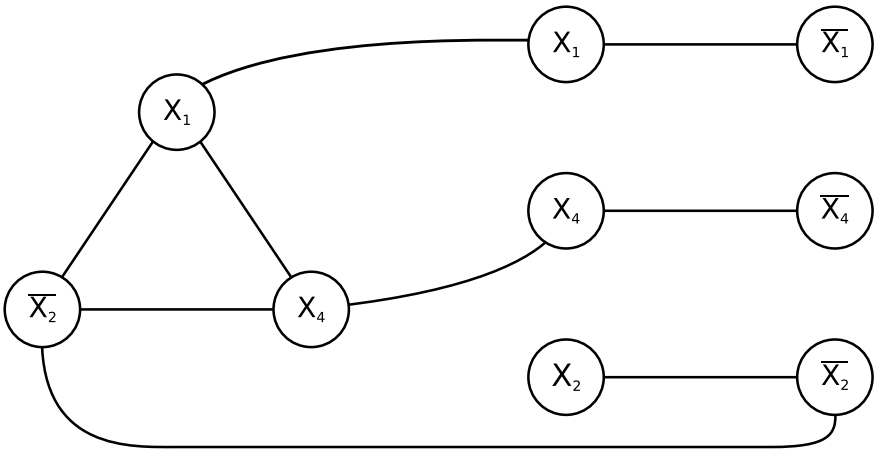


Figure 9.2.4: Connecting the gadgets

Furthermore, we show that a vertex cover of exactly this size exists if and only if the original formula ϕ is satisfiable.

Assume that such a vertex cover exists. Then, for each of the gadgets X_i, it contains exactly one vertex. If we set each x_i so that the node in the vertex cover corresponds to a boolean value of true, we will have a satisfying assignment. To see this, consider any clause c_i. Since the vertex cover is of size exactly $n + 2m$, only two of the vertices in the gadget C_i are included. Now, the vertex which wasn't selected is connected to the node corresponding to its associated literal. Thus, since we have a vertex cover, that node must be in it. So, the unselected literal must have a boolean value of true in our setting of the x_i, and so the clause is satisfied.

Similarly, if an assignment of the x_i exists which satisfies ϕ, we can produce a vertex cover of size $n+2m$. In each gadget X_i, we simply choose the vertex corresponding to a boolean value of true in our satisying assignment. Then, since it is a satisfying assignment, each clause contains at least one literal with a boolean value of true. If we leave the corresponding vertex, and include the others, we get a vertex cover. The edges inside the gadget C_i are satisfied, since we have chosen all but one of its vertices; the edges from it to variable gadgets are satisfied, since for each one, either the vertex in C_i has been included or the vertex in the corresponding X_i has been chosen.

Thus, we can see that the original formula is satisfiable if and only if the constructed graph has a vertex cover of size $n + 2m$. Thus, since 3SAT is NP-Hard, we get that VC is NP-Hard as well. As we have previously noted, VC $\in$ NP, and hence VC is NP-Complete. ■

9.3 Approximation Algorithms

In this section we consider another topic: approximation algorithms. There are many cases where no polynomial time algorithm is known for finding an optimal solution. In those situations, we can compromise by seeking a polynomial time near-optimal solution. However, we want to guarantee in this case that this alternative is close to optimal in some well-defined sense. To demonstrate

that a solution generated by an approximation algorithm is close to optimal, we need to compare its solution to the optimal one. One way of formalizing the idea of "close enough" is that of α-approximations which may be formalized with the following definitions.

For an NP optimization problem $A = (F,\ val)$, where given an instance $I \in A$:

1. $F(I)$ denotes the set of feasible solutions to this problem.
2. $val(x, I)$ denotes the value of solution $x \in F(I)$.

Our purpose is to find the optimal solution $OPT(I)$:

$$OPT(I) = \min_{x \in F(I)} val(x, I) \tag{9.3.1}$$

Suppose y^* is the optimal solution for instance I, then an α-approximation y to the optimal solution must satisfy:

$$\frac{val(y, I)}{OPT(I)} \leqslant \alpha \tag{9.3.2}$$

if A is a minimization problem, and

$$\frac{OPT(I)}{val(y, I)} \leqslant \alpha \tag{9.3.3}$$

if A is a maximization problem.

(9.3.4)

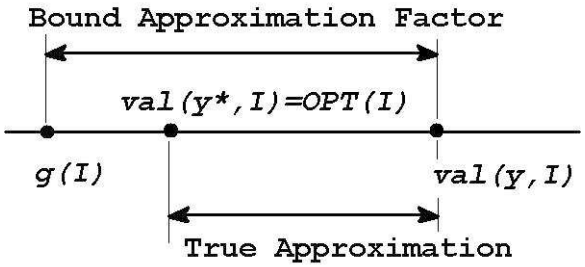


Figure 9.3.5: Lower bound for optimal solution

An optimal solution is usually computationally hard to find, so we can't directly judge whether a solution y is α-approximation or not; however, for some problems, it is possible to find a lower bound for the optimal solution $g(I)$ (Figure 9.3.5) which is good enough. Instead of computing the exact value of α, we obtain a reasonably upper bound for it using our lower bound on the optimal solution. If we can show $\frac{val(y,I)}{g(I)} \leqslant \alpha$, then we know y to be an α-approximation.

9.3.1 α-Approximation of Vertex Cover Problem

Now, we try to get an α-approximation for the vertex cover problem.

Lemma 9.3.1 *Given a graph G, the size of any matching in G is a lower bound on the size of any vertex cover.*

Proof: The key insight here is that the edges in a matching do not share any common vertices. So according to the definition of the vertex cover problem, any vertex cover problem must choose at least one vertex from any edge in the matching. Therefore, the size of any matching provides a lower bound for the size of any vertex cover. ■

Using a maximal matching with the above will give us a sufficiently good bound to derive an α-approximation.

Lemma 9.3.2 *Given a graph G and any maximal matching M on G, we can find a vertex cover for G of size at most $2|M|$.*

Proof: First, we prove that for any maximal matching M, there exists a solution for the vertex cover problem with size no more than $2|M|$. We start by choosing both vertices incident on each of the edges in M. This gives us a set of vertices of size $2|M|$. We claim this vertex set constitutes a valid vertex cover. Assume by way of contradiction that it does not; then we can find a edge e of which neither node is picked present in our set of vertices. But then, we may add e to our original matching M to form a larger matching M'. But by assumption M is a maximal, and so this is a contradiction. Thus, we have found a vertex cover for G of size $2|M|$ ■

Theorem 9.3.3 *There exists a polynomial time 2-approximation for vertex cover problem.*

Proof: Finding a maximum matching M can be done in polynomial time. Once we find M, we simply apply the strategy outlined above to get our vertex cover. Since we showed that the size of any matching provides a lower bound for the size of any vertex cover, we may conclude that the optimal vertex cover must be of size at least $|M|$. Thus, since our proposed vertex cover will have size $2|M|$, our algorithm will provide a 2-approximation for the vertex cover problem. ■

CS787: Advanced Algorithms

Scribe: Yiying Zhang and Baris Aydinlioglu
Lecturer: Shuchi Chawla
Topic: Local Search Based Appoximation
Date: October 03, 2007

Today we talk about an important technique of approximation algorithms: local search. We first give fundamental ideas of local search approximation. Then we look into two problems where local search based approximation algorithms apply, max-cut problem and facility location problem.

10.1 Fundamentals of Local Search

The basic idea of local search is to find a good local optima quickly enough to approximate the global optima. This idea is somewhat like hill climbing. Starting from an arbitrary point, we try to go down or climb up a little step and compare the objective value at the new point to that at the old point. We then choose the step that increase the objective value. In this manner, we finally reach a local maximal. Such algorithm is also known as *ascending or descending algorithm*.

We can also explain local search method using a graph. We consider a graph, whose nodes are feasible solutions and whose edges connect two "neighboring" feasible solutions. Two feasible solutions are neighbors if from either one we can obtain the other by a single "local" step. What constitutes a local step depends on the problem, as we will see through examples later. Note that we do not construct the entire graph, as its size would usually be exponential.

Once the structure of the graph is decided, in order to find a local optimum, we start with an arbitrary node in the graph of feasible solutions, and we look at its neighbors to see if there is a node with a better objective value. If there is, we find we move to that node feasible solution in the graph. We don't necessarily look for the neighboring node with the best objective value; in most cases a better value is enough. We repeat this process until there is no neighbor with a better objective value. At that point we find the local optimum. See figure below.

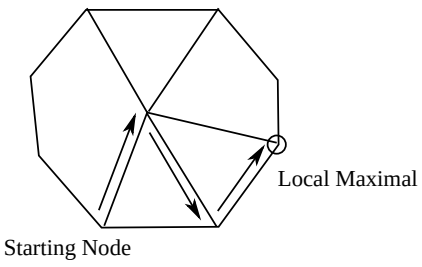


Figure 10.1.1: Graph Representation of Local Search

Now, the problem is how to design a good neighborhood structure for the graph. On one hand, we cannot allow for any node to be neighbor to too many other nodes. Otherwise in the extreme case we would get a complete graph and it would take exponential time to consider the neighbors of even a single node. On the other hand, we cannot have too few neighbors for a node, because we want any locally optimal node in the graph to have objective value within a factor of that of the global optimum.

In summary, we seek the following properties in a graph of feasible solutions:

1. Neighborhood of every solution is small
2. Local optima are nearly as good as global optima
3. We can find some local optima in a polynomial number of steps

The third point is often proven by showing that at each step we improve the objective value by some large enough factor. If the original solution has objective value V and the local optimal solution has objective value V^*, and at each step the objective value is improved by $(1+\epsilon)$, then we will have $\log_{(1+\epsilon)} \frac{V^*}{V}$ steps. We will give more details in the following examples.

10.2 Max-Cut Problem

Problem Definition:

Given: Unweighted graph $G = (V, E)$

Goal: Determine a cut $(S, \overline{S})$, where $S \neq V$ and $S \neq \emptyset$, such that $val(s) = \sum_{e \in E \cap (S \times \overline{S})} w_e = |E \cap (S \times \overline{S})|$ is maximized.

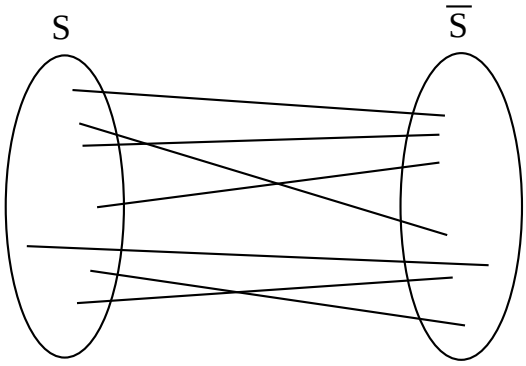


Figure 10.2.2: A cut (S, $\overline{S}$)

This problem is known to be NP-hard, which may appear surprising since its minimization version is efficiently solvable.

Since any nontrivial cut yields a feasible solution, we already know the nodes of our graph of feasible solutions. We need to determine a neighborhood structure on this graph, in other words we need to decide on what constitutes a local step.

Given a solution $(S, \overline{S})$, we consider a local step as moving a vertex from one side of the partition to another. So if v is in S (resp. $\overline{S}$), then moving v to $\overline{S}$ (S) would be a local step.

Our algorithm then is the following. Start with any nontrivial partition. Take local steps until a partition is reached such that none of its neighbors give a larger cut. Return this last partition.

Theorem 10.2.1 *The max-cut algorithm described above runs in $O(n^2)$ steps, where n is the number of vertices in the graph.*

Proof: At every step, we improve the size of the cut by at least 1. Therefore, the number of steps cannot be more than the total number of edges in the graph, which is $O(n^2)$. ■

Theorem 10.2.2 *If $(S, \overline{S})$ is a local optimal cut and $(S^*, \overline{S^*})$ is the global optimal cut, then $val(S) \geq \frac{1}{2} val(S^*)$.*

Proof: Let $(S, \overline{S})$ be locally optimal. For any vertex v in the graph, denote by $C(v)$ the number of neighbors of v that are across the cut. So if v is in S, then $C(v) = |(u, v) \in E : \ u \in \overline{S}|$. Similarly, denote by $C'(v)$ the number of neighbors of v that are in the same side of the cut as v. So if $v \in S$, then $C'(v) = |(u, v) \in E : \ u \in S|$.

Observe that for any vertex v in the graph we must have $C'(v) \leq C(v)$, for otherwise we can move v to the other partition and obtain a bigger cut, contradicting the local optimality of $(S, \overline{S})$.

Now, since $val(S)$ is the number of edges in the cut, the number of edges that are not in the cut is $|E| - val(S)$. Thus we have

$$val(S) = \frac{1}{2} \sum_{v \in V} C(v) \quad \geq \quad \frac{1}{2} \sum_{v \in V} C'(v) = |E| - val(S) \ .$$

This tells us that the number of edges in the cut is at least half of the total number of edges in the graph, $|E|$. But we know that $|E|$ is an upper bound on $val(S^*)$, the number of edges in the global max-cut. Therefore we have proved the claim. ■

Theorem 10.2.2 shows that our local search algorithm is a 2-approximation. There is another algorithm that yields a 2-approximation, which uses randomization. In that algorithm, we put a vertex in S or $\overline{S}$ at random (with a probability of $\frac{1}{2}$, v gets assigned to S). Then any edge in the graph has a probability of $\frac{1}{2}$ to be in the cut. So in expectation we get at least half of the edges in the cut. We will not go into details of this algorithm.

For a long while a less-than-2-approximation to this problem was not known, but now there is an algorithm that achieves a 1.3-approximation that uses sophisticated techniques. We will study this algorithm in a few weeks.

10.3 Facility Location

Most of the difficulty in the study of local search approximation algorithms tends to be in the analysis activity rather than in the design activity. We now move on to a problem that is more representative of this fact: a simple algorithm, but an involved analysis.

10.3.1 Problem Definition

Consider the following prominent problem in Operations Research, named **Facility Location**. A company wants to distribute its product to various cities. There is a set I of potential locations where a storage facility can be opened and a fixed "facility cost" f_i associated with opening a storage facility at location $i \in I$. (Sometimes we will say "facility i" to mean "facility at location i"). There also is a set J of cities to which the product needs to be sent, and a "routing cost" $c(i, j)$ associated with transporting the goods from a facility at location i to city j. The goal is to pick a subset S of I so that the total cost of opening the facilities and transporting the goods is minimized.

In short, we want to minimize

$$C(S) = C_f(S) + C_r(S) \ , \tag{10.3.1}$$

where

$$C_f(S) = \sum_{i \in S} f_i \ , \tag{10.3.2}$$

and

$$C_r(S) = \sum_{j \in J} \min_{i \in S} c(i, j) \ . \tag{10.3.3}$$

Notice that once a subset S of I is chosen, which facilities service which cities is automatically determined by the way $C_r(S)$ is defined.

In the way we have stated it so far, this problem is a generalized version of `SET COVER`, which can be seen by the following reduction: Given a `SET COVER` instance $(S, \mathcal{P})$, where $\mathcal{P} \subset 2^S$, map each $P \in \mathcal{P}$ to a facility and each $s \in S$ to a city, then set f_P to 1 and set $c(P, s)$ to either 0 or $+\infty$ depending on if $s \in P$ or not, respectively.

This reduction implies that the Facility Location problem would not allow a better approximation than what we can best do with `SET COVER`. In today's lecture we want to study a constant factor approximation to Facility Location, and for this purpose we impose some restrictions on the problem.

We restrict the costs $c(i, j)$ to form a *metric*. A metric is a function that is symmetric and that satisfies the triangle equality. For the costs in our problem this means (respectively) that we have $c(i, j) = c(j, i)$ for each $i, j \in I \cup J$, and that we have $c(i, j) \leq c(i, k) + c(k, j)$ for all $i, j, k \in I \cup J$. Notice that we have extended the notion of cost to in-between facilities and also cities. To make sense out of this, just think of costs as physical distances, and interpret equation (10.3.3) as each city being serviced by the nearest open facility.

10.3.2 A local search algorithm

Our local search algorithm works as follows. We pick any subset S of the set I of facilities. This gives us a feasible solution right away. We then search for the local neighborhood of the current feasible solution to see if there is a solution with a smaller objective value; if we find one we update our feasible solution to that one. We repeat this last step until we do not find a local neighbor that yields a reduction in the objective value. There are two types of "local steps" that we take in searching for a neighbor: i) remove/add a facility from/to the current feasible solution, or ii) swap a facility in our current solution with one that is not.

It is possible to consider variations on the local steps such as removing/adding more than one facility, or swapping d facilities out and bringing e facilities in, etc.. With more refined local steps, we can reach better local optima, but then we incur additional runtime due to the neighborhood graph getting denser. For our purposes the two local steps we listed above suffice, and indeed we show below that our algorithm with these two local steps give us a 5-factor approximation[1].

10.3.3 Analysis

Theorem 10.3.1 *Let S be a local optimum that the above algorithm terminates with, and S^* be a global optimum. Then*

$$C(S) \leq 5 \cdot C(S^*) \ .$$

We break up the proof of this theorem into two parts. First we show an upper bound on the routing cost of a local optimum, then we do the same for the facility cost.

Lemma 10.3.2

$$C_r(S) \leq C(S^*) \ .$$

Proof:

Consider adding some facility i^* in the globally optimal solution S^* to the locally optimal S to obtain S'. Clearly, $C(S) \geq C(S')$ because our algorithm must have explored S' as a local step before concluding that S is locally optimal. If i^* was already in S then there is no change. So we have

$$C_f(S') - C_f(S) + C_r(S') - C_r(S) \geq 0 \ .$$

Since $S' = S \cup \{i^*\}$, we rewrite the above as

$$f_{i^*} + C_r(S') - C_r(S) \geq 0 \ . \tag{10.3.4}$$

Before moving on we introduce some notation for brevity. Given (a feasible solution) $S \subset I$, a facility location i, and a city j, we define the following:

— $N_S(i)$ = the set of cities that are serviced by facility i in S,[2]

[1] Actually, these two steps give us a 3-factor approximation, but we do not cover that analysis here.

[2] For reasons that will become clear later, a better way to think of the "facility i services city j in S" relation is "facility i is the closest to j among all facilities in S". For brevity we will continue to speak of the "service" relation, but the latter interpretation lends itself better to generalizations and should be kept in mind.

— $\sigma^S(j)$ = the facility that services city j in S – i.e., facility i such that $j \in N_S(i)$,

— $\sigma^*(j) = \sigma^{S^*}(j)$,

— $\sigma(j) = \sigma^{S'}(j)$.

As mentioned earlier, by the way the objective function C was defined (see 10.3.3), given the set of facilities S', the decision of which city gets served by which facility is automatically determined: each city gets served by the nearest facility and indeed this is the best we can do with S'. We exploit this fact next, by doing worse with S'.

Consider $N_{S^*}(i^*)$, the set of cities that are serviced by i^* in the globally optimal solution S^*. Now, some of these cities may be serviced, in the solution S', by facilities other than i^*, because S' may contain facilities that are closer to them. What if we "re-route" S' so that each of these cities in $N_{S^*}(i^*)$ actually gets served by i^*, and not necessarily by the nearest open facility to it? Clearly we would incur a non-negative increase in the total routing cost of S'.

We can state the last argument as it corresponds to the problem formulation. Consider $C_r(S')$, the routing costs of S'. $C_r(S')$ is the sum of the distances of each city to a nearest facility in S'. What we really mean by "re-route" is to just take some terms in this sum and substitute it with terms that are not smaller than the ones we take out. In particular, for each city $j \in N_{S^*}(i^*)$, we take out $c(\sigma(j), j)$, the shortest distance between j and a facility in S', and instead substitute it with $c(i^*, j)$, a possibly larger–but surely not smaller–distance. Therefore the resulting sum is not smaller than $C_r(S')$. Call this new sum $\bar{C}_r(S')$.

Observe that by construction, $\bar{C}_r(S')$ differs from $C_r(S)$ only in the terms $j \in N_{S^*}(i^*)$, and that we have:

$$\bar{C}_r(S') - \sum_{j \in N_{S^*}(i^*)} c(i^*, j) \;=\; C_r(S) - \sum_{j \in N_{S^*}(i^*)} c(\sigma(j), j) \ .$$

Combining the above with (10.3.4), we obtain:

$$0 \;\leq\; f_{i^*} + \bar{C}_r(S') - C_r(S) \;=\; f_{i^*} + \sum_{j \in N_{S^*}(i^*)} [c(i^*, j) - c(\sigma(j), j)] \ .$$

Repeating this for all $i^* \in S^*$ and summing up, we get

$$\sum_{i^* \in S^*} \left[f_{i^*} + \sum_{j \in N_{S^*}(i^*)} [c(i^*, j) - c(\sigma(j), j)] \right] \geq 0 \ .$$

Rearranging, we get

$$\sum_{i^* \in S^*} f_{i^*} + \sum_{\substack{i^* \in S^* \\ j \in N_{S^*}(i^*)}} [c(i^*, j) - c(\sigma(j), j)] \geq 0 \ .$$

Now, observing that the double sum in the last inequality is going through all the cities, we replace it with a single sum and write the inequality as

$$\sum_{i^* \in S^*} f_{i^*} + \sum_{j \in J} [c(\sigma^*(j), j) - c(\sigma(j), j)] \geq 0 \ ,$$

which is the same thing as

$$C_f(S^*) + C_r(S^*) - C_r(S) \geq 0 \ ,$$

proving the lemma. ■

Lemma 10.3.3

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

Proof: We begin with some more notation. In the proof of the previous lemma we defined $\sigma(j)$ to be a function of cities. Now we extend it to facility locations. Given a solution S, a facility location i, we define:

– $\sigma^S(i) = \text{argmin}_{i' \in S}\, c(i, i')$. In words, the nearest facility in S to i.

– $\sigma^*(i) = \sigma^{S^*}(i)$.

The rest of the proof proceeds somewhat along the lines of the proof of the previous lemma. Therefore we omit the details when we employ the same type of arguments as before.

Consider swapping some facility i in the locally optimal solution S with $\sigma^*(i)$, the nearest facility in S^* to i. Clearly, this gives us a solution S' with an objective function value larger than or equal to that of S, because S is locally optimal. Note that if $\sigma^*(i)$ was already in S, there is no change.

With i swapped out and $\sigma^*(i)$ swapped in, we now consider the following. If we "route" the goods in S' so that each city that was in $N_S(i)$ now gets served by $\sigma^*(i)$, and not necessarily by the nearest open facility to it, then this would again yield a non-negative change in the objective function value of S'. As a result, the combined increase in the cost of this configuration, call it δ, would be

$$0 \leq \delta = f_{\sigma^*(i)} - f_i + \sum_{j \in N_s(i)} [c(j, \sigma^*(i)) - c(j, i)] \ .$$

Recalling that the function c is a metric, we observe the following inequality regarding the term inside the summation in the last equation:

$$c(j, \sigma^*(i)) - c(j, i) \quad \leq \quad c(i, \sigma^*(i)) \quad \leq \quad c(i, \sigma^*(j)) \quad \leq \quad c(i, j) + c(j, \sigma^*(j))$$

The first and last inequalities follow from the triangle inequality. The second inequality follows from the fact that in S^*, the distance between city i and its serving facility $\sigma^*(i)$ is no larger than that between i any other facility in S^*, in particular $\sigma^*(j)$. (Here we have used the σ^* function with both a city and a facility location as argument). We now have

$$0 \leq \delta \leq f_{\sigma^*(i)} - f_i + \sum_{j \in N_S(i)} [c(j, i) + c(j, \sigma^*(j))] \ .$$

Expanding the summation on the right, we first get $\sum_{j \in N_S(i)} c(j, i)$, which is the routing cost of the cities served by i in S; denote this by R_i. Second, we get $\sum_{j \in N_S(i)} c(j, \sigma^*(j))$, which is the routing cost of the cities served by i in S^*; denote this by R_i^*.

Now we can rewrite the last inequality as

$$0 \leq \delta \leq f_{\sigma^*(i)} - f_i + R_i + R_i^*$$

Summing this equation over all $i \in S$, we obtain

$$\sum_{i \in S} f_{\sigma^*(i)} - C_f(S) + C_r(S) + C_r(S^*) \geq 0 \ .$$

We are not quite done yet. The first sum is counting some facilities in S^* more than once and not counting some at all. We will finish this proof in the next lecture.

■

CS787: Advanced Algorithms

Scribe: Amanda Burton, Leah Kluegel
Lecturer: Shuchi Chawla
Topic: Facility Location ctd., Linear Programming
Date: October 8, 2007

Today we conclude the discussion of local search algorithms with by completeling the facility location problem started in the previous lecture. Afterwards, we discuss the technique of linear programming and its uses in solving NP-hard problems through integer programming and rounding.

11.1 Facility Location

Recall the problem definition -

Given: A set of locations for opening facilities, I, were each each facility $i \in I$ has an associated cost of opening, f_i. We are also given a set of cities, J, where each city $j \in J$ has an associated cost of going to a facility i, $c(i, j)$. The routing costs c form a metric.

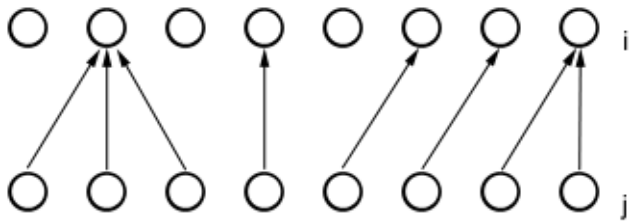


Figure 11.1.1: An example routing of cities to facilites.

Goal: Open some subset of facilities, S, and route every $j \in J$ to some facility $i \in S$ such that the total cost of opening the facilities plus the total cost of routing each of the cities to a facility is minimized. The total cost of any solution, $C(S)$, is its facility opening cost, $C_f(S)$, plus its routing cost, $C_R(S)$, $C(S) = C_f(S) + C_R(S)$.

11.1.1 Review

Last time we discussed a local search algorithm for solving this problem approximately. The local search algorithm started by initially assigning the locations in S to some arbitrary subset of the total set of locations I or started with S being the empty set. At every step, the algorithm can perform one of the following operations: adding a facility, removing a facility, or swapping one

facility for another. The algorithm performs one of these operations if doing so leads to a reduction of the total cost $C(S)$, and continues adding, subtracting, or swapping until it reaches some local optimum. Last lecture, we began to prove that the cost of any local optimum, given this kind of a local search algorithm, is within some small factor of the cost of any global optimum.

Theorem 11.1.1 *For any local optimum S and any global optimum S^*, $C(S) \leq 5 \cdot C(S^*)$.*

We made some partial progress towards proving this last time by bounding the routing cost of the set S. We picked some local optimum S, and argued that the routing cost of S is no more than the total cost of the global optimum. Our high-level proof was to use some local step and analyze the increase in cost for each local step taken. Since S is a local optimum, we know that for every step, the cost is going to increase. Rearranging the expression for the change in cost on each local step taken, we produced the following lemma.

Lemma 11.1.2 $C_R(S) \leq C(S^*)$

The local step used added a facility from the optimal solution $i^* \in S^*$ not present in the current solution S and routed some cities from facilities in S to this new facility. To conclude the proof, all that remains to be shown is that the facility opening cost of S is also small. Our second lemma stated that the facility opening cost of S is no more than twice the routing cost of S plus twice the total cost of S^*.

Lemma 11.1.3 $C_f(S) \leq 2 \cdot C_R(S) + 2 \cdot C(S^*)$

This together with the first lemma tells us that the facility opening cost of S is no more than four times the total cost of S^*. Together these Lemmas bound the total cost of S by five times the total cost of S^*.

During the previous lecture, we were able to prove Lemma 11.1.2 and had begun in the process of proving Lemma 11.1.3.

11.1.2 Notation

Recall that we introduced some new notation at the end of last class:

For some facility $i^* \in S^*$, we used $N_{S^*}(i^*)$ to denote the neighborhood of facility i^*. That is, $N_{S^*}(i^*)$ refered to all the $j \in J$ that are assigned to i^* in S^*.

For all $j \in N_{S^*}(i^*)$, $\sigma^*(j) = i^*$, where $\sigma^*(j)$ denotes the facility in the optimal solution to which j is assigned.

Likewise, for some facility $i \in S$, we used $N_S(i)$ to denote the neighborhood of facility i, where

$N_S(i)$ refers to all the $j \in J$ that are assigned to i in S.

For all $j \in N_S(i)$, $\sigma(j) = i$, where $\sigma(j)$ denotes the facility in the solutions to which j is assigned.

Furthermore, for any $i \in S$, $\sigma^*(i) = \arg\min i^* \in S^* c(i, i^*)$, i.e. the facility $i^* \in S^*$ that is closest to i.

The cost of all of the cities routed to $i \in S$, $R_i = \sum_{j \in N_S(i)} c(j, \sigma(j))$.

The routing cost, in the global solution, of all of the cities routed to $i \in S$, $R_i^* = \sum_{j \in N_S(i)} c(j, \sigma^*(j))$.

11.1.3 Completion of Facility Location

Using this notation we are able to prove the following claim:

Claim 11.1.4 *For any facility $i \in S$, $f_i \leq f_{\sigma^*(i)} + R_i + R_i^*$.*

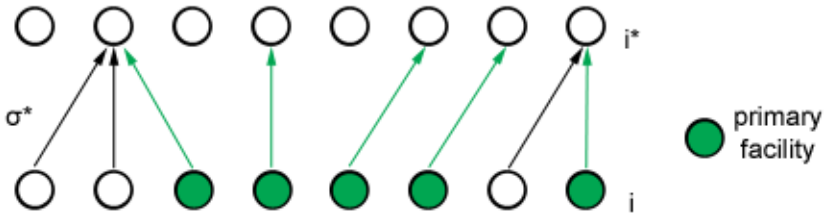


Figure 11.1.2: An example selection of primary facilities.

Claim 11.1.4 is useful, because when we compare it to Lemma 11.1.3, the lemma we are trying to prove, and sum over all the $i \in S$, we know that by definition $\sum_i R_i = C_R(S)$ and $\sum_i R_i^* = C_R(S^*)$.

The new inequality that we obtain from adding over all $i \in S$, contains the facility opening cost of S, the routing cost of S, the routing cost of S^*, and the term $\sum_i f_{\sigma^*(i)}$. This last term is problematic because this term over-counts some facilities in S^* and does not count others and is thus not equal to $C_f(S^*)$. To solve this problem, we are only going to use Claim 11.1.4 for some of the facilities in S. In particular, for every facility i^* we are going to identify at most one facility in S, called the primary facility. This facility in S should be the facility closest to i^*, so for $i^* \in S^*$, let $P(i^*) = \arg\min i \in S : \sigma^*(i) = i^* c(i, i^*)$.

By the definition of a primary facility, when we sum Claim 11.1.4 over the primary facilities, we are not over-counting any one facility cost in S^*. Thus we can use Claim 11.1.4 to account for the primary facilities opening cost. For the remaining facilities, called the secondary facilities, by using a slightly different algorithm we can bound the facility opening costs of the secondary facilities as well.

Claim 11.1.5 *For any secondary facility $i \in S$, $f_i \leq 2 \cdot (R_i + R_i^*)$.*

Once we prove Claim 11.1.5, the proof of Lemma 11.1.3 follows from that.

Proof: Lemma 11.1.3

Let P denote the primary facilities. Then $S - P$ is the set of secondary facilities.

$$\begin{aligned} C_f(S) &= \textstyle\sum_i f_i \\ &= \textstyle\sum_{i \in P} f_i + \sum_{i \in S-P} f_i \\ &\leq C_f(S^*) + 2 \cdot C_R(S) + 2 \cdot C_R(S^*) \\ &\leq 2 \cdot C_R(S) + 2 \cdot C(S^*) \end{aligned}$$

■

Proof: Claim 11.1.5

In order to get some kind of bound on the facility opening cost of S, we will consider some facility $i \in S$, perform some local step that involves this facility, and look at the increase in cost from that local step. Our local step is the removal of some facility i from S and the routing all of its associated cities to the primary facility to which i is assigned: $P(\sigma^*(i)) = i'$. Removing i will decrease the cost by f_i, but we will also incur a corresponding increase in routing cost, because for every city $j \in N_S(i)$ the routing cost increases to $c(j, i')$ from $c(j, i)$. So the increase in cost will be

$\Delta cost = -f_i + \sum_{j \in N_S(i)} (c(j, i') - c(j, i)) \geq 0$

We know that this cost is non-negative because we know that S is a local optimum. For some city j that was previously routed to i and is now routed to i', we are interested in the difference between the distance from j to i and the distance from j to i'.

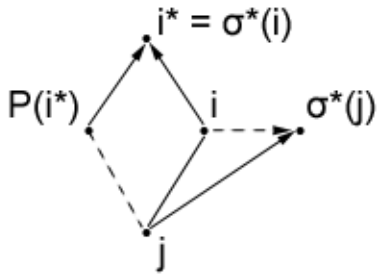


Figure 11.1.3: The relative distances between re-routed cities and facilities.

Using the Triangle Inequality, we can determine that,

$c(j, i') - c(j, i) \leq c(i, i')$

Again using the Triangle Inequality, we can further determine that,

$c(i, i') \leq c(i', i^*) + c(i^*, i)$

This bound is not quite convenient enough for us yet, as these values are not more meaningful to us than the original cost, so we must further bound this quantitiy. Since i' is a primary facility, we know that i' is closer to i^* than i is, i.e. $c(i', i^*) \leq c(i^*, i)$. Therefore,

$c(i', i^*) + c(i^*, i) \leq 2 \cdot c(i, i^*)$

This is still not quite convenient for us, because we want everything represented in terms of the routing cost of j.

Consider the facility $\sigma^*(j)$ that j got routed to in the optimal solution. We know that the closest optimal facility to i was i^*, therefore i^* is the facility in S^* closest to i and since $\sigma^*(j) \in S^*$, $c(i, i^*) \leq c(i, \sigma^*(j))$. Thus we have

$2 \cdot c(i, i^*) \leq 2 \cdot c(i, \sigma^*(j))$

Using the Triangle Inequality for the last time,

$2 \cdot c(i, \sigma^*(j)) \leq 2 \cdot (c(j, i) + c(j, \sigma^*(j)))$

But this is just the cost of routing j to i plus the cost of routing j to $\sigma^*(j)$. So we can write the total increase in cost as

$0 \leq \Delta cost \leq -f_i + 2 \cdot \sum_{j \in N_S(i)} (c(j, i) + c(j, \sigma^*(j)))$

Therefore,

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

Which implies

$f_i \leq 2 \cdot (R_i + R_i^*)$.

■

We have now proved Claim 11.1.5 which in turn proves Lemma 11.1.3 and Lemma 11.1.3 proves Theorem 11.1.1.

11.1.4 Wrap-up

The high-level idea of the proof is that, if you start from this locally optimal solution and if there is a globally optimal solution that is far better, it suggests a local improvement step that can be applied to improve the locally optimal solution. Given that such a local improvement step does not exist we know that the globally optimal solution cannot be far better than the locally optimal solution. How this is executed for one problem or another depends on the specifics of the problem, but we now have a grasp of the basic intuition that is going on.

This is not the best analysis for this particular algorithm. This algorithm is known to have a 3-approximation. In several places we didn't use the tightest possible bounds, and it is possible to tighten these bounds to get a 3-approximation. It is also possible to use other kinds of dynamics such as adding, removing, or swapping some d different facilities at a time. This also leads to an improved approximation factor. We might also have used different local steps to obtain a better

approximation factor. In the analysis that we did today we used all three of the local improvement steps to argue that the locally optimal solution is close to the global one. If we ended up not using one we could have gotten by without using it in our algorithm.

Facility location problems come in many different flavors. One particular flavor is picking out exactly k different locations while minimizing the sum of the routing costs. This has a similar local search algorithm: start with some subset of k locations and perform a series of swaps to get to a locally optimal solution. This algorithm is very general and works in a number of situations, but we can do much better than a 3-approximation for facility location using other techniques.

11.2 Linear Programming

11.2.1 Introduction

Linear Programming is one of the most widely used and general techniques for designing algorithms for NP-hard problems.

Definition 11.2.1 *A linear program is a collection of linear constraints on some real-valued variables with a linear objective function.*

Example:

Suppose we have two variables: x,y.
Maximize: $5 \cdot x + y$
Subject to : $2 \cdot x - y \leq 3$
$x - y \leq 2$
$x, y \geq 0$

To get an idea of what this looks like plot the set of pairs (x, y) that satisfy the constraints. See Figure 11.2.4 for a plot of the feasible region.

The feasible region, the region of all points that satisfy the constraints is the shaded region. Any point inside the polytope satisfies the constraints. Our goal is to find a pair that maximizes $5 \cdot x + y$.

The solution is the point $(x, y) = (7, 5)$.

■

Every point in the feasible region is a feasible point, i.e., it is a pair of values that satisfy all the constraints of the program. The feasible region is going to be a polytope which is an n-dimensional volume all faces of which are flat. The optimal solution, either a minimum or a maximum, always occurs at a corner of the polytope (feasible region). The extreme points (corners) are also called basic solutions.

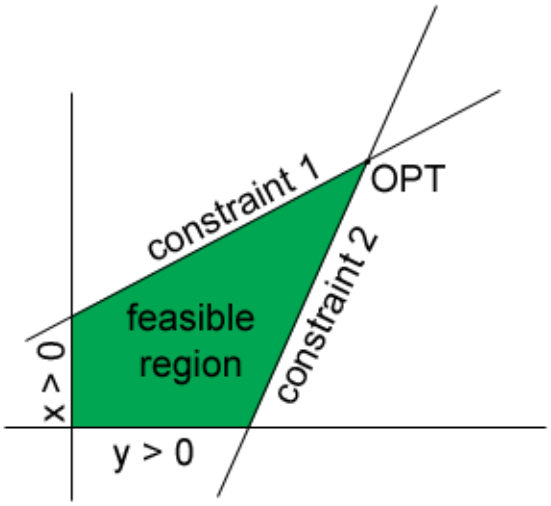


Figure 11.2.4: The feasible region in some linear programming problem.

11.2.2 Solving a Linear Program

Linear programs can be solved in polynomial time. If you have n variables and m constraints, then the linear program can be solved in time polynomial in n and m.

One way to solve a linear program is to:

1. Enumerate all extreme points of the polytope.
2. Check the values of the objective at each of the extreme points.
3. Pick the best extreme point.

This algorithm will find the optimal solution, but has a major flaw because there could be an exponential number of extreme points. If you have n variables, then the polytope is in some n-dimensional space. For example, say the polytope is an n-dimensional hypercube, so all of the variables have constraints such that $0 \leq x_i \leq 1$. There are 2^n extreme points of the hypercube, so we can't efficiently enumerate all the possible extreme points of the polytope and then check the function values on each of them.

What is typically done to find the solution is the Simplex Method. The Simplex Method starts from some extreme point of the polytope, follows the edges of the polytope from one extreme point to another doing hill climbing of some sort and then stops when it reaches a local optimum. The "average case complexity" of this algorithm is polynomial-time, but on the worst case it is exponential. There are other algorithms that are polynomial-time that follow points in the polytope until you find an optimal solution. The benefit of Linear Programming is that once you write down the constraints, there exists some polynomial-time algorithm that is going to solve the problem.

Linear Programming is interesting to us because 1. it is solvable in polynomial time and 2. a closely related problem (Integer Programming) is NP-hard. So we can take any NP-complete problem, reduce it to an instance of Integer Programming, and then try to solve the new problem using the same techniques that solve Linear Programming problems.

11.2.3 Integer Programming

Definition 11.2.2 *An Integer Program is a Linear Program in which all variables must be integers.*

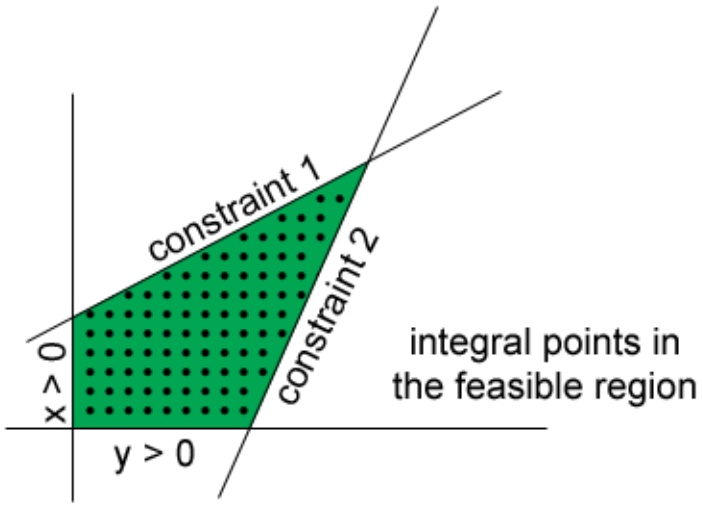


Figure 11.2.5: The feasible region of an Integer Programming problem.

In an Integer Program you are trying to optimize some linear function over some set of linear constraints with the additional constraint that all the variables must be integers. If we were to solve Example 11.2.1 subject to the constraint that x and y are integers, we would get the same optimal solution. This is not generally the case. In general, if you plot the feasible set of an Integer Program, you get some set of points defined by some constraints, and our goal is to optimize over the set of integer points inside the feasible region. A Linear Program will optimize over the larger set which contains the integer points and the real-valued points inside the feasible region. In general, the optimum of the Integer Program will be different from the optimum of the corresponding Linear Program. With an Integer Program, the list of constraint again form a polytope in some n-dimensional space, but the additional constraint is that you are only optimizing over the integer points inside this polytope. Note: The integer points in the feasible region do not necessarily lie on the boundaries and the extreme points, in particular, are not necessarily integer points. A Linear Programming algorithm is going to find an extreme point as a solution to the problem. This is not necessarily the same as the optimal solution to the Integer Program, but hopefully it is close enough.

11.2.4 Solving NP-hard Problems

In general, the way we use Linear Programming is to

1. reduce an NP-hard optimization problem to an Integer Program
2. relax the Integer Program to a Linear Program by dropping the integrality constraint
3. find the optimal fractional solution to the Linear Program
4. round the optimal fractional solution to an integral solution

Note: In general the optimal fractional solution to the Linear Program will not be an integral solution. Thus the final integral solution is not necessarily an optimal solution, but it came from an optimal fractional solution, so we hope it is close to an optimal integral solution.

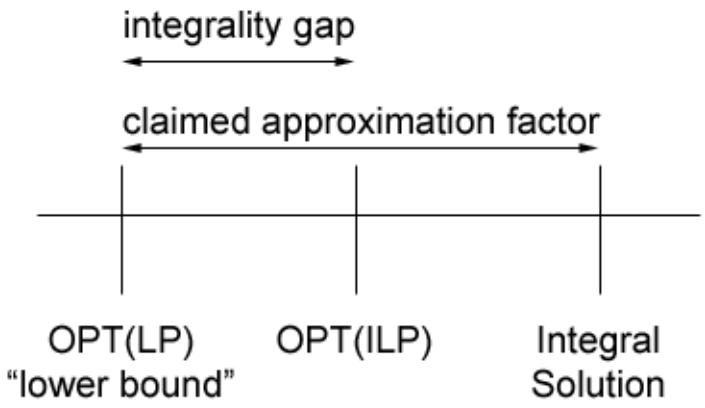


Figure 11.2.6: The range of solutions for Linear vs. Integer Programming.

In terms of the values of the solutions, the optimal value of the Integer Program which is equal to the solution of the NP-hard problem falls in between the optimal solution to the Linear Program and the integral solution determined from rounding the optimal solution to the Linear Program. See Figure 11.2.6 for an example of a minimization problem. The gap between the optimal solution to the Integer Program and the Linear Program is known as the integrality gap. If this gap is large, then we can expect a large approximation factor. If this gap is small, then the approximation factor is likely to be small. The integrality gap characterizes if we relax integrality, what is the improvement in the optimal solution. If we are using this particular algorithm for solving an NP-hard problem we cannot prove an approximation factor that is better than the integrality gap. The approximation factor is the difference between the optimal solution to the Linear Program and the integral solution that is determined from that optimal fractional solution. When we do the transformation from the NP-hard problem to an Integer Program our goal is to come up with some

Integer Program that has a low integrality gap and the right way to do the rounding step from the fractional solution to the integral solution. Note: Finding a fractional optimal solution to a Linear Program is something that is known (and can be done in polynomial-time), so we can take it as given.

The good thing about Linear Programs is that they give us a lower bound to any NP-hard problem. If we take the NP-hard problem and reduce it to an Integer Program, relax the Integer Program to a Linear Program and then solve the Linear Program, the optimal solution to the Linear Program is a lower bound to the optimal solution for the Integer Program (and thus the NP-hard problem). This is a very general way to come up with lower bounds. Coming up with a good lower bound is the most important step to coming up with a good approximation algorithm for the problem. So this is a very important technique. Most of the work involved in this technique is coming up with a good reduction to Integer Programming and a good rounding technique to approximate the optimal fractional solution.

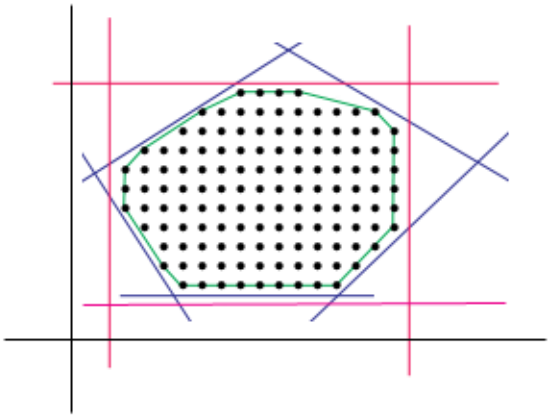


Figure 11.2.7: The possible constraints on an ILP problem.

Note: There are multiple ways to enclose a given set of integer solutions using linear constraints. See Figure 11.2.7. The smallest that encloses all feasible integer solutions without too many constraints and without enclosing any more integer points is best. It is an art to come up with the right set of linear constraints that give a low integrality gap and a small number of constraints.

11.2.5 Vertex Cover Revisited

We have already seen a factor of 2 approximation using maximum matchings for the lower bound. Today we will see another factor of 2 approximation based on Linear Programming. There is yet another factor of 2 approximation that we will see in a few lectures. For some problems many techniques work.

Given: $G = (V, E)$ with weights $w : V \to \mathbb{R}^+$

Goal: Find the minimum cost subset of vertices such that every edge is incident on some vertex in that subset.

1. Reducing Vertex Cover to an Integer Program

Let the variables be:

x_v for $v \in V$ where $x_v = 1$ if $v \in VC$ and $x_v = 0$ otherwise.

Let the constraints be:

For all $(u, v) \in E$, $x_u + x_v \geq 1$ (each edge has at least one vertex)

For all $v \in V$, $x_v \in \{0, 1\}$ (each vertex is either in the vertx cover or not)

We want to minimize $\sum_{v \in V} w_v \cdot x_v$ (the total weight of the cover)

Note that all the constraints except the integrality constraints are linear and the objective function is also linear because the w_v are constants given to us. Thus this is an Integer Linear Program.

Note: This proves that Integer Programming is NP-hard because Vertex Cover is NP-hard and we reduced Vertex Cover to Integer Programming.

2. Relax the Integer Program to a Linear Program

Now relax the integrality constraint $x_v \in \{0, 1\}$ to $x_v \in [0, 1]$. to obtain a Linear Program.

3. Find the optimal fractional solution to the Linear Program

Say x^* is the optimal fractional solution to the Linear Program.

Lemma 11.2.3 *$Val(x^*) \leq OPT$ where OPT is the optimal solution to the Integer Program.*

Example:

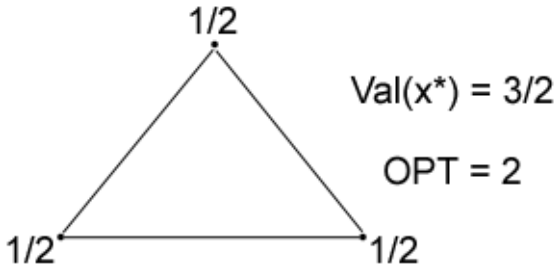


Figure 11.2.8: A possible LP soltion to the vertex cover problem.

Consider a graph that is just a triangle with $w_v = 1$ for all $v \in V$. See Figure 11.2.8. Our Linear Program is looking for fractional values at each of the vertices such that the fractional values for the vertices on each edge sum up to at least one and the sum of all of the fractional values times their vertex weight is minimized.

One fractional solution is to assign $x_v^* = \frac{1}{2}$ for all $v \in V$. Note: This fractional solution is also optimal. In this case, $Val(x^*) = \frac{3}{2}$.

The optimal solution to vertex cover is going to have to include at least two vertices. So $OPT = 2$. We can see that the Linear Program is doing better than the Integer Program.

■

4. Round the optimal fractional solution to an integral solution.

Given an optimal fractional solution, we clearly want to include all $x_v^* = 1$ in the vertex cover and we don't want to include any $x_v^* = 0$ in the vertex cover. Further we can obtain an integral solution by picking all vertices that are at least $\frac{1}{2}$. This covers all edges because for every edge $e = (u, v)$ $x_u^* + x_v^* \geq 1$ which implies that either x_u^* or x_v^* is at least $\frac{1}{2}$ and so will be picked for the vertex cover.

Algorithm: Return all vertices v with $x_v^* \geq \frac{1}{2}$.

Theorem 11.2.4 *This algorithm is a 2-approximation.*

Proof: Theorem 11.2.4 $\tilde{x_v} = 1$ if the algorithm picks v and $\tilde{x_v} = 0$ otherwise. Let x^* be the optimal fractional solution to the Linear Program. This implies that $\tilde{x_v} \leq 2 \cdot x_v$ for all $v \in V$ because some subset of the vertices was doubled and the rest were set to 0.

The algorithm's cost is $\sum_{v \in V} w \cdot \tilde{x_v}$
$$\begin{aligned} &\leq 2 \cdot \sum_{v \in V} w_v \cdot x_v^* \\ &= 2 \cdot cost(LinearProgram) \\ &\leq 2 \cdot OPT. \end{aligned}$$

■

This proves that this is a 2-approximation, so the integrality gap is at most 2. In Example 11.2.5 the integrality gap is at least $\frac{4}{3}$ which is less than 2. For Vertex Cover with this Linear Program, the integrality gap can be arbitrarily close to 2. As an exercise, the reader should come up with an instance of vertex cover where the integrality gap is indeed very close to 2.

11.3 Next Time

Next time we will see some more examples of Linear Programming and in particular how to do the rounding step.

CS787: Advanced Algorithms

Scribe: Archit Gupta, Pavan Kuppili **Lecturer:** Shuchi Chawla
Topic: Randomized rounding, concentration bounds **Date:** 10/10/2007

12.1 Set Cover

Let $E = \{e_1, e_2, .., e_n\}$ be a set of n elements. Let $S_1, S_2, .., S_m$ be subsets of E with associated costs $c_1, .., c_m$. As explained in a previous lecture, the problem of set cover is to choose 1 or more of the m subsets such that the union of them cover every element in E, and the objective is to find the set cover with the minimum cost. In a previous lecture, we have seen a greedy solution to the set cover which gave a log n approximation. In this lecture, we will see a linear programming solution with randomized rounding to achieve an $O(logn)$ approximation.

12.1.1 ILP formulation of set cover

Let X_i be a random variable associated with each subset S_i.
$X_i = 1$, if S_i is in the solution, and 0 otherwise.

The ILP formulation:
Minimize $\sum_{i=1}^{m} c_i X_i$ s.t
$\forall e \in E, \sum_{i:e \in S_i} X_i \geq 1$ - (1)
$\forall X_i, X_i \in \{0, 1\}$ - (2)

The constraints in (1) ensure that every element is present in atleast one of the chosen subsets. The constraints in (2) just mean that every subset is either chosen or not. The objective function chooses a feasible solution with the minimum cost.

It can be noted that this problem formulation is a generalization of the vertex cover. If we map every edge to an element in E, and every vertex to one of the the subsets of E, the vertex cover is the same as the set cover problem with the additional constraint that every element e appears in exactly 2 subsets (corresponding to the 2 vertices it is incident on).

12.1.2 Relaxing to an LP problem

Instead of $X_i \in \{0, 1\}$, relax the constraint so that X_i is in the range [0, 1]. The LP problem can be solved optimally in polynomial time. Now we need to round the solution back to an ILP solution.

12.1.3 Deterministic rounding

First, let us see the approximation we get by using a deterministic rounding scheme analogous to the one we used in the vertex cover problem.

Theorem 12.1.1 *We have a deterministic rounding scheme which gives us an F-approximation*

to the solution, where F is the maximum frequency of an element.

Proof: Let F be the maximum frequency of an element, or the maximum number of subsets an element appears in.
Pick a threshold t = $\frac{1}{F}$.
If $x_1, .., x_m$ is the solution to the LP, the ILP solution is all subsets for which $x_i \geq \frac{1}{F}$.

This results in a feasible solution because in any of the constraints in (1) we have atmost F variables on the LHS, and atleast one of them should be $\geq \frac{1}{F}$, so we will set atleast one of those variables to 1 during the rounding. So, every element will be covered. It can be noted here that the vertex cover has F = 2, and hence we used a threshold of $\frac{1}{2}$ during the previous lecture. If the values of $X_1, .., X_m$ are $x_1^{'}, .., x_m^{'}$ after rounding, $\forall i, x_i^{'} \leq Fx_i$ [since if $x_i \geq \frac{1}{F}$ we round it to 1, and 0 otherwise]
So the cost of the ILP solution $\leq$ F(cost of the LP solution). So, this deterministic rounding gives us an F approximation to the optimal solution. ■

12.1.4 Randomized rounding

Algorithm:
Step 1. Solve the LP.
Step 2 (Randomized rounding). $\forall$ i, pick S_i independently with probability x_i (where x_i is the value of X_i in the LP solution).
Step 3. Repeat step 2 until all elements are covered.

The intution behind the algorithm is that higher the value of x_i in the LP solution, the higher the probability of S_i being picked during the random rounding.

Theorem 12.1.2 *With a probability* $(1 - \frac{1}{n})$*, we get a 2logn approximation.*

Proof:

Lemma 12.1.3 *The expected cost in each iteration of step 2 is the cost of the LP solution.*

Proof: Let Y_i be a random variable.
$Y_i = c_i$ if S_i is picked, and 0 otherwise.
Let Y = $\sum_{i=1}^{n} Y_i$,
E[Y] = $\sum_{i=1}^{n} E[Y_i]$ [by linearity of expectation]
= $\sum_{i=1}^{n} c_i x_i$
= cost of the LP solution. ■
This is a nice result as the expected cost of the ILP solution is exactly equal to the cost of the LP solution.

Lemma 12.1.4 *The number of iterations of step 2 is 2logn with a high probability (whp).*

Proof: Fix some element e ∈ E.
Pr[e is not covered in any one execution of step] = $\prod_{i:e\in S_i}$**Pr**[S_i is not picked] (since the subsets are picked independently)
= $\prod_{i:e\in S_i}(1-x_i)$
≤ $\prod_{i:e\in S_i} e^{-x_i}$ (using the fact that if $x \in [0,1], (1-x) \leq e^{-x}$)
= $e^{-\sum_{i:e\in S_i} x_i}$
≤ $\frac{1}{e}$ (since the LP solution satisfies constraint (1) which means $\sum_{i:e\in S_i} x_i \geq 1$)

If we execute step 2 (2logn) times,
Pr[element e is still uncovered] ≤ $\frac{1}{e^{2logn}}$ (since each execution is independent)
= $\frac{1}{n^2}$.
By the union bound, **Pr**[∃ an uncovered element after 2logn executions] ≤ $\frac{1}{n}$
Hence with a high probability, the algorithm will terminate in 2log n iterations of step 2. ■

The total expected cost is just the expected cost of each iteration multiplied by the number of iterations.
So E[cost] = (2log n).Cost(LP solution) with a probability $(1-\frac{1}{n})$ (using Lemmas 12.1.3 and 12.1.4).
■

12.2 Concentration Bounds

In analyzing randomized techniques for rounding LP solutions, it is useful to determine how close to its expectation a random variable is going to turn out to be. This can be done using concentration bounds: We look at the probability that given a certain random variable X, the probability that X lies in a particular range of values (Say, the deviation from the expectation value). For instance, we want **Pr**$[X \geq \lambda]$ for some value of λ. Note that if a random variable has small variance, or (as is often the case with randomized rounding algorithms) is a sum of many independent random variables, then we can give good bounds on the probability that it is much larger or much smaller than its mean.

12.2.1 Markov's Inequality

Given a random variable $X \geq 0$, we have,

$$\mathbf{Pr}[X \geq \lambda] \leq \frac{\mathbf{E}[X]}{\lambda} \tag{12.2.1}$$

Also, given some $f : X \to \Re^+ \cup \{0\}$,

$$\mathbf{Pr}[f(X) \geq f(\lambda)] \leq \frac{\mathbf{E}[f(X)]}{f(\lambda)} \tag{12.2.2}$$

If the above function f is monotonically increasing then in addition to (12.2.2), the following also

holds:

$$\mathbf{Pr}[X \geq \lambda] = \mathbf{Pr}[f(X) \geq f(\lambda)] \leq \frac{\mathbf{E}[f(X)]}{f(\lambda)} \tag{12.2.3}$$

12.2.2 Chebyshev's Inequality

We now study a tighter bound called the Chebyshev's bound.
Let $f(X) = (X - \mathbf{E}[X])^2$. Note that f is an increasing function if $X > \mathbf{E}[X]$.
We have,

$$\mathbf{Pr}[|X - \mathbf{E}[X]| \leq \lambda] = \mathbf{Pr}\big[(X - \mathbf{E}[X])^2 \leq \lambda^2\big] \tag{12.2.4}$$

$$= \frac{\mathbf{E}\big[(X - \mathbf{E}[X])^2\big]}{\lambda^2} = \frac{\sigma^2(X)}{\lambda^2} \tag{12.2.5}$$

That is, the deviation of X from $\mathbf{E}[X]$ is a function of its variance($\sigma(X)$). If the variance is small, then we have a tight bound.

Also not that with probability p, $X \in \mathbf{E}[X] \pm \sqrt{\frac{1}{p}}\sigma(X)$

12.2.3 Chernoff's bound

We present Chernoff's bound which is tighter compared to the Chebyshev and Markov bounds.

Let random variables X_1, X_2, ..., X_n be independent and identically distributed random variables, where $X_i \in [0, 1]$. Note that the class of indicator random variables lie in this category.

Let $X = \sum_{i=1}^{n} X_i$. Also $\mathbf{E}[X] = \sum_{i=1}^{n} \mathbf{E}[X_i] = \mu$

For any $\delta > 0$,

$$\mathbf{Pr}[X \geq (1+\delta)\mu] \leq \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu} \tag{12.2.6}$$

It can also be shown for $\delta \geq 0$,

$$\mathbf{Pr}[X \geq (1+\delta)\mu] \leq e^{-\left(\frac{\delta^2}{2+\delta}\right)\mu} \tag{12.2.7}$$

Note that the above probability resembles a gaussian/bell curve. When $0 \leq \delta \leq 1$

$$\mathbf{Pr}[X \geq (1+\delta)\mu] \leq e^{-\frac{\delta^2}{3}\mu} \tag{12.2.8}$$

$$\mathbf{Pr}[X \leq (1-\delta)\mu] \leq e^{-\frac{\delta^2}{2}\mu} \tag{12.2.9}$$

Example: Coin tossing to show utility of bounds. We toss n independent unbiased coins. We want to find a t such that: $\mathbf{Pr}[\text{No. of times we get heads in } n \text{ tosses} \geq t] \leq 1/n$.

We use indicator random variables to solve this: If the ith coin toss is heads, let $X_i = 1$, otherwise $X_i = 0$. Let $X = \sum_{i=1}^{n} X_i$. Here X is the total number of heads we get in n independent tosses. Note that $\mu = \mathbf{E}[X] = \sum_{i=1}^{n} \mathbf{E}[X_i] = n/2$, since $\mathbf{E}[X_i] = 1/2$ (The probability of getting a heads).

To get bounds on $\mathbf{Pr}[X \geq t]$, we first apply Markov's inequality:

$$\mathbf{Pr}[X \geq t] \leq \frac{\mathbf{E}[X]}{t} = \frac{n/2}{t}$$

$$\mathbf{Pr}[X \geq t] = 1/n \Rightarrow \frac{n/2}{t} = 1/n \Rightarrow t = n^2/2$$

But, we don't do any more than n coin tosses, so this bound is not useful. Note that Markov's bound is weak.

Applying Chebyshev's inequality:

$$\mathbf{Pr}[X \geq t] \leq \mathbf{Pr}[|X - \mu| \geq t - \mu] \leq \frac{\sigma^2(X)}{(t-\mu)^2}$$

Evaluating $\sigma(X)$ where $X_i \in \{0, 1\}$,

$$\mathbf{E}[X_i] = 1/2$$

$$\sigma^2(X_i) = \mathbf{E}\big[(X_i - \mathbf{E}[X_i])^2\big] = \frac{1}{2}(\frac{1}{4}) + \frac{1}{2}(\frac{1}{4}) = 1/4$$

Since this is a sum of independent random variables, the variances can be summed together:

$$\sigma^2(X) = \sum_{i=1}^{n} \sigma^2(X_i) = n/4$$

Evaluating t,

$$\mathbf{Pr}[X \geq t] \leq \frac{\sigma^2(X)}{(t-\mu)^2} = 1/n$$

$$\Rightarrow t = \mu + n/2 = n$$

Again this bound is quite weak.

Applying Chernoff's bound:

$$\mathbf{Pr}[X \geq (1+\delta)\mu] \leq e^{-\frac{\delta^2}{3}\mu} = e^{-\frac{\delta^2}{6}n}$$

Taking,

$$e^{-\frac{\delta^2}{6}n} = 1/n$$

we get,

$$\delta = \sqrt{\frac{6logn}{n}}$$

$$\Rightarrow t = (1+\delta)\mu = n/2 + \sqrt{\frac{3nlogn}{2}}$$

This bound is much nicer than what we obtained earlier, showing us this gradual increase in tightness of the bounds across Markov, Chebyshev and Chernoff. ■

CS787: Advanced Algorithms

Scribe: Archit Gupta, Pavan Kuppili **Lecturer:** Shuchi Chawla
Topic: Randomized rounding (contd.), LP duality. **Date:** 10/15/2007

13.1 Routing to minimize congestion

We are given a graph $G = (V, E)$ and k "commodities". Each commodity i has a source s_i and a destination t_i. The goal is to find an (s_i, t_i) path in the graph G for every commodity i, while minimizing the maximum congestion along any edge. The latter can be written formally as
Minimize congestion $C = max_e|\{i : P_i \ni e\}|$, where P_i is the (s_i, t_i) path of commodity i and e is an edge.

It can be noted that this problem is similar to a network flow problem. But it is not a network flow problem because for each commodity i, we need to have a single path between s_i and t_i. We cannot have the flow splitting along multiple branches.

This problem is an NP-hard problem. So we will solve it by formulating it as an ILP problem, relaxing it to an LP problem, solving the LP, and rounding the solution to an ILP solution.

13.1.1 ILP formulation with exponential number of variables and constraints

Let $i = 1, \ldots, k$ be the k commodities.
Let P_i be the set of all paths from s_i to t_i.
We have a variable x_P for every $P \in P_i$, $\forall i$

Minimze t s.t

$$\sum_{P \in P_i} x_P = 1, \forall i \tag{13.1.1}$$

$$\sum_i \sum_{P \in P_i, e \in P} x_P \leq t, \forall e \in E \tag{13.1.2}$$

$$t \geq 0 \tag{13.1.3}$$

$$x_P \in \{0, 1\} \forall P \tag{13.1.4}$$

Note that the actual objective function which is in a min-max form is not linear. So, we employ a trick of introducing a new variable t. We introduce the constraint (13.1.2) that congestion on any edge $\leq$ t, and so all we need to do now is minimze t, which is equivalent to minimizing the maximum congestion. The constaint 13.1.1 makes sure that we select exactly one path for each commodity.

The problem with this ILP formulation is that we can have an exponential number of paths between s_i and t_i (for example, in a completely connected graph). So even if we relax it to an LP problem,

we will still have an exponential number of variables and constraints. We would like a formulation with a polynomial (polynominal in $|V|$, $|E|$, and k) number of variables and constraints. Next we consider an alternative ILP formulation.

13.1.2 ILP formulation with polynomial number of variables and constraints

Rather than looking at all the paths, we look at an edge granularity and see if a commodity is routed along any edge.
We have variables $x_{e,i}$, $\forall e \in E, i = 1, \ldots, k$
$x_{e,i} = 1$, if $e \in P_i$, and 0 otherwise. [P_i is the chosen (s_i, t_i) path of commodity i]
The ILP formulation is

Minimize t s.t

$$\sum_{e \in \delta^+(t_i)} x_{e,i} = \sum_{e \in \delta^-(s_i)} x_{e,i} = 1 \forall i \tag{13.1.5}$$

$$\forall i, \forall v \neq s_i, t_i \sum_{e \in \delta^+(v)} x_{e,i} = \sum_{e \in \delta^-(v)} x_{e,i} \tag{13.1.6}$$

$$\sum_i x_{e,i} \leq t \forall e \in E \tag{13.1.7}$$

$$t \geq 0 \tag{13.1.8}$$

$$x_{e,i} \in \{0, 1\} \tag{13.1.9}$$

In the above formulation, δ^+ indicates the flow coming into a vertex, and δ^- indicates the flow going out of a vertex. Constraint 13.1.5 makes sure that we have a unit flow being routed out of every s_i, and a unit flow being routed into every t_i. Constraint 13.1.6 is like flow conservation at every other vertex. Congestion along an edge is just the number of commodities being routed along that edge, and constraint 13.1.7 ensures that $t \geq$ congestion along any edge. The objective function is to minimze t.

The problem can be relaxed to an LP problem by letting $x_{e,i} \in [0, 1]$. It can be noted that this formulation (let's call it edge formulation) is equivalent to the formulation in Section 13.1.1 (let's call it path formulation). From the LP solution to this edge formulation, one can obtain the LP solution to the path formulation. (This is left as an exercise to the reader). Now, we have a fractional solution $\{x_P\}$ for each variable x_P. We also know that the solution to the LP $t \leq C^*$, where C^* is the optimal solution to the ILP. Now, we need to round the LP solution to an ILP solution such that we achieve a reasonable approximation to t, the LP solution.

13.1.3 Deterministic rounding

First, we can note that a deterministic rounding doesn't help us get a good approximation. For example, let us look at a deterministic strategy. A reasonable strategy would be to look at all $P \in P_i$, and round the one with highest value to 1, and others to 0. Say if P_i has n paths, it might

be the case that in the LP solution, all the n paths have a weight of $\frac{1}{n}$. So while rounding, the value of a path can increase by a factor of n. Now the optimal congestion t on an edge is caused by some paths, and if the value of each of these paths increase by a factor of n during the rounding, we can only get an n factor approximation. Since n can be an exponential number, this is not a very useful approximation factor. Likewise it can be seen that other deterministic rounding strategies do not help in getting a good approximation factor.

13.1.4 Randomized rounding

For every commodity i, consider the probability distribution on P_i given by $\{x_P\}_{P \in P_i}$, and pick one of the P_i's according to this probability distribution. For example, say we have three paths with values 0.5, 0.3, and 0.2. Get a random number between 0 and 1. If the number is between 0 and 0.5, pick the first path. If the number is between 0.5 and 0.8, pick the second path, and if the number is between 0.8 and 1, pick the third path.

For any edge e, $\mathbf{Pr}[\text{commodity } i \text{ is routed along } e] = x_{e,i}$.
Let X_e be a random variable , and let X_e = number of commodities i with $P_i \ni e$. In other words, X_e is a random variable which indicates the level of congestion along an edge e. To get the $\mathbf{E}[X_e]$, we can use indicator random varibales.
Let $X_{e,i} = 1$, if $P_i \ni e$, and 0 otherwise.
$X_e = \sum_i X_{e,i}$
$\Rightarrow \mathbf{E}[X_e] = \sum_i \mathbf{E}[X_{e,i}]$ [By linearity of expectation]
$= \sum_i x_{e,i} \leq t$
This means that the expected value of congestion along any edge $\leq t$, the solution to the LP problem. Now if we can show that for any edge, $\mathbf{Pr}[X_e \geq \lambda t] \leq \frac{1}{n^3}$, the by the union bound, $\mathbf{Pr}[\exists$ edge e s.t $X_e \geq \lambda t] \leq \frac{|E|}{n^3} \leq \frac{1}{n}$, and we can get a λ approximation with a high probability.

$$X_e = \sum_i X_{e,i}$$

$$\mathbf{E}[X_e] = \mu \leq t$$

$$\mathbf{Pr}[X_e > \lambda t] \leq \mathbf{Pr}[X_e > \lambda \mu]$$

Using Chernoff's bounds:

$$\mathbf{Pr}[X_e > \lambda t] \leq \left(\frac{e^{\lambda - 1}}{\lambda^\lambda}\right)^\mu$$

$$\approx \lambda^{-\lambda\mu}$$

$$\approx \lambda^{-\lambda}$$

(Taking $\mu \approx 1$, atleast edge picked once in expectation.) Now,

$$\lambda^\lambda = n^3 \Rightarrow \lambda = O\left(\frac{logn}{loglogn}\right)$$

. We have with high probability, λ approximation.

13.2 LP Duality

The motivation behind using an LP dual is they provide lower bounds on LP solutions. For instance, consider the following LP problem.
Minimize $7x + 3y$, such that

$$x + y \geq 2$$
$$3x + y \geq 4$$
$$x, y \geq 0$$

Say, by inspection, we get a solution $x = 1, y = 1$, with an objective function value of 10. We can show that this is the optimal solution in the following way.
Proof: If we multiply the constraints with some values and add them such that the coeffiecients of x and y are < 7 and 3, respectively, we can get a lower bound on the solution. For instance, if we add the two constraints, we get $4x + 2y \geq 6$, and since $x, y \geq 0$, we have $7x + 3y > 4x + 2y \geq 6$. So, 6 is a lower bound to the optimal solution. Likewise, if we multiply the first constraint with 1, and the second constraint with 2, we have

$$7x + 3y = (x + y) + 2(3x + y) \geq 2 + 2(4) = 10$$

Hence, 10 is a lower bound on the solution, and so $(x = 1, y = 1)$ with an objective function value of 10 is an optimal solution. ■
Similar to the above example, LP duality is a general way of obtaining lower bounds on the LP solution. The idea is we multiply each constraint with a multiplier and add them, such that the sum of coefficients of any variable is $\leq$ the coefficient of the variable in the objective function. This gives us a lower bound on the LP solution. We want to choose the multipliers such that the lower bound is maximized (giving us the tightest possible lower bound).

13.2.1 Converting a primal problem to the dual form

To convert the primimal to the dual, we do the following.
1. For each constraint in the primal, we have a corresponding variable in the dual. (this variable is like the multiplier).
2. For each variable in the primal, we have a corresponding constraint in the dual. These constraints say that when we multiply the primal constraints with the dual variables and add them, the sum of coefficients of any primal variable should be less than or equal to the coefficient of the variable in the primal objective function.
3. The dual objective function is to maximize the sum of products of right hand sides of primal constraints and the corresponding dual variables. (This is maximizing the lower bound on the primal LP solutions).
The following is a more concrete example showing the primal to dual conversion.
A linear programming problem is of the form:

Minimize $\sum_i c_i x_i$, such that,

$$\sum_i A_{ij} x_i \geq b_j \forall j$$

$$x_i \geq 0 \forall i$$

We call this the primal LP. This LP can be expressed in the matrix form as:

Minimize $c^T x$, such that,

$$Ax \geq b$$

$$x \geq 0$$

The corresponding dual problem is: Maximize $\sum_j b_j y_j$, such that,

$$\sum_j A_{ij} y_j \leq c_i \forall i$$

$$y_j \geq 0 \forall j$$

Expressed in matrix form, the dual problem is, Maximize $b^T y$, such that

$$A^T y \leq c$$

$$y \geq 0$$

Note that the dual of a dual LP is the original primal LP.

Theorem 13.2.1 Weak LP duality theorem *If x is any primal feasible solution and y is any dual feasible solution, then $Val_P(x) \geq Val_D(y)$*

Proof:

$$\begin{aligned}\sum_i c_i x_i &\geq \sum_i \left(\sum_j A_{ij} y_j \right) x_i \\ &= \sum_j \left(\sum_i A_{ij} x_i \right) y_j \\ &\geq \sum_j b_j y_j\end{aligned}$$

■

So, the weak duality thoerem says that any dual feasible solution is a lower bound to the primal optimal solution. This is a particularly nice result in the context of approximation algorithms. In the previous lectures, we were solving the primal LP exactly and using the LP solution as a lower bound to the optimal ILP solution. By using the weak duality theorem, instead of solving the LP exactly to obtain a lower bound on the optimal value of a problem, we can (more easily) use any dual feasible solution to obtain a lower bound.

Theorem 13.2.2 Strong LP duality theorem *If the primal has an optimal solution x^* and the dual has an optimal solution y^*, then $c^T x^* = b^T y^*$, i.e, the primal and the dual have the same optimal objective function value.*

In general, if the primal is infeasible (there is no feasible point which satisfies all the constraints), the dual is unbounded (the optimal objective function value is unbounded). Similarly, if the dual is infeasible, the primal is unbounded. However, if both the primal and dual are feasible (have atleast one feasible point), the strong LP duality theorem says that the optimal solutions to the primal and the dual have the exact same objective function value.

CS787: Advanced Algorithms

Scribe: Amanda Burton, Leah Kluegel
Lecturer: Shuchi Chawla
Topic: Primal-Dual Algorithms
Date: 10-17-07

14.1 Last Time

We finished our discussion of randomized rounding and began talking about LP Duality.

14.2 Constructing a Dual

Suppose we have the following primal LP.

$$\min \ \sum_i c_i x_i \ \ s.t.$$

$$\sum_i A_{ij} x_i \geq b_j \quad \forall j = 1, ..., m$$

$$x_i \geq 0$$

In this LP we are trying to minimize the cost function subject to some constraints. In considering this canonical LP for a minimization problem, lets look at the following related LP.

$$\max \ \sum_j b_j y_j \ \ s.t.$$

$$\sum_j A_{ij} y_j \leq c_i \quad \forall i = 1, ..., n$$

$$y_j \geq 0$$

Here we have a variable y_j for every constraint in the primal LP. The objective function is a linear combination of the b_j multiplied by the y_j. To get the constraints of the new LP, if we multiply each of the constraints of the primal LP by the multiplier y_j, then the coefficients of every x_i must sum up to no more than c_i. In this way we can construct a dual LP from a primal LP.

14.3 LP Duality Theorems

Duality gives us two important theorems to use in solving LPs.

Theorem 14.3.1 ***(Weak LP Duality Theorem)*** *If x is any feasible solution to the primal and y is any feasible solution to the dual, then $Val_P(x) \geq Val_D(y)$.*

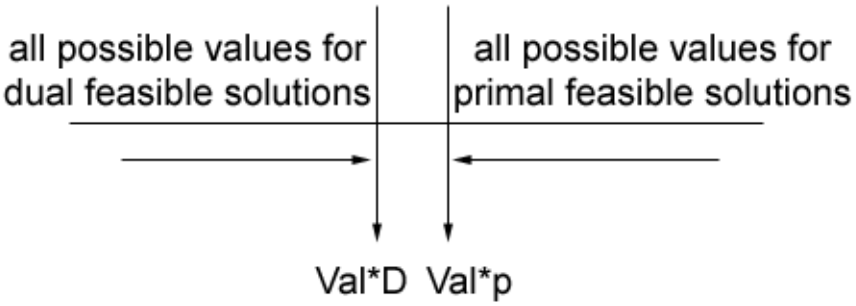


Figure 14.3.1: Primal vs. Dual in the Weak Duality Theorem.

On the number line we see that all possible values for dual feasible solutions lie to the left of all possible values for primal feasible solutions.

Last time we saw an example in which the optimal value for the dual was exactly equal to some value for the primal. This introduces the question: was it a coincidence that this was the case? The following theorem claims that no, it was not a coincidence. In fact, this is always the case.

Theorem 14.3.2 ***(Strong LP Duality Theorem)*** *When P and D have non-empty feasible regions, then their optimal values are equal, i.e. $Val_P^* = Val_D^*$.*

On the number line we see that the maximum value for dual feasible solutions is equivalent to the minimum value for primal feasible solutions.

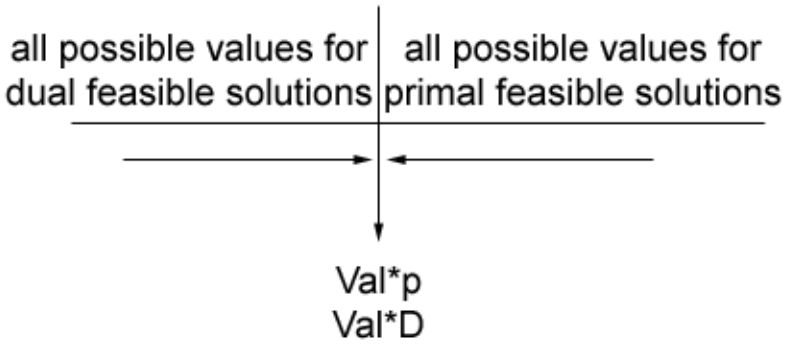


Figure 14.3.2: Primal vs. Dual in the Strong Duality Theorem.

It could happen that there is some LP with no solution that satisfies all of the constraints. In this case the feasible region would be empty. In this case the LP's dual will be unbounded in the sense that you can achieve any possible value in the dual.

The Strong LP Duality Theorem has a fairly simple geometric proof that will not be shown here due to time constraints. Recall from last time the proof of the Weak LP Duality Theorem.

Proof: (Theorem 14.3.1)

Start with some feasible solution to the dual LP, say y. Let y's objective function value be $Val_D(y)$. Let x be a feasible solution to the primal LP with objective function value $Val_P(x)$. Since y is a feasible solution for the dual and x is a feasible solution to the primal we have

$$\begin{aligned} Val_D(y) &= \sum_j b_j y_j \\ &\leq \sum_j (\sum_i A_{ij} x_i) y_j \\ &= \sum_i \sum_j A_{ij} x_i y_j \\ &= \sum_i (\sum_j A_{ij} y_j) x_i \\ &\leq \sum_i c_i x_i \\ &= Val_P(x) \end{aligned}$$

■

So just rewriting the equations shows us that the value of the dual is no more than the value of the primal. What happens if these values are exactly equal to one another? This happens when x and y are optimal solutions for the primal and the dual respectively. What can we deduce from this?

First, both of the inequalities in the proof of 14.3.1 must both be equalities. If these are equal we have

$$\sum_j b_j y_j = \sum_j (\sum_i A_{ij} x_i) y_j \tag{14.3.1}$$

and

$$\sum_i (\sum_j A_{ij} y_j) x_i = \sum_i c_i x_i \tag{14.3.2}$$

Then each term on the left hand side of Equation 14.3.1 must equal the corresponding term on the right hand side of Equation 14.3.1 and each term on the left hand side of Equation 14.3.2 must equal the corresponding term on the right hand side of Equation 14.3.2. This is the case if, in Equation 14.3.1, $b_j = \sum_i A_{ij} x_i$ and, in Equation 14.3.2, $c_i = \sum_j A_{ij} y_j$. What if $b_j \neq \sum_i A_{ij} x_i$ or $c_i \neq \sum_j A_{ij} y_j$, can we still have the equalities in Equation 14.3.1 and Equation 14.3.2? Equation 14.3.1 can if $y_j = 0$ and Equation 14.3.2 can if $x_i = 0$.

If x and y are primal feasible and dual feasible, respectively, such that $Val_D(y) = Val_P(x)$, then they must satisfy the following:

1. $\forall j$: either $y_j = 0$ or $b_j = \sum_i A_{ij} x_i$.
2. $\forall i$: either $x_i = 0$ or $c_i = \sum_j A_{ij} y_j$.

If $b_j = \sum_i A_{ij} x_i$, we say that the j^{th} constraint in the primal is tight. Condition 1 is called dual complementary slackness (DCS). Similarly, if $c_i = \sum_j A_{ij} y_j$, we say the the i^{th} constraint in the dual is tight. Condition 2 is called primal complementary slackness (PCS).

For a nicer view of the correspondence between the primal and the dual consider the following way of viewing this property.

Primal LP	**Dual LP**
$\min \sum_i c_i x_i \;\; s.t.$	$\max \sum_j b_j y_j \;\; s.t.$
$\sum_i A_{i1} x_i \geq b_1$	$y_1 \geq 0$
$\sum_i A_{i2} x_i \geq b_2$	$y_2 \geq 0$
$\vdots$	$\vdots$
$\sum_i A_{im} x_i \geq b_m$	$y_m \geq 0$
$x_1 \geq 0$	$\sum_j A_{1j} y_j \geq c_1$
$x_2 \geq 0$	$\sum_j A_{2j} y_j \geq c_2$
$\vdots$	$\vdots$
$x_n \geq 0$	$\sum_j A_{nj} y_j \geq c_n$

Here we have associated regular constraints in the primal to non-negativity constraints in the dual and non-negativity constraints in the primal to regular constraints in the dual along the rows of this table. Notice that the primal has m regular constraints and n non-negativity constraints on it and the dual has a variable for each of the m regular constraints and a regular constraint for each of the n non-negativity constraints. Therefore, there is a one-to-one correspondence between constraints in the primal and constraints in the dual. If x and y are optimal solutions for the primal and the dual, then for each row in this table, either the left side or the right side of the table must be tight, i.e. an equality rather than an inequality.

From the Strong LP Duality Theorem we have

Corollary 14.3.3 (x^*, y^*) *are primal and dual optimal solutions respectively if and only if they satisfy DCS and PCS.*

14.4 Simple Example

Consider the following minimization LP.

$$
\begin{aligned}
\min \;\; & x_1 + 3x_2 + 4x_3 + x_5 \;\; s.t. \\
& 5x_1 + 2x_4 \geq 1 \\
& 4x_2 + 3x_3 + x_4 + x_5 \geq 2 \\
& x_1 + x_3 + x_5 \geq 7 \\
& x_1 \geq 0 \\
& \vdots \\
& x_5 \geq 0
\end{aligned}
$$

We want to compute the dual of this LP. We want to introduce a variable for each regular constraint in the primal, so we will have variables y_1, y_2, and y_3 in the dual. We want to introduce a constraint

for each variable in the primal, so we will have 5 constraints in the dual. Each constraint in the dual is written as a function of the variables y_1, y_2, and y_3 in such a way that the coefficients of x_i sum up to no more than the coefficient of x_i in the objective function of the primal. Finally, we determine the objective function of the dual by letting the right hand sides of the constraints in the primal be the coefficients of the y_j. This gives us the following dual

$$\begin{aligned}
\max\ & y_1 + 2y_2 + 7y_3 \ \ s.t. \\
& 5y_1 + y_3 \leq 1 \\
& 4y_2 \leq 3 \\
& 3y_2 + y_3 \leq 4 \\
& 2y_1 + y_2 \leq 0 \\
& y_2 + y_3 \leq 1 \\
& y_1 \geq 0 \\
& y_2 \geq 0 \\
& y_3 \geq 0
\end{aligned}$$

In matrix form this transposes the coefficient matrix

x_1	x_2	x_3	x_4	x_5
5	0	0	2	0
0	4	3	1	1
1	0	1	0	1

into a coefficient matrix by transposing the first column of coefficients of the primal into the first row of coefficients of the dual, the second column into the second row, and so on.

y_1	y_2	y_3
5	0	1
0	4	0
0	3	1
2	1	0
0	1	1

14.5 The Power of LP-Duality

We would now like to use LP-Duality for designing algorithms. The basic idea behind why we might want to use LP-Duality to design algorithms is to solve a problem more quickly than using Linear Programming on its own. When we need to solve an algorithm with Linear Programming, we relax some Integer Program so that the corresponding LP gives us a good lower bound, solve the LP for some optimal linear solution, and round it off. It is possible to solve Linear Programs in polynomial time for its optimal solution, but the known algorithms still run fairly slowly in practice.

Also, we sometimes want to reduce some very large Integer Program and relax it to some exponential sized Linear Program with a small integrality gap, so that you get a good approximation. In which case, you cannot solve the linear program to optimality in polynomial time. LP-duality lets us take these programs, and instead of finding the optimal point of the LP exactly, we use some feasible solution to the Dual LP to get some lower bound on the optimal solution and often times that suffices. In fact, sometimes this process gives us a way to design a purely combinatorial for approximating a problem.

Instead of writing down the LP and solving the LP, we simultaneously construct feasible solutions to both the Primal and Dual LPs in such a way that these are within a small factor of each other, and then construct an integer solution using those. This is much faster and much simpler than solving the LP to optimality. Sometimes, this can even be used even for LPs of exponential size.

Another nice thing about LP-duality, is that it is such a non-trivial relationship between two seemingly different LPs, that it often exposes very beautiful min-max kinds of theorems about combinatorial objects. One such theorem is the Max-flow Min-cut Theorem, and we will shortly prove that using LP-Duality.

14.6 Max-flow Min-cut

Max-flow Min-cut is one of the most widely stated examples of LP-duality, when in fact there are many other examples of the theorems of that kind that arise from this particular relationship. For this example, though, we will talk about Max-flow Min-cut.

To explore this relationship, we would first like to create some LP that solves a Max-flow problem, find its Dual, and then see how the Dual relates to the Min-cut problem.

For the Max-flow problem, are given a graph $G = (V, E)$ with source s, sink t, and some capacities on edges c_e.

14.6.1 Max-flow LP

First we define the variables of our Max-flow LP:

x_e : amount of flow on edge e

Next we define the constraints on the variables:

$x_e \leq c_e \ \ \forall e$ - the flow on x_e does not exceed the capacity on e

$x_e \geq 0 \ \ \forall e$ - the flow on e is non-negative

$\sum_{e \in \delta^+(v)} x_e = \sum_{e \in \delta^-(v)} x_e \ \ \forall v \neq s, t$ - the flow entering v equals the flow leaving v

Subject to these constraints, our objective function maximizes the amount of flow from s to t, by summing over all the flow entering t or all the flow leaving s:

$\max \sum_{e \in \delta^-(s)} x_e$

14.6.2 Alternate Max-flow LP

There exists another equivalent way of writing the Max-flow LP in terms of the flow on paths:

x_p : amount of flow on an s-t path p

$\mathcal{P}$: set if all paths from s-t

This LP has the constraints on the variables:

$x_p \geq 0 \ \ \forall p \in \mathcal{P}$ - the flow on p is non-negative

$\sum_{p \ni e} x_p \leq c_e \ \ \forall e$ - the flow on e is no larger than the edge's capacity

Subject to these constraints, our objective function maximizes the amount of flow on the x_p paths:

$\max \sum_{p \in \mathcal{P}} x_p$

This is an equivalent LP, and the Primal we are going to talk about, because this one will be easier to work with when constructing the Dual.

14.6.3 The Dual to the Max-flow LP

First, we must define our variables for the Dual, where each variable in the Dual corresponds to a constraint in the Primal. Since the Primal contains one non-trivial constraint for every edge, the Dual must contain one variable for every edge in the Primal:

y_e : variable for edge $e \in E$

The Dual LP must have one constraint for every variable. Like we did in the Simple Example, for each variable in the Primal, we look at the coefficient that you get when you multiply the Primal constraints by the new variables, y_e, and sum over the number of constraints e. We want to find the coefficient of any fixed variable, and write the constraint coresponding to that variable. For any fixed path p, we analyze the coefficients using the constraints on x_p:

$\sum_e \sum_{p \ni e} x_p y_e \leq \sum_e c_e y_e$

$\sum_p x_p \sum_{e \in p} y_e \leq \sum_e c_e y_e$

From this analysis, we get the constraints on the variable y_e:

$y_e \geq 0 \ \ \forall e$

$\sum_{e \in p} y_e \leq 1 \ \ \forall p$

Subject to these constraints, our objective function is

$\min \sum_e y_e c_e$

So far, we have mechanically applied a procedure to a Primal LP and determined its Dual. When you start from a combinatorial problem and obtain its Dual, it is a good question to ask what these variables and constraints actually mean. Do they have a nice combinatorical meaning? Usually,

if you start from some combinatorial optimization problem, then its Dual does turn out to have a nice combinatorial meaning.

So what is the meaning of this Dual? It is saying that it wants to assign some value to every edge, such that looking at any s-t path in the graph, the sum total of the values on the edges in that path should be at least one. In order to understand what kind of solution this LP is asking for, think of an integral solution to the same LP. Say that we require $y_e \in \{0, 1\}$. What, then, is a integral feasible solution to this program? If $y_e \in \{0, 1\} \quad \forall e$, then we are picking some subset of the edges, specifically the subset of edges where $y_e = 1$. What properties should those edges satisfy? It should be that for all s-t paths in the graph, the collection of edges that we pick should have at least one edge from that path, and these edges form a cut in the graph G. If we were to minimize the solution over all the cuts in the graph, this would give us a Min-cut.

Fact 14.6.1 *all integral feasible solutions to the Dual form the set of all s-t cuts.*

Because the Dual is minimizing over some values, any feasible solution to the Dual is going to be greater than or equal to any feasible solution to the Primal, by the Weak Duality Theorem.

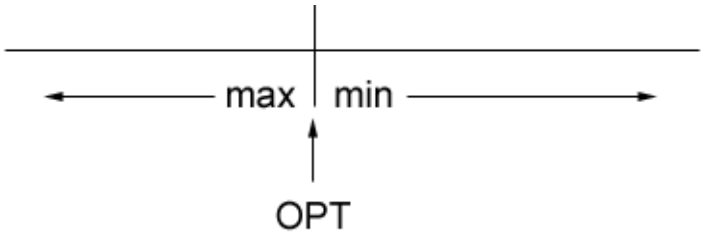


Figure 14.6.3: Ranges of minimization vs. maximazation feasible solutions.

From this we determine the following corollary:

Corollary 14.6.2 *Min-cut $\geq$ Max-flow*

This result is unsurprising, though determining that Min-cut = Max-flow is still non-trivial. Thankfully, the Strong Duality Theorem tells us:

Corollary 14.6.3 *Min-fractional-cut = Max-flow*

The optimal value of the Dual will be exactly equal to the optimal value of the Primal. So far, we do not know if the optimal solution to the Dual is an integral solution or not. If the solution to the Dual is integral, than it will prove the Max-flow Min-cut Theorem, but if it turns out to be a fractional solution, than we cannot determine the best solution to the problem. The Strong Duality Theorem gives us this weaker version of Max-flow Min-cut, that any fractional Min-cut is equal to the Max-flow of the graph. A fractional Min-cut is definted by this LP, where we assign some fractional value to edges, so that the total value of a path is less than or equal to 1. In order to get the Max-flow Min-cut Theorem from this weaker version, we need to show that for every fractional solution there exists integral solution that is no less optimal. We will give a sketch of the proof of this idea.

Theorem 14.6.4 *There exists an integral optimal solution to the Min-cut LP.*

Proof: To prove this, we will start with any fractional feasible solution to this LP, and derive from it an integral solution that is no worse than the fractional solution. Assume some fractional solution to the LP, and think of the values on each edge as lengths. Then the LP is assigning a length to every edge, with the property that any s-t path is going to have a length of at least one. This means that the distance from s to t is at least one.

We are going to lay out the graph in such a way that these y_e's exactly represent the lengths of the edges.

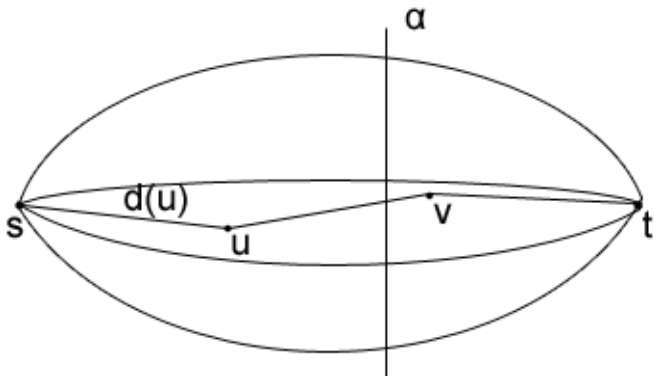


Figure 14.6.4: Distances y_e's from s to t.

Then, we will assign the distance to any point v from the source s as the minimum length of an s-v path. For any point v, we can look at the length of all paths from s to v, and the shortest such path gives me the distance from s to v, $d(v)$, with lengths according to the y_e's. From this we get the distances from s to all the points in the path, and we also know that $d(t) \geq 1$.

Then, we pick some value $\alpha \in [0, 1]$ uniformly at random, and look at all of the vertices that are within distance α of s. This will create a cut, because there are some number of edges crossing over between the set of vertices within distance α those with distances greater than α.

Suppose that we pick α uniformly at random from this range, than if we look at any particular edge in the graph, say (u, v), than the probability that the edge (u, v) is cut is no larger than its length. Why is this? The probability that (u, v) was cut is the probability that $d(u) \leq \alpha$ and $d(v) > \alpha$ (assuming $d(u) \leq d(v)$).

As we draw concentric circles of the distance α from s, then Figure 14.6.5 shows that the difference in the radii of these circles is no more than the distance from u to v. The only way that the edge can be cut, is if we chose an α somewhere between these two concentric circles. So the probability that (u, v) is cut is the difference in the radii, which is no more than the length of (u, v):

$$Pr\left[(u, v) \; is \; cut\right] \leq y_{u,v}$$

Then, the expectation of the size of the cut will be:

$$E\left[size \; of \; cut\right] \leq \sum_{e \in E} c_e Pr\left[e \; is \; cut\right]$$

$$E\left[size \; of \; cut\right] \leq \sum_{e \in E} c_e y_e$$

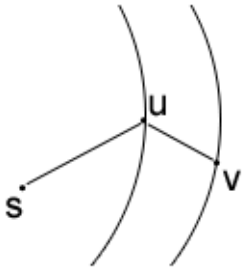


Figure 14.6.5: Distance from u to v vs. possible distances on α.

This expectation represents the size of the cut. If an edge e is in the cut, it contributes c_e to the size of the cut, and 0 otherwise. So we can assign indicator random variables to every edge that takes on a value 1 if it is in the cut, and 0 if it is not. So then the size of the cut is the weighted sum of the indicator variables. Then its expectation is the sum of its expected values of its variables, so it will turn out to be just the cost of each edge times the probability that the edge is in the cut, which is summed over all the edges.

As $\sum_{e \in E} c_e y_e$ is the value of the fractional solution, the expected value of the integer cut is no greater than that of the fractional solution. If we are picking a cut with some probability and the expected value of that cut is small, then there has to be at least one cut in the graph which has small value. So, there exists one integral cut with value at most the value of y.

■

14.7 Next Time

We will see more applications of LP-duality next time, as well as an algorithm based on LP-duality.

CS787: Advanced Algorithms

Scribe: Mayank Maheshwari, Dan Rosendorf
Lecturer: Shuchi Chawla
Topic: Primal-Dual Algorithms
Date: October 19 2007

15.1 Introduction

In the last lecture, we talked about Weak LP-Duality and Strong LP-Duality Theorem and gave the concept of Dual Complementary Slackness (DCS) and Primal Complementary Slackness (PCS). We also analyzed the problem of finding maximum flow in a graph by formulating its LP.

In this lecture, we will develop a 2-approximation for Vertex Cover using LP and introduce the Basic Primal-Dual Algorithm. As an example, we analyze the primal-dual algorithm for vertex cover and later on in the lecture, give a brief glimpse into a 2-player zero-sum game and show how the pay-offs to players can be maximized using LP-Duality.

15.2 Vertex Cover

We will develop a 2-approximation for the problem of weighted vertex cover. So for this problem:

Given: A graph $G(V, E)$ with weight on vertex v as w_v.

Goal: To find a subset $V' \subseteq V$ such that each edge $e \in E$ has an end point in V' and $\sum_{v \in V'} w_v$ is minimized.

The Linear Program relaxation for the vertex cover problem can be formulated as:

The variables for this LP will be x_v for each vertex v. So our objective function is

$$min \sum_v x_v w_v$$

subject to the constraints that

$$x_u + x_v \geq 1 \qquad \forall (u, v) \in E$$

$$x_v \geq 0 \qquad \forall v \in V$$

The Dual for this LP can be written with variables for each edge $e \in E$ as maximizing its objective function:

$$max \sum_{e \in E} y_e$$

subject to the constraints

$$\sum_{v:(u,v) \in E} y_{uv} \leq w_u \qquad \forall u \in V$$

$$y_e \geq 0 \qquad \forall e \in E$$

Now to make things simpler, let us see how the unweighted case for vertex cover can be formulated in the form of a Linear Program. We have the objective function as:

$$max \sum_{e \in E} y_e$$

such that

$$\sum_{e \text{ incident on } u} y_e \leq 1 \qquad \forall u \in V$$

$$y_e \geq 0 \qquad \forall e \in E$$

These constraints in the linear program correspond to finding a matching in the graph G and so the objective function becomes finding a maximum matching in the graph. Hence, this is called the *Matching LP*.

Now remember from the previous lecture where we defined Dual and Primal Complementary Slackness. These conditions follow from the Strong Duality Theorem.

Corollary 15.2.1 (Strong Duality) *x and y are optimal for Primal and Dual LP respectively iff they satisfy:*

1. *Primal Complementary Slackness (PCS) i.e. $\forall i$, either $x_i = 0$ or $\sum_j A_{ij} y_j = c_i$.*
2. *Dual Complementary Slackness (DCS) i.e. $\forall j$, either $y_j = 0$ or $\sum_i A_{ij} x_i = b_j$.*

15.3 Basic Primal-Dual Algorithm

1. Start with $x = 0$ (variables of primal LP) and $y = 0$ (variables of dual LP) . The conditions that:

   - y is feasible for Dual LP.
   - Primal Complementary Slackness is satisfied.

   are invariants and hence, hold for the algorithm. But the condition that:

   - Dual Complementary Slackness is satisfied.

   might not hold at the beginning of algorithm. x does not satisfy the primal LP as yet.

2. *Raise* some of the y_j's, either simultaneously or one-by-one.
3. Whenever a dual constraint becomes tight, *freeze* values of corresponding y's and raise value of corresponding x.
4. Repeat from *Step 2* until all the constraints become tight.

Now let us consider the primal-dual algorithm for our earlier example of vertex cover.

15.3.1 Primal-Dual Algorithm for Vertex Cover

1. Start with $x = 0$ and $y = 0$.
2. Pick any edge e for which y_e is not frozen yet.
3. Raise the value of y_e until some vertex constraint v goes tight.
4. Freeze all y_e's for edges incident on v. Raise x_v to 1.
5. Repeat until all y_e's are frozen.

Let us see an example of how this algorithm works on an instance of the vertex cover problem. We consider the following graph:

Example: Given below is a graph with weights assigned to vertices as shown in the figure and we start with assigning $y_e = 0$ for all edges $e \in E$.

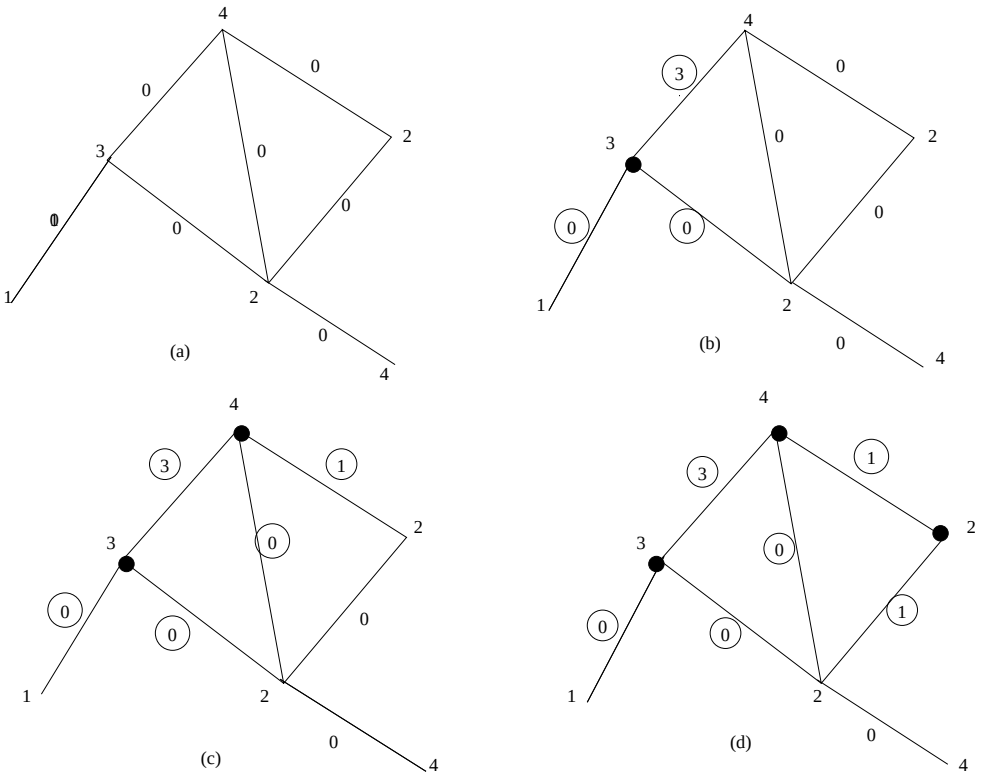


Fig 1: The primal-dual algorithm for vertex cover.

So the algorithm proceeds as shown in the figure above. In steps (a)-(d), an edge is picked for which y_e is not frozen and the value of y_e is raised until the corresponding vertex constraint goes tight. All the edges incident on that vertex are then frozen and value of x_v is raised to 1.

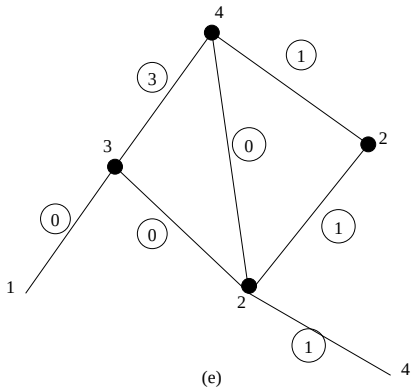


Fig 1: The vertex cover for the example graph.

When all the y_e's get frozen, the algorithm terminates. So the *Value of Primal=11* and the *Value of Dual=6*.

Hence,

$$Val_p(x) \leq 2Val_d(y)$$

and so we get the 2-approximation for our instance of vertex cover.

Lemma 15.3.1 *When the algorithm ends, x is a feasible solution for the primal and y is a feasible solution for the dual.*

Proof: That y is a feasible solution is obvious since at every step we make sure that y is a feasible solution, so it is feasible at the last when the algorithm ends. To see that x is a feasible solution we proceed by contradiction. Let us suppose it is not a feasible solution. Then there is a constraint $x_u + x_v \geq 1$ which is violated. That means both x_v and x_u are zero and thus it must be possible to raise the value of the edge between them since neither of them has a tight bound. This is a contradiction with the fact that all y_e's are frozen. ■

Lemma 15.3.2 *x and y satisfy PCS.*

Proof: This is obvious since at every step we make sure that x and y satisfy PCS. ■

Definition 15.3.3 *Let x and y be feasible solutions to the primal and dual respectively. Then we say that x and y satisfy α-approximate DCS if $\forall j, (y_j \neq 0) \rightarrow (\sum_i A_{ij}x_i \leq \alpha b_j)$.*

Lemma 15.3.4 *x and y satisfy 2-approximate DCS.*

Proof: This follows from the fact that x_v is either 0 or 1 for any v so $x_u + x_v \leq 2$ for any $e = (u, v)$. ■

The next lemma shows us why we would want an α-approximation for DCS.

Lemma 15.3.5 *Suppose x and y satisfy PCS are feasible for Primal and Dual respectively and α-approximate DCS then $Val_P(x) \leq \alpha Val_D(y)$.*

Proof: To prove this we only need to write out the sums. We have $Val_P(x) = \sum_i c_i x_i$ now since we know that x,y satisfy PCS we have that $\sum_i c_i x_i = \sum_i (\sum_j A_{ij} y_j) x_i$ by reordering the

summation we get $\sum_j(\sum_i A_{ij}x_i)y_j \leq \sum_j \alpha b_j y_j = \alpha \sum_j b_j y_j = \alpha Val_D(y)$ where the $\leq$ follows from α-approximate DCS. ■

The last two lemmas then directly yield the desired result which is that our algorithm is a 2-approximation for Vertex cover.

We should also note that we never used anywhere in our analysis what order we choose our edges in or how exactly we raise the values of y_e's. A different approach might be to raise the values of all our y_e's simultaneously until some condition becomes tight. Using this approach with the graph we had earlier would give a different result. We would start by raising the values of all edges to $\frac{1}{2}$ at which point the vertex at the bottom of the diamond would become full and it's tightness condition would be met. We freeze all the edges adjecent to that vertex add it to our vertex cover and continue raising the values of the other y_e's. The next condition to be met will be the left bottom most node when we raise the value of the incoming edge to 1. After that we continue rasing the values of the remaining unfrozen edges until $1\frac{1}{2}$ when both the left and right edge of the diamond become full and we freeze all the remaining edges. The run of this algorithm with the circled numbers denoting frozen edge capacities and the emptied nodes denoting nodes in the graph cover can be seen in the following figures.

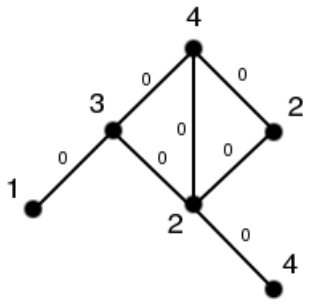


Figure 15.3.1: Step 0

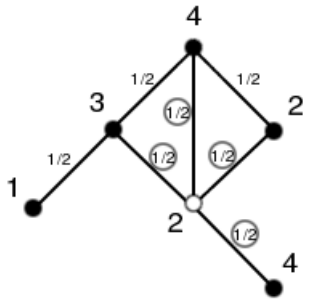


Figure 15.3.2: Step 1

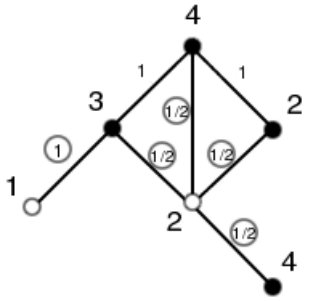


Figure 15.3.3: Step 2

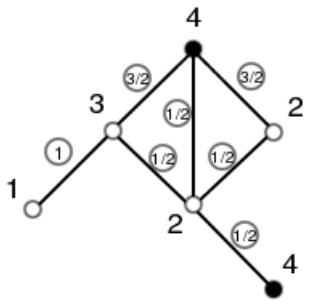


Figure 15.3.4: Step 3

Note that while this approach gives a better result in this case than the first one we used, it is not that case in general. In fact there are no known polynomial time algorithms that give a result of

$(2-\epsilon)$-approximation for any ϵ small constant.

15.4 Minimax principle

Next we will look at an application of LP-duality in the theory of games. In particular we will prove the Minimax theorem using LP-duality. First though we will need to explain some terms. By a 2-player zero sum game, we mean a protocol in which 2 players choose strategies in turn and given two strategies x and y, we have a valuation function $f(x,y)$ which tells us what the payoff for the first player is. Since it is a zero sum game, the payoff for the second player is exactly $-f(x,y)$. We can view such a game as a matrix of payoffs for one of the players.

As an example take the game of Rock-paper-scissors, where the payoff is one for the winning party or 0 if there is a tie. The matrix of winnings for player one will then be the following:

$$\begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{pmatrix}$$

Where A_{ij} corresponds to the payoff for player one if player one picks the i-th element and player two the j-th element of the sequence $(\text{Rock}, \text{Paper}, \text{Scissors})$. We will henceforth refer to player number two as the column player and player number one as the row player. If the row player goes first, he obviously wants to minimize the possible gain of the column player. So the payoff to the row player in that case will be $min_j(max_i(A_{ij}))$. On the other hand if the column player goes first, we get the payoff being $max_i(min_j(A_{ij})$. It is clearly the case that $max_i(min_j(A_{ij})) \leq min_j(max_i(A_{ij}))$.

The Minimax theorem states that if we allow the players to choose probability distributions instead of a given column or row then equality holds or slightly more formally:

Theorem 15.4.1 *If x and y are probability vectors then $max_y(min_x(y^T Ax) = min_x(max_y(y^T Ax))$.*

Proof: We will only give a proof sketch. Notice that once we have chosen a strategy, if our opponent wants to minimize his loss, he will always just pick the one row which gives the best result, not a distribution. For $max_y(min_x(y^T Ax))$, we wish to find a distribution over the rows such that whatever column gets picked, we get at least a payoff of t such that t is maximal. This can be formulated in terms of an LP-problem as maximizing t given the following set of constraints:

$$\forall j \quad \sum_i y_i A_{ij} \geq t$$

and

$$\sum_i y_i = 1$$

, which can be changed into a more standard form by writing the equations as

$$\forall j \quad t - \sum_i y_i A_{ij} \leq 0$$

and relaxing the second condition as

$$\sum_j y_j \leq 1.$$

On the other hand $min_x(max_y(y^T Ax))$ can be thought of as trying to minimize the loss and can thus be rewritten in terms of an LP-problem in a similar fashion as minimizing s given that

$$\sum_j x_j \leq 1$$

and

$$\forall i \quad s - \sum_j A_{ij} x_j \leq 0$$

.

In other words, we are trying to minimize the loss, no matter which column our opponent chooses. We leave it as an exercise to prove that the second problem is a dual of the first problem and thus by the Strong duality theorem, the proof is concluded. ■

CS787: Advanced Algorithms

Scribe: David Malec and Xiaoyong Chai **Lecturer:** Shuchi Chawla

Topic: Minimax Theorem and Semi-Definite Programming **Date:** October 22 2007

In this lecture, we first conclude our discussion of LP-duality by applying it to prove the Minimax theorem. Next we introduce vector programming and semi-definite programming using the Max-Cut problem as a motivating example.

16.1 L.P. Duality Applied to the Minimax Theorem

The Minimax Theorem relates to the outcome of two player zero-sum games. Assume we have a payoff matrix A for the game, where columns and rows represent moves that each of the players can employ (we refer to the players hereafter as the Column Player and the Row Player respectively), and the entries in the matrix represent the payoff for the Row Player. Note that we only need the one matrix to describe the outcome, since in a zero-sum game the payoff to the Column Player and the payoff to the Row Player must sum to zero. Now, if we call the strategy employed by the Row Player x and the strategy employed by the Column Player y, we can express the optimal payoff to the Row player as

$$\max_x \min_y x^T A y \qquad \text{if the Row Player goes first} \qquad (16.1.1)$$

$$\min_y \max_x x^T A y \qquad \text{if the Column Player goes first} \qquad (16.1.2)$$

The Minimax Theorem says that if we allow the Row Player and Column Player to select probability distributions over the rows and columns of A, respectively, for their strategies, then (16.1.1) is equal to (16.1.2). We can show this by considering a Linear Program for finding (16.1.1), determining its Dual, and then recognizing that the Dual will find (16.1.2). The Minimax Theorem then follows immediately from the Strong LP-Duality Theorem. Consider the Linear Programs

Primal:

$$\begin{array}{lll} \max t \text{ subject to} & & \\ t - \sum_j A_{ij} y_j & \leq 0 \quad \forall i & : x_i \\ \sum_j y_j & \leq 1 & : s \\ y_j & \geq 0 \quad \forall j & \end{array}$$

Dual:

$$\begin{array}{lll} \min s \text{ subject to} & & \\ s - \sum_i A_{ij} x_i & \geq 0 \quad \forall j & : y_j \\ \sum_i x_i & \geq 1 & : t \\ x_i & \geq 0 \quad \forall i & \end{array}$$

Note that while technically we require that $\sum_j y_j = \sum_i x_i = 1$, since both are probability distributions, the relaxations of this conditions seen in the Primal and Dual Linear Programs are sufficient, since $\sum_j y_j < 1$ and $\sum_i x_i > 1$ correspond to suboptimal values for their respective objective functions.

The preceding demonstrates the Minimax Theorem to be true, since with a little thought it can be seen that maximizing t in the Primal Linear Program corresponds exactly to finding the value of (16.1.1), and minimizing s in the Dual Linear Program corresponds exactly to finding the value of (16.1.2); hence, the Strong LP-Duality Theorem gives us that these values are in fact equal.

16.2 Max-Cut Problem

Some problems can not be reasonably approximated using Linear Programming, because there is no Linear Programming relaxation that both is of small size and has a good integrality gap. We can sometimes use the technique of Semi-Definite Programming to find approximations to such problems. We will consider the Max-Cut problem as an example of this type of problem. Formally, we can state the Max-Cut Problem as follows:

Given: a graph $G = (V, E)$, and a cost function on the edges c_e.

Goal: Form a partition (V_1, V_2) of V such that $\sum_{e \in E'} c_e$ is maximized, where $E' = E \cap (V_1 \times V_2)$.

We can form a program for this problem by considering making a variable for each $v \in V$ to represent which of the sets V_1, V_2 contains it, say $x_v \in \{0, 1\}$, where a value of 0 indicates it is in V_1 and a value of 1 indicates it is in V_2. Then an edge (u, v) crosses the partition if and only if $x_u \neq x_v$, which is equivalent to the condition $|x_u - x_v| = 1$, by the definition of our variables. Thus, we can translate our original goal into an objective function to get the program:

$$\begin{array}{ll} \max \sum_{(u,v) \in E} |x_u - x_v| c_{(u,v)} \text{ subject to} & \\ x_v \in \{0, 1\} & \forall v \in V \end{array}$$

Looking more closely at this program, however, we can see that there is a fundamental issue with it — specifically, the objective function makes use of the absolute value function, which isn't linear. Previously, we have seen cases where we were able to convert an objective function containing an absolute value into a linear objective function by adding variables and constraints, so it might

seem like we could do so here. The problem with that idea is that our objective function is to be maximized in this case. Before, we rewrote

$$\min|t|$$

as

$$\begin{aligned}&\min t' \text{ subject to}\\ &t' \geq t\\ &t' \geq -t\end{aligned}$$

This worked because we could think in terms of minimizing an upper bound on both t and $-t$. If we try the same approach with a maximization problem, however, we would be proposing to rewrite

$$\max|t|$$

as

$$\begin{aligned}&\min t' \text{ subject to}\\ &t' \geq t\\ &t' \leq -t\end{aligned}$$

This doesn't work, because the two constraints on t' conflict at a very basic level. Thus, if we want to solve this problem, we need to find another way to deal with the issue of nonlinearity in our program instead of trying to make the proposed program linear. So instead of focusing on this, we will instead focus on how to make it easier to relax the program, since this is necessary to finding an approximation regardless of our approach. We rewrite our program as an equivalent Quadratic Program, specifically

$$\begin{aligned}&\max \sum_{(u,v)\in E} \frac{1 - x_u x_v}{2} c_{(u,v)} \text{ subject to}\\ &x_v \in \{-1, 1\} \qquad \forall v \in V\end{aligned}$$

This is preferable to our previous program, since we can relax the integrality constraint $x_v \in \{-1, 1\} \qquad \forall v \in V$ to the constraint $x_v^2 = 1 \qquad \forall v \in V$.

Now, since the Max-Cut Problem is NP-Hard, we can see that this implies that Quadratic Programs, unlike Linear Programs, must be NP-Hard. There are certain types that we can find decent approximations to in a reasonable amount of time. We consider one such type next.

16.3 Vector Programs

A Vector Program is a program over vectors $y_1, y_2, \ldots, y_n$ (in general, these vectors are high-dimensional), where we write the program in the form

$$\min / \max \sum_{i,j} c_{ij} y_i \cdot y_j \text{ subject to}$$
$$\sum_{i,j} A_{ij} y_i \cdot y_j \geq b$$

In the above expressions, $(\cdot)$ represents taking a dot product. These can be thought of as Linear Programs in the dot products of the vectors they are over. While we can solve such programs, it is important to note that we can't constrain the dimension of the y_i — they could end up being very high-dimensional.

While we could reinterpret our original program for the Max-Cut Problem as a Vector Program, there are some very basic issues with this. Specifically, in order to be able to give a meaningful interpretation of the values of the x_v, we would need to require that they be essentially one-dimensional; however, as we just noted, we can place no such constraints on the number of dimensions the vectors in such a program have.

16.4 Semi-Definite Programming

One limitation of linear programming is that it cannot handle nonlinear constraints. Semi-definite programming, as a generalization of linear programming, enables us to specify in addition to a set of linear constraints a "semi-definite" constraint, a special form of nonlinear constraints. In this section, we introduce the basic concept of semi-definite programming.

16.4.1 Definitions

First Let us define a positive semi-definite (*p.s.d.*) matrix A, which we denote as $A \succeq 0$.

Definition 16.4.1 *A matrix $A \in \mathbb{R}^{n \times n}$ is positive semi-definite if and only if (1) A is symmetric, and (2) for all $x \in \mathbb{R}^n$, $x^T A x = \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij} x_i x_j \geq 0$.*

Then we define a semi-definite program (SDP) as follows.

Definition 16.4.2 *An SDP is a mathematical program with four components:*

- *a set of variables x_{ij};*
- *a linear objective function to minimize/maximize;*
- *a set of linear constraints over x_{ij};*
- *a semi-definite constraint $X = [x_{ij}] \succeq 0$.*

The semi-definite constraint is what differentiates SDPs from LPs. Interestingly, the constraint can be interpreted as an infinite class of linear constraints. This is because by Definition 16.4.1, if

$X \succeq 0$, then for all $v \in \mathbb{R}^n$, $v^T X v \geq 0$. Each possible real vector v gives us one linear constraint. Altogether, we have an infinite number of linear constraints. We will use this interpretation later in this section.

16.4.2 A Different View of Semi-Definite Programming

In fact, semi-definite programming is equivalent to vector programming. To show this, we first present the following theorem.

Theorem 16.4.3 *For a symmetric matrix A, the following statements are equivalent:*

1. *$A \succeq 0$.*
2. *All eigenvalues of A are non-negative.*
3. *$A = C^T C$, where $C \in \mathbb{R}^{m \times n}$.*

Proof: We prove the theorem by showing statement 1 implies statement 2, statement 2 implies statement 3, and statement 3 implies statement 1.

$1 \Rightarrow 2$**:** If $A \succeq 0$, then all eigenvalues of A are non-negative.

Recall the concept of eigenvectors and eigenvalues. The set of eigenvectors ω for A is defined as those vectors that satisfy $A\omega = \lambda\omega$, for some scalar λ. The corresponding λ values are called eigenvalues. Moreover, when A is a symmetric matrix, all the eigenvalues are real-valued. Now let us look at each eigenvalue λ. First of all, we have $A\omega = \lambda\omega$, where ω is the corresponding eigenvector. Multiplying both sides by ω^T, then we have:

$$\omega^T A \omega = \lambda \omega^T \omega \tag{16.4.3}$$

The LHS of Equation 16.4.3 is non-negative by the definition of positive semi-definite matrix A. On the RHS of the equation, $\omega^T \omega \geq 0$. Thus $\lambda \geq 0$.

$2 \Rightarrow 3$**:** If all eigenvalues of A are non-negative, then $A = C^T C$ for some real matrix C.

Let Λ denote the diagonal matrix with all of A's eigenvalues:

$$\begin{bmatrix} \lambda_1 & & 0 \\ & \ddots & \\ 0 & & \lambda_n \end{bmatrix}.$$

Let W denote the matrix of the corresponding eigenvectors $[\omega_1\ \omega_2\ \cdots\ \omega_n]$. Since eigenvectors are orthogonal to one another (*i.e.*, $\omega_i^T \omega_j = 1$ if $i = j$; 0, otherwise), W is an orthogonal matrix. Given Λ and W, we obtain the matrix representation of eigenvalues: $AW = W\Lambda$. Multiplying both side by W^T, we have:

$$AWW^T = W\Lambda W^T. \tag{16.4.4}$$

Since W is an orthogonal matrix, $WW^T = I$ and thus the LHS of Equation 16.4.4 is equal to A. Since all the eigenvalues are non-negative, we can decompose Λ into $\Lambda^{\frac{1}{2}}(\Lambda^{\frac{1}{2}})^T$, where $\Lambda^{\frac{1}{2}}$ is defined as:

$$\begin{bmatrix} (\lambda_1)^{\frac{1}{2}} & & 0 \\ & \ddots & \\ 0 & & (\lambda_n)^{\frac{1}{2}} \end{bmatrix}.$$

Thus $W\Lambda W^T = W\Lambda^{\frac{1}{2}}(\Lambda^{\frac{1}{2}})^T W^T = W\Lambda^{\frac{1}{2}}(W\Lambda^{\frac{1}{2}})^T$. Let $C^T = W\Lambda^{\frac{1}{2}}$. Equation 16.4.4 is equivalent to $A = C^T C$.

$3 \Rightarrow 1$**:** If $A = C^T C$ for some real matrix C, then $A \succeq 0$.

We prove it from the definition of A being *p.s.d.*:

$$\begin{aligned} A &= C^T C \\ \Leftrightarrow x^T A x &= x^T C^T C x, \quad \text{where } x \in \mathbb{R}^n \\ \Leftrightarrow x^T A x &= y^T y, \quad \text{where } y = Cx \\ \Rightarrow x^T A x &\geq 0, \quad \text{since } y^T y \geq 0 \end{aligned}$$

■

Note that by statement 3, a positive semi-definite matrix can be decomposed into $C^T C$. Let $C = [C_1, C_2 \cdots C_n]$, where $C_i \in \mathbb{R}^m$. Thus A_{ij} is the dot product of C_i and C_j. This gives us the following corollary.

Corollary 16.4.4 *SDP is equivalent to vector programming.*

16.4.3 Feasible Region of an SDP

Similar to an LP, a linear constraint in an SDP produces a hyper-plane (or a flat face) that restricts the feasible region of the SDP. The semi-definite constraint, which is nonlinear, produces a non-flat face. In fact, as we discussed at the end of Section 16.4.1, this nonlinear constraint can be interpreted as an infinite number of linear constraints. As an example, the feasible region of an SDP can be visualized as in Figure 16.4.3. In the figure, C_1, C_2 and C_3 are three linear constraints, and C_4 is the nonlinear constraint. Constraints C_a and C_b are two instances among the infinite number of linear constraints corresponding to C_4. These infinite linear constraints produces the non-flat face of C_4.

The optimal solution to an SDP can lie on the non-flat face of the feasible region and thus can be irrational. Below we give a simple example of an irrational optimum to illustrate this point.

Example: Minimize x subject to the constraint:

$$\begin{bmatrix} x & \sqrt{2} \\ \sqrt{2} & x \end{bmatrix} \succeq 0.$$

For positive semi-definite matrices, all the leading principal minors are non-negative. The leading principal minors of an $n \times n$ matrix are the determinants of the submatrices obtained by deleting

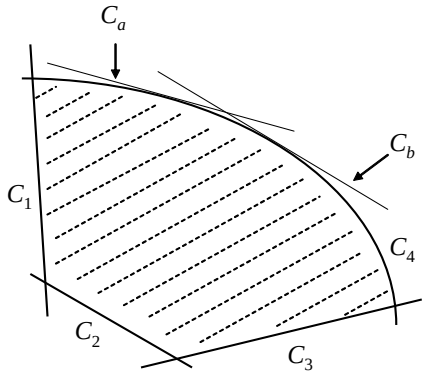


Figure 16.4.1: The feasible region of an SDP

the last k rows and the last k columns, where $k = n-1, n-2, \ldots, 0$. In the example, the matrix has two leading principal minors:

$$\begin{aligned} m_1 &= \big|[x]\big| = x, \quad \text{and} \\ m_2 &= \left|\begin{bmatrix} x & \sqrt{2} \\ \sqrt{2} & x \end{bmatrix}\right| = x^2 - 2. \end{aligned}$$

Thus we have $x \geq 0$ and $x^2 - 2 \geq 0$. It immediately follows that the minimum value of x is $\sqrt{2}$. ■

Finally we state without proving the following theorem on SDP approximation.

Theorem 16.4.5 *We can achieve a (1+ϵ)-approximation to an SDP in time polynomial to n and $1/\epsilon$, where n is the size of the program.*

CS787: Advanced Algorithms

Scribe: Amanda Burton, Leah Kluegel
Lecturer: Shuchi Chawla
Topic: Max-Cut via SDP ctd., Streaming Algorithms
Date: October 24, 2007

17.1 Last Time

Last time we discussed Semi-Definite Programming. We saw that Semi-Definite Programs are a generalization of Linear Programs subject to the nonlinear constraint that a matrix of variables in the program is Positive Semi-Definite. Semi-Definite Programs are also equivalent to Vector Programs. Vector Programs are programs over vectors instead of real uni-dimensional variables. The constraints and objective function of these programs are linear over dot products of vectors. We know how to solve these programs to within a $(1+\epsilon)$-factor for any constant $\epsilon > 0$ in time that is polynomial in the size of the program and $1/\epsilon$. So we can assume that we can get the optimal solution, or at least a near optimal solution, to these programs. To design approximation algorithms, instead of expressing programs as Integer Programs and then relaxing them to Linear Programs, we can express programs as Integer Programs or Quadratic Integer Programs, relax them to Vector Programs, solve the Vector Program using Semi-Definite Programming, and then somehow round the solutions to integer solutions.

17.2 Max-Cut

Recall that for the Max-Cut problem we want to find a non-trivial cut of a given graph such that the edges crossing the cut are maximized, i.e.

Given: $G = (V, E)$ with c_e =cost on edge e.
Goal: Find a partition (V_1, V_2), $V_1, V_2 \neq \phi$, $\max \sum_{e \in (V_1 \times V_2) \cap E} c_e$.

17.2.1 Representations

How can we convert Max-Cut into a Vector Program?

17.2.1.1 Quadratic Programs

First write Max-Cut as a Quadratic Program.

Let $x_u = 0$ if vertex u is on the left side of the cut and $x_u = 1$ if vertex u is on the right side of the cut. Then we have

Program 1:

$$\begin{array}{lll} \text{maximize} & \sum_{(u,v)\in E} (x_u(1-x_v) + x_v(1-x_u))c_{uv} & \text{s.t.} \\ & x_u \in \{0,1\} & \forall u \in V \end{array}$$

Alternatively, let $x_u = -1$ if vertex u is on the left side of the cut and $x_u = 1$ if vertex u is on the

right side of the cut. Then we have

Program 2:

$$\begin{array}{lll} \text{maximize} & \sum_{(u,v)\in E} \frac{(1-x_u x_v)}{2} c_{uv} & \text{s.t.} \\ & x_u \in \{-1, 1\} & \forall u \in V \end{array}$$

Note that $x_u x_v = -1$ exactly when the edge (u, v) crosses the cut.

We can express the integrality constraint as a quadratic constraint yielding

Program 3:

$$\begin{array}{lll} \text{maximize} & \sum_{(u,v)\in E} \frac{(1-x_u x_v)}{2} c_{uv} & \text{s.t.} \\ & x_u^2 = 1 & \forall u \in V \end{array}$$

In these programs an edge contributes c_{uv} exactly when the edge crosses the cut. So in any solution to these quadratic programs the value of the objective function exactly equals the total cost of the cut. We now have an exact representation of the Max-Cut Problem. If we can solve Program 3 exactly we can solve the Max-Cut Problem exactly.

17.2.1.2 Vector Program

Now we want to relax Program 3 into a Vector Program.

Recall:
A Vector Program is a Linear Program over dot products.

So relax Program 3 by thinking of every product as a dot product of two n-dimensional vectors, x_u.

Program 4:

$$\begin{array}{lll} \text{maximize} & \sum_{uv\in E} \frac{(1-x_u \cdot x_v)}{2} c_{uv} & \text{s.t.} \\ & x_u \cdot x_u = 1 & \forall u \in V \end{array}$$

This is something that we know how to solve.

17.2.1.3 Semi-Definite Program

How would we write this as a Semi-Definite Program?

Recall:

Definition 17.2.1 *A Semi-Definite Program has a linear objective function subject to linear constraints over x_{ij} together with the semi-definite constraint $[x_{ij}] \succeq 0$.*

Theorem 17.2.2 *For any matrix A, A is a Positive Semi-Definite Matrix if the following holds.*

$$\begin{aligned} A \succeq 0 &\Rightarrow v^T A v \geq 0 \quad \forall v \in V \\ &\Leftrightarrow A = C^T C, \text{ where } C \in \mathbb{R}^{m \times n} \\ &\Leftrightarrow A_{ij} = C_i \cdot C_j \end{aligned}$$

To convert a Vector Program into a Semi-Definite Program replace dot products with variables and add the constraint that the matrix of dot products is Positive Semi-Definite.

So our Vector Program becomes

Program 5:

$$\begin{array}{lll} \text{maximize} & \sum_{(u,v)\in E} \frac{(1-y_{uv})}{2} c_{uv} & \text{s.t.} \\ & y_{uu} = 1 & \forall u \in V \\ & [y_{ij}] \succeq 0 & \end{array}$$

If we have any feasible solution to this Semi-Definite Program (Program 5), then by the above theorem there are vectors $\{x_u\}$ such that $y_{uv} = x_u \cdot x_v \forall u, v \in V$. The vectors $\{x_u\}$ are a feasible solution to the Vector Program (Program 4). Thus the Semi-Definite Program (Program 5) is exactly equivalent to the Vector Program (Program 4).

17.2.2 Solving Vector Programs

We know how to get arbitrarily close to the exact solution to a Vector Program. So we get some set of vectors for which each vertex is mapped to some n-dimensional vector around the origin. These vectors are all unit vectors by the constraint that $x_u \cdot x_u = 1$, so the vectors all lie in some unit ball around the origin.

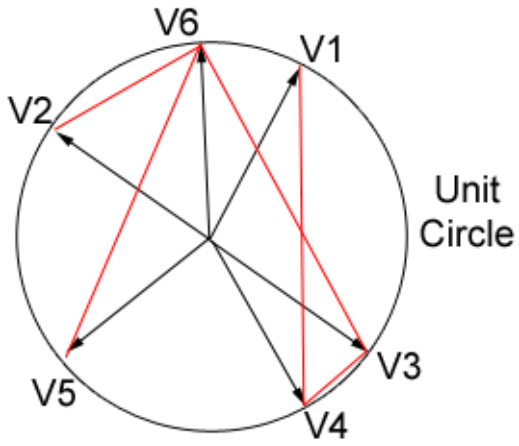


Figure 17.2.1: The unit vectors in the Vector Program solution.

Now we have some optimal solution to the Vector Program. Just as in the case when we had an exact representation as an Integer Program and relaxed it to a Linear Program to get an optimal solution that was even better than the optimal integral solution to the Integer Program, here we have some exact representation of the problem and we are relaxing it to some program that solves the program over a larger space. The set of integral solutions to Program 3 form a subset of

feasible solutions to Program 4, i.e., Program 4 has a larger set of feasible solutions. Thus the optimal solution for the Vector Program is going to be no worse than the optimal solution to the original problem.

Fact: $OPT_{VP} \geq OPT_{Max-Cut}$.

Goal: Round the vector solution obtained by solving the VP to an integral solution (V_1, V_2) such that the total value of our integral solution $\geq \alpha OPT_{VP} \geq \alpha OPT_{Max-Cut}$.

Our solution will put some of the vectors on one side of the cut and the rest of the vectors on the other side of the cut. Our solution is benefitting from the edges going across the cut that is produced. Long edges crossing the cut will contribute more to the solution than short edges crossing the cut because the dot product is smaller for the longer edges than the shorter edges. We want to come up with some way of partitioning these vertices so that we are more likely to cut long edges.

17.2.3 Algorithm

In two dimensions good cut would be some plane through the origin such that vectors on one side of the plane go on one side of the cut and vectors that go on the other side of the plane go on the other side of the cut. In this way we divide contiguous portions of space rather than separating the vectors piecemeal.

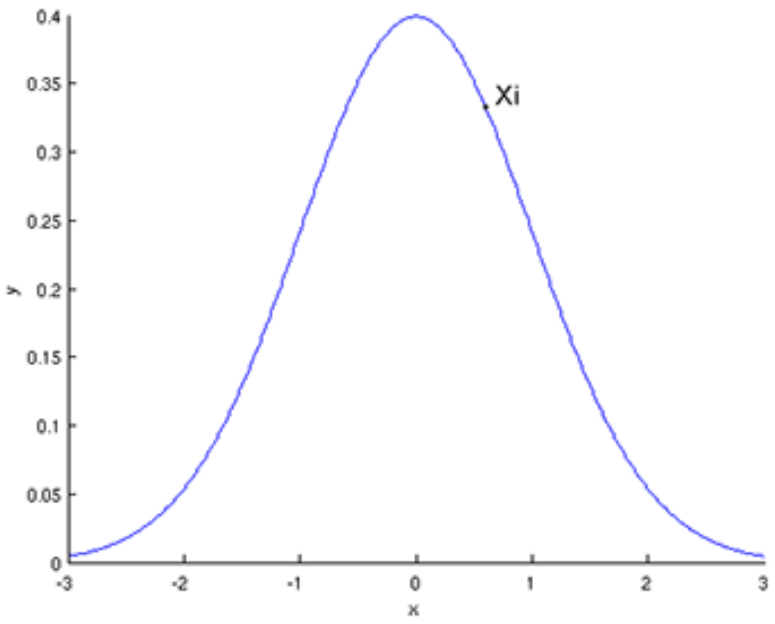


Figure 17.2.2: A sample Gaussian variable, x.

1. Pick a "random" direction - unit vector $\hat{n}$.

2. Define $V_1 = \{v | x_v \cdot \hat{n} \geq 0\}$, $V_2 = \{v | x_v \cdot \hat{n} < 0\}$.

3. Output (V_1, V_2).

To pick a "random" direction we want to pick a vector uniformly at random from the set of all unit vectors. Suppose we know how to pick a normal, or gaussian, variable in one dimension. Then pick from this normal distribution for every dimension.

This gives us a point that is spherically symmetric in the distribution. The density of any point x in the distribution is proportional to $exp^{-\frac{1}{2}x^2}$. So if each of the components, x_i is picked using this density, the density of the vector is proportional to $\prod_i exp^{-\frac{1}{2}x_i^2} = exp^{(-\frac{1}{2}\sum_i x_i^2)}$, which depends only on the length of the vector and not the direction. Now normalize the vector to make it a unit vector.

We now want to show that the probability that an edge is cut is proportional to it's length. In this way we are more likely to cut the longer edges that contribute more to the value of the cut. So what is the probability that we will cut any particular edge?

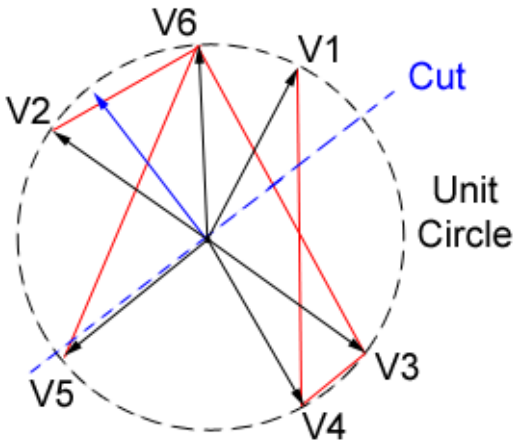


Figure 17.2.3: An example cut across the unit circle.

From Figure 17.2.3 we can see that

$Pr[$ our algorithm cuts edge $(u, v)] = \frac{\theta_{uv}}{\pi}$.

This is for two dimensions, what about n dimensions? If the cutting plane is defined by some random direction in n dimensions, then we can project down to two dimensions and it is still uniformly at random over the n dimensions. So we have the same probability for n dimensions. So the expected value of our solution is

$E[$ our solution $] = \sum_{(u,v)\in E} \frac{\theta_{uv}}{\pi} c_{uv}$

In terms of θ_{uv} the value of our Vector Program is

$Val_{VP} = \sum_{(u,v)\in E} (\frac{1-cos(\theta_{uv})}{2}) c_{uv}$

Now we want to say that $E[$ our solution $]$ is not much smaller than Val_{VP}. So we look at the ratio of $E[$ our solution $]$ to Val_{VP} is not small.

Claim 17.2.3 *For all* θ, $\frac{2\theta}{\pi(1-cos\theta)} \geq 0.878$

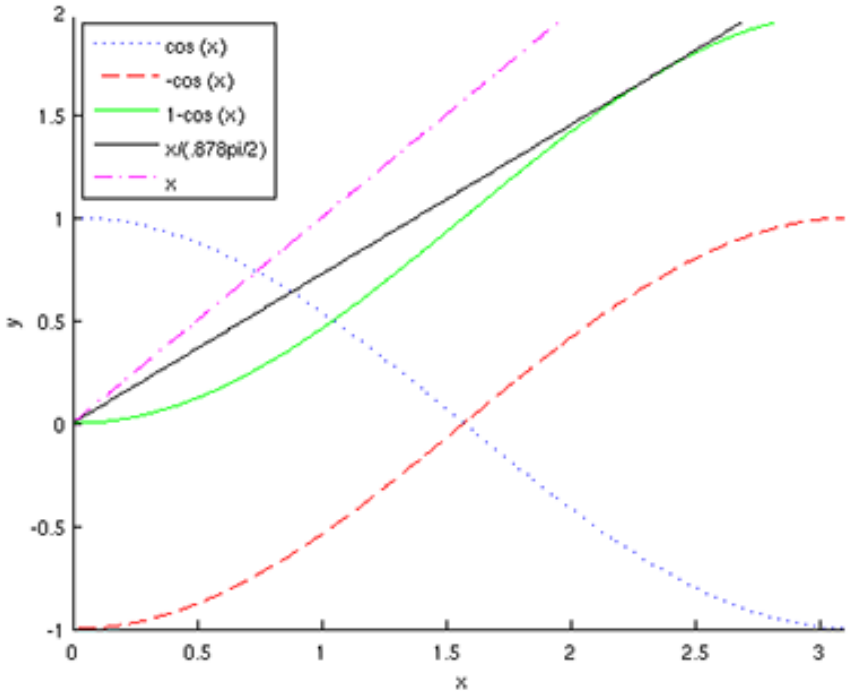


Figure 17.2.4: A visual representation of the .878-approximation to the solution.

As a corollary we have the following theorem. (Note: $1.12 \approx 1/0.878$.)

Theorem 17.2.4 *We get a* 1.12*-approximation.*

17.2.4 Wrap-up

This algorithm is certainly better than the 2-approximation that we saw before. Semi-Definate programming is a powerful technique, and for a number of problems gives us stronger approximations than just Linear Programming relaxations. As it turns out, the Semi-Definate solution for Max-cut is the currently best-known approximation for the problem.

There is reason to believe that unless $P = NP$, you cannot do better than this approximation. This result is highly surprising, given that we derived this number from a geometric argument, which seems like it should have nothing to do with the $P = NP$ problem. The conjecture is that you cannot get a better approximation unless a certain problem can be solved in P, and that certain problem is strongly suspected to be NP-hard. In comes down to a combinatorial problem that in

the end has little to do with the geometric argument we used to get an approximation.

17.3 Streaming Algorithms

Sreaming Algorithms are useful under very extreme conditions, where we do not have very much space or very much time to work with at all. A good example to keep in mind for these kinds of algorithms is analyzing packets of information sent across some network, such as the internet. Say we want to be able to compute statistics about the packets being sent or detect attacks or determine customer usage costs, we need to be able to read the packets as they're being sent over the network and compute statistics. The problem under these conditions is that there is a huge amount of data going by and we don't have much time to read the packet or do any sophisticated analysis on that data.

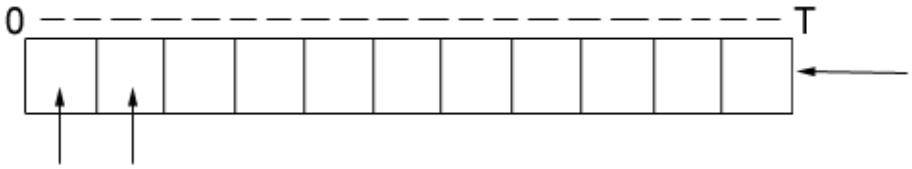


Figure 17.3.5: An example input stream.

The model is that we have a stream of input data, of some very large length T. Ideally, the amount of storage spacewe would like to use would be some constant, or barring that, logarithmic in terms of T. Also, at every point in time, we read some new data from the stream, do some computation, and update our data structures accordingly. We would also like this update time to be sufficiently fast, ideally some constant or amoritized constant time, or some time logarithmic in terms of T.

Storage Space: $\Theta(1)$ or $O(polylog(T))$

Update Time: $\Theta(1)$ or $O(polylog(T))$

Under such strong constraints, there are a lot of problems that we cannot solve. There are fairly simply lower bound arguments that tell us there are certain things you cannot hope to do at all. In these cases, we resort to some kind of approximation. The typical techniques that one would use are some amount of randomization, some amount of sampling, and so on. We will see more as we further explore Streaming Algorithms in this class.

We allow ourselves some randomization, and we allow ourselves some error, but we really want to respect the time and space bounds.

17.3.1 Examples

So what kind of problems might we want to solve here? Ususally one wants to solve problems such as computing statistics on the data, or compressing the data to do calculations on later, and so on. So let's take some specific examples, and see how algorithms can develop for those examples.

Figure 17.3.6: An example $a_t \in [n]$.

Let us use a more specific model of streaming: At time t you get some element $a_t \in [n] \ \ \forall t \in [T]$. As the stream goes by, we get n different kinds of objects, and we see one element at a time. We are going to think of n as very large, where the different kinds of items we can see are also very large, such as distinct source IP addresses on the internet. There are a lot of source IP addresses, and we cannot use a data structure of size n to store all the data we want about the packets. So again, we would like to use some storage space and some update time logarithmic not only in time of T but also in time of n.

Space: $O(polylog(n))$

Time: $O(polylog(n))$

Of course, one amount of information we get from the stream for all these objects is the frequency of each object. So let m_i denote the frequency of the i^th element of $[n]$, $\forall i \in [n]$ let $m_i = |\{j|a_j = i\}|$.

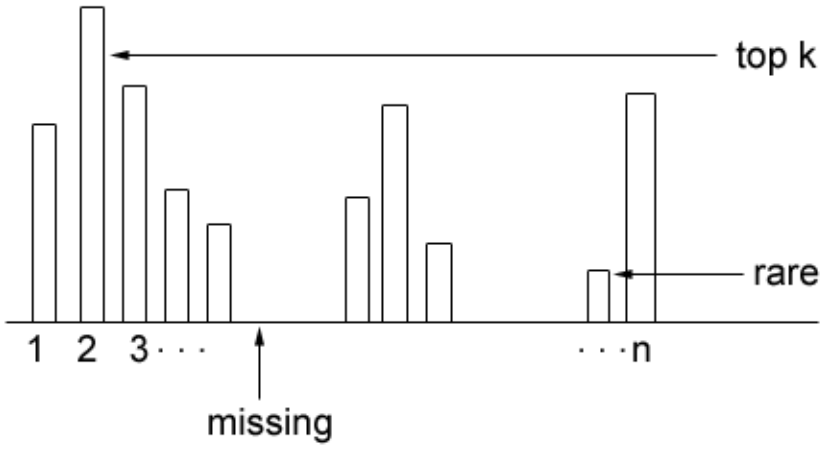


Figure 17.3.7: A sample histogram of frequencies.

So you have these elements with their frequencies, and there are various different kinds of statistics we might want to compute on this data. What are the top k elements, in terms of frequencies? What are the missing elements? What is the number of distinct elements? How many rare elements are

there, where rare is defined as occuring at least once, but having a frequency below some number.

We might want to measure how even the distribution is, as measured by expected value and standard deviation, and these are called frequency moments. The k^{th} freqency moment of the distribution $= \sum_i m_i^k$. The zeroth frequency moment, where m_i^0, is equal to the number of distinct elements. The first frequency moment, where m_i^1, is the sum of the m_i's, and is equal to the number of packets. The second frequency moment, where m_i^2, represents the variance. If everything had equal frequencies, than the sum of squares would be small. If the frequency was uneven, where one element occured much more than the rest, than the sum of squares would be large. Higher moments give us some idea of the skews in the distribution. These kinds of things can be used to determine useful information about the stream, such as frequency of outliers, how typical is a particular stream, etc.

What we are concerned with in streaming algorithms are determining good ways of either exactly computing or estimating these quantities. We will see this as we work through some examples.

17.3.2 Number of Distinct Elements

How would you compute the number of distinct elements exactly? Let us momentarily forget about the polylog requirements on space and time, and think about what is the least amount of space we could use to keep track of these elements. We could keep an array of n bits, to keep track of every element. But what if n is really large? What if it is even larger than T, the length of the stream? For example, if the stream were all of the words in a webpage, and you wanted to keep track of the number of distinct words. The number of distinct words could be much larger than the number of words in the webpage.

So if n is larger than T, can we do something on the order of T rather than n? We can use a hash table of size T, and map every element to its position in the hash table, with at most T distinct elements. We can pick a hash table large enough that we never see a collision, by picking a hash table of $O(T)$ or $O(T^2)$ if you want there to be no collisions. Let's define a hash function, $h : [n] \rightarrow T^2$. Whenever we see a_t, compute $h(a_t)$ and update our array, A, at $A[h(a_t)]$. The number of distinct elements in the stream is equal to the number of nonzero elements in A. We can then calculate the exact number of distinct elements in $O(T^2)$ or the approximate number of distinct elements in $O(T)$.

Can we do any better if we want to solve this problem exactly? It turns out that we cannot do any better than $T \log n$.

Theorem 17.3.1 *we cannot computer the number of distinct elements exactly in $o(T \log n)$ space.*

Proof:

Let's assume that the set of elements that we saw was some $X \subseteq [n]$, where X is a uniformly at random subset of $[n]$ of size T. Then, let's assume that we have some algorithm that lets us compute the number of distinct elements in X with storage $o(T \log n)$, then we give one more element from $[n]$ to this particular algorithm, and if that element was not previously in X, then it increments its counter, and if that element was previously in X, then it does not increment its counter. This algorithm lets us solve the question of memebership into X. Then the amount of storage the

algorithm uses, exactly represents the set x. But how many bits does it need to represent set X? There are $\binom{n}{T}$ possible different X's that we could have, and for any exact representation of these, we need to have at least log of that many bits. So we need exactly $\Omega(T \log n)$ space. ■

17.4 Next Time

Next time we are going to look at approximate ways of getting very close to the number of distinct elements. As long as we allow ourselves a little bit of randomness, and a little bit of error, we can improve these bounds.

CS787: Advanced Algorithms

Scribe: Priyananda Shenoy and Shijin Kong **Lecturer:** Shuchi Chawla
Topic: Streaming Algorithms(continued) **Date:** 10/26/2007

We continue talking about streaming algorithms in this lecture, including algorithms on getting number of distinct elements in a stream and computing second moment of element frequencies.

18.1 Introduction and Recap

Let us review the fundamental framework of streaming. Suppose we have a stream coming in from time 0 to time T. At each time $t \in [T]$, the coming element is $a_t \in [n]$. Frequency for any element i is defined as $m_i = |\{j|a_j = i\}|$. We will see a few problems to solve under this framework. For different problems, we expect to have a complexity of only $O(\log n)$ and $O(\log T)$ on either storage or update time. We should only make a few passes over the stream under these strong constraints.

First problem we will solve in this lecture is getting the number of distinct elements. A simple way to do this is to perform random sampling on all elements, maintain statistics over sampling, and then extraploate real number of distinct elements from statistics. For example, if we pick k of the n elements uniformly at random, and count the number of distinct elements in samples, we can roughly estimate the number of distinct elements over all elements by multiplying the result with n. However, in many cases where elements are unevenly distributed (e. g. some elements dominate), random sampling with a small k will cause much lower estimation than actual number of distinct elements. In order to be accurate we need $k = \Omega(n)$

The second problem, computing second moment of element frequencies depends on accurate estimation of m_i. Again we can apply random sampling in similar way and estimate frequency of each element. But again we will have the problem of inaccuracy in random sampling. Some elements may not be sampled at all.

In order to better resolve those problems, we introduce another two different algorithms which perform better than random sampling based algorithm. Before that, let's define what an *unbiased estimator* is:

Definition 18.1.1 *An unbiased estimator for quantity Q is a random variable X such that $\mathbf{E}[X] = Q$.*

18.2 Number of Distinct Elements

Let's assign c to represent the number of distinct elements. We are going to prove that we can probabilistically distinguish between $c < k$ and $c \geqslant 2k$ by using a single bit of memory. k is related to a hash functions family H where,

$\forall h \in H$, $h : [n] \to [k]$

Algorithm 1:

Suppose the bit we have is b, initially set $b = 0$.

for some $t \in [T]$, if $h(a_t) = 0$, then set $b = 1$.

Let us compute some event probabilities:

$$\mathbf{Pr}[b = 0] = \left(1 - \frac{1}{k}\right)^c$$

$$\mathbf{Pr}[b = 0 | c < k] = \left(1 - \frac{1}{k}\right)^c \geqslant \left(1 - \frac{1}{k}\right)^k \geqslant \frac{1}{4}$$

$$\mathbf{Pr}[b = 0 | c \geqslant 2k] = \left(1 - \frac{1}{k}\right)^c \leqslant \left(1 - \frac{1}{k}\right)^{2k} \leqslant \frac{1}{e^2} \simeq \frac{1}{7.4}$$

Now we have separation between the cases $c < k$ and $c \geqslant 2k$. In the next step, we can use multiple bits to boost this separation.

Algorithm 2:

Maintain x bits $b_0, b_1, \ldots, b_x$ and run $Algorithm1$ independently over each bit.

Next we pick a value between $\frac{1}{4}$ and $\frac{1}{e^2}$, say $\frac{1}{6}$. If $|\{j | b_j = 0\}| > \frac{x}{6}$, output $c < k$, else output $c \geqslant 2k$.

Claim 18.2.1 *The error probability of Algorithm 2 is δ if $x = O(\log \frac{1}{\delta})$.*

Proof: Suppose $c < k$, then the expected number of bits that are 0 in all x bits is atleast $\frac{x}{4}$.

By Chernoff's bound:

$$\mathbf{Pr}\left[\text{actual number of bits that are zero} < \frac{x}{6}\right] =$$
$$\mathbf{Pr}\left[\text{actual number of bits that are } 0 < \left(1 - \frac{1}{3}\right)\frac{x}{4}\right] \leqslant e^{-\frac{1}{2}\frac{1}{3^2}\frac{x}{4}}$$

Using $x = O(\log \frac{1}{\delta})$ gives us the answser with probability $1 - \delta$. Similarly if $c \geqslant 2k$, we can show a similar bound on $\mathbf{Pr}\left[\text{number of actual bits that are } 0 \geqslant \frac{x}{6}\right]$. ∎

We repeat $\log n$ times, and set $\delta = \frac{\acute{\delta}}{\log n}$ for each time. Then, by union bound, the probability that any run fails is $\leq \log n . \frac{\acute{\delta}}{\log n} = \acute{\delta}$.

To get a $(1 + \epsilon)$ approximation, we would need $O\left(\frac{\log n}{\epsilon^2}\left(\log \log n + \log \frac{1}{\acute{\delta}}\right)\right)$ bits

18.3 Computing Second Moment

Recall that the k^{th} moment of the stream is defined as $\mu_k = \sum_{i\in[n]} m_i^k$. We will now discuss an algorithm to find μ_2. This measure is used in many applications as an estimate of how much the frequency varies.

Algorithm 3:
Step 1: Pick a random variable $Y_i \in_{u.a.r} \{-1, 1\} \forall i \in [n]$
Step 2: Let the random variable Z be defined as $Z = \sum_t Y_{a(t)}$
Step 3: Define the random variable X as $X = Z^2$
Step 4: output X

Claim 18.3.1 *X is an unbiased estimator for μ_2; i.e. the expected value of X equals μ_2*

Proof:

Z can be redefined as:

$$Z = \sum_{i\in[n]} m_i Y_i$$
$$X = Z^2 = \sum_i m_i^2 Y_i^2 + 2\sum_{i\neq j} m_i m_j Y_i Y_j$$
$$\mathbf{E}[X] = \sum_i m_i^2 \mathbf{E}\left[Y_i^2\right] + 2\sum_{i\neq j} m_i m_j \mathbf{E}[Y_i Y_j]$$

Since $Y_i \in \{-1, 1\}, Y_i^2 = 1$. Also the second term evaluates to 0 since $Y_i Y_j$ will evaluate to -1 or +1 with equal probability

$$\mathbf{E}[X] = \sum_i m_i^2 = \mu_2 \tag{18.3.1}$$

■

To get an accurate value, the above algorithm needs to be repeated. The following algorithm specifies how Algorithm 3 can be repeated to obtain accurate results.

Algorithm 4:
Step 1:
FOR $m = 1$ to k
Execute Algorithm 3. Let X_i be the output
ENDFOR
Step 2: Calculate the mean of X_i $\overline{X} = \frac{(X_1 + X_2 ... + X_k)}{k}$
Step 3: output $\overline{X}$

The expected value of $\overline{X}$ is taken as the value of μ_2. We will see what the value of k needs to be to get an accurate answer with high probability. To do this, we will apply Chebychev's bound. Consider the expected value of X^2 :

$$\begin{aligned}\mathbf{E}\big[X^2\big] = \mathbf{E}[\sum_i m_i^4 Y_i^4 + 4\sum_{i\neq j} m_i^3 m_j Y_i^3 Y_j + 6\sum_{i\neq j} m_i^2 m_j^2 Y_i^2 Y_j^2 + \\ 12\sum_{i\neq j\neq \acute{i}} m_i^2 m_j m_{\acute{i}} Y_i^2 Y_j Y_{\acute{i}} + 24\sum_{i\neq j\neq \acute{i}\neq \acute{j}} m_i m_j m_{\acute{i}} m_{\acute{j}} Y_i Y_j Y_{\acute{i}} Y_{\acute{j}}]\end{aligned}$$

Assuming that the variables Y_i are 4-way indepedent, we can simplify this to

$$\mathbf{E}\big[X^2\big] = \sum_i m_i^4 + 6\sum_{i\neq j} m_i^2 m_j^2$$

The variance of X_i (as defined in Algorithm 3) is given by

$$\begin{aligned} var[X] &= \mathbf{E}\big[X^2\big] - (\mathbf{E}[X])^2 \\ &= (\sum_i m_i^4 + 6\sum_{i\neq j} m_i^2 m_j^2) - (\sum_i m_i^4 + 2\sum_{i\neq j} m_i^2 m_j^2) \\ &= 4\sum_{i\neq j} m_i^2 m_j^2 \\ &\leq 2(\sum_i m_i^2)^2 \\ &\leq 2\mu_2^2 \\ var[\overline{X}] &= \frac{var[X]}{k} \leq \frac{2\mu_2^2}{k} \end{aligned}$$

By Chebychev's inequality:

$$\begin{aligned}\mathbf{Pr}\big[|\overline{X} - \mu_2| \geq \epsilon\mu_2\big] &\leq \frac{var[\overline{X}]}{\epsilon^2\mu_2^2} \\ &\leq \frac{2\mu_2^2}{k\epsilon^2\mu_2^2} \\ &\leq \frac{2}{k\epsilon^2}\end{aligned}$$

Hence, to compute μ_2 within a factor of $(1\pm\epsilon)$ with probability $(1-\delta)$, we need to run the algorithm $\frac{2}{\delta\epsilon^2}$ times.

18.3.1 Space requirements

Lets analyze the space requirements for the given algorithm. In each run on the algorithm, we need $O(\log T)$ space to maintain Z. If we explicitly store Y_i, we would need $O(n)$ bits, which is too expensive. We can improve upon this by contructing a hash function to generate values for Y_i on the fly. For the above analysis to hold, the hash function should ensure that any group of upto four Y_is are independent(i.e. the hash function belongs to a 4-Way Independent Hash Family). We skip the details of how to construct such a hash family, but this can be done using only $O(\log n)$ bits per hash function.

18.3.2 Improving the accuracy: Median of means method

In Algorithm 4, we use the mean of many trials to compute the required value. This has the disadvantage that some inaccurate trials could adversely affect the solution. So we need a large number of samples linear in $\frac{1}{\delta}$ to get reasonable accuracy. Instead of using the mean, the following procedure can be used to get better results. The idea is to take the median of the means of subsamples. The reason this works better is because the median is less sensitive to the outliers in a sample, as compared to the mean.

Group adjacent X_is into k groups of size $\frac{8}{\epsilon^2}$ each. For each group calculate the mean. Then the expected value of $\overline{X}$ is obtained by taking the median of the k means. The total number of samples of X we use are $k\frac{8}{\epsilon^2}$

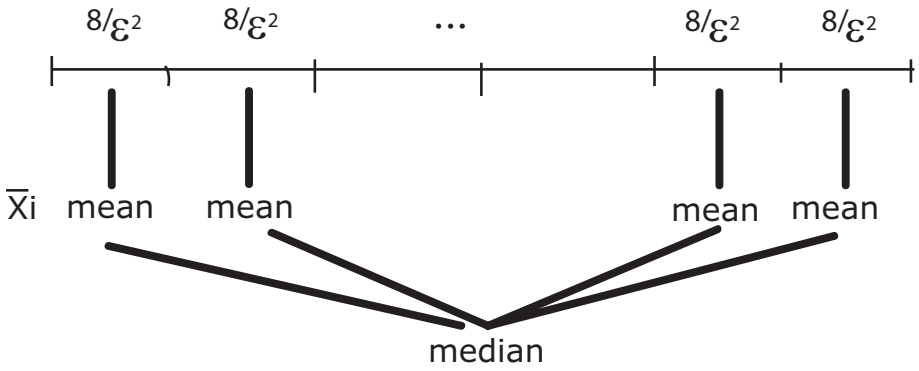


Fig 1: Median of mean method

To see how this improves the accuracy, consider a particular group as shown in the figure. Let $\overline{X_i}$ be the mean of the group.

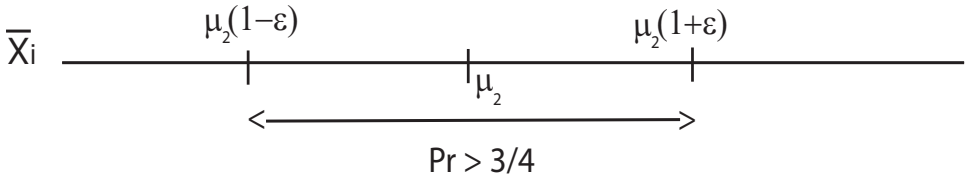


Fig 2: Probability of median falling inside the range

Using Chebychev's Inequality,

$$\begin{aligned}\mathbf{Pr}\left[|\overline{X_i} - \mu_2| > \epsilon\mu_2\right] &\leq \frac{var[X]}{\epsilon^2\mu_2^2} \\ &\leq \frac{2\mu_2^2}{\frac{8}{\epsilon^2}\epsilon^2\mu_2^2} \\ &= \frac{1}{4}\end{aligned}$$

Which essentially means that the probability of a value being outside the interval $[(1-\epsilon)\mu_2, (1+\epsilon)\mu_2]$ is atmost $\frac{1}{4}$. Using Chernoff's bound,

$$\begin{aligned}\mathbf{Pr}[\text{median is outside the interval}] &= \mathbf{Pr}[\text{more than half the samples are outside the interval}] \\ &= e^{-\frac{1}{2}\frac{1}{4^2}k}\end{aligned}$$

If the required probability is δ, we need to pick k so that this value is at most δ, i.e. $k = O(\log \frac{1}{\delta})$ So the number of trials required is $k.\frac{8}{\epsilon^2} = O(\frac{1}{\epsilon^2}log\frac{1}{\delta})$. And the total number of bits used is $O(\frac{1}{\epsilon^2}\log\frac{1}{\delta}(\log T + \log n))$.

CS787: Advanced Algorithms

Scribe: Yiying Zhang, Dan Rosendorf **Lecturer:** Shuchi Chawla
Topic: Streaming algorithms (contd.) and Online algorithms **Date:** October 29, 2007

19.1 Introduction

In the last lecture we talked about streaming algorithms and using unbiased estimators and sampling to solve problems in that setting. Recall the setting of streaming algorithms. We have a stream of elements indexed 0 to T where T is some large number and the elements are drawn from $[n]$ where again n is large. We have limited space and time to work with any element typically to $O(poly(log(n), log(T)))$. Also recall that we define $m_i = |\{j : a_j = i\}|$ to be the frequency of element i.

19.2 More on sampling

For our sampling approach we want to take some k-elements at random from $[n]$ and keep only track of frequencies of the elements in this sample at the end extrapolating data from it. The problem with this approach is that sometimes we have $[n]$ too large to be able to choose a good sample ahead of time since we expect to only see a small subset of $[n]$ to show up in our stream. An example of this are IP addresses on a router. In such a case we really wish to choose our sample only from the distinct elements we actually see.

Let us define $S = \{i \in [n] | m_i \geq 1\}$ to be the set of all elements that occur in our stream. Also $\forall t \leq T$ define $S_t = \{i \in [n] | \exists j \leq t(a_j = i)\}$ the set of all elements seen up to time t. To make the problem slightly easier we will limit ourselves to try and choose an element e_t uniformly at random from S_t.

To do this assume first we have a perfect hash function $h : [n] \to [n]$. Then we can can just let $e_t = argmin_{i \in S_t} h(i)$. Since we don't actually have a perfect hash function though we will need to use a family of hash functions and we will want them to be minwise independent.

Definition 19.2.1 *We say a family of m hash functions $h_i : [n] \to [n]$, $i \leq m$ is minwise independent if $\forall X \subset [n]$ and $x \in X$,*

$$\Pr_{1 \leq i \leq m}[h_i(x) = \min(h(X))] = \frac{1}{|X|}.$$

Unfortunately no such family of size $O(log(n))$ is known so we shall need to settle for a slightly weaker condition called ϵ-minwise independent.

Definition 19.2.2 *We say a family of m hash functions $h_i : [n] \to [n]$, $i \leq m$ is ϵ-minwise independent if $\forall X \subset [n]$ and $x \in X$,*

$$\Pr_{1 \leq i \leq m}[h_i(x) = \min(h(X))] = \frac{1}{|X|}(1 \pm \epsilon).$$

Since such families are known to exist of size $O(log(n)log(1/\epsilon))$ we have the necessary tool we need to choose uniformly at random from S_t. We need only keep track of our element with least hash and at each t check whether $h(a_t) < h(e_{t-1})$ and if so let $e_t = a_t$ otherwise do nothing. If we need to keep a sample of k distinct elements we need only use k different hash functions.

19.3 Online algorithms

In this section we will talk about online algorithms. In the case of online algorithms we again have some data coming to us in a stream but in this case the basic problem is we need to make decisions before we see all of the stream. That is we are trying to optimize but we need to commit to partial steps during the way. To make this concept clearer let us look at an example.

19.3.1 Rent or Buy (Ski rental)

In the *rent or buy* problem sometimes also called the *ski rental* problem we need to figure out whether to rent or buy skis. Suppose we go skiing every winter and thus need skis. We can either buy a pair for some cost B and then use them forever, or we can rent them every winter for cost R but then we will have to rent them again when we want to go skiing next year. If we know how many times we will go skiing in our lifetime it is easy to choose whether to buy or rent just by comparing the total cost of each option, but we do not actually know how many times we will go until it's far too late. Thus we need to come up with another solution.

An easy algorithm is to just rent every winter until we have payed B in rent and then buy the skis once that point comes. While this algorithm doesn't seem very smart the amount we pay is always within a factor of 2 from the optimal solution. Since if we go skiing fewer times then $\frac{B}{R}$ then we have actually chosen the optimal course of action and if not then the optimal solution is to buy the skis in the first place so the optimal cost is B and we have paid exactly $2B$ (assuming $\frac{B}{R}$ is integer).

Note that the 2 here is not an approximation ratio it is something we call a competitive ratio. That is the ratio of our cost to the best cost we can get given that we would actually know everything in advance.

Definition 19.3.1 *Assume we have a problem and some algorithm Alg such that given an instance I of our problem $ALG(I)$ gives the cost of the solution Alg comes up with. Furthermore assume $OPT(I)$ is the cost of the best possible solution of the problem for instance I. Then*

$$\max_I \frac{ALG(I)}{OPT(I)}$$

is called the competitive ratio of the algorithm Alg.

There is also an alternate definition which is sometimes called the strong competitive ratio.

Definition 19.3.2 *Assume the same as above, then we also call r the competitive ratio, if*

$$ALG(I) \leq r\, OPT(I) + c$$

holds for all I and some constant c independent of I. When c is 0, r is the strong competitive ratio.

19.3.2 List Update

One important application area of online algorithms is maintaining data structures efficiently. With incoming search requests and updates over time, we want to develop an algorithm that could dynamically adjust the data structure so as to optimize search time. Think of a binary search tree with n elements, the search time is generally no more than $\log n$. But if we know the frequency of search terms, we could change the structure of the tree so that the more frequent elements are at a higher level, and thus we have a better search term.

Here we study an online algorithm for an easier data structure, *list*. We have a list of size n. For simplicity, we also assume that there are no insertion or deletion to the list. Over time, we have a sequence of requests. At every step, the algorithm gets a request for element i. We pay a cost i to move to position i. There is no cost on moving i forward to any position before its original position. And there is a cost of 1 for a single transposition of i with a position after it. Our goal is to minimize the total cost over time.

We observe that if the distribution of input search terms is known ahead, we can order the list from most frequently requested element to the least frequently requested one. The problem is that we do not know this distribution ahead of time. We need to develop some online algorithm (with no prior knowledge of input distribution) to minimize the total cost.

Algorithm 1 (Move-To-Front): The first intuition is that every time when we see an element we move it to the front of the list. This seemingly naive algorithm turns out to have quite good performance (2-competitive ratio as wee see later).

Algorithm 2 (Frequency-Ordering): In this algorithm, we order elements in decreasing order of frequency of searched elements so far. At each step, when we see an element, we add one to its frequency and if necessary move it forward to a appropriate position. This algorithm seems to be better than the previous one, however it turns out to have poor performance ($\Omega(n)$-competitive ratio as we see later).

Algorithm 3 (Move-Ahead-One): In this algorithm, we move an element forward one position every time we see it. This algorithm also performs poorly ($\Omega(n)$-competitive ratio as we see later).

Now we give proof of the competitive ratios of these three algorithms. The analysis of online algorithms often include the process of a competing adversary trying to find a input on which the algorithm works poorly but there exists a good optimal algorithm (with prior knowledge).

Claim 1: Move-Ahead-One has $\Omega(n)$-competitive ratio.

Proof: Suppose the list is $1, 2, ..., n$. A bad input sequence (one that the adversary comes up) is $n, n-1, n, n-1, n, n-1, ...$ Move-Ahead-One will repeatedly move element n to position $n-1$, then move element $n-1$ to position $n-1$, then move element n to position $n-1$... This gives a cost of n at each step. If we know the input sequence, an optimal solution will be put element n and $n-1$ at the front two positions, giving a cost of $O(1)$ at each step. This shows that Move-Ahead-One has $\Omega(n)$-competitive ratio. ■

Claim 2: Frequency-Ordering has $\Omega(n)$-competitive ratio.

Proof: The bad input sequence the adversary constructs is as follow. In the first $\binom{n}{2}$ steps, element

i is requested $n-i+1$ time in the order from 1 to n, i.e. 1 is requested n times, then 2 is requested $n-1$ times ... n is requested 1 time. Notice that the cost of these steps for Frequency-Ordering is the same as the optimal algorithm with knowledge of input sequence.

¿From then on, every element is requested n times in the order from n to 1. After the first $\binom{n}{2}$ steps, Frequency-Ordering algorithm gives the list $1, 2, ..., n$. Then when element n is requested n times, it moves from the tail to the front of the list, costing totally $n(n+1)/2 = O(n^2)$. When element $n-1$ is requested for the first time, it is at the tail of the list. For n requests of $n-1$, the element moves from tail to front, costing $n(n+1)/2 = O(n^2)$. This analysis applies to all the n elements, giving a total cost of $O(n^3)$. However, if we know the input sequence, we can construct the optimal algorithm to bring element n to front the first time it is requested after the first $\binom{n}{2}$ steps, which cost $O(n)$ for element n. The same analysis apply to the rest of the elements, giving a total cost of $O(n^2)$. This shows that Frequency-Ordering gives $\Omega(n)$-competitive ratio. ■

Claim 3: Move-To-Front has 2-competitive ratio.

Proof: To prove this claim, we first compare Move-To-Front to a static list (say $1, 2, ..., n$), and show the competitive ratio between them is 2.

Here we use an idea similar to amortized analysis. The idea is that we pay some cost for the elements being affected by a move-to-front act. Later on when we search for an element, we use the money we paid for it before. More specifically, when we move an element i at position p to the front, the $p-1$ elements originally before position p are moved backward for 1 position. Thus, we pay \$1 cost for each element that is at a position before p in the Move-To-Front list before moving i and is at a position before i in the static list, i.e. whose value is smaller than i. Notice that we have at most $i-1$ such elements. We get \$1 cost for each element that is at a position before p in the Move-To-Front list before moving i and is at a position after i in the static list. We have at least $p-i$ such elements. The amount money we pay can be also thought of as a potential function, which is always non-negative.

Now let TC_t be the cost of Move-To-Front plus the total amount of money we pay after the first t steps. We have $TC_0 = 0$. For step t, the amount of money we pay is no more than $i-1$ and the amount of money we get is no less than $p-i$. The cost of search element i in the static list is i. Therefore,

$$TC_t - TC_{t-1} \leq p + (i-1) - (p-i) = 2i - 1$$

$$TC_t - TC_{t-1} \leq 2 * (cost\ of\ static\ list\ at\ step\ t)$$

When we sum up $TC_t - TC_{t-1}$ over t, we have

$$TC_t \leq 2 * (total\ cost\ of\ static\ list)$$

This shows that Move-To-Front has 2-competitive ratio as compare to a static list.

Now we generalize this analysis to comparing Move-To-Front to an arbitrary algorithm. Let σ be the sequence of requests. Let $OPT(\sigma)$ be the optimal algorithm for sequence σ. At step t, consider

Move-To-Front's list L_{MTF} and optimal algorithm's list L_{OPT}. We define $\Pi(L_{MTF})$ as the number of different positions in the Move-To-Front list and in the optimal list.

$$\Pi(L_{MTF}) = |\{(x, y)|x \textit{ is ahead of } y \textit{ in } L_{MTF} \textit{ and } y \textit{ is ahead of } x \textit{ in } L_{OPT}\}|$$

At step t, consider the difference between step $t-1$ and t, $\Delta cost_{MTF} + \Delta\Pi(L_{MTF})$. Here, $\Delta\Pi(L_{MTF})$ is the potential function, i.e. the amount of money we pay at step t. Suppose at step t, element i is searched and it is at position p in L_{MTF} and at position p' in L_{OPT}.

Now consider the first case when optimal algorithm doesn't move i or move to a position that is before its original position. The cost we pay for moving i to the front in Move-To-Front is for the elements that are before i in L_{OPT} and are originally before i in L_{MTF}. There are at most $p' - 1$ such elements. We get money for elements that are after i in L_{OPT} and are originally before i in L_{MTF}. There are at least $p - p'$ such elements. Thus, we have

$$\Delta\Pi(L_{MTF}) \leq (p' - 1) - (p - p') = 2p' - p - 1$$

If optimal algorithm exchange i with some position behind, we gain in addition the cost of exchanging positions in the optimal algorithm. Therefore, we have

$$\Delta\Pi(L_{MTF}) \leq 2p' - p - 1 + (\textit{number of paid exchange})$$

And the cost of searching i in L_{MTF} is p. All together, we have

$$\Delta cost_{MTF} + \Delta\Pi(L_{MTF}) \leq p + 2p' - p - 1 + (\textit{number of paid exchange})$$
$$\Delta cost_{MTF} + \Delta\Pi(L_{MTF}) \leq 2p' - 1 + (\textit{number of paid exchange})$$

When we sum this up over t, we have $(\textit{Total cost of } MTF) \leq 2 * (\textit{total cost of } OPT)$. This gives a 2-competitive ratio. ■

In fact, 2-competitive ratio is the best for any deterministic algorithm. However, using randomized algorithms, we can reach a 1.6-competitive ratio. We can briefly reason the advantage of randomized algorithm being that the adversary cannot see the output of a randomized algorithm and thus cannot come up a competing input sequence, while for deterministic algorithms, the adversary can observe the output and come up with a competing input sequence.

Next time, we will see another example of online algorithm, namely the cashing problem. We will develop both a deterministic algorithm and a randomized algorithm for it.

CS787: Advanced Algorithms

Scribe: Dalibor Zeleny and Xiaoyong Chai **Lecturer:** Shuchi Chawla
Topic: Caching Algorithms **Date:** October 31 2007

Today we discussed the issue of caching—a problem that is well suited for the use of online algorithms. We talked about deterministic and randomized caching algorithms, and proved bounds on their competitive ratios.

20.1 Caching

In today's computers, we have to make tradeoffs between the amount of memory we have and the speed at which we can access it. We solve the problem by having a larger but slower *main memory*. Whenever we need data from some page of the main memory, we must bring it into a smaller section of fast memory called the *cache*. It may happen that the memory page we request is already in the cache. If this is the case, we say that we have a cache *hit*. Otherwise we have a cache *miss*, and we must go in the main memory to bring the requested page. It may happen that the cache is full when we do that, so it is necessary to *evict* some other page of memory from the cache and replace it with the page we read from main memory, or we can choose not to place the page we brought from main memory in the cache. Cache misses slow down programs because the program cannot continue executing until the requested page is fetched from the main memory. Managing the cache in a good way is, therefore, necessary in order for programs to run fast. The goal of a *caching algorithm* is to evict pages from the cache in a way that minimizes the number of cache misses. A similar problem, called *paging*, arises when we bring pages from a hard drive to the main memory. In this case, we can view main memory as a cache for the hard drive.

For the rest of this lecture, assume that the cache has a capacity of k pages and that main memory has a capacity of N pages. For example consider a cache with $k = 3$ pages and a main memory with $N = 5$ pages. A program could request page 4 from main memory, then page 1, then 2 etc. We call the sequence of memory pages requested by the program the *request sequence*. One request sequence could be 4, 1, 2, 1, 5, 3, 4, 4, 1, 2, 3 in our case. Initially, the cache could contain pages 1, 2, and 3. When we execute this program, we need access to page number 4. Since it's not in the cache, we have a cache miss. We could evict page 2 from the cache and replace it with page 4. Next, our program needs to access page 1, which is in the cache. Hence, we have a cache hit. Now our program needs page 2, which is not in the cache, so we have a miss. We could choose to evict page 1 from the cache and put page 2 in its place. Next, the program requests page 1, so we have another miss. This time we could decide to just read page 1 and not place it in the cache at all. We continue this way until all the memory requests are processed.

In general, we don't know what the next terms in the request sequence are going to be. Thus, caching is a place where we can try to apply an online algorithm. For a request sequence σ and a given algorithm ALG, we call $\text{ALG}(\sigma)$ the *cost* of the algorithm ALG on request sequence σ, and define it to be the number of cache misses that happen when ALG processes the memory requests and maintains the cache.

20.2 Optimal Caching Algorithm

If we knew the entire request sequence before we started processing the memory requests, we could use a greedy algorithm that minimizes the number of cache misses that occur. Whenever we have a cache miss, we go to main memory to fetch the memory page p we need. Then we look at this memory page and all the memory pages in the cache. We evict the page for which the next request occurs the latest in the future from among all the pages currently in the cache, and replace that page with p in the cache. If the next time p is requested comes after the next time all the pages in the cache are requested again, we don't put p in the cache.

Once again, take our sample request sequence 4, 1, 2, 1, 5, 3, 4, 4, 1, 2, 3 and assume that pages 1 through 3 are in the cache. The optimal algorithm has a cache miss when page 4 is requested. Since page 4 is requested again later than pages 1 through 3 are, the algorithm doesn't put 4 in the cache and doesn't evict any page. The next miss occurs when page 5 is requested. Once again, the next times pages 1 through 3 are requested occur before page 5 is requested the next time, so page 5 is not brought in the cache. The next cache miss happens when page 4 is requested. At that point, page 3 gets evicted from the cache and is replaced with page 4 because pages 1, 2 and 4 get requested again before page 3 does. Finally, the last cache miss occurs when page 3 is requested (last term in the request sequence). At this point, the algorithm could choose to evict page 1 and put page 3 in the cache instead. Thus, the optimal algorithm has 4 cache misses on this request sequence.

Notice that in the case of caching, we have an optimal algorithm assuming that we know the entire input (in our case the input is the request sequence). Last lecture, we discussed the list update problem. We do not have an algorithm that runs in polynomial time and can find an optimal solution to the list update problem, even when it has access to the entire input at the very beginning. This fact makes the analysis of online algorithms for caching easier because we know what algorithm we compare the online algorithms against.

For the remainder of this lecture, we will assume that a caching algorithm always has to bring the memory page in the cache on a cache miss, and doesn't have the option of just looking at it and not putting it in the cache, as it will make our proofs simpler. This version is also much closer to what happens in hardware.

20.3 Deterministic Caching

There are many deterministic online algorithms for caching. We give some examples below. In each of the cases, when a cache miss occurs, the new memory page is brought into the cache. The name of the algorithm suggests which page should be evicted from the cache if the cache is full.

LRU (Least Recently Used) The page that has been in the cache for the longest time without being used gets evicted.

FIFO (First In First Out) The cache works like a queue. We evict the page that's at the head of the queue and then enqueue the new page that was brought into the cache.

LFU (Least Frequently Used) The page that has been used the least from among all the pages in the cache gets evicted.

LIFO (Last In First Out) The cache works like a stack. We evict the page that's on the top of the stack and then push the new page that was brought in the cache on the stack.

The first two algorithms, LRU and FIFO have a competitive ratio of k where k is the size of the cache. The last two, LFU and LIFO have an *unbounded competitive ratio*. This means that the competitive ratio is not bounded in terms of the parameters of the problem (in our case k and N), but rather by the size of the input (in our case the length of the request sequence).

First we show that LFU and LIFO have unbounded competitive ratios. Suppose we have a cache of size k. The cache initially contains pages 1 through k. Also suppose that the number of pages of main memory is $N > k$. Suppose that the last page loaded in the cache was k, and consider the request sequence $\sigma = k+1, k, k+1, k, \ldots, k+1, k$. Since k is the last page that was put in the cache, it will be evicted and replaced with page $k+1$. The next request is for page k (which is not in the cache), so we have a cache miss. We bring k in the cache and evict $k+1$ because it was brought in the cache last. This continues until the entire request sequence is processed. We have a cache miss for each request in σ, whereas we have only one cache miss if we use the optimal algorithm. This cache miss occurs when we bring page $k+1$ at the beginning and evict page 1. There are no cache misses after that. Hence, LIFO has an unbounded competitive ratio.

To demonstrate the unbounded competitive ratio of LFU, we again start with the cache filled with pages 1 through k. First we request each of the pages 1 through $k-1$ m times. After that we request page $k+1$, then k, and alternate them m times. This gives us $2m$ cache misses because each time k is requested, $k+1$ will be the least frequently used page in the cache so it will get evicted, and vice versa. Notice that on the same request sequence, the optimal algorithm makes only one cache miss. This miss occurs during the first request for page $k+1$. At that point, the optimum algorithm evics page 1 and doesn't suffer any cache misses afterwards. Thus, if we make m large, we can get any competitive ratio we want. This shows that LFU has an unbounded competitive ratio.

We now show that no deterministic algorithm can have a better competitive ratio than the size of the cache, k. After that, we demonstrate that the LRU algorithm has this competitive ratio.

Claim 20.3.1 *No deterministic online algorithm for caching can achieve a better competitive ratio than k, where k is the size of the cache.*

Proof: Let ALG be a deterministic online algorithm for caching. Suppose the cache has size k and that it currently contains pages 1 through k. Suppose that $N > k$. Since we know the replacement policy of ALG, we can construct an adversary that causes ALG to have a cache miss for every element of the request sequence. To do that, we simply look at the contents of the cache at any time and make a request for the page in $\{1, 2, \ldots, k+1\}$ that is currently not in the cache.

The only page numbers requested by the adversary are 1 through $k+1$. Thus when the optimal algorithm makes a cache miss, the page it evicts will be requested no sooner than after at least k other requests. Those requests will be for pages in the cache. Thus, another miss will occur after

at least k memory requests. It follows that for every cache miss the optimal algorithm makes, ALG makes at least k cache misses, which means that the competitive ratio of ALG is at least k. ■

Claim 20.3.2 *LRU has a competitive ratio of k.*

Proof: First, we divide the request sequence σ into phases as follows:

- Phase 1 begins at the first page of σ;
- Phase i begins at the first time we see the k-th distinct page after phase $i-1$ has begun.

As an example, suppose $\sigma = 4, 1, 2, 1, 5, 3, 4, 4, 1, 2, 3$ and $k = 3$. We divide σ into three phases as in Figure 20.3.1 below. Phase 2 begins at page 5 since page 5 is the third distinct page after phase 1 began (pages 1 and 2 are the first and second distinct pages, respectively).

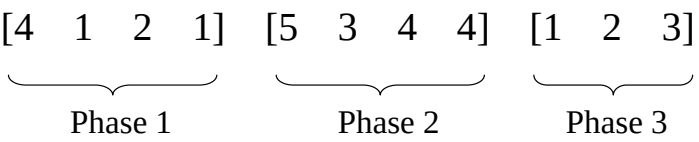


Figure 20.3.1: Three phases of sequence σ.

Next, we show that OPT makes at least one cache miss each time a new phase begins. Denote the j-th distinct page in phase i as p_j^i. Consider pages p_2^i-p_k^i and page p_1^{i+1}. These are k distinct pages by the definition of a phase. Then if none of the pages p_2^i-p_k^i incur a cache miss, p_1^{i+1} must incur one. This is because pages p_2^i-p_k^i and p_1^{i+1} are k distinct pages, page p_1^i is in the cache, and only k pages can reside in the cache. Let N be the number of phases. Then we have $\text{OPT}(\sigma) \geq N - 1$. On the other hand, LRU makes at most k misses per phase. Thus $\text{LRU}(\sigma) \leq kN$. As a result, LRU has a competitive ratio of k. ■

This proof relies on the fact that on a page fault, OPT (and any other caching algorithm) has to bring the page from memory into the cache, and evict another page if the cache is full. If this were not the case, we could only demonstrate a competitve ratio of $2k$. It is also possible to show that this bound on the competitive ratio is tight.

20.3.1 Deterministic 1-Bit LRU

As a variant of the above LRU algorithm, the 1-bit LRU algorithm (also known as the marking algorithm) associates each page in the cache with 1 bit. If a cache page is recently used, the corresponding bit value is 1 (marked); otherwise, the bit value is 0 (unmarked). The algorithm works as follows:

- Initially, all cache pages are unmarked;
- Whenever a page is requested:
    - If the page is in the cache, mark the page;

- Otherwise:
  - If there is at least one unmarked page in the cache, evict an arbitrary unmarked page, bring the requested page in, and mark it;
  - Otherwise, unmark all the pages and start a new phase.

Following the proof of Claim 20.3.2, we can easily see that the above deterministic 1-bit LRU algorithm also has a competitive ratio of k.

20.4 Randomized 1-Bit LRU

Given a deterministic LRU algorithm, an adversary can come up with a worst-case scenario by always requesting the page which was just evicted. As a way to improve the competitive ratio, we consider a randomized 1-bit LRU algorithm in which a page is evicted randomly. Before we go into the algorithm, we shall define the competitive ratio for a randomized online algorithm.

Definition 20.4.1 *A randomized online algorithm* ALG *has a* competitive ratio r *if for* $\forall\sigma$,

$$E[\mathrm{ALG}(\sigma)] \leq r \cdot \mathrm{OPT}(\sigma) + c,$$

where c is a constant independent of σ.

Note that the competitive ratio of a randomized algorithm is defined over the expected performance of the algorithm rather than the worst-case performance.

The randomized 1-bit LRU algorithm is rather simple: at every step, run the 1-bit LRU algorithm and evict an unmarked page uniformly at random when necessary.

Claim 20.4.2 *The randomized 1-bit LRU algorithm has a competitive ratio of $2H_k$, where H_k is the k-th harmonic number, which is defined as*

$$H_k = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{k} \in (\log k, \log(k) + 1).$$

Proof: Let us first analyze the expected number of cache misses in phase i. Recall that at the end of phase $i-1$, there are k pages in the cache. Let m_i denote the number of "new" pages in phase i, *i.e.*, those pages that were not requested in phase $i-1$. Call the rest of the pages in phase i "old" pages. Requesting a new page causes an eviction while requesting an old page may not. Assume for simplicity that the m_i new pages appear the first in the request sequence of phase i. Actually, one should convince oneself that this assumption indicates the worst-case scenario (think about how the number of cache misses changes if any one of these new page requests is delayed). Once the m_i new pages are requested, they are marked in the cache. The remaining $k - m_i$ pages in the cache are old pages. Now let us consider the probability of a cache miss when we request an old page for the first time, *i.e.*, when we request the (m_i+1)-th distinct page in phase i. This old page by definition was in the cache at the end of phase $i-1$. When requested, however, it may not be in the cache since it can be evicted for caching one of the m_i new pages. Then,

$$Pr[(m_i+1)\text{-th distinct page suffers a cache miss}] = \frac{\binom{k-1}{k-m_i}}{\binom{k}{k-m_i}} = \frac{m_i}{k}.$$

By the same reasoning, we have

$$Pr[(m_i+2)\text{-th distinct page suffers a cache miss}] = \frac{\binom{k-2}{k-m_i-1}}{\binom{k-1}{k-m_i-1}} = \frac{m_i}{k-1},$$

and so on.

Then the expected total number of cache misses in phase i is at most

$$m_i + \frac{m_i}{k} + \frac{m_i}{k-1} + \cdots + \frac{m_i}{k-(k-m_i)+1} \leq m_i H_k.$$

Let N be the number of phases. Then

$$E[\text{RAND-1-BIT-LRU}(\sigma)] \leq H_k \sum_{i=1}^{N} m_i. \tag{20.4.1}$$

Let us now analyze the number of cache misses of $\text{OPT}(\sigma)$. Note that requesting a new page p in phase i may not cause an eviction. This is because OPT may choose to fix p in the cache when p was requested in some earlier phase. In contrast, the randomized algorithm unmarks all pages in the cache at the beginning of phase i, and evicts randomly an unmarked page in case of a cache miss. Therefore, at the end of phase i, p cannot be in the cache. Although we cannot bound the number of cache misses in a single phase, we can bound the number in two consecutive phases. Let n_i be the number of cache misses of OPT in phase i. Since in total there are $k+m_i$ distinct pages in phases $i-1$ and i, at least m_i of them cause a miss. Thus we have

$$n_{i-1} + n_i \geq m_i,$$

and

$$\text{OPT}(\sigma) \geq \frac{1}{2} \sum_{i=1}^{N} m_i. \tag{20.4.2}$$

Combining Eq. (20.4.1) and Eq. (20.4.2), we have

$$E[\text{RAND-1-BIT-LRU}(\sigma)] \leq 2H_k \times \text{OPT}(\sigma).$$

In other words, the randomized 1-bit LRU algorithm is $2H_k$-competitive. ■

Claim 20.4.3 *No randomized online algorithm has a competitive ratio better than H_k.*

Proof: See the next lecture note. ■

CS787: Advanced Algorithms

Scribe: Chris Hinrichs, Xiaoyong Chai
Lecturer: Shuchi Chawla
Topic: Caching and K-Server problem
Date: 11/5/2007

21.1 K Server Problems

Suppose instead of k pages of memory there are k entities which can serve requests. Requests arrive in sequence, and the sequence is not known ahead of time. Each request comes with a location, and can only be served at that location. In order to serve a request, a server must be at the request location. The locations reside within a metric space M, which uses metric $d(x, y)$ as a distance metric. Recall that as a metric, $d(x, y)$ has the following properties: $d(x, y) \geq 0$ $d(x, y) = d(y, x)$ $d(x, y) \leq d(x, z) + d(z, y)$

Now suppose that we want to minimize the distance travelled by servers while servicing some n requests. Such a problem is a K Server problem.

One of the first things we can notice about K Server problems is that Caching is an instance of a K Server problem, where the distance metric has unit value for all pairs, except for the distance from x to x, which has 0 distance. The k pages of local memory are represented by k servers, which must be moved to the page requested.

An extension of Caching is the Weighted Caching problem. In this problem, each page of memory has a specific cost or weight, which is incurred when the page is brought into memory. In this case, the optimal offline algorithm has to do something more complicated than simply evicting from memory the page which will be used the furthest in the future, as in the unweighted case. The optimal algorithm must do a kind of dynamic programming in order to determine the least expensive way to allocate its resources. if X is a configuration of servers, i.e. a setting of the location of each server, then we can define $OPT_i(X)$ as the cost of serving i requests, ending in configuration X. This way,

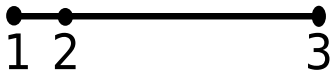
$$OPT_i(X) = \min_{Y \in r_i} OPT_{I-1}(Y) + d(Y, X)$$

Note that $d(x, y)$ is a distance between individual points in a metric space; however, $d(X, Y)$ is a translation of servers in X so that they end up in the positions defined in Y which incurs the minimum cost. Since each server in X must end up as a server in Y, this problem can be though of as a perfect matching problem, with minimum cost.

The best known algorithm for this problem takes $O(n^{2k})$ time. As yet there are no known efficient algorithms for this, (the optimal offline algorithm,) nor are there known online algorithms with good competetive ratios. (We will see later how good they can be.)

As an example, one algorithm for solving K Server problems is what we will call a "Greedy" algorithm. This algorithm simply finds the nearest server to each request, and moves it to the request location. This has the immediate benefit of minimizing the cost of moving a server to the

location of the request, but in the long term it may not benefit from strategically placed servers closer to later requests. As an example of a problem for which Greedy does poorly, consider the following scenario: all points in the space M lie on a line. Requests arrive for the point labelled "3", followed by a long sequence of requests for the points labelled "1" and "2", alternating between them.



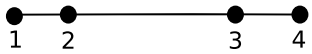
Clearly, if the requests for points 1 and 2 are serviced by the server nearest to them, the one at point 3 will stay there, and the other one will bounce back and forth between points 1 and 2, whereas the optimal algorithm would put both servers on 1 and 2 so that there would be no need to relocate any more servers.

Another algorithm which does slightly better is one we will call "Balance". Balance keeps track of the total distance each server has moved, and chooses a server to move which, after having moved (hypothetically) to the request location will have moved the minimum total distancei of any server. Given the same scenario which we gave Greedy, Balance will eventually decide to move the other server to be closer to the other, so that all requests after that will not require a server to move. However, it can be shown that Balance is only k competetive if $n = k + 1$.

Yet another algorithm is one called "Follow-OPT". Suppose we define $OPT_i(X)$ as the total distance travelled by servers following an OPT strategy for the first i requests in a series, and ending in configuration X. Follow-OPT computes $\text{argmin}_X \, OPT_i(X)$ We will show that Follow-OPT also has an unbounded competetive ratio.

An instance of a K Server problem which causes Follow-OPT to perform poorly is the following: Suppose again that all points in the space M lie on a line. There are points labelled "1", "2", "3", and "4".

Suppose that there are 3 servers, and the following sequence of requests arrives: 1 2 3 4 3 4. For this sequence, OPT will put a server on each of 3 and 4, and if Follor-OPT sees a sequence that starts this way, it will too. However, for the sequence 1 2 3 4 3 4 1 2 1 2, OPT will put servers on 1 and 2, and use the 3rd to oscillate between 3 and 4. As a pattern grows in this way, Follow-OPT will move servers across the large gap between 2 and 3 an unbounded number of times as it switches between optimal strategies, but OPT will simply place servers on 1 and 2 or 3 and 4, and will not have to cross the large gap after that.



The best known algorithm for this type of problem is the Work Function algorithm. (WF) WF computes the configuration X which is generated by OPT_i and which is closest to the previous configuration X_{i-1}. We will see later that WF is 2k-1 competetive.

CS787: Advanced Algorithms

Scribe: Mayank Maheshwari and Priyananda Shenoy **Lecturer:** Shuchi Chawla
Topic: k-server Problem(continued); Online learning **Date:** 11/07/2007

22.1 Introduction and Recap

In the last lecture we were discussing the k-server problem, which is formulated as follows: Let M be a metric space with n points. There are k servers which can service requests. When a request occurs from a particular location, a server must be placed at that location. The problem is to minimize the total distance traveled by the servers. In this class we will see one algorithm for this problem, the Work Function Algoritm.

22.2 Work Function Algorithm

Let X be a configuration of the servers; i.e. the set of locations the servers are located. Then we define $OPT_i(X)$ as the optimal cost of serving requests $\sigma[1:i]$ and ending at configuration X. Then the Work Function algorithm, for servicing ith request, chooses the configuration X_i using the following rule:

$$X_i = \arg\min_X \{OPT_i(X) + d(X_{i-1}, X)\}$$

Theorem 22.2.1 *WFA is (2k-1) competitive*

Proof: We will be proving that for any sequence of requests σ

$$WFA(\sigma) \leqslant (2k-1)OPT(\sigma) + k^2 D$$

where D is the diameter of the metric space.

Let $X_0, X_1, ..., X_t$ be the configurations chosen by the WF Algorithm for servicing requests $r_0, r_1, ..., r_t$. We are interested in seeing how $OPT_i(X_i)$ evolves over time. So when the ith request r_i comes, we keep track of the difference in the optimal cost of servicing i requests and ending up at the configuration X_i chosen by our algorithm, and the optimal cost of servicing $i-1$ requests and ending up at the previous configuration X_{i-1}. This difference is given by

$$OPT_i(X_i) - OPT_{i-1}(X_{i-1})$$

Adding and subtracting $OPT_i(X_{i-1})$ we get,

$$(OPT_i(X_i) - OPT_i(X_{i-1})) + (OPT_i(X_{i-1}) - OPT_{i-1}(X_{i-1})) \tag{1}$$

The term $(OPT_i(X_{i-1}) - OPT_{i-1}(X_{i-1}))$ keeps track of how the optimal cost of ending at X_{i-1} changes due to the ith request. The term $(OPT_i(X_i) - OPT_i(X_{i-1}))$ tells us how our movement from X_{i-1} to X_i changes the potential function.

In the offline case, we used a dynamic programming approach to arrive at an optimal solution. The recurrence used by the DP was

$$OPT_i(X) = \min_{Y \ni r_i} OPT_{i-1}(Y) + d(X, Y)$$

Since $Y \ni r_i$,we have $OPT_{i-1}(Y) = OPT_i(Y)$. By taking $X = X_{i-1}$, we see that $Y = X_i$. This gives us

$$\begin{aligned} OPT_i(X_{i-1}) &= OPT_i(X_i) + d(X_i, X_{i-1}) \\ OPT_i(X_i) - OPT_i(X_{i-1}) &= -d(X_i, X_{i-1}) \end{aligned}$$

Substituting this in (1) we get,

$$(OPT_i(X_{i-1}) - OPT_{i-1}(X_{i-1})) - d(X_i, X_{i-1})$$

Summing over all the steps across the evolution,

$$\sum_i \{OPT_i(X_{i-1}) - OPT_{i-1}(X_{i-1})\} - \sum_i d(X_i, X_{i-1})$$

This sum is equal to the optimal cost of servicing all the requests and ending at the last configuration chosen by the WF Algorithm X_t. And $\sum_i d(X_i, X_{i-1})$ is nothing but the total cost incurred by our algorithm. Therefore we get

$$\begin{aligned} \sum_i \{OPT_i(X_{i-1}) - OPT_{i-1}(X_{i-1})\} - WFA(\sigma) &= OPT_t(X_t) \\ &\geqslant OPT(\sigma) \\ OPT(\sigma) + WFA(\sigma) &\leqslant \sum_i \{OPT_i(X_{i-1}) - OPT_{i-1}(X_{i-1})\} \end{aligned}$$

To get a bound on the RHS expression, let us define the "extended cost" EXT_i as

$$EXT_i = \max_X \{OPT_i(X) - OPT_{i-1}(X)\}$$

Then

$$WFA(\sigma) + OPT(\sigma) \leqslant \sum_i EXT_i \tag{2}$$

We will now try to get a bound on EXT_i's. First we will try to get a bound on this term in the special case where $n = k + 1$, and then we will get a bound for the general case.

Claim 22.2.2 *if $n = k + 1$, then WFA the competitive ratio k*

We will prove this claim using a potential function argument. Let us define a potential function Φ as

$$\Phi_i = \sum_X OPT_i(X)$$

with $\Phi_0 = 0$. Let us consider the increase in the potential function value for a particular request i. From the way we defined EXT_i, this increase is atleast EXT_i.

$$\Phi_i - \Phi_{i-1} \geqslant EXT_i$$

Summing over all requests, we get

$$\Phi_t \geqslant \sum_i EXT_i \tag{3}$$

By definition we have,

$$\Phi_t = \sum_X OPT_t(X)$$

Since there are only $k + 1$ configurations possible, we can get an upper bound for the RHS value

$$\Phi_t \leqslant (k+1) \max_X OPT_t(X)$$

We know that $OPT(\sigma) = \min_X OPT_t(X)$. We can get an upper bound on $\max_X OPT_t(X)$ using the fact

$$\max_X OPT_t(X) \leqslant OPT(\sigma) + \max_{X,\acute{X}} d(X, \acute{X})$$

We know that $\max_{X,\acute{X}} d(X, \acute{X}) \leqslant kD$ where D is the diameter of the space. So we get

$$\Phi_t \leqslant (k+1)(OPT(\sigma) + kD)$$

Substituting this in (3)

$$\begin{aligned}\sum_i EXT_i &\leqslant \Phi_t \\ &\leqslant (k+1)OPT(\sigma) + (k+1)kD\end{aligned}$$

Using this in (2), we get

$$WFA(\sigma) \leqslant kOPT(\sigma) + (k+1)kD$$

Hence our algorithm is k-competitive for the case where $n = k+1$. We will now try to get a bound for the general case.

Claim 22.2.3 *For any values of k and n, WFA is (2k-1) competitive*

To show this, we will do the following construction. Let M be our metric space. Then we will construct a metric space $\overline{M}$ in the following manner. For every point a in M, we create a point $\overline{a}$, called the *antipode* of a , such that $d(a, \overline{a}) = D$. The distance between antipodes is equal to the distance between their corresponding original points; i.e. $d(a, b) = d(\overline{a}, \overline{b})$. The distance between a point and an antipode of another point is given by $d(a, \overline{b}) = D - d(a, b)$. It is easy to see that this construction yields a valid metric space.

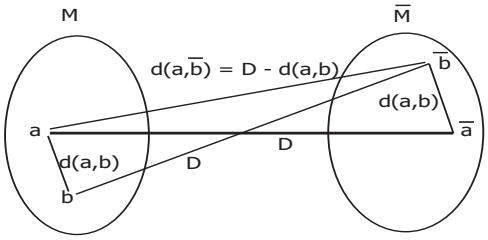


Fig 1: The metric spaces M and $\overline{M}$

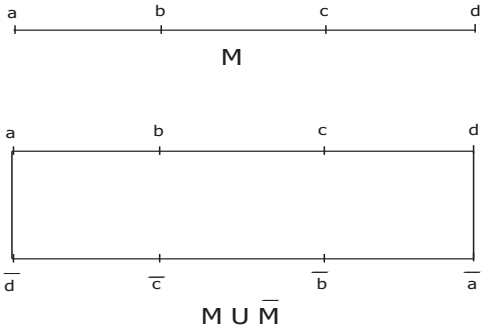


Fig 2: An example where all points lie on a line.

We transform the space M to $M \cup \overline{M}$. The problem doesn't change, since none of the newly added points issue requests.

We will consider the "worst case scenario" which maximizes the distance covered at step i. We will claim, and later give a proof sketch that this happens when a request comes from location r_i and all servers are at its antipode $\overline{r_i}$.

Claim 22.2.4 $arg\max_X\{OPT_i(X) - OPT_{i-1}(X)\} = (\overline{r_i}, \overline{r_i}, ..., \overline{r_i})$

Assuming that this claim holds good, we will proceed with obtaining an upper bound for $\sum_i EXT_i$.

Let $Y_i = (\overline{r_i}, \overline{r_i}, ..., \overline{r_i})$ be the "worst case" configuration. Then we have

$$\begin{aligned} EXT_i &= OPT_i(Y_i) - OPT_{i-1}(Y_i) \\ \sum_i EXT_i &= \sum_i OPT_i(Y_i) - \sum_i OPT_{i-1}(Y_i) \end{aligned} \tag{4}$$

Let us examine the sequence in which the optimal algorithm OPT services the requests.

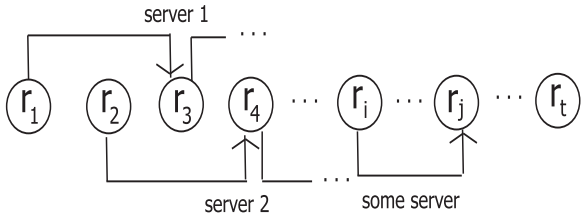


Fig 3: The sequence of how OPT services the requests

If OPT moves a server from r_i to r_j , we will match i with j. Then the following holds true

$$OPT_i(Y_i) \leqslant OPT_{j-1}(Y_j) + kd(r_i, r_j)$$

This works for most of the r_i's except the k r_i's at the end where the servers stop. For these servers

$$OPT_i(Y_i) \leqslant OPT + kD$$

Summing over all i

$$\begin{aligned}\sum_i OPT_i(Y_i) &\leqslant k(OPT + kD) + \sum_j \{OPT_{j-1}(Y_j) + kd(r_i, r_j)\} \\ &\leqslant kOPT + k^2D + \sum_j OPT_{j-1}(Y_j) + k\sum_j d(r_i, r_j)\end{aligned}$$

but $\sum_j d(r_i, r_j) \leqslant OPT$ so

$$\begin{aligned}\sum_i OPT_i(Y_i) - \sum_j OPT_{j-1}(Y_j) &\leqslant kOPT + k^2D + kOPT \\ \sum_i EXT_i &\leqslant 2kOPT + k^2D\end{aligned}$$

Substituting this in (2)

$$\begin{aligned}WFA(\sigma) + OPT(\sigma) &\leqslant 2kOPT + k^2D \\ WFA(\sigma) &\leqslant (2k-1)OPT + k^2D\end{aligned}$$

Hence the Work Function Algorithm is $(2k-1)$-competitive.

■

We will now give a brief sketch of proving the claim we made in the general case analysis.

Proof of Claim 22.2.4(Sketch):

For simplicity, we will assume that there is only one server. Similar analysis can be done for multiple servers as well.

Suppose we get ith request r_i from location a. Let the previous request r_{i-1} be from location b. Since there is only one server, the configuration X is just a single location where the server is located, say x.

$$
\begin{aligned}
&OPT_i(x) - OPT_{i-1}(x) \\
&= (OPT_{i-1}(b) + d(a,b) + d(a,x)) - (OPT_{i-1}(b) + d(b,x)) \\
&= d(a,x) - d(b,x) + d(a,b)
\end{aligned}
$$

if $x = \overline{a}$ then,

$$d(a,\overline{a}) - d(b,\overline{a}) = D - (D - d(a,b)) = d(a,b)$$

So we see that the distance is maximized when the server was located at position $\overline{a}$.

■

22.3 Metric Task System Problem

The Metric Task system problem is a generalization of the k-server problem. The Work Function Algorithm is used to solve this problem and it gives a competitive ratio of $2N - 1$ where N is the number of points in the metric space.

Given: A Metric space M where $|M| = N$ and one server. M has a distance function d which satisfies the following properties:

$\forall x, y, z \in M$

- $d(x,y) \geq 0$
- $d(x,y) = d(y,x)$
- $d(x,y) \leq d(x,z) + d(z,y)$

The setting can be viewed as the server needs to execute tasks coming in a stream on machines located at points in the metric space M. A task can be well suited to a machine and so the cost of executing it at that point is low. We can also move the server to machine at another point in M which is suited to the task but we incur a cost in moving the task from one point to another. So we get a task and the server should execute it on best machine possible without incurring too much cost in moving from one machine to another.

Goal: The server needs to execute the incoming tasks on the machines located at the $|M|$ points such that the cost of executing those tasks is minimized.

To solve the *k-server problem* using metrical task system, we let each state of the system correspond to one possible configuration of the k-server problem. Therefore, the number of states is $\binom{N}{k}$. This is a more general problem and we will not delve into the intricacies of it. But the Work-Function Algorithm covered in the previous section can be used to solve the Metrical Task System problem and it gives a competitive ratio of $2N - 1$.

22.4 Online Learning

The online learning algorithms are a method to solve class of prediction problems. The problem is generally characterized by number of experts (say) n and every expert has a cost vector corresponding to the prediction associated with it. The objective is to maximize the cost associated with the cost vectors of an expert and do as well as the expert. The algorithm at every step needs to pick an expert and then the cost vector (consisting of 0's and 1's is revealed). This is similar to the Metrical Task System problem with a few differences,namely, there are two underlying assumptions for this class of problems:

1. It is free to move from one point (expert) to another i.e. the algorithm doesn't incur any cost.
2. The algorithm ALG is not compared with the optimal OPT in hindsight.

Goal: The algorithm should do nearly as well as the expert.

We will consider a few specific problems in this case. The first one is the Pick-a-Winner problem.

22.4.1 Pick-a-Winner Problem

In this problem, we get profits instead of costs and we need to maximize it.

Problem Statement: There are n different buckets and an adversary throws coins in the bucket. In this game, we need to pick a bucket and if the adversary oblivious of our choice throws the coin in that bucket, we get the coin. The game gets over when there are maximum d coins thrown in a bucket.

This is a case of oblivious adversary in which we take the help of randomization to work to our advantage. If we pick the buckets deterministically, the adversary would know our choice and would always throw the coins in the other buckets.

So if we pick the bucket uniformly at random, the expected number of coins that we get

$\mathbf{E}$[number of coins won]$= d \cdot \mathbf{Pr}$[picking a bucket u.a.r.] $= \frac{d}{n}$.

With this approach, we get a competitive ratio of $\frac{1}{n}$.

22.4.1.1 Multiplicative Updates Method(Exponential weights)

In this method of solving the Pick-a-Winner problem, we increase the probability of picking a bucket by a factor of 2 each time the adversary throws a coin in that bucket. So the algorithm can be seen as

Algorithm

- Start with, for all buckets i, $w_i = 1$ where w_i is the weight of bucket i.
- Whenever a coin falls into a bucket, double its weight.
- At every step, pick an i with probability proportional to w_i.

- Game ends when a bucket has d coins.

The adversary wants to end the game quickly and to distribute the coins over the n buckets so that the profit the algorithm achieves is not high.

Analysis

Let us say after t coin tosses, the total weight of the n buckets in the problem be $W_t = \sum_i w_i(t)$. Let the expected gain be F_t and the probability of success or winning the game is $\frac{w_i}{W_t}$ where i is the bucket we choose in round $t+1$. Now let us see how W_t evolves in the algorithm.

$W_{t+1} = W_t + w_i = W_t + W_t \cdot F_t$

So $W_{t+1} = W_t(1+F_t)$. Whenever there is a large increase in weight of the bucket, our gain is large.

At the end of the game i.e. when a bucket receives d coins, the W_{final} can be written as, where $\sum_t F_t$ is our total expected gain:

$2^d \leq W_{final} = n\prod_t(1+F_t)$

Taking natural logs on both sides,

$$\begin{aligned} d\log_e 2 &\leq \log_e n + \sum_t \log_e(1+F_t) \\ &\leq \ln n + \sum_t F_t \qquad [\ln(1+x) \leq x] \end{aligned}$$

This implies $\sum_t F_t \geq d\ln 2 - \ln n \approx 0.7d - \ln n$.

We can see that the algorithm gives a profit with a factor close to the maximum profit but we lose an additive term. To minimize the additive loss in our profit, we can modify the algorithm in a way that instead of doubling the weight of a bucket each time a coin falls in it, we increase the weight by a factor of $(1+\epsilon)$. So we can rewrite all the equations in the analysis with the factor of ϵ introduced. So now $W_{t+1} = W_t(1+\epsilon F_t)$ and $(1+\epsilon)^d \leq W_{final} = n\prod_t(1+\epsilon F_t)$

$$\begin{aligned} d\log(1+\epsilon) &\leq \log n + \sum_t \log(1+\epsilon F_t) \\ &\leq \log n + \sum_t \epsilon F_t \end{aligned}$$

$$\begin{aligned} ALG = \sum_t F_t &\geq \frac{d}{\epsilon}\log(1+\epsilon) - \frac{1}{\epsilon}\log n \qquad [\log(1+\epsilon) \geq \epsilon - \frac{\epsilon^2}{2}] \\ &\geq (1-\frac{\epsilon}{2})d - \frac{1}{\epsilon}\log n \end{aligned}$$

Now if $\epsilon = \sqrt{\frac{\log n}{d}}$, then the $ALG \geq d - \frac{3}{2}\sqrt{d\log n}$.

The additive factor which we lose here is known as the *regret bound* or the *regret of algorithm* which is the factor the algorithm looses in order to converge to the profit of the expert. We will see more on the regret bounds in the next lecture.

References

[1] Yair Bartal Lecture Notes on Online computation and network Algorithms. *http://www.cs.huji.ac.il/ algo2/on-line/on-line-course.html.*

CS787: Advanced Algorithms

Scribe: Archit Gupta, Pavan Kuppili **Lecturer:** Shuchi Chawla
Topic: Weighted Majority, Mistake Bound Model, and Winnow Algorithm **Date:** 11/12/2007

23.1 Prediction Problem

Last class we looked at the technique of maximizing reward using Multiplicative Update technique. Today we look at a similar problem for online learning, where our goal is to minimize costs.
Let there be n experts giving predictions whether it'll rain or not. Expert i predicts $p_i \in \{-1, 1\}$ where $p_i = 1$ if the prediction is it'll rain and vice versa. Each expert i is associated with a weight w_i.

Now we have to decide whether it'll rain. Earlier on, we picked an expert to bet on. Note that instead of picking an expert with probability proportional to w_i, here we take a weighted vote of experts. This algorithm is called the Weighted Majority algorithm.

Algorithm: Let $w_i = 1$ at time $t = 0$. After each instance 't' of a prediction, for each of the expert i, that make a mistake: $w_i \leftarrow w_i(1 - \epsilon)$.
To predict the next event $t + 1$, we take the weighted vote of the experts.
We pick value 1 if $\sum(w_i p_i) > 0$
We pick value -1 if $\sum(w_i p_i) < 0$,
where w_i is the weight associated with the expert i and p_i is the prediction made by the expert i.

23.1.1 General case

We can generalize the above problem by letting the costs take any continuous value between 0 and 1. Now, each expert i makes a prediction $p_{i,t}$ which can lead to a loss $l_{i,t} \in [0, 1]$ at time t of the prediction. Our goal remains to minimize our costs.

Let $w_{i,t}$ be the weight of expert i at time t, $l_{i,t}$ be the loss of expert i at time t, and let W_t be the total weight of all experts at time t. The algorithm is as follows.

Algorithm:
For each expert initialize $w_{i,0} = 1$.
At each step, pick expert i with prob. $\frac{w_{i,t}}{W_t}$, where $W_t = \sum_i(w_{i,t})$
Update $w_{i,t+1} = w_{i,t}(1 - \epsilon)^{l_{i,t}}$ where $l_{i,t}$ is the loss of expert i.

Claim 23.1.1 *$W_T \geq (1 - \epsilon)^L$, where T is the final time step, and L is the total loss of the best expert.*

Proof: Let n be the total number of experts, and let j be the best expert. So $W_T = \sum_{i=1}^{n} w_{i,T} \geq w_{j,T} = (1 - \epsilon)^L$. ■

Let F_t be the expected cost of the algorithm at time t. So $F_t = \sum_i \frac{w_{i,t} l_{i,t}}{W_t}$
Our total expected cost $= \sum_t \sum_i \frac{w_{i,t} l_{i,t}}{W_t}$
Now let us get a relation between F_t and W_T, so that using claim 1, we can get a relationship between F_t and L.

$$W_{t+1} = \sum_i w_{i,t}(1-\epsilon)^{l_{i,t}}$$

$$\leq \sum_i w_{i,t}(1-\epsilon l_{i,t})$$

$$= \sum_i w_{i,t} - \epsilon \sum_i w_{i,t} l_{i,t}$$

$$= W_t - \epsilon W_t F_t$$

$$W_{t+1} \leq W_t(1-\epsilon F_t)$$

This gives,

$$W_T \leq n \prod_t (1-\epsilon F_t)$$

Using claim 1,

$$(1-\epsilon)^L \leq n \prod_t (1-\epsilon F_t)$$

$$L \log(1-\epsilon) \leq \log n + \sum_t \log(1-\epsilon F_t)$$

(Note that $\log(1+x) \leq x$)

$$L \log(1-\epsilon) \leq \log n - \epsilon \sum_t F_t$$

$$\sum_t F_t \leq -L \log(1-\epsilon)/\epsilon + \log n/\epsilon$$

If ϵ is small, using Taylor's series expansion, we get our total expected cost $= \sum_t F_t \approx (1 + \epsilon/2)L + \log n/\epsilon$ If instead of loss we were to gain G, the equivalent equation becomes $\sum_t F_t \leq (1-\epsilon/2)G - 1/\epsilon \log n$
Now, let $\epsilon = \sqrt{\log n/L}$. If L is large enough, ϵ is small and we get,

$$\sum_t F_t \approx L + O(\sqrt{L \log n})$$

This is a good result as our expected cost is within an additive term of $O(\sqrt{L \log n})$ of the best expert.

23.2 Mistake Bound Model for Learning

This is a variant of the prediction problem in which we believe that atleast one expert is always correct. The goal is to minimize the number of mistakes our algorithm makes before we converge to the "correct" expert. Note that the above multiplicative update method gives us a bound of $1/\epsilon \log n$ in this setting as $L = 0$. Can we do better?

A simple naive algorithm would be to choose a random expert at each step, and eliminate all those experts who make a mistake. But in the worst case, this algorithm will end up making $(n-1)$ mistakes, where n is the number of experts. (The worst case is when in each step, only the expert we pick makes a mistake).

23.2.1 Halving algorithm

This algorithm makes a total of only $\log_2 n$ mistakes. The algorithm is as follows.
1. Start with all the experts.
2. For each example, take a majority vote of the remaining experts, and eliminate all the experts who make a mistake.
3. Keep repeating step 2 for each incoming example on which we need to make a prediction.

It is easy to note that whenever we make a mistake, we remove atleast half of the remaining experts (since we voted in accordance with the majority of these experts). So the halving algorithm makes atmost $log_2 n$ mistakes before we eliminate all but the "correct" expert, and from then on we never make a mistake.

23.3 Concept Learning

Often times, we can never maintain an explicit list of all the experts and do the Halving algorithm. So in this section, we will look at implicit ways of doing the same thing (without maintaining an explicit list of all remaining experts). For example, if we take the spam example, each email can have multiple attributes. For example, 'contains the word Viagra' can be one attribute, 'fraction of spelling mistakes > 0.2' can be one attribute, and so on. An expert is a function over these attributes. Concept class is the class of all the experts we consider. For example, one concept class can be the class of decision trees with at most n nodes. While this concept class is smaller than the class of all possible experts, it is still very huge (exponential) and we cannot use the explicit halving algorithm. Next we consider two algorithms on specific concept classes in the mistake bound model.

23.3.1 Concept class of OR functions

Here each expert is an OR of some of the attributes. So, if we have n attributes, there are a total of 2^n experts (each attribute may or may not be present in the expert). The mistake bound algorithm is as follows.
1. Start with a list L of all the attributes.
2. We predict accoring to the OR of all the remaining attributes in L.
3. If we make a mistake on an example e, we remove all the attribute in e from our list L.

4. Keep repeating steps 2 and 3 for each example on which we need to make predictions.

Theorem 23.3.1 *The number of mistakes made by the above mistake bound algorithm on the concpt class of OR functions $\leq n$, where n is the number of attributes.*

Proof: It can be noted that we maintain the invariant that L is a superset of all the attributes in the true expert. If an example contains atleast one attribute in the true expert, the example is true, and we also predict true. We never end up removing an attribute in the true expert. However an example might be false, and we might predict it as true. (For example in the beginning, when we have all the attributes in L). However, each time we make a mistake, we remove atleast one attribute. So we make atmost n mistakes before we elminate all the attributes not in the true expert. ■

23.3.2 Concept class of OR functions of maximum size k

Here, $|C| = \sum_{i=0}^{k} \binom{n}{k} \approx n^k$. So, we want a mistake bound model which has a maximum of $O(k \log n)$ mistakes. Let A be the set of n attributes. Let w_i be the weight of the i^{th} attribute which we keep updating. In any example, let $x_i = 1$, if the example contains the i^{th} attribute, and $x_i = 0$, otherwise. The following Winnow algorithm makes a maximum of $O(klogn)$ mistakes.

Winnow algorithm:
1. Initialize $w_i = 1, \forall i \in A$.
2. If $\sum_i w_i x_i \geq n$, predict positive, else predict negative.
3. On a mistake, if the true label was positive, $w_i = 2w_i, \forall$ attributes i present in the example.
4. On a mistake, if the true label was negative, $w_i = \frac{w_i}{2}, \forall$ attributes i present in the example.
5. Keep repeating steps 2, 3, and 4 on all incoming examples on which we need to make predictions.

Claim 23.3.2 *Let P be the number of mistakes made on positive examples (the true label on the example is positive). $P \leq k \log_2 n$.*

Proof: Let us focus on any particular attribute j in the true expert. Whenever we make a mistake on an example containing the attribute j, we double the weight of that attribute. So we make atmost $log_2 n$ mistakes on positive examples containg attribute j before w_j becomes more than n, and from then on we do not make any mistakes on positive examples containg attribute j. Since we have atmost k such attributes in the true expert, we make atmost $klog_2 n$ mistakes before we stop making mistakes on positive examples. ■

Claim 23.3.3 *Let N be the number of mistakes made on negative examples (the true label on the example is negative). $N \leq 2P + 2$.*

Proof: Whenever we make a mistake on a positive example (the sum of weights on attributes in that example $< n$), the increase in sum of weights is atmost n. Likewise when we make a mistake on

a negative example (the sum of weights on attributes in that example $\geq n$), the reduction in sum of weights $\geq \frac{n}{2}$. The initial sum of weights of attributes is n. After P mistakes on positive examples, and N mistakes on negative examples, the sum of weights of all attributes $\leq n + nP - \frac{n}{2}N$. The total weight never falls below 0. So, $n + nP - \frac{n}{2}N \geq 0$. So, $N \leq 2P + 2$. ■

Theorem 23.3.4 *The bound on total number of mistakes is $O(k \log n)$ using the above two claims.*

23.4 r out of k Concept Class

Here the true expert again has k attributes, and if any r of them are present in the example, then the example is true, else false. In the next class, we will see how the Winnow algorithm (using a multiplicative factor of $(1+\epsilon)$ instead of 2) can be used to get a mistake bound of $O(krlogn)$ (using $\epsilon = \frac{1}{2r}$).

CS787: Advanced Algorithms

Scribe: Dan Wong, Blayne Field
Lecturer: Shuchi Chawla
Topic: MB Model, Perceptron Alg. and PAC Model
Date: November 14, 2007

24.1 Introduction

Continuing from the last lecture, we resume our discussion of concept learning in the mistake bound (MB) model. Recalling from last time, we want to give an online algorithm for learning a function from a concept (aka hypothesis) class C. We are given examples with n features, and we wish to predict the value of our target function. After each step, we learn if our prediction was correct or not, and update our future predictions accordingly. Our goal is to do this in a manner that guarantees some bound on the number of mistakes that our algorithm makes.

Definition 24.1.1 *A class C is said to be learnable in the MB model if there exists an online algorithm that given examples labeled by a target function $f \in C$ makes $O(log|C|)$ prediction mistakes and the run time for each step is bounded by $O(n, log|C|)$*

It turns out that this requirement for the run time is necessary, since without it, the halving algorithm (discussed in a previous lecture) would give us a $O(log|C|)$ bound for any concept class C. It was demonstrated in the last lecture that the class of boolean OR functions is learnable in the MB model, along with boolean OR functions with at most k literals. We also saw that the winnow algorithm (described below) solves (r of k) majority functions in the MB model.

24.2 Winnow Algorithm

Consider the problem of learning a (r of k) concept class. Here the class we are trying to learn is one where the target function contains k attributes and if at least r of them are present in the example then it is true otherwise it is false.

Recall the Winnow Algorithm:

- Start with $w_{i,0} = 1 \forall i$
- Predict 1 if $\sum w_{i,t} x_{i,t} \geq n$ else predict 0.
- Update weights on mistake as follows:
– Positive Example: For every i with $x_{i,t} = 1$ set $w_{i,t+1} = (1 + \epsilon) w_{i,t}$
– Negative Example: For every i with $x_{i,t} = 1$ set $w_{i,t+1} = \frac{w_{i,t}}{(1+\epsilon)}$

Note that if the weight of an attribute is greater than n, it will never increase further.

Let P = The number of mistakes on positive examples, and likewise N be the number of mistakes on negative examples.
Recall: $\sum_i w_{i,0} = n$

The total increase in $\sum_i w_i \leq P\epsilon n$
The total decrease in $\sum_i w_i \geq \frac{N\epsilon n}{1+\epsilon}$

Since our weights never go negative we get an inequality:

$$n + P\epsilon n \geq \frac{N\epsilon n}{1+\epsilon}$$

To make another inequality we will count the total number of increases or decreases to any relevant attribute seen by our set of k attributes. Consider the analogy of throwing coins in buckets again where every bucket is one of the relevant attributes and every time we increase an attributes weight we add a coin and every time we decrease its weight we remove one. We add at least r new coins for each positive example since there must have been at least r target attributes in our example since it was labeled positive. For each negative example we remove at most $(r-1)$ coins since there must have been at most $(r-1)$ target attributes since the example was labeled negative. The maximum net increase seen by any single one of the k attributes is $\log_{1+\epsilon} n$.

Therefore the number of increases minus the decreases should be less than or equal to the maximum net increase:

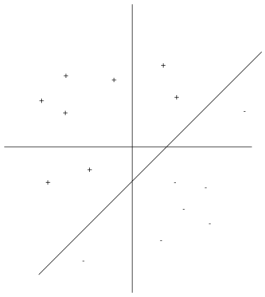
$$Pr - (r-1)N \leq klog_{(1+\epsilon)}n$$

Solving the 2 inequalities for P and N yields:

$$P, N = O(rk \log n), \text{for } \epsilon = \frac{1}{2r}$$

24.3 Linear Threshold Functions

One class of functions that we are interested in learning is that of linear threshold functions. A linear threshold function in n dimensions is a hyperplane that splits the space in two, where the points on different sides correspond to positive and negative labels. The function is defined by a set of weights $W = \{w_1, w_2, ..., w_n\}$, where the label of the function then becomes $sign[\sum w_i x_i - w_0]$. This type of function may arise in many applications, such as if we were given examples of people



by their height and weight, labeled by whether they were obese, and we wished to learn a function of height and weight that predicted whether a person is obese.

Note that for this class of functions, we can't apply the definition of being learnable, since the class has infinite size. Instead, we will derive a mistake bound based on the geometric properties of the concept class.

24.4 Perceptron Algorithm

Suppose we are given some target function f from the class of linear threshold functions (which we will assume WLOG that $w_0 = 0$). We want to have an algorithm to learn this function with a reasonable bound on the number of errors. It turns out that such an algorithm exists (the perceptron algorithm), which gives us a bound on the number of errors as a function of the margin of the target function. If we define $\hat{w_f}$ to be the normal of the plane defining our target function, then $\gamma(f) = min_{x:|x|=1}|x \cdot \hat{W_f}|$ is known as the margin of f. At a high level, the perceptron algorithm works by maintaining a vector w (denoted at time t by w_t). At each step, we make a prediction according to $sign(w \cdot x)$, where x is our input.

The Perceptron Algorithm goes as follows:
- Start with $w_0 = \vec{0}$
- At each step, predict the label of x_t according to $sign(w_t \cdot x_t)$.
- If we made a mistake, set $w_{t+1} = w_t + l(x_t)x_t$, where l_t is the true label of x_t
- Otherwise, set $w_{t+1} = w_t$ (make no change)

It turns out that the perceptron algorithm makes no more than $\frac{1}{\gamma^2}$ mistakes, which we prove below.

Lemma 24.4.1 *If we make a mistake at step t, then* $w_{t+1} \cdot \hat{w_f} \geq w_t \cdot \hat{w_f} + \gamma$

Proof: To proof Lemma 24.4.1, we just go to the definition:
$w_{t+1} \cdot \hat{w_f} = (w_t + l(x_t)x + t) \cdot \hat{w_f} = w_t \cdot \hat{w_f} + l(x_t)x_t \cdot \hat{w_f} \geq w_t \cdot \hat{w_f} + \gamma$. Since $|x \cdot \hat{w_f}| \geq \gamma$ by definition. Also, we know that signs work out correctly, since if we made a mistake, $l(x_t)$ has the same sign as $x_t \cdot \hat{w_f}$ ■

Alone, this lemma does not neccesarily mean we are approaching the correct value, since the dot product could just be getting larger due to the size of the vector increasing. In the next lemma, we show that the lengths of the vectors don't grow very rapidly.

Lemma 24.4.2 *If step t is a mistake, then* $||w_{t+1}||^2 \leq ||w_t||^2 + 1$.

Proof: To proof Lemma 24.4.2, once again we just use the definition: $||w_{t+1}||^2 = ||w_t + l(x_t)x_t||^2 = ||w_t||^2 + ||x_t||^2 + 2w_t \cdot x_t l(x_t)$, since $w_t x_t$ and $l(x_t)$ have opposite signs, so $2w_t \cdot x_t l(x_t) < 0$. ■

Combining these two lemmas gives us that after M mistakes $\sqrt{M} \geq |w_t| + 1 \geq w_t \cdot \hat{w_f} \geq \gamma * M \rightarrow M \leq \frac{1}{\gamma^2}$

24.5 Extensions to the prediction problem

It also turns out that we extend the weighted majority algorithm to solve variations of the prediction problem as well. These variations include the situation where we don't learn the costs of the experts we did not pick (a very real possibility in a real-world situation). It may also be the case that not all experts are available or relevant at all time steps, and the algorithm must be modified to deal with these changes. Another possible situation is that we may want to assign some random initial offset to our experts.

24.6 Introduction to PAC Model

The PAC Model, Probably Approximately Correct Model, is a learning model where we claim that the function we are trying to learn is approximately correct.

Definition 24.6.1 *A concept class, C, is PAC learnable if there exists an algoirthm, A, such that $\forall \epsilon, \delta > 0$ for every distribution D, and any target function, $f \in C$, given a set S of examples drawn from D labled by f, A produces with probability $(1-\delta)$ a function h such that $\mathbf{Pr}_{x \sim D}[h(x) = f(x)] \geq (1-\epsilon)$.*

Note that by allowing the constructed function h to differ from f with some probability ϵ, we are protected against negligible weight examples. In a similar sense δ protects against the event that when sampling S from D that we get an unrepresentative set S. Since low weight examples are unlikely to be seen in the sample, it would be unreasonable to expect the algorithm to perform well in either case.

In addition we require the algorithm to work in reasonable amount of time and only use a reasonable amount of examples.
Specifically:

The size of S,$|S|$ should be $poly(\frac{1}{\epsilon}, \frac{1}{\gamma}, n, \log|C|)$
Runtime should be $poly(\frac{1}{\epsilon}, \frac{1}{\gamma}, n, \log|C|)$

CS787: Advanced Algorithms	
Scribe: Chris Hinrichs and Baris Aydinlioglu	**Lecturer:** Shuchi Chawla
Topic: PAC Learning	**Date:** November 16, 2007

In this lecture we continue our discussion of learning in the PAC (short for Probably Approximately Correct) framework.

25.1 PAC Learning

In the PAC framework, a **concept** is an efficiently computable function on a domain. The elements of the domain can be thought of as objects, and the concept can be thought of as a classification of those objects. For example, the boolean function $f : \{0,1\}^n \to \{0,1\}$ classifies all $0,1$ n-vectors into two categories. A **concept class** is a collection of concepts.

In this framework, a learning algorithm has off-line access to what we can call a "training set". This set consists of $\langle element, value \rangle$ pairs, where each *element* belongs to the domain and *value* is the concept evaluated on that element. We say that the algorithm can learn the concept if, when we execute it on the training set, it outputs a hypothesis that is consistent with the entire training set, and that gives correct values on a majority of the domain. We require that the training set is of small size relative to that of the domain but still is representative of the domain. We also require that the algorithm outputs its hypothesis efficiently, and further that the hypothesis itself can be computed efficiently on any element of the domain.

We used a number of qualitative terms in the above description (e.g., how do we tell if the training set represents the domain?). We formalize these concepts in a probabilistic setting in the following definition.

Definition 25.1.1 (Learnability) *Let C be a class of concepts that are defined over a domain T. We say C is* **PAC-learnable** *if there is an algorithm ALG such that for any distribution D over T, any concept $f \in C$, any $\delta > 0$ and $\epsilon > 0$, when ALG is given a set of examples $S = \{\langle x, f(x)\rangle : x \sim D\}$, of size $|S| = \text{poly}\left(\frac{1}{\epsilon}, \frac{1}{\delta}, n, \log|C|\right)$, ALG produces, in time $\text{poly}(|S|)$, a hypothesis $h \in C$ satisfying:*
(i) h is polytime computable,
(ii) h agrees with f on S,
(iii) $\mathbf{Pr}_{S\sim D}\left[\ \mathbf{Pr}_{x\sim D}\left[h(x) \neq f(x)\right] \leq \epsilon\ \right] \geq 1-\delta$.

In this definition $1-\delta$ captures our confidence in the algorithm and $1-\epsilon$ captures the accuracy of the algorithm. By forming the training set S with respect to a distribution D, and measuring the accuracy of the algorithm by drawing elements from the domain T according to the same distribution, we enforce that the training set represents the domain. The condition on the size of S captures the intuition that for greater accuracy or confidence we have to train the algorithm with more examples.

There are a number of concept classes that are learnable in the PAC framework. Next we study one of these, Decision Lists.

25.1.1 Decision Lists

A Decision List (DL) is a way of representing certain class of functions over n-tuples. Each n-tuple can be thought of as describing an object through a fixed set of n attributes. A DL consists of a series of if-then-else type rules, each of which references exactly one attribute, except the last rule which references at most one attribute.

For example, suppose we have the concept $f : \{0,1\}^5 \to \{0,1\}$, and we are given a set A of sample objects, $A \subset \{0,1\}^5$, along with the value of f on each element $x = \langle x_1, ..., x_5 \rangle$ of A, as follows:

x_1	x_2	x_3	x_4	x_5	$f(x)$
0	1	1	0	0	1
0	0	0	1	0	0
1	1	0	0	1	1
1	0	1	0	1	0
0	1	1	1	1	0

Then the following is a decision list that is consistent with, or that represents, f restricted to A.

if $x_4 = 1$ then $f(x) = 0$
else if $x_2 = 1$ then $f(x) = 1$
else $f(x) = 0$.

A rough upper bound on the number of all possible boolean decision lists on n boolean variables is $n!4^n = O(n^n)$, since there are $n!$ ways of permuting the attributes (x_i), 2^n ways of specifying conditions $(x_i = j)$ on those permuted attributes, and another 2^n ways of assigning values to the function $(f(x) = k)$ when each condition is true. From this we see that we cannot represent all possible boolean functions on n variables with decision lists as there are 2^{2^n} of them.

25.1.1.1 A PAC-learning algorithm for DLs

Let $f : T \to \{0,1\}$ be a concept that is representable as a decision list, defined over a domain $T = \{0,1\}^n$. Let $S = \{\langle x, f(x)\rangle : x \sim D\}$ be a training set.

The following algorithm takes as input S, and outputs a decision list h that is consistent with S:

repeat until $|B| = 1$, where $B = \{f(x) : \langle x, f(x)\rangle \in S\}$
  repeat until a rule is output
    let x_i be an attribute that doesn't appear in a rule yet
    let $B_j = \{f(x) : \langle x, f(x)\rangle \in S$ and $x_i = j\}$, where $j \in \{0,1\}$
    if $|B_j| = 1$ then
      output rule "if $x_i = j$ then $f(x) = k$ ", where $k \in B_j$
      output partial rule "else"
      $S \leftarrow S - \{\langle x, f(x)\rangle : x_i = j\}$
output rule "$f(x) = l$" where $l \in B$.

Since we know f is representable as a decision list, it follows that this algorithm always terminates. Clearly it runs in polynomial time. It is also clear that for any x in the domain, $h(x)$ is efficiently computable.

What remains to be shown, in order to prove that this algorithm can learn in the PAC framework, is the following. If we want our algorithm to output, with probability at least $1-\delta$, a hypothesis h that has an accuracy of $1-\epsilon$, it suffices to provide as input to our algorithm a training set S of size as specified in definition 1.

Lemma 25.1.2 *Let f be the target concept that our algorithm is to learn, and let h be its output. Suppose S is of size $|S| \geq \frac{1}{\epsilon}\left(N + \frac{1}{\delta}\right)$, where $N = O(n \log n)$ is the natural logarithm of the number of different decision lists. Then $\mathbf{Pr}_{S\sim D}[h \text{ is } 1-\epsilon \text{ accurate}] \geq 1-\delta$.*

Proof: Suppose $|S|$ is as in the statement of the claim. It suffices to show

$$\mathbf{Pr}_{S\sim D}[h \text{ is more than } \epsilon \text{ inaccurate}] \leq \delta\,. \quad (*)$$

Consider what it means for h to have an inaccuracy ratio more than ϵ. One way to interpret this is that our algorithm is "fooled" by the training set $S \sim D$, in the sense that even though the target concept f and the hypothesis h agree on all of S, they disagree on too-large a fraction of the domain. (We view the domain through D.) So we can rewrite (*) as

$$\mathbf{Pr}_{S\sim D}[\text{our algorithm is fooled}] \leq \delta\,. \quad (**)$$

For a given S, partition the set of all concepts that agree with f on S into a "bad" set and a "good" set. The bad set contains those hypotheses which, if our algorithm outputs one of them, would be fooled.

$$\mathcal{H}_{bad} = \{h \in C : h(x) = f(x)\,\forall x \in S\,, \text{ but } \mathbf{Pr}_{x\sim D}[h(x) \neq f(x)] \geq \epsilon\}\,.$$

Here C is the set of all decision lists. In order to succeed, our algorithm should output a member of $\mathcal{H}_{good}$.

$$\mathcal{H}_{good} = \{h \in C : h(x) = f(x)\,\forall x \in S\,, \text{ and } \mathbf{Pr}_{x\sim D}[h(x) \neq f(x)] \leq 1-\epsilon\}\,.$$

Notice that for a given S, $\mathcal{H}_{good}$ is never empty (contains f at least) but $\mathcal{H}_{bad}$ may be, in which case our algorithm definitely does not get fooled. Now we can rewrite (**) as

$$\mathbf{Pr}_{S\sim D}[\mathcal{H}_{bad} \text{ is non-empty and our algorithm outputs a member of it}] \leq \delta\,. \quad (***)$$

To show (***), it suffices to show

$$\mathbf{Pr}_{S\sim D}[\mathcal{H}_{bad} \text{ is non-empty}] \leq \delta\,,$$

which we shall do in the rest of the proof.

Consider any $h \in C$ such that $\mathbf{Pr}_{x\sim D}[h(x) \neq f(x)] \geq \epsilon$. The probability that we pick S such that h ends up in $\mathcal{H}_{bad}$ is

$$\mathbf{Pr}_{S\sim D}[h(x) = f(x)\,\forall x \in S] \leq (1-\epsilon)^m\,,$$

where $m = |S|$. Here we used the fact that S is formed by independently picking elements from the domain according to D.

What we have just obtained is the bound on a particular inaccurate $h \in C$ ending up in $\mathcal{H}_{bad}$. There are at most $|C|$ such h, so by a union bound,

$$\mathbf{Pr}_{S\sim D}\left[\mathcal{H}_{bad} \text{ is non-empty}\right] \leq |C|\,(1-\epsilon)^m\ .$$

All that is left is to show that the right hand side of the last inequality is at most δ.

Note that by definition $|C| = e^N$, and using $1-\epsilon \leq e^{-\epsilon}$, we have $\frac{1}{\epsilon} \geq \frac{1}{\log\left(\frac{1}{1-\epsilon}\right)}$. Putting together, we get

$$\begin{aligned} m &\geq \frac{1}{\log\left(\frac{1}{1-\epsilon}\right)}\left(\log|C| + \log\frac{1}{\delta}\right) \\ -m\log\left(1-\epsilon\right) &\geq \log|C| + \log\frac{1}{\delta} \\ \log\delta &\geq \log|C| + m\log\left(1-\epsilon\right), \end{aligned}$$

which implies that

$$\mathbf{Pr}_{S\sim D}\left[\mathcal{H}_{bad} \text{ is non-empty}\right] \leq |C|\,(1-\epsilon)^m \leq \delta\,,$$

proving the lemma. ■

25.1.2 Other Concept Classes

Notice that in the above lemma we used the concept class being DLs only when we substituted $n\log n$ with $\log|C|$. This suggests the following:

Theorem 25.1.3 (Consistency ⇒ Learnability) *Let C be a concept class. Suppose that there is an algorithm that, given a sample set S, can efficiently (in time* $\text{poly}\,|S|$*) solve the consistency problem (i.e. produce a hypothesis $h \in C$ that is consistent with S). Then C is PAC-learnable with sample complexity $|S| = \frac{1}{\epsilon}\left(\log|C| + \log\frac{1}{\delta}\right)$.*

Proof: Once we show such an algorithm exists, the PAC-learnability of C follows by an invocation of the above lemma with $|S| \geq \frac{1}{\epsilon}\left(\log|C| + \log\frac{1}{\delta}\right)$. ■

We apply this theorem to some other concept classes below.

25.1.2.1 Decision Trees

A Decision Tree (DT) is an extension of DLs where each condition can refer to more than one attribute. If DLs and DTs are represented as graphs with rules as vertices and connections between rules as edges, then a DL has a linked list structure, and a DT has a tree structure.

DTs are powerful enough to represent any boolean function. For example, $f : \{0,1\}^n \to \{0,1\}$ can be represented by a DT with root node labeled x_1, two children of the root node both labeled x_2 at the second level, and so on where at the last (nth) level there are 2^n nodes all labeled x_n.

Given a sample set S, we can efficiently produce a consistent hypothesis by essentially "fitting" a DT to S. For each sample in S, we add a branch that encodes that sample to the DT. The final hypothesis will look like a DT as described in the previous paragraph, though only with $|S|$ leaves. Clearly this procedure runs in time $\text{poly}\,|S|$.

It follows by the Theorem 25.1.3 that DTs are PAC-learnable. But usually we are interested in representing data with small decision trees, and not decision trees of arbitrary size. So one might ask, can we learn decision trees with depth at most k, or decision trees with at most k nodes, etc. Unfortunately solving the consistency problem for such questions is NP-hard.

For example, suppose our target concept is a DT with number of nodes at most $g(|S|)$, where $g(|S|) = o(|S|)$. The concept class in this case is all DTs with less than $g(|S|)$ nodes. So $|C| = n^{g(|S|)}$ and we can PAC-learn C with $|S| \geq \frac{1}{\epsilon}\left(g(|S|)\log n + \log\frac{1}{\delta}\right)$, provided that we come up with an efficient algorithm that can solve the consistency problem on S.

The problem is that such an algorithm is unlikely to exist, because it is an NP-hard problem to produce, given S, a decision tree with $o(|S|)$ nodes that is consistent with the entire sample set.

So DTs of restricted size are not PAC-learnable, although those of arbitrary size are.

25.1.2.2 AND-formulas

An AND-formula is a conjunction of literals, where a literal is an attribute or its negation. For example, $x_1 \wedge \neg x_2 \wedge x_n$ is an AND-formula. There are 2^{2n} AND-formulas on n literals.

Given a training set S we can efficiently produce a consistent hypothesis as follows. Start out with the formula $x_1 \wedge \neg x_1 \wedge ... \wedge x_n \wedge \neg x_n$, then for each $x \in S$ with $f(x) = 1$, if $x_i = 0$ then eliminate x_i from the formula, otherwise eliminate $\neg x_i$; do this for $i = 1..n$.

So this class is PAC-Learnable.

25.1.2.3 OR-formulas

Similar to AND-formulas. In this case we construct a hypothesis by looking at $x \in S$ with $f(x) = 0$. So this class is also PAC-Learnable.

25.1.2.4 3-CNF formulas

A 3-CNF formula is a conjunction of clauses, each of which is a disjunction of exactly 3 literals. There are $2^{\text{poly}(n)}$ 3-CNF formulas on n literals.

We can reduce the problem of constructing a hypothesis to that in the AND formulas section, by mapping each of the $\binom{2n}{3}$ possible clauses that can be formed from n literals into new variables y_i. Since this reduction is polytime, and the algorithm for the AND-formulas is polytime, we obtain a polytime procedure that outputs a hypothesis.

So this class is PAC-learnable.

25.1.2.5 3-DNF formulas

A 3-DNF formula is a disjunction of clauses, each of which is a conjunction of exactly 3 literals.

The analysis is similar to 3-CNF (in this case it reduces to the OR-formulas).

25.1.2.6 3-term DNF formulas

A 3-term DNF formula is a disjunction of exactly 3 clauses, each of which is an AND-formula.

Since a 3-term DNF formula is functionally equivalent to a 3-CNF formula, we may be tempted to conclude that 3-term DNF formulas are also learnable, as we already showed above an algorithm that can efficiently solve the consistency problem for 3-CNF formulas.

The problem with this is that 3-CNF formulas are a superset of 3-term DNF formulas, and if we reuse the algorithm from the 3-CNF formula case, we may get a hypothesis that does not belong to the class of 3-term DNF formulas. It could take exponential time to go through all possible 3-CNF formulas consistent with the sample set until we find one that is also a 3-term DNF formula—and that is assuming we can quickly tell if a 3-CNF formula is also a 3-DNF formula.

In fact, it turns out that it is an NP-hard problem, given S, to come up with a 3-term DNF formula that is consistent with S. Therefore this concept class is not PAC-learnable—but only for now, as we shall soon revisit this class with a modified definition of PAC-learning.

25.2 PAC-Learning—a revised definition

It seems somewhat unnatural to require that the hypothesis be in the same class of concepts as that of the target concept. If, instead, we allow the hypothesis to be from a larger concept class then we might be able to learn some concepts that would otherwise be not learnable. This leads to the following revised definition of PAC-learning.

Definition 25.2.1 (Learnability—revised) *Let C be a class of concepts that are defined over a domain T. We say C is* **PAC-learnable** *if there is an algorithm ALG such that for any distribution D over T, any concept $f \in C$, any $\delta > 0$ and $\epsilon > 0$, when ALG is given a set of examples $S = \{\langle x, f(x)\rangle : x \sim D\}$, of size $|S| = \text{poly}\left(\frac{1}{\epsilon}, \frac{1}{\delta}, n, \log|C|\right)$, ALG produces, in time $\text{poly}(|S|)$, a hypothesis $h \in C'$— where C' is some (possibly different) concept class—satisfying:*
(i) h is polytime computable,
(ii) h agrees with f on S,
(iii) $\mathbf{Pr}_{S\sim D}\left[\ \mathbf{Pr}_{x\sim D}\left[h(x) \neq f(x)\right] \leq \epsilon\ \right] \geq 1 - \delta$.

With this new definition, Theorem 25.1.3 also needs to be revised.

Theorem 25.2.2 (Consistency ⇒ Learnability—revised) *Let C be a concept class. Suppose that there is an an algorithm that, given a sample set S, can efficiently (in time $\text{poly}\,|S|$) solve the consistency problem (i.e. produce a hypothesis $h \in C'$ that is consistent with S). Suppose further that $\log|C'| = O(\text{polylog}\,|C|)$. Then C is PAC-learnable with sample complexity $|S| = \frac{1}{\epsilon}\left(\log|C'| + \log\frac{1}{\delta}\right)$.*

Proof: Invoke Lemma 25.1.2 with $|S| \geq \frac{1}{\epsilon}\left(\log|C'| + \log\frac{1}{\delta}\right)$. ■

We now turn to the two concept classes that we previously concluded not-learnable based on the original definition, and see if the revised definition makes a difference.

- **3-term DNF formulas**: Now we are allowed to output a 3-CNF formula even if it is not equivalent to a 3-term DNF formula. So we can run the algorithm from the 3-CNF

formulas discussion. Then C' is the class of 3-CNF formulas, and $\log|C'| = \log 2^{\text{poly}\,(n)} = \text{polylog}\, 2^{O(n)} = O(\text{polylog}\,|C|)$, so 3-term DNF formulas are now PAC-learnable.

- **Decision Trees of Small Size**: Suppose we run the algorithm from the above discussion regarding unrestricted DTs. Then C' is the class of all DTs, and we can conclude that DTs of small size are PAC-learnable if it is the case that $\log|C'| = \text{polylog}\,|C|$. Unfortunately it is usually not the case. For example, if we are interested in learning decision trees with $\text{poly}\, n$ nodes, then $\log|C'| = \log 2^{2^n} \gg \text{poly}\, n = \text{polylog}\,(n^{\text{poly}\, n}) = \text{polylog}\,|C|$. Therefore, even with the revised definition, the class of DTs of small size are not learnable—at least when C' is the class of all DTs.

25.3 Occam's Razor

One use of the theory of PAC learnability is to explain the principle Occam's Razor, which informally states that simpler hypotheses should be preferred over more complicated ones. Slightly more formally we can say that

$\text{size}(f) \leq s \forall f \in C \implies |C| \leq 2^s$

where $\text{size}(f)$ refers to the amount of information (usually measured in bits) sufficient to represent f. In other words, if all functions $f \in C$ have descriptions which fit into s bits then there are at most only 2^s such functions. If we can choose a hypothesis belonging to a smaller concept class C, and having description length s, then we can show the relationship between s and expected error rate ϵ if we treat δ as fixed.

Since $\delta \geq |C|(1-\epsilon)^m$ we can say that

$$\delta \geq 2^s(1-\epsilon)^m$$

$$\log(\delta) \geq s - \epsilon m$$

$$\epsilon \approx \frac{s + \log(\frac{1}{\delta})}{m}$$

Thus ϵ is directly related to $\frac{s}{m}$ (where m is the number of training examples).

25.3.1 Learning in the Presence of Noisy Data

We saw in Theorems 25.1.3 and 25.2.2 that if we can efficiently find a hypothesis h that is consistent with a given sample set S then h has, with high probability, the desired accuracy on the domain, provided that S is large enough. We also saw in our discussion on DTs of a certain size that we sometimes don't know how to efficiently find such h. But sometimes it may be even impossible to solve the consistency problem, no matter how much computational power we have, because S may have erroneous data (noise), so that no concept in the target concept class is actually consistent with S. In this section we discuss whether it is possible, even in such a setting, to produce h that has desired accuracy with high probability.

25.3.1.1 Uniform Convergence Bounds (a.k.a. Generalization Error Bounds)

In this noisy setting our examples take the form $S = \{\langle x, f(x)\rangle_{\sim D}\}$, where f is not necessarily in C, (though h is). We seek a function in C that has the least error with respect to f.

The least such error is:

$$\min_{h \in C} \mathbf{Pr}_{x \sim D}[h(x) \neq f(x)]$$

Now we train our classifier h on the data S, but there is one subtle difference: we choose h which minimizes the error on examples x drawn from S, (i.e. empirical error,) rather than the error on examles drawn from D, (i.e. the true error).

$$h = \arg\min_{h \in C} \mathbf{Pr}_{x \in S}[h(x) \neq f(x)]$$

We will show that for large enough S, an h with low empirical error will have low true error, with high probability.

We define ϵ' as the true error of h on f.

$$\epsilon' = \mathbf{Pr}_{x \sim D}[h(x) \neq f(x)]$$

then since the samples x are drawn independently from D, we can apply Chernoff's bound:

$$\mathbf{Pr}\left[\frac{|x \in S : h(x) \neq f(x)|}{|S|} < \epsilon' - \epsilon\right] < e^{-(2\epsilon^2 m)}$$

This means that with probability $1 - e^{-2\epsilon^2 m}$, the true error of h is no more than ϵ plus its empirical error on S. Now if we define

$$\delta = |C| e^{-(2\epsilon^2 m)}$$
$$\frac{\delta}{|C|} = e^{-(2\epsilon^2 m)}$$

and take the log, then

$$2\epsilon^2 m = \log(|C|) + \log(\frac{1}{\delta})$$

and so

$$\epsilon = \sqrt{\frac{\log(|C|) + \log(\frac{1}{\delta})}{2m}}$$

which gives us ϵ and δ. Now we know that $\mathbf{Pr}[\exists h \in C |$true error of h $>$ empirical error of h on S $+ \epsilon] < \delta$

In other words, with probability $1 - \delta$, $\forall h \in C$, the true error of $h \leq$ empirical error $+ \sqrt{\frac{log(|C|)+\frac{1}{\delta}}{2m}}$
Note that this bound is not quite as good as the bound on the case where we are drawing both f and h from C, (different by a square root factor, and m has a 2 in front of it).

References

[1] Wikipedia. http://en.wikipedia.org/wiki/William_of_Ockham

CS787: Advanced Algorithms

Scribe: Evan Driscoll and Dalibor Zelený
Lecturer: Shuchi Chawla
Topic: Boosting in the PAC Model
Date: November 29, 2007

26.1 Review of Last Class

26.1.1 PAC-learnability and consistency

We defined PAC-learnability as follows:

Definition 26.1.1 *A concept class C is PAC-learnable if there is an algorithm A such that for all $\epsilon > 0$ and $\delta > 0$, for all distributions D on the domain, and for all target functions $f \in C$, given a sample S drawn from D, A produces a hypothesis h in some concept class C' such that with probability at least $1 - \delta$, $\mathbf{Pr}_{x \sim D}[h(x) \neq f(x)] < \epsilon$.*

Both the running time of algorithm A and the size of S should be $\text{poly}(1/\varepsilon, 1/\delta, n, \log|C|)$*. Finally, $h(x)$ should be easy to compute.*

Recall that we do not require h to be drawn from the same concept class as C. The motivating example was that we were trying to find functions in the concept class of 3-term DNF formulas, but doing so is NP-hard. However, by instead finding a consistent 3-DNF formula, we could still get accurate results, and the problem becomes easy.

Finally, the requirement that the running time of A and the size of S should be polynomial in $\log|C|$ makes sense if one considers the fact that, since there are $|C|$ possible functions, simply recording which we chose requires at least $\log|C|$ bits. Thus outputting the function we chose gives a lower bound on the running time of $\log|C|$.

There is an easy goal that, if it is possible to reach, implies that a class is PAC-learnable:

Theorem 26.1.2 *If there is an algorithm A that, given a sample S, solves consistency in time* $\text{poly}(|S|, n)$ *(in other words, produces an $h \in C'$ consistent with S), then we can PAC-learn C if we choose a sample size of at least $|S| \geq \frac{1}{\epsilon}(\log\frac{1}{\delta} + \log|C'|)$.*

As long as C' isn't too much bigger than C, this satisfies the requirements given above for PAC-learnability. A proof of this was given last class.

26.1.2 Noisy Data

We also discussed how to compensate for noisy data. Specifically, there may not be a function f that is completely consistent with any sample. In this case, our goal is to try to learn a function that makes the fewest mistakes.

One way of looking at what we did above is as follows. There is a function $f \in C$ that classifies all data with an error rate of zero: $\text{error}_D(f) = 0$. We draw $S \sim D$ and use it to produce a hypothesis h that classifies all data in S with error zero: $\text{error}_S(h) = 0$. When we try generalize this h to the whole domain, h does well: with probability at least $1 - \delta$, we have $\text{error}_D(h) \leq \epsilon$.

With noisy data, our model changes to the following. There is a function $f \in C$ that classifies all data with an non-zero error rate of p: $\text{error}_D(f) = p$. We draw $S \sim D$ and use it to produce a hypothesis h that classifies all data in S with the lowest possible error rate of all possible h: $\text{error}_S(h) = \min_{h' \in C'} \text{error}_S(h') = p'$. When we generalize this h to the whole domain, h does well: with probability at least $1 - \delta$, we have $\text{error}_D(h) \leq p' + \sqrt{(\log \frac{1}{\delta} + \log |C|)/(2|S|)}$.

26.2 Weak PAC-learnability

We now turn our attention to another question. The conditions required for PAC-learnability seem a bit strong: algorithm A needs to work for any δ and any ϵ. However, suppose that we only had an algorithm that worked for a specific value of δ and ϵ? Is it still possible to do PAC learning? It turns out the answer is "yes," even if $1 - \delta$ is close to zero (i.e. we have very low confidence) and ϵ is just under $1/2$ (i.e. we are only doing slightly better than raw guessing).

We define a new notion, called "weak PAC-learnability," and refer to the previous definition as "strong PAC-learnability." Weak PAC-learnability is defined as follows:

Definition 26.2.1 *A concept class C is weakly PAC-learnable if there exists a $\gamma > 0$ and a $\delta' > 0$ such that there exists an algorithm A such that for all distributions D and all target functions $f \in C$, given a sample $S \sim D$ then A will produce a hypothesis h such that with probability δ', $\mathbf{Pr}_{x \sim D}[h(x) \neq f(x)] \leq \frac{1}{2} - \gamma$.*

Our question can now be rephrased to ask "does weak PAC-learnability imply strong PAC-learnability?" We will that it can by two steps. The first shows that it is possible to boost the confidence from δ' to any arbitrary δ, while not increasing the error rate "too much." The second step will boost the accuracy from $\frac{1}{2} - \gamma$ to an arbitrary ϵ while leaving the confidence unchanged.

26.2.1 Boosting the confidence

We first show how to boost the confidence. Suppose that we are given an algorithm A that gives an h with $\mathbf{Pr}_{x \sim D}[h(x) \neq f(x)] \leq \varepsilon$ with probability δ'. We will show that we can generate a h' such that $\mathbf{Pr}_{x \sim D}[h(x) \neq f(x)] \leq 2\varepsilon$ with probability at least $1 - \delta$, for any $\delta > 0$.

We will first look at an idea that doesn't work. Imagine drawing k samples $S_1, \ldots, S_k$, using A to generate hypotheses $h_1, \ldots, h_k$, then combining the h_is in some way. Taking a majority function doesn't work, because most of the h_i's can be wrong. (Recall that δ' can be close to 0.) Taking the opposite doesn't work either, because they *could* mostly be correct. Because of the very weak guarantees in our problem statement, there is essentially no way to combine the h_i's to get what we want.

However, we can use a similar process to get something that *will* work. We draw k samples and produce hypotheses as before. Note that if we draw enough, chances are that at least *one* of the hypotheses will be good; we just have to figure out which one it is. (We use the term "good" to refer to a hypothesis h for which the bounds on the error apply, and "bad" for the others.) To figure out which h_i is the best hypothesis, we test the accuracy of each of them on another sample S_{k+1}.

For this process to work with high probability, we must ensure two things:

1. We must choose k large enough that we are sufficiently assured that there will be a good hypothesis in the set of h_i's
2. We must choose the size of S_{k+1} so that it is large enough that we are sufficiently assured that the good hypothesis will do a good enough job on S_{k+1} that we will choose it over a bad hypothesis

26.2.1.1 Choosing k

This is the easy part. For each $i \in [k]$, the chance that h_i is good is at least δ'. Thus the probability that *no* h_i is good is $(1-\delta')^k$. We will set k so that this is at most $\delta/2$ for our target δ. (The division by 2 leaves us some leeway for error in the next step.) This ensures that with probability at least $1-\delta/2$, there is a good hypothesis in the set.

Setting $(1-\delta')^k = \frac{\delta}{2}$, we can solve for k, to be $\log\frac{\delta}{2}/\log(1-\delta')$. By using the approximation that $\log\frac{1}{1-x} \approx x$, we can simplify this to $\frac{1}{\delta'}\log\frac{2}{\delta}$.

26.2.1.2 Choosing $|S_{k+1}|$

For any good hypothesis h_i, we want the error rate over the sample S_{k+1} to be, with high probability, about the same as the error rate over the entire distribution. Specifically, we want it to be within $\varepsilon/2$. More formally, we want $\mathbf{Pr}\left[|\operatorname{error}_D(h_i) - \operatorname{error}_{S_{k+1}}(h_i)| > \frac{\varepsilon}{2}\right]$ to be "small enough."

We can bound the above probability using Chernoff's bound to $2\cdot\exp\left(-2\frac{\varepsilon^2}{4}|S_{k+1}|\right)$. To see this, we an alternate form of Chernoff's bound presented in Kearns and Vazirani ([1], section 9.3). Suppose that $X = \Sigma_{j=1}^{n} X_j$ with every X_j a Boolean random variable as in the original formulation, and for all j, $\mathbf{Pr}[X_j = 1] = p$. Then if we take a sample and observe that a proportion $\hat{p}$ of the drawn X_j's are equal to 1, then we can bound the difference between p and $\hat{p}$ as $\mathbf{Pr}[|\hat{p} - p| \geq \gamma] \leq 2e^{-2n\gamma^2}$.

In our example, the samples we are drawing for the X_j's are each member of S_{k+1}, and $X_j = 0$ if the hypothesis h_i is correct and $X_j = 1$ if it is incorrect. Thus n is $|S_{k+1}|$. Because the probability p of $X_j = 1$ is equal to the overall error rate of h_i, $p = \operatorname{error}_D(h_i)$; our estimator is then $\hat{p} = \operatorname{error}_{S_{k+1}}(h_i)$. Finally, $\gamma = \frac{\varepsilon}{2}$. Plugging these into the alternate form of Chernoff's inequality gives us our goal.

We now want to choose $|S_{k+1}|$ so that is is bounded by $\delta/2k$; we will see how this works out in a moment. We want to ensure that *no* good hypothesis is too far off across all the h_i's. Taking the union bound, we get that $\mathbf{Pr}\left[\exists i.\ |\operatorname{error}_D(h_i) - \operatorname{error}_{S_{k+1}}(h_i)| > \frac{\varepsilon}{2}\right] < k\cdot\exp\left(-2\frac{\varepsilon^2}{4}|S_{k+1}|\right) < \frac{\delta}{2}$. Thus with probability of at most $1-\frac{\delta}{2}$, no h_i has a difference in error of more than $\frac{\varepsilon}{2}$ between the global true error ε and the emperical error on S_{k+1}, and thus no h_i has an emperical error greater than $\frac{3}{2}\varepsilon$.

Thus when we choose the h_k with the least error, with probability $1-\frac{\delta}{2}$ it will have an emperical error of at most $\frac{3}{2}\varepsilon$. Going again to the probabilistic bound of $\frac{1}{2}\varepsilon$ between the emperical and false error, this means that the true error is at most $\frac{3}{2}\varepsilon + \frac{1}{2}\varepsilon = 2\varepsilon$.

26.2.1.3 Final confidence boosting algorithm

For us to not choose an h with low enough error, either there must have not been one present (a failure of the first part) or we chose the wrong hypothesis (a failure of the second part). Both parts have a probability of $\delta/2$ of failure, so by union bound the probability of either failing is at most δ. This means that with probability at least $1-\delta$, we will get a good enough hypothesis.

With the preceding parts in place, we can give a formal presentation of the algorithm to boost confidence:

for i from 1 to k, where $k = \frac{1}{\delta'}\log\frac{2}{\delta}$:
    draw $S_i \sim D$, run A to find h_i
Draw $S_{k+1} \sim D$ with size $O(1/\epsilon^2 \log k/\delta)$ (ensuring that $\exp\left(-2\frac{\varepsilon^2}{4}|S_{k+1}|\right) \leq \frac{\delta}{2k}$)
Find errors $\text{error}_{S_{k+1}}(h_i)$ for each i
Output $\text{argmin}_{h_i} \text{error}_{S_{k+1}}(h_i)$

26.3 Error Reduction

In the previous section, we were interested in raising the confidence in our hypothesis. In this section, we study a boosting algorithm that decreases the error probability, given that the error probability of the hypothesis produced by our algorithm is bounded away from 1/2.

Suppose that with probability $1-\delta$, some algorithm A produces a hypothesis that has an error probability of no more than $1/2-\gamma$. For $\delta = 0$, we first show how to design an algorithm that produces (with probability $1-\delta$) a hypothesis that has an error probability of $1/2-\gamma'$ with $\gamma' > \gamma$. We then apply this procedure recursively to get an error probability of ϵ for any $\epsilon > 0$.

First consider the following approach that doesn't work. Following the ideas of probability amplification, we would be tempted to get k samples $S_1, \ldots, S_k$ from a distribution D, and use A to create k different hypotheses $h_1, \ldots, h_k$. Given some x taken from our distribution D, we will then compute $h_i(x)$ for all i and side with the majority. By Chernoff's bound, the majority vote is correct with high probability.

This approach doesn't work. The sample sets $S_1, \ldots, S_k$ are independent of each other; however, the resulting hypotheses $h_1, \ldots, h_k$ are correlated. Therefore, we cannot use Chernoff's bound to say that the majority vote is correct with high probability. Other concentration bounds discussed in class don't give us a good probability of the correctness of the majority vote either, so we will have to modify our approach.

Suppose that with probability 1, A gives a hypothesis h that has error probability $p = 1/2-\gamma$. We would like to decrease this error probability to $1/2-\gamma'$ with $\gamma' > \gamma$. In order to do that, we invoke A three times, each time with a slightly different distribution. Let D be our original distribution. We let D_1 be D, and use A on a sample S_1 from D_1 to get a hypothesis h_1. In the next round, we would like to focus more on the examples x for which h_1 errs. Unfortunately, we have no quick way of sampling from the subset of the examples that are marked incorrectly by h_1. We can skew the distribution so that the total weight of the correctly marked examples is 1/2, which makes the

total weight of the incorrectly marked examples 1/2 as well. We achieve this as follows:

$$D_2(x) = \begin{cases} \frac{D_1(x)}{2(1-p)} & \text{if } h_1(x) \text{ is correct,} \\ \frac{D_1(x)}{2p} & \text{if } h_1(x) \text{ is incorrect.} \end{cases}$$

We pick our second sample, S_2, from this distribution, and use A to come up with a hypothesis h_2. Finally, our last distribution, D_3, consists only of those examples x for which $h_1(x) \neq h_2(x)$ (examples on which h_1 and h_2 disagree). We take a sample S_3 from this distribution and use A to construct a hypothesis h_3 from it. Our final hypothesis then becomes $h = Maj(h_1, h_2, h_3)$.

We now bound the error of our new hypothesis. First assume that the hypotheses h_1, h_2, and h_3 are independent. Then the probability that both h_1 and h_2 err on x is p^2, and the probability that h_3 errs and that h_1 and h_2 disagree on input x is $2p^2(1-p)$. Therefore, the total probability of error is at most $3p^2 - 2p^3$. This is strictly less than p when $p \in (0, 1/2)$. Thus there exists $\gamma' > \gamma$ such that the error probability of our new hypothesis h is at most $1 - \gamma'$.

We now show that we get the same error probability when our three intermediate hypotheses are not independent. Call the probability that h_1 and h_2 both err on x m_1, and the probability that h_1 is correct and h_2 errs on x m_2. The probability that h_1 and h_2 disagree can then be epxressed as $p - m_1 + m_2$. To see that, first notice that p is the probability that h_1 errs. If we subtract m_1 from it, we get the probability that h_1 errs and h_2 doesn't. Finally, we add the probability that h_2 errs and h_1 doesn't. We can do this because the three events we mentioned are disjoint.

By construction of D_2, $\text{error}_{D_2}(h_2) = \frac{m_1}{2p} + \frac{m_2}{2(1-p)}$, which is equal to p by assumption. Then $m_1 = 2p^2 - \frac{m_2 p}{1-p}$. Now the probability that h errs on x is $m_1 + p(p - m_1 + m_2)$. The first term is the probability that h_1 and h_2 both err. The second term is the probability that h_3 errs and that h_1 and h_2 disagree. This is equal to $(1-p)m_1 + p^2 + pm_2 = (1-p)2p^2 - m_2 p + p^2 + pm_2 = 3p^2 - 2p^3$, which is what we wanted to show.

We can repeat this process recursively to get majorities of majorities. Figure 26.3.1 shows the recursion tree. When the recursion tree becomes high enough, we will get a hypothesis that makes an error with probability at most ϵ. At each level, we recursively apply the approach described earlier and take the majority of the majorities one level lower in the tree. We obtain the hypotheses that appear in the leaves by sampling from a distribution and running the algorithm A. We obtain the hypotheses that are in the nodes (labeled by the majority function in Figure 26.3.1) of the tree recursively. At a given node, we are given a distribution D. We use the first child of that node (we make a recursive call) to produce a hypothesis h_1 using $D_1 = D$ as the distribution. After that, we skew D_1 to get D_2 as discussed before, and make another recursive call to get a hypothesis h_2 using D_2 as the distribution (this is represented by the second child of a node). Finally, we construct the distribution D_3 and make our final recursive call to get a hypothesis h_3. We output the hypothesis that takes the majority of h_1, h_2, and h_3. The order in which we find the intermediate hypotheses is determined by a preorder traversal of the tree in Figure 26.3.1.

We now show that the tree has logarithmic depth. We will do this in two stages. First, we find the number of levels sufficient to get $\gamma' \geq 1/4$. After that, we assume $\gamma \geq 1/4$ and find the number of levels necessary to get $\gamma' = 1/2 - \epsilon$.

We saw that after getting three hypotheses, our new combined hypothesis has an error probability

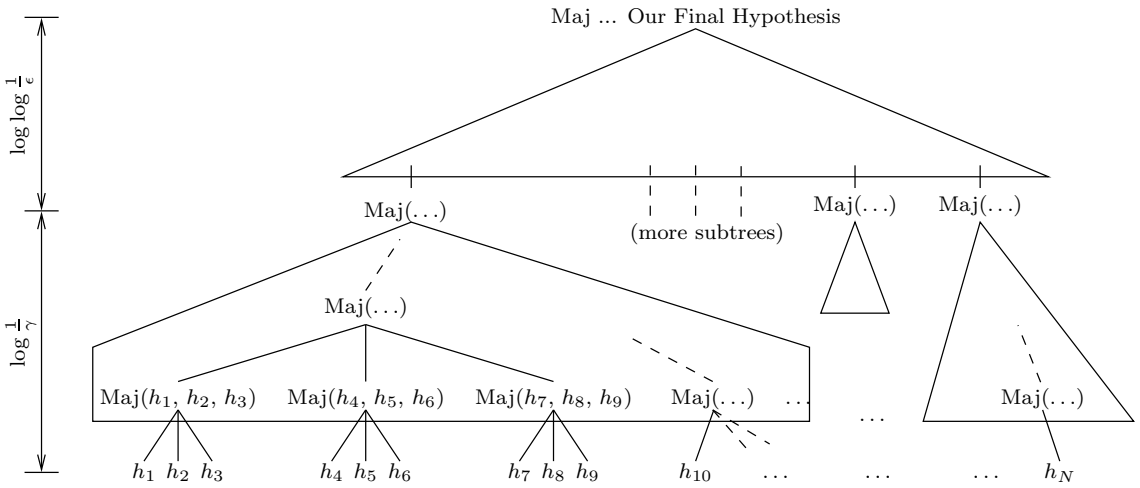


Figure 26.3.1: Recursion tree representing all the majorities we have to compute in order to create our final hypothesis. Each subtree in the bottom part corresponds to producing one hypothesis with error probability at most $1/2 - \gamma$ where $\gamma > 1/4$. The top of the tree then represents taking all these hypotheses and producing a hypothesis with error probability at most ϵ. The total height of the tree is $\mathcal{O}(\log \frac{1}{\gamma} + \log\log \frac{1}{\epsilon})$.

of $3p^2 - 2p^3 = p[3p - 2p^2]$. This means that to get an upper bound on the error probability of the new hypothesis, we multiply the error probability of the old hypothesis by $3p - 2p^2$. If we use the fact that $p = 1/2 - \gamma$, we get that the new error probability p' is at most $p \cdot 2(\frac{1}{2} - \gamma)(1 + \gamma) = (\frac{1}{2} - \gamma) \cdot 2(\frac{1}{2} - \gamma)(1 + \gamma)$. To get the new value of γ, we express $\gamma' = 1 - p'$ in terms of γ, which gives us $\gamma' = \gamma(\frac{3}{2} - 2\gamma^2)$, so we can obtain the next value of γ from the previous one if we multiply it by $\frac{3}{2} - 2\gamma^2$. Since $\gamma < 1/4$ by assumption, $\frac{3}{2} - 2\gamma^2 > \frac{11}{8}$. This means that each level of recursion increases the value of γ by a constant factor. To find the number of levels of recursion before γ is at least $1/4$, we solve the inequality $\gamma(\frac{11}{8})^k > \frac{1}{4}$ for k.

$$\begin{aligned} \gamma\left(\frac{11}{8}\right)^k &> \frac{1}{4} \\ \log\gamma + k\log\frac{11}{8} &> \log\frac{1}{4} \\ k &> \frac{\log\frac{1}{4} - \log\gamma}{\log\frac{11}{8}} \end{aligned}$$

We can rewrite the numerator of the last fraction as $\log\frac{1}{4} - \log\gamma = \log\frac{1}{4\gamma} = \log\frac{1}{\gamma} - \log 4 = \mathcal{O}(\log\frac{1}{\gamma})$. Since the denominator of that fraction is constant, we can conclude that $k = \mathcal{O}(\log\frac{1}{\gamma})$. Thus, to get $\gamma' > 1/4$ (and $p' < 1/4$), we need $\mathcal{O}(\log 1/\gamma)$ levels of recursion.

Now p is at most $1/4$, so $\gamma > 1/4$. After an additional level of recursion, we will have $p' = p(3p - 2p^2) < 3p^2$. Then let p_1 be the error probability of the hypothesis one level of recursion higher than the hypothesis with error probability $p < 1/4$, and define p_2 similarly using p_1 instead of p. In general, we have $p_i < 3p_{i-1}^2$. Then

$$
\begin{aligned}
p_1 &< 3p^2 = 3^1 p^2 < (\tfrac{3}{4})^1 p \\
p_2 &< 3p_1^2 < 3(3p^2)^2 = 3^3 p^4 < (\tfrac{3}{4})^3 p \\
p_3 &< 3p_2^2 < 3(3p_1^2)^2 = 3^3 p_1^4 < 3^3(3p^2)^4 = 3^7 p^8 < (\tfrac{3}{4})^7 p \\
&\vdots \\
p_k &< (\tfrac{3}{4})^{2^k-1} p
\end{aligned}
$$

The maximum value of p is $1/4$, and we would like $p_k < \epsilon$. To find a lower bound on k, we need to solve the inequality below.

$$
\begin{aligned}
\frac{1}{4}\left(\frac{3}{4}\right)^{2^k-1} &< \epsilon \\
(2^k-1)\log\frac{3}{4} &< \log 4\epsilon \\
2^k &> \frac{\log 4\epsilon + \log\frac{3}{4}}{\log\frac{3}{4}} \\
k &> \log\left(\frac{\log 4\epsilon + \log\frac{3}{4}}{\log\frac{3}{4}}\right) \\
&> \log\left(\frac{\log 4\epsilon + \log\frac{3}{4}}{-1}\right) \\
&> \log\left(-\log\frac{3}{4} - \log 4\epsilon\right) \\
&> \log\left(-\log\frac{3}{4} + \log\frac{1}{4\epsilon}\right) \\
&> \log\left(-\log\frac{3}{4} + \log\frac{1}{\epsilon} - \log 4\right) \\
&= \mathcal{O}\left(\log\log\frac{1}{\epsilon}\right).
\end{aligned}
$$

Therefore, we need additional $\log\log 1/\epsilon$ levels of recursion to turn γ into $1/2-\epsilon$ and get a hypothesis with error probability of at most ϵ.

We have left out the discussion of sampling from D_2 and D_3. It can be shown that we can sample from D_2 and D_3 efficiently if p is not too close to 0 or 1. Since p is at most $1/2$, the only case we have to worry about is when p is close to 0. But if p is close to zero, we can stop and use the hypothesis whose error probability is p (close to zero) as our hypothesis that has low error probability.

26.4 Closing Remarks

The recursive approach from the previous section requires us to take too many samples. It turns out that we can repeat the process of picking D_2 by creating the distribution D_{i+1} from D_i and

$h_i$using

$$D_{i+1}(x) = \begin{cases} \frac{D_i(x)}{2(1-p)} & \text{if } h_i(x) \text{ is correct,} \\ \frac{D_i(x)}{2p} & \text{if } h_i(x) \text{ is incorrect.} \end{cases} .$$

Intuitively, in each step, we give all the incorrectly marked examples more weight before we take another sample and produce another hypothesis.

After repeating this process for $(1/\epsilon^2)\log(1/\gamma)$ steps, we let our final hypothesis h be the majority vote of the hypotheses created so far. This boosting algorithm is called Adaboost. It can be shown that the hypothesis produced by Adaboost makes an error with probability at most ϵ. Adaboost also yields a much simpler hypothesis, namely a majority of one set of hypotheses, whereas the algorithm we explain in Section 26.3 produces a complex hypothesis that requires us to recursively find majorities of majorities of smaller sets of hypotheses.

26.5 Next Time

So far we have pushed the discussion of sampling from various distributions aside. In the next lecture, we will talk more about picking samples from various distributions. After that, we will talk about random walks and Markov chains.

References

[1] M. Kearns and U. Vazirani. An Introduction to Computational Learning Theory *The MIT Press*, 1994.

CS787: Advanced Algorithms

Scribe: Blayne Field | **Lecturer:** Shuchi Chawla
Topic: Random Walks and Markov Chains | **Date:** November 26, 2007

27.1 Introduction

As we have seen in the course up to this point, randomization has been a useful tool for developing simple and efficient algorithms. So far, most of these algorithms have used some sort of independent coin tosses. In this lecture, we started looking at choosing a sample from a large space, with our choices being dependent on previous choices. We looked at this concept through the framework of random walks and Markov chains.

Definition 27.1.1 *A random walk is a process in which at every step we are at a node in an undirected graph and follow an outgoing edge chosen uniformly at random. A Markov chain is similar, except the outgoing edge is chosen according to an arbitrary distribution.*

27.1.1 Examples of random walks

Random walks can occur in many situations, and are useful for analyzing various scenarios. Consider the following betting game: a player bets \$1, and either loses it or wins an addition dollar with probability $\frac{1}{2}$. Since the probability of either thing happening is equal, we can think of this as a random walk on a line graph, where each node represents the amount of wealth at any point of time. This allows us to learn about numerous aspects of the game, such as the probability distribution of the amount of money at a given time. We can also ask about the probability of the player running out of money before winning a certain amount, and if that happens, what is the expected amount of time before that happens.

Another situation where random walks occur is shuffling a deck of cards. One possible way of drawing a permutation u.a.r. is to start at an arbitrary permutation and apply a local shuffling operation multiple times. This can be thought of as a random walk on a graph of all permutations. Our graph would contain nodes for each of the 52! permutations, and the connectivity of the graph would be determined by what local operations are allowed. For example, if we could just randomly choose 1 card, and move it to the top, each node would have degree 52 (if we include self-loops). Analyzing this random walk may allow us to determine how many local steps we would need to take in order to be reasonably close to the uniform distribution.

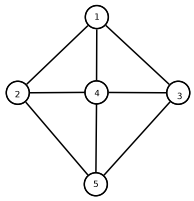
27.2 Properties of random walks

There are numerous properties of random walks that we may be interested in. They are listed and defined informally below:
Stationary distribution: what is the distribution of our current location if we run the random walk for an infinite number of steps.
Hitting time: denoted h_{uv} is the expected time to get from u to v.

Commute time: denoted C_{uv} is the expected time to get from u to v, and back to u
Cover time (starting at u): denoted C_u is the expect time to visit every node (starting at node u)
Cover time (for a graph): denoted $C(G) = \max_u C_u$.

27.3 Transition matrix

A random walk (or Markov chain), is most conveniently represented by its transition matrix P. P is a square matrix denoting the probability of transitioning from any vertex in the graph to any other vertex. Formally, P_{uv} = Pr[going from u to v, given that we are at u]. Thus for a random walk, $P_{uv} = \frac{1}{d_u}$ if $(u, v) \in E$, and 0 otherwise (where d_u is the degree of u). Thus for the example graph given below the transition matrix would be:



$$\mathbf{P} = \begin{pmatrix} 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & 0 \\ \frac{1}{3} & 0 & 0 & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & 0 & 0 & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & \frac{1}{4} \\ 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & 0 \end{pmatrix}$$

27.4 Stationary distributions

Thus if we have a distribution π over the nodes, we can obtain the distribution after one step by computing $\pi' = P^T \cdot \pi$. Using this definition, we can define a stationary distribution π_s as a distribution with the property that $P^T \cdot \pi_s = \pi_s$. Stationary distributions are not always unique, but under certain conditions, they are. It also is the case that under certain conditions, $lim_{t\to\infty}(P^T)^t\pi = \pi_s$ for all starting distributions π. We will discuss these conditions in the next lecture.

For the following examples, consider a graph $G = (V, E)$, with $n = |V|$ and $m = |E|$. Let d_u denote the degree of vertex u.

Lemma 27.4.1 *$\pi_v = \frac{d_v}{2m}$ is a stationary distribution for G*

Proof: $(P^T \cdot \pi)_u = \sum_v P_{vu}\pi_v = \sum_{v:(v,u)\in E} \frac{d_v}{2m} * \frac{1}{d_v} = \sum_{v:(v,u)\in E} \frac{1}{2m} = \pi_u$ (because $P_{vu} = \frac{d_v}{2m}$ if $(v, u) \in E$ and 0 otherwise). ■

27.5 Random walks on edges

For our next result, we will think about random walks in a slightly different way, where instead of walking from node to node, our walk goes from edge to edge. Whenever we are on an edge uv, we choose the next edge u.a.r from the set of edges incident to v. Then we can see that the uniform distribution on edges ($\pi_{u\to v} = \frac{1}{2m}\ \forall(u \to v) \in E$) is a stationary distribution. This is because $(P^T \cdot \pi)_{v\to w} = \sum_{u:(u,v)\in E} \frac{1}{2m}\frac{1}{d_v} = \frac{1}{2m} = \pi_{v\to w}$

Lemma 27.5.1 *$\forall(u, v) : (u, v) \in E$, we have $C_{uv} \leq 2m$*

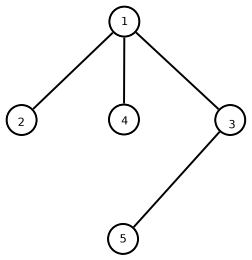
Proof: Note: this is just a sketch of the proof, as certain technical details are omitted. If we view the process as a random walk on sequence of edges, we can bound the commute time by the expected amount of time between consecutive occurences of the edge $u \to v$. The expected length of the gap between consecutive occurences if we run for t steps is simply t divided by the actual number of times we see the edge $u \to v$. We also know that since the stationary distribution is uniform, we expect to see the edge $\frac{t}{2m}$ times. As t goes to infinity, the actual number of times we see $u \to v$ appraoches its expectation $\frac{t}{2m}$ with probability 1 (due to the law of large numbers). We can then approximate the actual number seen by the expected number seen, and thus we expect the length of the gap to be $\frac{t}{\frac{t}{2m}} = 2m$. ■

27.6 Cover time

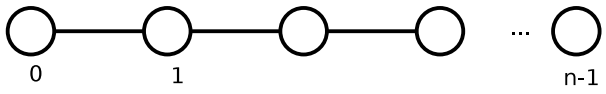
If we had a bound on the commute time for all pairs (u, v) (call this bound x), we could get a bound (in expectation) on the cover time by running the random walk for $x * n$ steps. Unfortunately, the bound given by lemma 27.5.1 is only valid for pairs (u, v) where there is an edge between u and v. However, we can still come up with a different method for bounding the cover time.

Lemma 27.6.1 $C(G) \leq 2m(n-1)$

Proof: Let T be an arbitrary spanning tree of G. For each edge (u, v) in T, add the edge (v, u). We can then bound the cover time of G with the expected time needed to complete an Euler tour of T. Since each node in T has even degree (due to the doubling of the edges), we know that an Euler tour must exist. If we list the vertices visited as $v_1, v_2, \ldots, v_k = v_0$, we have $C(G) \leq h(v_0v_1) + h(v_1v_2) + \ldots + h(v_{k-1}v_0) = \sum_{uv\in T} h(u, v) + h(v, u) = \sum_{uv\in T} C_{uv} \leq 2m(n-1)$, since each of the $(n-1)$ edges that was in T orginally shows up in both directions. For example, the cover time of the graph given before could be bounded by using the spanning tree below, so $C(G) \leq h_{21} + h_{14} + h_{41} + h_{13} + h_{35} + h_{53} + h_{31} + h_{12}$. ■

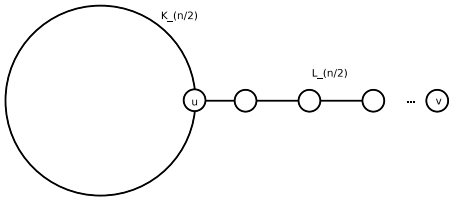


It turns out that for some graphs, the bound given in lemma 27.6.1 is tight. One example of this is the line graph with n vertices, depicted below (L_n). According to the lemma, $C(G) \leq 2(n-1)^2 = O(n^2)$. Also, we can note that $h_{1,0} \leq 2(n-1)-1$, since $h_{0,1} = 1$, and $C_{uv} = h_{uv} + h_{vu}$ by linearity of expectation. We will show in the next lecture that the cover time for this graph is indeed $\Theta(n^2)$.



However, the bound is not always tight, as in the case of the complete graph, K_n. In this case, $m = \frac{n(n-1)}{2}$, so $C(K_n) = O(n^3)$ by the lemma. However, we can get a much tighter bound on the cover time for this graph. Since in the complete graph, we can go from any node to any other in one step, the problem of visiting all nodes can be viewed as an instance of the coupon collector problem. As we saw in a previous lecture, this would give us a bound of $O(n \log n)$.

One last example is the lollipop graph (pictured below) , which has n vertices, half of which form $K_{\frac{n}{2}}$, with the remainder forming $L_{\frac{n}{2}}$ (and attached to the complete graph portion). The lemma in this case gives $C(G) = O(n^3)$, which happens to be tight. This is because it takes $\Omega(n^3)$ time to get from u to v. We can see this by the following analysis: for just the line graph, it should take $\Omega(n^2)$ steps, $\Omega(n)$ of which will be spent at u, since the nodes should approach a uniform distribution. However, if we are at u, there is a $\frac{2}{n}$ probability of leaving the clique, see we need to visit u $\Omega(n)$ times to 'escape' back to the line graph. However, if we are in the clique portion it takes $\Omega(n)$ steps to get back to u. Thus each time we end up in u from the line graph, we expect to take $\Omega(n^2)$ steps to get back into the line graph. Thus the expected number of steps is $\Omega(n^3)$. This illustrates that the number of edges doesn't always give the best estimate, since both this and the last example had $O(n^2)$ edges.



27.7 Testing s-t connectivity

Since any graph has at most $O(n^2)$ edges, the cover time is $O(n^3)$ for any graph. Thus we can use random walks to test for s-t connectivity using very smalls amounts of space. This problem can be solved using either BFS or DFS, but they use $\Omega(n)$ space. This bound on the cover time suggests a simple randomized algorithm for testing s-t connectivity. We just run a random walk starting at s for $2n^3$ steps and output 'connected' if we ever encounter t on our walk. This algorithm will succeed with probability $\frac{1}{2}$, so if we desired gives an upper bound on the expected time of our algorithm, and allows us to use $\log n$ bits to store our current location in the walk, and $O(\log n)$

bits as a counter for our time in the walk. This shows that $s - t$ connectivity can be solved in randomized log space. It was recently shown that the procedure can be derandomized, while still still using logarithmic space.

CS787: Advanced Algorithms

Scribe: Evan Driscoll, Daniel Wong **Lecturer:** Shuchi Chawla
Topic: Random Walks & Markov chains: the resistance method. **Date:** December 7, 2007

28.1 Introduction

For this lecture, consider a graph $G = (V, E)$, with $n = |V|$ and $m = |E|$. Let d_u denote the degree of vertex u.

Recall some of the quantities we were interested in from last time:

Definition 28.1.1 *The transition matrix is a matrix P such that P_{uv} denotes the probability of moving from u to v: $P_{uv} = \mathbf{Pr}[$random walk moves from u to v given it is at $u]$.*

Definition 28.1.2 *Stationary Distribution for the graph starting at v, π^*, is the distribution over nodes such that $\pi^* = \pi^* P$*

Definition 28.1.3 *The hitting time from u to v, h_{uv}, is the expected number of steps to get from u to v.*

Definition 28.1.4 *The commute time between u and v, C_{uv}, is the expected number of steps to get from u to v and then back to u.*

Definition 28.1.5 *The cover time of a graph, $C(G)$, is the maximum over all nodes in G, of the expected number of steps starting at a node and walking to every other node in G.*

Last time we showed the following:

- For a random walk over an undirected graph where each vertex v has degree d_v, $\pi^*_{2m} = \frac{d_v}{2m}$ is a stationary distribution
- In any graph, we can bound the commute time between adjacent nodes: if $(u, v) \in E$ then $C_{uv} \leq 2m$
- In any graph, we can bound the cover time: $C(G) \leq 2m(n-1)$

28.2 Resistive Networks

Recall the relationships and values in electrical circuits.

The 3 principle values we will be looking at are voltage, V, current, i, and resistance r. For more information on the intuition of these properties I direct the reader to the wikipeida entry on Electrical circuits, http://en.wikipedia.org/wiki/Electrical_circuits.

Ohm's Law:

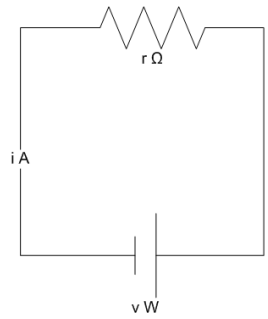
Definition 28.2.1 $V = ir$

Kirchoff's Law:

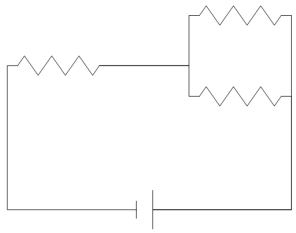
Definition 28.2.2 *At any junction,* $i_{in} = i_{out}$

In other words, current is conserved.

Here is an example circuit diagram.



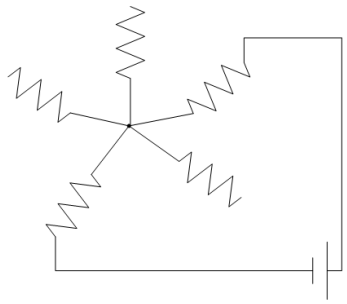
Slightly more complicated circuit:



Recall that for resistors in series the net resistance is the sum of the individual resistances, $r_{net} = \sum_{r \text{ in series}} r$. Similarly for resistors in parallel the multiplicative inverse of the net resistance is the sum of the multiplicative inverses of each individual resistor, $\frac{1}{r_{net}} = \sum_{r \text{ in parallel}} \frac{1}{r}$.

28.3 Analysis of Random Walks with Resistive Networks

Consider an undirected unweighted graph G. Replace every edge by a 1Ω resistor. Let R_{uv} be the effective resistance between nodes u and v in this circuit.



As you can see by applying these operations to a star graph, the effective resistance is 2Ω between the indicated u and v.

Lemma 28.3.1 $C_{uv} = 2mR_{uv}$

Proof:

The proof of Lemma 28.3.1 will be shown by considering two schemes for applying voltages in resistive networks and then showing that the combination of the two schemes show the lemma.

Part 1. We will analyze what would happen if we connect v to ground, then apply a current to each other vertex w of amount d_w amps. (d_w is the degree of w.) The amount of current that flows into the ground at v is $2m - d_v$, since each edge contributes one amp at each end. Let ϕ_w be the voltage at node w.

Consider each neighbor w' of w. There is a 1Ω resistor going between them. By Ohm's law, the current across this resistor is equal to the voltage drop from w to w', which is just $\phi_w - \phi_{w'}$. Look at the sum of this quantity across all of w's neighbors:

$$d_w = \sum_{w':(w,w')\in E} (\phi_w - \phi_{w'}) = d_w\phi_w - \sum_{w':(w,w')\in E} \phi_{w'}$$

Rearranging:

$$\phi_w = 1 + \frac{1}{d_w} \sum_{w':(w,w')\in E} \phi_{w'} \tag{28.3.1}$$

At this point, we will take a step back from the interpretation of the graph as a circuit. Consider the hit time h_{wv} in terms of the hit time of w's neighbors, $h_{w'v}$. In a random walk from w to v, we will take one step to a w' (distributed with probability $1/d_w$ to each w'), then try to get from w' to v. Thus we can write h_{uv} as:

$$h_{wv} = 1 + \frac{1}{d_w} \sum_{w':(w,w')\in E} h_{w'v} \tag{28.3.2}$$

However, note that equation (28.3.2) is the same as equation (28.3.1)! Because of this, as long as these equations have a unique solution, $h_{uv} = \phi_w$. We will argue that this is the case. The voltage at a node is one more than the average voltage of its neighbors. Consider two solutions $\phi^{(1)}$ and $\phi^{(2)}$. Look at the vertex w where $\phi_w^{(1)} - \phi_w^{(2)}$ is largest. Then one of the neighbors of w must also have a large difference because of the average. In both solutions, $\phi_v = 0$, so the difference in v's neighbors has to average out to zero.

Part 2. We will now analyze what happens with a different application of current. Instead of applying current everywhere (except v) and drawing from v, we will apply current at u and draw from everywhere else.

We are going to apply $2m - d_u$ amps at u, and pull d_w amps for all $w \neq u$. (We continue to keep v grounded.) Let the voltage at node w under this setup be ϕ'_w.

Through a very similar argument, $h_{wu} = \phi'_u - \phi'_w$. Thus $h_{vu} = \phi'_u - 0 = \phi'_u$.

Part 3. We will now combine the conclusions of the two previous parts. At each node w, apply $\phi_w + \phi'_w$ volts. We aren't changing resistances, so currents also add. This means that each w ($\neq u$

and $\neq v$) has no current flowing into or out of it, and the only nodes with current entering or exiting are u and v.

At v, $2m - d_v$ amps were exiting during part 1, and d_v amps were exiting during part 2, which means that now $2m$ amps are exiting. By a similar argument (and conservation of current), $2m$ amps are also entering u.

Thus the voltage drop from u to v is given by Ohm's law:

$$(\phi_u + \phi'_u) - 0 = R_{uv} \cdot 2m$$

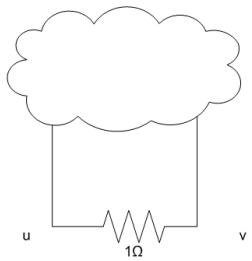
But $\phi_u = h_{uv}$ and $\phi'_u = h_{vu}$, so that gives us our final goal:

$$h_{uv} + h_{vu} = C_{uv} = 2mR_{uv}$$

■

28.4 Application of the resistance method

A couple of the formulas we developed last lecture can be re-derived easily using lemma 28.3.1. Lemma 27.5.1 says that, for any nodes u and v, if there is an edge $(u, v) \in E$, then $C_{uv} \leq 2m$. This statement follows immediately by noting that $R_{uv} \leq 1\Omega$. If a 1Ω resistor is connected in parallel with another circuit (for instance, see the following figure), the effective resistance R_{uv} is less than the minimum of the resistor and the rest of the circuit.



In addition, last lecture we showed (in lemma 27.6.1) that $C(G) \leq 2m(n-1)$. We can now develop a tighter bound:

Theorem 28.4.1 *Let $R(G) = \max_{u,v \in V} R_{u,v}$ be the maximum resistance between any two points. Then $mR(G) \leq C(G) \leq mR(G)2e^3 \ln n + n$.*

Proof: The lower bound is fairly easy to argue. Consider a pair of nodes, (u, v), that satisfy $R_{uv} = R(G)$. Then $\max\{h_{uv}, h_{vu}\} \geq C_{uv}/2$ because either h_{uv} or h_{vu} makes up at least half of the commute time. Lemma 28.3.1 and the above inequality shows the lower bound.

To show the upper bound on $C(G)$, we proceed as follows. Consider running a random walk over G starting from node u. Run the random walk for $2e^3 mR(G)$ steps. For some vertex v, the chance that we have not seen v is $1/e^3$. We know that from 28.3.1 the hitting time from any u to v is at

most $2mR(G)$. From Markov's inequality:

$$\begin{aligned}\mathbf{Pr}\big[\#\text{ of steps it takes to go from } u \text{ to } v \geq 2e^3 mR(G)\big] &\leq \frac{\mathbf{E}[\#\text{ of steps it takes to go from } u \text{ to } v]}{2e^3 mR(G)} \\ &\leq \frac{2mR(G)}{2e^3 mR(G)} \\ &\leq \frac{1}{e^3}\end{aligned}$$

(Note that this holds for any starting node $u \in V$.)

If we perform this process $\ln n$ times — that is, we perform $\ln n$ random walks starting from u ending at u' the probability that we have not seen v on *any* of the walks is $(1/e^3)^{\ln n} = 1/n^3$. Because $h_{uv} \leq 1/e^3$ for all u, we can begin each random walk at the last node of the previous walk. By union bound, the chance that there exists a node that we have not visited is $1/n^2$.

If we have still not seen all the nodes, then we can use the algorithm developed last time (generating a spanning tree then walking it) to cover the graph in an expected time of $2n(m-1) \leq 2n^3$.

Call the first half of the algorithm (the $\ln n$ random walks) the "goalless portion" of the algorithm, and the second half the "spanning tree portion" of the algorithm.

Putting this together, the expected time to cover the graph is:

$$\begin{aligned}C(G) &\leq \mathbf{Pr}[\text{goalless portion reaches all nodes}] \cdot (\text{time of goalless portion}) \\ &\quad + \mathbf{Pr}[\text{goalless portion omits nodes}] \cdot (\text{time of spanning tree portion}) \\ &\leq \left(1 - \frac{1}{n^2}\right) \cdot (2e^3 mR(G) \cdot \ln n) + (1/n^2) \cdot (n^3) \\ &\leq 2e^3 mR(G) \ln n + n\end{aligned}$$

■

28.5 Examples

28.5.1 Line graphs

Last lecture we said that $C(G) = O(n^2)$. We now have the tools to show that this bound is tight. Consider u at one end of the graph and v at the other; then $R_{uv} = n - 1$, so by lemma 28.3.1, $C_{uv} = 2mR_{uv} = 2m(n-1)$, which is exactly what the previous bound gave us.

28.5.2 Lollipop graphs

Last lecture we showed that $C(G) = O(n^3)$. Consider u at the intersection of the two sections of the graph, and v at the end. Then $R_{uv} = n/2$, so $C_{uv} = 2m\frac{n}{2} = 2\Theta(n^2)\frac{n}{2} = \Theta(n^3)$. Thus again our previous big-O bound was tight.

28.6 Application of Random Walks

We conclude with an example of using random walks to solve a concrete problem. The 2-SAT problem consists of finding a satisfying assignment to a 2-CNF formula. That is, the formula takes the form of $(x_1 \vee x_2) \wedge (x_3 \vee \overline{x_1}) \wedge \ldots$. Let n be the number of clauses.

The algorithm works as follows:

1. Begin with an arbitrary assignment
2. If the formula is satisfied, halt
3. Pick any unsatisfied clause
4. Pick one of the variables in that clause UAR and invert it's value
5. Return to step 2

Each step of this algorithm is linear in the length of the formula, so we just need to figure out how many iterations we expect to have before completing.

This algorithm can be viewed as performing a random walk on a line graph with $n + 1$ nodes. Each node corresponds to the number of variables in the assignment that differ from a satisfying assignment (if one exists). When we invert some x_i, either we change it from being correct to incorrect and we move one node away from 0, or we change it from being incorrect to being correct and move one step closer to the 0 node.

However, there is one problem with this statement, which is that our results are for random walks where we take any outgoing edge uniformly. Thus we should argue that the probability of taking each edge out of a node is $\frac{1}{2}$. In the case where the algorithm chooses a clause with both a correct and incorrect variable, the chances in fact do work out to be $\frac{1}{2}$ in each direction. In the case where the algorithm chooses a clause where both variables are incorrect, it will *always* move towards the 0 node. Thus the probability the algorithm moves toward 0 is at least $\frac{1}{2}$. It may be better, but that only biases the results in favor of shorter running times.

Thus the probability of the random walk proceeding from i to $i-1$, and hence closer to a satisfying assignment, is at least $1/2$. Because of this, we can use the value of the hitting time we developed for line graphs earlier. Hence the number of iterations of the above we need to perform, is $O(n^2)$.

28.7 Next time

Next time we will return to a question brought up when we were beginning discussions of random walks, which is how fast do we converge to a stationary distribution, and under what conditions are stationary distributions unique.