6.854 Advanced Algorithms

Lecture 1: September 7, 2005 Lecturer: David Karger

Scribes: David G. Andersen, Ioana Dumitriu, John Dunagan, Akshay Patil (2003)

Fibonaccci Heaps

1.1 Motivation and Background

Priority queues are a classic topic in theoretical computer science. As we shall see, Fibonacci Heaps provide a fast and elegant solution. The search for a fast priority queue implementation is motivated primarily by two network optimization algorithms: Shortest Path and Minimum Spanning Tree (MST).

1.1.1 Shortest Path and Minimum Spanning Trees

Given a graph $G(V, E)$ with vertices V and edges E and a length function $l : E \rightarrow \Re^+$. We define the Shortest Path and MST problems to be, respectively:

shortest path. For a fixed source $s \in V$, find the shortest path to all vertices $v \in V$

minimum spanning tree (MST). Find the minimum length set of edges $F \subset E$ such that F connects all of V.

Note that the MST problem is the same as the Shortest Path problem, except that the source is not fixed. Unsurprisingly, these two problems are solved by very similar algorithms, Prim's for MST and Djikstra's for Shortest Path. The algorithm is:

1. Maintain a priority queue on the vertices
2. Put s in the queue, where s is the start vertex (Shortest Path) or any vertex (MST). Give s a key of 0.
3. Repeatedly delete the minimum-key vertex v from the queue and mark it "scanned"

   For each neighbor w of v:

   If w is not in the queue and not scanned, add it with key:

   - Shortest Path: $key(v) + length(v \rightarrow w)$
   - MST: $length(v \rightarrow w)$

   If, on the other hand, w is in the queue already, then decrease its key to the minimum of the value calculated above and w's current key.

1.1.2 Heaps

The classical answer to the problem of maintaining a priority queue on the vertices is to use a binary heap, often just called a heap. Heaps are commonly used because they have good bounds on the time required for the following operations:

insert	$O(\log n)$
delete-min	$O(\log n)$
decrease-key	$O(\log n)$

If a graph has n vertices and m edges, then running either Prim's or Djikstra's algorithms will require $O(n \log n)$ time for inserts and deletes. However, in the worst case, we will also perform m decrease-keys, because we may have to perform a key update every time we come across a new edge. This will take $O(m \log n)$ time. Since the graph is connected, $m \geq n$, and the overall time bound is given by $O(m \log n)$.

Since $m \geq n$, it would be nice to have cheaper key decreases. A simple way to do this is to use *d-heaps.*

1.1.3 d-Heaps

d-heaps make key reductions cheaper at the expense of more costly deletions. This trade off is accomplished by replacing the binary heap with a d-ary heap—the branching factor (the maximum number of children for any node) is changed from 2 to d. The depth of the tree then becomes $\log_d(n)$. However, delete-min operations must now traverse all of the children in a node, so their cost goes up to $d \log_d(n)$. Thus, the running time of the algorithm becomes $O(nd \log_d(n) + m \log_d(n))$. Choosing the optimal $d = m/n$ to balance the two terms, we obtain a total running time of $O(m \log_{m/n} n)$.

When $m = n^2$, this is $O(m)$, and when $m = n$, this is $O(n \log n)$. This seems pretty good, but it turns out we can do much better.

1.1.4 Amortized Analysis

Amortized analysis is a technique for bounding the running time of an algorithm. Often we analyse an algorithm by analyzing the individual operations that the algorithm performs and then multiplying the total number of operations by the time required to perform an operation. However, it is often the case that an algorithm will on occasion perform a very expensive operation, but most of the time the operations are cheap. Amortized analysis is the name given to the technique of analyzing not just the worst case running time of an operation but the average case running time of an operation. This will allow us to balance the expensive-but-rare operations against their cheap-and-frequent peers.

There are several methods for performing amortized analysis; for a good treatment, see *Introduction to Algorithms* by Cormen, Leiserson, and Rivest. The method of amortized analysis used to analyze Fibonacci heaps is the potential method:

- Measure some aspect of the data structure using a potential function. Often this aspect of the data structure corresponds to what we intuitively think of as the complexity of the data structure or the amount by which it is out of kilter or in a bad arrangement.

- If operations are only expensive when the data structure is complicated, and expensive operations can also clean up ("uncomplexify") the data structure, and it takes many cheap operations to noticeably increase the complexity of the data structure, then we can *amortize* the cost of the expensive operations over the cost of the many cheap operations to obtain a low average cost.

Therefore, to design an efficient algorithm, we want to force the user to perform many operations to make the data structure complicated, so that the work doing the expensive operation and cleaning up the data structure is amortized over those many operations.

We compute the *potential* of the data structure by using a potential function Φ that maps the data structure (DS) to a real number $\Phi(DS)$. Once we have defined Φ, we calculate the cost of the i^{th} operation by:

$$cost_{amortized}(operation_i) = cost_{actual}(operation_i) + \Phi(DS_i) - \Phi(DS_{i-1})$$

where DS_i refers to the state of the data structure after the i^{th} operation. The sum of the amortized costs is then

$$\sum cost_{actual}(operation_i) + \Phi_{final} - \Phi_{initial}$$

.

If we can prove that $\Phi_{final} \geq \Phi_{initial}$, then we've shown that the amortized costs bound the real costs, that is, $\sum cost_{amortized} \geq \sum cost_{actual}$. Then we can just analyze the amortized costs and show that this isn't too much, knowing that our analysis is useful. Most of the time it is obvious that $\Phi_{final} \geq \Phi_{initial}$ and the real work is in coming up with a good potential function.

1.2 Fibonacci Heaps

The Fibonacci heap data structure invented by Fredman and Tarjan in 1984 gives a very efficient implementation of the priority queues. Since the goal is to find a way to minimize the number of operations needed to compute the MST or SP, the kind of operations that we are interested in are *insert*, *decrease-key*, *merge*, and *delete-min.* (We haven't covered why *merge* is a useful operation yet, but it will become clear.) The method to achieve this minimization goal is laziness – **"do work only when you must, and then use it to simplify the structure as much as possible so that your future work is easy"**. This way, the user is forced to do many cheap operations in order to make the data structure complicated.

Fibonacci heaps make use of heap-ordered trees. A heap-ordered tree is one that maintains the *heap property*, that is, where $key(parent) \leq key(child)$ for all nodes in the tree.

A Fibonacci heap H is a collection of heap-ordered trees that have the following properties:

1. The roots of these trees are kept in a doubly-linked list (the "root list" of H);
2. The root of each tree contains the minimum element in that tree (this follows from being a heap-ordered tree);
3. We access the heap by a pointer to the tree root with the overall minimum key;
4. For each node x, we keep track of the *rank* (also known as the *order* or *degree*) of x, which is just the number of children x has; we also keep track of the *mark* of x, which is a Boolean value whose role will be explained later.

For each node, we have at most four pointers that respectively point to the node's parent, to one of its children, and to two of its siblings. The sibling pointers are arranged in a doubly-linked list (the "child list" of the parent node). Of course, we haven't described how the operations on Fibonacci heaps are implemented, and their implementation will add some additional properties to H. Here are some elementary operations used in maintaining Fibonacci heaps.

1.2.1 Inserting, merging, cutting, and marking.

Inserting a node x. We create a new tree containing only x and insert it into the root list of H; this is clearly an $O(1)$ operation.

Merging two trees. Let x and y be the roots of the two trees we want to merge; then if the key in x is no less than the key in y, we make x the child of y; otherwise, we make y the child of x. We update the appropriate node's rank and the appropriate child list; this takes $O(1)$ operations.

Cutting a node. If x is a root in H, we are done. If x is not a root in H, we remove x from the child list of its parent, and insert it into the root list of H, updating the appropriate variables (the rank of the parent of x is decremented, etc.). Again, this takes $O(1)$ operations. (We assume that when we want to find a node, we have a pointer hanging around that accesses it directly, so actually finding the node takes $O(1)$ time.)

Marking. We say that x is marked if its mark is set to "true", and that it is unmarked if its mark is set to "false". A root is always unmarked. We mark x if it is not a root and it loses a child (i.e., one of its children is cut and put into the root-list). We unmark x whenever it becomes a root. We will make sure later that no marked node loses another child before it itself is cut (and reverted thereby to unmarked status).

1.2.2 Decreasing keys and Deleting mins

At first, *decrease-key* does not appear to be any different than *merge* or *insert*; just find the node and cut it off from its parent, then insert the node into the root list with a new key. This requires removing it from its parent's child list, adding it to the root list, updating the parent's rank, and (if necessary) the pointer to the root of smallest key. This takes $O(1)$ operations.

The *delete-min* operation works in the same way as *decrease-key*: Our pointer into the Fibonacci heap is a pointer to the minimum keyed node, so we can find it in one step. We remove this root of smallest key, add its children to the root-list, and scan through the linked list of all the root nodes to find the new root of minimum key. Therefore, the cost of a *delete-min* operation is $O(\#$ of children $)$ of the root of minimum key plus $O(\#$ of root nodes$)$; in order to make this sum as small as possible, we have to add a few bells and whistles to the data structure.

1.2.3 Population Control for Roots

We want to make sure that every node has a small number of children. This can be done by ensuring that the total number of descendants of any node is exponential in the number of its children. In the absence of any "cutting" operations on the nodes, one way to do this is by **only** merging trees that have the same number of children (i.e, the same rank). It is relatively easy to see that if we only merge trees that have the same rank, the total number of descendants (counting onself as a descendant) is always $(2^{\#\text{ of children}})$. The resulting structure is called a binomial tree because the number of descendants at distance k from the root in a tree of size n is exactly $\binom{n}{k}$. Binomial heaps preceded Fibonacci heaps and were part of the inspiration for them. We now present Fibonacci heaps in full detail.

1.2.4 Actual Algorithm for Fibonacci Heaps

- Maintain a list of heap-ordered trees.
- *insert*: add a degree 0 tree to the list.
- *delete-min*: We can find the node we wish to delete immediately since our handle to the entire data structure is a pointer to the root with minimum key. Remove the smallest root, and add its children to the list of roots. Scan the roots to find the next minimum. Then consolidate all the trees (merging trees of equal rank) until there is ≤ 1 of each rank. (Assuming that we have achieved the property that the number of descendants is exponential in the number of children for any node, as we did in the binomial trees, no node has rank $> c \log n$ for some constant c. Thus consolidation leaves us with $O(\log n)$ roots.) The consolidation is performed by allocating buckets of sizes up to the maximum possible rank for any root node, which we just showed to be $O(\log n)$. We put each node into the appropriate bucket, at cost $O(\log n) + O(\#$ of roots$)$. Then we march through the buckets, starting at the smallest one, and consolidate everything possible. This again incures cost $O(\log n) + O(\#$ of roots$)$.
- *decrease-key*: cut the node, change its key, and insert it into the root list as before, Additionally, if the parent of the node was unmarked, mark it. If the parent of the node was marked, cut it off also. Recursively do this until we get up to an unmarked node. Mark it.

1.2.5 Actual Analysis for Fibonacci Heaps

Define $\Phi(DS) = (k\cdot \#$ of roots in $DS + 2 \cdot \#$ marked bits in $DS)$. Note that *insert* and *delete-min* do not ever cause nodes to be marked - we can analyze their behaviour without reference to marked and unmarked bits. The parameter k is a constant that we will conveniently specify later. We now analyze the costs of the operations in terms of their amortized costs (defined to be the real costs plus the changes in the potential function).

- *insert*: the amortized cost is O(1). O(1) actual work plus k * O(1) change in potential for adding a new root. O(1) + kO(1) = O(1) total amortized cost.
- *delete-min*: for every node that we put into the root list (the children of the node we have deleted), plus every node that is already in the root list, we do constant work putting that node into a bucket corresponding to its rank and constant work whenever we merge the node. Our real costs are putting the roots into buckets (O(#*roots*)), walking through the buckets (O($\log n$)), and doing the consolidating tree merges (O(#*roots*)). On the other hand, our change in potential is $k*(\log n - \#roots)$ (since there are at most $\log n$ roots after consolidation). Thus, total amortized cost is O(#*roots*) + O($\log n$) + $k * (\log n - \#roots)$ = O($\log n$).
- *decrease-key*: The real cost is O(1) for the cut, key decrease and re-insertion. This also increases the potential function by O(1) since we are adding a root to the root list, and maybe by another 2 since we may mark a node. The only problematic issue is the possibility of a "cascading cut" - a cascading cut is the name we give to a cut that causes the node above it to cut because it was already marked, which causes the ndoe above it be cut since it too was alrady marked, etc. This can increase the actual cost of the operation to (# of nodes already marked). Luckily, we can pay for this with the potential function! Every cost we incur from having to update pointers due to a marked node that was cut is offset by the decrease in the potential function when that previously marked node is now left unmarked in the root list. Thus the amortized cost for this operation is just O(1).

The only thing left to prove is that for every node in every tree in our Fibonacci heap, the number of descendants of that node is exponential in the number of children of that node, and that this is true even in the presence of the "weird" cut rule for marked bits. We must prove this in order to substantiate our earlier assertion that all nodes have degree $\leq \log n$.

1.2.6 The trees are big

Consider the children of some node x in the order in which they were added to x.

Lemma : The i^{th} child to be added to x has rank at least $i - 2$.

Proof : Let y be the i^{th} child to be added to x. When it was added, y had at least $i - 1$ children. This is true because we can currently see $i - 1$ children that were added earlier, so they were there at the time of the y's addition. This means that y had at least $i - 1$ children at the time of it's merger, because we only merge equal ranked nodes. Since a node could not lose more than one child without being cut itself, it must be that y has at least $i - 2$ children ($i - 1$ from when it was added, and no more than a potential 1 subsequently lost). ■

Note that if we had been working with a binomial tree, the appropriate lemma would have been $rank = i - 1$ not $\geq i - 2$.

Let S_k be the minimum number of descendants of a node with k children. We have $S_0 = 1, S_1 = 2$ and,

$$S_k \geq \sum_{i=0}^{k-2} S_i$$

This recurrence is solved by $S_k \geq F_{k+2}$, the $(k+2)^{th}$ Fibonacci number. Ask anyone on the street and that person will tell you that the Fibonacci numbers grow exponentially; we have proved $S_k \geq 1.5^k$, completing our analysis of Fibonacci heaps.

1.2.7 Utility

Only recently have problem sizes increased to the point where Fibonacci heaps are beginning to appear in practice. Further study of this issue might make an interesting term project; see David Karger if you're curious.

Fibonacci Heaps allow us to improve the running time in Prim's and Djikstra's algorithms. A more thorough analysis of this will be presented in the next class.

6.854 Advanced Algorithms

Lecture 2: September 9, 2005 Lecturer: David Karger

Scribes: Sommer Gentry, Eddie Kohler

Persistent Data Structures

2.1 Introduction and motivation

So far, we've seen only ephemeral data structures. Once changes have been made to an ephemeral data structure, no mechanism exists to revert to previous states. Persistent data structures are really data structures with archaeology.

Partial persistence lets you make modifications only to the present data structure but allows queries of any previous version. These previous versions might be accessed via a timestamp.

Full persistence lets you make queries and modifications to all previous versions of the data structure. With this type of persistence the versions don't form a simple linear path — they form a version tree.

The obvious way to provide persistence is to make a copy of the data structure each time it is changed.

This has the drawback of requiring space and time proportional to the space occupied by the original data structure.

In turns out that we can achieve persistence with O(1) additional space and O(1) slowdown per operation for a broad class of data structures.

2.1.1 Applications

In addition to the obvious 'look-back'applications, we can use persistent data structures to solve problems by representing one of their dimensions as time.

Once example is the computational geometry problem of planar point location. Given a plane with various polygons or lines which break the area up into a number of regions, in which region is a query point is located?

In one dimension, the linear point location problem can be solved with a splay tree or a binary tree that simply searches for the two objects on either side of the query point.

To solve the problem in two dimensions, break the plane into vertical slices at each vertex or point where lines cross. These slices are interesting because crossovers don't happen inside slices: inside each slice, the dividing lines between regions appear in a fixed order, so the problem reduces to the

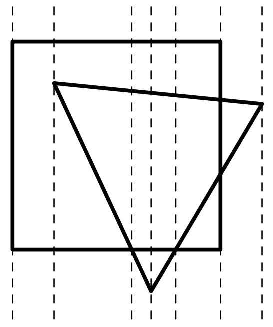

Figure 2.1: Breaking the plane into slices for planar point location

linear case and requires a binary search (plus a bit of linear algebra). Figure 2.1 shows an example of how these slices look. To locate a point, first find the vertical slice it is in with a search on the point's x coordinate, and then, within that slice, find the region it is in with a search on the point's y coordinate (plus algebra). To do two binary searches takes only $O(\log n)$ time, so we can locate a point in $O(\log n)$ time. However, setting up the trees for searching a figure with n vertices will require n different trees, taking $O(n^2 \log n)$ time and $O(n^2)$ space to do the preprocessing.

Notice that between two adjacent slices of the picture there will only be one change. If we treat the horizontal direction as a timeline and use a persistent data structure, we can find the horizontal location of the point as a 'version' of the vertical point location data structure. In this way, we can preserve the $O(\log n)$ query time and use only $O(n)$ space and $O(n \log n)$ preprocessing time.

2.2 Making pointer-based data structures persistent

Now let's talk about how to make arbitrary pointer-based data structures persistent. Eventually, we'll reveal a general way to do this with $O(1)$ additional space and $O(1)$ slowdown, first published by Sleator and Tarjan et al. We're mainly going to discuss partial persistence to make the explanation simpler, but their paper achieves full persistence as well.

2.2.1 First try: fat nodes

One natural way to make a data structure persistent is to add a modification history to every node. Thus, each node knows what its value was at any previous point in time. (For a fully persistent structure, each node would hold a version tree, not just a version history.)

This simple technique requires $O(1)$ space for every modification: we just need to store the new data. Likewise, each modification takes $O(1)$ additional time to store the modification at the end of the modification history. (This is an amortized time bound, assuming we store the modification history in a growable array. A fully persistent data structure would add $O(\log m)$ time to every modification, since the version history would have to be kept in a tree of some kind.)

Unfortunately, accesses have bad time behavior. We must find the right version at each node as we traverse the structure, and this takes time. If we've made m modifications, then each access operation has $O(\log m)$ slowdown. (In a partially persistent structure, a version is uniquely identified by a timestamp. Since we've arranged the modifications by increasing time, you can find the right version by binary search on the modification history, using the timestamp as key. This takes $O(\log m)$ time to find the last modification before an arbitrary timestamp. The time bound is the same for a fully persistent structure, but a tree lookup is required instead of a binary search.)

2.2.2 Second try: path copying

Another simple idea is to make a copy of any node before changing it. Then you have to cascade the change back through the data structure: all nodes that pointed to the old node must be modified to point to the new node instead. These modifications cause more cascading changes, and so on, until you reach a node that nobody else points to—namely, the root. (The cascading changes will always reach the root.) Maintain an array of roots indexed by timestamp; the data structure pointed to by time t's root is exactly time t's data structure. (Some care is required if the structure can contain cycles, but it doesn't change any time bounds.)

Figure 2.2 shows an example of path copying on a binary search tree. Making a modification creates a new root, but we keep the old root around for later use; it's shown in dark grey. Note that the old and new trees share some structure (light grey nodes).

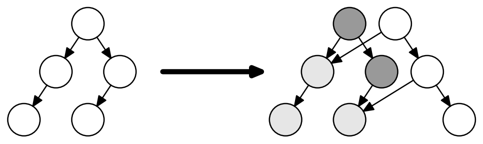

Figure 2.2: Path copying on binary search trees

Access time does better on this data structure. Accesses are free, except that you must find the correct root. With m modifications, this costs $O(\log m)$ additive lookup time—much better than fat nodes' multiplicative $O(\log m)$ slowdown.

Unfortunately, modification time and space is much worse. In fact, it's bounded by the size of the structure, since a single modification may cause the entire structure to be copied. That's $O(n)$.

Path copying applies just as well to fully persistent data structures.

2.2.3 Sleator, Tarjan et al.

Sleator, Tarjan et al. came up with a way to combine the advantages of fat nodes and path copying, getting $O(1)$ access slowdown and $O(1)$ modification space and time. Here's how they did it, in the special case of trees.

In each node, we store one *modification box.* This box can hold one modification to the node—either a modification to one of the pointers, or to the node's key, or to some other piece of node-specific data—and a timestamp for when that modification was applied. Initially, every node's modification box is empty.

Whenever we access a node, we check the modification box, and compare its timestamp against the access time. (The access time specifies the version of the data structure that we care about.) If the modification box is empty, or the access time is *before* the modification time, then we ignore the modification box and just deal with the normal part of the node. On the other hand, if the access time is *after* the modification time, then we use the value in the modification box, overriding that value in the node. (Say the modification box has a new `left` pointer. Then we'll use it instead of the normal `left` pointer, but we'll still use the normal `right` pointer.)

Modifying a node works like this. (We assume that each modification touches one pointer or similar field.) If the node's modification box is empty, then we fill it with the modification. Otherwise, the modification box is full. We make a copy of the node, but *using only the latest values.* (That is, we overwrite one of the node's fields with the value that was stored in the modification box.) Then we perform the modification directly on the new node, without using the modification box. (We overwrite one of the new node's fields, and its modification box stays empty.) Finally, we cascade this change to the node's parent, just like path copying. (This may involve filling the parent's modification box, or making a copy of the parent recursively. If the node has no parent—it's the root—we add the new root to a sorted array of roots.)

Figure 2.3 shows how this works on a persistent search tree. The modification boxes are shown in grey.

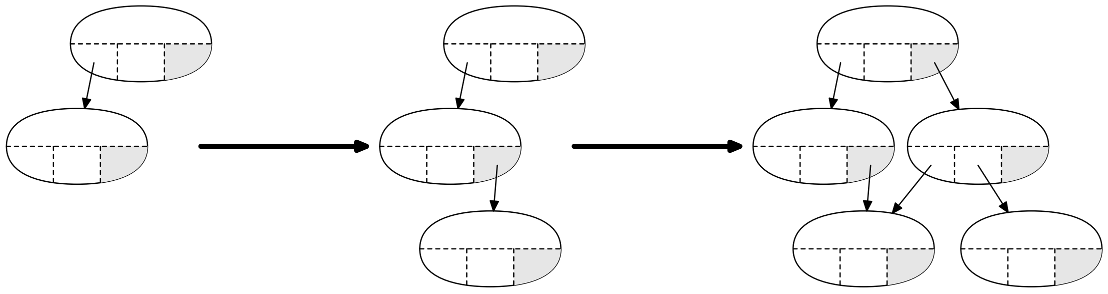

Figure 2.3: Modifying a persistent search tree.

With this algorithm, given any time t, at most one modification box exists in the data structure with time t. Thus, a modification at time t splits the tree into three parts: one part contains the data from before time t, one part contains the data from after time t, and one part was unaffected by the modification.

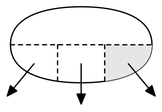

Figure 2.4: How modifications split the tree on time.

How about time bounds? Well, access time gets an $O(1)$ slowdown (plus an additive $O(\log m)$ cost for finding the correct root), just as we'd hoped! (We must check the modification box on each node we access, but that's it.)

Time and space for modifications require amortized analysis. A modification takes $O(1)$ amortized space, and $O(1)$ amortized time. To see why, use a potential function φ, where $\varphi(T)$ is *the number of full live nodes in* T. The live nodes of T are just the nodes that are reachable from the current root at the current time (that is, after the last modification). The full live nodes are the live nodes whose modification boxes are full.

So, how much does a modification cost? Each modification involves some number of copies, say k, followed by 1 change to a modification box. (Well, not quite—you could add a new root—but that doesn't change the argument.) Consider each of the k copies. Each costs $O(1)$ space and time, but decreases the potential function by one! (Why? First, the node we copy must be full and live, so it contributes to the potential function. The potential function will only drop, however, if the old node isn't reachable in the new tree. But we know it isn't reachable in the new tree—the next step in the algorithm will be to modify the node's parent to point at the copy! Finally, we know the copy's modification box is empty. Thus, we've replaced a full live node with an empty live node, and φ goes down by one.) The final step fills a modification box, which costs $O(1)$ time and increases φ by one.

Putting it all together, the change in φ is $\Delta\varphi = 1 - k$. Thus, we've paid $O(k + \Delta\varphi) = O(1)$ space and $O(k + \Delta\varphi + 1) = O(1)$ time!

What about non-tree data structures? Well, they may require more than one modification box. The limiting factor is the *in-degree* of a node: how many other nodes can point at it. If the in-degree of a node is k, then we must use k extra modification boxes to get $O(1)$ space and time cost.

2.2.4 The geometric search problem

Let's return to the geometric search problem discussed in Section 2.1. We now know how to make a persistent tree; but what kind of balanced tree should we use?

It turns out that this is one application where splay trees crash and burn. The reason is splaying. Every rotation while we access a splay tree is a modification, so we do $O(\log n)$ modifications (costing an additional $O(\log n)$ space) per access—including reads!

A less sexy balanced tree, like a red-black tree, is a better choice. Red-black trees keep themselves balanced with at most one rotation per modification (and a bunch of fiddling with red/black bits). This looks good—accesses are cheaper, and modifications cost $O(1)$—almost. The "almost" is because of red/black bit fiddling, which may affect a lot more than one node on the tree. A fully persistent red-black tree would need to keep the proper values for the red/black bits for every single version of the tree (so that further modifications could be made). This would mean that changing a red/black bit would count as a modification, and would have a persistence-related cost. Luckily, in the geometric search problem, we won't need to look at the red/black bits for old versions of the tree, so we can keep them only for the latest version of the tree and pay $O(1)$ persistence-related cost per modification.

6.854 Advanced Algorithms		
Lecture 3: 09/12/2005		Lecturer: David Karger
	Scribes: Xin Zhang	

Splay Trees

3.1 Introduction

Splay trees are binary search trees with good balance properties when amortized over a sequence of operations.

When a node x is accessed, we perform a sequence of **splay steps** to move x to the root of the tree. There are 6 types of splay steps, each consisting of 1 or 2 rotations (see Figures 3.1, 3.2, and 3.3).

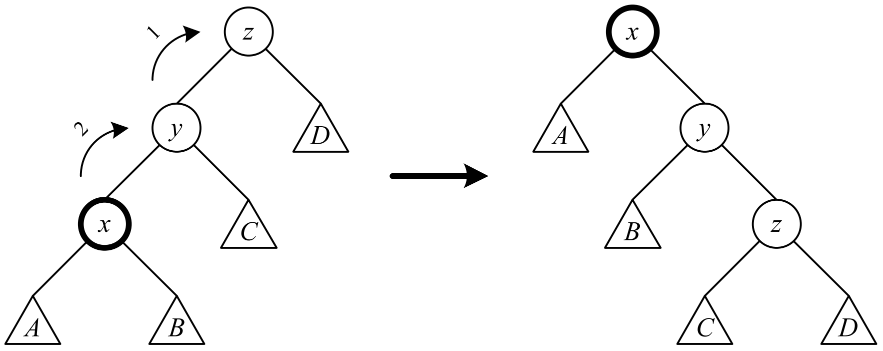

Figure 3.1: **The** rr **splay step:** This is performed when x and x's parent are both left children. The splay step consists of first a right rotation on z and then a right rotation on y (hence rr). The ll splay step, for x and x's parent being right children, is analogous.

We perform splay steps to x (rr, ll, lr, or rl, depending on whether x and x's parent are left or right children) until x is either the root or a child of the root. In the latter case, we need to perform either a r or l splay step to make x the root. This completes a **splay** of x.

We will show that splay operations have amortized cost $O(\log n)$, and that consequently all splay tree operations have amortized cost $O(\log n)$.

3.2 Analysis of Splay Steps

For amortized analysis, we define the following for each node x:

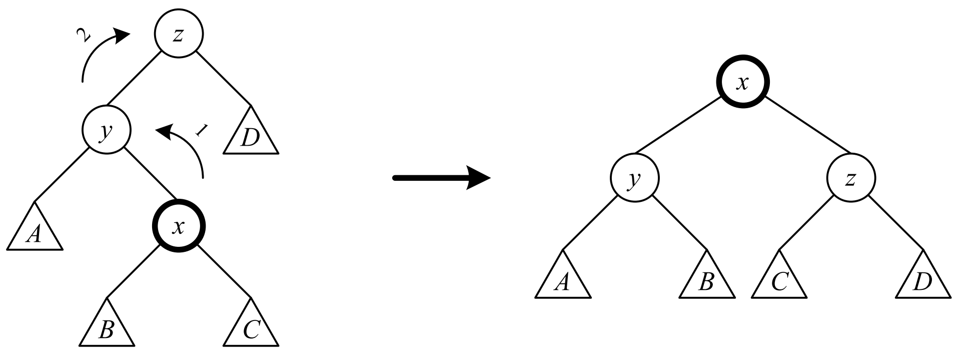

Figure 3.2: **The *lr* splay step:** This is performed when x is a right child and x's parent is a left child. The splay step consists of first a left rotation on y and then a right rotation on z. The *rl* splay step, for x being a left child and x's parent being a right child, is analogous.

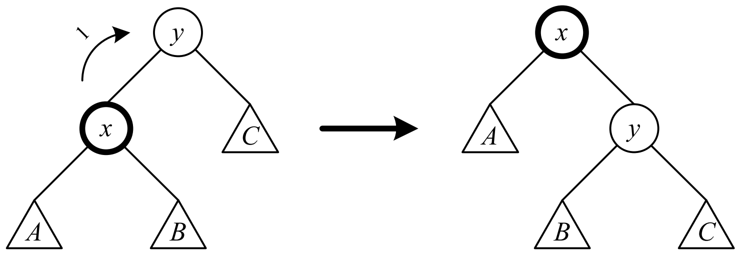

Figure 3.3: **The *r* splay step:** This is performed when x is the left child of the root y. The splay step consists of a right rotation on the root. The *l* splay step, for x being the right child of the root, is analogous.

- a constant *weight* $w(x) > 0$ (for the analysis, this can be arbitrary)
- *weight sum* $s(x) = \sum_{y \in \text{subtree}(x)} w(y)$ (where subtree(x) is the subtree rooted at x, including x)
- *rank* $r(x) = \log s(x)$

We use $r(x)$ as the potential of a node. The potential function after i operations is thus $\phi(i) = \sum_{x \in \text{tree}} r(x)$.

Lemma 1 *The amortized cost of a splay step on node x is $\leq 3(r'(x) - r(x)) + 1$, where r is rank before the splay step and r' is rank after the splay step. Furthermore, the amortized cost of the rr, ll, lr, and rl splay steps is $\leq 3(r'(x) - r(x))$.*

Proof:

We will consider only the rr splay step (refer to Figure 3.1). The actual cost of the splay step is 2 (for 2 rotations). The splay step only affects the potentials/ranks of nodes x, y, and z; we observe that $r'(x) = r(z)$, $r(y) \geq r(x)$, and $r'(y) \leq r'(x)$.

The amortized cost of the splay step is thus:

$$\begin{aligned}
\text{amortized cost} &= 2 + \phi(i+1) - \phi(i) \\
&= 2 + (r'(x) + r'(y) + r'(z)) - (r(x) + r(y) - r(z)) \\
&= 2 + (r'(x) - r(z)) + r'(y) + r'(z) - r(x) - r(y) \\
&\leq 2 + 0 + r'(x) + r'(z) - r(x) - r(x) \\
&= 2 + r'(x) + r'(z) - 2r(x)
\end{aligned}$$

The log function is concave, i.e., $\frac{\log a + \log b}{2} \leq \log\left(\frac{a+b}{2}\right)$. Thus we also have ($s$ is weight sum before the splay step and s' is weight sum after the splay step):

$$\begin{aligned}
\frac{\log(s(x)) + \log(s'(z))}{2} &\leq \log\left(\frac{s(x) + s'(z)}{2}\right) \\
\frac{r(x) + r'(z)}{2} &\leq \log\left(\frac{s(x) + s'(z)}{2}\right) \text{ (note that } s(x) + s'(z) \leq s'(x)) \\
&\leq \log\left(\frac{s'(x)}{2}\right) \\
&= r'(x) - 1 \\
r'(z) &\leq 2r'(x) - r(x) - 2
\end{aligned}$$

Thus the amortized cost of the rr splay step is $\leq 3(r'(x) - r(x))$.

The same inequality must hold for the ll splay step; the inequality also holds for the lr (and rl) splay steps. The $+1$ in the lemma applies for the r and l cases.

■

Corollary 1 *The amortized cost of a splay operation on node x is $O(\log n)$.*

Proof:

The amortized cost of a splay operation on x is the sum of the amortized costs of the splay steps on x involved:

$$\begin{aligned}
\text{amortized cost} &= \sum_i \text{cost}(\text{splay\_step}_i) \\
&\leq \sum_i \left(3(r^{i+1}(x) - r^i(x)\right) + 1 \\
&= 3(r(\text{root}) - r(x)) + 1
\end{aligned}$$

The $+1$ comes from the last r or l splay step (if necessary). If we set $w(x) = 1$ for all nodes in the tree, then $r(\text{root}) = \log n$ and we have:

$$\text{amortized cost} \leq 3\log n + 1 = O(\log n)$$

■

3.3 Analysis of Splay Tree Operations

3.3.1 Find

For the find operation, we perform a normal BST find followed by a splay operation on the node found (or the leaf node last encountered, if the key was not found). We can charge the cost of going down the tree to the splay operation. Thus the amortized cost of find is $O(\log n)$.

3.3.2 Insert

For the insert operation, we perform a normal BST insert followed by a splay operation on the node inserted. Assume node x is inserted at depth k. Denote the parent of x as y_1, y_1's parent as y_2, and so on (the root of the tree is y_k). Then the change in potential due to the insertion of x is (r is rank before the insertion and r' is rank after the insertion, s is weight sum before the insertion):

$$\begin{aligned}
\Delta\phi &= \sum_{j=1}^{k} (r'(y_j) - r(y_j)) \\
&= \sum_{j=1}^{k} (\log(s(y_j)+1) - \log(s(y_j))) \\
&= \sum_{j=1}^{k} \log\left(\frac{s(y_j)+1}{s(y_j)}\right) \\
&= \log\left(\prod_{j=1}^{k} \frac{s(y_j)+1}{s(y_j)}\right) \text{ (note that } s(y_j)+1 \leq s(y_{j+1})) \\
&\leq \log\left(\frac{s(y_2)}{s(y_1)} \cdot \frac{s(y_3)}{s(y_2)} \cdots \frac{s(y_k)}{s(y_{k-1})} \cdot \frac{s(y_k)+1}{s(y_k)}\right) \\
&= \log\left(\frac{s(y_k)+1}{s(y_k)}\right) \\
&\leq \log n
\end{aligned}$$

The amortized cost of the splay operation is also $O(\log n)$, and thus the amortized cost of insert is $O(\log n)$.

We have proved the following:

Theorem 1 *All splay tree operations have amortized cost $O(\log n)$.*

6.854 Advanced Algorithms

Lecture 4: September 14, 2005 Lecturer: David Karger

Suffix Trees and Fibonacci Heaps

4.1 Suffix Trees

Recall that our goal is to find a pattern of length m in a text of length n. Also recall that the trie will contain a size $|\Sigma|$ array at each node to give $O(m)$ lookups.

4.1.1 Size of the Trie

Previously, we have seen a construction algorithm that was linear in the size of the trie. We would like to show that the size of the trie is linear in the size of the text, so that the construction algorithm takes $O(n)$ time. We can achieve this size goal by using a compressed suffix tree. In a compressed tree, each node has strictly more than one child. Below, we see the conversion from a uncompressed suffix tree to a compressed suffix tree.

```
o
 \ b
  o                        o
   \ b              =>      \  bbb
    o                        o
     \ b
      o
```

How will this change the number of nodes in the trie? Since there are no nodes with only one child, this is a full binary tree (i.e. every internal node has degree ≥ 2).

Lemma 1 *In any full tree, the number of nodes is not more than twice the number of leaves.*

When we use the trailing \$, the number of leaves in the trie is the number of suffixes. So does this mean that there are n leaves and the tree is of size $O(n)$? Yes; however, the number of nodes isn't necessarily the full size of the tree – we must store the substrings as well. For a string with distinct characters, storing strings on the edges could lead to a $O(n^2)$ size algorithm. Instead, we just store the starting and ending index in the original text on each edge, meaning that the storage at each node is $O(1)$, so the total size for the tree is in fact $O(n)$.

With the compressed tree, we can still perform lookups in $O(m)$ time using the *slowfind* algorithm which compares one character at a time from the pattern to the text in the trie. When *slowfind* encounters a compressed node, it checks all of the characters in the node, just as if it were traversing the uncompressed series of nodes.

4.1.2 Building the Trie

A simple approach to building the compressed tree would be to build an uncompressed tree and then compress it. However, this approach would require quadratic time and space.

The construction algorithm for compressed tries will still insert $S_1 ... S_n$ in order. As we go down the trie to insert a new suffix, we may need to leave in the middle of an edge. For example, consider the trie that contains just `bbb`. To insert `ba`, we must split the edge :

```
o                        o
 \                        \ b
  \ bbb                    o
   \                    a / \ bb
    o                    o   o
```

Splitting an edge is easy; we will create one new node from the split and then one new node (a leaf) from the insertion.

One problem with compressed trie is where to put suffix links from the compressed edges. Another problem is that we previously described the time to operate on the tree in terms of n (the number of characters in the text); however, n may now be greater than the number of nodes.

fastfind is an algorithm for descending the trie if you know that the pattern is in the trie. *fastfind* only checks the first character of a compressed edge; all the other characters must match if the first does because the pattern is in the trie and because there is no branch in the edge (ie, if the pattern is there and there is no branch, it must match the entire edge or stop in the middle of the edge). If the pattern is shorter than the edge, then *fastfind* will stop in the middle of the edge. Consequently, the number of operations in a *fastfind* is linear in the number of checked nodes in the trie rather than the length of the pattern.

Suppose we have just inserted $S_i = aw$ and are at the newly created leaf which has parent node p_i. We maintain the following invariant:

Invariant: Every internal node except for the current parent has a suffix link (ignore for now the issue of where the suffix links point).

```
          /
        /     SL
  g_i  o ==============o
      / \              |  w1
 w1  /   ...           |
    o p_i              o alpha
    |
    |
 w2 |
    o s_i
```

Now we describe the consruction in detail. Let g_i be a parent of p_i. To insert S_{i+1}: ascend to g_i, traverse the suffix link there, and do a *fastfind* of w_1, which takes you to node α (thus maintaining the invariant for next time). Make a suffix link from p_i to α. From there, do a *slowfind* on w_2 and do the insertions that you need. Since p_i was previously a leaf node, it has no suffix link yet. g_i was previously an internal node, so it has a suffix link. w_1 is the part of S_i that was already in the trie below g_i (i.e., it was p_i), which is why we can use *fastfind* on it. w_2 is the part of S_i that was not previously in the trie.

The running time analysis will be in two parts. The first part is the cost from the suffix of g_i to the suffix of p_i. The second is the cost from the suffix of p_i to the bottom of the search. The cost of going up is constant, since there are only two steps thanks to the compressed edges.

Looking at the second cost (the *slowfind* part), we see that it is the number of charactersr in the length difference between the suffix of p_i and p_{i+1}, which is $|p_{i+1}| - |p_i| + 1$. The sum of this term over all i is $|p_n| - |p_0| + n = O(n)$.

For the first cost, recall that *fastfind*'s runtime will be upperbounded by the runtime of *slowfind*. It takes at most $|g_{i+1}| - |g_i|$ time to reach g_{i+1}. If g_{i+1} is below the suffix of p_i, then there is no cost. If the suffix of p_i is below g_{i+1}, then the suffix of p_i is p_{i+1} and the *fastfind* only takes one step from g_{i+1} to p_{i+1}, so the cost is $O(1)$.

The progression of the insert is

- suffix of g_i
- g_{i+1}
- suffix of p_i
- p_{i+1}

The total time is linear in the size of the compressed tree, which is linear in the size of the input.

4.2 Heaps

Prim and Dijkstra's algorithms for shortest paths and minimum spanning trees were covered in 6.046. Both are greedy algorithms that start by setting node distances to infinity and then relaxing the distances while choosing the shortest. To perform these operations, we use a priority queue (generally implemented as a heap). A heap is a data structure that will support insert, decrease-key, and delete-min operations (and perhaps others).

With a standard binary heap, all operations run in $O(\log n)$ time, so both algorithms take $O(m \log n)$ time. We'd like to improve the performance of the heap to get a better running time for these algorithms. We could show that $O(\log n)$ is a lower bound on the time for delete-min, so the improvement will have to come from somewhere else. The Fibonacci Heap performs a decrease-key operation in $O(1)$ time such that Prim and Dijkstra's algorithms require only $O(m + n \log n)$ time,

Idea: During insertions, perform the minimal work possible. Rather than performing the whole insert, we'll just stick the node onto the end of some list, taking $O(1)$ time. This would require us to do $O(n)$ work to perform delete-min. However, we can put that linear amount of work to good use to make the next delete-min more efficient.

The Fibonacci heap uses "Heap Ordered Trees," meaning that the children of every node have a key greater than their parent and that the minimum element is at the root. For Fibonacci heaps, we will have only 1 child pointer, a doubly linked list of children, and parent pointers at every node.

The time to merge two HOTs is constant: compare the two root keys and attach the HOT with the larger root as a child of the smaller root.

To insert into a HOT, compare the new element x and the root. If x is smaller, it becomes the new root and the old root is its child. If x is larger, it is added to the list of children.

To decrease a key, you prune the node from the list of children and then perform a merge.

The expensive operation is delete-min. Finding the minimum node is easy; it is the root. However, when we remove the root, we might have a large number of children that need to be processed. Therefore, we wish to keep the number of children of any node in the tree relatively small. We will see how to do this next lecture

6.854 Advanced Algorithms

Lecture 3: September 15, 2003 — Lecturer: David Karger

Scribes: abhi shelat (1999), Andrew Menard (1999), Akshay Patil (2003)

Van Emde Boas Queues

We would like to design a queue that only handles integers from $0...C$, but supports all operations in $O(\log\log C)$ time. This idea is first presented in "Design and Implementation of an efficient priority queue", Mathematical Systems Theory 10 (1977). Mikkel Thorup provides some extensions in "On RAM priority queues" in SODA 1996.

This is a fantastic example of an efficient recursive data structure. This structure also exploits the fact that keys in priority queues are most often integers and that computers have efficient operations for manipulating the binary representations of integers (shifts, masks, logical operations).

Each vEB queue maintains the following information:

Field	Notation
The current minimum	$Q.min$
vEB queue on high halfword	$Q.summ$
An array where each entry is a vEB queue on the low halfwords	$Q[x_h]$

Intuitively, we have the minimum item stored so we can return it quickly. VEB priority queues take the place of the buckets in the multi-level bucket structure that was presented earlier (see Scribe Notes 2).

There are $\sqrt{U}$ elements in the array, and each element in the array corresponds to a distinct high halfword x_h. (high halfword, x_h = the higher-order $\frac{w}{2}$ bits in the w-bit representation of a number, x; $w = \lg C$. low halfword, x_l = the rest). Thus when looking for an element, we can look for the upper high halfword of the number, then recurse on the low halfword

The purpose of the *summ* is to quickly determine which low halfword vEB queues actually contain elements. If a low halfword vEB is non-empty, then we store the vEB's address (x_h) in $Q.summ$. Thus $Q.summ$ is a vEB of size $\sqrt{U}$ just like the low halfword vEBs.

This is a rather straightforward approach and requires $O(U)$ space. (a more complex, randomized approach is possible, leading to $O(\sqrt{U})$ space.)

Inserts

Consider the algorithm for inserting an element into the queue.

First we update all Q.min information. This takes lines 1-5

Line 7 of the algorithm splits x into halfwords. This operation can be done with a bit shift operation and a bitwise-AND operation.

Algorithm 1 INSERT(x, Q)

```
if $Q.min = null$ then
    $Q.min = x$, return
end if
if $Q.min > x$ then
    swap $x \leftrightarrow Q.min$
end if
$(x_h, x_l) \Leftarrow x$ {divide $x$ into its high halfword, $x_h$, and its low halfword, $x_l$}
if $Q[x_h].min = null$ then
    INSERT$(x_h, Q.summ)$
    $Q[x_h].min = x_l$
else
    INSERT$(x_l, Q[x_h])$
end if
```

At this point, we have split the problem of inserting x into this queue into a smaller problem of either inserting the high halfword of x, or the low halfword of x. If we have already inserted x_h in a previous operation, then it suffices to insert x_l in the low halfword queue that belongs to x_h. If we have not already seen x_h, then we need to insert it into our high halfword queue. However, this means that we can insert x_l in $O(1)$ time by simply changing $Q[x_h].min$.

The recursions finishes when the queues become small enough to store as arrays (eg, 1-bit queues).

Notice that the function makes only one recursive call to the INSERT function. Because all of the other operations are $O(1)$ time, the recurrence is

$$T(b) = T(b/2) + O(1)$$

where b is the number of bits in x. Hence, $T(b) = O(\log b)$ and since $b = \log C$, the algorithm runs in $O(\log\log C)$ time.

Find-Min

Trivial, return Q.min. This should take $O(1)$ time.

Delete

At somepoint, every element is a minimum element for some vEB. We handle this base case by deleting the min and then recalculating what the minimum element is. We must be mindful to keep $Q.summ$ up to date as well. Otherwise, we merely drill down the levels of vEB until we reach our base case situation.

Once again, the running time for this algorithm is $O(\log\log C)$. The reason why this again results in only one call to a $O(\sqrt{U})$ problem is that even if DELETE is called twice, the second call implies that the first call took only constant time. (if $Q(x_h)$ is now empty, that means that the earlier call to DELETE was deleting that vEBs minimum, and only, element. Such a call takes constant time, so we can now use our $O(\sqrt{U})$ time on updating $Q.summ$).

Other supported operations

Note that it is easy to support all of the following operations in $O(\log\log C)$ time

Algorithm 2 DELETE(x, Q)

```
1: if $x < Q.min$ then
2:   return/die
3: end if
4: if $x = Q.min$ then
5:   $first = Q.summ.min$
6:   if $first = null$ then
7:     $Q.min = null$, return
8:   end if
9:   $x = Q.min = first|Q[first].min$ {create element with high halfword = “first”, low = $Q[first].min$ }
10: end if
11: DELETE$(x_l, Q[x_h])$
12: if $Q[x_h].min = null$ then
13:   DELETE$(x_h, Q.summ)$
14: end if
```

- MAX(). Symmetric to the min.
- FIND(x). Determines whether the element exists in the queue.
- SUCC(x). Finds the smallest value in the queue that is larger than x. Returns nothing if x is the largest.
- PRED(x). Finds the largest value in the queue that is smaller than x. Returns nothing if x is the smallest.
- DELETE-MIN(x). Special case of DELETE.
- DELETE-MAX(). Removes the max element from the queue. Works in the same way that DELETE-MIN works.

6.854 Advanced Algorithms

Lecture 6: 9/24/2003 Lecturer: Erik Demaine, David Karger

Scribes: Alexandr Andoni

Maximum Flows

6.1 The Maximum Flow Problem

In this section we define a flow network and setup the problem we are trying to solve in this lecture: the maximum flow problem.

Definition 1 *A network is a directed graph $G = (V, E)$ with a source vertex $s \in V$ and a sink vertex $t \in V$. Each edge $e = (v, w)$ from v to w has a defined capacity, denoted by $u(e)$ or $u(v, w)$. It is useful to also define capacity for any pair of vertices (v, w), with $u(v, w) = 0$ for any pair $(v, w) \notin E$. Let $m = |E|$ and $n = |V|$ be the number of edges and vertices in the graph, respectively.*

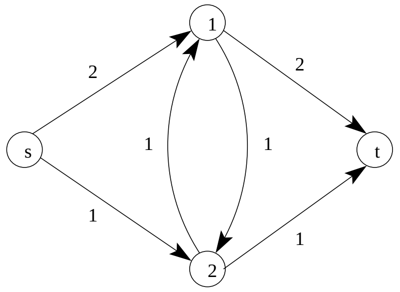

Figure 6.1: An example of a network with 4 vertices and 6 edges. The capacities of the edges are shown on the edges.

In a network flow problem, we assign a *flow* to each edge. There are two ways of defining a flow: raw (or gross) flow and net flow.

Definition 2 *Raw flow is a function $r(v, w) : V^2 \to \mathbb{R}$ that satisfies the following properties:*

- *Conservation:* $\underbrace{\sum_{w \in V} r(w, v)}_{\textit{incoming flow}} - \underbrace{\sum_{w \in V} r(v, w)}_{\textit{outgoing flow}} = 0$, *for all $v \in V \setminus \{s, t\}$.*
- *Capacity constraint:* $0 \leq r(v, w) \leq u(v, w)$.

For every vertex v except the source or sink, conservation requires that the total flow entering v must equal the total flow leaving v. The capacity constraint requires that the flow along any edge be positive and less than the capacity of that edge. We say that a flow f is *feasible* if satisfies these two conditions.

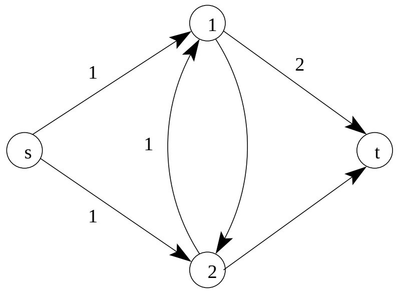

Figure 6.2: An example of a raw flow for the network above. The flow has a value of 2.

With a raw flow, we can have flows going both from v to w and flow going from w to v. In a net flow formulation however, we only keep track of the difference of these two flows.

Definition 3 *Net flow is a function $f(v, w) : V^2 \to \mathbb{R}$ that satisfies the following conditions:*

- *Skew symmetry: $f(v, w) = -f(w, v)$.*
- *Conservation: $\sum_{w \in V} f(v, w) = 0$, for all $v \in V \setminus \{s, t\}$.*
- *Capacity constraint: $f(v, w) \leq u(v, w)$ for all $v, w \in V$.*

A raw flow $r(v, w)$ can be converted into a net flow via the formula $f(v, w) = r(v, w) - r(w, v)$. For example, if we have 7 units of flow from v to w and 4 units of flow from w to v, then the net flow from v to w is $f(v, w) = 3$. Skew symmetry follows directly from this formula relating raw flows and net flows. Because we can convert from raw flows to net flows, for the rest of the lecture we consider only net flow problems.

Although skew symmetry relates $f(v, w)$ and $f(w, v)$, it is important to note that capacity is still directional for a net flow problem. The capacity in one direction $u(v, w)$ is independent of the capacity in the reverse direction, $u(w, v)$.

To simplify notation later in the lecture, we denote $\sum_{w \in S} f(v, w)$ by $f(v, S)$ or $-f(S, v)$.

Definition 4 *The value of a flow f is defined as $|f| = \sum_{v \in V} f(s, v)$.*

The value of a flow is the sum of the flow on all edges leaving the source s. We later show that this is equivalent to the sum of all the flow going into the sink t. The value of a flow represents how much we can transport from the source to the sink. Our goal in this lecture is to solve the maximum flow problem.

Definition 5 *Maximum flow problem: Given a network $G = (V, E)$, find a feasible flow f with maximum value.*

6.2 Flow Decomposition and Cuts

In this section, we show that any feasible flow can be decomposed into paths from the source to the sink and cycles. We use this fact to derive an upper bound on the maximum flow value in terms of cuts of the network.

Lemma 1 *(Flow decomposition). We can decompose any feasible flow f on a network G into at most m cycles and s-t paths.*

Proof: The following algorithm extracts the m paths and cycles.

1. Find a path with positive flow from the node s to node t. (If the flow is non-zero, there exists at least one such path.)
2. Anti-augment the flow on this path—that is, reduce the flow in the path until the flow on some edge becomes 0.
3. Add this path as an element of the flow decomposition.
4. Continue these operations until there are no more paths from s to t with positive flow.
5. If there are still some edges with non-zero flow, the remaining flow can be decomposed into cycles. Find a cycle in the following way: take any edge with non-zero flow and follow an outgoing edge with non-zero flow until a cycle is found.
6. Anti-augment on the cycle found.
7. Add the cycle as an element of the flow decomposition.
8. Continue finding cycles until there are no more edges with non-zero flow.

Each time we anti-augment a path or a cycle, we zero out the flow on some edge. There are at most m anti-augmentations, and, consequently, m paths/cycles in the flow decomposition. ■

In a network flow problem, it is useful to work with a *cut* of the graph, particularly an *s-t cut*.

Definition 6 *A cut of network G is a partition of the vertices V into 2 groups: S and $\bar{S} = V \setminus S$.*

Definition 7 *An s-t cut is a cut such that $s \in S$ and $t \in \bar{S}$.*

We will usually represent a cut as the pair $(S, \bar{S})$, or just S. We generalize the concept of the net flow and the capacity of an edge to define the net flow and capacity of a cut.

Definition 8 *The net flow along cut $(S, \bar{S})$ is defined as $f(S) = \sum_{v \in S} \sum_{w \in \bar{S}} f(v, w)$.*

Definition 9 *The value (or capacity) of a cut is defined as $u(S) = \sum_{v \in S} \sum_{w \in \bar{S}} u(v, w)$.*

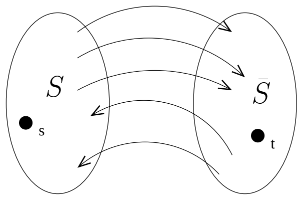

Figure 6.3: An illustration of the s-t cut. $s \in S$ and $t \in \bar{S}$. There might be both edges from S to $\bar{S}$ and from $\bar{S}$ to S.

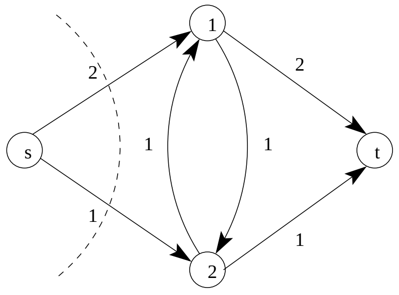

Figure 6.4: An example of a cut in a network. The s-t cut is represented by a dashed line. The value (capacity) of the cut is equal to 3. This is one of the minimum s-t cuts.

In summary, the flow (or capacity) of a cut is the sum of all flows (capacities) of edges that go from S to $\bar{S}$. Note that direction is important in these definitions. Flow or capacity along an edge in the reverse direction, from $w \in \bar{S}$ to $v \in S$, does not count.

Working with cuts is useful because of the following lemma:

Lemma 2 *Given a flow f, for any cut S, $f(S) = |f|$. In other words, all s-t cuts carry the same flow: the value of the flow f.*

Proof: We can use Lemma 1 to prove this statement directly. We decompose the flow into s-t paths and cycles. Each s-t path must end up in $\bar{S}$, so it must go from set S to $\bar{S}$ one more time than it goes from $\bar{S}$ to S. Therefore, an s-t path carring x flow along that path contributes exactly x to the value of the cut. A cycle must go from S to $\bar{S}$ the same number of times as it goes from $\bar{S}$ to S, contributing 0 to the value of the cut. Therefore the total value of the cut S is equal to the sum of the flows along every s-t path, which is equal to $|f|$.

Alternatively, we can prove the lemma by induction on the size of the sets S. For $S = s$, the claim is true. Now, suppose we move one vertex v from $\bar{S}$ to S. The value $f(S)$ changes in the following way:

- $f(S)$ increases by $f(v, \bar{S})$.
- $f(S)$ decreases by $f(S, v) = -f(v, S)$.

In conclusion, the total change in the value of $f(S)$ after moving the vertex v from S to $\bar{S}$ is equal to $f(v, \bar{S}) + f(v, S) = f(v, V) = 0$ (by conservation of flow). ■

For a flow network, we define a *minimum cut* to be a cut of the graph with minimum capacity. Then, Lemma 3 gives us an upper bound on the value of any flow.

Lemma 3 *If f is a feasible flow, then $|f| \leq u(S)$ for any cut S.*

Proof: For all edges e, $f(e) \leq u(e)$, so $f(S) \leq u(S)$ (the flow across any cut S is not more than the capacity of the cut). By Lemma 2, $|f| = f(S)$, so $|f| \leq u(S)$ for any cut S. ■

If we pick S to be a minimum cut, then we get an upper bound on the maximum flow value.

6.3 Max-Flow Min-Cut Theorem

In this section, we show that the upper bound on the maximum flow given by Lemma 3 is exact. This is the max-flow min-cut theorem.

To prove the theorem, we introduce the concepts of a residual network and an augmenting path.

Definition 10 *Let f be a feasible flow on a network G. The corresponding residual network, denoted G_f, is a network that has the same vertices as the network G, but has edges with capacities $u_f(v, w) = u(v, w) - f(v, w)$. Only edges with non-zero capacity, $u_f(v, w) > 0$, are included in G_f.*

Note that the feasibility conditions imply that $u_f(v, w) \geq 0$ and $u_f(v, w) \leq u(v, w) + u(w, v)$. This means all capacities in the residual network will be non-negative.

Definition 11 *An augmenting path is a directed path from the node s to node t in the residual network G_f.*

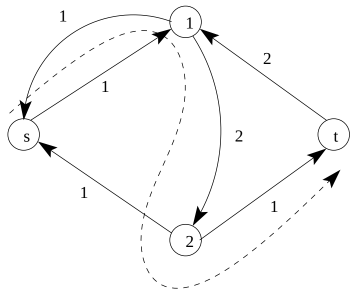

Figure 6.5: An example of a residual network. This residual network corresponds to the network depicted in Figure 6.1 and the flow in Figure 6.2. The dashed line corresponds to a possible augmenting path.

Note that if we have an augmenting path in G_f, then this means we can push more flow along such a path in the original network G. To be more precise, if we have an augmenting path $(s, v_1, v_2, \ldots v_k, t)$, the maximum flow we can push along that path is $\min\{u_f(s, v_1), u_f(v_1, v_2), u_f(v_2, v_3), \ldots u_f(v_{k-1}, v_k), u_f(v_k, t)\}$. Therefore, for a given network G and flow f, if there exists an augmenting path in G_f, then the flow f is not a maximum flow.

More generally, if f' is a feasible flow in G_f, then $f + f'$ is a feasible flow in G. The flow $f + f'$ still satisfies conservation because flow conservation is linear. The flow $f + f'$ is feasible because we can rearrange the inequality $f'(e) \leq u_f(e) = u(e) - f(e)$ to get $f'(e) + f(e) \leq u(e)$. Conversely, if f' is a feasible flow in G, then the flow $f - f'$ is a feasible in G_f.

Using residual networks and augmenting paths, we can state and prove the max-flow min-cut theorem.

Theorem 1 *(Max-flow min-cut theorem). In a flow network G, the following conditions are equivalent:*

1. *A flow f is a maximum flow.*
2. *The residual network G_f has no augmenting paths.*
3. *$|f| = u(S)$ for some cut S.*

These conditions imply that the value of the maximum flow is equal to the value of the minimum s-t cut: $\max_f |f| = \min_S u(S)$, *where f is a flow and S is as-t cut.*

Proof: We show that each condition implies the other two.

- $1 \Rightarrow 2$: If there is an augmenting path in G_f, then we previously argued that we can push additional flow along that path, so f was not a maximum flow. $1 \Rightarrow 2$ is the contrapositive of this statement.
- $2 \Rightarrow 3$:

  If the residual network G_f has no augmenting paths, s and t must be disconnected. Let $S = \{$vertices reachable from s in $G_f\}$. Since t is not reachable, the set S describes a s-t cut.

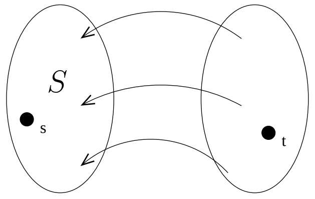

Figure 6.6: Network G_f is disconnected. The set S contains all the nodes that are reachable from s.

By construction, all edges (v, w) straddling the cut have residual capacity 0. This means in the original network G, these edges have $f(v, w) = u(v, w)$. Therefore, $|f| = f(S) = u(S)$.

- $3 \Rightarrow 1$: If for some cut S, $|f| = u(S)$, we know f must be a maximum flow. Otherwise, we would have a flow g with $|g| > u(S)$, contradicting Lemma 3.

From (1) and (3), we know that the maximum flow can not be less than the value of the minimum cut, because for some S, $|f| = u(S)$ and $u(S)$ is at least as big as the minimum cut value. Lemma 3 tells us that the maximum flow can not be greater than the minimum cut value. Therefore, the maximum flow value and the minimum cut value are the same. ■

6.4 Ford-Fulkerson Algorithm

The Ford-Fulkerson algorithm solves the problem of finding a maximum flow for a given network. The description of the algorithm is as follows:

1. Start with $f(v, w) = 0$.
2. Find an augmenting path from s to t (using, for example, a depth first search or similar algorithms).
3. Use the augmenting path found in the previous step to increase the flow.
4. Repeat until there are no more augmenting paths.

If the capacities are all integers, then the running time is $O(m|f|)$. This is true because finding an augmenting path and updating the flow takes $O(m)$ time, and every augmenting path we find must increase the flow by an integer that is at least 1.

In general, if we have integral capacities, then our solution satisfies an *integrality property*: there exists an integral maximal flow. This happens because every augmenting path increases flows by an integer amount.

Since the running time is directly proportional to the value of the maximal flow, this particular algorithm is only good for cases when the value $|f|$ is small. For example, when all capacities are at most 1, the maximum flow $|f|$ is at most n. In general, the algorithm may be as bad as linear in unary representation of the input. Figure 6.7 illustrates a bad case for this form of the Ford-Fulkerson algorithm.

We describe such an algorithm as being *pseudo-polynomial*, because it is polynomial in terms of variables we care about (but not necessarily the input).

If the capacities are rational, then it can be shown that the algorithm will finish. It might, however, require more than $O(m|f|)$ time. If the capacities are real, the algorithm might never finish, or even converge to a non-optimal value.

If we setup better rules for selecting the augmentation paths however, we might get better results. Before showing some improvements to the Ford-Fulkerson algorithm, we will introduce some new notions on the running time of algorithms.

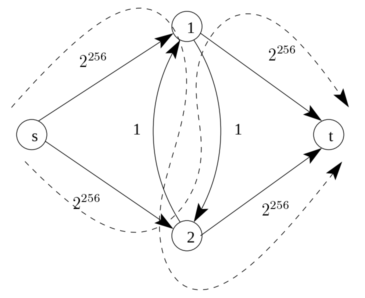

Figure 6.7: An example for which the Ford-Fulkerson, in the stated form, might perform very badly. The algorithm runs slowly if at each step, the augmentation path is either $s \to 1 \to 2 \to t$ or $s \to 2 \to 1 \to t$ (shown with dashed lines). At an augmentation, the flow will increase by at most 2.

Definition 12 *An algorithm is psuedo-polynomial if it is polynomial in the unary representation of the input.*

Definition 13 *An algorithm is weakly polynomial if it is polynomial in the binary representation of the input.*

Definition 14 *An algorithm is strongly polynomial if it is polynomial in combinatorial complexity of input. (For example, in the case of max-flow problem, the algorithm would have to be polynomial in n and m.)*

6.4.1 Improvements to the Ford-Fulkerson Algorithm

The are at least two possible ideas for improving the Ford-Fulkerson algorithm. Both of the improvements rely on a better choice of an augmenting path (rather than a random selection of an augmenting path).

1. Using breadth-first search, we can choose shortest-length augmenting path. With this path-selection rule, the number of augmentations is bounded by $n \cdot m$, and thus the running time of the algorithm goes down to $O(nm^2)$ time.
2. We can also choose the maximum-capacity augmenting path: the augmenting path among all augmenting paths that increases the flow the most (max-capacity augmenting path). It is possible to find such a path in $O(m \log n)$ time using a modified Dijkstra's algorithm (ignoring the cycles). The number of augmentations will be at most $m \ln |f| \leq m \ln(nU)$, where $U = \max\{u(v,w)\}$ (for integral capacities).

In this lecture we prove the time bound for the second improvement. Consider the maximum flow f in the current residual network. We apply the flow-decomposition lemma, Lemma 1 (discarding the cycles because they do not modify $|f|$). There are at most m paths carrying all the flow, so there must be at least one path carrying at least $|f|/m$ flow. Therefore, the augmenting path with maximum capacity increases the flow in the original network by at least $|f|/m$. This decreases the maximum possible flow in the residual graph from $|f|$ to $(1-1/m)|f|$ (remember, the smaller is the maximum possible flow in the residual graph, the greater is the corresponding flow in the original graph).

We need to decrease the flow $|f|$ by a factor of $(1-1/m)$ about $m \ln |f|$ times before we decrease the max flow in the residual graph to 1. This is because

$$|f| \left(1 - \frac{1}{m}\right)^{m \ln |f|} \approx |f| \left(\frac{1}{e}\right)^{\ln |f|} \approx 1.$$

In one more step, the residual graph will have a maximum flow of 0, meaning that the corresponding flow in the original graph is maximal. Thus, we need $O(m \ln |f|)$ augmentations. Since one augmentation step takes about $O(m \log n)$ time, the total running time is $O(m^2 \ln |f| \cdot \ln n)$. This algorithm is weakly polynomial, but not strongly polynomial.

6.4.2 Scaling Algorithm

We can also improve the running time of the Ford-Fulkerson algorithm by using a scaling algorithm. The idea is to reduce our max flow problem to the simple case, where all edge capacities are either 0 or 1.

The scaling idea, described by Gabow in 1985 and also by Dinic in 1973, is as follows:

1. Scale the problem down somehow by rounding off lower order bits.
2. Solve the rounded problem.
3. Scale the problem back up, add back the bits we rounded off, and fix any errors in our solution.

In the specific case of the maximum flow problem, the algorithm is:

1. Start with all capacities in the graph at 0.
2. Shift in the higher-order bit of each capacity. Each capacity is then either 0 or 1.
3. Solve this maximum flow problem.
4. Repeat this process until we have processed all remaining bits.

This description of the algorithm tells us how to scale down the problem. However, we also need to describe how to scale our algorithm back up and fix the errors.

To scale back up:

1. Start with some max flow for the scaled-down problem. Shift the bit of each capacity by 1, doubling all the capacities. If we then double all our flow values, we still have a maximum flow.

2. Increment some of the capacities. This restores the lower order bits that we truncated. Find augmenting paths in the residual network to re-maximize the flow.

We will need to find at most m augmenting paths. Before we scaled our problem back up, we had solved a maximum flow problem, so some cut in the residual network had 0 capacity. Doubling all the capacities and flows keeps this the same. When we increment the edges however, we increase the cut capacity by at most m: once for each edge. Each augmenting path we find increases the flow by at least 1, so we need at most m augmenting paths.

Each augmenting path takes at most $O(m)$ time to find, so we spend $O(m^2)$ time in each iteration of the scaling algorithm. If the capacity of any edge is at most U, which is an $O(\lg U)$ bit number, we require $O(\lg U)$ iterations of the scaling algorithm.

Therefore the total running time of the algorithm is $O(m^2 \lg U)$. This algorithm is also a weakly polynomial algorithm.

6.854 Advanced Algorithms

Lecture 1: 9/3/2003 Lecturer: Erik Demaine, David Karger

Scribes: Jiawen Chen

String Matching

1.1 Problem formulation

We consider a basic problem in string matching, and two different approaches to it. The first will be algorithmic, while the second will emphasize data structures. The basic problem we consider is: given a string T and a pattern P, determine if P appears as a substring in T, i.e.:

Input: A text T and a pattern P.
Output: Is P a substring of T? If so, where?

1.1.1 Motivation and Background

The string searching problem arises in many fields. An example is searching for common gene sequences in a long DNA sequence. A more familiar example are the type-ahead-find features of modern Internet browsers and text editors.

1.1.2 Obvious Algorithm

The obvious algorithm to solve this problem compare each P with T at each position, shifting over P at each iteration. The running time of this algorithm is $O(|T| * |P|)$. The algorithm needs to compare all $|P|$ characters of P at each iteration, and there are at worst $|T|$ iterations (actually, $|T| - |P|$).

1.1.3 Rabin-Karp String Matching

The Rabin-Karp string matching algorithm is a randomized algorithm that can solve the string matching problem in $O(|T| + |P|)$. The main technique used here is *fingerprinting*, which we will discuss in detail a bit later. The motivation is as follows: if we can find a way of compare the pattern P with a $|P|$ segment of T in $O(1)$ time, we can search the text in $O(|T|)$ time.

The idea of fingerprinting is as follows: comparing two strings x, y of length $|P|$ takes $|P|$ comparisons. Suppose we had a function f with the following property:

$$f(x) = f(y) \iff x = y$$

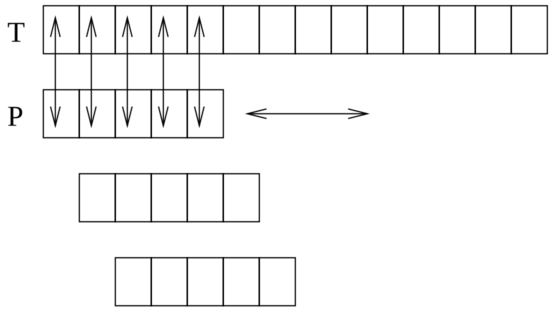

Figure 1.1: Naive String Matching

and checking $f(x) = f(y)$ could be done in constant time. This is not possible, so instead we settle for the following:

Definition 1 ***fingerprint of a string*** **($f(x)$):** *The fingerprint of a string x is a mapping from the string x to a set of integers such that if $x = y$ then $f(x) = f(y)$ and if $f(x) = f(y)$ then $x = y$* with high probability.

Assume (without loss of generality) that the input strings are over $\{0,1\}$. Treat a string x as the binary representation of a number. Define $f(x) =$ (number represented by x) mod p, where p is a random prime.

To do the comparisons, we need to compute $f(P)$ and $f(T[i : i+|P|-1])$, for every $1 \leq i \leq |T|-|P|$. Can we compute this in $O(|T| + |P|)$ time? Yes. The trick is to compute $f(T[i+1 : i+|P|])$ from $f(T[i : i+|P|-1])$ in constant time as follows: compute $T[i+1 : i+|P|-1]$ from $T[i : i+|P|-1]$ by "dropping" $T[i]$: subtract from the current fingerprint $2^{|P|-1} * T[i]$.
Then compute $T[i+1 : i+|P|]$ from $T[i+1 : i+|P|-1]$ by "adding the last letter and shifting": multiply the current fingerprint by 2 and add $T[i+|P|]$.

Analysis

A *false match* when $f(x) = f(y)$ but $x \neq y$. For our fingerprint, this occurs when $f(x) \equiv f(y)\ (mod\, p)$ which is exactly when $f(x) - f(y)$ is a multiple of p. (Slight notational difficulty here: P with a capital letter is the input text pattern we want to find in the text and is a string. p with a lower case is the random prime number that we pick.)

What is the probability that a *false match* occurs? This is exactly the following probability:

$$\Pr\,(n \text{ is a multiple of } p)$$

where $n = f(x) - f(y)$. We upper bound this probability as follows: how many prime factors does n have? Clearly, this is at most $\log n$. As long as the prime we choose is not one of these, we will be fine. Suppose we choose p as a random prime from $[2, Q]$. By the Prime Number Theorem, there are roughly $Q/\log Q$ primes in $[2, Q]$. Therefore, we have that:

$$\Pr(n \text{ is a multiple of } p) \leq \frac{\log n}{Q/\log Q}$$

Using the *Union Bound*, which states that:

$$\Pr(E_1 \cup E_2) \leq \Pr(E_1) + \Pr(E_2)$$

We have that:

$$\Pr(\exists \text{ a failing test}) \leq |T| \frac{\log n}{Q/\log Q}.$$

By choosing Q appropriately large, we can make this probability vanishingly small, so that our algorithm will almost always be correct.

Machine Model

One aspect that we did ignore in our analysis has been the machine model. We have assumed that we can compare and manipulate the fingerprints in $O(1)$ time. We reasonably assume that our machine can manipulate *machine words* in $O(1)$ time. A 32-bit machine word can address an incredibly large range, so it is reasonable to have our machine can manipulate, in constant time, machine words of size $O(\log n)$, where n is the problem size. This assumption is known as the *Transdichotomous assumption.*

For the Rabin-Karp string matching algorithm, we need to manipulate integers up to size Q. For $Q = (|T| * |P|)^2$ so that the probability of a false match is $\leq \frac{1}{|T| * |P|}$, we have $\log Q = 2(\log |T| + \log |P|)$. Therefore, using this machine model, Rabin-Karp can be run in $O(|T| + |P|)$.

Comments

Notice that even if Rabin-Karp finds a mismatch, we can easily check the result to see if it really did match. But now we must perform the check every time the fingerprints match, which increases the runtime.

The version of Rabin-Karp as stated above is known as a *Monte Carlo Algorithm*, which is "always fast and probably right."

If we check the results and slow it down, it becomes a *Las Vegas Algorithm*, which is "always right, and probably fast."

Given a Monte Carlo algorithm, it is easy to convert it to a Las Vegas algorithm if we can check our answers. Going the reverse way is more difficult.

1.1.4 Suffix Trees

We have discussed the string matching problem from the algorithms perspective and described a randomized algorithm to solve the problem in just $O(|T| + |P|)$ time. Here, we discuss a data structures perspective.

Although Rabin-Karp is "fast," we may want to preserve the work we have done if we have multiple queries. We want to design a data structure that can answer multiple queries efficiently.

Let n be the length of the text and m be the length of the pattern. We want to preprocess the text in $O(n)$ time and produce a data structure that takes $O(n)$ space and can answer queries in $O(m)$ time.

Trees and Tries

Here we give a description of various data structures that we might try to use:

1. **Tree Dictionary** Keep a set of k strings in a binary tree dictionary. Check if the pattern is one of the strings in tree. This would take $O(m \log k)$ time to do a search. The problem here is that we would have to store all $n(n+1)/2$ substrings in the dictionary. This is clearly too large a value for k.

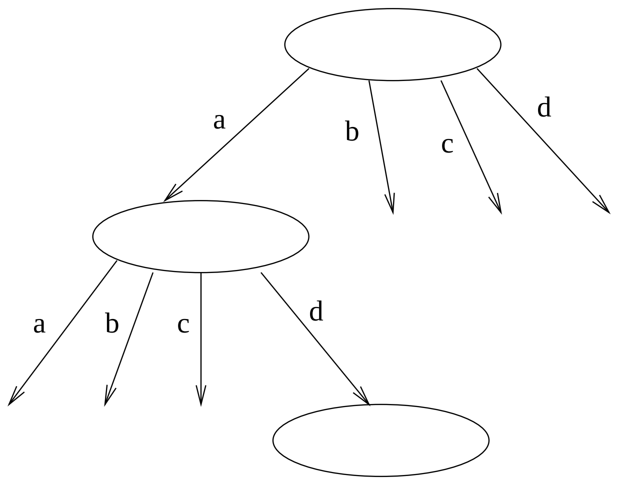

Figure 1.2: An example of a trie on {a,b,c,d}.

2. **Trie** A trie is another type of search tree. The difference between a trie and a regular search tree is that the outgoing edges of a node are the ones that correspond to characters. A walk down the path of a trie corresponds to iterating through the characters of a string. See Figure 1.2 for an example. The nodes store the index information for a search. At each node, the edges are implemented by an array of size $|\Sigma|$, where Σ is the alphabet and is thus directly indexable. Therefore, tries can search for a pattern in $O(m)$ time! The problem with a trie is that it also requires the storage of all possible substrings of the text and would take $O(n^2)$ time to build the trie.

3. **Suffix Tree** Since we realize that storing all possible substrings is highly redundent, instead build a trie of only the n *suffixes* of the text. Any substring of the text is a *prefix* to *some suffix* in the text; therefore, it reaches some internal node of the trie. We label each leaf node

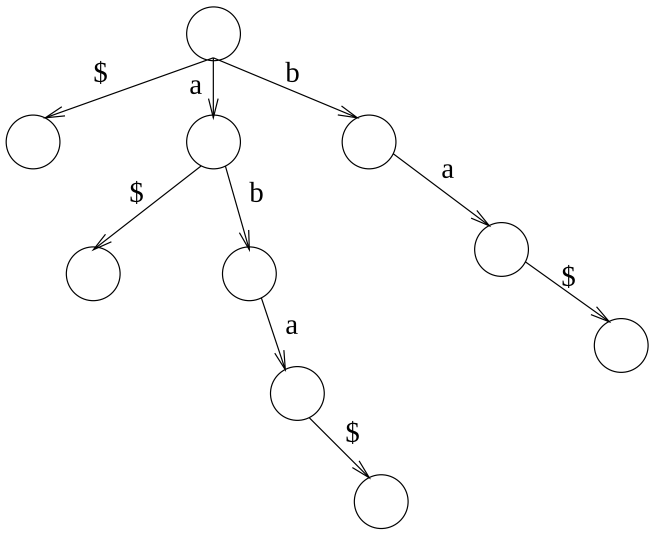

Figure 1.3: A suffix tree on aba$

of the suffix tree with the starting position of the suffix it represents. In order to ensure that every suffix is represented by a leaf node, we append to the input text a special character \$. The size of such a suffix tree is still $O(n^2)$. If the suffix tree is constructed in a naive way, for every character in every suffix we have to either perform a search or create a new node, requiring a total construction time of $\Theta(n^2)$.

Suffix Tree Construction

The goal will be to try to build suffix trees in linear time and space. We begin an attempt at that here.

Observation: Suppose we just inserted aw, where a is a character and w is a string, we know that the next insertion is going to be w.

Next, we define a "suffix link", which will be central to our linear time construction of suffix trees.

Definition 2 ***Suffix Links*** *A suffix link is a pointer from the node representing* ax *to the node representing* x*.*

We build a suffix tree by inserting the longest suffix (the entire word), then the second longest suffix, etc., and inserting suffix links from the word just inserted to the last word inserted. The following makes this precise.

Insertion Algorithm Assume aw was just inserted. Keep the pointer at the bottom of the path walked to insert aw. Walk up until we find a suffix link or reach the root. If we reached a suffix link, traverse it. If you found the root, stay there. Walk down as much as we walked up, extending the trie as needed. In other words, search/insert the substring of w you just walked up. For each node we walked up, add new suffix links.

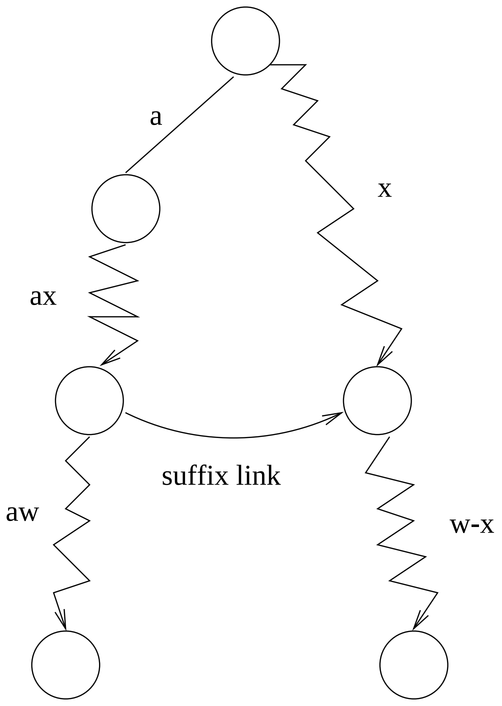

Figure 1.4: Suffix Tree Construction

Analysis

Observe that during construction, we never ascend past a node with a suffix link and each node we ascend gets a suffix link after this pass. Therefore, the total ascent work is at most the number of suffix links, which is at most the number of nodes in trie.

Since we descend as much as we ascend, the total work is Θ(size of suffix tree).

Even if we use suffix links, construction time is still not $O(n)$ since $|T| = \Theta(n^2)$. We will see next lecture how to *compress* the tree so that we can achieve $O(n)$ in both space and time.

6.854 Advanced Algorithms

Lecture 1: 10/13/2004 Lecturer: David Karger

Scribes: Jay Kumar Sundararajan

Duality

This lecture covers weak and strong duality, and also explains the rules for finding the dual of a linear program, with an example. Before we move on to duality, we shall first see some general facts about the location of the optima of a linear program.

1.1 Structure of LP solutions

1.1.1 Some intuition in two dimensions

Consider a linear program -

Maximize $y^T b$
subject to $y^T A \leq c$

The feasible region of this LP is in general, a convex polyhedron. Visualize it as a polygon in 2 dimensions, for simplicity. Now, maximizing $y^T b$ is the same as maximizing the projection of the vector y in the direction represented by vector b. For whichever direction b we choose, the point y that maximizes $y^T b$ cannot lie strictly in the interior of the feasible region. The reason is that, from an interior point, we can move further in any direction, and still be feasible. In particular, by moving along b, we can get to a point with a larger projection along b. This intuition suggests that the optimal solution of an LP will never lie in the interior of the feasible region, but only on the boundaries. In fact, we can say more. We can show that for any LP, the optimal solutions are always at the "corners" of the feasible region polyhedron. This notion is formalized in the next subsection.

1.1.2 Some definitions

Definition 1 (Vertex of a Polyhedron) *A point in the polyhedron which is uniquely optimal for some linear objective, is called a vertex of the polyhedron.*

Definition 2 (Extreme Point of a Polyhedron) *A point in the polyhedron which is not* *a convex combination of two other points in the polyhedron is called an extreme point of the polyhedron.*

Definition 3 (Tightness) *A constraint of the form $a^T x \leq b$, $a^T x = b$ or $a^T x \geq b$ in a linear program is said to be tight for a certain point y, if $a^T y = b$.*

Definition 4 (Basic Solution) *For an n-dimensional linear program, a point is called a basic solution, if n linearly independent constraints are tight for that point.*

Definition 5 (Basic Feasible Solution) *A point is a basic feasible solution, iff it is a basic solution that is also feasible.*

Note: If x is a basic feasible solution, then it is in fact, the unique point that is tight for all its tight constraints. This is because, there can be only one solution for a set of n linearly independent equalities, in n-dimensional space.

Theorem 1 *For a polyhedron P and a point $x \in P$, the following are equivalent:*

1. *x is a basic feasible solution*
2. *x is a vertex of P*
3. *x is an extreme point of P*

Proof: Assume the LP is in the canonical form.

1. **Vertex$\Rightarrow$ Extreme Point**
   Let v be a vertex. Then for some objective function c, $c^T x$ is uniquely minimized at v. Assume v is not an extreme point. Then, v can be written as $v = \lambda y + (1 - \lambda)z$ for some y, z neither of which is v, and some λ satisfying $0 \leq \lambda \leq 1$.

   Now, $c^T v = c^T[\lambda y + (1 - \lambda)z] = \lambda c^T y + (1 - \lambda)c^T z$

   This means $c^T y \leq c^T v \leq c^T z$. But, since v is a minimum point, $c^T v \leq c^T y$ and $c^T v \leq c^T z$. Thus, $c^T y = c^T v = c^T z$. This is a contradiction, since v is the unique point at which $c^T x$ is minimized.

2. **Extreme Point $\Rightarrow$ Basic Feasible Solution**
   Let x be an extreme point. By definition, it lies in the polyhedron and is therefore feasible. Assume x is not a basic solution. Let T be the set of rows of the constraint matrix A for which the constraints are tight at x. Let a_i (a $1 \times n$ vector) denote the

i^{th} row of A. For $a_i \notin T$, $a_i.x > b_i$. Since x is not a basic solution, T does not span $\mathcal{R}^n$. So, there is a vector $d \neq 0$ such that $a_i.d = 0 \; \forall a_i \in T$.

Consider $y = x + \epsilon d$ and $z = x - \epsilon d$. If $a_i \in T$, then $a_i.y = a_i.z = b_i$. If $a_i \notin T$, then, by choosing a sufficiently small ϵ: $0 < \epsilon \leq min_{i \notin T} \frac{a_i.x - b_i}{|a_i.d|}$, we can ensure that $a_i.y \geq b_i$ and $a_i.z \geq b_i$. Thus y and z are feasible. Since $x = y/2 + z/2$, x cannot be an extreme point – a contradiction.

3. **Basic Feasible Solution $\Rightarrow$ Vertex**
Let x be a basic feasible solution. Let $T = \{i \mid a_i.x = b_i\}$. Consider the objective as minimizing $c.y$ for $c = \sum_{i \in T} a_i$. Then, $c.x = \sum_{i \in T}(a_i.x) = \sum_{i \in T} b_i$.
For any $x' \in \mathcal{P}$, $c.x' = \sum_{i \in T}(a_i.x') \geq \sum_{i \in T} b_i$ with equality only if $a_i.x' = b_i \; \forall i \in T$. This implies that $x' = x$ and that x uniquely minimizes the objective $c.y$.

This proves that vertex, extreme point and basic feasible solution are equivalent terms. ■

Theorem 2 *Any bounded LP in standard form has an optimum at a basic feasible solution.*

Proof: Consider an optimal x which is not a basic feasible solution. Being optimal, it is feasible, hence it is not basic. As in the previous proof, let T be the set of rows of the constraint matrix A for which the constraints are tight at x. Since x is not a basic solution, T does not span $\mathcal{R}^n$. So, there is a vector $d \neq 0$ such that $a_i.d = 0 \; \forall a_i \in T$. For a scalar ϵ with sufficiently small absolute value, $y = x + \epsilon d$ is feasible, and represents a line containing x in the direction d. The objective function at y is $c^T x + \epsilon c^T d$. Since x is optimal, $c^T d = 0$, as otherwise, an ϵ of the opposite sign can reduce the objective. This means, all feasible points on this line are optimal. One of the directions of motion on this line will reduce some x_i. Keep going till some x_i reduces to 0. This results in one more tight constraint than before.

This technique can be repeated, till the solution becomes basic. ■

Thus, we can convert any feasible solution to a basic feasible solution of no worse value. In fact, this proof gives an algorithm for solving a linear program: evaluate the objective at all basic feasible solutions, and take the best one. Suppose there are m constraints and n variables. Since a set of n constraints defines a basic feasible solution, there can be upto $\binom{m}{n}$ basic feasible solutions. For each set of n constraints, a linear system of inequalities has to be solved, which by Gaussian elimination, takes $O(n^3)$ time. This is in general an exponential complexity algorithm in n. Note that the output size is polynomial in n, since the optimal solution is just the solution of a system of linear equalities.

1.2 The dual of a linear program

Given an LP in the standard form:

$$\begin{gathered}\text{Minimize } c.x \\ \text{subject to: } Ax = b; x \geq 0\end{gathered}$$

We call the above LP the primal LP. The decision version of the problem is: Is the optimum $c.x \leq \delta$? This problem is in NP, because, if we find a feasible solution with optimum value $\leq \delta$, we can verify that it satisfies these requirements, in polynomial time. A more interesting question is whether this problem is in *co-NP*. We need to find an easily verifiable proof for the fact that there is no x which satisfies $c.x < \delta$. To do this, we require the concept of duality.

1.2.1 Weak Duality

We seek a lower bound on the optimum. Consider a vector y (treat is as a row vector here). For any feasible x, $yAx = yb$ holds. If we require that $yA \leq c$, then $yb = yAx \leq cx$. Thus, yb is a lower bound on cx, and in particular on the optimum cx. To get the best lower bound, we need to maximize yb. This new linear program:

$$\begin{gathered}\text{Maximize } yb \\ \text{subject to: } yA \leq c\end{gathered}$$

is called the *dual linear program.* (Note: The dual of a dual program is the primal). Thus primal optimum is lower bounded by the dual optimum. This is called *weak duality.*

Theorem 3 (Weak Duality) *Consider the LP $z = Min\{c.x \mid Ax = b, x \geq 0\}$ and its dual $w = max\{y.b \mid yA \leq c\}$. Then $z \geq w$.*

Corollary 1 *If the primal is feasible and unbounded, then the dual is infeasible.*

1.3 Strong Duality

In fact, if either the primal or the dual is feasible, then the two optima are equal to each other. This is known as *strong duality.* In this section, we first present an intuitive explanation of the theorem, using a gravitational model. The formal proof follows that.

1.3.1 A gravitational model

Consider the LP $\min\{y.b|yA \geq c\}$. We represent this feasible region as a hollow polytope, with the vector b pointing "upwards". If a ball is dropped into the polytope, it will settle down at the lowest point, which is the optimum of the above LP. Note that any minimum is a global minimum, since the feasible region of an LP is a convex polyhedron. At the equilibrium point, there is a balance of forces – the gravitational force and the normal reaction of the floors (constraints). Let x_i represent the amount of force exerted by the i^{th} constraint. The direction of this force is given by the i^{th} column of A. Then the total force exerted by all the constraints Ax balances the gravity b: $Ax = b$.

The physical world also gives the constraints that $x \geq 0$, since the floors' force is always outwards. Only those floors which the ball touches exert a force. This means that for the constraints which are not tight, the corresponding x_i's are zero: $x_i = 0$ if $yA_i > c_i$. This can be summarized as

$$(c_i - yA_i)x_i = 0$$

. This means x and y satisfy:

$$y.b = \sum yA_i x_i = \sum c_i x_i = c.x$$

But weak duality says that $yb \leq cx$, for every x and y. Hence the x and y are the optimal solutions of their respective LP's. This implies strong duality – the optima of the primal and dual are equal.

1.3.2 A formal proof

Theorem 4 (Strong Duality) *Consider $w = min\{y.b \mid yA \geq c\}$ and $z = min\{c.x \mid Ax = b, x \geq 0\}$. Then $z = w$.*

Proof: Consider the LP $\min\{y.b|yA \geq c\}$. Consider the optimal solution y^*. Without loss of generality, ignore all the constraints that are loose for y^*. If there are any redundant constraints, drop them. Clearly, these changes cannot alter the optimal solution. Dropping these constraints leads to a new A with fewer columns and a new shorter c. We will prove that the dual of the new LP has an optimum equal in value to the primal. This dual optimal solution can be extended to an optimal solution of the dual of the original LP, by filling in zeros at places corresponding to the dropped constraints. The point is that we do not need those constraints to come up with the dual optimal solution.

After dropping those constraints, at most n tight constraints remain (where n is the length of the vector y). Since we have removed all redundancy, these constraints are linearly independent. In terms of the new A and c, we have new constraints $yA = c$. y^* is still the optimum.

Claim: There exists an x, such that $Ax = b$.
Proof: Assume such an x does not exist, i.e. $Ax = b$ is infeasible. Then "duality" for linear equalities implies that there exists a z such that $zA = 0$, but $zb \neq 0$. Without loss of generality, assume $z.b < 0$ (otherwise, just negate the z). Now consider $(y^* + z)$. $A(y^* + z) = Ay^* + Az = Ay^*$. Hence, it is feasible. $(y^* + z).b = y^*.b + z.b < y^*.b$, which is better than the assumed optimum – a contradiction. So, there is an x such that $Ax = b$. Let this be called x^*.

Claim: $y^*.b = c.x^*$.
Proof: $y^*.b = y^*.(Ax^*) = (y^*A).x^* = c.x^*$ (since $Ax^* = b$ and $y^*A = c$)

Claim: $x^* \geq 0$
Proof: Assume the contrary. Then, for some i, $x_i^* < 0$. Let $c' = c + e_i$, where e_i is all 0's except at the i^{th} position, where it has a 1. Since A has full rank, $yA \geq c'$ has a solution, say y'. Besides, since $c' \geq c$, y' is feasible for the original constraints $yA \geq c$. But, $y'.b = y'Ax^* = c'x^* < cx^* = y^*b$ (since c_i' is now higher and $x_i < 0$). This means y' gives a better objective value than the optimal solution – a contradiction. Hence, $x^* \geq 0$.

Thus, there is an x^* which is feasible in the dual, and whose objective is equal to the primal optimum. Hence, x^* must be the dual optimal solution, using weak duality. Thus, the optima of primal and dual are equal.

■

Corollary 2 *Checking for feasibility of a linear system of inequalities and optimizing an LP are equally hard.*

Proof: Optimizer → Feasibility checker
Use the optimizer to optimize any arbitrary function with the linear system of inequalities as the constraints. This will automatically check for feasibility, since every optimal solution is feasible.

Feasibility checker → Optimizer
We construct a reduction from the problem of finding an optimal solution of LP_1 to the problem of finding a feasible solution of LP_2. LP_1 is $min\{c.x \mid Ax = b, x \geq 0\}$. Consider $LP_2 = min\{0.x | Ax = b, x \geq 0, yA \leq c, c.x = b.y\}$. Any feasible solution of LP_2 gives an optimal solution of LP_1 due to the strong duality theorem. Finding an optimal solution is thus no harder than finding a feasible solution. ■

1.4 Rules for duals

Usually the primal is constructed as a minimization problem and hence the dual becomes a maximization problem. For the standard form, the primal is given by:

$$\begin{aligned} z &= \min\ (c^T x) \\ Ax &\geq b \\ x &\geq 0 \end{aligned}$$

while the dual is given by:

$$\begin{aligned} w &= \max\ (b^T y) \\ A^T y &\leq c \\ y &\geq 0 \end{aligned}$$

For a mixed form of the primal, the following describes the dual:

Primal:

$$\begin{aligned} z &= \min\ c_1x_1 + c_2x_2 + c_3x_3 \\ A_{11}x_1 + A_{12}x_2 + A_{13}x_3 &= b_1 \\ A_{21}x_1 + A_{22}x_2 + A_{23}x_3 &\geq b_2 \\ A_{31}x_1 + A_{32}x_2 + A_{33}x_3 &\leq b_3 \\ x_1 &\geq 0 \\ x_2 &\leq 0 \\ x_3 & \quad \text{UIS} \end{aligned}$$

(UIS = unrestricted in sign)

Dual:

$$\begin{aligned} w &= \max\ y_1b_1 + y_2b_2 + y_3b_3 \\ y_1A_{11} + y_2A_{21} + y_3A_{31} &\leq c_1 \\ y_1A_{12} + y_2A_{22} + y_3A_{32} &\geq c_2 \\ y_1A_{13} + y_2A_{23} + y_3A_{33} &= c_3 \end{aligned}$$

$$\begin{aligned} y_1 & \quad \text{UIS} \\ y_2 & \geq 0 \\ y_3 & \leq 0 \end{aligned}$$

These rules are summarized in the following table.

PRIMAL	Minimize	Maximize	DUAL
Constraints	$\geq b_i$	≥ 0	Variables
	$\leq b_i$	≤ 0	
	$= b_i$	Free	
Variables	≥ 0	$\leq c_j$	Constraints
	≥ 0	$\leq c_j$	
	Free	$= c_j$	

Each variable in the primal corresponds to a constraint in the dual, and vice versa. For a maximization, an upper bound constraint is a "natural" constraint, while for a minimization, a lower bound constraint is natural. If the constraint is in the natural direction, then the corresponding dual variable is non-negative.

An interesting observation is that, the tighter the primal gets, the looser the dual gets. For instance, an equality constraint in the primal leads to an unrestricted variable in the dual. Adding more constraints in the primal leads to more variables in the dual, hence more flexibility.

1.5 Shortest Path – an example

Consider the problem of finding the shortest path in a graph. Given a graph G, we wish to find the shortest path from a specified source node, to all other nodes. This can be formulated as a linear program:

$$w = \max\ (d_t - d_s)$$

$$\text{s.t. } d_j - d_i \leq c_{ij}, \qquad \forall i, j$$

In this formulation, d_i represents the distance of node i from the source node s. The c_{ij} constraints are essentially the triangle inequalities – the distance from the source to a node i should not be more than the distance to some node j plus the distance from j to

i. Intuitively, one can imagine stretching the network physically, to increase the source-destination distance. When we cannot pull any further without breaking an edge, we have found a shortest path.

The dual to this program is found thus. The constraint matrix in the primal has a row for every pair of nodes (i,j), and a column for every node. The row corresponding to (i,j) has a +1 in the i^{th} column and a -1 in the j^{th} column, and zeros elsewhere.

1. Using this, we conclude that the dual has a variable for each pair (i,j), say y_{ij}.
2. It has a constraint for each node i. The constraint has a coefficient of +1 for each edge entering node i and a -1 for each edge leaving i. The right side for the constraints are -1 for the node s constraint, 1 for the node t constraint, and 0 for others, based on the objective function in the primal. Moreover, all the constraints are equality constraints, since the d_i variables were unrestricted in sign in the primal.
3. The dual variables will have to have a non-negativity constraint as well, since the constraints in the primal were "natural" (upper bounds for a maximization).
4. The objective is to minimize $\sum_{i,j} c_{ij}y_{ij}$, since the right side of the primal constraints are c_{ij}.

Thus the dual is:

$$z = \min \sum_{i,j} c_{ij}y_{ij}$$

$$\begin{aligned}
\sum_j (y_{js} - y_{sj}) &= -1 \\
\sum_j (y_{jt} - y_{tj}) &= 1 \\
\sum_j (y_{ji} - y_{ij}) &= 0, \forall i \neq s,t \\
y_{ij} &\geq 0, \forall i,j
\end{aligned}$$

This is precisely the linear program to solve the minimum cost unit flow, in a gross flow formulation. The constraints correspond to the flow conservation at all nodes except at the source and sink. The value of the flow is forced to be 1. Intuitively, this says that we can use minimum cost unit flow algorithms to find the shortest path in a network.

Duality is a very useful concept, especially because it helps to view the optimization problem on hand from a different perspective, which might be easier to handle.

6.854 Advanced Algorithms

Lecture 10: 10/04/2004 Lecturer: David Karger

Scribe: Wendy Chang

Min-Cost Flow Algorithms

10.1 Shortest Augmenting Paths: Unit Capacity Networks

The shortest augmenting path algorithm for solving the MCF problem is the natural extension of the SAP algorithm for the max flow problem. Note that here the shortest path is defined by edge cost, not edge capacity.

For the unit capacity graph case, we assume that all arcs have unit capacity and that there are no negative cost arcs. Therefore, the value of any flow in the cycle must be less than or equal to n. Given that each augmenting path increases the value of the flow by 1, at most n augmentation steps will suffice in finding the MCF.

Shortest augmenting paths can be found using any single-source shortest path algorithm. We can use Dijkstra's algorithm since there are no negative-cost edges in the graph. Each path calculation takes $O(m \log n)$ time, for a total runtime of $O(nm \log n)$.

Two questions arise:

- what if augmentations create negative cost edges?
- how do we know the result is a MCF?

We answer both of these questions with the following claim.

Claim 1 *Under the SAP algorithm, there will never be a negative reduced-cost cycle in the residual graph.*

Proof: (by induction). We want to show that one SAP doesn't introduce negative cycles in G_f. Initially there are no negative cost cycles. Feasible prices can be computed by using shortest path distances from s. After finding the shortest s-t path, it has reduced cost 0. Every arc on the path has reduced length 0. This demonstrates that the triangle inequality property is tight on shortest path edges. When we augment along the path, therefore, the residual backwards arcs we create are of reduced cost 0. Therefore in the new G_f, the price function is still feasible. Furthermore, there are:

- no residual negative reduced cost arcs

- no negative reduced cost cycles
- no negative cost cycles

■

Proof of this claim also proves the correctness of the algorithm, since it will also apply to the residual graph at the time the algorithm terminates.

The SAP algorithm we present suffers from two limitations. It is applicable only to unit capacity graphs, and it cannot handle graphs with negative cost cycles.

10.2 MCF Scaling by Capacity: General Networks

We can extend the SAP algorithm to general-capacity networks by scaling. During each scaling phase, we roll in one bit of precision, for a total of $O(\log U)$ phases.

At the end of each phase we have an MCF and a feasible price function. After rolling in the next bit, though, we can introduce residual capaicty on negative reduced cost arcs. This will cause the price function no longer to be feasible. We can correct this problem by sending flow along the negative arcs. This introduces flow excesses (of one unit) at some nodes and deficits (of one unit) at others. We use an MCF to send the excesses back to deficits.

Since each arc can create at most one unit of excess, total excess is at most m units and m SAPs will suffice in returning all excesses to deficits. Using Dijkstra's for finding SAPs as before, runtime per phase is $O(m^2 \log n)$. The total runtime of the algorithm is $O(m^2 \log n \log U)$.

10.3 MCF Scaling by Cost

An alternative method of solving for MCF in a general network is by scaling by costs, rather than capacities. This is useful for graphs with integral costs, since all cycles will have integer costs. The idea is to allow for slightly negative cost arcs and continuously improve on the price function. We introduce the idea of ϵ-*optimality*:

Definition 1 *A price function p is ϵ-*optimal *if for all residual arcs (i, j), $c_p(i, j) \geq -\epsilon$.*

We start with a max flow and a zero price function, which will be C-optimal. During each phase, we go from an ϵ-optimal max flow to an $(\epsilon/2)$-optimal max flow. When can we terminate the algorithm?

Claim 2 *A $\frac{1}{n+1}$-optimal max flow is optimal.*

Proof: We start with the observation that the least negative cycle cost is -1 in a integral-cost graph. All cycles in the residual network cost at least $-\frac{n}{n+1}$, which is strictly larger than -1. Therefore the reduced cost of any residual cycle is at least $-\frac{n}{n+1}$, and a $\frac{1}{n+1}$-optimal max flow is optimal. ■

To get an $(\epsilon/2)$-optimal max flow from an ϵ-optimal max flow, we first saturate all negative-cost residual arcs. This makes all residual arcs have non-negative reduced cost, but introduces excesses and deficits into the network. We then use MCF to push the excesses back to the deficits, without allowing any edge costs to drop below $\epsilon/2$.

Using dynamic trees, the runtime of this algorithm is $O(mn \log n \log C)$.

10.4 State of the Art

The double-scaling algorithm combines cost- and capacity-scaling introduced here. It has the runtime of $O(mn \log C \log\log U)$.

Tardos' minimum mean-cost cycles algorithm ('85) is a strongly polynomial algorithm for MCF. The algorithm proceeds by finding the negative cycles in which the average cost per edge is most strongly negative. Thus short cycles of a particular negativity are preferred over long ones. The algorithm uses a cost scaling technique from the ideas of ϵ-optimality. After every m negative-cycle saturations, an edge becomes "frozen," meaning its flow value never changes again. The minimum mean-cost cycle algorithm has time bound $O(m^2 \text{ polylog } m)$.

18.415/6.854 Advanced Algorithms October 22, 2001

Lecture 11

Lecturer: Michel X. Goemans *Scribe: Mohammad Hajiaghayi, Vahab Mirrokni*

In the last lecture, we saw a cycle canceling algorithm for the minimum cost circulation problem. In this lecture we present a strongly polynomial time algorithm. This note is mainly based on the network flows lecture notes of previous years.

The problem of finding a strongly polynomial algorithm (and even its existence) for the minimum cost circulation problem was open for several years. In 1985, Éva Tardos developed the first such algorithm. In 1987, Goldberg and Tarjan produced an improved version presented below.

1 The Goldberg-Tarjan Algorithm

In the last lecture, we saw the following algorithm.

Goldberg-Tarjan algorithm:

1. Let $f = 0$.
2. While $\mu(f) < 0$ do
   push $\delta = \min_{(v,w)\in\Gamma} u_f(v,w)$ along a minimum mean cost cycle Γ of G_f.

The Goldberg-Tarjan algorithm is a cycle canceling algorithm since G_f has a negative directed cycle iff $\mu(f) < 0$.

2 Analysis of the Goldberg-Tarjan Algorithm

Before analyzing the Goldberg-Tarjan cycle canceling algorithm, we reveiw some definitions.

Definition 1 *A circulation f is ϵ-optimal if there exists p such that $c_p(v,w) \geq -\epsilon$ for all $(v,w) \in E_f$.*

For $\epsilon = 0$, we know that there exist potential p such that $c_p(v,w) \geq 0$ for $(v,w) \in E_f$, hence a 0-optimal circulation is a minumum cost circulation.

Definition 2 *$\epsilon(f)$ = minimum ϵ such that f is ϵ-optimal. In other words,*

$$\epsilon(f) = min\{\epsilon \mid \exists p : c_p(v,w) \geq -\epsilon \ \forall\, (v,w) \in E_f\}.$$

In this section, assume that all $c(u,w)$ are integers. We will see noninteger costs in this lecture note but not in this part. Also we have the following theorems which were proved in the last lecture.

Theorem 1 *If f is a circulation with $\epsilon(f) < \frac{1}{n}$ then f is optimal.*

Theorem 2 *For any circulation f, $\mu(f) = -\epsilon(f)$.*

We are now ready to analyze the algorithm. First, we show that, using $\epsilon(f)$ as a measure of near-optimality, the algorithm produces circulations which are closer and closer to optimal.

Theorem 3 *Let f be a circulation and let f' be the circulation obtained by canceling the minimum mean cost cycle Γ in E_f. Then $\epsilon(f) \geq \epsilon(f')$.*

Proof: By definition, there exists p such that

$$c_p(v,w) \geq -\epsilon(f) \tag{1}$$

for all $(v,w) \in E_f$. Moreover, for all $(v,w) \in \Gamma$, we have $c_p(v,w) = -\epsilon(f)$ since, otherwise, its mean cost would not be $-\epsilon(f)$. We claim that, for the same p, (1) holds for all $(v,w) \in E_{f'}$. Indeed, if $(v,w) \in E_{f'} \cap E_f$, (1) certainly holds. If $(v,w) \in E_{f'} \setminus E_f$ then (w,v) certainly belongs to Γ. Hence, $c_p(v,w) = -c_p(w,v) = \epsilon(f) \geq 0$ and (1) is also satisfied. □

Next, we show that $\epsilon(f)$ decreases after a certain number of iterations.

Theorem 4 *Let f be any circulation and let f' be the circulation obtained by performing m iterations of the Golberg-Tarjan algorithm. Then $\epsilon(f') \leq (1 - \frac{1}{n})\epsilon(f)$.*

Proof: Let p be such that $c_p(v,w) \geq -\epsilon(f)$ for all $(v,w) \in E_f$. Let Γ_i be the cycle canceled at the ith iteration. Let k be the smallest integer such that there exists $(v,w) \in \Gamma_{k+1}$ with $c_p(v,w) \geq 0$. We know that canceling a cycle removes at least one arc with negative reduced cost from the residual graph and creates only arcs with positive reduced cost. Therefore $k \leq m$. Let f' be the flow obtained after k iterations. By Theorem 2, $-\epsilon(f')$ is equal to the mean cost of Γ_{k+1} which is:

$$\begin{aligned} \frac{\sum_{(v,w)\in\Gamma_{k+1}} c_p(v,w)}{l} &\geq \frac{-(l-1)}{l}\epsilon(f) \\ &= -(1-\frac{1}{l})\epsilon(f) \geq -(1-\frac{1}{n})\epsilon(f), \end{aligned}$$

where $l = |\Gamma_{k+1}|$. Therefore, by Theorem 3, after m iterations, $\epsilon(f)$ decreases by a factor of $(1 - \frac{1}{n})$. □

Assuming that all $c(u,w)$ are integers, we have the following theorem:

Theorem 5 *Let $C = \max_{(v,w)\in E} |c(v,w)|$. Then the Goldberg-Tarjan algorithm finds a minimum cost circulation after canceling $nm\log(nC)$ cycles ($\log = \log_e$).*

Proof: The initial circulation $f = 0$ is certainly C-optimal since, for $p = 0$, we have $c_p(v,w) \geq -C$. Therefore, by Theorem 4, the circulation obtained after $nm \log nC$ iterations is $\epsilon-$optimal where:

$$\epsilon \leq \left(1 - \frac{1}{n}\right)^{n\log(nC)} C < e^{-\log(nC)}C = \frac{C}{nC} = \frac{1}{n},$$

where we have used the fact that $(1 - \frac{1}{n})^n < e^{-1}$ for all $n > 0$. The resulting circulation is therefore optimal by Theorem 1. □

The overall running time of the Goldberg-Tarjan algorithm is therefore $O(n^2m^2\log(nC))$ since the minimum mean cost cycle can be obtained in $O(nm)$ time.

3 Cancel and Tighten Algorithm

We can improve the algorithm presented in the previous sections by using a more flexible selection of cycles for canceling and explicitly maintaining potentials to help identify cycles for canceling. The idea is to use the potentials we get from the minimum mean cost cycle to compute the edge costs $c_p(v,w)$ and then push flow along all cycles with only negative cost edges. The algorithm Cancel and Tighten is described below.

Cancel and Tighten:

1. Cancel: As long as there exists a cycle Γ in G_f with $c_p(v,w) < 0, \forall (v,w) \in \Gamma$ push as much flow as possible along Γ.
2. Tighten: Compute a minimum mean cost cycle in G_f and update p.

We now show that the Cancel step results in canceling at most m cycles each iteration and the flow it gives is $(1-1/n)\epsilon(f)$ optimal.

Theorem 6 *Let f be a circulation and let f' be the circulation obtained by performing the Cancel step. Then we cancel at most m cycles to get f' and*

$$\epsilon(f') \le (1 - \frac{1}{n})\epsilon(f).$$

Proof: Let p be such that $c_p(v,w) \ge -\epsilon(f)$ for all $(v,w) \in E_f$. Let Γ be any cycle in f' and let l be the length of Γ. We know that canceling a cycle removes at least one arc with negative reduced cost from the residual graph and creates only arcs with positive reduced cost. Therefore we can cancel at most m cycles. Now $G_{f'}$ has no negative cycles therefore every cycle in $G_{f'}$ contains an edge (v,w) such that $c_p(v,w) \ge 0$. Hence the mean cost of Γ is at least:

$$\begin{aligned} \frac{\sum_{(v,w)\in\Gamma} c_p(v,w)}{l} &\ge \frac{-(l-1)}{l}\epsilon(f) \\ &= -(1-\frac{1}{l})\epsilon(f) \ge -(1-\frac{1}{n})\epsilon(f), \end{aligned}$$

□

The above result implies that the Cancel and Tighten procedure finds a minimum cost circulation in at most $n\log(nC)$ iterations (by an analysis which is a replication of Theorem 5). It also takes us $O(n)$ time to find a cycle on the admissible graph. This implies that each Cancel step takes $O(nm)$ steps due to the fact that we cancel at most m cycles and thus a running time of $O(nm)$ for one iteration of the Cancel and Tighten Algorithm. Therefore the overall running time of Cancel and Tighten is $O(n^2m\log(nC))$ (i.e. an amortized time of $O(n)$ per cycle canceled). We can further improve this by using dynamic trees to get an amortized time of $O(\log n)$ per cycle canceled and this results in an $O(nm\log n\log(nC))$ algorithm.

4 A Strongly Polynomial Bound

In this section, we give another analysis of the algorithm. This analysis has the advantage of showing that the number of iterations is strongly polynomial, i.e. that it is polynomial in n and m and does not depend on C. The first strongly polynomial algorithm for the minimum cost circulation problem is due to Tardos.

Definition 3 *An arc $(v,w) \in E$ is $\epsilon-$fixed if $f(v,w)$ is the same for all $\epsilon-$optimal circulations.*

There exists a simple criterion for deciding whether an arc is $\epsilon-$fixed.

Theorem 7 *Let $\epsilon > 0$. Let f be a circulation and p be node potentials such that f is ϵ-optimal with respect to p. If $|c_p(v,w)| \geq 2n\epsilon$ then (v,w) is $\epsilon-$fixed.*

Proof: The proof is by contradiction. Let f' be an ϵ-optimal circulation for which $f'(v,w) \neq f(v,w)$. Assume that $|c_p(v,w)| \geq 2n\epsilon$. Without loss of generality, we can assume by antisymmetry that $c_p(v,w) \leq -2n\epsilon$. Hence $(v,w) \notin E_f$, i.e. $f(v,w) = u(v,w)$. This implies that $f'(v,w) < f(v,w)$. Let $E_< = \{(x,y) \in E : f'(x,y) < f(x,y)\}$.

Claim 8 *There exists a cycle Γ in $(V, E_<)$ that contains (v,w).*

Proof: Since $(v,w) \in E_<$, it is sufficient to prove the existence of a directed path from w to v in $(V, E_<)$. Let $S \subseteq V$ be the nodes reachable from w in $(V, E_<)$. Assume $v \notin S$. By flow conservation, we have

$$\sum_{x \in S, y \notin S} (f(x,y) - f'(x,y)) = \sum_{x \in S} \sum_{y \in V} (f(x,y) - f'(x,y)) = 0.$$

However, $f(v,w) - f'(v,w) > 0$, i.e. $f(w,v) - f'(w,v) < 0$, and by assumption $w \in S$ and $v \notin S$. Therefore, there must exists $x \in S$ and $y \notin S$ such that $f(x,y) - f'(x,y) > 0$, implying that $(x,y) \in E_<$. This contradicts the fact that $y \notin S$. □

By definition of $E_<$, we have that $E_< \subseteq E_{f'}$. Hence, the mean cost of Γ is at least $\mu(f') = -\epsilon(f') = -\epsilon$. On the other hand, the mean cost of Γ is ($l = |\Gamma|$):

$$\begin{aligned} \frac{c(\Gamma)}{l} &= \frac{c_p(\Gamma)}{l} = \frac{1}{l}\left(c_p(v,w) + \sum_{(x,y) \in \Gamma \setminus \{(v,w)\}} c_p(x,y) \right) \\ &\leq \frac{1}{l}(-2n\epsilon + (l-1)\epsilon) < \frac{1}{l}(-l\epsilon) = -\epsilon, \end{aligned}$$

a contradiction. □

Theorem 9 *The Goldberg-Tarjan algorithm terminates after $O(m^2 n \log n)$ iterations.*

Proof: If an arc becomes fixed during the execution of the algorithm, then it will remain fixed since $\epsilon(f)$ does not increase. We claim that, as long as the algorithm has not terminated, one additional arc becomes fixed after $O(mn \log n)$ iterations. Let f be the current circulation and let Γ be the first cycle canceled. After $mn \log(2n)$ iterations, we obtain a circulation f' with

$$\epsilon(f') \leq \left(1 - \frac{1}{n}\right)^{n \log(2n)} \epsilon(f) < e^{-\log(2n)} \epsilon(f) = \frac{\epsilon(f)}{2n}$$

by Theorem 6. Let p' be potentials for which f' satisfies the $\epsilon(f')$-optimality constraints. By definition of Γ,

$$-\epsilon(f) = \frac{c_{p'}(\Gamma)}{|\Gamma|}.$$

Hence,

$$\frac{c_{p'}(\Gamma)}{|\Gamma|} < -2n\epsilon(f').$$

Therefore, there exists $(v, w) \in \Gamma$ such that $|c_{p'}(v, w)| > -2n\epsilon(f')$. By the previous Theorem, (v, w) is $\epsilon(f')-$fixed. Moreover, (v, w) is not $\epsilon(f)-$fixed since canceling Γ increased the flow on (v, w). This proves that, after $mn \log(2n)$ iterations, one additional arc becomes fixed and therefore the algorithm terminates in $m^2 n \log(2n)$ iterations. $\square$

Using the $O(mn)$ algorithm for the minimum mean cost cycle problem, we obtain a $O(m^3 n^2 \log n)$ algorithm for the minimum cost circulation problem. Using the Cancel and Tighten improvement we obtain a running time of $O(m^2 n^2 \log n)$. And if we implement Cancel and Tighten with the dynamic trees data structure we get a running time of $O(m^2 n \log^2 n)$.

The best known strongly polynomial algorithm for the minimum cost circulation problem is due to Orlin and runs in $O(m \log n(m + n \log n)) = O(m^2 \log n + mn \log^2 n)$ time.

6.854 Advanced Algorithms

Lecture 12: October 20, 2003 Lecturers: Erik Demaine, David Karger

Scribe: Jason Eisenberg

Approximation Algorithms

12.1 Introduction

So far in class, we have studied problems which are efficiently solvable (solvable in polynomial time), and we have asked how quickly we can solve them. For the next few lectures, however, we will consider problems which are not known to be efficiently solvable.

12.1.1 NP-Completeness

In studying such problems, we encounter the notion of NP-completeness. NP-complete problems comprise a family of thousands of distinct combinatorial and optimization problems for which

- no efficient algorithms are known; we can, however, use brute force to solve these problems in exponential time
- an efficient reduction exists from every other NP-complete problem; thus, if we have a black box which is able to solve one of these problems efficiently, we can solve all of the problems efficiently

Below are listed some examples of NP-complete problems. Each problem except SAT is formulated as an optimization problem, although strictly speaking it is the decision version of each problem that is NP-complete. As an aside, there does exist an optimization problem related to SAT, called MAXSAT. In this problem, we are given a boolean formula, and we must find an assignment to the variables which maximizes the number of satisfied clauses.

Satisfiability (SAT): Given a boolean formula, is there an assignment to the variables which satisfies the formula (makes the formula evaluate to true)?

Bin Packing: Given a set of items of specified sizes and unit-size bins, determine the minimum number of bins required to hold the items.

Max Independent Set: Given a graph, find a maximum-size subset of vertices such that no two vertices in the subset are adjacent.

Knapsack: Given a knapsack of fixed size and a set of items, each of a specified value and size, determine the maximum total value of any set of items which fits in the knapsack.

Parallel Machine Scheduling: Given a set of identical machines and a set of tasks, each of specified duration, find an assignment of tasks to machines which minimizes the time required for all machines to complete their assigned tasks.

Traveling Salesman Problem (TSP): Find a minimum-distance route through a set of cities which allows a salesman to begin and end in the same city and visit every other city exactly once.

That all of these problems cannot be solved efficiently depends on the assumption that P $\neq$ NP. Although this conjecture is not proven, it is widely believed to be true, and so we will simply assume that P $\neq$ NP.

12.1.2 Coping with NP-Completeness

Since we do not know how to solve NP-complete problems efficiently, what can we do?

heuristics: One possibility is to abandon searching for polynomial-time algorithms and to instead concentrate on developing heuristics which are almost polynomial-time in practice for instances which are not too large. But in general, it would be surprising to obtain algorithms which are even subexponential, as achieving this goal would have ramifications as to whether P and NP are equal.

average-case analysis: Rather than looking at the worst-case performance of algorithms for these problems, we can analyze the algorithms' behavior only on certain classes of inputs. This is done by determining their expected performance over some specified distribution of the input instances. But there is often much disagreement over which distribution to use.

approximation algorithms: We can attempt to find polynomial-time approximation algorithms which can be proven to be *approximately* correct.

We will explore the topic of approximation algorithms over the next few lectures.

12.2 Optimization Problems

Before discussing approximation algorithms, we must establish some terminology for optimization problems.

Definition 1 *An optimization problem has a set of* **problem instances**.

Definition 2 *Each instance I has a* **solution set S(I)**.

Definition 3 *The maximization/minimization problem is to find a solution $s \in S(I)$ of maximum/minimum objective* **value f(s)**. *We will assume the input and output of f are integers composed of a polynomial number of bits.*

Definition 4 *The value $f(s)$ of an optimum solution s for instance I is denoted* **OPT(I)**.

Each of the optimization problems given in Section 12.1.1 fits into this optimization framework. For example, consider the Max Independent Set problem. Each problem instance is a graph; the solution set for a graph consists of all subsets of vertices such that no two vertices in each subset are adjacent; and the value of a solution is the number of vertices in the subset.

Although we would like to refer to these optimization problems as being NP-complete, this term is usually reserved for decision problems and languages. Instead, we use the concept of NP-hardness.

Definition 5 *An optimization problem is* **NP-hard** *if some other NP-hard problem can be reduced to it in polynomial time.*

Usually, the NP-hard problem used in the reduction is the corresponding decision problem of whether $OPT(I)$ is at least (or at most) some value k.

12.3 Absolute Approximation Algorithms

Definition 6 *An* **approximation algorithm** *is a polynomial-time algorithm which when given an instance I, returns a solution s in the solution space $S(I)$.*

For example, in the bin-packing problem, one possible approximation algorithm is to simply place each item in its own bin. But doing so most likely produces a poor quality solution. To address the issue of quality, let us consider absolute approximation algorithms.

Definition 7 *Given an instance I, an* **α-absolute approximation algorithm** *finds a solution of value at most $OPT(I) + \alpha$.*

Note that this definition only makes sense for minimization problems; an α-absolute approximation algorithm for a maximization problem would return a solution of value at least $OPT(I) - \alpha$. Further, observe that when designing an absolute approximation algorithm, we would like α to be as small as possible.

12.3.1 Algorithms for Graph Coloring

Consider the problem of **planar graph coloring**, in which we are given a planar graph (one which can be drawn in a plane without its edges crossing) and we must find a coloring of the vertices such that no two neighboring vertices have the same color. As the following theorem demonstrates, this problem possesses an absolute approximation algorithm.

Theorem 1 *A 2-absolute approximation algorithm exists for planar graph coloring.*

Proof: By the Five Color Theorem, every planar graph is 5-colorable. Further, note that empty graphs (graphs without edges) are 1-colorable, bipartite graphs are 2-colorable, while all other graphs require at least 3 colors. These observations lead to the following algorithm:

1. If the graph is empty or bipartite, color it optimally.

2. Otherwise, color it with 5 colors.

Since this algorithm only uses 5 colors when the optimum number of colors is at least 3, it is a 2-absolute approximation algorithm. ■

We can also consider the problem of **edge-coloring**, in which we color the edges rather than the vertices. Unlike in planar graph coloring, there is no constant upper bound on the number of colors required, since the optimum number $OPT(I)$ is lower-bounded by the maximum vertex degree Δ. Nevertheless, we have the following theorem due to Vizing:

Theorem 2 *The edges of any graph can be colored using at most* $\Delta + 1$ *colors.*

Since his proof of the theorem is constructive, it provides us with an algorithm for finding an edge-coloring using at most 1 more color than the optimum.

Corollary 1 *A 1-absolute approximation algorithm exists for edge-coloring.*

12.3.2 Proving Negative Examples by Scaling

Although these coloring problems possess absolute approximation algorithms, most NP-hard problems do not. In fact, for most of these problems, we can prove that an absolute approximation algorithm cannot exist unless P equals NP. Such proofs use a technique called **scaling**. In scaling, we first increase (scale) certain parameters of the problem instance. We then show that if an absolute approximation algorithm existed, the solution it would provide for the modified instance could be rescaled to yield an optimum solution for the original instance. But this would imply the existence of an efficient algorithm for an NP-hard problem, and thus P would equal NP.

The following two examples illustrate the use of scaling.

Claim 1 *An absolute approximation algorithm does not exist for the Knapsack problem.*

Proof: Consider a Knapsack problem instance I in which each of the items i has profit p_i, and suppose we have an α-absolute approximation algorithm A for the problem. If we double the profit of each item to $2p_i$ to form a new instance (call it $2I$), the resulting optimum solution $OPT(2I)$ is twice the original optimum solution $OPT(I)$, since the set of items which originally yielded a profit of $OPT(I)$ now yields a profit of $2OPT(I)$. If we then run A on instance $2I$, we obtain a solution of at least $OPT(2I) - \alpha = 2OPT(I) - \alpha$. Finally, dividing this result by 2 yields a solution to the original instance of at least $OPT(I) - \alpha/2$. Thus, we have improved the value of α by a factor of 2.

In general, scaling the original instance I by a factor of r and then dividing the resulting solution of A by r allows us to reduce α to α/r. Hence, if we choose r to be $\lceil 2\alpha \rceil$, we can reduce α to $\alpha/\lceil 2\alpha \rceil \leq 1/2$, implying A can be used to obtain a solution s for I of at least $OPT(I) - 1/2$. If we assume I has integer sizes and profits, the maximum achievable profit $OPT(I)$ is also an integer, and so s must equal $OPT(I)$. Consequently, we have an efficient algorithm for solving integer instances of Knapsack, which contradicts our assumption that P $\neq$ NP. ■

Claim 2 *An absolute approximation algorithm does not exist for Max Independent Set.*

Proof: Suppose we have an α-absolute approximation algorithm A. If we modify an instance I by making a copy of the graph (call this new instance $2I$), the size of the optimum independent set in $2I$ is twice that in I. Thus, $OPT(2I)$ equals $2OPT(I)$, which implies that running A on instance $2I$ yields an independent set of size of at least $OPT(2I) - \alpha = 2OPT(I) - \alpha$. To transform this solution into one for the original I, we count the number of vertices in the independent set of $2I$. One of the graphs must have at least half of these vertices, and thus that graph has an independent set of size at least $(2OPT(I) - \alpha)/2 = OPT(I) - \alpha/2$, implying we have reduced α by a factor of 2. Generalizing this result, if we make $\lceil 2\alpha \rceil$ copies of the graph, we can use A to find an independent set of size at least $OPT(I) - 1/2$ in I. But this must equal $OPT(I)$, since the number of vertices is integral. Thus, we have an efficient algorithm for solving Max Independent Set, which is a contradiction. ■

12.4 Relative Approximation Algorithms

Since absolute approximation algorithms are known to exist for so few optimization problems, a better class of approximation algorithms to consider are relative approximation algorithms. Because they are so commonplace, we will refer to them simply as approximation algorithms.

Definition 8 *An* **α-approximation algorithm** *finds a solution of value at most $\alpha \cdot OPT(I)$.*

Note that although α can vary with the size of the input, we will only consider those cases in which it is a constant. To illustrate the design and analysis of an α-approximation algorithm, let us consider the Parallel Machine Scheduling problem, a generic form of load balancing.

Parallel Machine Scheduling: Given m machines m_i and n jobs with processing times p_j, assign the jobs to the machines to minimize the load

$$\max_i \sum_{j \in i} p_j,$$

the time required for all machines to complete their assigned jobs. In scheduling notation, this problem is described as P || Cmax.

A natural way to solve this problem is to use a greedy algorithm called **list scheduling**.

Definition 9 *A* **list scheduling** *algorithm assigns jobs to machines by assigning each job to the least loaded machine.*

Note that the order in which the jobs are processed is not specified. To analyze the performance of list scheduling, we must somehow compare its solution for each instance I (call this solution $A(I)$) to the optimum $OPT(I)$. But we do not know how to obtain an analytical expression for $OPT(I)$. Nonetheless, if we can find a meaningful lower bound $LB(I)$ for $OPT(I)$ and can prove that $A(I) \leq \alpha \cdot LB(I)$ for some α, we then have

$$\begin{aligned} A(I) &\leq \alpha \cdot LB(I) \\ &\leq \alpha \cdot OPT(I). \end{aligned}$$

Using this idea of lower-bounding $OPT(I)$, we can now determine the performance of list scheduling.

Claim 3 *List scheduling is a 2-approximation algorithm for Parallel Machine Scheduling.*

Proof: Consider the following two lower bounds for the optimum load $OPT(I)$:

- the maximum processing time $p = \max_j p_j$
- the average load $L = \sum_j p_j / m$

The maximum processing time p is clearly a lower bound, as the machine to which the corresponding job is assigned requires at least time p to complete its tasks. To see that the average load is a lower bound, note that if all of the machines could complete their assigned tasks in less than time L, the maximum load would be less than the average, which is a contradiction. Now suppose machine m_i has the maximum runtime $c_{\max}$, and let job j be the last job that was assigned to m_i. At the time job j was assigned, m_i must have had the minimum load (call it L_i), since list scheduling assigns each job to the least loaded machine. Thus,

$$\begin{aligned} L_i &\leq \text{average load when } j \text{ assigned} \\ &\leq \text{final average load } L, \end{aligned}$$

since the average load can only increase. Assigning job j to m_i added at most p to L_i, which implies that

$$\begin{aligned} c_{\max} &\leq L_i + p \\ &\leq L + p \\ &\leq 2OPT(I) \quad (L \text{ and } p \text{ are lower bounds for } OPT(I)). \end{aligned}$$

The solution returned by list scheduling is $c_{\max}$, and thus list scheduling is a 2-approximation algorithm for Parallel Machine Scheduling. ■

It is possible to show that by modifying list scheduling to assign the jobs in decreasing order of processing time, we obtain a 4/3-approximation algorithm. Further, note that list scheduling is an online algorithm. Newer online algorithms are able to achieve an α of about 1.8.

12.5 Polynomial Approximation Schemes

The obvious question to now ask is how good an α we can obtain.

Definition 10 *A* **polynomial approximation scheme (PAS)** *is a set of algorithms $\{A_\epsilon\}$ for which each A_ϵ is a polynomial-time $(1+\epsilon)$-approximation algorithm.*

Thus, given any $\epsilon > 0$, a PAS provides an algorithm that achieves a $(1+\epsilon)$-approximation. How do we devise a PAS? The most common method used is k-enumeration.

Definition 11 *An approximation algorithm using* **k-enumeration** *finds an optimal solution for the k most important elements in the problem and then uses an approximate polynomial-time method to solve the remainder of the problem.*

For example, an approximation algorithm which uses k-enumeration to solve Parallel Machine Scheduling is as follows:

1. Enumerate all possible assignments of the k largest jobs.
2. For each of these partial assignments, list schedule the remaining jobs.
3. Return as the solution the assignment with the minimum load.

Note that in enumerating all possible assignments of the k largest jobs, the algorithm will always find the optimal assignment for these jobs. The following claim demonstrates that this algorithm provides us with a PAS.

Claim 4 *For any fixed m, k-enumeration yields a polynomial approximation scheme for Parallel Machine Scheduling.*

Proof: As in the proof of Claim 3, let us consider the machine m_i with maximum runtime $c_{\max}$ and the last job j that m_i was assigned. If this job is not among the k largest, it was assigned during list scheduling, at which point there were at least k larger jobs which had already been scheduled. Thus, when job j was assigned, the average load (call it L_{assigned}) must have been at least kp_j/m, which implies that

$$\begin{aligned} p_j &\leq \frac{mL_{\text{assigned}}}{k} \\ &\leq \frac{mL}{k}. \end{aligned}$$

Since $c_{\max}$ is the sum of p_j and the load on m_i before job j was assigned (which was shown in the proof of Claim 3 to be at most L), we have

$$\begin{aligned} c_{\max} &\leq L + p_j \\ &\leq (1 + \frac{m}{k})L \\ &\leq (1 + \frac{m}{k})OPT(I). \end{aligned}$$

Given an $\epsilon > 0$, if we let k equal m/ϵ,

$$c_{\max} \leq (1+\epsilon)OPT(I).$$

This bound on $c_{\max}$ also holds if job j is among the k largest. In this case, job j is scheduled optimally, and $c_{\max}$ thus equals $OPT(I)$. Finally, to determine the running time of the algorithm, note that because each of the k largest jobs can be assigned to any of the m machines, there are $m^k = m^{m/\epsilon}$ possible assignments of these jobs. Since the list scheduling performed for each of these assignments takes $O(n)$ time, the total running time is $O(nm^{m/\epsilon})$, which is polynomial because m is fixed. Thus, given an $\epsilon > 0$, the algorithm is a $(1+\epsilon)$-approximation, and so we have a polynomial approximation scheme. ■

Obviously, we would prefer an approximation algorithm for which m does not have to be fixed. As a first step towards achieving this goal, let us reconsider Parallel Machine Scheduling when there are only k possible sizes (or types) of jobs, where k is bounded by a constant. In this case, it is possible to find an optimum solution in polynomial time using dynamic programming. Note that each set of jobs can be described by its "profile," the number of each type of job, and the number of possible profiles is at most n^k, which is polynomial in the input size. The dynamic program computes the function $M(a_1, a_2, \ldots, a_k)$, the minimum number of machines needed to complete the a_i type-i jobs in some fixed time T. After enumerating the set X of all profiles which can be completed by a single machine in time T and using X to initialize the appropriate entries in the table, the remaining entries are computed using

$$M(a_1, a_2, \ldots, a_k) = 1 + \min_{(x_1, x_2, \ldots, x_k) \in X} M(a_1 - x_1, a_2 - x_2, \ldots, a_k - x_k).$$

Thus, the minimum number of machines required to complete a profile y is found by exhaustively removing all possible single-machine profiles from y and looking up the minimum number of machines required to complete the rest of y. Finally, to determine the optimum time required to complete all of the jobs, the dynamic program can be used as a "subroutine" in a binary search over the values of T.

6.854 Advanced Algorithms

Lecture 14: 10/27/2003 Lecturer: David Karger

Scribe: Sachin Katti

Relaxation Techniques

14.1 Introduction

Given a problem P we solve a different relaxed version of the problem $P^{'}$ which is related to P. $P^{'}$ is chosen such that every solution to P is feasible for $P^{'}$. Hence the optimum for $P^{'}$ bounds the optimum for P.

But once $P^{'}$ has been solved, we have to find a way to convert that solution to that of P which will involve some conversion cost. This will bound the approximation algorithm. We consider some examples to illustrate the technique.

14.2 Travelling Salesman Problem

14.2.1 Problem formulation

Given a set of cities(vertices V) and distances(edge lengths E) between them the TSP is to find a hamiltonian cycle (i.e. a tour such that each vertex is visited exactly once) of minimum cost.

14.2.2 Metric TSP

The problem is NP-hard, hence there is no approximation algorithm for the problem. But if we restrict the problem to a TSP with the triangle inequality i.e. $d(i,j) + d(j,k) < d(i,k)$ for all $i, j, k \in V$, we can find approximation algorithms. This problem is called the Metric TSP. Consider the following algorithm

- Construct a Minimum Spanning Tree for the graph.
- Take an Euler tour around the MST.

The idea is to remove the constraint of takinbg a tour, instead visit everything in as cheap a manner as possible. Since every tour without the last edge is a spanning tree, Cost of MST < Cost of TSP. But in the Euler tour each edge is visited twice. Hence

$$\text{Cost of Euler Tour} \leq 2^*\text{Cost of MST} \leq 2^*\text{Cost of Opt TSP} \tag{14.1}$$

To get a tour, we just shortcut the Euler tour, i.e. when we are visitng a vertex for the second time while going back, instead take the shortest path from the previous vertex to the next vertes on the tour. Due to the triangle inequality above, the cost of the resulting tour will be less than that of the Euler tour. Thus we have a 2 approximation algorithm.

Christofedes' heuristic for Metric TSP

From the above algorithm we see that basically we need to backtrack from odd degree vertices. Hence if we can modify the graph such that all vertices have even degree, we can find a much better approximation. Hence the algorithm is as follows

- Compute the minumum spanning tree T of the graph $G = (V, E)$.
- Let O be the odd degree vertices in T. $|O|$ is even since the sum of the degress of vertices in T is even and if we consider only the odd degree vertices, their sum must add up to an even number which implies that $|O|$ is even.
- Compute a mim-cost perfect matching M on the graph induced by O.
- Add the edges in M to E. Now the degree of every vertex in G is even, Therefor G has an Eulerian tour. Trace the tour and take shortcuts when the same vertex is reached twice. This cannot increase the cost due to the triangle inequality.

We claim that the cost of M is no more than half the cost of $Opt(TSP)$. Consider the optimal tour visiting only the vertices in O. Clearly by the triangle inequality, this is of length no more than $Opt(TSP)$. There are an even number of vertices in the tour and hence even number of edges. The tour defines two disjoint matchings on the graph induced by O. Atleast one of these has cost $\leq \frac{1}{2} Opt(TSP)$, and hence the cost of M is no more than this.

Therefore the total cost of the resulting Eulerian tour is no more than $\frac{3}{2} Opt(TSP)$ and hence we get a $\frac{3}{2}$ approximation algorithm.

14.2.3 Directed TSP

The directed TSP can be relaxed to a problem of finding a min cost cycle cover. In min cost cycle cover we find a way to cover all the vertices with directed cycles. To go back to solving TSP, we replace each cycle by a vertex and recursively apply the above algorithm until we can find a TSP tour on the reduced graph. Once we have found a tour we use it to patch the cycles together into one cycle.
The patching cost is at most 2 times that of the cycle cover since we might have to go at most twice over each edge. Hence we get the following recursion

$$\text{Cost}(n) = 2 * \text{Cost}(\text{Cycle Cover}) + \text{Cost}(\frac{n}{2}) \leq 2 * \text{Cost}(\text{TSP}) + \text{Cost}(\frac{n}{2}) \leq 2 * \text{logn} * (\text{Cost}(\text{TSP})) \tag{14.2}$$

14.3 LP relaxation techniques

Almost any problem can be expressed as an integer LP. But integer LP can be relaxed very easily by just allowing non-integer solutuions. We then have to find a way of converting the non-integer solutions back to integer solutions. We use the vertex cover problem to illustrate the technique.

14.3.1 Minimum Cost Vertex Cover

Problem Formulation

A vertex cover U in a graph $G = (V, E)$ is a subset of the vertices such that every edge is incident to atleast one vertex in U. The vertex cover problem is defined as follows

Definition 1 ***Given a graph*** $G = (V, E)$ ***and costs*** $c(v)$ ***for each vertex*** $v \in V$***, find a vertex cover*** $U \in V$ ***which minimizes*** $c(u) = \sum_{v \in U} c(v)$***.***

LP formulation and relaxation

The corresponding integer linear program for the problem is as follows

$$C_{opt} = min \sum_{v \in V} c(v)x(v) \tag{14.3}$$

where

$$x(v) + x(w) \geq 1 \qquad \forall (v, w) \in E \tag{14.4}$$

$$x(v) \in \{0, 1\} \qquad \forall v \in V \tag{14.5}$$

This LP can be relaxed by allowing non integer solutiuons as follows

$$LB = min \sum_{v \in V} c(v)x(v) \tag{14.6}$$

where

$$x(v) + x(w) \geq 1 \qquad \forall (v, w) \in E \tag{14.7}$$

$$x(v) \geq 0 \qquad \forall v \in V \tag{14.8}$$

Rounding

Let x^* be the optimal solution of the LP relaxation. Let

$$U = \{v \in V : x^*(v) \geq \frac{1}{2}\} \tag{14.9}$$

We claim U is a 2-approximation of the minimum cost VC. Clearly U is a vertex cover because for $(u, v) \in E$ we have $x^*(u) + x^*(w) \geq 1$, which implies $x^*(u) \geq \frac{1}{2}$ or $x^*(v) \geq \frac{1}{2}$. Also

$$\sum_{v \in U} c(v) \leq \sum_{v \in V} c(v) 2x^*(v) = 2 * LB \tag{14.10}$$

since $2x^*(v) \geq 1$ for all $v \in U$.
Since $LB \leq C_{Opt}$ we have a 2-approximation algorithm for min-cost VC.

14.3.2 Max SAT

Problem Formulation

Given a collection of OR clauses C_j over some literals x_i, find an assignment of boolean values to the literals which maximises the number of true clauses.
The approximation technique is to use a random guess for each x_i. Since they are OR clauses

$$\text{Probability(given clause is satisfied)} \geq \frac{1}{2} \tag{14.11}$$

Hence

$$\text{E[No of satisfied clauses]} \geq \frac{\text{\# of clauses}}{2} \tag{14.12}$$

LP formulation and relaxation

Now consider the integer LP formulation for the problem

$$N_{opt} = max \sum z_j \tag{14.13}$$

$$\sum_{X_i \text{ positive in } C_j} y_i + \sum_{x_i \text{ negated in } C_j} (1 - y_i) \geq z_j \qquad 0 \leq y_i \leq 1 \qquad 0 \leq z_j \leq 1 \tag{14.14}$$

where

$$z_j = 1 \text{ if } C_j \text{ is satisfied else } 0 \tag{14.15}$$

$$y_i = 1 \text{ if } x_i \text{ is true, else } 0 \tag{14.16}$$

Now relaxing the integer LP, we allow non-integer solutions. To convert the solution back to the original problem we use the following rounding technique:

Set x_i true with probability y_i.
We claim that a k variable literal clause C_j is satisfied with probability $\geq \beta_k z_j$ where

$$\beta_k = 1 - (1 - \frac{1}{k})^k \tag{14.17}$$

To prove the above claim consider the following

$$\text{Probability}(C_j \text{ is satisfied}) = 1 - \prod_{x_i \in C_j} 1 - y_i \tag{14.18}$$

This probability is minimized when all of the y_is are equal, i.e. $y_i = \frac{z_j}{k}$. Hence a clause C_j is satisfied with probability $\geq (1 - \frac{1}{e})$.
Hence

$$\text{E[No. of satisfied clauses]} = N_{\text{expected}} \geq (1 - \frac{1}{e}) \sum z_j \tag{14.19}$$

But since $\sum z_j \geq N_{opt}$, the expected number of satisfied clauses, $N_{\text{expected}} \geq (1 - \frac{1}{e}) N_{opt}$. Hence we have a $(1 - \frac{1}{e})$ approximation algorithm.

6.854 Advanced Algorithms

Lecture 15: October 15, 2003 Lecturer: David Karger and Erik Demaine

Scribes: Nelson Lai

15.1 Addendum from last lecture

Theorem 1 *If the primal P (primal) or D (dual) are feasible, then they have the same value.*

15.2 Rules for Taking Duals

In general we construct the primal P as a minimization problem and, conversely, the dual D as a maximization problem. If P is a linear program in standard form given by:

$$\begin{aligned} z &= \min(c^T x) \\ Ax &\geq b \\ x &\geq 0 \end{aligned}$$

then the dual, D is given by:

$$\begin{aligned} w &= \max(b^T y) \\ A^T y &\leq c \\ y &\geq 0 \end{aligned}$$

In general, the form of the dual will depend on the form of the primal. If one is given a primal linear program P in mixed form:

$$\begin{aligned} x &= \min(c_1x_1 + c_2x_2 + c_3x_3) \\ A_{11}x_1 + A_{12}x_2 + A_{13}x_3 &= b_1 \\ A_{21}x_1 + A_{22}x_2 + A_{23}x_3 &\geq b_2 \\ A_{31}x_1 + A_{32}x_2 + A_{33}x_3 &\leq b_3 \\ x_1 &\geq 0 \\ x_2 &\leq 0 \\ x_3 & \quad \text{unrestricted in sign (UIS)} \end{aligned}$$

then the dual D is given by:

$$\begin{aligned}
w &= \max(b_1y_1 + b_2y_2 + b_3y_3) \\
y_1A_{11} + y_2A_{21} + y_3A_{31} &\le c_1 \\
y_1A_{12} + y_2A_{22} + y_3A_{32} &\ge c_2 \\
y_1A_{13} + y_2A_{23} + y_3A_{33} &= c_3 \\
y_1 & \quad \text{unrestricted in sign (UIS)} \\
y_2 &\ge 0 \\
y_3 &\le 0
\end{aligned}$$

By simple transformations, we can confirm that this is consistent with the dual for the standard form of the primal and that in fact the dual of the dual is the primal.

We can summarize these results with the following table which states the rules for taking duals. Note that each variable in the primal corresponds to a variable in the dual and each constraint in the primal corresponds to a variable in the dual.

PRIMAL	minimize	maximize	DUAL
constraints	$\ge b_i$ $\le b_i$ $= b_i$	≥ 0 ≤ 0 unrestricted	variables
variables	≥ 0 ≥ 0 unrestricted	$\le c_j$ $\le c_j$ $= c_j$	constraints

Note that this makes intuitive sense. For example, the primal minimization problem has lower bounds as the natural constraints. This corresponds to a positive variable in the dual maximization problem. Conversely, the primal maximization problem has upper bounds as natural constraints. The dual minimization problem now has a negative variable.

To develop an intuition for these relationships, we consider the effect of the sign of a variable in the minimization problem on the type of the corresponding constraint in the maximization problem. We know from weak duality that $c^Tx \ge yb = yAx$. Consider the case where $x_1 \ge 0$. Then in order to have $yAx_1 \le c_1x_1$, we must have $yA_{11} \le c_1$ for any y. Similarly, if $x_2 \le 0$, then we must have $yA_{12} \ge c_2$ in order for $c^Tx \ge yAx$. Finally, for x_3 unrestricted, we must have $yA_{13} = c_3$ since multiplying both sides by x might or might not change the direction of any inequality. In general, tighter constraints in the primal lead to looser constraints in the dual. An equal constraint leads to an unrestricted variable and adding new constraints creates new variables and more flexibility.

15.3 Shortest Paths

We now examine an example showing the relationship between the primal and dual problems. We consider formulating the shortest paths problem as a linear program. Given a graph G, we wish to find the shortest path from any one point (the source) to any other point. We formulate the problem as a dual (or maximization) linear program.

$$\begin{aligned} w &= \max(d_t - d_s) \\ d_j - d_i &\leq c_{ij} \\ d_j & \quad \text{unrestricted} \end{aligned}$$

Each variable d_i represents the distance to vertex i and each constraint represents the triangle inequality — that is, the the distance to vertex i should always be less than or equal to the distance to vertex j plus the distance from vertex j to vertex i. Any feasible solution to this would find a lower bound to the shortest path distances — the maximization objective makes sure these shortest path distances are valid. You can imagine physically holding up the source and the sink and pulling them apart slowly. The first time we cannot pull any further, this indicates the shortest path has been reached.

The constraint matrix A has n^2 rows and n columns of ± 1 or 0. Each row ij has a 1 in column i, -1 in column j, and 0 in all others. Thus we can write the primal as follows:

$$\begin{aligned} z &= \min(c^T x) \\ &= \sum_{i,j} c_{ij} x_{ij} \\ \sum_{j=1}^{n} x_{js} - x_{sj} &= -1 \\ \sum_{j=1}^{n} x_{jt} - x_{tj} &= 1 \\ \sum_{j=1}^{n} x_{ji} - x_{ij} &= 0 \qquad \forall i \neq s, t \end{aligned}$$

But this is simply a linear program for a minimum cost unit-flow! The constraints represent the conservation of flow with one unit of flow going into the sink and one unit coming out from the source. All other vertices are constrained to have the same amount of flow coming in as going out. Thus any feasible solution to the linear program will be a feasible flow. The objective function simply tries to minimize the cost of this flow. We see that often the dual of a LP allows us to understand the problem from a different (but equivalent) perspective.

15.4 The Gravitational Model

Consider a linear program $\min\{cx \mid Ax \geq b\}$. We consider a hollow polytope defined by a set of constraints. Let c be the gravitation vector, pointing straight up. We can put a ball in the polytope, and let it fall to the bottom.

At equilibrium point x^*, the forces exerted by the floors are balanced by the gravitational force. The normal forces by the floors are aligned with the A_i's. Therefore, we have $c = \sum y_i A_i$ for some **nonnegative** force coefficients y_i. In particular, y^* is a feasible solution for $\max\{yb \mid yA = c, y \geq 0\}$. Since the forces can be only be exerted by those walls touching the ball, we have $y_i = 0$ if $A_i x > b_i$. Therefore, we have

$$y_i(a_i x - b_i) = 0,$$

thus,

$$yb = \sum y_i(a_i x_i) = cx,$$

which means that y^* is dual optimal.

15.5 Complementary Slackness

The above example leads to the idea of *complementary slackness*. Given feasible solutions x and y, $cx - by \geq 0$ is called the *duality gap*. The solutions are optimal if and only if the gap is zero. Therefore, the gap is a good measure of "how far off" we are from the optimum.

Going back to original primal and dual forms, we can rewrite the dual: $yA + s = c$ for some $s \geq 0$ (that is, $s = c_j - yA_j$).

Theorem 2 *The followings are equivalent for feasible x and y:*

- *x and y are optimal*
- *$sx = 0$*
- *$x_j s_j = 0$ for all j*
- *$s_j > 0$ implies $x_j = 0$*

Proof: First, $cx = by$ if and only if

$$(yA + s)x = (Ax)y,$$

thus $sx = 0$. If $sx = 0$, then since $s, x \geq 0$, we have have $s_j x_j = 0$, so of course $s_j > 0$ forces $x_j = 0$. The converse is easy. ■

The basic idea of complementary slackness is that an optimum solution cannot have a variable x_j and corresponding dual constraint s_j slack at same time — one must be tight.

This can be stated in another way:

Theorem 3 *In arbitrary form LPs, feasible points optimal if:*

$$\begin{aligned} y_i(a_i x - b_i) &= 0 \quad \forall i \\ (c_j - yA_j)x_j &= 0 \quad \forall j \end{aligned}$$

Proof: Note that in the definition of primal/dual, feasibility means $y_i(a_i x - b_i) \geq 0$ (since $\geq$ constraint corresponds to nonnegative y_i). Also, $(c_j - yA_j)x_j \geq 0$, thus

$$\begin{aligned} \sum y_i(a_i x - b_i) + (c_j - yA_j)x_j &= yAx - yb + cx - yAx \\ &= cx - yb \\ &= 0 \end{aligned}$$

at optimum. But since all terms are nonnegative, they must be all 0. ■

15.6 Examples Using Complementary Slackness

In some linear optimization problems, we can gain new insight by investigating its primal and dual optimal solutions using complementary slackness. We are going to give two examples. In the first example, we will consider the LP formulation of the maximum flow problem. Using complementary slackness, we derive the *Max-Flow Min-Cut Theorem*. In the second example, we consider the minimum cost circulation problem. Using the linear programming framework, we give an alternative proof of the complementary slackness property introduced in lecture 13 (the lecture on minimum cost flow).

15.6.1 Max-flow Min-Cut Theorem

In the maximum flow problem, we can imagine the network has an arc (t, s) with infinite capacity. And we are maximizing the flow on that arc. Therefore, the max flow problem can be written as follows (in the gross flow form):

$$\begin{aligned} \max x_{ts}& \\ \sum_w x_{vw} - x_{wv} &= 0 \\ x_{vw} &\leq u_{vw} \\ x_{vw} &\geq 0 \end{aligned}$$

In the dual problem, for each node v there is a conservation constraint. Besides, for each edge (v, w) there is a capacity constraint. Therefore, in the primal formulation, we have a variable z_v for each conservation constraint and a variable y_{vw} for each capacity constraint. The primal formulation is therefore:

$$\begin{aligned} \min \sum_{vw} u_{vw} y_{vw} & \\ z_v - z_w + y_{vw} &\geq 0 \\ z_t - z_s + y_{ts} &\geq 1 \\ y_{vw} &\geq 0 \end{aligned}$$

We rewrite the first set of constraints as $y_{vw} \geq z_w - z_v$. Besides, the second constraint can be written as $z_t - z_s \geq 1$. This is because $y_{ts} = 0$ in any optimal solution. If $y_{ts} > 0$ in some optimal solution, the fact that $u_{ts} = \infty$ implies that $u_{ts} y_{ts} = \infty$ and therefore the optimal value is unbounded. This is impossible since the max flow problem is never infeasible (in particular, the zero flow is a feasible solution).

If we consider y_{vw} as the edge length of (v, w) and z_v as the distance from s to v, we can interpret the dual problem as follows: *Minimize the volume of the network by tuning the edge lengths, subject to the constraint that the distance from s to t is at least 1.* Here the volume of network is defined as the sum of edge volumes, which is the product of edge capacity u_{vw} and edge length y_{vw}.

Note that the optimal solution in this primal problem is at most the min-cut value of the network, as we can assign length 1 to the min-cut edges and 0 otherwise. This satisfies the s-t distance constraint (because any s-t path has to traverse some edge of a cut.) The value of this solution is the sum of min-cut edge capacities. By strong duality this implies max-flow $\leq$ min-cut. We now prove the other direction.

Denote z_v^*, y_{vw}^* as an optimal solution for the primal problem and similarly x_{vw}^* for the dual problem. Since z_v^* are distances, we can always rescale z_s^* to 0. Let $T = \{v | z_v^* \geq 1\}$. Note that $s \notin T$ and $t \in T$. Therefore T is a $s - t$ cut.

Now consider any edge (v, w) crossing the cut:

1. if $v \in T$ and $w \notin T$, then $z_v^* \geq 1$ and $z_w^* < 1$. Therefore, $z_v^* - z_w^* + y_{vw}^* \geq z_v^* - z_w^* > 0$. Therefore, the *constraint* for edge (v, w) in the primal problem is slack. By complementary slackness, the *variable* x_{vw}^* in the dual problem has to be tight, i.e., $x_{vw}^* = 0$.
2. if $v \notin T$ and $w \in T$, then $z_w^* \geq 1$ and $z_v^* < 1$. It follows that the *variable* $y_{vw}^* \geq z_w^* - z_v^* > 0$. Again, by complementary slackness, the *constraint* in the dual problem $x_{vw} \leq u_{vw}$ is tight. Therefore, $x_{vw}^* = u_{vw}$.

In other words, in a max flow, all edges entering T is saturated and all edge leaving T is empty. Therefore, in a max flow, the net flow entering T equals the cut value of T. Since the flow value equals the amount of net flow entering any $s - t$ cut, the max-flow value equals the cut value of T, which is at least the min-cut value. As a result, we have shown that max-flow $\geq$ min-cut, which completes the proof of the Max-Flow Min-Cut Theorem.

6.854 Advanced Algorithms

Lecture 17: 11/8/2004 Lecturer: David Karger

Scribes: Matt Rasmussen

Sweep Line

Definition 1 ***Sweep Line Technique:*** *Given some planar problem, sweep a line through the plane dealing with events that occur on the line and leaving behind a solved portion of the problem.*

17.1 Convex Hull

The Convex Hull problem is to find the smallest enclosing convex polygon of a set of given points in the plane.

17.1.1 Algorithm

One method for solving the convex hull problem is to use a sweep line technique to find the upper envelope of the hull. The lower evelope of the convex hull can be found by rerunning the following algorithm with only slight modifications.

Use a vertical sweep line that sweeps from negative infinity to positive infinity on the x-axis. As the line sweeps, we will maintain a partital convex hull for the points left of the sweep line. When the sweep line crosses a new point we will need to update the partial convex hull to include the new point. After the sweep line has gone past all the points, we will have a complete upper envelope of the convex hull.

To determine the order in which the sweep line will cross points we can use a priority queue such a min-heap that will order points by their x-coordinates. Therefore, the only remaining issue is how to insert a new point into an existing convex hull. When inserting a new point, two cases arise: (1) either the existing convex hull can be extended to include the new point or (2) the new point requires the existing convex hull to be modified. Informally, we can distinguish the two cases by noticing that in case (1) the new point causes a "right turn" in the hull's boundary, whereas in case (2) the new point causes a "left turn". This can be determined mathematically by finding the angle between the right most line segment of evelope and the line segment of the right most evelope point and the new point. If the angle is less than 180 degrees then we have a case (1), otherwise we have a case (2).

case 1: Since the new point can be safely added to the existing evelope without violating its convexity, the new point is directly added to the list of evelope points.

case 2: In this case, the new point cannot be safely added to the existing evelope without violating its convexity, therefore, the existing evelope must be modified to accommodate the new point. This can be done by the following procedure. At the new point the evelope. Peform a convexity check on the predecessor to the new point. If the convexity check fails, delete the point from the evelope. Now the new point has a new predecessor. Repeat the convexity check and deletion on the predecessor of the new point until the convexity check is satisfied, at which point we will have a valid upper evelope again.

17.1.2 Analysis

Let there be n points in the problem. Creating the min-heap and performing n extract-mins will have a $O(n \log n)$ runtime. Convexity checks take a constant amount of algebra and therefore take $O(1)$. Case (1) simply extends a linked list in $O(1)$ and may occur a maximum of n times, which implies a $O(n)$ runtime. To find the runtime contribution of case (2), we bound the total number of deletions that may be made from the evelope over the course of the algorithm. This bound is n since each point can be deleted at most once. Therefore, the amortized cost of case (2) is also $O(n)$. This gives a total runtime of $O(n \log n)$.

17.2 Segment Intersections

In the Segment Intersections problem, we are given n line segments and must output the coordinates of all pair-wise intersections.

17.2.1 Algorithm

In this algorithm, we will use a horizontal sweep line that sweeps from positive infinity to negative infinity along the y-axis. The idea, will be to output each intersection as we cross it. Also we will strive to make the algorithm *output sensitive*. That is, although there may be $O(n^2)$ intersections, which would require an $O(n^2)$-time algorithm, we will run in much less time if the number of intersections k is much less than n.

Let a segment be considered “active” if it crosses the current sweep line. As the sweep line sweeps, we will encouter three types of events:

1. new segment becomes active
2. old segment becomes inactive
3. two active segments cross

Events (1) and (2) can be handled with a min-heap containing segment endpoints that are ordered by their y-coordinates. In dealing with the third case, the key idea is that only “neighboring” segments on the sweep line can cross. To provide a quick lookup of neighboring segments we can use a Binary Search Tree (BST) to store the segments by the x-coordinate of their intersection with the sweep line. After inserting a new line into the BST, determine if it will eventually cross with its neighbors. If so, insert a crossing event into the event queue. Later, when a crossing event occurs, we output an intersection, swap the order of the lines segments in the BST, and find their two new neighbors and possible future crossings.

17.2.2 Analysis

Inserting and extracting line segments activations and deactivations from the event queue will take a total of $O(n \log n)$ time. The total time inserting into the BST will take $O(n \log n)$ and the total deletes from the BST will take $O(n)$. The number of crossing events is k, therefore the total time needed for inserting crossing events into the event queue is $O(k \log n)$. Therefore, the total runtime is $O((n + k) \log n)$.

6.854 Advanced Algorithms

Lecture 18: 11/05/2004 Lecturer: David Karger

Scribes: Chun-Chieh Lin

18.1 Lower Bounds for Competitive Ratios of Randomized Online Algorithms

Designing an online algorithm can be viewed as a game between the algorithm designer and the adversary. The algorithm designer chooses an algorithm A_i, and the adversary chooses an input σ_j. The payoff matrix contains the cost of the algorithm on the input $C_{A_i}(\sigma_j)$. The algorithm designer wants to minimize the cost, while the adversary wants to maximize the cost. A randomized online algorithm is a probability distribution over the deterministic algorithms, so it corresponds to a mixed strategy for the algorithm designer.

18.1.1 Game Theory Analysis

Von Neumann proved that for any game, there exist equilibrium (mixed) strategies for the players. At the equilibrium, neither side is able to improve (increase or decrease depending on the player) the cost any further by changing the strategy.

However, problem 4a of problem set 6 showed that if one player's mixed strategy is known (and fixed), the other player has a pure strategy as a best response. That means, if one of the players is using the equilibrium (optimal) mixed strategy, the other player has pure strategy as a best response, and the resulting cost is the equilibrium cost. Again, a pure strategy for the algorithm designer corresponds to a deterministic algorithm, and a mixed strategy for the algorithm designer corresponds to a randomized algorithm, so this leads to Yao's Minimax Principle:

Theorem 1 ***Yao's Minimax Principle:*** *If for some input distribution no deterministic algorithm is k-competitive, then no randomized k-competitive algorithm exists.*

18.1.2 Example: Paging

Suppose there are $k+1$ pages and n accesses, and for each access, the pages all have probability $1/(k+1)$ of being requested. In other words, this is a uniform distribution over inputs with length n of the $k+1$ pages.

Online Algorithm

No matter what the online algorithm does, there are only k pages in the memory at any point in time. So with probability $1/(k+1)$, the requested page is not in the memory. Therefore, the expected number of faults over the n accesses is $n/(k+1)$, and the expected number of requests per fault is $k+1$, which is $\Theta(k)$.

Offline Algorithm

Even though the sequence of requests is still chosen at random, the offline algorithm has access to the whole sequence before it starts running.

As shown in previous lectures, an optimal algorithm for the offline algorithm is the Farthest in Future algorithm, which evicts the page that is requested farthest in the future. This algorithm faults once every $k+1$ distinct pages seen, because after each fault, the evicted page is not requested again until after all other k pages are requested.

The expected number of requests it takes to see all $k+1$ distinct pages can be calculated as follows:

$E[\text{No. requests total}] = \Sigma_{i=1}^{k+1} E[\text{No. requests between the } i-1^{th} \text{ distinct request and the } i^{th}]$

$P(\text{each request after the } i-1^{th} \text{ distinct request is the } i^{th} \text{ distinct request}) = \frac{k+2-i}{k+1}$

$E[\text{No. requests between the } i-1^{th} \text{ distinct request and the } i^{th}] = \frac{k+1}{k+2-i}$

$E[\text{No. requests total}] = \Sigma_{i=1}^{k+1} \frac{k+1}{k+2-i} = (k+1) * (\frac{1}{k+1} + \frac{1}{k} + \frac{1}{k-1} + \frac{1}{k-2} + ... + \frac{1}{1}) = \Theta(klogk)$

Conclusions

The expected number of pages per fault for the online algorithm is $\Theta(k)$. The expected number of pages per fault for the offline algorithm is $\Theta(klogk)$. So the ratio of fault counts is $\Theta(logk)$.

Using Yao's Minimax Principle, this shows that no randomized algorithm can have competitive ratio better than $\Theta(logk)$ for paging.