Design and Analysis of Algorithms: Course Notes

Prepared by

Samir Khuller
Dept. of Computer Science
University of Maryland
College Park, MD 20742
samir@cs.umd.edu
(301) 405 6765

August 14, 2003

Preface

These are my lecture notes from **CMSC 651: Design and Analysis of Algorithms**. This course has been taught several times and each time the coverage of the topics differs slightly. Here I have compiled all the notes together. The course covers core material in data structures and algorithm design, and also helps students prepare for research in the field of algorithms. The reader will find an unusual emphasis on graph theoretic algorithms, and for that I am to blame. The choice of topics was mine, and is biased by my personal taste.

The material for the first few weeks was taken primarily from the textbook on Algorithms by Cormen, Leiserson and Rivest. In addition, I have used material from several other books such as the Combinatorial Optimization book by Papadimitriou and Steiglitz, as well as the Network Flow book by Ahuja, Magnanti and Orlin and the edited book on Approximation Algorithms by Hochbaum. A few papers were also covered, that I personally feel give some very important and useful techniques that should be in the toolbox of every algorithms researcher.

Contents

CMSC 651 Advanced Algorithms	Lecture 1
Lecturer: Samir Khuller	

Original notes by Hsiwei Yu.

1 Overview of Course

The course will cover many different topics. We will start out by studying various combinatorial algorithms together with techniques for analyzing their performance. We will also study linear programming and understand the role that it plays in the design of combinatorial algorithms. We will then go on to the study of NP-completeness and NP-hard problems, along with polynomial time approximation algorithms for these hard problems.

1.1 Amortized Analysis

Typically, most data structures provide absolute guarantees on the worst case time for performing a single operation. We will study data structures that are unable to guarantee a good bound on the worst case time *per* operation, but will guarantee a good bound on the *average* time it takes to perform an operation. (For example, a sequence of m operations will be guaranteed to take $m \times T$ time, giving an average, or *amortized time* of T per operation. A single operation could take time more than T.)

Example 1 (not covered in class): Consider a STACK with the following two operations: `Push(x)` pushes item x onto the stack, and `M-POP(k)` pop's the top-most k items from the stack (if they exist). Clearly, a single `M-POP` operation can take more than $O(1)$ time to execute, in fact the time is $\min(k, s)$ where s is the stack-size at that instant.

It should be evident, that a sequence of n operations however runs only in $O(n)$ time, yielding an "average" time of $O(1)$ per operation. (Each item that is pushed into the stack can be popped at most once.)

There are fairly simple formal *schemes* that formalize this very argument. The first one is called the **accounting method**. We shall now assume that our computer is like a vending machine. We can put in \$ 1 into the machine, and make it run for a *constant* number of steps (we can pick the constant). Each time we push an item onto the stack we use \$ 2 in doing this operation. We spend \$ 1 in performing the push operation, and the other \$ 1 is stored *with* the item on the stack. (This is only for analyzing the algorithm, the actual algorithm does not have to keep track of this money.) When we execute a multiple pop operation, the work done for each pop is paid for by the money stored with the item itself.

The second scheme is the **potential method**. We define the potential Φ for a data structure D. The potential maps the current "state" of the data structure to a real number, based on its current configuration.

In a sequence of operations, the data structure transforms itself from state D_{i-1} to D_i (starting at D_0). The *real cost* of this transition is c_i (for changing the data structure). The potential function satisfies the following properties:

- $\Phi(D_i) \geq 0$.
- $\Phi(D_0) = 0$

We define the *amortized cost* to be $c'_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$, where c_i is the true cost for the i^{th} operation.

Clearly,

$$\sum_{i=1}^{n} c'_i = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0).$$

Thus, if the potential function is always positive and $\Phi(D_0) = 0$, then the amortized cost is an upper bound on the real cost. Notice that even though the cost of each individual operation may not be constant, we may be able to show that the cost over any sequence of length n is $O(n)$. (In most applications, where data structures are used as a part of an algorithm; we need to use the data structure for over a sequence of operations and hence analyzing the data structure's performance over a sequence of operations is a very reasonable thing to do.)

In the stack example, we can define the potential to be the number of items on the stack. (Exercise: work out the amortized costs for each operation to see that Push has an amortized cost of 2, and M-Pop has an amortized cost of 1.)

Example 2: The second example we consider is a k-bit counter. We simply do INCREMENT operations on the k-bit counter, and wish to count the total number of bit operations that were performed over a sequence of n operations. Let the counter be $= < b_k b_{k-1} \ldots b_1 >$. Observe that the least significant bit b_1, changes in every step. Bit b_2 however, changes in every alternate step. Bit b_3 changes every 4^{th} step, and so on. Thus the total number of bit operations done are:

$$n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} \ldots \leq 2n.$$

A potential function that lets us prove an amortized cost of 2 per operation, is simply the number of 1's in the counter. Each time we have a cascading carry, notice that the number of 1's decrease. So the potential of the data structure falls and thus pays for the operation. (Exercise: Show that the amortized cost of an INCREMENT operation is 2.)

CMSC 651 Advanced Algorithms | Lecture 2
Lecturer: William Gasarch | Tu. Jan. 24, 1995

Original notes by Mark Carson.

2 Splay Trees

Read [21] Chapter 4.3 for Splay trees stuff.

Splay trees are a powerful data structure developed by Sleator and Tarjan [20], that function as search trees without any explicit balancing conditions. They serve as an excellent tool to demonstrate the power of amortized analysis.

Our basic operation is: $splay(k)$, given a key k. This involves two steps:

1. Search through out the tree to find node with key k.
2. Do a series of rotations (a splay) to bring it to the top.

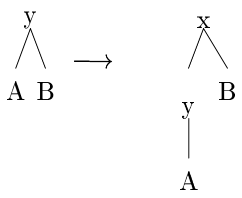
The first of these needs a slight clarification:

If k is not found, grab the largest node with key less than k instead (then splay this to the top.)

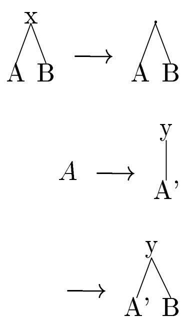
2.1 Use of Splay Operations

All tree operations can be simplified through the use of splay:

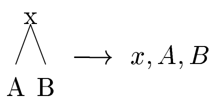
1. $Access(x)$ - Simply splay to bring it to the top, so it becomes the root.
2. $Insert(x)$ - Run $splay(x)$ on the tree to bring y, the largest element less than x, to the top. The insert is then trivial:



3. $Delete(x)$ - Run $splay(x)$ on the tree to bring x to the top. Then run $splay(x)$ again in x's left subtree A to bring y, the largest element less than x, to the top of A. y will have an empty right subtree in A since it is the largest element there. Then it is trivial to join the pieces together again without x:



4. *Split*(x) - Run *splay*(x) to bring x to the top and split.



Thus with at most 2 splay operations and constant additional work we can accomplish any desired operation.

2.2 Time for a Splay Operation

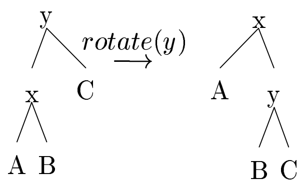
How much work does a splay operation require? We must:

1. Find the item (time dependent on depth of item).
2. Splay it to the top (time again dependent on depth)

Hence, the total time is $O(2\times$ depth of item$)$.

How much time do k splay operations require? The answer will turn out to be $O(k \log n)$, where n is the size of the tree. Hence, the amortized time for one splay operation is $O(\log n)$.

The basic step in a splay operation is a *rotation*:



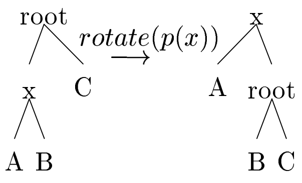
Clearly a rotation can be done in $O(1)$ time. (Note there are both left and right rotations, which are in fact inverses of each other. The sketch above depicts a right rotation going forward and a left rotation going backward. Hence to bring up a node, we do a right rotation if it is a left child, and a left rotation if it is a right child.)

A splay is then done with a (carefully-selected) series of rotations. Let $p(x)$ be the parent of a node x. Here is the splay algorithm:

Splay Algorithm:

while $x \neq root$ **do**

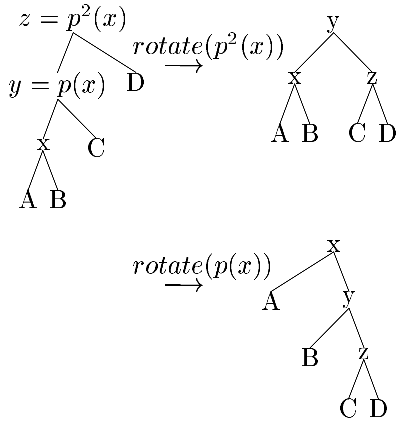
  if $p(x) = root$ **then** $rotate(p(x))$



  else if both x and $p(x)$ are left (resp. right) children, **do** right (resp. left) rotations: **begin**

    $rotate(p^2(x))$
    $rotate(p(x))$

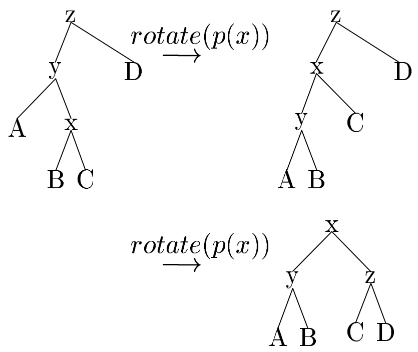
  end



else /* x and $p(x)$ are left/right or right/left children */ **begin**

$rotate(p(x))$
$rotate(p(x))$ /* note this is a new $p(x)$ */

end



fi

od

When will accesses take a long time? When the tree is long and skinny.
What produces long skinny trees?

- a series of inserts in ascending order 1, 3, 5, 7, ...
- each insert will take $O(1)$ steps – the splay will be a no-op.
- then an operation like $access(1)$ will take $O(n)$ steps.
- *HOWEVER* this will then result in the tree being balanced.
- Also note that the first few operations were very fast.

Therefore, we have this general idea – splay operations tend to balance the tree. Thus any long access times are "balanced" (so to speak) by the fact the tree ends up better balanced, speeding subsequent accesses.

In potential terms, the idea is that as a tree is built high, its "potential energy" increases. Accessing a deep item releases the potential as the tree sinks down, paying for the extra work required.

CMSC 651 Advanced Algorithms	Lecture 3
Lecturer: Samir Khuller	Th. Jan. 26, 1995

Original notes by Mark Carson.

3 Amortized Time for Splay Trees

Read [21] Chapter 4.3 for Splay trees stuff.

Theorem 3.1 *The amortized time of a splay operation is* $O(\log n)$.

To prove this, we need to define an appropriate potential function.

Definition 3.2 *Let* s *be the splay tree. Let* $d(x)$ = *the number of descendants of* x *(including* x*). Define the rank of* $x, r(x) = \log d(x)$ *and the potential function*

$$\Phi(s) = \sum_{x \epsilon s} r(x).$$

Thus we have:

- $d(\text{leaf node}) = 1$, $d(\text{root}) = n$
- $r(\text{leaf node}) = 0$, $r(\text{root}) = \log n$

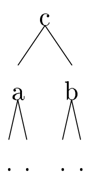
Clearly, the better balanced the tree is, the lower the potential Φ is. (You may want to work out the potential of various trees to convince yourself of this.)

We will need the following lemmas to bound changes in Φ.

Lemma 3.3 *Let* c *be a node in a tree, with* a *and* b *its children. Then* $r(c) > 1 + min(r(a), r(b))$.

Proof:

Looking at the tree, we see $d(c) = d(a) + d(b) + 1$. Thus we have $r(c) > 1 + min(r(a), r(b))$.



□

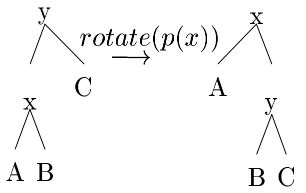
We apply this to

Lemma 3.4 (Main Lemma) *Let* $r(x)$ *be the rank of* x *before a rotation (a single splay step) bringing* x *up, and* $r'(x)$ *be its rank afterward. Similarly, let* s *denote the tree before the rotation and* s' *afterward. Then we have:*

1. $r'(x) \geq r(x)$
2. *If* $p(x)$ *is the root then* $\Phi(s') - \Phi(s) < r'(x) - r(x)$
3. *if* $p(x) \neq root$ *then* $\Phi(s') - \Phi(s) < 3(r'(x) - r(x)) - 1$

Proof:

1. Obvious as x gains descendants.
2. Note in this case we have

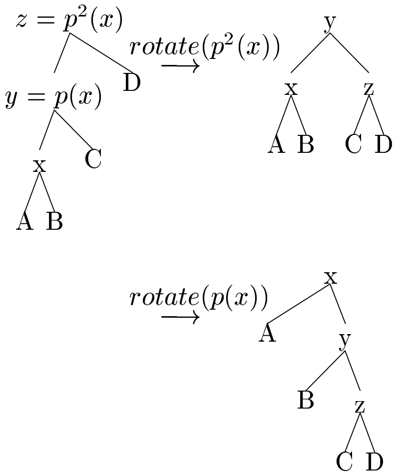


so that clearly $r'(x) = r(y)$. But then since only x and y change rank in s',

$$\Phi(s') - \Phi(s) \begin{array}{l} = (r'(x) - r(x)) + (r'(y) - r(y)) \\ = r'(y) - r(x) < r'(x) - r(x) \end{array}$$

since clearly $r'(y) < r'(x)$.

3. Consider just the following case (the others are similar):



Let r represent the ranks in the initial tree s, r'' ranks in the middle tree s'' and r' ranks in the final tree s'. Note that, looking at the initial and final trees, we have

$$r(x) < r(y)$$

and

$$r'(y) < r'(x)$$

so

$$r'(y) - r(y) < r'(x) - r(x)$$

Hence, since only x, y and z change rank,

$$\begin{aligned} \Phi(s') - \Phi(s) &= (r'(x) - r(x)) + (r'(y) - r(y)) + (r'(z) - r(z)) \\ &< 2(r'(x) - r(x)) + (r'(z) - r(z))(*) \end{aligned}$$

Next from Lemma 1, we have $r''(y) > 1 + min(r''(x), r''(z))$. But looking at the middle tree, we have

$$\begin{array}{l} r''(x) = r(x) \\ r''(y) = r'(x)(= r(z)) \\ r''(z) = r'(z) \end{array}$$

so that

$$r(z) = r'(x) > 1 + min(r(x), r'(z))$$

Hence, either we have

$$r'(x) > 1 + r(x), \quad so \;\; r'(x) - r(x) > 1$$

or

$$r(z)) > 1 + r'(z), \quad so \;\; r'(z) - r(z) < -1$$

In the first case, since

$$r'(z) < r(z) => r'(z) - r(z) < 0 < r'(x) - r(x) - 1$$

clearly

$$\Phi(s') - \Phi(s) < 3(r'(x) - r(x)) - 1$$

In the second case, since we always have $r'(x) - r(x) > 0$, we again get

$$\begin{aligned} \Phi(s') - \Phi(s) &< 2(r'(x) - r(x)) - 1 \\ &< 3(r'(x) - r(x)) - 1 \end{aligned}$$

□

We will apply this lemma now to determine the amortized cost of a splay operation. The splay operation consists of a series of splay steps (rotations). For each splay step, if s is the tree before the splay, and s' the tree afterwards, we have

$$at = rt + \Phi(s') - \Phi(s)$$

where at is the amortized time, rt the real time. In this case, $rt = 1$, since a splay step can be done in constant time.
Note: Here we have scaled the time factor to say a splay step takes one time unit. If instead we say a splay takes c time units for some constant c, we change the potential function Φ to be

$$\Phi(s) = c \sum_{x \epsilon s} r(x).$$

Consider now the two cases in Lemma 2. For the first case (x a child of the root), we have

$$\begin{aligned} at = 1 + \Phi(s') - \Phi(s) &< 1 + (r'(x) - r(x)) \\ &< 3\Delta r + 1 \end{aligned}$$

For the second case, we have

$$\begin{aligned} at = 1 + \Phi(s') - \Phi(s) &< 1 + 3(r'(x) - r(x)) - 1 \\ &= 3(r'(x) - r(x)) \\ &= 3\Delta r \end{aligned}$$

Then for the whole splay operation, let $s = s_0, s_1, s_2, \ldots, s_k$ be the series of trees produced by the sequence of splay steps, and $r = r_0, r_1, r_2, \ldots, r_k$ the corresponding rank functions. Then the total amortized cost is

$$at = \sum_{i=0}^{k} at_i < 1 + \sum_{i=0}^{k} 3\Delta r_i$$

But the latter series telescopes, so

$$at < 1 + 3(\; final \; rank \; of \; x - \; initial \; rank \; of \; x)$$

Since the final rank of x is $\log n$, we then have

$$at < 3 \log n + 1$$

as desired.

3.1 Additional notes

1. (Exercise) The accounting view is that each node x stores $r(x)$ dollars [or some constant multiple thereof]. When rotates occur, money is taken from those nodes which lose rank to pay for the operation.

2. The total time for k operations is actually $O(k \log m)$, where m is the largest size the tree ever attains (assuming it grows and shrinks through a series of inserts and deletes).

3. As mentioned above, if we say the real time for a rotation is c, the potential function Φ is

$$\Phi(s) = \sum_{x \epsilon s} cr(x)$$

CMSC 651 Advanced Algorithms	Lecture 2
Lecturer: Samir Khuller	Th. Jan 29, 1998

Original notes by Matos Gilberto and Patchanee Ittarat.

4 Maintaining Disjoint Sets

Read Chapter 22 for Disjoint Sets Data Structure and Chapter 24 for Kruskal's Minimum Spanning Tree Algorithm [5].

I assume everyone is familiar with Kruskal's MST algorithm. This algorithm is the nicest one to motivate the study of the disjoint set data structure. There are other applications for the data structure as well. We will not go into the details behind the implementation of the data structure, since the book gives a very nice description of the UNION, MAKESET and FIND operations.

4.1 Disjoint set operations:

- Makeset(x) : $A \leftarrow Makeset(x) \equiv A = \{x\}$.
- Find(y) : given y, find to which set y belongs.
- Union(A, B) : $C \leftarrow Union(A, B) \equiv C = A \cup B$.

Observe that the Union operation can be specified either by two sets or by two elements (in the latter case, we simply perform a union of the sets the elements belong to).

The name of a set is given by the element stored at the root of the set (one can use some other naming convention too). This scheme works since the sets are disjoint.

4.2 Data structure:

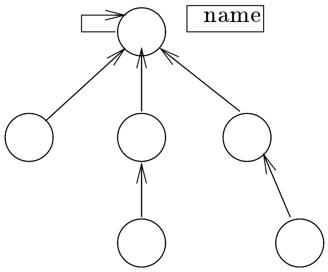


Figure 1: Data Structure

We will maintain the sets as rooted trees. Performing the UNION operation is done by linking one tree onto another. To perform a FIND, we traverse the path from the element to the root.

To improve total time we always hook the shallower tree to the deeper tree (this will be done by keeping track of ranks and not the exact depth of the tree).

Ques: Why will these improve the running time?

Ans: Since the amount of work depends on the height of a tree.

We use the concept of the "rank" of a node. The rank of a vertex denotes an upper bound on the depth of the sub-tree rooted at that node.

$rank(x) = 0$ for $\{x\}$

Union(A,B) - see Fig. 2.

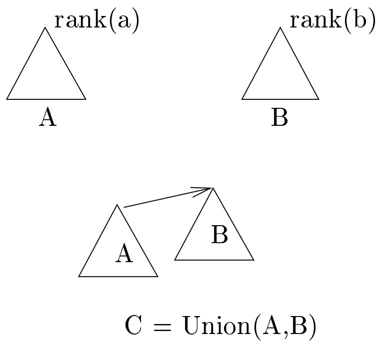


Figure 2: Union

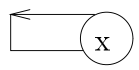


Figure 3: Rank 0 node

If $rank(a) < rank(b)$, $rank(c) = rank(b)$.
If $rank(a) = rank(b)$, $rank(c) = rank(b) + 1$.

4.3 Union by rank

Union by rank guarantees "at most $O(\log n)$ of depth".

Lemma 4.1 *A node with rank k has at least 2^k descendants.*

Proof:
[By induction on size of tree]
$rank(x) = 0 \Rightarrow x$ has one descendant (itself).
The rank and the number of descendants of any node are changed only by the Union operation, so let's consider a Union operation in Fig. 2.

Case 1 *$rank(a) < rank(b)$:*

$$\begin{aligned}
rank(c) &= rank(b) \\
desc(c) &\geq desc(b) \\
&\geq 2^{rank(b)}
\end{aligned}$$

Case 2 *$rank(a) = rank(b)$:*

$$\begin{aligned}
rank(c) &= rank(b) + 1 \\
desc(a) &\geq 2^{rank(a)} \quad and \\
desc(b) &\geq 2^{rank(b)} \\
desc(c) &= node(a) + node(b) \\
&\geq 2^{rank(a)} + 2^{rank(b)} \\
&\geq 2^{rank(b)+1}
\end{aligned}$$

$\square$

In the Find operation, we can make a tree shallower by path compression. During path compression, we take all the nodes on the path to the root and make them all point to the root (to speed up future find operations).

The UNION operation is done by hooking the smaller rank vertex to the higher rank vertex, and the FIND operation performs the path compression while we traverse the path from the element to the root. Path compression takes two passes – first to find the root, and second time to change the pointers of all nodes on the path to point to the root. After the find operation all the nodes point directly to the root of the tree.

4.4 Upper Bounds on the Disjoint-Set Union Operations

Simply recall that each vertex has a rank (initially the rank of each vertex is 0) and this rank is incremented by 1 whenever we perform a union of two sets that have the same rank. We only worry about the time for the FIND operation (since the UNION operation takes constant time). The parent of a vertex always has higher rank than the vertex itself. This is true for all nodes except for the root (which is its own parent).

The following theorem gives two bounds for the total time taken to perform m find operations. (A union takes constant time.)

Theorem 4.2 *m operations (including makeset, find and union) take total time $O(m \log^* n)$ or $O(m + n \log n)$, where $\log^* n = \{min(i) | \log^{(i)} n \leq 1\}$.*

$$\begin{aligned} \log^* 16 &= 3, \ \textit{and} \\ \log^* 2^{16} &= 4 \end{aligned}$$

To prove this, we need some observations:

1. Rank of a node starts at 0 and goes up as long as the node is a root. Once a node becomes a non-root the rank does not change.
2. $Rank(p(x))$ is non-decreasing. In fact, each time the parent changes the rank of the parent must increase.
3. $Rank(p(x)) > rank(x)$.

The rank of any vertex is lesser than or equal $\log_2 n$, where n is the number of elements.

First lets see the $O(m + n \log n)$ bound (its easier to prove). This bound is clearly not great when $m = O(n)$.

The cost of a single find is charged to 2 accounts

1. The find pays for the cost of the root and its child.
2. A bill is given to every node whose parent changes (in other words, all other nodes on the find path).

Note that every node that has been issued a bill in this operation becomes the child of the root, and won't be issued any more bills until its parent becomes a child of some other root in a union operation. Note also that one node can be issued a bill at most $\log_2 n$ times, because every time a node gets a bill its parent's rank goes up, and rank is bounded by $\log_2 n$. The sum of bills issued to all nodes will be upper bounded by $n \log_2 n$

We will refine this billing policy shortly.

Lemma 4.3 *There are at most $\frac{n}{2^r}$ nodes of rank r.*

Proof:

When the node gets rank r it has $\geq 2^r$ descendants, and it is the root of some tree. After some union operations this node may no longer be root, and may start to lose descendants, but its rank will not change. Assume that every descendant of a root node gets a timestamp of r when the root first increases its rank to r. Once a vertex gets stamped, it will never be stamped again since its new roots will have rank strictly more than r. Thus for every node of rank r there are at least 2^r nodes with a timestamp of r. Since there are n nodes in all, we can never create more than $\frac{n}{2^r}$ vertices with rank r. □

We introduce the fast growing function F which is defined as follows.

1. $F(0) = 1$
2. $F(i) = 2^{F(i-1)}$

4.5 Concept of Blocks

If a node has rank r, it belongs to block $B(\log^* r)$.

- $B(0)$ contains nodes of rank 0 and 1.
- $B(1)$ contains nodes of rank 2.
- $B(2)$ contains nodes of rank 3 and 4.
- $B(3)$ contains nodes of rank 5 through 16.
- $B(4)$ contains nodes of rank 17 through 65536.

Since the rank of a node is at most $\log_2 n$ where n is the number of elements in the set, the number of blocks necessary to put all the elements of the sets is bounded by $\log^*(\log n)$ which is $\log^* n - 1$. So blocks from $B(0)$ to $B(\log^* n - 1)$ will be used.

The find operation goes the same way as before, but the billing policy is different. The find operation pays for the work done for the root and its immediate child, and it also pays for all the nodes which are not in the same block as their parents. All of these nodes are children of some other nodes, so their ranks will not change and they are bound to stay in the same block until the end of computation. If a node is in the same block as its parent it will be billed for the work done in the find operation. As before find operation pays for the work done on the root and its child. Number of nodes whose parents are in different blocks is limited to $\log^* n - 1$, so the cost of the find operation is upper bounded by $\log^* n - 1 + 2$.

After the first time a node is in the different block from its parent, it is always going to be the case since the rank of the parent only increases. This means that the find operation is going to pay for the work on that node every time. So any node will be billed for the find operations a certain number of times, and after that all subsequent finds will pay for their work on the element. We need to find an upper bound for the number of times a node is going to be billed for the find operation.

Consider the block with index i; it contains nodes with the rank in the interval from $F(i-1)+1$ to $F(i)$. The number of nodes in this block is upper bounded by the possible number of nodes in each of the ranks. There are at most $n/2^r$ nodes of rank r, so this is a sum of a geometric series, whose value is

$$\sum_{r=F(i-1)+1}^{F(i)} \frac{n}{2^r} \leq \frac{n}{2^{F(i-1)}} = \frac{n}{F(i)}$$

Notice that this is the only place where we make use of the exact definition of function F.

After every find operation a node changes to a parent with a higher rank, and since there are only $F(i) - F(i-1)$ different ranks in the block, this bounds the number of bills a node can ever get. Since the block $B(i)$ contains at most $n/F(i)$ nodes, all the nodes in $B(i)$ can be billed at most n times (its the product of the number of nodes in $B(i)$ and the upper bound on the bills a single node may get). Since there are at most $\log^* n$ blocks the total cost for these find operations is bounded by $n \log^* n$.

This is still not the tight bound on the number of operations, because Tarjan has proved that there is an even tighter bound which is proportional to the $O(m\alpha(m,n))$ for m union, makeset and and find operations, where α is the inverse ackerman function whose value is lower than 4 for all practical applications. This is quite difficult to prove (see Tarjan's book). There is also a corresponding tight lower bound on *any* implementation of the disjoint set data structure on the pointer machine model.

CMSC 651 Advanced Algorithms Lecturer: Samir Khuller	Lecture 2

Original notes by King-Ip Lin.

5 Binomial heaps

These are heaps that also provide the power to merge two heaps into one.

Definition 5.1 (Binomial tree) *A* binomial tree of height k *(denoted as B_k) is defined recursively as follows:*

1. *B_0 is a single node*
2. *B_{i+1} is formed by joining two B_i heaps, making one's root the child of the other*

Basic properties of B_k

- Number of nodes : 2^k
- Height : k
- Number of children of root (degree of root) : k

In order to store n nodes in binomial tress when $n \neq 2^k$, write n in binary notation; for every bit that is "1" in the representation, create a corresponding B_k tree, treating the rightmost bit as 0.

Example : $n = 13 = 1101_2 \longrightarrow$ use B_3, B_2, B_0

Definition 5.2 (Binomial heap) *A* binomial heap *is a (set of) binomial tree(s) where each node has a key and the heap-order property is preserved. We also have the requirement that for any given i there is at most one B_i.*

Algorithms for the binomial heap :

Find minimum Given n, there will be $\log n$ binomial trees and each tree's minimum will be its root. Thus only need to find the minimum among the roots.

$Time = \log n$

Insertion Invariant to be maintained : only 1 B_i tree for each i

Step 1: create a new B_0 tree for the new element

Step 2: $i \leftarrow 0$

Step 3: **while** there are still two B_i trees do
  join the two B_i trees to form a B_{i+1} tree
  $i \leftarrow i + 1$

Clearly this takes at most $O(\log n)$ time.

Deletion

Delete min Key observation: Removing the root from a B_i tree will form i binomial trees, from B_0 to B_{i-1}

Step 1: Find the minimum root (assume its B_k)

Step 2: Break B_k, forming k smaller trees

Step 3: **while** there are still at least two B_i trees do
join the two B_i trees to form a B_{i+1} tree
$i \leftarrow i + 1$

Note that at each stage there will be at most three B_i trees, thus for each i only one join is required.

Delete

Step 1: Find the element

Step 2: Change the element key to $-\infty$

Step 3: Push the key up the tree to maintain the heap-order property

Step 4: Call Delete min

Steps 2 and 3 are grouped and called DECREASE_KEY(x)
Time for insertion and deletion : $O(\log n)$

5.1 Fibonacci Heaps (F-Heaps)

Amortized running time :

- Insert, Findmin, Union, Decrease-key : $O(1)$
- Delete-min, Delete : $O(\log n)$

Added features (compared to the binomial heaps)

- Individual trees are not necessary binomial (denote trees by B'_i)
- Always maintain a pointer to the smallest root
- permit many copies of B'_i

Algorithms for F-heaps:

Insert

Step 1: Create a new B'_0

Step 2: Compare with the current minimum and update pointer if necessary

Step 3: Store \$1 at the new root

(Notice that the \$ is only for accounting purposes, and the implementation of the data structure does not need to keep track of the \$'s.)

Delete-min

Step 1: Remove the minimum, breaking that tree into smaller tree again

Step 2: Find the new minimum, merge trees in the process, resulting in at most one B'_i tree for each i

Decrease-key

Step 1: Decrease the key

Step 2: **if** heap-order is violated
break the link between the node and its parent (note: resulting tree may not be a true binomial tree)

Step 3: Compare the new root with the current minimum and update pointer if necessary

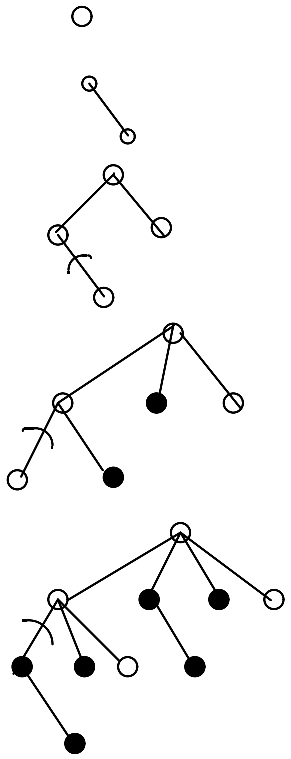
Step 4: Put \$1 to the new root

Step 5: Pay \$1 for the cut

Problem with the above algorithm: Can result in trees where the root has a very large number of children.
Solution:

- whenever a node is being cut
  - mark the parent of the cut node in the original tree
  - put \$2 on that node
- when a second child of that node is lost (by that time that node will have \$4), recursively cut that node from its parent, use the \$4 to pay for it:
  - \$1 for the cut
  - \$1 for new root
  - \$2 to its original parent
- repeat the recursion upward if necessary

Thus each cut requires only \$4.
Thus decrease-key takes amortized time $O(1)$
Define rank of the tree = Number of children of the root of the tree.
Consider the minimum number of nodes of a tree with a given rank:



	Rank	Worst case size	Size of binomial tree
B0	0	1	1
B1	1	2	2
B2	2	3	4
B3	3	5	8
B4	4	8	16

Marked node from previous deletion

Figure 4: Minimum size B_i' trees

Original notes by Patchanee Ittarat.

6 F-heap

6.1 Properties

F-heaps have the following properties:

- maintain the minimum key in the heap all the time,
- relax the condition that we have at most one B_k tree for any k,i.e., we can have any number of B_k trees co-existing.
- trees are not true binomial trees.

For example,

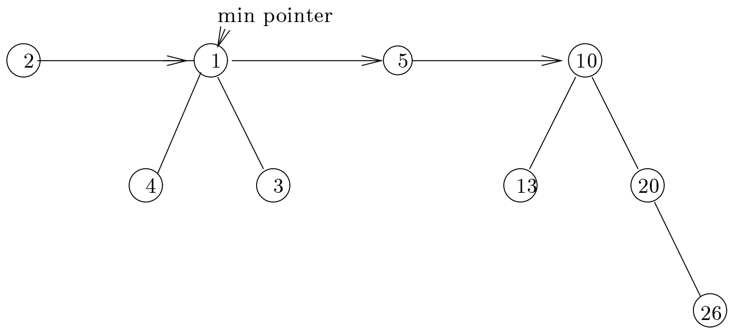


Figure 5: Example of F-heap

Property: size of a tree, rooted at x, is in exponential in the degree of x.

In Binomial trees we have the property that $size(x) \geq 2^k$, here we will not have as strong a property, but we will have the following:

$$size(x) \geq \phi^k$$

where k is degree of x and $\phi = \frac{1+\sqrt{5}}{2}$.

Note that since $\phi > 1$ this is sufficient to guarantee a $O(\log n)$ bound on the depth and the number of children of a node.

6.2 Decrease-Key operation

Mark strategy:

- when a node is cut off from its parent, tick one mark to its parent,
- when a node gets 2 marks, cut the edge to its parent and tick its parent as well,
- when a node becomes a root, erase all marks attached to that node,
- every time a node is cut, give \$1 to that node as a new root and \$2 to its parent.

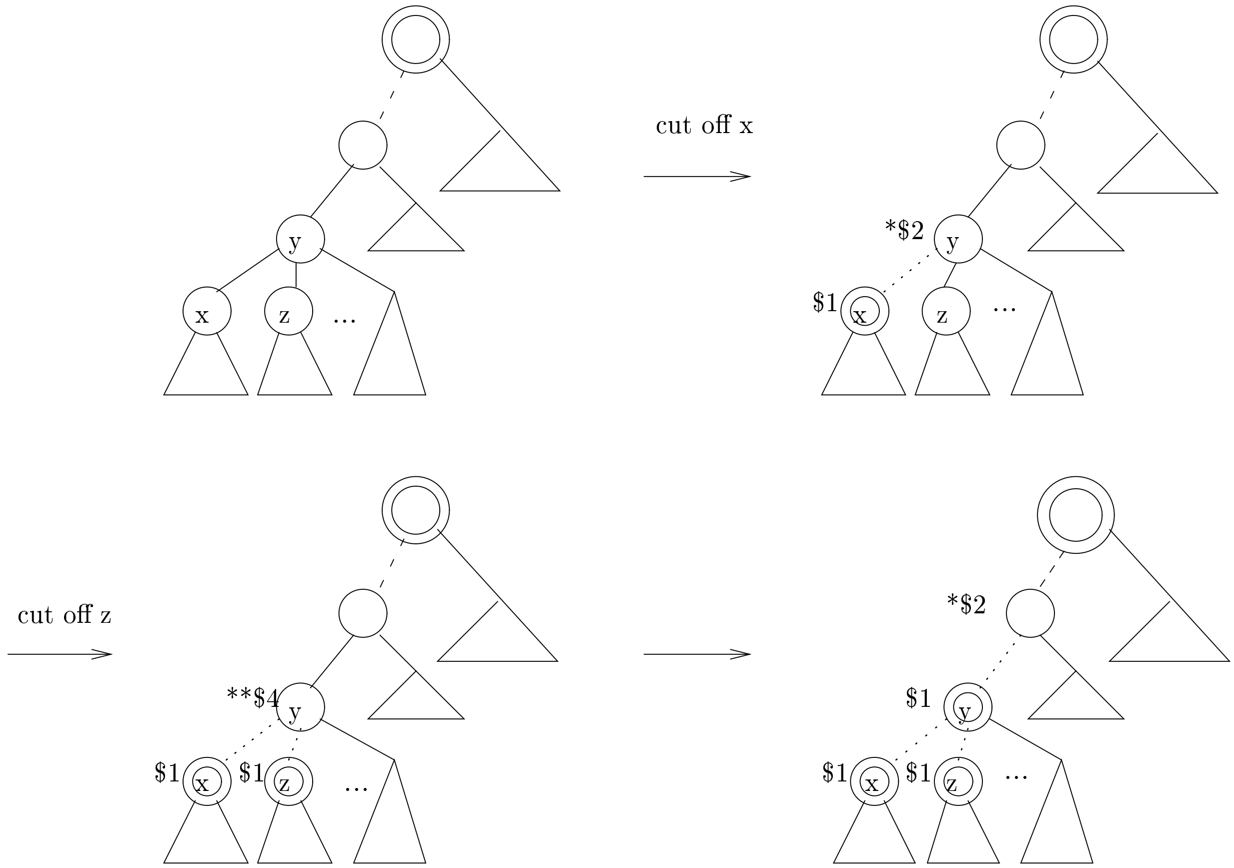


Figure 6: Marking Strategy

The cost of Decrease-Key operation = cost of cutting link + \$3. We can see that no extra dollars are needed when the parent is cut off recursively since when the cut off is done, that parent must have \$4 in hand (\$2 from each cut child and there must be 2 children have been cut), then we use those \$4 dollars to pay for cutting link cost (\$1), giving to its parent (\$2), and for itself as a new root (\$1).

Example: How does Decrease-Key work (see CLR also)?

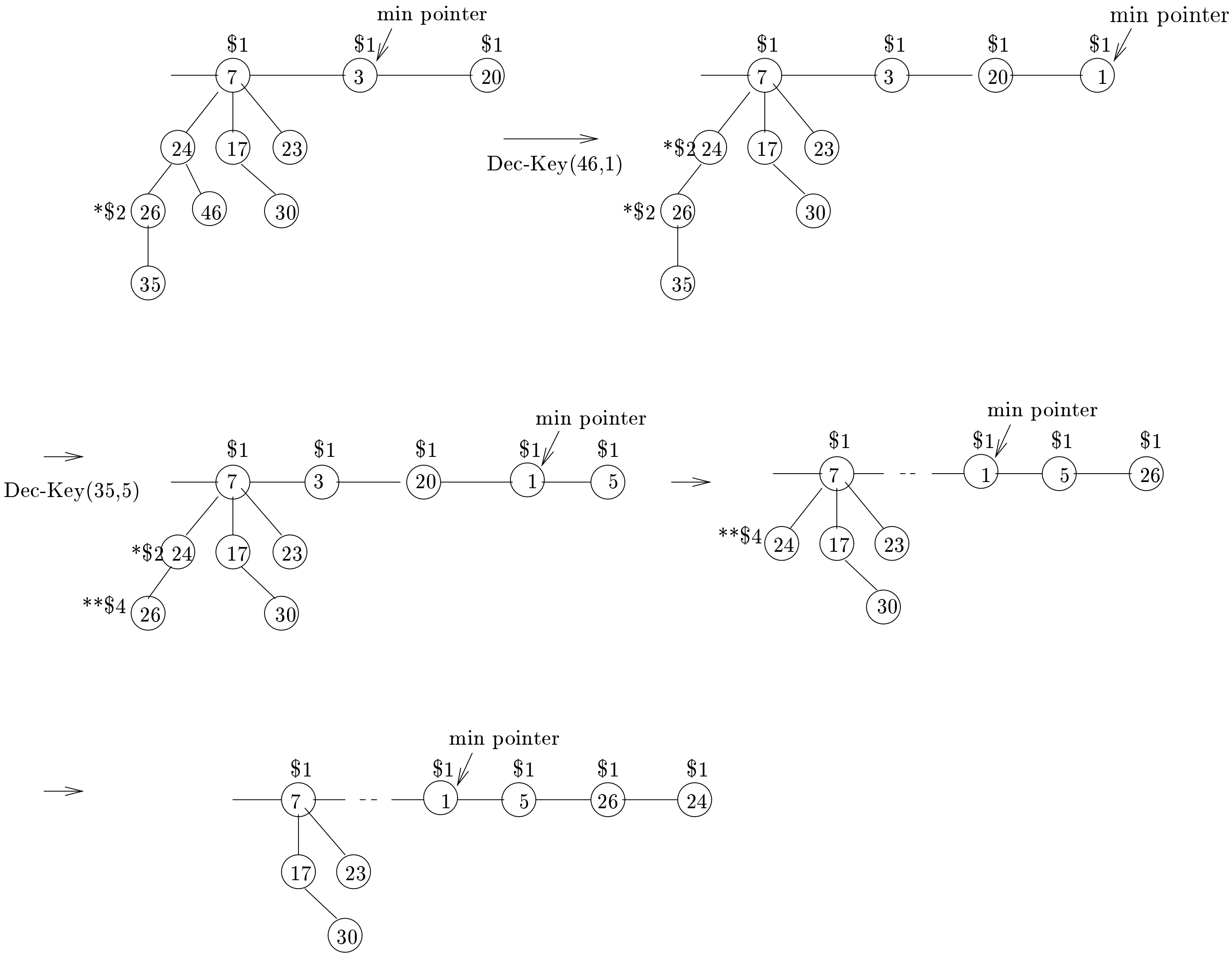


Figure 7: Example

The property we need when two trees are merged, is the degree of the roots of both trees should be the same.

Let y_1 be the oldest child and y_k be the youngest child. Consider y_i, at the point y_i was made a child of the root x, x had degree at least $i-1$ and so does y_i. Since y_i has lost at most 1 child, now y_i has at least degree $i-2$.

We now study some properties about Fibonacci numbers, and see how they relate to the sizes of the trees with the decrease-key operation. Define

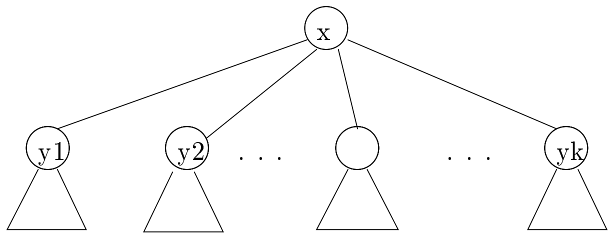


Figure 8: Example

$$F_k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2 \end{cases}$$

These numbers are called Fibonacci numbers.

Property 6.1 $F_{k+2} = 1 + \sum_{i=1}^{k} F_i$

Proof:
[By induction]

$$\begin{array}{lrcl} k = 1: & F_3 & = & 1 + F1 \\ & & = & 1 + 1 \\ & & = & F_1 + F_2 \quad (\text{where } F_2 = F_1 + F_0 = 1 + 0) \end{array}$$

Assume $F_{k+2} = 1 + \sum_{i=1}^{k} F_i$ for all $k \leq n$

$$\begin{array}{lrcl} k = n+1: & F_{n+3} & = & 1 + \sum_{i=1}^{n+1} F_i \\ & & = & 1 + \sum_{i=1}^{n} F_i + F_{n+1} \\ & & = & F_{n+2} + F_{n+1} \end{array}$$

□

Property 6.2 $F_{k+2} \geq \phi^k$

Proof:
[By induction]

$$\begin{array}{lrcl} k = 0: & F_2 & = & 1 \\ & & \geq & \phi^0 \\ k = 1: & F_3 & = & 2 \\ & & \geq & \phi = \frac{1+\sqrt{5}}{2} = 1.618.. \end{array}$$

Assume $F_{k+2} \geq \phi^k$ for all $k < n$

$$
\begin{array}{lrcl}
k = n: & F_{n+2} & = & F_{n+1} + F_n \\
 & & \geq & \phi^{n-1} + \phi^{n-2} \\
 & & \geq & \dfrac{1+\phi}{\phi^2} \cdot \phi^n \\
 & & \geq & \phi^n
\end{array}
$$

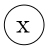
□

Theorem 6.1 *x is a root of any subtree. $size(x) \geq F_{k+2} \geq \phi^k$ where k is a degree of x*

Proof:

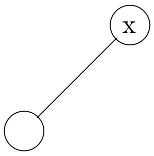
[By induction]

$k = 0$:



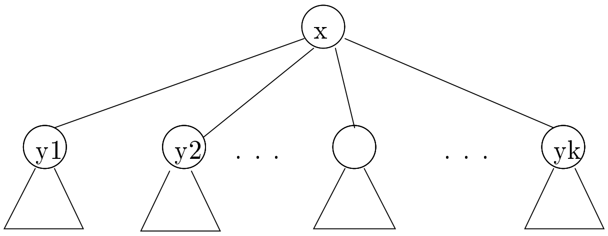
$size(x) = 1 \geq 1 \geq 1$

$k = 1$:



$size(x) = 2 \geq 2 \geq \frac{1+\sqrt{5}}{2}$

Assume $size(x) \geq F_{k+2} \geq \phi^k$ for any $k \leq n$



$k = n + 1$:

$$
\begin{array}{rcl}
size(x) & = & 1 + size(y_1) + ... + size(y_i) + ... + size(y_k) \\
 & \geq & 1 + 1 + 1 + F_3 + ... + F_k \ \ (from \ \ assumption) \\
 & \geq & 1 + F_1 + F_2 + ... + F_k \\
 & \geq & F_{k+2} \ \ (from \ \ property1) \\
 & \geq & \phi^k \ \ (from \ \ property2) \\
\log_\phi size(x) & \geq & k
\end{array}
$$

So the number of children of node x is bounded by $\log_\phi size(x)$. □

CMSC 651 Advanced Algorithms	Lecture 4
Lecturer: Samir Khuller	Th. Feb. 5, 1998

Notes by Samir Khuller.

7 Minimum Spanning Trees

Read Chapter 6 from [21]. Yao's algorithm is from [23].

Given a graph $G = (V, E)$ and a weight function $w : E \rightarrow R^+$ we wish to find a spanning tree $T \subseteq E$ such that its total weight $\sum_{e \in T} w(e)$ is minimized. We call the problem of determining the tree T the minimum-spanning tree problem and the tree itself an MST. We can assume that all edge weights are distinct (this just makes the proofs easier). To enforce the assumption, we can number all the edges and use the edge numbers to break ties between edges of the same weight.

We will present now two approaches for finding an MST. The first is Prim's method and the second is Yao's method (actually a refinement of Boruvka's algorithm). To understand why all these algorithms work, one should read Tarjan's *red-blue* rules. A red edge is essentially an edge that is not in an MST, a blue edge is an edge that is in an MST. A *cut* (X, Y) for $X, Y \subset V$ is simply the set of edges between vertices in the sets X and Y.

Red rule: consider any cycle that has no red edges on it. Take the highest weight uncolored edge and color it red.

Blue rule: consider any cut $(S, V - S)$ where $S \subset V$ that has no blue edges. Pick the lowest weight edge and color it blue.

All the MST algorithms can be viewed as algorithms that apply the red-blue rules in some order.

7.1 Prim's algorithm

Prim's algorithm operates much like Dijkstra's algorithm. The tree starts from an arbitrary vertex v and grows until the tree spans all vertices in V. At each step our currently connected set is S. Initially $S = \{v\}$. A lightest edge connecting a vertex in S with a vertex in $V - S$ is added to the tree. Correctness of this algorithm follows from the observation that a partition of vertices into S and $V - S$ defines a cut, and the algorithm always chooses the lightest edge crossing the cut. The key to implementing Prim's algorithm efficiently is to make it easy to select a new edge to be added to the tree formed by edges in MST. Using a Fibonacci heap we can perform EXTRACT-MIN and DELETE-MIN operation in $O(\log n)$ amortized time and DECREASE-KEY in $O(1)$ amortized time. Thus, the running time is $O(m + n \log n)$.

It turns out that for sparse graphs we can do even better! We will study Yao's algorithm (based on Boruvka's method). There is another method by Fredman and Tarjan that uses F-heaps. However, this is not the best algorithm. Using an idea known as "packeting" this The FT algorithm was improved by Gabow-Galil-Spencer-Tarjan (also uses F-heaps). More recently (in Fall 1993), Klein and Tarjan announced a linear time randomized MST algorithm based on a linear time verification method (i.e., given a tree T and a graph G can we verify that T is an MST of G?).

7.2 Yao/Boruvka's algorithm

We first outline Boruvka's method. Boruvka's algorithm starts with a collection of singleton vertex sets. We put all the singleton sets in a queue. We pick the first vertex v, from the head of the queue and this vertex selects a lowest weight edge incident to it and marks this edge as an edge to be added to the MST. We continue doing this until all the vertices in the queue, are processed. At the end of this round we contract the marked edges (essentially merge all the connected components of the marked edges into single nodes – you should think about how this is done). Notice that no cycles are created by the marked edges if we assume that all edge weights are distinct.

Each connected component has size *at least* two (perhaps more). (If v merges with u, and then w merges with v, we get a component of size three.) The entire processing of a queue is called a phase. So at the end of phase i, we know that each component has at least 2^i vertices. This lets us bound the number of phases by $\log n$. (Can be proved by induction.) This gives a running time of $O(m \log n)$ since there are at most $\log n$ phases. Each phase can be implemented in $O(m)$ time if each node marks the cheapest edge incident to it, and then we contract all these edges and reconstruct an adjacency list representation for the new "shrunken" graph where a connected component of marked edges is viewed as a vertex. Edges in this graph are edges that connect vertices in two different components. Edges between nodes in the same component become self-loops and are discarded.

We now describe Yao's algorithm. This algorithm will maintain connected components and will not explicitly contract edges. We can use the UNION-FIND structure for keeping track of the connected components.

The main bottleneck in Boruvka's algorithm is that in a subsequent phase we have to recompute (from scratch) the lowest weight edge incident to a vertex. This forces us to spend time proportional to $\sum_{v \in T_i} d(v)$ for each tree T_i. Yao's idea was to "somehow order" the adjacency list of a vertex to save this computational overhead. This is achieved by partitioning the adjacency list of each vertex v into k groups $E_v^1, E_v^2, \ldots, E_v^k$ with the property that if $e \in E_v^i$ and $e' \in E_v^j$ and $i < j$, then $w(e) \leq w(e')$. For a vertex with degree $d(v)$, this takes $O(d(v) \log k)$ time. (We run the median finding algorithm, and use the median to partition the set into two. Recurse on each portion to break the sets, and obtain four sets by a second application of the median algorithm, each of size $\frac{d(v)}{4}$. We continue this process until we obtain k sets.) To perform this for all the adjacency lists takes $O(m \log k)$ time.

Let T be a set of edges in the MST. VS is a collection of vertex sets that form connected components. We may assume that VS is implemented as a doubly linked list, where each item is a set of vertices. Moreover, the root of each set contains a pointer to the position of the set in the doubly linked list. (This enables the following operation: given a vertex u do a FIND operation to determine the set it belongs to, and then yank the set out of the queue.)

The root of a set also contains a list of all items in the set (it is easy to modify the UNION operation to maintain this invariant.) $E(v)$ is the set of edges incident on vertex v. $\ell[v]$ denotes the current group of edges being scanned for vertex v. small$[v]$ computes the weight of the lowest weight edge incident on v that leaves the set W. If after scanning group $E_v^{\ell[v]}$ we do not find an edge leaving W, we scan the next group.

```
proc Yao-MST(G);
    T, VS ← ∅;
    ∀v : ℓ[v] ← 1;
    while |VS| > 1 do
        Let W be the first set in the queue VS;
        For each v ∈ W do;
            small[v] ← ∞;
            while small[v] = ∞ and ℓ[v] ≤ k do
                For each edge e = (v, v') in E_v^ℓ[v] do
                    If find(v') = find(v) then delete e from E_v^ℓ[v]
                     else small[v] ← min (small[v], w(e))
                If small[v] = ∞ then ℓ[v] ← ℓ[v] + 1
            end-while
        end-for
        Let the lowest weight edge out of W be (w, w') where w ∈ W and w' ∈ W';
        Delete W and W' from VS and add union(W, W') to VS;
        Add (w, w') to T
    end-while
end proc;
```

Each time we scan an edge to a vertex in the same set W, we delete the edge and charge the edge the cost for the find operation. The first group that we find containing edges going out of W, are the edges we pay for (at most $\frac{d(v)}{k} \log^* n$). In the next phase we start scanning at the point that we stopped last time. The overall cost of scanning in one phase is $O(\frac{m}{k} \log^* n)$. But we have $\log n$ phases in all, so the total running time amounts to $O(\frac{m}{k} \log^* n \log n) + O(m \log k)$ (for the preprocessing). If we pick $k = \log n$ we get $O(m \log^* n) + O(m \log \log n)$, which is $O(m \log \log n)$.

Notes by Samir Khuller.

8 Fredman-Tarjan MST Algorithm

Fredman and Tarjan's algorithm appeared in [6].

We maintain a forest defined by the edges that have so far been selected to be in the MST. Initially, the forest contains each of the n vertices of G, as a one-vertex tree. We then repeat the following step until there is only one tree.

High Level

start with n trees each of one vertex
repeat
    procedure GROWTREES
    procedure CLEANUP
until only one tree is left

Informally, the algorithm is given at the start of each round a forest of trees. The procedure GROWTREES grows each tree for a few steps (this will be made more precise later) and terminates with a forest having fewer trees. The procedure CLEANUP essentially "shrinks" the trees to single vertices. This is done by simply *discarding* all edges that have both the endpoints in the same tree. From the set of edges between two different trees we simply retain the lowest weight edge and discard all other edges. A linear time implementation of CLEANUP will be done by you in the homework (very similar to one phase of Boruvka's algorithm).

The idea is to grow a single tree only until its heap of neighboring vertices exceeds a certain critical size. We then start from a new vertex and grow another tree, stopping only when the heap gets too large, or if we encounter a previously stopped tree. We continue this way until every tree has grown, and stopped because it had too many neighbours, or it collided with a stopped tree. We distinguish these two cases. In the former case refer to the tree has having stopped, and in the latter case we refer to it as having halted. We now condense each tree into a single supervertex and begin a new iteration of the same kind in the condensed graph. After a sufficient number of passes, only one vertex will remain.

We fix a parameter k, at the start of every phase – each tree is grown until it has more than k "neighbours" in the heap. In a single call to Growtrees we start with a collection of *old trees*. Growtrees connects these trees to form *new* larger trees that become the *old trees* for the next phase. To begin, we *unmark* all the trees, create an empty heap, pick an unmarked tree and grow it by Prim's algorithm, until either its heap contains more than k vertices or it gets connected to a marked old tree. To finish the growth step we mark the tree. The F-heap maintains the set of all trees that are adjacent to the current tree (tree to grow).

Running Time of GROWTREES

Cost of one phase: Pick a node, and mark it for growth until the F-heap has k neighbors or it collides with another tree. Assume there exist t trees at the start and m is the number of edges at the start of an iteration.

$$\text{Let } k = 2^{2m/t}.$$

Notice that k increases as the number of trees decreases. (Observe that k is essentially 2^d, where d is the average degree of a vertex in the super-graph.) Time for one call to Growtrees is upperbounded by

$$\begin{aligned} O(t \log k + m) &= O(t \log(2^{2m/t}) + m) \\ &= O(t2m/t + m) \\ &= O(m) \end{aligned}$$

This is the case because we perform at most t deletemin operations (each one reduces the number of trees), and $\log k$ is the upperbound on the cost of a single heap operation. The time for CLEANUP is $O(m)$.

Data Structures

Q	=	F-heap of trees (heap of neighbors of the current tree)
e	=	array[trees] of edge ($e[T]$ = cheapest edge joining T to current tree)
mark	=	array[trees] of (true, false), (true for trees already grown in this step)
tree	=	array[vertex] of trees (tree[v] = tree containing vertex v)
edge-list	=	array[trees] of list of edges (edges incident on T)
root	=	array[trees] of trees (first tree that grew, for an active tree)

proc GROWTREES(G);
  MST $\leftarrow \emptyset$;
  $\forall T$: mark[T] $\leftarrow$ false;
  while there exists a tree T_0 with mark[T_0]= false do
    mark[T_0] $\leftarrow$ true; (* grow T_0 by Prim's algorithm *)
    $Q \leftarrow \emptyset$; root[T_0] $\leftarrow T_0$
    For edges (v, w) on edge-list[T_0] do;
      $T \leftarrow$ tree[w]; (* assume $v \in T_0$ *)
      insert T into Q with key $= w(v, w)$;
      $e[T] \leftarrow (v, w)$;
    end-for
    done $\leftarrow (Q = \emptyset) \vee (|Q| > k)$;
    while not done do
      $T \leftarrow deletemin(Q)$;
      add edge $e[T]$ to MST; root[T] $\leftarrow T_0$;
      If not mark[T] then for edges (v, w) on edge-list[T] do
        $T' \leftarrow$ tree[w]; (* assume $v \in T$ *)
        If root[T'] $\neq T_0$ then (* edge goes out of my tree *)
          If $T' \notin Q$ then *insert* T' into Q with key $= w(v, w)$ and $e[T'] \leftarrow (v, w)$;
          else if $T' \in Q$ and $w(v, w) < w(e[T'])$ then *dec-key* T' to $w(v, w)$ and $e[T'] \leftarrow (v, w)$;
      end-if
      done $\leftarrow (Q = \emptyset) \vee (|Q| > k) \vee$ mark[T];
      mark[T] $\leftarrow$ true;
    end-while
  end-while
end proc;

Comment: It is assumed that *insert* will check to see if the heap size exceeds k, and if it does, no items will be inserted into the heap.

We now try to obtain a bound on the number of iterations. The following simple argument is due to Randeep Bhatia. Consider the effect of a pass that begins with t trees and $m' \leq m$ edges (some edges may have been discarded). Each tree that stopped due to its heap's size having exceeded k, has at least k edges incident on it leaving the tree. Let us "charge" each such edge (hence we charge at least k edges).

If a tree halted after colliding with an initially grown tree, then the merged tree has the property that it has charged at least k edges (due to the initially stopped tree). If a tree T has stopped growing because it has too many neighbors, it may happen that due to a merge that occurs later in time, some of its neighbors are in the same tree (we will see in a second this does not cause a problem). In any case, it is true that *each* final tree has charged at least k edges. (A tree that halted after merging with a stopped tree does not need to charge any edges.) Each edge may get charged twice; hence $t' \leq \frac{2m'}{k}$. Clearly, $t' \leq \frac{2m}{k}$. Hence $k' = 2^{\frac{2m}{t'}} \geq 2^k$. In the first round, $k = 2^{2m/n}$. When $k \geq n$, the algorithm runs to completion. How many rounds does this take? Observe that k is increasing exponentially in each iteration.

Let $\beta(m, n) = \min\{i | \log^{(i)} n \leq \frac{2m}{n}\}$. It turns out that the number of iterations is at most $\beta(m, n)$. This gives us an algorithm with total running time $O(m\beta(m, n))$. Observe that when $m > n \log n$ then $\beta(m, n) = 1$. For graphs that are not too sparse, our algorithm terminates in one iteration! For very sprase graphs ($m = O(n)$) we actually run for $\log^* n$ iterations. Since each iteration takes $O(m)$ time, we get an upper bound of $O(m \log^* n)$.

CLEANUP will have to do the work of updating the root and tree data structures for the next iteration.

CMSC 651 Advanced Algorithms	Lecture 24
Lecturer: Samir Khuller	Th. Apr 25, 2002

Notes by Samir Khuller.

9 Branching Problem

Consider a directed graph $G^D = (V, E)$ with a specified vertex r as the root.

A *branching* of G is a subgraph that is a spanning tree if the directions of the edges are ignored, and which contains a directed path from r to every other vertex.

Given a cost function $c : E \rightarrow R^+$, the problem is to obtain a branching of least total cost. This problem looks similar to the minimum spanning tree problem, but is actually much trickier. The greedy algorithm does not solve it!

Notice the following property about a min-cost branching.

Lemma 9.1 *G^D contains a branching rooted at r if and only if for all $v \in V$, G^D contains a path from r to v.*

The proof is quite easy and left as an exercise. Notice that if T is a branching then every vertex in T (except for r) has in-degree 1.

We are going to assume that the costs on edges are non-negative (this is not restrictive, since we can add a large constant K to the cost of each edge without affecting the structure of the optimal solution).

Consider all the incoming edges to a single vertex v ($\neq r$). A branching will contain exactly one of these edges. Hence we can subtract any constant from the costs of these edges without affecting the structure of the optimal solution. For every vertex, we add a negative constant to the incoming edges to v, so as to make the min-cost incoming edge have cost 0. We can now assume that the cheapest incoming edge to any single vertex has cost 0. If we consider the subgraph of 0 cost edges, we can either find a path from r to v by walking backwards on the 0 cost edges, or we find a 0 cost cycle. If we find a path, we simply delete the path and continue. If we find a cycle, we shrink the cycle to a single node and find a branching in the smaller graph. This gives an easy polynomial time algorithm for the problem.

We now prove that the algorithm actually gives us the min cost branching. The proof uses a notion of "combinatorial dual" that should remind you of linear programming duality. (In fact, if you write the IP for Min Cost Branching and relax it to an LP, and then write the dual of this LP, you get exactly the problem of computing a weight function $w(F)$ for each r-cut. In class we called this a dual variable y_S.)

We define a *rooted r-cut* as follows: partition V into sets $S, V \setminus S$ with $r \in S$. The set of edges from S to $V \setminus S$ is called a rooted r-cut.

Let $\mathcal{F}$ be the set of all rooted r-cuts. We define a weight function $w : \mathcal{F} \rightarrow R^+$. This weight function is said to be *valid* if for any edge e

$$c(e) \geq \sum_{F \text{ contains } e} w(F).$$

Theorem 9.2 *If w is a valid weight function then*

$$c(B) \geq \sum_{F \in \mathcal{F}} w(F)$$

where B is any branching.

Proof:

The proof of this theorem is easy. Consider any branching B, and add up the costs of the edges in the branching.

$$c(B) = \sum_{e \in B} c(e)$$

$$c(B) \geq \sum_{e \in B} \sum_{F \text{ contains } e} w(F)$$

Each $w(F)$ term occurs in the summation as many times as the number of edges of F that are in common with B. Since each F is crossed by every branching, for any rooted r-cut F and B, we have $\mid F \cap B \mid \geq 1$. Hence the theorem follows. □

We now prove that there is a valid weight function for which $\sum_{F \in \mathcal{F}} w(F)$ is equal to $c(B)$, where B is the branching produced by the algorithm. Thus B is an optimal branching.

Theorem 9.3 *If w is a valid weight function then*

$$\min_B(c(B)) = \max_w \sum_{F \in \mathcal{F}} w(F).$$

Proof:

The proof is by induction on (number of vertices + number of non-zero cost edges). The proof is easy to verify for a graph with 2 nodes (base case). We will prove the $\leq$ part of the equality (the other follows from the previous theorem).

The first stage of the algorithm reduces the cost on all edges incoming to v (let's call this set of edges F_v). We change the cost of the min-cost branching by K (cost of cheapest edge in F_v). The new graph has the same number of vertices, but fewer non-zero cost edges, so we can apply the induction hypothesis. F_v is a rooted r-cut that has weight 0 in the new graph G'. In G^D we define its weight to be K; it can be checked that this yields a valid weight function. The other rooted r-cuts get the same weight as they received in the shrunken graph.

The other case is when the algorithm contracts a 0 cycle. We now get a graph with fewer vertices, so we can apply the induction hypothesis to the shruken graph G'. We can obtain a min-cost branching for G^D with the same cost as that for G'. The rooted r-cuts for G' correspond to the rooted r-cuts for G^D for which all the vertices of the 0 cycle are in the same set of the partition. We keep their weights unchanged, and define the weights of the other rooted r-cuts to be 0. The weight function clearly achieves the bounds in the lemma. □

CMSC 651 Advanced Algorithms	Lecture 9
Lecturer: Samir Khuller	Th. Feb 16, 1995

Notes by Samir Khuller.

10 Light Approximate Shortest Path Trees

See the algorithm described in [14]. Also directly relevant is work by Awerbuch, Baratz and Peleg referenced in [14]. The basic idea is due to [1], further improvements are due to [4, 14].

Does a graph contain a tree that is a "light" tree (at most a constant times heavier than the minimum spanning tree), such that the distance from the root to any vertex is no more than a constant times the true distance (the distance in the graph between the two nodes). We answer this question in the affirmative and also give an efficient algorithm to compute such a tree (also called a Light Approximate Shortest Path Tree (LAST)). We show that $w(T) < (1+\frac{2}{\epsilon})w(T_M)$ and $\forall v$ $\text{dist}_T(r,v) \le (1+\epsilon)\text{dist}(r,v)$. Here T_M refers to the MST of G. dist_T refers to distances in the tree T, and $\text{dist}(r,v)$ refers to the distance from r to v in the original graph G. T_S refers to the shortest path tree rooted at r (if you do not know what a shortest path tree is, please look up [CLR]).

The main idea is the following: first compute the tree T_M (use any of the standard algorithms). Now do a DFS traversal of T_M, adding certain paths to T_M. These paths are chosen from the shortest path tree T_S and added to bring "certain" nodes closer to the root. This yields a subgraph H, that is guaranteed (we will prove this) to be only a little heavier than T_M. In this subgraph, we do a shortest path computation from r obtaining the LAST T (rooted at r). The LAST tree is not much heavier than T_M (because H was light), and all the nodes are close to the root.

The algorithm is now outlined in more detail. Fix ϵ to be a constant ($\epsilon > 0$). This version is more efficient than what was described in class (why?).

Algorithm to compute a LAST

Step 1. Construct a MST T_M, and a shortest path tree T_S (with r as the root) in G.
Step 2. Traverse T_M in a DFS order starting from the root. Mark nodes $B_0, \ldots, B_k$ as follows. Let B_0 be r. As we traverse the tree we measure the distance between the previous marked node to the current vertex. Let P_i be the shortest path from the root to marked node B_i. (Clearly, $w(P_0)$ is 0.) Let $P[j]$ be the shortest path from r to vertex j.
Step 3. If the DFS traversal is currently at vertex j, and the last marked node is B_i, then compare $(w(P_i) + dist_{T_M}(B_i, j))$ with $(1+\epsilon)w(P[j])$. If $(w(P_i) + dist_{T_M}(B_i, j)) > (1+\epsilon)w(P[j])$ then set vertex j to be the next marked node B_{i+1}.
Step 4. Add paths P_i to T_M, for each marked node B_i, to obtain subgraph H.
Step 5. Do a shortest path computation in H (from r) to find the LAST tree T.

10.1 Analysis of the Algorithm

We now prove that the tree achieves the desired properties. We first show that the entire weight of the subgraph H, is no more than $(1+\frac{2}{\epsilon})w(T_M)$. We then show that distances from r to any vertex v is not blown up by a factor more than $(1+\epsilon)$.

Theorem 10.1 *The $w(H) < (1+\frac{2}{\epsilon})w(T_M)$*

Proof:

Assume that B_k is the last marked node. Consider any marked node B_i ($0 < i \le k$), we know that $(w(P_{i-1}) + dist_{T_M}(B_{i-1}, B_i)) > (1+\epsilon)w(P_i)$. Adding up the equations for all values of i, we get

$$\sum_{i=1}^{k}[w(P_{i-1}) + dist_{T_M}(B_{i-1}, B_i)] > (1+\epsilon)\sum_{i=1}^{k} w(P_i).$$

Clearly,

$$\sum_{i=1}^{k} dist_{T_M}(B_{i-1}, B_i) > \epsilon \sum_{i=1}^{k} w(P_i).$$

Hence the total weight of the paths added to T_M is $< \frac{2}{\epsilon} w(T_M)$. (The DFS traversal traverses each edge exactly twice, hence the weight of the paths between consecutive marked nodes is at most twice $w(T_M)$.) Hence $w(H) < (1 + \frac{2}{\epsilon}) w(T_M)$. □

Theorem 10.2

$$\forall v \; dist_T(r, v) \leq (1 + \epsilon) dist(r, v)$$

Proof:

If v is a marked node, then the proof is trivial (since we actually added the shortest path from r to v, $dist_H(r, v) = dist(r, v)$. If v is a vertex between marked nodes B_i and B_{i+1}, then we know that $(w(P_i) + dist_{T_M}(B_i, v)) \leq (\alpha w(P[v]))$. This concludes the proof (since this path is in H). □

10.2 Spanners

This part of the lecture is taken from the paper "On Sparse Spanners of Weighted Graphs" by Althofer, Das, Dobkin, Joseph and Soares in [2].

For any graph H, we define $\text{dist}_H(u, v)$ as the distance between u and v in the graph H. Given a graph $G = (V, E)$ a t-spanner is a spanning subgraph $G' = (V, E')$ of G with the property that for all pairs of vertices u and v,

$$\text{dist}_{G'}(u, v) \leq t \cdot \text{dist}_G(u, v).$$

In other words, we wish to compute a subgraph that provides approximate shortest paths between each pair of vertices. Clearly, our goal is to minimize the size and weight of G', given a fixed value of t (also called the stretch factor). The size of G' refers to the *number* of edges in G', and the weight of G' refers to the total weight of the edges in G'.

There is a very simple (and elegant) algorithm to compute a t-spanner.

proc SPANNER(G);
 $G' \leftarrow (V, \emptyset)$;
 Sort all edges in increasing weight; $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$;
 for $i = 1$ to m do;
  let $e_i = (u, v)$;
  Let $P(u, v)$ be the shortest path in G' from u to v;
  If $w(P(u, v)) > t \cdot w(e_i)$ then;
   $G' \leftarrow G' \cup e_i$;
 end-for
end proc;

Lemma 10.3 *The graph G' is a t-spanner of G.*

Proof:

Let $P(x, y)$ be the shortest path in G between x and y. For each edge e on this path, either it was added to the spanner *or* there was a path of length $t \cdot w(e)$ connecting the endpoints of this edge. Taking a union of all the paths gives us a path of length at most $t \cdot P(x, y)$. (Hint: draw a little picture if you are still puzzled.) □

Lemma 10.4 *Let C be a cycle in G'; then size$(C) > t + 1$.*

Proof:

Let C be a cycle with at most $t+1$ edges. Let $e_{max} = (u, v)$ be the heaviest weight edge on the cycle C. When the algorithm considers e_{max} all the other edges on C have already been added. This implies that there is a path of length at most $t \cdot w(e_{max})$ from u to v. (The path contains t edges each of weight at most $w(e_{max})$.) Thus the edge e_{max} is not added to G'. □

Lemma 10.5 *Let C be a cycle in G' and e an arbitrary edge on the cycle. We claim that $w(C-e) > t \cdot w(e)$.*

Proof:

Suppose there is a cycle C such that $w(C-e) \leq t \cdot w(e)$. Assume that e_{max} is the heaviest weight edge on the cycle. Then $w(C) \leq (t+1) \cdot w(e) \leq (t+1) \cdot w(e_{max})$. This implies that when we were adding the last edge e_{max} there was a path of length at most $t \cdot w(e_{max})$ connecting the endpoints of e_{max}. Thus this edge would not be added to the spanner. □

Lemma 10.6 *The MST is contained in G'.*

The proof of the above lemma is left to the reader.

Definition: Let $E(v)$ be the edges of G' incident to vertex v that do not belong to the MST.

We will prove that $w(E(v)) \leq \frac{2w(MST)}{(t-1)}$.

From this we can conclude the following (since each edge of G' is counted twice in the summation, and we simply plug in the upper bound for $E(v)$).

$$w(G') \leq w(MST) + \frac{1}{2}\sum_{v} w(E(v)) \leq w(MST)(1 + \frac{n}{t-1}).$$

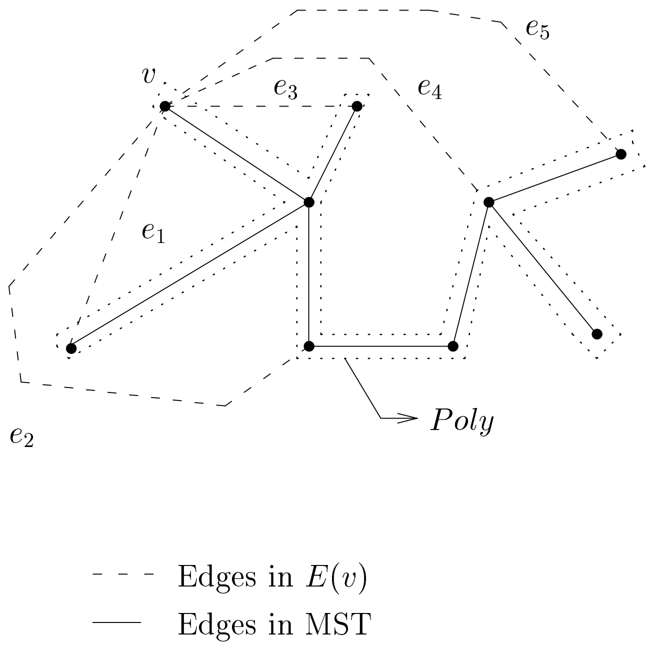


Figure 9: Figure to illustrate polygon, MST and $E(v)$.

Theorem 10.7

$$w(E(v)) \leq \frac{2w(MST)}{t-1}.$$

Proof:

This proof is a little different from the proof we did in class. Let the edges in $E(v)$ be numbered $e_1, e_2, \ldots, e_k$. By the previous lemma we know that for any edge e in G' and path P in G' that connects the endpoints of e, we have $t \cdot w(e) < w(P)$.

Let $Poly$ be a polygon defined by "doubling" the MST. Let W_i be the perimeter of the polygon after i edges have been added to the polygon (see Fig. 9). (Initially, $W_0 = 2w(MST)$.)

Let $T_i = \sum_{j=1}^{i} w(e_j)$.

How does the polygon change after edge $e_i = (u, v)$ has been added?

$$W_{i+1} = W_i + w(e_i) - w(P(u,v)) < W_i - w(e_i)(t-1).$$

$$T_{i+1} = T_i + w(e_i).$$

A moment's reflection on the above process will reveal that as T increases, the perimeter W drops. The perimeter clearly is non-zero at all times and cannot drop below zero. Thus the final value of T_k is upper bounded by $\frac{2w(MST)}{t-1}$. But this is the sum total of the weights of all edges in $E(v)$. $\square$

The proof I gave in class was a more "pictorial" argument. You could write out all the relevant equations by choosing appropriate paths to charge at each step. It gives the same bound.

Much better bounds are known if the underlying graph G is planar.

Original notes by Michael Tan.

11 Matchings

Read: Chapter 9 [21]. Chapter 10 [19].

In this lecture we will examine the problem of finding a maximum matching in bipartite graphs.

Given a graph $G = (V, E)$, a **matching** M is a subset of the edges such that no two edges in M share an endpoint. The problem is similar to finding an independent set of edges. In the maximum matching problem we wish to maximize $|M|$.

With respect to a given matching, a **matched edge** is an edge included in the matching. A **free edge** is an edge which does not appear in the matching. Likewise, a **matched vertex** is a vertex which is an endpoint of a matched edge. A **free vertex** is a vertex that is not the endpoint of any matched edge.

A **bipartite** graph $G = (U, V, E)$ has $E \subseteq U \times V$. For bipartite graphs, we can think of the matching problem in the following terms. Given a list of boys and girls, and a list of all marriage compatible pairs (a pair is a boy and a girl), a matching is some subset of the compatibility list in which each boy or girl gets at most one partner. In these terms, $E = \{$ all marriage compatible pairs $\}$, $U = \{$ the boys$\}$, $V = \{$ the girls$\}$, and $M = \{$ some potential pairing preventing polygamy$\}$.

A **perfect matching** is one in which all vertices are matched. In bipartite graphs, we must have $|V| = |U|$ in order for a perfect matching to possibly exist. When a bipartite graph has a perfect matching in it, the following theorem holds:

11.1 Hall's Theorem

Theorem 11.1 (Hall's Theorem) *Given a bipartite graph $G = (U, V, E)$ where $|U| = |V|, \forall S \subseteq U, |N(S)| \geq |S|$ (where N(S) is the set of vertices which are neighbors of S) iff G has a perfect matching.*

Proof:

($\leftarrow$) In a perfect matching, all elements of S will have at least a total of $|S|$ neighbors since every element will have a partner. ($\rightarrow$) We give this proof after the presentation of the algorithm, for the sake of clarity.
$\square$

For non-bipartite graphs, this theorem does not work. Tutte proved the following theorem (you can read the Graph Theory book by Bondy and Murty for a proof) for establishing the conditions for the existence of a perfect matching in a non-bipartite graph.

Theorem 11.2 (Tutte's Theorem) *A graph G has a perfect matching if and only if $\forall S \subseteq V, o(G - S) \leq |S|$. ($o(G - S)$ is the* **number** *of connected components in the graph $G - S$ that have an odd number of vertices.)*

Before proceeding with the algorithm, we need to define more terms.

An **alternating path** is a path whose edges alternate between matched and unmatched edges.

An **augmenting path** is an alternating path which starts and ends with unmatched vertex (and thus starts and ends with a free edge).

The matching algorithm will attempt to increase the size of a matching by finding an augmenting path. By inverting the edges of the path (matched becomes unmatched and vice versa), we increase the size of a matching by exactly one.

If we have a matching M and an augmenting path P (with respect to M), then $M \oplus P = (M - P) \cup (P - M)$ is a matching of size $|M| + 1$.

11.2 Berge's Theorem

Theorem 11.3 (Berge's Theorem) *M is a maximum matching iff there are no augmenting paths with respect to M.*

Proof:

($\rightarrow$) Trivial. ($\leftarrow$) Let us prove the contrapositive. Assume M is not a maximum matching. Then there exists some maximum matching M' and $|M'| > |M|$. Consider $M \oplus M'$. All of the following will be true of the graph $M \oplus M'$:

1. The highest degree of any node is two.
2. The graph is a collection of cycles and paths.
3. The cycles must be of even length, with half the edges from M and half from M'.
4. Given these first three facts (and since $|M'| > |M|$), there must be some path with more M' edges than M edges.

This fourth fact describes an augmenting path (with respect to M). This path begins and ends with M' edges, which implies that the path begins and ends with free nodes (i.e., free in M). $\square$

Armed with this theorem, we can outline a primitive algorithm to solve the maximum matching problem. It should be noted that Edmonds (1965) was the first one to show how to find an augmenting path in an arbitrary graph (with respect to the current matching) in polynomial time. This is a landmark paper that also defined the notion of polynomial time computability.

Simple Matching Algorithm [Edmonds]:

Step 1: Start with $M = \emptyset$.

Step 2: Search for an augmenting path.

Step 3: Increase M by 1 (using the augmenting path).

Step 4: Go to 2.

We will discuss the search for an augmenting path in bipartite graphs via a simple example (see Fig. 13). See [19] for a detailed description of the pseudo-code for this algorithm.

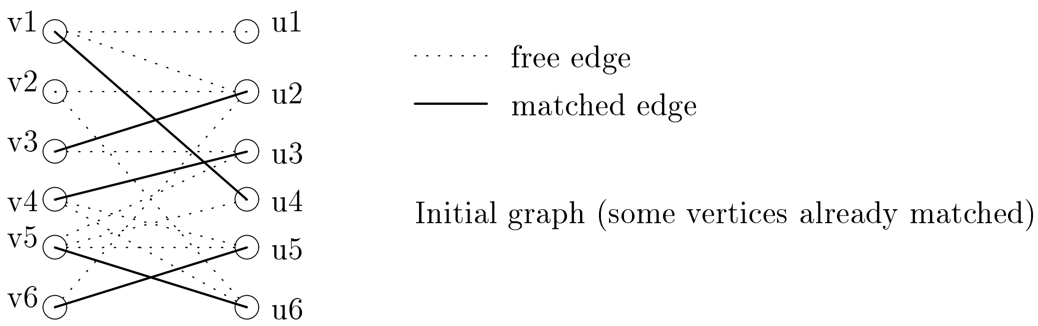
Theorem 11.4 (Hall's Theorem) *A bipartite graph $G = (U, V, E)$ has a perfect matching if and only if $\forall S \subseteq V, |S| \leq |N(S)|$.*

Proof:

To prove this theorem in one direction is trivial. If G has a perfect matching M, then for any subset S, $N(S)$ will always contain all the `mates` (in M) of vertices in S. Thus $|N(S)| \geq |S|$. The proof in the other direction can be done as follows. Assume that M is a maximum matching and is not perfect. Let u be a free vertex in U. Let Z be the set of vertices connected to u by alternating paths w.r.t M. Clearly u is the only free vertex in Z (else we would have an augmenting path). Let $S = Z \cap U$ and $T = Z \cap V$. Clearly the vertices in $S - \{u\}$ are matched with the vertices in T. Hence $|T| = |S| - 1$. In fact, we have $N(S) = T$ since every vertex in $N(S)$ is connected to u by an alternating path. This implies that $|N(S)| < |S|$. $\square$

The upper bound on the number of iterations is $O(|V|)$ (the size of a matching). The time to find an augmenting path is $O(|E|)$ (use BFS). This gives us a total time of $O(|V||E|)$.

In the next lecture, we will learn the Hopcroft-Karp $O(\sqrt{|V|}|E|)$ algorithm for maximum matching on a bipartite graph. This algorithm was discovered in the early seventies. In 1981, Micali-Vazirani extended this algorithm to general graphs. This algorithm is quite complicated, but has the same running time as the Hopcroft-Karp method. It is also worth pointing out that both Micali and Vazirani were first year graduate students at the time.



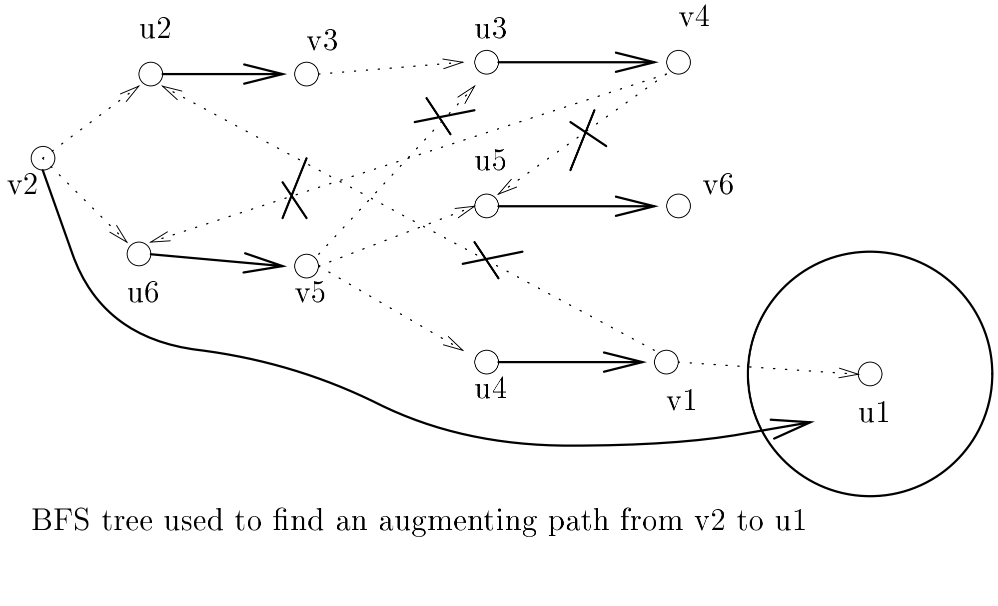


Figure 10: Sample execution of Simple Matching Algorithm.

CMSC 651 Advanced Algorithms	Lecture 6
Lecturer: Samir Khuller	Th. Feb 14, 2002

Notes by Samir Khuller.

12 Hopcroft-Karp Matching Algorithm

The original paper is by Hopcroft and Karp [SIAM J. on Computing, 1973].

We present the Hopcroft-Karp matching algorithm that finds a maximum matching in bipartite graphs.

The main idea behind the Hopcroft-Karp algorithm is to augment along a *set* of shortest augmenting paths *simultaneously*. (In particular, if the shortest augmenting path has length k then in a single phase we obtain a *maximal* set S of vertex disjoint augmenting paths all of length k.) By the maximality property, we have that *any* augmenting path of length k will intersect a path in S. In the next phase, we will have the property that the augmenting paths found will be strictly longer (we will prove this formally later). We will implement each phase in linear time.

We first prove the following lemma.

Lemma 12.1 *Let M be a matching and P a shortest augmenting path w.r.t M. Let P' be a shortest augmenting path w.r.t $M \oplus P$ (symmetric difference of M and P). We have*

$$|P'| \geq |P| + 2|P \cap P'|.$$

$|P \cap P'|$ refers to the edges that are common to both the paths.

Proof:

Let $N = (M \oplus P) \oplus P'$. Thus $N \oplus M = P \oplus P'$. Consider $P \oplus P'$. This is a collection of cycles and paths (since it is the same as $M \oplus N$). The cycles are all of even length. The paths may be of odd or even length. The odd length paths are augmenting paths w.r.t M. Since the two matchings differ in cardinality by 2, there must be two odd length augmenting paths P_1 and P_2 w.r.t M. Both of these must be longer than P (since P was the shortest augmenting path w.r.t M).

$$|M \oplus N| = |P \oplus P'| = |P| + |P'| - 2|P \cap P'| \geq |P_1| + |P_2| \geq 2|P|.$$

Simplifying we get

$$|P'| \geq |P| + 2|P \cap P'|.$$

□

We still need to argue that after each phase, the shortest augmenting path is strictly longer than the shortest augmenting paths of the previous phase. (Since the augmenting paths always have an odd length, they must actually increase in length by two.)

Suppose that in some phase we augmented the current matching by a maximal set of vertex disjoint paths of length k (and k was the length of the shortest possible augmenting path). This yields a new matching M'. Let P' be an augmenting path with respect to the new matching. If P' is vertex disjoint from each path in the previous set of paths it must have length strictly more than k. If it shares a vertex with some path P in our chosen set of paths, then $P \cap P'$ contains at least one edge in M' since every vertex in P is matched in M'. (Hence P' cannot intersect P on a vertex and not share an edge with P.) By the previous lemma, P' exceeds the length of P by at least two.

Lemma 12.2 *A maximal set S of disjoint, minimum length augmenting paths can be found in $O(m)$ time.*

Proof:

Let $G = (U, V, E)$ be a bipartite graph and let M be a matching in G. We will grow a "Hungarian Tree" in G. (The tree really is not a tree but we will call it a tree all the same.) The procedure is similar to BFS and very similar to the search for an augmenting path that was done in the previous lecture. We start by putting the free vertices in U in level 0. Starting from even level $2k$, the vertices at level $2k+1$ are obtained by following free edges from vertices at level $2k$ that have not been put in any level as yet. Since the graph is bipartite the odd(even) levels contain only vertices from $V(U)$. From each odd level $2k+1$, we simple add the matched neighbours of the vertices to level $2k+2$. We repeat this process until we encounter a free vertex in an odd level (say t). We continue the search only to discover all free vertices in level t, and stop when we have found all such vertices. In this procedure clearly each edge is traversed at most once, and the time is $O(m)$.

We now have a second phase in which the maximal set of disjoint augmenting paths of length k is found. We use a technique called *topological erase*, called so because it is a little like topological sort. With *each* vertex x (except the ones at level 0), we associate an integer counter initially containing the number of edges entering x from the previous level (we also call this the indegree of the vertex). Starting at a free vertex v at the last level t, we trace a path back until arriving at a free vertex in level 0. The path is an augmenting path, and we include it in S.

At all points of time we maintain the invariant that there is a path from each free node (in V) in the last level to some free node in level 0. This is clearly true initially, since each free node in the last level was discovered by doing a BFS from free nodes in level 0. When we find an augmenting path we place all vertices along this path on a deletion queue. As long as the deletion queue is non-empty, we remove a vertex from the queue, delete it together with its adjacent vertices in the Hungarian tree. Whenever an edge is deleted, the counter associated with its right endpoint is decremented if the right endpoint is not on the current path (since this node is losing an edge entering it from the left). If the counter becomes 0, the vertex is placed on the deletion queue (there can be no augmenting path in the Hungarian tree through this vertex, since all its incoming edges have been deleted). This maintains the invariant that when we grab paths coming backwards from the final level to level 0, then we automatically delete nodes that have no paths backwards to free nodes.

After the queue becomes empty, if there is still a free vertex v at level t, then there must be a path from v backwards through the Hungarian tree to a free vertex on the first level; so we can repeat this process. We continue as long as there exist free vertices at level t. The entire process takes linear time, since the amount of work is proportional to the number of edges deleted. □

Let's go through the following example (see Fig. 11) to make sure we understand what's going on. Start with the first free vertex v_6. We walk back to u_6 then to v_5 and to u_1 (at this point we had a choice of u_5 or u_1). We place all these nodes on the deletion queue, and start removing them and their incident edges. v_6 is the first to go, then u_6 then v_5 and then u_1. When we delete u_1 we delete the edge (u_1, v_1) and decrease the indegree of v_1 from 2 to 1. We also delete the edges (v_6, u_3) and (v_5, u_5) but these do not decrease the indegree of any node, since the right endpoint is already on the current path.

Start with the next free vertex v_3. We walk back to u_3 then to v_1 and to u_2. We place all these nodes on the deletion queue, and start removing them and their incident edges. v_3 is the first to go, then u_3 (this deletes edge (u_3, v_4) and decreases the indegree of v_4 from 2 to 1). Next we delete v_1 and u_2. When we delete u_2 we remove the edge (u_2, v_2) and this decreases the indegree of v_2 which goes to 0. Hence v_2 gets added to the deletion queue. Doing this makes the edge (v_2, u_4) get deleted, and drops the indegree of u_4 to 0. We then delete u_4 and the edge (u_4, v_4) is deleted and v_4 is removed. There are no more free nodes in the last level, so we stop. The key point is that it is not essential to find the maximum set of disjoint augmenting paths of length k, but only a maximal set.

Theorem 12.3 *The total running time of the above described algorithm is $O(\sqrt{n}m)$.*

Proof:

Each phase runs in $O(m)$ time. We now show that there are $O(\sqrt{n})$ phases. Consider running the algorithm for exactly $\sqrt{n}$ phases. Let the obtained matching be M. Each augmenting path from now on is of length at least $2\sqrt{n}+1$. (The paths are always odd in length and always increase by at least two after each phase.) Let M^* be the max matching. Consider the symmetric difference of M and M^*. This is a collection of cycles and paths, that contain the augmenting paths w.r.t M. Let $k = |M^*| - |M|$. Thus there are k augmenting paths w.r.t M that yield the matching M^*. Each path has length at least $2\sqrt{n}+1$, and they are disjoint. The total length of the paths is at most n (due to the disjointness). If l_i is the length of each augmenting path we have:

$$k(2\sqrt{n}+1) \leq \sum_{i=1}^{k} l_i \leq n.$$

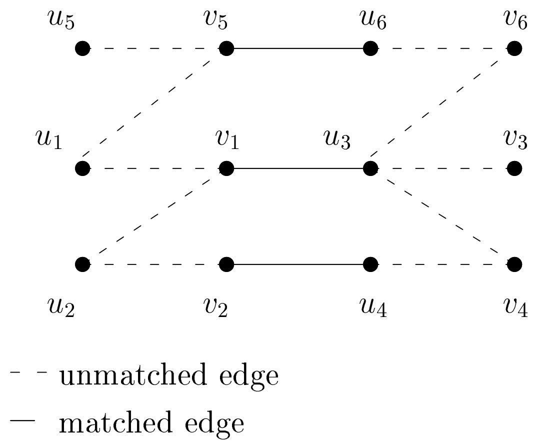


Figure 11: Sample execution of Topological Erase.

Thus k is upper bounded by $\frac{n}{2\sqrt{n}+1} \leq \frac{n}{2\sqrt{n}} \leq \frac{\sqrt{n}}{2}$. In each phase we increase the size of the matching by at least one, so there are at most k more phases. This proves the required bound of $O(\sqrt{n})$. $\square$

CMSC 651 Advanced Algorithms	Lecture 4
Lecturer: Samir Khuller	Th. Feb 7. 2002

Notes by Samir Khuller.

13 Two Processor Scheduling

We will also talk about an application of matching, namely the problem of scheduling on two processors. The original paper is by Fujii, Kasami and Ninomiya [7].

There are two identical processors, and a collection of n jobs that need to be scheduled. Each job requires unit time. There is a precedence graph associated with the jobs (also called the DAG). If there is an edge from i to j then i must be finished before j is started by either processor. How should the jobs be scheduled on the two processors so that all the jobs are completed as quickly as possible. The jobs could represent courses that need to be taken (courses have prerequisites) and if we are taking at most two courses in each semester, the question really is: how quickly can we graduate?

This is a good time to note that even though the two processor scheduling problem can be solved in polynomial time, the three processor scheduling problem is not known to be solvable in polynomial time, or known to be NP-complete. In fact, the complexity of the k processor scheduling problem when k is *fixed* is not known.

From the acyclic graph G we can construct a *compatibility* graph G^* as follows. G^* has the same nodes as G, and there is an (undirected) edge (i, j) if there is no directed path from i to j, or from j to i in G. In other words, i and j are jobs that can be done together.

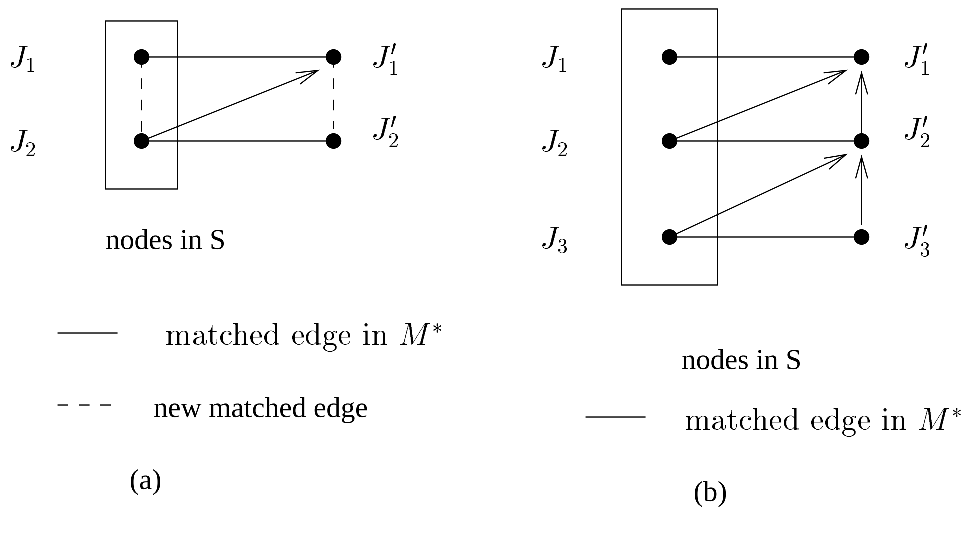


Figure 12: (a) Changing the schedule (b) Continuing the proof.

A maximum matching in G^* is the indicates the maximum number of pairs of jobs that can be processed simultaneously. Clearly, a solution to the scheduling problem can be used to obtain a matching in G^*. More interestingly, a solution to the matching problem can be used to obtain a solution to the scheduling problem!

Suppose we find a maximum matching in M in G^*. An unmatched vertex is executed on a single processor while the other processor is idle. If the maximum matching has size m^*, then $2m^*$ vertices are matched, and $n - 2m^*$ vertices are left unmatched. The time to finish all the jobs will be $m^* + n - 2m^* = n - m^*$. Hence a maximum matching will minimize the time required to schedule all the jobs.

We now argue that given a maximum matching M^*, we can extract a schedule of size $n - m^*$. The key idea is that each matched job is scheduled in a slot together with another job (which may be different from the job it was matched to). This ensures that we do not use more than $n - m^*$ slots.

Let S be the subset of vertices that have indegree 0. The following cases show how a schedule can be constructed from G and M^*. Basically, we have to make sure that for all jobs that are paired up in M^*, we pair them up in the schedule. The following rules can be applied repeatedly.

1. If there is an unmatched vertex in S, schedule it and remove it from G.
2. If there is a pair of jobs in S that are matched in M^*, then we schedule the pair and delete both the jobs from G.

If none of the above rules are applicable, then all jobs in S are matched; moreover each job in S is matched with a job not in S.

Let $J_1 \in S$ be matched to $J'_1 \notin S$. Let J_2 be a job in S that has a path to J'_1. Let J'_2 be the mate of J_2. If there is path from J'_1 to J'_2, then J_2 and J'_2 could not be matched to each other in G^* (this cannot be an edge in the compatibility graph). If there is no path from J'_2 to J'_1 then we can *change* the matching to match (J_1, J_2) and (J'_1, J'_2) since (J'_1, J'_2) is an edge in G^* (see Fig. 12 (a)).

The only remaining case is when J'_2 has a path to J'_1. Recall that J_2 also has a path to J'_1. We repeat the above argument considering J'_2 in place of J'_1. This will yield a new pair (J_3, J'_3) etc (see Fig. 12 (b)). Since the graph G has no cycles at each step we will generate a distinct pair of jobs; at some point of time this process will stop (since the graph is finite). At that time we will find a pair of jobs in S we can schedule together.

Notes by Samir Khuller.

14 Assignment Problem

Consider a complete bipartite graph, $G(X, Y, X \times Y)$, with weights $w(e_i)$ assigned to every edge. (One could think of this problem as modeling a situation where the set X represents workers, and the set Y represents jobs. The weight of an edge represents the "compatability" factor for a (worker,job) pair. We need to assign workers to jobs such that each worker is assigned to exactly one job.) The **Assignment Problem** is to find a matching with the greatest total weight, i.e., the maximum-weighted perfect matching (which is not necessarily unique). Since G is a complete bipartite graph we know that it has a perfect matching.

An algorithm which solves the Assignment Problem is due to Kuhn and Munkres. We assume that all the edge weights are non-negative,

$$w(x_i, y_j) \geq 0.$$

where

$$x_i \in X, y_j \in Y.$$

We define a *feasible vertex labeling* l as a mapping from the set of vertices in G to the real numbers, where

$$l(x_i) + l(y_j) \geq w(x_i, y_j).$$

(The real number $l(v)$ is called the label of the vertex v.) It is easy to compute a feasible vertex labeling as follows:

$$(\forall y_j \in Y) \quad [l(y_j) = 0].$$

and

$$l(x_i) = \max_j w(x_i, y_j).$$

We define the **Equality Subgraph**, G_l, to be the spanning subgraph of G which includes all vertices of G but only those edges (x_i, y_j) which have weights such that

$$w(x_i, y_j) = l(x_i) + l(y_j).$$

The connection between equality subgraphs and maximum-weighted matchings is provided by the following theorem:

Theorem 14.1 *If the Equality Subgraph, G_l, has a perfect matching, M^*, then M^* is a maximum-weighted matching in G.*

Proof:
Let M^* be a perfect matching in G_l. We have, by definition,

$$w(M^*) = \sum_{e \in M^*} w(e) = \sum_{v \in X \cup Y} l(v).$$

Let M be any perfect matching in G. Then

$$w(M) = \sum_{e \in M} w(e) \leq \sum_{v \in X \cup Y} l(v) = w(M^*).$$

Hence,

$$w(M) \leq w(M^*).$$

□

In fact note that the sum of the labels is an upper bound on the weight of the maximum weight perfect matching.

High-level Description:

The above theorem is the basis of an algorithm, due to Kuhn and Munkres, for finding a maximum-weighted matching in a complete bipartite graph. Starting with a feasible labeling, we compute the equality subgraph and then find a maximum matching in this subgraph (now we can ignore weights on edges). If the matching found is perfect, we are done. If the matching is not perfect, we add more edges to the equality subgraph by revising the vertex labels. We also ensure that edges from our current matching do not leave the equality subgraph. After adding edges to the equality subgraph, either the size of the matching goes up (we find an augmenting path), or we continue to grow the hungarian tree. In the former case, the phase terminates and we start a new phase (since the matching size has gone up). In the latter case, we grow the hungarian tree by adding new nodes to it, and clearly this cannot happen more than n times.

Let S = the set of free nodes in X. Grow hungarian trees from each node in S. Let T = all nodes in Y encountered in the search for an augmenting path from nodes in S. Add all nodes from X that are encountered in the search to S.

Some More Details:

We note the following about this algorithm:

$$\overline{S} = X - S.$$

$$\overline{T} = Y - T.$$

$$|S| > |T|.$$

There are no edges from S to $\overline{T}$, since this would imply that we did not grow the hungarian trees completely. As we grow the Hungarian Trees in G_l, we place alternate nodes in the search into S and T. To revise the labels we take the labels in S and start decreasing them uniformly (say by λ), and at the same time we increase the labels in T by λ. This ensures that the edges from S to T do not leave the equality subgraph (see Fig. 13).

As the labels in S are decreased, edges (in G) from S to $\overline{T}$ will potentially enter the Equality Subgraph, G_l. As we increase λ, at some point of time, an edge enters the equality subgraph. This is when we stop and update the hungarian tree. If the node from $\overline{T}$ added to G_l is matched to a node in $\overline{S}$, we move both these nodes to S and T, which yields a larger Hungarian Tree. If the node from $\overline{T}$ is free, we have found an augmenting path and the phase is complete. One phase consists of those steps taken between increases in the size of the matching. There are at most n phases, where n is the number of vertices in G (since in each phase the size of the matching increases by 1). Within each phase we increase the size of the hungarian tree at most n times. It is clear that in $O(n^2)$ time we can figure out which edge from S to $\overline{T}$ is the first one to enter the equality subgraph (we simply scan all the edges). This yields an $O(n^4)$ bound on the total running time. Let us first review the algorithm and then we will see how to implement it in $O(n^3)$ time.

The Kuhn-Munkres Algorithm (also called the Hungarian Method):

Step 1: Build an Equality Subgraph, G_l by initializing labels in any manner (this was discussed earlier).

Step 2: Find a maximum matching in G_l (not necessarily a perfect matching).

Step 3: If it is a perfect matching, according to the theorem above, we are done.

Step 4: Let S = the set of free nodes in X. Grow hungarian trees from each node in S. Let T = all nodes in Y encountered in the search for an augmenting path from nodes in S. Add all nodes from X that are encountered in the search to S.

Step 5: Revise the labeling, l, *adding edges to* G_l until an augmenting path is found, adding vertices to S and T as they are encountered in the search, as described above. Augment along this path and increase the size of the matching. Return to step 4.

Only edges in G_l are shown

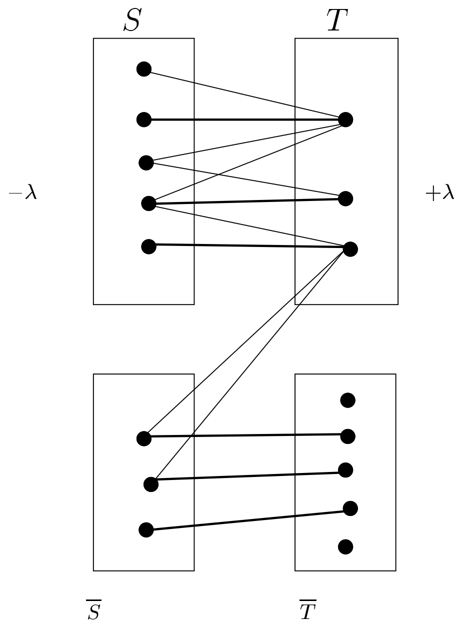


Figure 13: Sets S and T as maintained by the algorithm.

More Efficient Implementation:
We define the slack of an edge as follows:

$$slack(x, y) = l(x) + l(y) - w(x, y).$$

Then

$$\lambda = \min_{x \in S, y \in \overline{T}} slack(x, y)$$

Naively, the calculation of λ requires $O(n^2)$ time. For every vertex in $\overline{T}$, we keep track of the edge with the smallest slack, i.e.,

$$slack[y_j] = \min_{x_i \in S} slack(x_i, y_j)$$

The computation of $slack[y_j]$ requires $O(n^2)$ time at the start of a phase. As the phase progresses, it is easy to update all the *slack* values in $O(n)$ time since all of them change by the same amount (the labels of the vertices in S are going down uniformly). Whenever a node u is moved from $\overline{S}$ to S we must recompute the slacks of the nodes in $\overline{T}$, requiring $O(n)$ time. But a node can be moved from $\overline{S}$ to S at most n times.

Thus each phase can be implemented in $O(n^2)$ time. Since there are n phases, this gives us a running time of $O(n^3)$.

CMSC 651 Advanced Algorithms	Lecture 8
Lecturer: Samir Khuller	Th. Feb 21, 2002

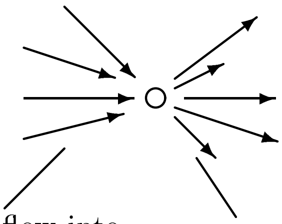
Original notes by Sibel Adali and Andrew Vakhutinsky.

15 Network Flow - Maximum Flow Problem

Read [21, 5, 19].

The problem is defined as follows: Given a directed graph $G^d = (V, E, s, t, c)$ where s and t are special vertices called the source and the sink, and c is a capacity function $c : E \rightarrow \Re^+$, find the maximum flow from s to t.
Flow is a function $f : E \rightarrow \Re$ that has the following properties:

1. **(Skew Symmetry)** $f(u, v) = -f(v, u)$
2. **(Flow Conservation)** $\Sigma_{v \in V} f(u, v) = 0$ for all $u \in V - \{s, t\}$.
   (Incoming flow) $\Sigma_{v \in V} f(v, u) =$ (Outgoing flow) $\Sigma_{v \in V} f(u, v)$



Flow Conservation

3. **(Capacity Constraint)** $f(u, v) \leq c(u, v)$

Maximum flow is the maximum value $|f|$ given by

$$|f| = \Sigma_{v \in V} f(s, v) = \Sigma_{v \in V} f(v, t).$$

Definition 15.1 (Residual Graph) G_f^D *is defined with respect to some flow function* f, $G_f = (V, E_f, s, t, c')$ *where* $c'(u, v) = c(u, v) - f(u, v)$. *Delete edges for which* $c'(u, v) = 0$.

As an example, if there is an edge in G from u to v with capacity 15 and flow 6, then in G_f there is an edge from u to v with capacity 9 (which means, I can still push 9 more units of liquid) and an edge from v to u with capacity 6 (which means, I can cancel all or part of the 6 units of liquid I'm currently pushing) [1]. E_f contains all the edges e such that $c'(e) > 0$.

Lemma 15.2 *Here are some easy to prove facts:*

1. f' *is a flow in* G_f *iff* $f + f'$ *is a flow in* G.
2. f' *is a maximum flow in* G_f *iff* $f + f'$ *is a maximum flow in* G.
3. $|f + f'| = |f| + |f'|$.

[1] Since there was no edge from v to u in G, then its capacity was 0 and the flow on it was -6. Then, the capacity of this edge in G_f is $0 - (-6) = 6$.

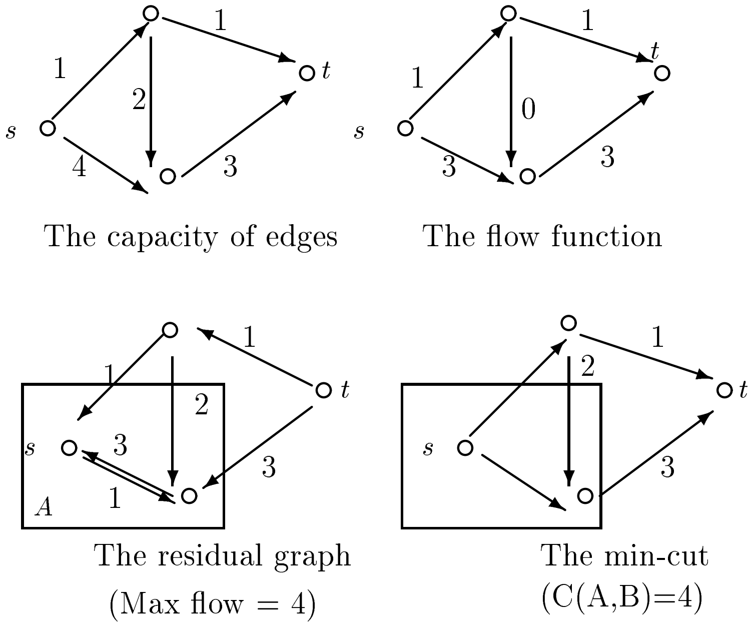


Figure 14: An example

4. If f is a flow in G, and f^* is the maximum flow in G, then $f^* - f$ is the maximum flow in G_f.

Definition 15.3 (Augmenting Path) *A path P from s to t in the residual graph G_f is called augmenting if for all edges (u, v) on P, $c'(u, v) > 0$. The* residual capacity *of an augmenting path P is $\min_{e \in P} c'(e)$.*

The idea behind this definition is that we can send a positive amount of flow along the augmenting path from s to t and "augment" the flow in G. (This flow increases the real flow on some edges and cancels flow on other edges, by reversing flow.)

Definition 15.4 (Cut) *An (s, t) cut is a partitioning of V into two sets A and B such that $A \cap B = \emptyset$, $A \cup B = V$ and $s \in A, t \in B$.*

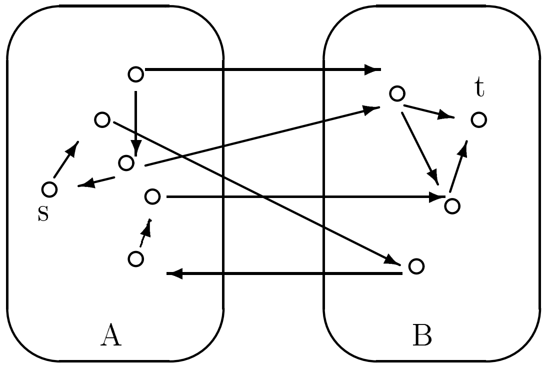


Figure 15: An (s, t) Cut

Definition 15.5 (Capacity Of A Cut) *The capacity $C(A, B)$ is given by*

$$C(A, B) = \Sigma_{a \in A, b \in B} \ c(a, b).$$

By the capacity constraint we have that $|f| = \Sigma_{v \in V} f(s,v) \leq C(A,B)$ for any (s,t) cut (A,B). Thus the capacity of the minimum capacity s,t cut is an upper bound on the value of the maximum flow.

Theorem 15.6 (Max flow - Min cut Theorem) *The following three statements are equivalent:*

1. *f is a maximum flow.*
2. *There exists an (s,t) cut (A,B) with $C(A,B) = |f|$.*
3. *There are no augmenting paths in G_f.*

An augmenting path is a directed path from s to t.

Proof:
We will prove that $(2) \Rightarrow (1) \Rightarrow (3) \Rightarrow (2)$.

((2) $\Rightarrow$ (1)) Since no flow can exceed the capacity of an (s,t) cut (i.e. $f(A,B) \leq C(A,B)$), the flow that satisfies the equality condition of **(2)** must be the maximum flow.

((1) $\Rightarrow$ (3)) If there was an augmenting path, then I could augment the flow and find a larger flow, hence f wouldn't be maximum.

((3) $\Rightarrow$ (2)) Consider the residual graph G_f. There is no directed path from s to t in G_f, since if there was this would be an augmenting path. Let $A = \{v | v \text{ is reachable from } s \text{ in } G_f\}$. A and $\overline{A}$ form an (s,t) cut, where all the edges go from $\overline{A}$ to A. The flow f' must be equal to the capacity of the edge, since for all $u \in A$ and $v \in \overline{A}$, the capacity of (u,v) is 0 in G_f and $0 = c(u,v) - f'(u,v)$, therefore $c(u,v) = f'(u,v)$. Then, the capacity of the cut in the original graph is the total capacity of the edges from A to $\overline{A}$, and the flow is exactly equal to this amount. □

A "Naive" Max Flow Algorithm:

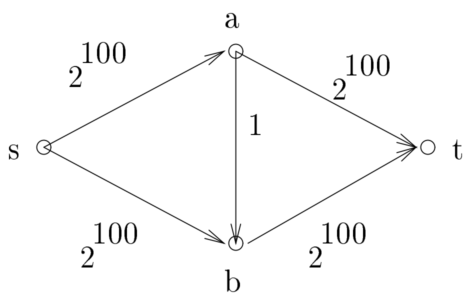
Initially let f be the 0 flow
while (there is an augmenting path P in G_f) **do**
  $c(P) \leftarrow \min_{e \in P} c'(e)$;
  send flow amount $c(P)$ along P;
  update flow value $|f| \leftarrow |f| + c(P)$;
  compute the new residual flow network G_f

Analysis: The algorithm starts with the zero flow, and stops when there are no augmenting paths from s to t. If all edge capacities are integral, the algorithm will send at least one unit of flow in each iteration (since we only retain those edges for which $c'(e) > 0$). Hence the running time will be $O(m|f^*|)$ in the worst case ($|f^*|$ is the value of the max-flow).

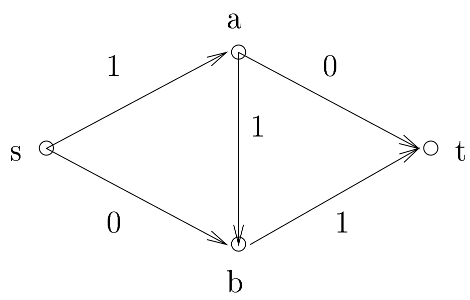
A worst case example. Consider a flow graph as shown on the Fig. 16. Using augmenting paths (s,a,b,t) and (s,b,a,t) alternatively at odd and even iterations respectively (fig.1(b-c)), it requires total $|f^*|$ iterations to construct the max flow.

If all capacities are rational, there are examples for which the flow algorithm might never terminate. The example itself is intricate, but this is a fact worth knowing.

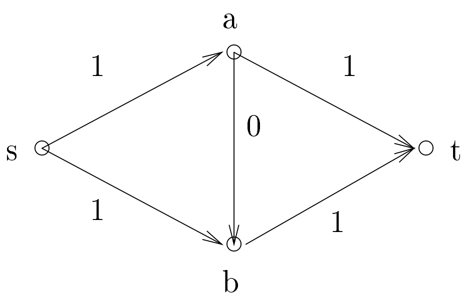
Example. Consider the graph on Fig. 17 where all edges except (a,d), (b,e) and (c,f) are unbounded (have comparatively large capacities) and $c(a,d) = 1$, $c(b,e) = R$ and $c(c,f) = R^2$. Value R is chosen such that $R = \frac{\sqrt{5}-1}{2}$ and, clearly (for any $n \geq 0$, $R^{n+2} = R^n - R^{n+1}$. If augmenting paths are selected as shown on Fig. 17 by dotted lines, residual capacities of the edges (a,d), (b,e) and (c,f) will remain 0, R^{3k+1} and R^{3k+2} after every $(3k+1)$st iteration ($k = 0,1,2,\ldots$). Thus, the algorithm will never terminate.



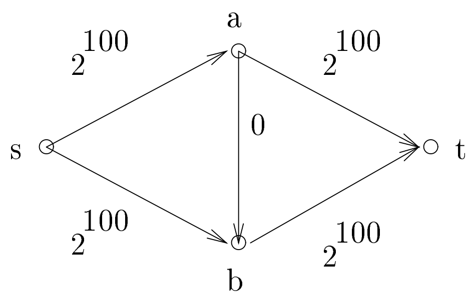
(a) Initial flow graph with capacities



(b) Flow in the graph after the 1st iteration



(c) Flow in the graph after the 2nd iteration



(d) Flow in the graph after the final iteration

Figure 16: Worst Case Example

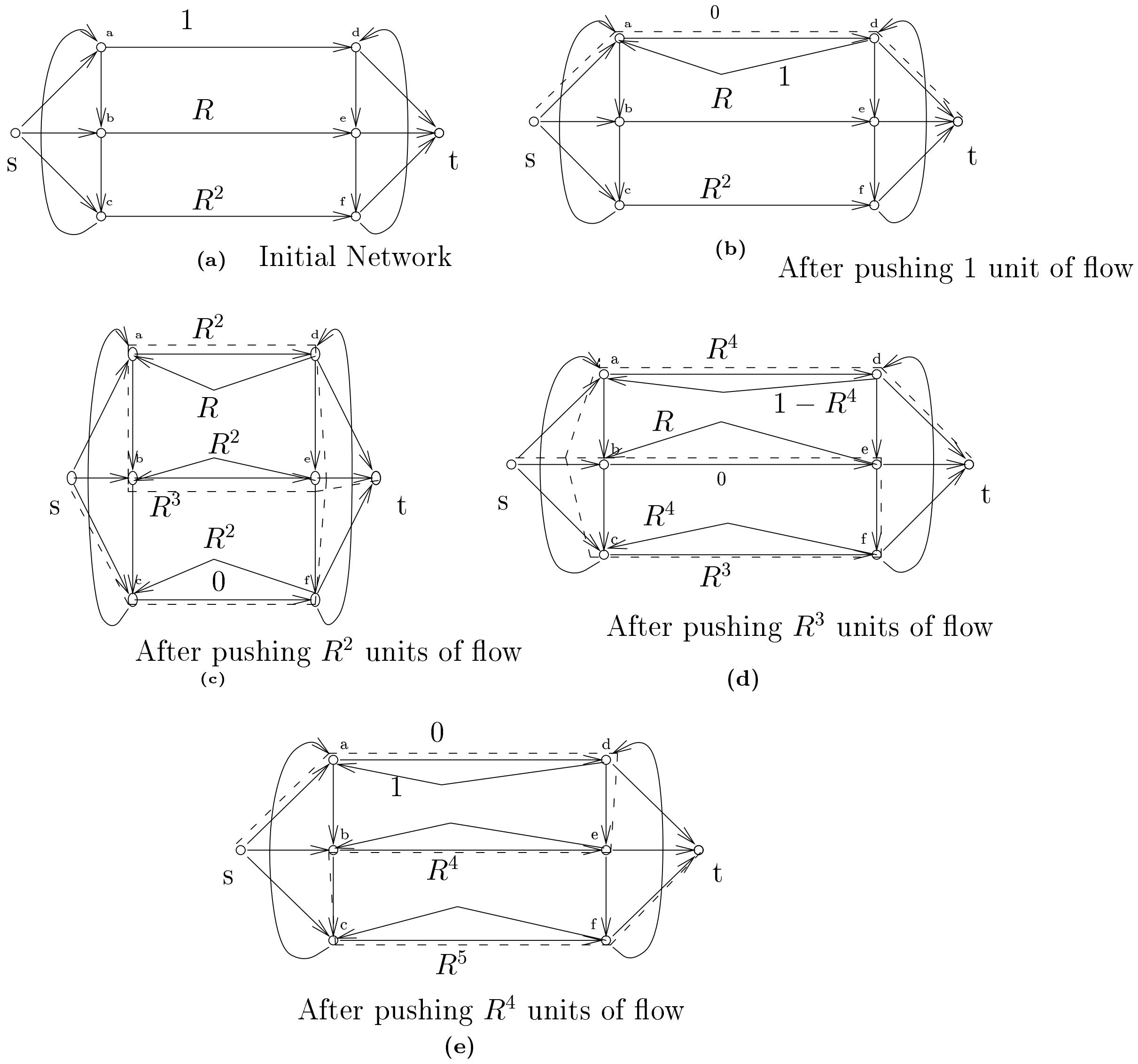


Figure 17: Non-terminating Example

CMSC 651 Advanced Algorithms	Lecture 09
Lecturer: Samir Khuller	Feb 26, 2002

Notes by Samir Khuller.

16 The Max Flow Problem

Today we will study the Edmonds-Karp algorithm that works when the capacities are integral, and has a much better running time than the Ford-Fulkerson method. (Edmonds and Karp gave a second heurisitc that we will study later.)

Assumption: Capacities are integers.

1st Edmonds-Karp Algorithm:

while (there is an augmenting path $s - t$ in G_f) **do**
    pick up an augmenting path (in G_f) with the highest residual capacity;
    use this path to augment the flow;

Analysis: We first prove that if there is a flow in G of value $|f|$, the highest capacity of an augmenting path in G_f is $\geq \frac{|f|}{m}$. In class we covered two different proofs of this lemma. The notion of flow decompositions is very useful so I am describing this proof in the notes.

Lemma 16.1 *Any flow in G can be expressed as a sum of at most m path flows in G and a flow in G of value 0, where m is the number of edges in G.*

Proof:
Let f be a flow in G. If $|f| = 0$, we are done. (We can assume that the flow on each edge is the same as the capacity of the edge, since the capacities can be artifically reduced without affecting the flow. As a result, edges that carry no flow have their capacities reduced to zero, and such edges can be discarded. The importance of this will become clear shortly.) Otherwise, let p be a path from s to t in the graph. Let $c(p) > 0$ be the bottleneck capacity of this path (edge with minimum flow/capacity). We can reduce the flow on this path by $c(p)$ and we output this flow path. The bottleneck edge now has zero capacity and can be deleted from the network, the capacities of all other edges on the path is lowered to reflect the new flow on the edge. We continue doing this until we are left with a zero flow ($|f| = 0$). Clearly, at most m paths are output during this procedure. □

Since the entire flow has been decomposed into at most m flow paths, there is at least one augmenting path with a capacity at least $\frac{|f|}{m}$.

Let the max flow value be f^*. In the first iteration we push at least $f_1 \geq \frac{f^*}{m}$ amount of flow. The value of the max-flow in the residual network (after one iteration) is at most

$$f^*(1 - 1/m).$$

The amount of flow pushed on the second iteration is at least

$$f_2 \geq (f^* - f_1)\frac{1}{m}.$$

The value of the max-flow in the residual network (after two iteations) is at most

$$f^* - f_1 - f_2 \leq f^* - f_1 - (f^* - f_1)\frac{1}{m} = f^*(1 - \frac{1}{m}) - f_1(1 - \frac{1}{m}) \leq f^*(1 - \frac{1}{m}) - \frac{f^*}{m}(1 - \frac{1}{m}).$$

$$= f^*(1 - \frac{1}{m})^2.$$

Finally, the max flow in the residual graph after the k^{th} iteration is

$$\leq f^*(\frac{m-1}{m})^k.$$

What is the smallest value of k that will reduce the max flow in the residual graph to 1?

$$f^*(\frac{m-1}{m})^k \leq 1 \ ,$$

Using the approximation $\log m - \log(m-1) = \Theta(\frac{1}{m})$ we can obtain a bound on k.

$$k = \Theta(m \log f^*).$$

This gives a bound on the number of iterations of the algorithm. Taking into a consideration that a path with the highest residual capacity can be picked up in time $O(m + n\log n)$, the overall time complexity of the algorithm is $O((m + n\log n)m\log f^*)$.

Tarjan's book gives a slightly different proof and obtains the same bound on the number of augmenting paths that are found by the algorithm.

History
Ford-Fulkerson (1956)
Edmonds-Karp (1969) $O(nm^2)$
Dinic (1970) $O(n^2m)$
Karzanov (1974) $O(n^3)$
Malhotra-Kumar-Maheshwari (1977) $O(n^3)$
Sleator-Tarjan (1980) $O(nm\log n)$
Goldberg-Tarjan (1986) $O(nm\log n^2/m)$

CMSC 651 Advanced Algorithms	Lecture 14
Lecturer: Samir Khuller	Th. Mar. 4, 1993

Notes by Marios Leventopoulos

17 An $O(n^3)$ Max-Flow Algorithm

The max-flow algorithm operates in phases. At each phase we construct the residual network G_f, and from it we find the layered network L_{G_f}.

In order to construct L_{G_f} we have to keep in mind that the shortest augmenting paths in G with respect to f correspond to the shortest paths from s to t in G_f. With a breadth-first search we can partition the vertices of G_f into *disjoint* layers according to their distance (the number of arcs in the shortest path) from s. Further more, since we are interested only in *shortest* $s-t$ paths in G_f we can discard any vertices in layers greater than that of t, and all other than t in the same layer as t, because they cannot be in any shortest $s-t$ path. Additionally a shortest path can only contain arcs that go from layer j to layer $j+1$, for any j. So we can also discard arcs that go from a layer to a lower layer, or any arc that joins two nodes of the same layer.

An augmenting path in a layered network with respect to some flow g is called *forward* if it uses no backward arcs. A flow g is called *maximal* (not *maximum*) if there are no **F**orward **A**ugmenting **P**aths (FAP).

We then find a maximal flow g in L_{G_f}, add g to f and repeat. Each time at least one arc becomes saturated, and leaves the net. (the backward arc created is discarded). Eventually s and t become disconnected, and that signals the end of the current phase. The following algorithm summarizes the above:

Step 1: $f = 0$

Step 2: Construct G_f

Step 3: Find *maximal* flow g in L_{G_f}

Step 4: $f \leftarrow f + g$

Step 5: goto step 2 (next phase)

In L_{G_f} there are no forward augmenting paths with respect to g, because g is maximal. Thus all augmenting paths have length greater than $s-t$ distance. We can conclude that the $s-t$ distance in the layered network is increasing from one phase to the other. Since it can not be greater than n, the number of phases is $O(n)$.

Throughput(v) of a vertex v is defined as the sum of the outgoing capacities, or the incoming capacities, whichever is smaller, i.e it is the maximum amount of flow that can be pushed through vertex v.

The question is, given a layered net G_f (with a source and sink node), how can we efficiently find a maximal flow g?

Step 1: Pick a vertex v with minimum throughput,

Step 2: Pull that much flow from s to v and push it from v to t

Step 3: Repeat until t is disconnected from s

By picking the vertex with the smallest throughput, no node will ever have to handle an amount of flow larger than its throughput, and hence no backtracking is required. In order to push flow from v to t we process nodes layer by layer in breadth-first way. Each vertex sends flow away by completely saturating each of its outgoing edges, one by one, so there is at most one outgoing edge that had flow sent on it but did not get saturated.

Each edge that gets saturated is not further considered in the current phase. We charge each such edge when it leaves the net, and we charge the node for the partially saturated edge.

The operation required to bring flow from s to v is completely symmetrical to pushing flow from v to t. It can be carried out by traversing the arcs backwards, processing layers in the same breadth-first way and processing the incoming arcs in the same organized manner.

To summarize we have used the following techniques:

1. Consider nodes in non-decreasing order of throughput
2. Process nodes in layers (i.e. in a breadth-first manner)
3. While processing a vertex we have at most one unsaturated edge (consider only edges we have sent flow on)

After a node is processed it is removed from the net, for the current phase.

Work done: We have $O(n)$ phases. At each phase we have to consider how many times different arcs are processed. Processing an arc can be either *saturating* or *partial*. Once an arc is saturated we can ignore it for the current phase. Thus saturated pushes take $O(m)$ time. However one arc may have more than one unsaturated push per phase, but note that the total number of pulls and pushes at each stage is at most n because every such operation results to the deletion of the node with the lowest throughput; the node where the this operation was started. Furthermore, for each push or pull operation, we have at most n partial steps, for each node processed. Therefore the work done for partial saturation at each phase is $O(n^2)$.

So we have $O(n^3)$ total work done for all phases.

CMSC 651 Advanced Algorithms	Lecture 6
Lecturer: Samir Khuller	Tu. Feb. 16, 1999

Notes by Samir Khuller.

18 Vertex Covers and Network Flow

We first show how to reduce the problem of finding a minimum weight vertex cover in a bipartite graph to the minimum cut problem in a directed flow network. Minimum cuts can be computed by finding maximum flows.

18.1 Reducing Bipartite Vertex Covers to Min Cuts

Let the bipartite graph G have vertex sets $X \cup Y$ and edge set E. We will construct a flow network as follows. Add a source s and a sink t. Direct all the edges from X to Y. Add edges (s, x) for each $x \in X$ with capacity $w(x)$ and edges (y, t) for each $y \in Y$ with capacity $w(y)$. Let the capacity of edges in E be a very high value M (say nW where W is the max vertex weight and n is the size of the graph).

Find a min-cut $(S, T = V - S)$ separating s from t (this can be done by finding a maxflow in the flow network and defining S to be the set of vertices reachable from s in the residual graph). Define the vertex cover as follows: $VC = (S \cap Y) \cup (T \cap X)$. We claim that the weight of the cut is exactly the same as the weight of the vertex cover VC. To see this, notice that the edges that cross the cut are the edges from s to $T \cap X$ and edges from $S \cap Y$ to t. These have weight exactly equal to the capacity of the cut. There can be no edges from $S \cap X$ to $T \cap Y$ since the capacity of each edge in E exceeds the capacity of the min-cut, hence VC is a vertex cover in $G = (X, Y, E)$.

To complete the proof, we need to argue that there is no vertex cover of smaller weight. To see this notice that *a min weight* vertex cover, induces a cut of exactly the same capacity as the weight of the cover. Hence the min cut corresponds to the min weight vertex cover.

Now recall that we can find a 2 approximation for vertex cover by reducing it to the problem of finding a minimum weight 2-cover in a graph. The minimum weight 2-cover can be found by reducing it to the problem of finding a min vertex cover in an appropriate bipartite graph.

Original notes by Vadim Kagan.

19 Planar Graphs

The planar flow stuff is by Hassin [10]. If you want to read more about planar flow, see the paper by Khuller and Naor [13].

A *planar embedding* of a graph, is a mapping of the vertices and edges to the plane such that no two edges cross (the edges can intersect only at their ends). A graph is said to be *planar*, if it has a planar embedding. One can also view planar graphs as those graphs that have a planar embedding on a sphere. Planar graphs are useful since they arise in the context of VLSI design. There are many interesting properties of planar graphs that can provide many hours of entertainment (there is a book by Nishizeki and Chiba on planar graphs [18]).

19.1 Euler's Formula

There is a simple formula relating the numbers of vertices (n), edges (m) and faces (f) in a connected planar graph.

$$n - m + f = 2.$$

One can prove this formula by induction on the number of vertices. From this formula, we can prove that a simple planar graph with $n \geq 3$ has a linear number of edges ($m \leq 3n - 6$).

Let f_i be the number of edges on face i. Consider $\sum_i f_i$. Since each edge is counted exactly twice in this summation, $\sum_i f_i = 2m$. Also note that if the graph is simple then each $f_i \geq 3$. Thus $3f \leq \sum_i f_i = 2m$. Since $f = 2 + m - n$, we get $3(2 + m - n) \leq 2m$. Simplifying, yields $m \leq 3n - 6$.

There is a linear time algorithm due to Hopcroft and Karp (based on DFS) to obtain a planar embedding of a graph (the algorithm also tests if the graph is planar). There is an older algorithm due to Tutte that finds a straight line embedding of a triconnected planar graph (if one exists) by reducing the problem to a system of linear equations.

19.2 Kuratowski's characterization

Definition: A *subdivision* of a graph H is obtained by adding nodes of degree two on edges of H. G contains H as a *homeomorph* if G contains a subdivision of H as a subgraph.

The following theorem very precisely characterizes the class of planar graphs. Note that K_5 is the complete graph on five vertices. $K_{3,3}$ is a complete bipartite graph with 3 nodes in each partition.

Theorem 19.1 (Kuratowski's Theorem) *G is planar if and only if G does not contain a subdivision of either K_5 or $K_{3,3}$.*

We will prove this theorem in the next lecture. Today, we will study flow in planar graphs.

19.3 Flow in Planar Networks

Suppose we are given a (planar) undirected flow network, such that s and t are on the same face (we can assume, without loss of generality, that s and t are both on the infinite face). How do we find max-flow in such a network (also called an st-planar graph)?

In the algorithms described in this section, the notion of a *planar dual* will be used: Given a graph $G = (V, E)$ we construct a corresponding planar dual $G^D = (V^D, E^D)$ as follows (see Fig. 18). For each face in G put a vertex in G^D; if two faces f_i, f_j share an edge in G, put an edge between vertices corresponding to f_i, f_j in G^D (if two faces share two edges, add two edges to G^D).

Note that $G^{DD} = G, |V^D| = f, |E^D| = m$, where f is the number of faces in G.

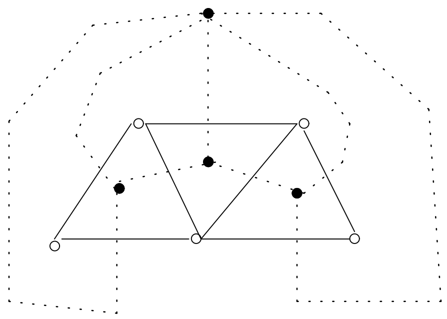


Figure 18: Graph with corresponding dual

19.4 Two algorithms

Uppermost Path Algorithm (Ford and Fulkerson)

1. Take the uppermost path (see Fig. 19).
2. Push as much flow as possible along this path (until the minimum residual capacity edge on the path is saturated).
3. Delete saturated edges.
4. If there exists some path from s to t in the resulting graph, pick new uppermost path and go to step 2; otherwise stop.

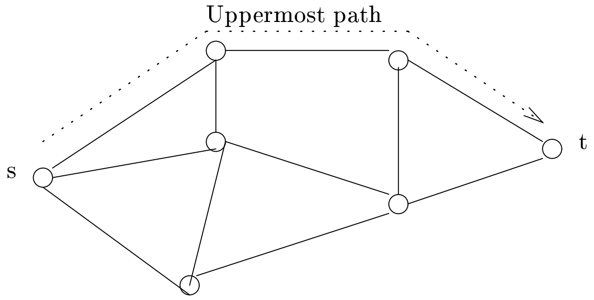


Figure 19: Uppermost path

For a graph with n vertices and m edges, we repeat the loop m times, so for planar graphs this is a $O(nm) = O(n^2)$ algorithm. We can improve this bound to $O(n \log n)$ by using heaps for min-capacity edge finding. (This takes a little bit of work and is left as an exercise for the reader.) It is also interesting to note that one can reduce *sorting* to flow in st-planar graphs. More precisely, any implementation of the uppermost path algorithm can be made to output the sorted order of n numbers. So implementing the uppermost path algorithm is at least as hard as sorting n numbers (which takes $\Omega(n \log n)$ time on a decision tree computation model).

We now study a different way of finding a max flow in an st-planar graph. This algorithm gets more complicated when the source and sink are not on the same face, but one can still solve the problem in $O(n \log n)$ time.

Hassin's algorithm

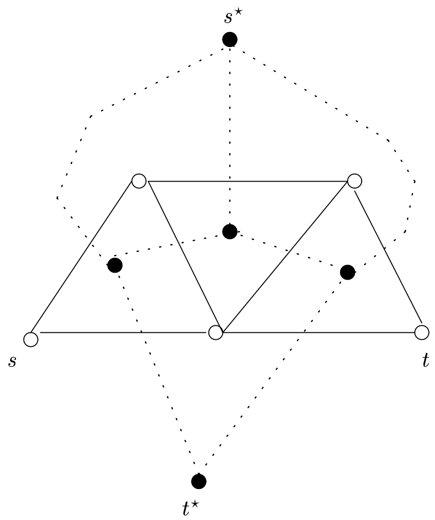


Figure 20: $G^{D\star}$

Given a graph G, construct a dual graph G^D and split the vertex corresponding to the infinte face into two nodes $s^\star$ and $t^\star$.

Node $s^\star$ is added at the top of G^D, and every node in G^D corresponding to a face that has an edge on the top of G, is connected to $s^\star$. Node $t^\star$ is similarly placed and every node in G^D, corresponding to a face that has an edge on the bottom of G, is connected to $t^\star$. The resulting graph is denoted $G^{D\star}$.

If some edges form an $s - t$ cut in G, the corresponding edges in $G^{D\star}$ form a path from $s^\star$ to $t^\star$. In $G^{D\star}$, the weight of each edge is equal to the capacity of the corresponding edge in G, so the min-cut in G corresponds to a shortest path between $s^\star$ and $t^\star$ in $G^{D\star}$. There exists a number of algorithms for finding shortest paths in planar graphs. Frederickson's algorithm, for example, has running time $O(n\sqrt{\log n})$. More recently, this was improved to $O(n)$, but the algorithm is quite complicated.

Once we have found the value of maxflow, the problem to find the exact flow through each individual edge still remains. To do that, label each node v_i in $G^{D\star}$ with d_i, v_i's distance from $s^\star$ in $G^{D\star}$ (as usual, measured along the shortest path from $s^\star$ to v_i). For every edge e in G, we choose the direction of flow in a way such that the larger label, of the two nodes in $G^{D\star}$ "adjacent" to e is on the right (see Fig. 21).

If d_i and d_j are labels for a pair of faces adjacent to e, the value of the flow through e is defined to be the difference between the larger and the smaller of d_i and d_j. We prove that the defined function satisfies the flow properties, namely that for every edge in G the flow through it does not exeed its capacity and for every node in G the amount of flow entering it is equal to the amount of flow leaving it (except for source and sink).

Suppose there is an edge in G having capacity $C(e)$ (as on Fig. 22), such that the amount of flow $f(e)$ through it is greater than $C(e)$: $d_i - d_j > C(e)$ (assuming, without loss of generality, that $d_i > d_j$).

By following the shortest path to v_j and then going from d_j to d_i across e, we obtain a path from $s^\star$ to v_i. This path has length $d_j + C(e) < d_i$. This contradicts the assumption that d_i was the shortest path.

To show that for every node in G the amount of flow entering it is equal to the amount of flow leaving it (except for source and sink), we do the following. We go around in some (say, counterclockwise) direction and add together flows through all the edges adjacent to the vertex (see Figure 23). Note that, for edge e_i, $d_i - d_{i+1}$ gives us negative value if e_i is incoming (i.e. $d_i < d_{i+1}$) and positive value if e_i is outgoing. By adding together all such pairs $d_i - d_{i+1}$, we get $(d_1 - d_2) + (d_2 - d_3) + \ldots + (d_k - d_1) = 0$, from which it follows that flow entering the node is equal to the amount of flow leaving it.

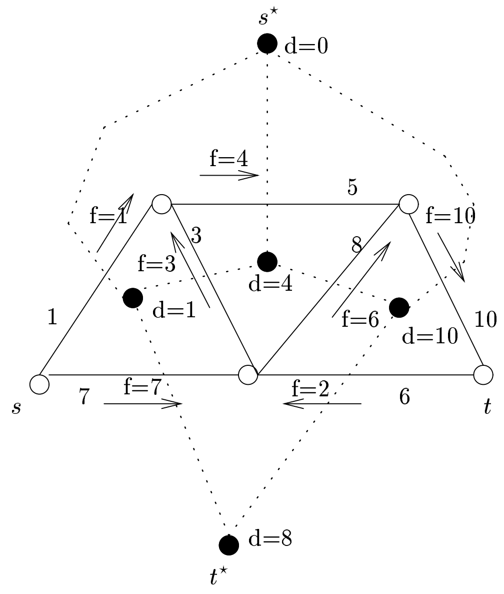


Figure 21: Labeled graph

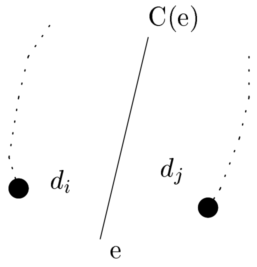


Figure 22:

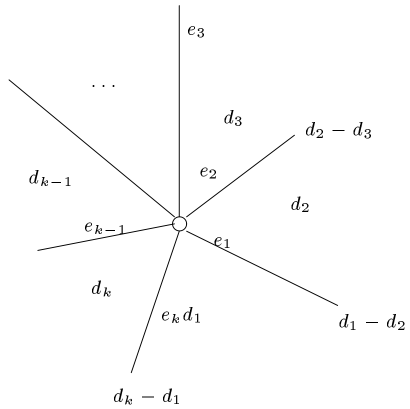


Figure 23:

CMSC 651 Advanced Algorithms	Lecture 21
Lecturer: Samir Khuller	Th. April 13, 1995

Original notes by Marsha Chechik.

20 Planar Graphs

In this lecture we will prove Kuratowksi's theorem (1930). For more details, one is referred to the excellent Graph Theory book by Bondy and Murty [3]. The proof of Kuratowski's theorem was done for fun, and will not be in the final exam. (However, the rest of the stuff on planar graphs is part of the exam syllabus.)

The following theorem is stated without proof.

Theorem 20.1 *The smallest non-planar graphs are K_5 and $K_{3,3}$.*

Note: Both K_5 and $K_{3,3}$ are embeddable on a surface with genus 1 (a doughnut).
Definition: *Subdivision* of a graph G is obtained by adding nodes of degree two on edges of G.
G contains H as a *homeomorph* if we can obtain a subdivision of H by deleting vertices and edges from G.

20.1 Bridges

Definition: Consider a cycle C in G. Edges e and e' are said to be on the same bridge if there is a path joining them such that no internal vertex on the path shares a node with C.

By this definition, edges form equivalence classes that are called bridges. A trivial bridge is a singleton edge.
Definition: A common vertex between the cycle C and one of the bridges with respect to C, is called *a vertex of attachment*.

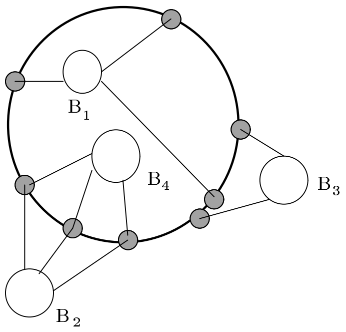
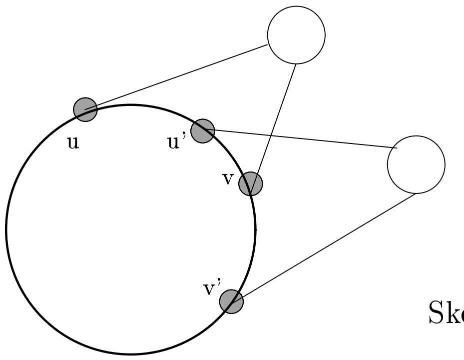


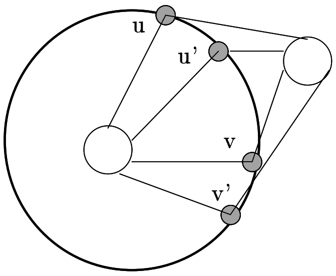
Figure 24: Bridges relative to the cycle C.

Definition: Two bridges *avoid* each other with respect to the cycle if all vertices of attachment are on the same segment. In Fig. 24, B_2 avoids B_1 and B_3 does not avoid B_1. Bridges that avoid each other can be embedded on the same side of the cycle. Obviously, bridges with two identical attachment points are embeddable within each other, so they avoid each other. Bridges with three attachment points which coincide (3-equivalent), do not avoid each other (like bridges B_2 and B_4 in Fig. 24). Therefore, two bridges either

1. Avoid each other
2. Overlap (don't avoid each other).
   - Skew: Each bridge has two attachment vertices and these are placed alternately on C (see Fig. 25).



Skew Bridges



Two bridges with 4 attachment vertices

Figure 25:

- 3-equivalent (like B_2 and B_4 in Fig. 24)

It can be shown that other cases reduce to the above cases. For example, a 4-equivalent graph (see Fig.25) can be classified as a skew, with vertices u and v belonging to one bridge, and vertices u' and v' belonging to the other.

20.2 Kuratowski's Theorem

Consider all non-planar graphs that do not contain K_5 or $K_{3,3}$ as a homeomorph. (Kuratowski's theorem claims that no such graphs exist.) Of all such graphs, pick the one with the least number of edges. We now prove that such a graph must always contain a Kuratowksi homeomorph (i.e., a K_5 or $K_{3,3}$). Note that such a graph is minimally non-planar (otherwise we can throw some edge away and get a smaller graph).

The following theorem is claimed without proof.

Theorem 20.2 *A minimal non-planar graph that does not contain a homeomorph of K_5 or $K_{3,3}$ is 3-connected.*

Theorem 20.3 (Kuratowski's Theorem) *G is planar iff G does not contain a subdivision of either K_5 or $K_{3,3}$.*

Proof:

If G has a subdivision of K_5 or $K_{3,3}$, then G is not planar. If G is not planar, we will show that it must contain K_5 or $K_{3,3}$. Pick a minimal non-planar graph G. Let (u, v) be the edge, such that $H = G - (u, v)$ is planar. Take a planar embedding of H, and take a cycle C passing through u and v such that H has the largest number of edges in the interior of C. (Such a cycle must exist due to the 3-connectivity of G.)

Claims:

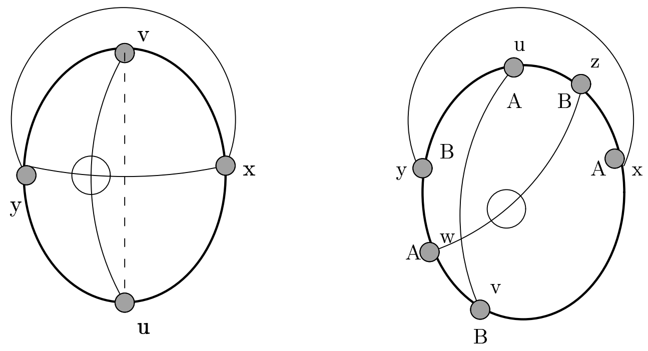
1. Every external bridge with respect to C has exactly two attachment points. Otherwise, a path in this bridge going between two attachment points, could be added to C to obtain more edges inside C.

2. Every external bridge contains exactly one edge. This result is from the 3-connectivity property. Otherwise, removal of attachment vertices will make the graph disconnected.

3. At least one external bridge must exist, because otherwise the edge from u to v can be added keeping the graph planar.

4. There is at least one internal bridge, to prevent the edge between u and v from being added.

5. There is at least one internal and one external bridge that prevents the addition of (u, v) and these two bridges do not avoid each other (otherwise, the inner bridge could be moved outside).



a) This graph contains K_5 homeomorph b) This graph contains $K_{3,3}$ homeomorph

Figure 26: Possible vertex-edge layouts.

Now, let's add (u, v) to H. The following cases is happen:

1. See Fig. 26(a). This graph contains a homeomorph of K_5. Notice that every vertices are connected with an edge.

2. See Fig. 26(b). This graph contains a homeomorph of $K_{3,3}$. To see that, let $A = \{u, x, w\}$ and $B = \{v, y, z\}$. Then, there is an edge between every $v_1 \in A$ and $v_2 \in B$.

The other cases are similar and will not be considered here. $\square$

CMSC 651 Advanced Algorithms	Lecture 18
Lecturer: Samir Khuller	Tu. March 30, 1993

Notes by Vadim Kagan.

21 Graph Coloring

Problem : Given a graph $G = (V, E)$, assign colors to vertices so that no adjacent vertices have the same color.

Note that all the vertices of the same color form an independent set (hence applications in parallel processing).

Theorem 21.1 (6-colorability) *If $G = (V, E)$ is planar,then G is 6-colorable.*

Proof:

Main property: there is a vertex in G with degree ≤ 5. Suppose all the vertices in G have degree ≥ 6. Then since

$$|E| = \frac{\sum_V deg(V)}{2}$$

we get $|E| \geq 3n$ which contradicts the fact that for planar graphs $|E| \leq 3n - 6$ for $n \geq 3$. So there is (at least one) vertex in G with degree ≤ 5.

Proof of 6-colorability of G is done by induction on the number of vertices. Base case is trivial, since any graph with ≤ 6 vertices is clearly 6-colorable. For inductive case, we assume that any planar graph with n vertices is 6-colorable and try to prove that the same holds for a graph G with $n + 1$ vertices.
Let us pick a minimum-degree vertex v and delete it from G. The resulting graph has n vertices and is 6-colorable by the induction hypothesis. Let C be some such coloring. Now let's put v back in the graph. Since v was the minimal-degree vertex, the degree of v was at most 5 (from the lemma above). Even if all the neighbours of v are assigned different colors by C, there is still one unused color left for v, such that in the resulting graph no two adjacent vertices have the same color. This gives an $O(n)$ algorithm for 6 coloring of planar graphs. □

We will now prove a stronger property about planar graph coloring.

Theorem 21.2 (5-colorability of planar graphs) *Every planar graph is 5-colorable.*

Proof:

We will use the same approach as for the 6-colorability. Proof will again be on induction on the number of vertices in the graph. The base case is trivial. For inductive case, we again find the minimum degree vertex. When we delete it from the graph, the resulting graph is 5-colorable by induction hypothesis. This time, though, putting the deleted vertex back in the graph is more complicated than in 6-colorability case. Let us denote the vertex that we are trying to put back in the graph by v. If v has ≤ 4 neighbours, putting it back is easy since one color is left unused and we just assign v this color. Suppose v has 5 neighbours (since v was the minimum-degree vertex in a planar graph, its degree is ≤ 5) and all of them have different colors (if any two of v's neighbours have the same color, there is one free color and we can assign v this color). To put v back in G we have to change colors somehow (see Fig. 27). How do we change colors? Let us concentrate on the subgraph of nodes of colors 2 and 4. Suppose colors 2 and 4 belong to different connected components. Then flip colors 2 and 4 in the component that vertex 4 belongs to, i.e., change color of all 4-colored nodes to color 2 and nodes with color 2 to color 4. Since colors 2 and 4 are in the different connected components, we can do this without violating the coloring (in the resulting graph, all the adjacent nodes will still have different colors). But now we have a free color - namely, color 4 - that we can use for v.
What if colors 2 and 4 are in the same component? In this case, there is a path P of alternating colors (as shown on Fig. 27).

Note that node 3 is now "trapped" - any path from node 3 to node 5 crosses P. If we now consider a subgraph consisting of 3-colored and 5-colored nodes, nodes 3 and 5 cannot be in the same connected component - it would violate planarity of G. So we can flip colors 3 and 5 (as we did for 2 and 4) and get one free color that could be used for v; this concludes the proof of 5-colorability of planar graphs. □

Figure 27: Graph with alternating colors path

CMSC 651 Advanced Algorithms	Lecture 17
Lecturer: Bill Gasarch	Th, Mar. 25, 1993

Notes by Michael Tan and Samir Khuller.

22 Graph Minor Theorem and other CS Collectibles

In this lecture, we discuss graph minors and their connections to polynomial time computability.

Definition 22.1 (minor) *We say that H is a* **minor** *of G ($H \leq G$) if H can be obtained from G by edge removal, vertex removal and edge contraction.*

Definition 22.2 (closed under minor) *Given a class of graphs $\mathcal{C}$, and graphs G and H, $\mathcal{C}$ is closed under taking minors if $G \in \mathcal{C}$ and $H \leq G$ implies $H \in \mathcal{C}$.*

An example of a class of graphs that is closed under taking minors is the class of planar graphs. (If G is planar, and H is a minor of G then H is also a planar graph.)

Theorem 22.3 (Graph Minor Theorem Corollary) $[Robertson - Seymour]$ *If $\mathcal{C}$ is any class of graphs closed under taking minors, then there exists a finite obstruction set $H_1, \ldots, H_l$ such that $G \in \mathcal{C} \leftrightarrow (\forall i)[H_i \not\leq G]$.*

The proof of this theorem is highly non-trivial and is omitted.
Example: For genus 1 (planar graphs), the finite obstruction set is K_5 and $K_{3,3}$. G is planar if and only if K_5 and $K_{3,3}$ are not minors of G. Intuitively, the "forbidden" minors (in this case, K_5 and $K_{3,3}$) are the minimal graphs that are not in the family $\mathcal{C}$.

Theorem 22.4 *Testing $H \leq G$, for a fixed H, can be done in time $O(n^3)$, if n is the number of vertices in G.*

Notice that the running time does not indicate the dependence on the size of H. In fact, the dependence on H is huge! There is an embarassingly large constant that is hidden in the big-O notation. This algorithm is far from practical for even very small sizes of H.
Some results of the above statements:
1. If $\mathcal{C}$ ={all planar graphs}, we can test $G \in \mathcal{C}$ in O(n^3) time. (To do this, test if K_5 and $K_{3,3}$ are minors of G. Each test takes O(n^3) time.)

2. If $\mathcal{C}$ = {all graphs in genus k} (fixed k), we can test if ($G \in \mathcal{C}$) in O(n^3) time. (To do this, test if $H_i \leq G (i = 1..L)$ for all graphs of $\mathcal{C}$'s obstruction set. The time to do this is O($L * n^3$) = O(n^3), since L is a constant for each fixed k.)

3. We can solve Vertex Cover for fixed k in POLYNOMIAL time. (The set of all graphs with (vertex cover size) $\leq k$, is closed under taking minors. Therefore, it has a finite obstruction set. We can test a given graph G as we did in 2.)

22.1 Towards an algorithm for testing planarity in genus k

We can use a trustworthy oracle for SAT to find a satisfying assignment for a SAT instance (set a variable, quiz the oracle, set another variable, quiz the oracle, ...). The solution will be guaranteed. With an untrustworthy oracle, we may use the previous method to find a satisfying assignment, but the assignment is not guaranteed to be correct. However, this is something that we can test for in the end. We use this principle in the algorithm below.

The following theorem is left to the reader to verify (see homework).

Theorem 22.5 *Given a graph G and an oracle that determines planarity on a surface with genus k, we can determine if G is planar in genus k and find such a planar embedding in polynomial time.*

We will now use this theorem, together with the graph minor theorem to obtain an algorithm for testing if a graph can be embedded in a surface of genus k (fixed k).

ALGORITHM for testing planarity in genus k and giving the embedding: (uses an untrustworthy "Planarity in k" Oracle)

The oracle uses a "current" list H of forbidden minors to determine if a given graph is planar or non-planar. The main point is that if the oracle is working with an incomplete list of forbidden minors, and lies about the planarity of a graph then in the process of obtaining a planar embedding we will determine that the oracle lied. However, some point of time we will have obtained the correct list of forbidden minors (even though we will not know that), the oracle does not lie thereafter. Since there is only a finite forbidden list, this will happen after constant time.

1. Input(G)
2. For $i := 1$ to ∞
2.1 - generate all graphs $F_1, F_2, ...F_i$.
2.2 - for every graph f in $F_1..F_i$
    - Check if f is planar (in k) (using brute force to try all possible embeddings of f)
    - if f is not planar, put f in set H [We are constructing the finite obstruction set]
2.3 - test if $(H_1 \leq G), (H_2 \leq G), ...(H_l \leq G)$
2.4 - if any $H \leq G$, then RETURN(G IS NOT PLANAR)
2.5 - else
2.5.1 - try to make some embedding of G (We do this using H, which gives us an untrustworthy oracle to determine if a graph is planar in k. There exists an algorithm (see homework) which can determine a planar embedding using this PLANARITY oracle.)
2.5.2 - if we can, RETURN (the planar embedding, G IS PLANAR);
    - else keep looping (H must not be complete yet)

Running time:
The number of iterations is bounded by a constant number (independent of G) since at the point when the **finite** obstruction set is completely built, statement 2.4 or 2.5.2 MUST suceed.

22.2 Applications:

Since we know VC for fixed k is in P, a good polynomial algorithm may exist. Below are several algorithms :

Algorithm 1 [Fellows]:

```
build a tree as follows:
- find edge (u,v) which is uncovered
- on left branch, insert u into VC, on right branch insert v into VC
- descend, recurse


                  . VC={}, (u,v) is uncovered
                 / \
```

```
    VC = {u} /   \  VC = {v}
              .      .
             / \   / \
VC={u,v4}/   \ /   \  VC = {v, v2}
         etc.
      . . . . . . . . .
```

Build tree up to height k. There will be O(2^k) leaves. If there exists a vertex cover of size $\leq k$, then it will be in one of these leaves. Total time is $O(n * 2^k) = O(n)$.

A second simpler algorithm is as follows: any vertex that has degree $> k$ must be in the VERTEX COVER (since if it is not, all its neighbours have to be in any cover). We can thus include all such nodes in the cover (add them one at a time). We are now left with a graph in which each vertex has degree $\leq k$. In this graph we know that there can be at most a constant number of edges (since a vertex cover of constant size can cover a constant number of edges in this graph). The problem is now easy to solve.

Notes by Samir Khuller.

23 The Min Cost Flow Problem

Today we discuss a generalization of the min weight perfect matching problem, the max flow problem and the shortest path problem called the min cost flow problem. Here you have a flow network, which is basically a directed graph. Each edge (i, j) has a cost c_{ij} and capacity u_{ij}. This is denoted by a tuple (c_{ij}, u_{ij}) for the edge (i, j). Nodes v have supplies/demands $b(v)$. If $b(v) > 0$ then v is a supply vertex with supply value $b(v)$. If $b(v) < 0$ then v is demand vertex with demand $-b(v)$.

Our goal is to find a flow function $f : E \rightarrow \Re$ such that each vertex has a net outflow of $b(v)$ and the total cost is minimized. As before, the flow on an edge cannot exceed the corresponding capacity. The *cost* of the flow function is defined as $\sum_{e \in E} f(e) \cdot c(e)$. (In other words, supply nodes have a net outflow and demand nodes have a net inflow equal to their $|b(v)|$ value.)

We will assume that the total supply available is exactly equal to the total demand in the network. (In general, for a feasible solution to exist we need to assume that the total supply is at least the total demand. We can add a "dummy" vertex that absorbs the surplus flow, and thus can assume that the total supply and demand values are the same.)

The reader can easily see that the shortest path problem is a very special case when we need to ship one unit of flow from s to t at min cost, where each edge has capacity one. The min weight perfect matching problem is also a special case when each vertex on one side has $b(i) = 1$ and the vertices on the other side have $b(i) = -1$. When all edges have cost zero, it clearly models the max flow problem.

We first discuss an application of this problem to show how powerful it is.

Caterer problem: A caterer needs to provide d_i napkins on each of the next n days. He can buy new napkins at the price of α cents per napkin, or have the dirty napkins laundered. Two types of laundry service are available: regular and express. The regular service requires two working days and costs β cents per napkin, the express service costs γ cents per napkin and requires one working day. The problem is to compute a purchasing/laundring policy at min total cost.

This problem can be formulated as a min cost flow problem and can handle generalizations like inventory costs etc.

The basic idea is to create a source vertex s that acts as a supply node of new napkins. For each day $i, \leq i \leq n$, we create two vertices x_i and y_i. We set $b(x_i) = -d_i$ so this is a demand vertex with the demand equal to the requirement for day i. We set $b(y_i) = d_i$ so this is a supply vertex that can provide the dirty napkins to future days. We set $b(s) = \sum_{i=1}^{n} d_i$. From vertex s there is an edge to each x_i vertex with parameters (α, d_i). From $y_i, 1 \leq i \leq n-2$ there is an edge to x_{i+2} with parameters (γ, d_i). From $y_i, 1 \leq i \leq n-3$ there is an edge to x_{i+3} with parameters (β, d_i). These edges denote the fact that we can launder napkins by using the express service and provide them after a gap of one day at cost γ per napkin, and launder napkins by using the regular service and provide them after a gap of two days at a cost of β per napkin. The total number of such napkins is clearly upper bounded by d_i (number of dirty napkins on day i). Finding a min cost flow, gives us the way to find a min cost solution. A formal proof of this is left to the reader. You should note that we need to create two vertices for each day since one node absorbs the flow (clean napkins) and one provides the flow (used napkins). We also create one sink vertex t that can absorb the surplus napkins at zero cost. (Since the total supply in the network exceeds the demand.)

Residual Graph: We define the concept of residual graphs G_f with respect to a flow function f. The capacity function is the same as before. The cost of the reverse edge is $-c_{ij}$ if the cost of edge (i, j) is c_{ij}. This corresponds to the fact that we can reduce the cost by pushing flow along this edge, since this only reduces the flow that we were pushing in the forward direction.

We now discuss a very simple min cost flow algorithm. We can first compute a feasible solution, by ignoring costs and solving a max flow problem. We create a single source vertex s, add edges from s to each supply node v with capacity $b(v)$. We create a single sink vertex t, and add edges from each demand vertex

v to t with capacity $|b(v)|$. If the max flow does not saturate all the edges coming out of the source then there is no feasible solution.

Unfortunately, this solution may have a very high cost! We now show how to reduce the cost of the max flow. Construct the residual graph, and check to see if this has a negative cost cycle (one can use a Bellman-Ford shortest path algorithm for detecting negative cost cycles). If it does have a negative cost cycle we can push flow around the cycle until we cannot push any more flow on it. Pushing a flow of δ units around the cycle reduces the total flow cost by $\delta|C|$ where C is the cost of the cycle. If there is no negative cost cycle in G_f then we can prove that f is an optimal solution. Notice that pushing flow around a cycle does not change the net incoming flow to any vertex, so all the demands are still satisfied. We only reduce the cost of the solution.

The next question is: how do we pick a cycle around which to cancel flow? Goldberg and Tarjan proved that picking a min mean cycle[2] gives a polynomial bound on the total number of iterations of the algorithm. One can also try to find the cycle with the least cost, but then one might only be able to push a small amount of flow around this cycle. One could also try to identify the cycle which would reduce the cost by the maximum possible amount.

Lemma 23.1 *A flow f is a min cost flow if and only if G_f has no negative cost cycles in it.*

Proof:

If G_f has a negative cost cycle, we can reduce the cost of flow f, hence it was not an optimal flow. If G_f has no negative cost cycles, we need to argue that f is an optimal solution. Suppose f is not optimal, then there is a flow f^* that has the same flow value, but at lower cost. Consider the flow $f^* - f$; this is a set of cycles in G_f. Hence f can be converted to f^* by pushing flow around this set of cycles. Since these cycles are non-negative, this would only increase the cost! Hence $c(f) \leq c(f^*)$. Since f^* is an optimal flow, the two must have the same cost. $\square$

In the next lecture we will study a different min cost flow algorithm.

[2] A min mean cycle minimizes the ratio of the cost of the cycle to the number of edges on the cycle.

CMSC 651 Advanced Algorithms	Lecture 17
Lecturer: Samir Khuller	Apr 2, 2002

Notes by Samir Khuller.

24 Shortest Path based Algorithm for Min Cost Flows

Recall that we have a flow network, which is basically a directed graph. Each edge (i,j) has a cost c_{ij} and capacity u_{ij}. This is denoted by a tuple (c_{ij}, u_{ij}) for the edge (i,j). Nodes v have supplies/demands $b(v)$. If $b(v) > 0$ then v is a supply vertex with supply value $b(v)$. If $b(v) < 0$ then v is demand vertex with demand $-b(v)$. As before we assume that the total supply equals the total demand.

Our goal is to find a flow function $f : E \rightarrow \Re$ such that each vertex has a net outflow of $b(v)$ (in other words, supply nodes have a net outflow and demand nodes have a net inflow equal to their $|b(v)|$ value) and the total cost is minimized. As before, the flow on an edge cannot exceed the corresponding capacity. The *cost* of the flow function is defined as $\sum_{e\in E} f(e) \cdot c(e)$.

24.1 Review of Shortest Paths

A shortest path *tree* is a rooted tree with the property that the unique path from the root to any vertex is a shortest path to that vertex. (Notice that Dijkstra's algorithm implicitly computes such a tree while computing shortest paths.)

Shortest Path Verification: Given any rooted tree T, how does one verify that it is a shortest path tree? Suppose we use distances in the tree to compute distance labels $d(i)$ for each vertex i. For each directed edge (i,j) in the graph, we check the condition

$$c_{ij} + d(i) \geq d(j).$$

If this is true for all edges, we claim that T encodes shortest paths. It is easy to see that if for some edge $d(j) > d(i) + c_{ij}$ then we can find a shorter path from s to j by going through vertex i.

Consider any path $P = [s = v_0, v_1, v_2, \ldots, v_k]$ from s to v_k. $c(P) = \sum_{(v_i,v_{i+1})} c_{v_iv_{i+1}} \geq \sum_{(v_i,v_{i+1})}(d(i+1) - d(i)) \geq d(v_k) - d(v_0) \geq d(v_k)$. Hence path P is longer than the path in the tree from s to v_k.

We can define *reduced costs of edges* as follows: $c'_{ij} = c_{ij} + d(i) - d(j)$. It is easy to see that $c'_{ij} \geq 0$ and $c'_{ij} = 0$ for edges on a shortest path.

24.2 Min Cost Flows

Similar to the concept of reduced costs via distance labels, we can define the concept of reduced costs via *node potentials*. We define a function $\pi(i)$ that denotes the potential of vertex i. We define the reduced cost of edges in the flow network as $c^{\pi}_{ij} = c_{ij} + \pi(i) - \pi(j)$.

The proof of the following lemma is trivial.

Lemma 24.1 *For any path P from vertex k to ℓ, we have*

1. $\sum_{(i,j)\in P} c^{\pi}_{ij} = \sum_{(i,j)\in P} c_{ij} + \pi(k) - \pi(\ell)$.
2. *for any directed cycle W,* $\sum_{(i,j)\in W} c^{\pi}_{ij} = \sum_{(i,j)\in W} c_{ij}$.

Lemma 24.2 *(Reduced Cost Optimality) A feasible flow x is optimal if and only if there exist node potentials π such that $c^{\pi}_{ij} \geq 0$ for all edges in G_x.*

Proof:

We recall that a feasible flow x is optimal if and only if there are no negative cost cycles in G_x. If the flow is optimal, we consider G_x and define $\pi(i) = d(i)$ where $d(i)$ is the distance to i from an arbitrary vertex s (since the residual graph has no negative cost cycles, distances are well defined). By the shortest path property all edges have non negative reduced costs.

To prove the converse we see that if such node potentials exist, and we consider the total cost of any cycle W; by the previous lemma, the total length of this cycle is non-negative and the same as the total length of the cycle with the original cost function. Hence the residual graph has no negative cycles and the solution is optimal. $\square$

We define a pseudoflow f, as a flow function that meets the capacity requirement, but may not meet the demand(s). One can view this as a partial solution. The main idea will be to augment flow along shortest paths from a supply vertex to a demand vertex. We will eventually meet the full demand. We start with the zero flow as the pseudoflow, and in each iteration decrease the remaining total demand. In the end, we will have a flow function that satisfies reduced cost optimality.

The algorithm is based on the following two lemmas.

Lemma 24.3 *Let x be a pseudoflow that satisfies reduced cost optimality conditions with respect to node potentials π. Let vector d represent shortest path distances from a vertex s to all other nodes in G_x with respect to cost c_{ij}^{π}. The following properties are valid:*

1. *Pseudoflow x also satisfies reduced cost optimality with respect to node potentials $\pi' = \pi + d$.*
2. *The reduced costs $c_{ij}^{\pi'}$ are zero for all edges on shortest paths from s to nodes in G_x with costs c_{ij}^{π}.*

Proof:

We first show that $c_{ij}^{\pi'} \geq 0$. By definition, $c_{ij}^{\pi'} = c_{ij} + \pi'(i) - \pi'(j) = c_{ij} + \pi(i) + d(i) - \pi(j) - d(j) = c_{ij}^{\pi} + d(i) - d(j) \geq 0$ (by the shortest path property).

The last equation is met with equality for edges on shortest paths, so $c_{ij}^{\pi'} = 0$. $\square$

Lemma 24.4 *Suppose x is a pseudoflow satisfying reduced cost optimality. If we obtain x' from x by sending flow along a shortest path from s to some other demand node k, then x' also satisfies reduced cost optimality.*

Proof:

If we define potentials π' from π such that x satisfies reduced cost optimality with respect to both node potentials. When we saturate a shortest path, the reduced cost of this path is zero and we create edges in the residual graph in the opposite direction with zero cost. These still satisfy reduced cost optimality. $\square$

We can now discuss the algorithm. The main idea is to start with a pseudoflow of zero initially, and $\pi(i) = 0$ for each vertex. Since the initial costs are non-negative, the edges satisfy reduced cost optimality. We compute the new potential π' and saturate flow on a zero cost path from a supply to a demand vertex. (Pick the source s as a supply vertex to compute shortest paths.) We then recompute the new shortest paths, update the potentials and saturate the next zero cost shortest path from a supply to a demand node. In each step we update the excess remaining flow in the supply/demand nodes. The algorithm terminates when we convert our pseudoflow into a real flow.

```
proc Min cost flow(G);
    x = 0; π = 0; ∀i  e(i) = b(i);
    E = {i|e(i) > 0}; D = {i|e(i) < 0};
    while E ≠ ∅ do
        Select a node k ∈ E, ℓ ∈ D;
        Find shortest distances d(j) from k to nodes in G_x with edge costs c^π;
        ∀i   π(i) = π(i) + d(i);
        Let P be a shortest path from k to ℓ;
        δ = min(e(k), e(ℓ), min(r_ij, (i, j) ∈ P));
        augment δ units of flow along P, update x, G_x, E, D etc.
    end-while
end proc;
```

CMSC 651 Advanced Algorithms	Lecture 24
Lecturer: Samir Khuller	Th. Apr 25, 2002

Notes by Samir Khuller.

25 NP-completeness

To really understand NP-completeness, you will need to read Chapter 36. This handout summarizes the basic concepts we discussed in the first class on NP-completeness.

For the last few months, we have seen many different problems, all of which could be solved fairly efficiently (we were able to design fast algorithms for them). We are going to define an *efficient algorithm* as one that runs in worst case polynomial time – $O(n^c)$ for an input of size n, where c is a fixed constant. However, there are many problems for which the best known algorithms take exponential time in the worst case – these problems arise in various contexts, graphs, layout, scheduling jobs, logic, compiler optimization etc. to name a few.

To make life simple, we will concentrate on rather simple versions of these problems, which are the decision versions of these problems. For example, rather than asking for the shortest path between a pair of vertices in an unweighted graph, we could ask the question: given a graph G and an integer K, is there a path from s to v of length at most K? This question has a simple YES/NO answer. Note that we do not even ask for the path itself! (This was just to illustrate the concept of decision problems, this particular problem is not NP-complete! In fact, we could run BFS, find out the shortest path length from s to v and then output YES/NO appropriately.) It should be clear that if we wanted to determine the length of the shortest path, we could ask "Is there a path of length 1", if the answer was "NO" we could ask "Is there a path of length 2" and so on until we determine the length of the shortest path. Since the path length varies from 0 to $n-1$, asking n questions is sufficient. (Notice that we could also use binary search to find the length of the shortest path.)

Let $\mathcal{P}$ be the class of problems that can be solved in polynomial time. In other words, given an instance of a decision problem in this class, there is a polynomial time algorithm for deciding if the answer is YES/NO.

Let $\mathcal{NP}$ denote the class of problems for which we can *check* a proposed YES solution in polynomial time. In other words, if I claim that there is a path of length at most K between s and v and I can give you a proof that you can check in polynomial time, we say that the problem belongs to $\mathcal{NP}$. Clearly, this problem is in $\mathcal{NP}$, but its also in $\mathcal{P}$. There are many other problems in $\mathcal{NP}$, that are not known to be in $\mathcal{P}$. One example is LONGEST PATH: given a graph G, and an integer L, is there a simple path of length *at least* L from s to v? Clearly, this problem is in $\mathcal{NP}$, but not known to be in $\mathcal{P}$. (To show that it is in $\mathcal{NP}$, I can give you a path of length at least L and in polynomial time it can be checked for validity – namely, that it has length at least L and is a simple path.) Usually we do need to argue that problem is in $\mathcal{NP}$ since there are problems that are not in $\mathcal{NP}$ (more on this later).

Reductions: We say problem $X \propto Y$ (X reduces to Y) if we can show the following: if there is a polynomial time algorithm for problem Y and we can use it to design a polynomial time algorithm for problem X, then X can be reduced to solving Y.

NP-completeness: A problem Q is said to be NP-complete, if we can prove (a) $Q \in \mathcal{NP}$ and (b) $\forall X \in \mathcal{NP} \quad X \propto Q$.

The notion of NP-completeness tries to capture the "hardest" problems in the class $\mathcal{NP}$. In other words, the way we have defined the class of NP-complete problems, if *ever* we were able to find a polynomial time algorithm for Q, an NP-complete problem, we can then find a polynomial time algorithm for *any* problem X that is in the class $\mathcal{NP}$ (and hence for every NP-complete problem). This follows by the definition of NP-completeness.

The next question is: do NP-complete problems even exist? At first glance it appears pretty hard to show that there is a general way to reduce ANY problem in $\mathcal{NP}$ to problem Q. We need to do this to prove that

Q is NP-complete. Cook did precisely that in 1971. He showed that the following problem is NP-complete.

SATISFIABILITY PROBLEM: Given n boolean variables $X_1, \ldots, X_n$ (each can be TRUE or FALSE), is there a way to assign values to the variables so that a given boolean formula evaluates to TRUE? The boolean formula could take the form:
$(X_i \cup \overline{X_j} \cup X_k) \cap (\overline{X_p} \cup X_m \cup \overline{X_k}) \cap (X_j) \cap (X_j \cup X_i) \ldots \cap (X_p \cup X_k)$.

Cook's theorem: COOK proved that the SATISFIABILITY problem is NP-complete.

It is easy to see that the problem belongs to the class $\mathcal{NP}$. Understanding the proof requires knowledge of Turing Machines and a more formal description of what an algorithm is.

Now that we have one problem Q that is NP-complete, our task is much easier. If we can prove that $Q \propto Z$ AND $Z \in \mathcal{NP}$ then we claim that Z is NP-complete. If we show that $Q \propto Z$ then we have shown that if Z can be solved in polynomial time then so can Q. But if Q can be solved in polynomial time and since it is NP-complete, we can solve any problem $X \in \mathcal{NP}$ in polynomial time! Thus we have shown $\forall X \in \mathcal{NP}$ $X \propto Z$, and thus established that Z is NP-complete.

We now discuss a few specific NP-complete problems. All of these problems can be proven to be NP-complete by reducing SATISFIABILITY (a known NP-complete problem) to each of them. The proofs for some of them are complex. For fun, I will outline the reduction from SATISFIABILITY to CLIQUE in class.

HAMILTONIAN CYCLE PROBLEM: Given a graph $G = (V, E)$ is there a cycle that visits each vertex exactly once? (Notice that the closely related Euler tour problem, where we desire a tour (or cycle) that visits each edge exactly once can be solved in polynomial time.)

CLIQUE PROBLEM: Given a graph $G = (V, E)$ and an integer K, is there a set $S \subset V$ such that $|S| = K$ and there is an edge between each pair of nodes in S.

TRAVELING SALESMAN PROBLEM: Given a matrix D of distances between n cities (assume entries in D are integers for convenience), and an integer L – Is there a traveling salesman tour of length at most L? (A traveling salesman tour visits each city exactly once.)

BIN PACKING: Given n items, each with a size s_i, and an integer K – can we pack the items into K bins of unit size?

Notice that there are problems that may not be in $\mathcal{NP}$. Given a graph G, and an integer L, is there exactly one path of length L from s to v? Even if i claim that the answer is YES, and show you a path of length L – how do I convince you that there is no other path of length L? So, we dont know if this problem is in $\mathcal{NP}$. I just wanted to point out that there are problems for which we cannot even check a proposed solution in polynomial time (at least for now). Maybe some day, we will be able to figure out a polynomial time algorithm (someone from this class might be able to!).

CMSC 651 Advanced Algorithms	Lecture 23
Lecturer: Samir Khuller	Th. Apr. 20, 1995

Notes by Samir Khuller.

26 NP-Completeness

In the last lecture we showed every non-deterministic Turing Machine computation can be encoded as a SATISFIABILITY instance. In this lecture we will assume that every formula can be written in a restricted form (3-CNF). In fact, the form is $C_1 \cap C_2 \cap \ldots \cap C_m$ where each C_i is a "clause" that is the disjunction of exactly three "literals". (A literal is either a variable or its negation.) The proof of the conversion of an arbitrary boolean formula to one in this form is given in [5].

The reductions we will study today are:

1. SAT to CLIQUE.
2. CLIQUE to INDEPENDENT SET.
3. INDEPENDENT SET to VERTEX COVER.
4. VERTEX COVER to DOMINATING SET.
5. SAT to DISJOINT CONNECTING PATHS.

The first three reductions are taken from Chapter 36.5 [5] pp. 946–949.

Vertex Cover Problem: Given a graph $G = (V, E)$ and an integer K, does G contain a vertex cover of size at most K? A vertex cover is a subset $S \subseteq V$ such that for each edge (u, v) either $u \in S$ or $v \in S$ (or both).

Dominating Set Problem: Given a graph $G' = (V', E')$ and an integer K', does G' contain a dominating set of size at most K'? A dominating set is a subset $S \subseteq V$ such that each vertex is either in S or has a neighbor in S.

DOMINATING SET PROBLEM:

We prove that the dominating set problem is NP-complete by reducing vertex cover to it. To prove that dominating set is in NP, we need to prove that given a set S, we can verify that it is a dominating set in polynomial time. This is easy to do, since we can check membership of each vertex in S, or of one of its neighbors in S easily.

Reduction: Given a graph G (assume that G is connected), we replace each edge of G by a triangle to create graph G'. We set $K' = K$. More formally, $G' = (V', E')$ where $V' = V \cup V_e$ where $V_e = \{v_{e_i} | e_i \in E\}$ and $E' = E \cup E_e$ where $E_e = \{(v_{e_i}, v_k), (v_{e_i}, v_\ell) | e_i = (v_k, v_\ell) \in E\}$. (Another way to view the transformation is to subdivide each edge (u, v) by the addition of a vertex, and to also add an edge directly from u to v.)

If G has a vertex cover S of size K then the same set of vertices forms a dominating set in G'; since each vertex v has at least one edge (v, u) incident on it, and u must be in the cover if v isnt. Since v is still adjacent to u, it has a neighbor in S.

For the reverse direction, assume that G' has a dominating set of size K'. Without loss of generality we may assume that the dominating set only picks vertices from the set V. To see this, observe that if v_{e_i} is ever picked in the dominating set, then we can replace it by either v_k or v_ℓ, without increasing its size. We now claim that this set of K' vertices form a vertex cover. For each edge e_i, v_{e_i}) must have a neighbor (either v_k or v_ℓ) in the dominating set. This vertex will cover the edge e_i, and thus the dominating set in G' is a vertex cover in G.

DISJOINT CONNECTING PATHS:

Given a graph G and k pairs of vertices (s_i, t_i), are there k vertex disjoint paths $P_1, P_2, \ldots, P_k$ such that P_i connects s_i with t_i ?

This problem is NP-complete as can be seen by a reduction from 3SAT.

Corresponding to each variable x_i, we have a pair of vertices s_i, t_i with two internally vertex disjoint paths of length m connecting them (where m is the number of clauses). One path is called the *true* path and the other is the *false* path. We can go from s_i to t_i on either path, and that corresponds to setting x_i to true or false respectively.

For clause $C_j = (x_i \wedge \overline{x_\ell} \wedge x_k)$ we have a pair of vertices s_{n+j}, t_{n+j}. This pair of vertices is connected as follows: we add an edge from s_{n+j} to a node on the false path of x_i, and an edge from this node to t_{n+j}. Since if x_i is true, the s_i, t_i path will use the true path, leaving the false path free that will let s_{n+j} reach t_{n+j}. We add an edge from s_{n+j} to a node on the true path of x_ℓ, and an edge from this node to t_{n+j}. Since if x_ℓ is false, the s_ℓ, t_ℓ path will use the false path, leaving the true path free that will let s_{n+j} reach t_{n+j}. If x_i is true, and x_ℓ is false then we can connect s_{n+j} to t_{n+j} through either node. If clause C_j and clause $C_{j'}$ both use x_i then care is taken to connect the s_{n+j} vertex to a distinct node from the vertex $s_{n+j'}$ is connected to, on the false path of x_i. So if x_i is indeed true then the true path from s_i to t_i is used, and that enables both the clauses C_j and $C_{j'}$ to be true, also enabling both s_{n+j} and $s_{n+j'}$ to be connected to their respective partners.

Proofs of this construction are left to the reader.

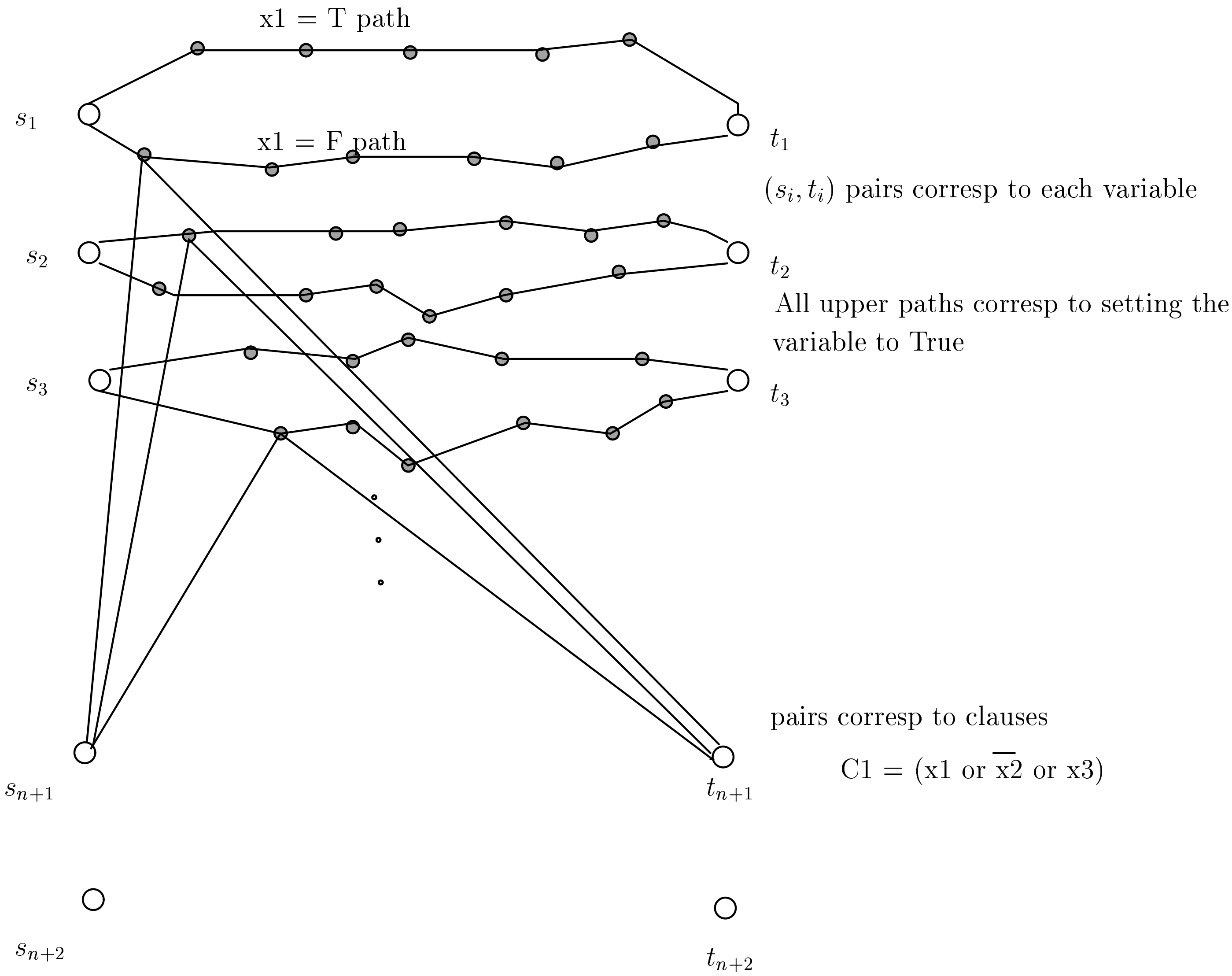


Figure 28: Graph corresponding to formula

CMSC 651 Advanced Algorithms	Lecture 20
Lecturer: Samir Khuller	Tu. Apr. 5, 1993

Notes by Samir Khuller.

27 NP-Completeness

We will assume that everyone knows that SAT is NP-complete. The proof of Cook's theorem was given in CMSC 650, which most of you have taken before. The other students may read it from [CLR] or the book by Garey and Johnson.

The reduction we will study today is: CLIQUE to MULTIPROCESSOR SCHEDULING.

MULTIPROCESSOR SCHEDULING: Given a DAG representing precedence constraints, and a set of jobs J all of unit length. Is there a schedule that will schedule all the jobs on M parallel machines (all of the same speed) in T time units ?

Essentially, we can execute upto M jobs in each time unit, and we have to maintain all the precedence constraints and finish all the jobs by time T.

Reduction: Given a graph $G = (V, E)$ and an integer k we wish to produce a set of jobs J, a DAG as well as parameters M, T such that there will be a way to schedule the jobs if and only if G contains a clique on k vertices.

Let n, m be the number of vertices and edges in G respectively.

We are going to have a set of jobs $\{v_1, \ldots, v_n, e_1, \ldots, e_m\}$ corresponding to each vertex/edge. We put an edge from v_i to e_j (in the DAG) if e_j is incident on v_i in G. Hence all the vertices have to be scheduled before the edges they are incident to. We are going to set $T = 3$.

Intuitively, we want the k vertices that form the clique to be put in time slot 1. This makes $C(k, 2)$ edges available for time slot 2 along with the remaining $n - k$ vertices. The remaining edges $m - C(k, 2)$ go in slot 3. To ensure that no more than k vertices are picked in the first slot, we add more jobs. There will be jobs in 3 sets that are added, namely B, C, D. We make each job in B a prerequisite for each job in C, and each job in C a prerequisite for each job in D. Thus all the B jobs have to go in time 1, the C jobs in time 2, and the D jobs in time 3.

The sizes of the sets B, C, D are chosen as follows:

$$|B| + k = |C| + (n - k) + C(k, 2) = |D| + (m - C(k, 2)).$$

This ensures that there is no flexibility in choosing the schedule. We choose these in such a way, that $\min(|B|, |C|, |D|) = 1$.

It is trivial to show that the existence of a clique of size k implies the existence of a schedule of length 3. The proof in the other direction is left to the reader.

CMSC 651 Advanced Algorithms	Lecture 21
Lecturer: Samir Khuller	Th. Apr. 7, 1993

Notes by Samir Khuller.

28 More on NP-Completeness

The reductions we will study today are:

1. 3 SAT to 3 COLORING.
2. 3D-MATCHING (also called 3 EXACT COVER) to PARTITION.
3. PARTITION to SCHEDULING.

I will outline the proof for 3SAT to 3COLORING. The other proofs may be found in Garey and Johnson.

We are given a 3SAT formula with clauses $C_1, \ldots, C_m$ and n variables $x_1, \ldots, x_n$. We are going to construct a graph that is 3 colorable if and only if the formula is satisfiable.

We have a clique on three vertices. Each node has to get a different color. W.l.o.g, we can assume that the three vertices get the colors $T, F, *$. (T denotes "true" F denotes "false".)

For each variable x_i we have a pair of vertices X_i and $\overline{X_i}$ that are connected to each other by an edge and also connected to the vertex $*$. This forces these two vertices to be assigned colors different from each other, and in fact there are only two possible colorings for these two vertices. If X_i gets the color T, then $\overline{X_i}$ gets the color F and we say that variable x_i is true. Notice that each variable can choose a color with no interference from the other variables. These vertices are now connected to the gadgets that denote clauses, and ensure that each clause is true.

We now need to construct an "or" gate for each clause. We want this gadget to be 3 colorable if and only if one of the inputs to the gate is T (true). We construct a "gadget" for *each* clause in the formula. For simplicity, let us assume that each clause has exactly three literals.

If all three inputs to this gadget are F, then there is no way to color the vertices. The vertices below the input must all get color $*$. The vertices on the bottom line now cannot be colored. On the other hand if any input is T, then the vertex below the T input can be given color F and the vertex below it (on the bottom line) gets color $*$, and the graph can be three colored.

If the formula is satisfiable, it is really easy to use this assignmenmt to the variables to produce a three coloring. We color each vertex X_i with T if x_i is True, and with F otherwise. (The vertex $\overline{X_i}$ is colored appropriately.) We now know that each clause has a true input. This lets us color each OR gate (since when an input it true, we can color the gadget). On the other hand, if the graph has a three coloring we can use the coloring to obtain a satisfying assignment (verify that this is indeed the case).

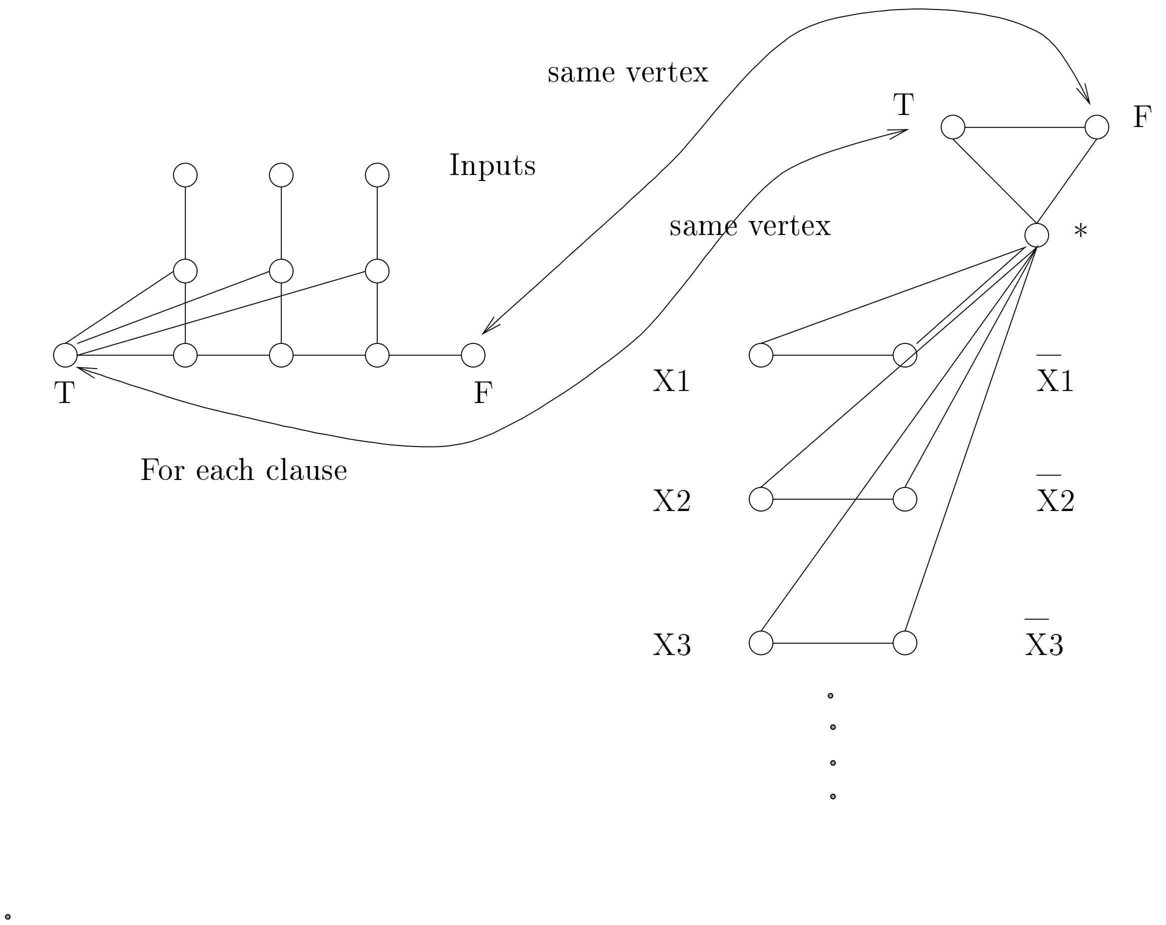


Figure 29: Gadget for OR gate

CMSC 651 Advanced Algorithms	Lecture 24
Lecturer: Samir Khuller	Th. Apr 25, 2002

Notes by Samir Khuller.

29 2 Satisfiability

See Chapter 36.4 (page 942) for a description of the Satisfiability problem. This lecture provides a solution to the 2-SAT problem (Problem 36.4-7).

Construct a directed graph G as follows. Corresponding to each literal we create a vertex (hence two nodes for each variable are created, one for x_i and one for $\overline{x_i}$. If there is a clause $x \vee y$ where x and y are literals, then we know that $\overline{x} \rightarrow y$ and $\overline{y} \rightarrow x$. (In other words, if x is F then y must be T, and vice-versa.) Add directed edges in G from $\overline{x}$ to y and $\overline{y}$ to x.

Claim: The formula has a satisfying assignment if and only if there is no variable x_i such that the nodes corresponding to x_i and $\overline{x_i}$ are in the same strongly connected component.

If x_i and $\overline{x_i}$ are in the same strongly connected component then we cannot set x_i to either T or F (setting x_i to T would imply that we need to set $\overline{x_i}$ to T as well, setting it to F would imply that $\overline{x_i}$ is F and so is x_i).

We now show how to compute a satisfying assignment if there is no variable and its negation that are in the same strongly connected component. In the following, whenever the value of a literal is set, we immediately set the value of its negation. Start with variable x_1. There are three cases: (i) x_1 has a path to $\overline{x_1}$ (ii) $\overline{x_1}$ has a path to x_1 (iii) no path in either direction. In (i) set $x_1 = F$. All literals that are reachable from $\overline{x_1}$ are set to T. One might wonder if this assignment is consistent (you should wonder!). Can it be that $\overline{x_1}$ has a path to x_j and also a path to $\overline{x_j}$ (for some j)? Observe that if $\overline{x_1}$ has a path to x_j then $\overline{x_j}$ has a path to x_1. This would imply that x_1 and $\overline{x_1}$ are on a cycle, a contradiction.

In (ii) set $x_1 = T$. All literals that are reachable from x_1 are set to T. Can it be that x_1 has a path to x_j and also a path to $\overline{x_j}$ (for some j)? Observe that if x_1 has a path to x_j then $\overline{x_j}$ has a path to $\overline{x_1}$. This would imply that x_1 and $\overline{x_1}$ are on a cycle, a contradiction.

In (iii) set $x_1 = T$. All literals that are reachable from x_1 are set to T. Can it be that x_1 has a path to x_j and also a path to $\overline{x_j}$ (for some j)? Observe that if x_1 has a path to x_j then $\overline{x_j}$ has a path to $\overline{x_1}$. This implies that x_1 has a path to $\overline{x_1}$, a contradiction.

Thus we can use this procedure to set the values to all the variables in order. At each step a consistent value is assigned to each literal. We just ensure that if a literal is set to T, then its entire reachable set is also set to T. This is sufficient to guarantee a satisfying assignment.

Another way of finding the assignment is to build the DAG of strong components and to find a topological order for it. Once we have the topological order, we can denote $O(x_i)$ as the topological number assigned to x_i. Clearly $O(x_i)$ and $O(\overline{x_i})$ are different since they belong to different strong components. If $O(x_i) < O(\overline{x_i})$ then we set $x_i = F$, otherwise we set $x_i = T$. Clearly, if there is a path from x_i to $\overline{x_i}$ we are guaranteed that we set $x_i = F$ (other case is similar). The main check we need to perform is if there is a path from a TRUE literal to a FALSE literal, which would yield a clause that was false. Suppose $x_i = T$ and there is a path from x_i to $x_j = F$. Clearly, $O(\overline{x_i}) < O(x_i)$ and $O(x_j) < O(\overline{x_j})$. Since there is a path from x_i to x_j, $O(x_i) < O(x_j)(< O(\overline{x_j}))$. By the construction of the graph if there is a path from x_i to x_j then there must be a path from $\overline{x_j}$ to $\overline{x_i}$. Since $O(\overline{x_i}) < O(x_i) < O(x_j) < O(\overline{x_j})$ this is not possible.

Notes by Samir.

30 Linear Programming

Linear programming is one of the most general problems known to be solvable in polynomial time. Many optimization problems can be cast directly as polynomial-size linear programs and thus solved in polynomial time. Often the theory underlying linear programming is useful in understanding the structure of optimization problems. For this reason, L.P. theory is often useful in *designing* efficient algorithms for an optimization problem and for understanding such algorithms within a general framework.

Here is an example of a linear programming problem.

$$\begin{array}{ll} \max & (5x_1 + 4x_2 + 3x_3) \\ \text{s.t.} & 2x_1 + 3x_2 + x_3 \leq 5 \\ & 4x_1 + x_2 + 2x_3 \leq 11 \\ & 3x_1 + 4x_2 + 2x_3 \leq 8 \\ & x_1, x_2, x_3 \geq 0 \end{array}$$

The goal is to find values (real numbers) for the x_i's meeting the constraints so that the *objective function* is maximized. Note that all of the constraints are linear inequalities or equations and the objective function is also linear. In general, a *linear program* is a problem of maximizing or minimizing a linear function of some variables subject to linear constraints on those variables.

Simplex Method. The most common method of solving linear programs is the *simplex algorithm*, due to Dantzig. The method first converts the linear inequalities into equality constraints by introducing *slack* variables. The original variables are referred to as the *decision* variables.

$$\begin{array}{ll} \text{define} & \\ & x_4 = 5 - 2x_1 - 3x_2 - x_3 \\ & x_5 = 11 - 4x_1 - x_2 - 2x_3 \\ & x_6 = 8 - 3x_1 - 4x_2 - 2x_3 \\ & z = 5x_1 + 4x_2 + 3x_3 \\ & \quad x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \\ \max & z \end{array}$$

It then starts with an initial *feasible solution* (an assignment to the variables meeting the constraints, but not necessarily optimizing the objective function). For instance, let $x_1, x_2, x_3 = 0$, so that $x_4 = 5, x_5 = 11, x_6 = 8$ gives a feasible solution. The simplex method then does successive improvements, changing the solution and increasing the value of z, until the solution is optimal.

Clearly, if we can increase x_1 without changing x_2 or x_3, we can increase the value of z. We consider increasing x_1 and leaving x_2 and x_3 zero, while letting the equations for x_4, x_5, and x_6 determine their values. If we do this, how much can we increase x_1 before one of x_4, x_5 or x_6 becomes zero? Clearly, since $x_4 \geq 0$, we require that $x_1 \leq \frac{5}{2}$. Since $x_5 \geq 0$, we require that $x_1 \leq \frac{11}{4}$. Similarly, since $x_6 \geq 0$, we require that $x_1 \leq \frac{8}{3}$. We thus increase x_1 to $\frac{5}{2}$, and see that $z = 12.5$. Now $x_2, x_3, x_4 = 0$ and x_1, x_5, x_6 are non-zero.

This operation is called a *pivot*. A pivot improves the value of the objective function, raises one zero variable, brings one non-zero variable down to zero.

How do we continue ? We rewrite the equations so that all the non-zero variables (simplex calls these the "basic" variables) are expressed in terms of the zero variables. In this case, this yields the following equations.

$$\begin{array}{ll} \max & z \\ \text{s.t.} & x_1 = \frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4 \end{array}$$

$$
\begin{aligned}
x_5 &= 1 + 5x_2 + 2x_4 \\
x_6 &= \tfrac{1}{2} + \tfrac{1}{2}x_2 - \tfrac{1}{2}x_3 + \tfrac{3}{2}x_4 \\
z &= \tfrac{25}{2} - \tfrac{7}{2}x_2 + \tfrac{1}{2}x_3 - \tfrac{5}{2}x_4 \\
& \quad x_1, x_2, x_3, x_4, x_5, x_6 \geq 0
\end{aligned}
$$

The natural choice for the variable to increase is x_3. It turns out that we can increase x_3 to 1, and then x_6 becomes 0. Then, the new modified system we get (by moving x_3 to the LHS) is:

$$
\begin{array}{ll}
\max & z \\
\text{s.t.} & x_3 = 1 + x_2 + 3x_4 - 2x_6 \\
& x_1 = 2 - 2x_2 - 2x_4 + x_6 \\
& x_5 = 1 + 5x_2 + 2x_4 \\
& z = 13 - 3x_2 - x_4 - x_6 \\
& \quad x_1, x_2, x_3, x_4, x_5, x_6 \geq 0
\end{array}
$$

Now we have $z = 13$ (with the current solution in which $x_2, x_4, x_6 = 0$). Also notice that increasing any of the zero variables will only decrease the value of z. Since this system of equations is equivalent to our original system, we have an optimal solution.

Possible Problems:

Initialization: Is it easy to get a starting feasible solution ?

Progress in an iteration: Can we always find a variable to increase if we are not at optimality?

Termination: Why does the algorithm terminate ?

Their are fairly easy ways to overcome each of these problems.

Linear Programming Duality

Given an abstract linear programming problem (with n variables and m constraints), we can write it as:

$$\begin{array}{lll} \max & z = c^T x & c^T = (1 \times n),\ x = (n \times 1) \\ \text{s.t.} & Ax \le b & A = (m \times n),\ b = (m \times 1) \\ & x \ge 0 & \end{array}$$

Simplex obtains an optimal solution (i.e., maximizes z). Notice that *any feasible* solution to the `primal` problem yields a lower bound on the value of z. The entire motivation for studying the linear programming `dual` is a quest for an *upper* bound on the value of z.

Consider the following linear programming problem:

$$\begin{array}{ll} \max & z = 4x_1 + x_2 + 5x_3 + 3x_4 \\ \text{s.t.} & x_1 - x_2 - x_3 + 3x_4 \le 1 \\ & 5x_1 + x_2 + 3x_3 + 8x_4 \le 55 \\ & -x_1 + 2x_2 + 3x_3 - 5x_4 \le 3 \\ & x_1, x_2, x_3, x_4 \ge 0 \end{array}$$

Notice that $(0, 0, 1, 0)$ is a feasible solution to the primal, and this yields the value of $z = 5$, implying that $z^* \ge 5$ (z^* is the optimum value). The solution $(3, 0, 2, 0)$ is also feasible and shows that $z^* \ge 22$.

If we had a solution, how would we know that it was optimal ?

Here is one way of giving an upper bound on the value of z^*. Add up equations (2) and (3). This gives us $4x_1 + 3x_2 + 6x_3 + 3x_4 \le 58$. Clearly, for non-negative values of x_i this is always $\ge z$, $(4x_1 + x_2 + 5x_3 + 3x_4 \le 4x_1 + 3x_2 + 6x_3 + 3x_4 \le 58)$. This gives us an upper bound on the value of z^* of 58.

In general, we are searching for a linear combination of the constraints to give an upper bound for the value of z^*. What is the best combination of constraints that will give us the smallest upper bound ? We will formulate this as a linear program ! This will be the dual linear program.

Consider multiplying equation (i) $(i = 1, 2, 3)$ in the above example by y_i and adding them all up. We obtain

$$(y_1 + 5y_2 - y_3)x_1 + (-y_1 + y_2 + 2y_3)x_2 + (-y_1 + 3y_2 + 3y_3)x_3 + (3y_1 + 8y_2 - 5y_3)x_4 \le y_1 + 55y_2 + 3y_3$$

We also require that all $y_i \ge 0$.

We would like to use this as an upper bound for z. Since we would like to obtain the tightest possible bounds (want to minimize the RHS), we could rewrite it as the following LP.

$$\begin{array}{l} y_1 + 5y_2 - y_3 \ge 4 \\ -y_1 + y_2 + 2y_3 \ge 1 \\ -y_1 + 3y_2 + 3y_3 \ge 5 \\ 3y_1 + 8y_2 - 5y_3 \ge 3 \\ y_1, y_2, y_3 \ge 0 \end{array}$$

If these constraints are satisfied, we conclude that $z \le y_1 + 55y_2 + 3y_3$. If we try to minimize $(y_1 + 55y_2 + 3y_3)$ subject to the above constraints, we obtain the *dual linear program.*

The dual linear program can be written succinctly as:

$$\begin{array}{lll} \min & b^T y & b^T = (1 \times m),\ y = (m \times 1) \\ \text{s.t.} & y^T A \ge c^T & A = (m \times n),\ c^T = (1 \times n) \\ & y \ge 0 & \end{array}$$

Primal LP (n variables, m equations):

$$\begin{array}{lll} & \max \sum_{j=1}^{n} c_j x_j & \\ \text{s.t.} & \sum_{j=1}^{n} a_{ij} x_j \le b_i & (i = 1 \ldots m) \\ & x_j \ge 0 & (j = 1 \ldots n) \end{array}$$

Dual LP (m variables, n equations):

$$\begin{array}{lll} & \min \sum_{i=1}^{m} b_i y_i & \\ \text{s.t.} & \sum_{i=1}^{m} a_{ij} y_i \ge c_j & (j = 1 \ldots n) \\ & y_i \ge 0 & (i = 1 \ldots m) \end{array}$$

Theorem 30.1 (Weak Duality Theorem)
For every primal feasible solution x, and every dual feasible solution y, we have:

$$\sum_{j=1}^{n} c_j x_j \le \sum_{i=1}^{m} b_i y_i.$$

Proof:
The proof of this theorem is really easy and follows almost by definition.

$$\sum_{j=1}^{n} c_j x_j \le \sum_{j=1}^{n} (\sum_{i=1}^{m} a_{ij} y_i) x_j = \sum_{i=1}^{m} (\sum_{j=1}^{n} a_{ij} x_j) y_i \le \sum_{i=1}^{m} b_i y_i$$

□

It is easy to see that if we obtain x^* and y^*, such that the equation in the Weak Duality theorem is met with equality, then both the solutions, x^*, y^* are optimal solutions for the primal and dual programs. By the Weak Duality theorem, we know that for any solution x, the following is true:

$$\sum_{j=1}^{n} c_j x_j \le \sum_{i=1}^{m} b_i y_i^* = \sum_{j=1}^{n} c_j x_j^*$$

Hence x^* is an optimal solution for the primal LP. Similarly, we can show that y^* is an optimal solution to the Dual LP.

Theorem 30.2 (Strong Duality Theorem)
If the primal LP has an optimal solution x^*, then the dual has an optimal solution y^* such that:

$$\sum_{j=1}^{n} c_j x_j^* = \sum_{i=1}^{m} b_i y_i^*.$$

Proof:
To prove the theorem, we only need to find a (feasible) solution y^* that satisfies the constraints of the Dual LP, and satisfies the above equation with equality. We solve the primal program by the simplex method, and introduce m slack variables in the process.

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j \quad (i = 1, \ldots, m)$$

Assume that when the simplex algorithm terminates, the equation defining z reads as:

$$z = z^* + \sum_{k=1}^{n+m} \overline{c}_k x_k.$$

Since we have reached optimality, we know that each $\overline{c}_k$ is a nonpositive number (in fact, it is 0 for each basic variable). In addition z^* is the value of the objective function at optimality, hence $z^* = \sum_{j=1}^{n} c_j x_j^*$. To produce y^* we pull a rabbit out of a hat ! Define $y_i^* = -\overline{c}_{n+i} \quad (i = 1, \ldots, m)$. To show that y^* is an optimal dual feasible solution, we first show that it is feasible for the Dual LP, and then establish the strong duality condition.

¿From the equation for z we have:

$$\sum_{j=1}^{n} c_j x_j = z^* + \sum_{k=1}^{n} \overline{c}_k x_k - \sum_{i=1}^{m} y_i^* (b_i - \sum_{j=1}^{n} a_{ij} x_j).$$

Rewriting it, we get

$$\sum_{j=1}^{n} c_j x_j = (z^* - \sum_{i=1}^{m} b_i y_i^*) + \sum_{j=1}^{n} (\overline{c}_j + \sum_{i=1}^{m} a_{ij} y_i^*) x_j.$$

Since this holds for all values of x_i, we obtain:

$$z^* = \sum_{i=1}^{m} b_i y_i^*$$

(this *establishes the equality*) and

$$c_j = \overline{c}_j + \sum_{i=1}^{m} a_{ij} y_i^* \quad (j = 1, \ldots, n).$$

Since $\overline{c}_k \leq 0$, we have

$$y_i^* \geq 0 \quad (i = 1, \ldots, m).$$

$$\sum_{i=1}^{m} a_{ij} y_i^* \geq c_j \quad (j = 1, \ldots, n)$$

This *establishes the feasibility* of y^*. □

Complementary Slackness Conditions:

Theorem 30.3 *Necessary and Sufficient conditions for x^* and y^* to be optimal solutions to the primal and dual are as follows.*

$$\sum_{i=1}^{m} a_{ij} y_i^* = c_j \text{ or } x_j^* = 0 \text{ (or both) for } j = 1, \ldots, n$$

$$\sum_{j=1}^{n} a_{ij} x_j^* = b_i \text{ or } y_i^* = 0 \text{ (or both) for } i = 1, \ldots, m$$

In other words, if a variable is non-zero then the corresponding equation in the dual is met with equality, and vice versa.

Proof:

We know that

$$c_j x_j^* \leq (\sum_{i=1}^{m} a_{ij} y_i^*) x_j^* \qquad (j = 1, \ldots, n)$$

$$(\sum_{j=1}^{n} a_{ij} x_j^*) y_i^* \leq b_i y_i^* \qquad (i = 1, \ldots, m)$$

We know that at optimality, the equations are met with equality. Thus for any value of j, either $x_j^* = 0$ or $\sum_{i=1}^{m} a_{ij} y_i^* = c_j$. Similarly, for any value of i, either $y_i^* = 0$ or $\sum_{j=1}^{n} a_{ij} x_j^* = b_i$. □

CMSC 858K Advanced Algorithms	Lectures 4 and 5
Lecturer: Samir Khuller	February 9 and 11

Original Notes by Chi-Jiun Su.

31 Vertex Cover

The vertex cover problem is defined as follows:

> *Given a graph G= (V,E) find a subset $C \subseteq V$ such that for all $(u,v) \in E$, at least one of u or v is included in C and the cardinality of set C is minimized.*

This problem is a special case of the set cover problem in which the elements correspond to edges. There is a set corresponding to each vertex which consists of all the edges incident upon it and each element is contained exactly in two sets. Although the greedy algorithm for set cover we mentioned in the last class can be applied for vertex cover, the approximation factor is still $H(\Delta(G))$, where $\Delta(G)$ is the maximum degree of a node in the graph G. Can we hope to find algorithms with better approximation factors? The answer is yes and we will present a number of 2-approximation algorithms.

31.1 2-Approximation Algorithms

31.1.1 Simple Algorithm

Input: Unweighted graph G(V,E)
Output: Vertex cover C

1. $C \leftarrow \emptyset$
2. **while** $E \neq \emptyset$ **do begin**
   Pick any edge $e = (u,v) \in E$ and choose *both* end-points u and v.
   $C \leftarrow C \cup \{u,v\}$
   $E \leftarrow E \setminus \{e \in E \mid e = (x,y) \text{ such that } u \in \{x,y\} \text{ or } v \in \{x,y\}\}$
   end.
3. **return** C.

The algorithm achieves an approximation factor of 2 because for a chosen edge in each iteration of the algorithm, it chooses both vertices for vertex cover while the optimal one will choose at least one of them.

Note that picking only one of the vertices arbitrarily for each chosen edge in the algorithm will fail to make it a 2-approximation algorithm. As an example, consider a star graph. It is possible that we are going to choose more than one non-central vertex before we pick the central vertex.

Another way of picking the edges is to consider a maximal matching in the graph.

31.1.2 Maximal Matching-based Algorithm

Input: Unweighted graph G(V,E)
Output: Vertex cover C

1. Pick any maximal matching $M \subset E$ in G.
2. $C \leftarrow \{v \mid v \text{ is matched in } M\}$.
3. **return** C.

It is also a 2-approximation algorithm. The reason is as follows. Let M be a maximal matching of the graph. At least one of the end-points in all the edges in $E \setminus M$ is matched since, otherwise, that edge could be added to M, contradicting the fact that M is a maximal matching. This implies that every edge in E has at least one end-point that is matched. Therefore, C is a vertex cover with exactly $2\ |M|$ vertices.

To cover the edges in M, we need at least $|M|$ vertices since no two of them share a vertex. Hence, the optimal vertex cover has size at least $|M|$ and we have

$$VC = 2\ |M| \ \leq\ 2\ VC^{optimal}$$

where VC is the size of vertex cover.

32 Weighted Vertex Cover

> *Given a graph $G = (V, E)$ and a positive weight function $w : V \to R^+$ on the vertices, find a subset $C \subseteq V$ such that for all $(u, v) \in E$, at least one of u or v is contained in C and $\sum_{v \in C} w(v)$ is minimized.*

The greedy algorithm, which choose the vertex with minimum weight first, does not give a 2-approximation factor. As an example, consider a star graph with $N + 1$ nodes where the central node has weight $1 + \epsilon$, $\epsilon << 1$, and other non-central nodes have weight 1. Then, it is obvious that the greedy algorithm gives vertex cover of weight N while the optimal has only weight $1 + \epsilon$.

32.1 LP Relaxation for Minimum Weighted Vertex Cover

Weighted vertex cover problem can be expressed as an integer program as follows.

$$\begin{array}{rccc} \text{OPT-VC} = & \text{Minimize} & \sum_{v \in V} w(v)X(v) & \\ \text{subject to} & & X(u) + X(v) \geq 1 & \forall e = (u, v) \in E \\ \text{where} & & X(u) \in \{0, 1\} & \forall u \in V \end{array}$$

If the solution to the above problem has $X(u) = 1$ for a vertex u, we pick vertex u to be included in the vertex cover.

But, since the above integer program is NP-complete, we have to relax the binary value restriction of $X(u)$ and transform it into a linear program as follows.

$$\begin{array}{rccc} \text{LP-Cost} = & \text{Minimize} & \sum_{v \in V} w(v)X(v) & \\ \text{subject to} & & X(u) + X(v) \geq 1 & \forall e = (u, v) \in E \\ \text{where} & & X(u) \geq 0 & \forall u \in V \end{array}$$

Nemhauser and Trotter showed that there exists an optimal solution of the linear program of the form: $X^*(u) = \left\{ \begin{array}{c} 0 \\ \frac{1}{2} \\ 1 \end{array} \right.$.

How do we interpret the results from LP as our vertex cover solution? One simple way is to include a vertex u in our vertex cover if $X^*(u) \geq \frac{1}{2}$. That is, the resultant vertex cover is

$$C = \{u \in V | X^*(u) \geq \frac{1}{2}\}$$

This forms a cover since for each edge, it cannot be that the X values of both end points is $< \frac{1}{2}$. This turns out to give 2-approximation of minimum weighted vertex cover. Why?

$$\sum_{u \in C} w(u) \ \leq\ \sum_{u \in C} 2X^*(u)w(u) \ \leq\ \sum_{u \in V} 2X^*(u)w(u) \ =\ 2 \text{ LP-cost}$$

Moreover, since LP is obtained by the relaxation of binary value restriction of $X(u)$ in the original integer program,

$$\text{LP-Cost} \leq \text{OPT-VC}.$$

Therefore, the cost of the vertex cover we rounded from the LP is

$$\text{VC} = \sum_{u \in C} w(u) \ \leq \ 2 \ \text{OPT-VC}$$

32.1.1 Primal-Dual Applied to Weighted Minimum Cover

Although relaxation to LP and rounding from the LP solutions gives us a 2-approximation for the minimum weighted vertex cover, LP is still computationally expensive for practical purposes. How about the dual form of the relaxed LP?

Given a LP in standard form:

Primal (**P**)

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

where x, b, c are column vectors, A is a matrix and T denotes the transpose. We can transform it into its dual form as follows:

Dual (**D**)

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

Then, the dual of our relaxed LP is

$$\begin{array}{rlcll} \text{DLP-Cost} \ = & \text{Maximize} \ \sum_{e \in E} Y(e) & & & \\ \text{subject to} & \sum_{u: e=(u,v) \in E} Y(e) & \leq & w(v) & \forall v \in V \\ & Y(e) & \geq & 0 & \forall e \in E \end{array}$$

Y_e is called "*packing function*". A packing function is any function $Y : E \rightarrow R^+$ which satisfies the above two constraints.

According to *Weak Duality Theorem*, if PLP-Cost and DLP-Cost are the costs for a *feasible* solution to the primal and dual problem of the LP respectively,

$$\text{DLP-Cost} \ \leq \ \text{PLP-Cost}$$

Therefore,

$$\text{DLP-Cost} \ \leq \ \text{DLP-Cost}^* \ \leq \ \text{PLP-Cost}^*$$

But, remember that PLP-Cost* $\leq$ OPT-VC. Therefore, we don't need to find the optimal solution to the dual problem. Any feasible solution is just fine, as long as we can upper bound the cost of the vertex cover as a function of the dual feasible solution. How do we find a feasible solution to the dual problem? It reduces to the problem: "how to find a packing function". In fact, any *maximal packing function* will work.

By *maximal*, we mean that for an edge $e(u, v) \in E$, increase $Y(e)$ as much as possible until the inequality in the first constraint becomes equality for u or v. Suppose u is an end-point of the edge e for which the constraint becomes equality. Add u to the vertex cover, C, and remove all the edges incident on u. Obviously, C is a vertex cover. Does this vertex cover give 2-approximation?

Note that

$$w(v) \ = \sum_{e(u,v): u \in N(v)} Y(e) \qquad \forall v \in C$$

where $N(v)$ is the set of vertices adjacent to vertex v. Therefore,

$$\sum_{v \in C} w(v) \;=\; \sum_{v \in C} \sum_{e(u,v):u \in N(v)} Y(e) \;=\; \sum_{e=(u,v) \in E} |C \cap \{u,v\}|\, Y(e) \;\leq\; \sum_{e(u,v) \in E} 2\, Y(e)$$

However, according to the constraint of the packing function,

$$w(v) \;\geq\; \sum_{e(u,v):u \in N(v)} Y_e$$

$$\sum_{v \in C^*} w(v) \;\geq\; \sum_{v \in C^*} \sum_{e(u,v):u \in N(v)} Y_e$$

where C^* is the optimal weighted vertex cover.

Since some of the edges are counted twice in the right-side expression of the above inequality,

$$\sum_{v \in C^*} \sum_{e(u,v):u \in N(v)} Y_e \;\geq\; \sum_{e \in E} Y_e$$

Hence,

$$\sum_{v \in C} w(v) \;\leq\; 2 \sum_{v \in C^*} w(v) \;=\; 2\ \text{OPT-VC}$$

One simple packing function is, for some vertex v, to distribute its weight $w(v)$, onto all the incident edges *uniformly*. That is, $Y_e = \frac{w(v)}{d(v)}$ for all the incident edges e where $d(v)$ is the degree of the vertex v. For other vertices, adjust Y_e appropriately. This way of defining packing function leads to the following algorithm.

32.1.2 Clarkson's Greedy Algorithm

Input: Graph G(V,E) and weight function w on V
Output: Vertex cover C

1. **for all** $v \in V$ **do** $W(v) \leftarrow w(v)$
2. **for all** $v \in V$ **do** $D(v) \leftarrow d(v)$
3. **for all** $e \in E$ **do** $Y_e \leftarrow 0$
4. $C \leftarrow \emptyset$
5. **while** $E \neq \emptyset$ **do begin**
   Pick a vertex $v \in V$ for which $\frac{W(v)}{D(v)}$ is minimized
   for all edges $e = (u,v) \in E$ **do begin**
   $E \leftarrow E \setminus e$;
   $W(u) \leftarrow W(u) - \frac{W(v)}{D(v)}$ and $D(u) \leftarrow D(u) - 1$
   $Y_e \leftarrow \frac{W(v)}{D(v)}$
   end
   $C \leftarrow C \cup \{v\}$ and $V \leftarrow V \setminus \{v\}$
   $W(v) \leftarrow 0$
   end.
6. **return** C.

32.1.3 Bar-Yehuda and Even's Greedy Algorithm

Input: Graph G(V,E) and weight function w on V
Output: Vertex cover C

1. **for all** $v \in V$ **do** $W(v) \leftarrow w(v)$
2. **for all** $e \in E$ **do** $Y_e \leftarrow 0$
3. $C \leftarrow \emptyset$
4. **while** $E \neq \emptyset$ **do begin**
   Pick an edge $(p, q) \in E$
   Suppose $W(p) \leq W(q)$
   $Y_e \leftarrow W(p)$
   $W(q) \leftarrow W(q) - W(p)$
   for all edges $e = (p, v) \in E$ **do begin**
   $E \leftarrow E \setminus e$;
   end
   $C \leftarrow C \cup \{p\}$ and $V \leftarrow V \setminus \{p\}$
   end.
5. **return** C.

Note that both these algorithms come up with a maximal packing (or maximal dual solution) and pick those vertices in the cover for which the dual constraint is met with equality. Any method that comes up with such a maximal dual solution would in fact yield a 2 approximation. These are just two ways of defining a maximal dual solution.

Original notes by K. Subramani

33 A $(2 - f(n))$ Approximation Algorithm for the Vertex Cover Problem

The Vertex Cover problem has been introduced in previous lectures. To recapitulate, we are given a graph $G = (V, E)$ and the aim is to determine a cover of of minimum weight, where a cover is defined a set $X \subseteq V$, such that for every edge $e = (u, v) \in E$ we have at least one of $u \in X$ or $v \in X$. Technically speaking, a vertex cover is any subset of vertices that covers all the edges. We are interested in approximations to the Minimum Vertex Cover. We have seen several factor 2 approximations for this problem in previous lectures.

In this lecture, we aim to improve this approximation to a factor of $(2 - \frac{\log \log n}{2 \log n})$.

33.1 Local Ratio Theorem and Subgraph Removal

Consider a pattern graph H and its occurrence as an induced subgraph in the given graph G. If we can determine that G does not have the pattern H, then we know that G belongs to the class of graphs that does not contain H as a subgraph. In such an event, it may be possible to find better approximations for this special class of graphs. (Assume the existence of an algorithm A_H that gives an approximation factor of $(2 - f(n))$ for graphs that do not contain any subgraph isomorphic to H.) On the other hand, if G does contain this special pattern then we can use the following approach:

1. Eliminate[3] certain vertices from G_H (the subgraph of G that is isomorphic to H), in a manner specified by the algorithm given below. Elimination of these vertices from G also eliminates the edges incident on them, so we are left with a smaller graph $G - G_H$
2. Find the Vertex Cover for $G - G_H$ recursively.

We terminate the above procedure when we determine that the residual graph, say G' does not have any subgraph that is isomorphic to H. At this point, we apply the special case algorithm to G'. It is easily seen that the above sequence of steps results in a cover for G. What remains to be shown is that the vertex cover thus obtained gives a $(2 - f(n))$ approximation for G, with $f(n) = \frac{\log \log n}{2 \log n}$.

Let H' be a set of graphs. Let $A_{H'}$ be an approximation algorithm for the weighted vertex cover problem in the case when the input graph does not have any subgraph isomorphic to a pattern in H'. w is the weight function associated with the input graph G. The algorithm will continuously modify w.

We now formalize the ideas presented above by presenting the Bar-Yehuda and Even algorithm (a.k.a. Subgraph Removal Algorithm.)

Function BAR–YEHUDA AND EVEN(G, w)
begin

Set $w_0 = w$.

while there there exists a subgraph H of G that is isomorphic to some member of H' **do**

Let $\delta = \{\min w_0(v) | v \in V(H0)\}$.
$\forall v \in V(H)$ set $w_0(v) \leftarrow w_0(v) - \delta$.

Let $VC_0 \leftarrow \{v \in G | w_0(v) = 0\}; V_1 \leftarrow V - VC_0$.

Let $VC_1 = A_H(G_{V_1}, w_0)$.

return$(VC \leftarrow VC_0 \cup VC_1)$
end.

[3] i.e. include them in the cover

33.2 Analysis

Let us define for each $H \in H'$, $r_H = \frac{n_H}{c_H}$, where n_H is the number of vertices in H and c_H is the cardinality of a minimum unweighted vertex cover in H. r_H is called the *local ratio* of H.

We will require the following lemma and theorem to aid us in our analysis.

Lemma 33.1 *Let $G = (V, E)$ be a graph and w, w_0 and w_1 be weight functions on the vertex set V, such that $w(v) \geq w_0(v) + w_1(v)$ for all $v \in V$. Let C^*, C_0^* and C_1^* be optimal weighted vertex covers with respect to the three weight functions, then*

$$w(C^*) \geq w_0(C_0^*) + w_1(C_1^*).$$

Proof:

$$w(C^*) = \sum_{v \in C^*} w(v) \geq \sum_{v \in C^*} (w_0(v) + w_1(v))$$

$$= w_0(C^*) + w_1(C^*) \geq w_0(C_0^*) + w_1(C_1^*).$$

□

Theorem 33.2 Local Ratio Theorem– *Let $r_{H'} = \max_{H \in H'} r_H$ and let $r_{A_{H'}}$ be the approximation factor guaranteed by the special-case algorithm on a graph G' that does not have a subgraph isomorphic to any of the patterns in H'. Then the vertex cover returned by the Subgraph Removal algorithm has weight at most $r = \max\{r_{H'}, r_{A_{H'}}\}$.*

Proof:

The proof is by induction on s, the number of times that the *do-while* loop in Step 1 of the algorithm is iterated through. For $s = 0$, the theorem is trivially true.

Consider the case $s = 1$. Let $H \in H'$ be the pattern that is isomorphic to a subgraph in G'. Let C^* and C_0^* be the optimal solutions w.r.t w and w_0 respectively and let C be the vertex cover returned by the algorithm. Then

$$w(C) \leq w_0(C) + \delta \cdot n_H \leq r_{A_H} \cdot w_0(C^*) + r_H \cdot \delta \cdot c_H$$

$$\leq r(w_0(C_0^*) + \delta c_H) \leq r \cdot w(C^*).$$

(The last equation follows by applying the previous lemma.)

For $s > 1$, imagine running Step 1 through one iteration. Then, by considering the remainder of the algorithm as "$A_{H'}$" with performance guarantee r (from the induction hypothesis), we are in the case $s - 1$, from which the result follows.

□

Thus by selecting H' appropriately and designing $A_{H'}$ to take advantage of the absence of such subgraphs, we can obtain improved approximation guarantees.

Let us analyze the situation when $H' = \{$all odd cycles of length $(2k - 1)$, where k is chosen to be as small as possible such that $(2k - 1)^k > n.\}$. Thus $k = \frac{2 \log n}{\log \log n}$. For this set, we have,

$$r_H = \frac{2k - 1}{k} = 2 - \frac{1}{k} \tag{1}$$

Assume that during the course of the algorithm, we have eliminated all odd cycles of length $2k - 1$. Let G_k denote the graph that is so obtained. Our goal now is to design an improved approximation algorithm for G_k.

We now apply the Nemhauser-Trotter Algorithm (compute a fractional solution to the LP with values $1, 0, \frac{1}{2}$) to G_k, which partitions the set of vertices into three sets. We denote these classes as T_1, T_0 and $T_{\frac{1}{2}}$ respectively. The vertices in T_i have fractional value i in the optimal fractional solution. Nemhauser and Trotter proved that there is an optimal integer solution (optimal vertex cover) that includes all the vertices in T_1, none of the vertices in T_0 and a subset of the vertices in $T_{\frac{1}{2}}$. We know that the vertices in T_1 are definitely included in a minimum vertex cover so we include all these vertices in our cover and delete the edges that are incident upon them. Likewise the vertices in T_0 are not part of the vertex cover. Now we are left with a graph G'_k which only has the vertices in $T_{\frac{1}{2}}$ and the edges between them. Since the fractional weight of this set is at least $\frac{1}{2}w(T_{\frac{1}{2}})$, we can assume that we are now working with a graph where the optimal cover has at least half the weight of all the vertices! (So, picking all the vertices gives us a factor 2 approximation.)

Accordingly we get the following algorithm:

Step 1: Build a Breadth-First Search Tree T_{BS} starting from the heaviest vertex $v_P \in G'_k$.

Step 2: Divide the vertices into classes as follows: Let L_i denote the set of vertices at a distance i from v_P. Accordingly, we have, $L_0 = v_P$, $L_1 =\{$ all vertices at unit distance from $v_P\}$ and so on.

Step 3: Observe that G'_k cannot have an edge between the vertices of any class L_i, for $i < k$ as this would imply an odd cycle of length $\leq (2k-1)$, clearly a contradiction. Thus we need to pick only alternate layers in our vertex cover.

Step 4: Define (for even t)

$$B_t = L_t \cup L_{t-2} \ldots \cup L_0$$

i.e., B_t is the union of vertex classes beginning at L_t and picking every alternate class before it, till we hit L_0. For odd values of t, we define B_t in a similar way until we hit L_1.

Step 5: Continue building additional layers L_t as long as $w(B_t) \geq (2k-1) \cdot w(B_{t-1})$. (This is a rapidly expanding BFS.)

Step 6: Suppose we stop at step m with $w(B_m) < (2k-1)w(B_{m-1})$. Pick B_m in the cover, and delete all vertices in $B_m \cup B_{m-1}$ since all the edges incident on them are covered.

Step 7: Recurse on G''.

When we stop the BFS, say at L_m, we know that $w(B_m) < (2k-1)w(B_{m-1})$. We now argue that the rapidly expanding BFS cannot go on for more than k steps. If this were not the case

$$w(B_k) \geq (2k-1) \cdot w(B_{k-1}) \ldots \geq (2k-1)^k . w(B_0) > n \cdot w(B_0),$$

which is a contradiction, since $B_0 = v_P$ is the heaviest vertex in the graph. In other words our BFS process must terminate in $m < k$ steps.

Consider the total weight of the nodes in all the layers when we stop. We have,

$$w(D_m) = w(B_m) + w(B_{m-1})$$

$$w(D_m) > w(B_m) + \frac{w(B_m)}{(2k-1)}$$

$$w(B_m) < (\frac{2k-1}{2k})w(D_m)$$

However from our earlier discussions, we are assured that $w(C^*) \geq \sum_m \frac{w(D_m)}{2}$ The cost of the vertex cover obtained by our algorithm is

$$w(C) = \sum w(B_m)$$

$$\leq \sum (\frac{2k-1}{2k})w(D_m) \leq (\frac{2k-1}{2k})(2w(C^*))$$

$$\leq (2 - \frac{1}{k})w(C^*).$$

Thus we have shown that in a graph with no "small" odd cycles we can achieve an approximation factor of $(2-\frac{1}{k})$. Likewise, from equation (1) we have a local ratio of $(2-\frac{1}{k})$ for odd cycles of length $(2k-1)$. Applying the local ratio theorem, we have the desired result i.e., the algorithm presented indeed provides a $(2-\frac{1}{k})$ approximation to the Vertex Cover of the graph. Picking $k = \frac{2\log n}{\log\log n}$ satisfies the condition on k and gives the result. (Also note that the running time depends crucially on k.)

CMSC 651 Advanced Algorithms	Lecture 4
Lecturer: Samir Khuller	Mon. Feb. 26, 2001

Notes by Samir Khuller

34 Nemhauser-Trotter Theorem

Consider the IP formulation for the weighted vertex cover problem. When we relax this to a linear program and solve it, it turns out that any extreme point solution (there always exists an optimal solution extreme point solution) has the property that the variables take one of three possible values $\{0, \frac{1}{2}, 1\}$. There are various ways of proving this theorem. However, our primary concern is of showing the following interesting theorem originally proven by Nemhauser and Trotter. The proof given here is simpler than their proof.

First some defintions.

1. V_0 is the set of nodes whose fractional values are 0.
2. V_1 is the set of nodes whose fractional values are 1.
3. $V_{\frac{1}{2}}$ ais the set of nodes whose fractional values are 1/2.

Theorem 34.1 *(Nemhauser and Trotter) There exists an optimal solution OPT with the following properties*

(a) OPT is a subset of $(V_1 \cup V_{\frac{1}{2}})$.

(b) OPT includes all of V_1.

Proof:

We will only prove (a). Note that if OPT satisfies (a) then it must also satisfy (b). Let us see why. Suppose OPT does not include $v \in V_1$. If v does not have any neighbors in V_0 then we can reduce the fractional variable X_v and get a BETTER LP solution. This assumes that the weight of the node is > 0. If the weight of the node is 0, then clearly we can include it in OPT without increasing the weight of OPT. If v has a neighbor in V_0, then OPT does not cover this edge!

We will now prove (a). Suppose OPT picks nodes from V_0. Let us call $(OPT \cap V_0)$ as B_I (bad guys in) and $(V_1 \setminus OPT)$ as L_O (left out guys). The nodes in B_I can have edges to ONLY nodes in V_1. The edges going to L_O are the only ones that have one end chosen, all the other edges are double covered. The nodes in L_O have no edges to nodes in $(V_0 \setminus OPT)$, otherwise these edges are not covered by OPT. IF $w(B_I) \geq w(L_O)$ then we can replace B_I by L_O in OPT and get another OPTIMAL solution satisfying (a). If $w(B_I) < w(L_O)$ then we can pick an ϵ and then modify the fractional solution by reducing the X_v values of the nodes in L_O by ϵ and increasing the fractional X_v values of the nodes in B_I to produce a lower weight fractional solution, contradicting the assumption that we started off with an optimal LP solution. □

Using this theorem, lets us concentrate our attention on finding a vertex cover in the subgraph induced by the vertices in $V_{\frac{1}{2}}$. If we find a cover S in this subgraph, then $S \cup V_1$ gives us a cover in the original graph. Moreover, if we can prove bounds on the weight of S then we could use this to derive an approximation factor better than 2. Note that the optimal solution in $V_{\frac{1}{2}}$ has weight $\geq \frac{1}{2}w(V_{\frac{1}{2}})$.

CMSC 858K Advanced Algorithms	Lecture 01
Lecturer: Samir Khuller	Jan 28, 1999

Original Notes by Charles Lin

35 Approximation Algorithms: Set Cover

35.1 Introduction

The goal of algorithm design is to develop efficient algorithms to solve problems. An algorithm is usually considered efficient if its worst case running time on an instance of a problem is polynomial in the size of the problem. However, there are problems where the best algorithms have running times which are exponential (or worse) in the worst case. Most notably, the class of NP-hard problems currently have algorithms which are exponential in the worst case.

The need to develop efficient algorithms has lead to the study of approximation algorithms. By relaxing the optimality of the solution, polynomial algorithms have been developed that solve NP complete problems to within a constant factor of an optimal solution.

35.2 Approximation Algorithms

We can formalize the notion of an approximation algorithm some more by considering what it means to approximate a minimization problem. A minimization problem is one where the solution minimizes some quantity under certain criteria. For example, vertex cover is usually considered a minimization problem. Given a graph G, the goal is to find the fewest vertices that "cover" all edges.

Let P be the set of problem instances for a minimization problem. Let OPT be the optimal algorithm for solving the problems in P. If the problem is vertex cover, then P contains all undirected graphs, and OPT would find the minimum vertex cover for each of the instances. Let $I \in P$ be a problem instance. Then, we will say that $OPT(I)$ is the minimum solution for I. For vertex cover, we can say that this is the size of the minimum vertex cover.

An α-approximation algorithm A (we often refer to α as the approximation factor), approximates the optimal solution as follows:

$$\forall I,\ A(I) \le \alpha\, OPT(I) \qquad \text{where}\ \alpha > 1.$$

If there were a 2-approximation algorithm for vertex cover and the optimal solution for a given graph had 3 vertices, then the approximate solution would do no worse than find a vertex cover with 6 vertices. Note that an α-approximation algorithm only gives an upper bound to how far away the approximation is from optimal. While there may exist problem instances as bad as α times the optimal, the algorithm may produce solutions close to optimal for many problem instances.

Some α-approximation algorithms have α depend on the size of the input instance or some other property of the input, whereas others produce approximations independent of the input size. Later on, for example, we will consider an $H(|S_{max}|)$-approximation algorithm where the approximation factor is a function of the maximum set size in a collection of sets.

For a maximization problem, an α-approximation algorithm would satisfy the following formula:

$$\forall I,\ OPT(I) \ge A(I) \ge \alpha\, OPT(I) \qquad \text{where}\ \alpha < 1.$$

35.3 Polynomial Approximation Schemes

For some problems, there are polynomial approximation schemes (PAS). PAS are actually a family of algorithms that approximate the solution to a given problem *as well as we wish*. However, the closer the solution is to the optimal, the longer it takes to find it. More formally, for any $\epsilon > 0$, there is an approximation algorithm, A_ϵ such that

$$\forall I,\ A_\epsilon(I) \le (1+\epsilon)\, OPT(I)$$

There is a similar formula for maximization problems.

Although a problem may have a PAS, these schemes can have a very bad dependence on ϵ. A PAS can produce algorithms that are exponential in $\frac{1}{\epsilon}$, e.g., $O(n^{\frac{1}{\epsilon}})$. Notice that for any fixed ϵ, the complexity is polynomial. However, as ϵ approaches zero, the polynomial grows very quickly. A fully polynomial approximation scheme (FPAS) is one whose complexity is polynomial in both n (the problem size) and $\frac{1}{\epsilon}$. Very few problems have a FPAS and those that do are often impractical because of their dependence on ϵ. Bin packing is one such example.

35.4 Set Cover

We will start by considering approximation algorithms for set cover, which is a very fundamental NP-complete problem, as well as problems related to set cover.

The input to set cover are two sets, X and S.

$$\begin{array}{rcll} X & = & \{x_1, x_2, \ldots, x_n\} & \\ S & = & \{S_1, S_2, \ldots, S_m\} & \quad \text{where } S_i \subseteq X \end{array}$$

We further assume that $\bigcup_{S_i \in S} S_i = X$. Hence, each element of X is in some set S_i. A *set cover* is a subset $S' \subseteq S$ such that

$$\bigcup_{S_j \in S'} S_j = X$$

The set cover problem attempts to find the minimum number of sets that cover the elements in X. A variation of the problem assigns weights to each set of S. An optimal solution for this problem finds a set cover with the least total weight. The standard set cover problem can then be seen as a weighted set cover problem where the weights of each set are 1.

The set cover problem can also be described as the *Hitting Set Problem*. The formulation of the hitting set problem is different from set cover, but the problems are equivalent. In the hitting set problem, we have a hypergraph, $H = (V, E)$ where

$$\begin{array}{rcl} V & = & \{v_1, v_2, \ldots, v_n\} \\ E & = & \{e_1, e_2, \ldots, e_m\} \end{array}$$

In a normal graph, an edge is incident on two vertices. In a hypergraph, an edge can be incident on one or more vertices. We can represent edges in a hypergraph as subset of vertices. Hence, $e_i \subseteq V$ for all $e_i \in E$. In the Hitting Set Problem, the goal is to pick the fewest vertices which "cover" all hyperedges in H.

To show that that hitting set is equivalent to set cover, we can transform an instance of hitting set to that of set cover.

$$X = E$$

$$S = \{S_1, S_2, \ldots, S_n\}$$

where $e \in S_i$ iff edge $e \in E$ touches vertex v_i. For every vertex v_i in V, there is a corresponding set S_i in S. The reverse transformation from set cover to hitting set is similar.

The Hitting Set Problem can be seen as a generalization of the vertex cover problem on graphs, to hypergraphs. The two problems are exactly the same except that vertex cover adds the constraint that edges touch exactly two vertices. Vertex cover can also be seen as a specialization of set cover where each element of X appears in exactly two sets in S.

35.5 Approximating Set Cover

We will consider an approximation algorithm for set cover whose value for α is $H(|S_{max}|)$ where S_{max} is the largest set in S. H is the harmonic function and is defined as

$$H(d) = \sum_{i=1}^{d} \frac{1}{i}.$$

A simple heuristic is to greedily pick sets from S. At each step, pick $S_j \in S$ which contains the most number of uncovered vertices. Repeat until all vertices are covered. The following is pseudo-code for that greedy algorithm.

```
GREEDY-SET-COVER(X, S)
U ← X      // U is the set of uncovered elements
S' ← ∅     // S' is the current set cover
while U ≠ ∅ do
    pick S_j ∈ S such that |S_j ∩ U| is maximized    // pick set with most uncovered elements
    S' ← S' ∪ S_j
    U ← U − S_j
end
```

To show that the approximation factor is $H(|S_{max}|)$, we will use a "charging" scheme. The cost of choosing a set is one (dollar). If we pick a set with k uncovered elements, each of the uncovered elements will be charged $\frac{1}{k}$ dollars. Since each element is covered only once, it is charged only once. The cost of the greedy algorithm's solution will be the total charge on all the elements.

$$\text{Greedy Cost} = \sum_{x_i \in X} c(x_i) \leq \sum_{S_i \in OPT} C(S_i) \leq |OPT| \cdot H(|S_{max}|)$$

where $c(x_i)$ is the cost assigned to element x_i. S_i is a set in the optimum set cover. $C(S_i)$ is the cost of set S_i by summing the charges of the elements in S_i. That is,

$$C(S_i) = \sum_{x_i \in S_i} c(x_i)$$

$c(x_i)$ is still the cost assigned to x_i from the greedy algorithm.

The inequality, $\sum_{x_i \in X} c(x_i) \leq \sum_{S_i \in OPT} C(S_i)$, says that the cost of the optimum set cover is at least the sum of the cost of the elements. We can see this more readily if we rewrite the cost of the optimum set cover as

$$\sum_{S_i \in OPT} C(S_i) = \sum_{x_i \in X} c(x_i) \cdot n(x_i)$$

where $n(x_i)$ is the number of times element x_i appears in the optimum set cover. Since each element appears at least once, the inequality naturally follows.

To show the second inequality, $\sum_{S_i \in OPT} C(S_i) \leq |OPT| \cdot H(|S_{max}|)$, it suffices to show that $C(S_i) \leq H(|S_i|)$. The inequality would then hold as follows:

$$\sum_{S_i \in OPT} C(S_i) \leq \sum_{S_i \in OPT} H(|S_i|) \leq \sum_{S_i \in OPT} H(|S_{max}|) = |OPT| \cdot H(|S_{max}|).$$

To show $C(S_i) \leq H(|S_i|)$, consider an arbitrary set $S_i \in S$. As sets are being picked by the greedy algorithm, we want to see what happens to the elements of S_i. Let U_0 be the set of uncovered elements in S_i when the algorithm starts. As sets are picked, elements of S_i will be covered a piece at a time. Suppose we are in the middle of running the algorithm. After j iterations, U_j represents the uncovered elements of S_i. If the algorithm picks a set which covers elements in U_j, we write the new set of uncovered elements as U_{j+1}. Note that U_{j+1} is a proper subset of U_j. Hence, $U_0, U_1, \ldots, U_k, U_{k+1}$ is the history of uncovered elements of S_i, where $U_{k+1} = \emptyset$.

The following illustrates the history of uncovered elements in S_i.

$$U_0 \xrightarrow{|U_0 - U_1|} U_1 \xrightarrow{|U_1 - U_2|} U_2 \ldots U_k \xrightarrow{|U_k|} U_{k+1} = \emptyset$$

Above each arrow is the number of elements that were charged. Going from U_j to U_{j+1}, the number of elements charged is $|U_j - U_{j+1}|$. How much are these elements charged?

Suppose we are at the point in the algorithm where U_j represents the set of uncovered elements in S_i, and the greedy algorithm is just about to pick a set. Assume that it picks a set which covers at least one uncovered element in S_i (otherwise, let the algorithm continue until this happens).

We want to show that the elements that get covered are charged no more than $1/|U_j|$. If it picks S_i, then each element in U_j is charged exactly $1/|U_j|$. Let's assume it doesn't pick S_i, but picks T instead. The only reason it would pick T is if T had at least $|U_j|$ uncovered elements. In this case, each of uncovered elements in T is charged at most $1/|T| \leq 1/|U_j|$.

By assumption, we said some elements of T overlap uncovered elements in S_i. Specifically, $T \cap U_j$ was covered when T was picked. The total additional cost to S_i from picking T is at most

$$|T \cap U_j| \cdot \frac{1}{|U_j|} = (U_j - U_{j+1}) \cdot \frac{1}{|U_j|}$$

The total cost charged to S_i is therefore

$$C(S_i) \leq \frac{|U_0 - U_1|}{|U_0|} + \frac{|U_1 - U_2|}{|U_1|} + \ldots + \frac{|U_k - U_{k+1}|}{|U_k|}$$

Since $U_{k+1} = \emptyset$, the last term simplifies to 1.

Each of the terms of sum have the form $(x-y)/x$ for $x > y$. The claim is that this is less than $H(x)-H(y)$. This can be shown by expanding $H(x)$ and $H(y)$.

$$\begin{array}{rcl}
\text{Consider} & & H(x) - H(y) \\
& = & (1 + \frac{1}{2} + \ldots + \frac{1}{x}) - (1 + \frac{1}{2} + \ldots + \frac{1}{y}) \\
& = & \frac{1}{y+1} + \frac{1}{y+2} + \ldots \frac{1}{x} \\
& \geq & (x - y) \cdot \frac{1}{x}
\end{array}$$

The last equation comes from the fact that line 3 contains $x - y$ terms, each term being at least $1/x$.

Using this fact, we can rewrite the total charge on S_i as

$$\begin{array}{rcl}
C(S_i) & \leq & (H(|U_0|) - H(|U_1|)) + (H(|U_1|) - H(|U_2|)) + \ldots + (H(|U_k|) - H(|U_{k+1}|)) \\
& = & H(|U_0|) - H(|U_{k+1}|) \\
& = & H(|U_0|)
\end{array}$$

Since U_0 is just S_i, then $C(S_i) \leq H(|S_i|)$.

One might ask how tight can $C(S_i)$ be made? Suppose S_i contains 5 elements. In one step 3 elements are covered and in a subsequent step 2 more elements are covered. The cost of the set is at most $\frac{1}{5} + \frac{1}{5} + \frac{1}{5} + \frac{1}{2} + \frac{1}{2}$ which is less than $H(5)$. Can we make $C(S_i)$ exactly equal to $H(|S_i|)$? Yes, but only if at most one element is covered in S_i whenever the greedy algorithm picks a set to be added to the set cover.

Here is a scenario where S_0 can be charged exactly $H(|S_0|)$. Let $S = \{S_0, S_1, \ldots, S_m\}$. Let $|S_j| = j$ (the size of the set is the subscript) except for S_0 which has size m. Let $S_0 = \{x_1, \ldots, x_m\}$. The sets $S_1, \ldots, S_m$ will have no elements in common. Each set only has one element in common with S_0. Specifically, $S_j \cap S_0 = \{x_j\}$ where $j > 0$.

If the greedy algorithm picks its set in the following order, $S_m, S_{m-1}, \ldots, S_1$, then $C(S_0)$ will be exactly $H(|S_0|)$.

CMSC 858K Advanced Algorithms	Lecture 02
Lecturer: Samir Khuller	Feb 2, 1999

Notes by Samir Khuller.

36 Approximation Algorithms: Set Cover and Max Coverage

In this lecture we will study a different proof of the approximation guarantee for the greedy set cover algorithm. The bound we derive is not as good as the bound we obtained last time, but this proof method is often used for other problems (like node-weighted Steiner trees, which we will come to later in the semester).

We will also study a linear programming based approach for set cover due to Chvatal, and see how this leads very naturally to the analysis we learnt in the first lecture on Set Cover (see Section 3.2 in the book). (If you did not follow this part of the lecture, do not worry about it for now, since we will come back to studying linear programming duality, and its role in proving approximation guarantees.)

Finally, we study a problem closely related to set cover called max coverage. Here we are given a set of elements X with each element having an associated weight. We are also given a collection of subsets of X. We wish to pick k subsets so as to maximize the total weight of the elements that are included in the chosen subsets. For this problem we can show a constant $(1-\frac{1}{e})$ approximation factor (see Section 3.9 in the book).

36.1 Johnson's set cover proof

Let n_i be the number of uncovered elements at the start of the i^{th} iteration of the greedy set cover algorithm. Initially, $n_1 = n$ the number of elements.

Lemma 36.1 *Let S_i be the set picked by the greedy set cover algorithm in the i^{th} iteration.*

$$\frac{w(S_i)}{|S_i|} \leq \frac{w(OPT)}{n_i}.$$

Proof:

Note that S_i is the set that achieves the minimum "weight to size" ratio. Suppose this ratio is α. We wish to prove that $\alpha \leq \frac{w(OPT)}{n_i}$.

Consider the optimal set cover. Let $\mathcal{S}'$ be the collection of sets in the optimal set cover. Note that for each set $S_j \in \mathcal{S}'$ we have that $\frac{w(S_j)}{|S_j|} \geq \alpha$. This implies that $w(S_j) \geq \alpha |S_j|$. Summing over all sets in the optimal solution, this yields $\sum_{S_j \in \mathcal{S}'} w(S_j) \geq \alpha \sum_{S_j \in \mathcal{S}'} |S_j| \quad \geq \alpha n_i$. Since the sum of the weights of the sets is exactly $w(OPT)$, we get $w(OPT) \geq \alpha n_i$. This proves the lemma. □

Note that

$$n_{i+1} = n_i - |S_i|$$

Using the lemma, we obtain

$$n_{i+1} \leq n_i - \frac{w(S_i)}{w(OPT)} n_i.$$

$$n_{i+1} \leq n_i (1 - \frac{w(S_i)}{w(OPT)}).$$

$$\frac{n_{i+1}}{n_i} \leq (1 - \frac{w(S_i)}{w(OPT)}).$$

Using the fact that for $0 \leq x \leq 1$ we have $1 - x \leq e^{-x}$, we get

$$\frac{n_{i+1}}{n_i} \leq e^{-\frac{w(S_i)}{w(OPT)}}.$$

Assuming that S_p is the last set that we pick, so that $n_{p+1} = 0$, we get

$$\frac{n_p}{n_{p-1}} \leq e^{-\frac{w(S_{p-1})}{w(OPT)}}.$$

Taking the product of all these terms (for each value of $i = 1 \ldots (p-1)$) we obtain.

$$\frac{n_p}{n_1} \leq e^{-\sum_{i=1}^{p-1} \frac{w(S_i)}{w(OPT)}}.$$

Taking natural log's and multiplying by -1 we get

$$\ln \frac{n_1}{n_p} \geq \sum_{i=1}^{p-1} \frac{w(S_i)}{w(OPT)}.$$

$$w(OPT) \ln \frac{n}{n_p} \geq \sum_{i=1}^{p-1} w(S_i).$$

Since $n_p \geq 1$ we get

$$w(OPT) \ln n \geq \sum_{i=1}^{p-1} w(S_i).$$

Since we only pick one more set, S_p that has cost at most $w(OPT)$, our cost is at most $(1 + \ln n) w(OPT)$.

Notes by Samir Khuller.

37 Approximation Algorithms: K-Centers

The basic K-center problem is a fundamental facility location problem and is defined as follows: given an edge-weighted graph $G = (V, E)$ find a subset $S \subseteq V$ of size at most K such that each vertex in V is "close" to some vertex in S. More formally, the objective function is defined as follows:

$$\min_{S \subseteq V} \max_{u \in V} \min_{v \in S} d(u, v)$$

where d is the distance function. For example, one may wish to install K fire stations and minimize the maximum distance (response time) from a location to its closest fire station. The problem is known to be NP-hard. See Section 9.4 for more details.

37.1 Gonzalez's Algorithm

Gonzalez describes a very simple greedy algorithm for the basic K-center problem and proves that it gives an approximation factor of 2. The algorithm works as follows. Initially pick any node v_0 as a center and add it to the set C. Then for $i = 1$ to K do the following: in iteration i, for every node $v \in V$, compute its distance $d^i(v, C) = \min_{c \in C} d(v, c)$ to the set C. Let v_i be a node that is farthest away from C, i.e., a node for which $d^i(v_i, C) = \max_{v \in V} d(v, C)$. Add v_i to C. Return the nodes $v_0, v_1, \ldots, v_{K-1}$ as the solution.

We claim that this greedy algorithm obtains a factor of 2 for the K-center problem. First note that the radius of our solution is $d^K(v_K, C)$, since by definition v_K is the node that is farthest away from our set of centers. Now consider the set of nodes $v_0, v_1, \ldots, v_K$. Since this set has cardinality $K+1$, at least two of these nodes, say v_i and v_j, must be covered by the same center c in the optimal solution. Assume without loss of generality that $i < j$. Let R^* denote the radius of the optimal solution. Observe that the distance from each node to the set C does not increase as the algorithm progresses. Therefore $d^K(v_K, C) \leq d^j(v_K, C)$. Also we must have $d^j(v_K, C) \leq d^j(v_j, C)$ otherwise we would not have selected node v_j in iteration j. Therefore

$$d(c, v_i) + d(c, v_j) \geq d(v_i, v_j) \geq d^j(v_j, C) \geq d^K(v_K, C)$$

by the triangle inequality and the fact that v_i is in the set C at iteration j. But since $d(c, v_i)$ and $d(c, v_j)$ are both at most R^*, we have the radius of our solution $= d^K(v_K, C) \leq 2R^*$.

37.2 Hochbaum-Shmoys method

We give a high-level description of the algorithm. We may assume for simplicity that G is a complete graph, where the edge weights satisfy the triangle inequality. (We can always replace any edge by the shortest path between the corresponding pair of vertices.)

High-Level Description

The algorithm uses a thresholding method. Sort all edge weights in non-decreasing order. Let the (sorted) list of edges be $e_1, e_2, \ldots e_m$. For each i, let the threshold graph G_i be the unweighted subgraph obtained from G by including edges of weight at most $w(e_i)$. Run the algorithm below for each i from 1 to m, until a solution is obtained. (Hochbaum and Shmoys suggest using binary search to speed up the computation. If running time is not a factor, however, it does appear that to get the best solution (in practice) we should run the algorithm for all i, and take the best solution.) In each iteration, we work with the unweighted subgraph G_i. Since G_i is an unweighted graph, when we refer to the distance between two nodes, we refer to the number of edges on a shortest path between them. In iteration i, we find a solution using some number of centers. If the number of centers exceeds K, we prove that there is no solution with cost at most $w(e_i)$. If the number of centers is at most K, we find a solution.

Observe that if we select as centers a set of nodes that is sufficiently well-separated in G_i then no two centers that we select can share the same center in the optimal solution, if the optimal solution has radius $w(e_i)$. This suggests the following approach. First find a maximal independent set I in G_i^2. (G_i^2 is the graph obtained by adding edges to G_i between nodes that have a common neighbor.) Let I be the chosen set of centers. If I has cardinality at most K we are done, since each vertex has a neighbour in the independent set at distance $2w(e_i)$. This means that all vertices have a center close to them. If the cardinality of I exceeds K, then there is no solution with cost $w(e_i)$. This is because no two vertices in I can be covered by a common center. (If there was a vertex that could cover both the nodes in I at distance $w(e_i)$ then, in G_i, these two nodes have a common neighbor and cannot both be in a maximal independent set.

Since the optimal solution has some cost $\delta = w(e_j)$ for some j. The claim is that we will find a maximal independent set of size at most K when we try $i = j$ and build the graph G_i. This graph has a dominating set of size at most K. If $N(k)$ is the set of vertices dominated (either k or a vertex adjacent to k in G_j) by $k \in OPT$ (OPT is the optimal solution), then we can pick at most one vertex in any maximal independent set in $N(k)$. This implies that we will find a maximal independent set of size at most K in G_i^2. We might find one for some $i < j$, in which case we obtain a solution with cost $2w(e_i) \leq 2w(e_j)$.

CMSC 858K Advanced Algorithms	Lecture 26
Lecturer: Samir Khuller	Tu. May 4, 1999

Original notes by Chung-Yeung Lee and Gisli Hjaltason.

38 Bin Packing

Problem Statement: Given n items $s_1, s_2, ..., s_n$, where each s_i is a rational number, $0 < s_i \leq 1$, our goal is to minimize the number of bins of size 1 such that all the items can be packed into them.

Remarks:

1. It is known that the problem is NP-Hard.
2. A Simple Greedy Approach (First-Fit) can yield an approximation algorithm with a performance ratio of 2.

38.1 Approximate Schemes for bin-packing problems

In the 1980's, two approximate schemes were proposed. They are

1. (Fernandez de la Vega and Lueker) $\forall \epsilon > 0$, there exists an Algorithm A_ϵ such that

$$A_\epsilon(I) \leq (1+\epsilon)OPT(I) + 1$$

   where A_ϵ runs in time polynomial in n, but exponential in $1/\epsilon$ (n=total number of items).

2. (Karmarkar and Karp) $\forall \epsilon > 0$, there exists an Algorithm A_ϵ such that

$$A_\epsilon(I) \leq OPT(I) + O(\log^2(OPT(I))$$

   where A_ϵ runs in time polynomial in n and $1/\epsilon$ (n=total number of items). They also guarantee that $A_\epsilon(I) \leq (1+\epsilon)OPT(I) + O(\frac{1}{\epsilon^2})$.

We shall now discuss the proof of the first result. Roughly speaking, it relies on two ideas:

- Small items does not create a problem.
- Grouping together items of similar sizes can simplify the problem.

38.1.1 Restricted Bin Packing

We consider the following restricted version of the bin packing problem (RBP). We require that

1. Each item has size $\geq \delta$.
2. The size of each item takes only one of m distinct values $v_1, v_2, ..., v_m$. That is we have n_i items of size v_i $(1 \leq i \leq m)$, with $\sum_{i=1}^{m} n_i = n$.

For constant δ and m, the above can be solved in polynomial time (actually in $O(n + f(m,\delta)))$. Our overall strategy is therefore to reduce BP to RBP (by throwing away items of size $< \delta$ and grouping items carefully), solve it optimally and use $RBP(\delta, m)$ to compute a soluton to the original BP.

Theorem 38.1 *Let J be the instance of BP obtained from throwing away the items of size less than δ from instance I. If J requires β bins then I needs only $max(\beta, (1+2\delta)OPT(I)+1)$ bins.*

Proof:

We observe that from the solution of J, we can add the items of size less than δ to the bins until the empty space is less than δ. Let S be the total size of the items, then we may assume the number of items with size $< \delta$ is large enough (otherwise I needs only β bins) so that we use β' bins.

$$S \geq (1-\delta)(\beta'-1)$$

$$\beta' \leq 1 + \frac{S}{1-\delta}$$

$$\beta' \leq 1 + \frac{OPT(I)}{1-\delta}$$

$$\beta' \leq 1 + (1+2\delta)OPT(I)$$

as $(1-\delta)^{-1} \leq 1 + 2\delta$ for $0 < \delta < \frac{1}{2}$. □

Next, we shall introduce the grouping scheme for RBP. Assume that the items are sorted in descending order. Let n' be the total number of items. Define G_1=the group of the largest k items, G_2=the group that contains the next k items, and so on. We choose

$$k = \lfloor \frac{\epsilon^2 n'}{2} \rfloor.$$

Then, we have m+1 groups $G_1, .., G_{m+1}$, where

$$m = \lfloor \frac{n'}{k} \rfloor.$$

Further, we consider groups H_i = group obtained from G_i by setting all items sizes in G_i equal to the largest one in G_i. Note that

- size of any item in $H_i \geq$ size of any items in G_i.
- size of any item in $G_i \geq$ size of any items in H_{i+1}.

The following diagram (Fig 1) illustrates the ideas:

We then define J_{low} be the instance consisting of items in $H_2, .., H_{m+1}$. Our goal is to show

$$\text{OPT}(J_{\text{low}}) \leq \text{OPT}(J) \leq \text{OPT}(J_{\text{low}}) + k,$$

The first inequality is trivial, since from OPT(J) we can always get a solution for J_{low}.

Using OPT(J_{low}) we can pack all the items in $G_2, \ldots, G_{m+1}$ (since we over allocated space for these by converting them to H_i). In particular, group G_1, the group left out in J_{low}, contains k items, so that no more than k extra bins are needed to accommodate those items.

Since $(H_1 \cup J_{\text{low}})$ is an instance of a Restricted Bin Packing Problem we can solve it optimally, and then replace the items in H_i with items in G_i to get a solution for J.

Directly from this inequality, and using the definition of k, we have

$$\mathit{Soln}(J) \leq OPT(H_1 \cup \ldots \cup H_{m+1}) \leq \text{OPT}(J_{\text{low}}) + k \leq \text{OPT}(J) + k \leq \text{OPT}(J) + \frac{\epsilon^2 n'}{2}.$$

Choosing $\delta = \epsilon/2$, we get that

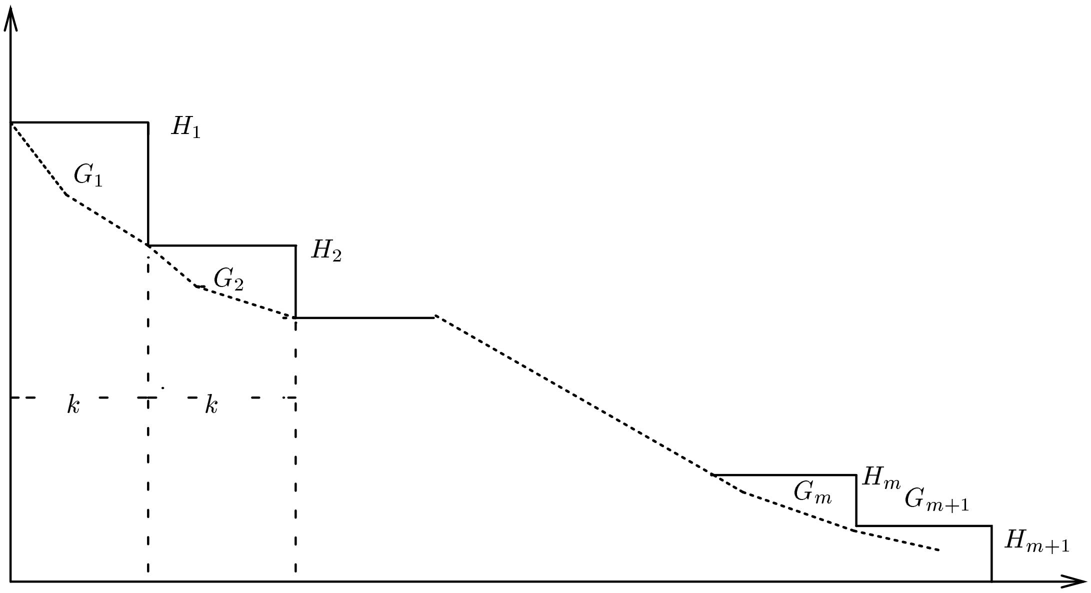
$$\text{OPT}(J) \geq \sum_{i=1}^{n'} s_i \geq n' \frac{\epsilon}{2},$$

so we have

$$\mathit{Soln}(J) \leq \text{OPT}(J) + \frac{\epsilon^2 n'}{2} \leq \text{OPT}(J) + \epsilon\text{OPT}(J) = (1+\epsilon)\text{OPT}(J).$$

By applying Theorem 38.1, using $\beta \leq (1+\epsilon)\text{OPT}(J)$ and the fact that $2\delta = \epsilon$, we know that the number of bins needed for the items of I is at most

$$\max\{(1+\epsilon)\text{OPT}(J), (1+\epsilon)\text{OPT}(I) + 1\} \leq (1+\epsilon)\text{OPT}(I) + 1.$$



Grouping Scheme for RBP

Figure 30: Grouping scheme

The last inequality follows since $OPT(J) \leq OPT(I)$.

We will turn to the problem of finding an optimal solution to RBP. Recall that an instance of RBP(δ, m) has items of sizes $v_1, v_2, \ldots, v_m$, with $1 \geq v_1 \geq v_2 \geq \cdots \geq v_m \geq \delta$, where n_i items have size v_i, $1 \leq i \leq m$. Summing up the n_i's gives the total number of items, n. A bin is completely described by a vector $(T_1, T_2, \ldots, T_m)$, where T_i is the number of items of size v_i in the bin. How many different different (valid) bin types are there? From the bin size restriction of 1 and the fact that $v_i \geq \delta$ we get

$$1 \geq \sum_i T_i v_i \geq \sum_i T_i \delta = \delta \sum_i T_i \Rightarrow \sum_i T_i \leq \frac{1}{\delta}.$$

As $\frac{1}{\delta}$ is a constant, we see that the number of bin types is constant, say p.

Let $T^{(1)}, T^{(2)}, \ldots, T^{(p)}$ be an enumeration of the p (valid) different bin types. A solution to the RBP can now be stated as having x_i bins of type $T^{(i)}$. Let $T_j^{(i)}$ denote the number of items of size v_j in $T^{(i)}$. The problem of finding the optimal solution can be posed as an integer linear programming problem:

$$\min \sum_{i=1}^{p} x_i,$$

such that

$$\sum_{i=1}^{p} x_i T_j^{(i)} = n_j \quad \forall j = 1, \ldots, m.$$

$$x_i \geq 0, x_i \text{ integer} \quad \forall i = 1, \ldots, p.$$

This is a constant size problem, since both p and m are constants, independent of n, so it can be solved in time independent of n. This result is captured in the following theorem, where $f(\delta, m)$ is a constant that depends only on δ and m.

Theorem 38.2 *An instance of RBP(δ, m) can be solved in time $O(n, f(\delta, m))$.*

An approximation scheme for BP may be based on this method. An algorithm A_ϵ for solving an instance I of BP would proceed as follows:

Step 1: Get an instance J of $\mathrm{BP}(\delta, n')$ by getting rid of all elements in I smaller than $\delta = \epsilon/2$.

Step 2: Obtain H_i from J, using the parameters k and m established earlier.

Step 3: Find an optimal packing for $H_1 \cup \ldots \cup H_{m+1}$ by solving the corresponding integer linear programming problem.

Step 4: Pack the items of G_i in place of the corresponding (larger) items of H_i as packed in step 3.

Step 5: Pack the small items in $I \setminus J$ using First-Fit.

This algorithm finds a packing for I using at most $(1+\epsilon)\mathrm{OPT}(I) + 1$ bins. All steps are at most linear in n, except step 2, which is $O(n \log n)$, as it basically amounts to sorting J. The fact that step 3 is linear in n was established in the previous algorithm, but note that while $f(\delta, m)$ is independent of n, it is exponential in $\frac{1}{\delta}$ and m and thus $\frac{1}{\epsilon}$. Therefore, this approximation scheme is polynomial but not fully polynomial. (An approximation scheme is fully polynomial if the running time is a polynomial in n and $\frac{1}{\epsilon}$.

CMSC 858K Advanced Algorithms	Lecture 9
Lecturer: Yoram J. Sussmann	Feb 27, 1997

Notes by Samir Khuller and Yoram J. Sussmann.

39 Multi-cost Minimum Spanning Trees

This talk summarizes results presented by R. Ravi and M. X. Goemans in SWAT '96.
The Constrained MST problem

Given:

- Undirected graph $G = (V, E)$.
- Two cost functions w_e (weight) and l_e (length) on the edges.
- Budget L on the length.

Find: Spanning tree of total edge *length* at most L that has minimum possible total weight.

Definition 39.1 ((α, β)-approximation algorithm) A poly-time algorithm that always outputs a spanning tree with total length at most αL and total weight at most $\beta W(L)$, where $W(L)$ is the minimum weight of *any* spanning tree with total length at most L.

Main Result: A $(1+\epsilon, 1)$-approximation for the constrained MST problem.

Integer Program IP for the constrained MST problem: Let S be the set of incidence vectors of spanning trees of G. In short, we are defining an integer variable x_e for each edge $e \in E$. This may be either 0 or 1. If we consider the vector $\mathbf{x}$ of all x_e values, we say that $\mathbf{x} \in S$ if $\mathbf{x}$ defines a spanning tree.

The integer program is:

$$W = \min \sum_{e \in E} w_e x_e$$

subject to:

$$\mathbf{x} \in S$$

$$\sum_{e \in E} l_e x_e \leq L.$$

Lagrangean Relaxation P_z of IP:

$$c(z) = \min \sum_{e \in E} w_e x_e + z(\sum_{e \in E} l_e x_e - L)$$

subject to:

$$\mathbf{x} \in S.$$

Notes:

1. P_z is an MST problem w.r.t. the cost function $c_e = w_e + zl_e$, since zL is a fixed term.

2. $W \geq LR = \max_{z \geq 0} c(z)$. For any fixed value of $z \geq 0$ notice that the optimal tree with length at most L is a tree with $\sum_{e \in E} l_e x_e - L \leq 0$ and hence the min cost tree for P_z has cost lower than W. Since this is true for *any* z, the claim is proven.

3. Let z^* be the value of z for which $c(z^*) = LR$.

4. Let $c_e^* = w_e + z^* l_e$.

5. For any spanning tree T, let $w(T)$ denote the total weight of T and $l(T)$ denote the total length of T.

Theorem 39.2 *Let T be an MST under cost function c^*. Then $w(T) \leq LR$* **iff** *$l(T) \geq L$.*

Proof:
$LR = c(z^*) = w(T) + z^*(l(T) - L)$. If $l(T) \geq L$ and $z^* \geq 0$ then $w(T) \leq LR$. The reverse direction can be argued similarly. □

The main point is that if we find an MST T (under the projected cost) for some z with $l(T) \geq L$ then we are guaranteed that the tree also has a lower weight than W. The key question is: how can we guarantee that the length will not be too much more than L? We will guarantee this by finding another tree T' (this is guaranteed to exist for the optimal value z^* since two lines cut each other, one with a positive slope and one with a negative slope) with length $l(T') \leq L$ and "convert" T' to T by changing one edge at a time. At some step, we will go from a tree with length at most L to a tree with length $> L$. Since we only add one edge in doing so, we can increase the weight of the tree by at most the weight of one edge.

High-level structure of $(2, 1)$-approximation:

1. Remove edges of length greater than L from E. These edges would never be used in an optimal solution since they are too long.

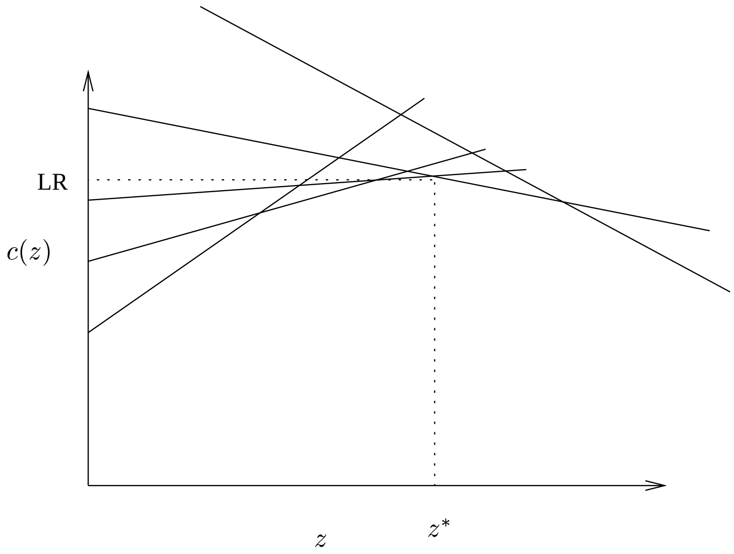
2. Use Megiddo's parametric search technique[4] to search for z^*. To test a value of z, compute trees T_{min} and T_{max} with respectively the shortest and longest total length of any MSTs under cost function c. If $l(T_{min}) > L$ then $z^* > z$; if $l(T_{max}) < L$ then $z^* < z$ (see Fig. 31). This is because at the optimal value of $z = z^*$, we have a tree with $l(T) \leq L$ and a tree with $l(T) \geq L$. (If we find one tree with $l(T) = L$ that is even better!)

3. Given z^*, T_{min} and T_{max}, compute a sequence of MSTs $T_{min} = T_0, T_1, \ldots, T_k = T_{max}$ under c^* such that adjacent trees differ by exactly one edge swap[5]. Return the first tree T_i in the sequence with total length *at least* L. Then $w(T) \leq LR$ (by Thm. 39.2), and $l(T) \leq l(T_{i-1}) + l_{max} \leq L + L = 2L$, where $l_{max} = \max_{e \in E} l_e$.

Polynomial-time approximation scheme: To modify the $(2, 1)$-approximation algorithm in order to get a scheme that gives a $(1 + \epsilon, 1)$-approximation for any $\epsilon > 0$, we prune more edges from the graph. If we prune all edges of weight at most ϵL, then we have $l_{max} \leq \epsilon L$, giving us a tree of length at most $(1 + \epsilon)L$. But we may remove edges that are used in the optimal solution.

[4]One approach is to guess a value of z, solve P_z and consider the length of trees T_{min} and T_{max} the shortest and longest length minimum spanning trees under cost c. These can be computed by choosing the shortest or longest length edges when we have to break ties under the projected cost. If $l(T_{min}) > L$ then $z^* > z$, otherwise if $l(T_{max}) < L$ then $z^* < z$. Otherwise we have found two trees, one with length $\leq L$ and the other with length $\geq L$. The problem is that this search may never terminate since we are searching over the real line. Megiddo's method gets around this problem in a clever way. Start running an MST computation *without* specifying z. When we compare the costs of two edges, we are comparing two linear functions in z with one crossing point λ. We choose this crossing point $z = \lambda$ and run a new MST computation (with a specified value for z) to decide if z^* is $< \lambda$ or $> \lambda$. Once we know where z^* is, we know which edge is smaller under the comparison, we continue. This runs in time $O(T^2)$ where T is the time for an MST computation, since for each comparison we run a new MST computation with a specified z value. Megiddo uses parallel algorithms to speed up the computation.

[5]We essentially have two spanning trees of the same projected cost, but with different lengths. We use the property that one tree can be converted into the other by a sequence of edge swaps.



Trees with positive slope have $l(T) > L$

Trees with negative slope have $l(T) < L$

Figure 31: Figure to show various tree costs as a function of z.

Key observation: At most $\frac{1}{\epsilon}$ of the pruned edges may be used in any optimal solution. Therefore we can try all $O(n^{O(\frac{1}{\epsilon})})$ (polynomial!) possible subsets of these edges for inclusion in the tree. For each subset, we include its edges in the tree, shrink the connected components, and run the above algorithm on the resulting graph, modifying the budget by subtracting the total length of the included edges. We return as our solution the tree with minimum total weight among all the solutions returned by the sub-algorithm (note that all trees returned have length at most $(1+\epsilon)L$).

Remark: We can also obtain a pseudopolynomial time algorithm that gives a $(1, 1+\epsilon)$-approximation by searching on a budget for the weight. This means that given a budget on the length L, we can obtain a tree where we meet the budget on the length and get a weight bound of $(1+\epsilon)W$ where W is the lowest weight of any tree that achieves length L.

To do this, we have to "guess" a weight w_i, run the algorithm by *fixing a budget on the weight.* If we obtain a length of ℓ_i and $\ell_i > L$ then we know that we have to increase our guess for w_i. If $\ell_i < L$ then we have to decrease our guess for w_i. When we guess the correct weight W we are guaranteed to get a tree of length L with weight $(1+\epsilon)W$. (This works if we assume that the weights are all integers.)

CMSC 858K Advanced Algorithms	Lecture 12
Lecturer: Samir Khuller	March 11, 1997

Notes by Kalvis Apsitis.

40 Set and Vertex Cover Approximations: Randomized Algorithms

Two randomized algorithms for the set cover and vertex cover problems are proposed. We start with Hitting Set problem which is a close analogue to Set Cover.

40.1 Randomized Hitting Set Algorithm

Given set $X = \{e_1, \ldots, e_n\}$ of weighted elements and a collection of subsets of X: $\{S_1, \ldots, S_m\}$, find a minimum weight subset $C \subseteq X$ which intersects every S_j.

Integer program for Hitting Set (variable $x_i = 1$ iff we pick element e_i):

$$\begin{array}{ll} \text{minimize:} & \sum_{i=1}^{n} w_i x_i \\ \text{subject to:} & \sum_{e_i \in S_j} x_i \geq 1 \ \ (j = 1, \ldots, m); \\ & 0 \leq x_i \leq 1 \ \ (i = 1, \ldots, n); \\ & x_i \text{ is integer } \ (i = 1, \ldots, n). \end{array}$$

By dropping (relaxing) the last condition we get a linear programming problem which can be solved in polynomial time.

Let LP^* be the minimum value of the linear program and C^* — the (unknown) optimal solution of the Hitting Set problem. Then

$$LP^* \leq w(C^*) = \sum_{i: e_i \in C^*} w_i.$$

Solution $x_1, \ldots, x_n$ of the relaxed problem typically consists of nonintegers. We want to round x_i to 0 or 1, to obtain a feasible Hitting Set solution C.

In the first round of our randomized algorithm we consider each element $e_i \in X$ and add it to our hitting set C with probability x_i. There is no guarantee that at the end of the first round C will be feasible; it might even happen that we have not picked any e_i. Although, after $O(\log m)$ such rounds of picking elements for C we will arrive at feasible solution with probability greater than $1/2$. Here is our proof.

The expected cost of the elements picked in the first round:

$$\sum_{e \in X} w_i \cdot \Pr(e_i \text{ is chosen}) = \sum w_i x_i = LP^*.$$

Compute the probability that the elements picked in some round will hit a particular set S_j:

$$\Pr(S_j \text{ is hit}) = 1 - \Pr(S_j \text{ is not hit}) = 1 - \prod_{e_i \in S_j} (1 - x_i) \geq 1 - \left(1 - \frac{1}{k}\right)^k \geq 1 - \frac{1}{e},$$

where k is the size of S_j. The last inequality follows from the definition of the number e. The inequality preceding it uses the fact that the geometric average does not exceed the arithmetic average (equality happens when all numbers $1 - x_i$ are equal) as well as the restriction $\sum_{e_i \in S_j} x_i \geq 1$.

Suppose that we have t rounds of random picking accordingly to the same probability distribution $x_1, \ldots, x_n$. Then

$$\Pr(S_i \text{ is left uncovered after } t \text{ rounds}) < (1/e)^t.$$

Choosing $t = 2 \log m$ we get $1/2m \geq (1/e)^t$. So, after t rounds:

$$\Pr(\text{All sets are covered}) > \left(1 - \frac{1}{2m}\right)^m \to \frac{1}{\sqrt{e}} > 50\%.$$

In practice we could delete all the sets that are hit after each round, formulate a new linear program and pick elements of X accordingly to its solution. This modification does not improve the approximation factor.

40.2 Randomized Weighted Vertex Cover Algorithm

Given a graph $G = (V, E)$ and a weight function $w : V \to R^+$, select a set of vertices C so that each edge has at least one endpoint in C and the total weight of C is minimal.

Randomized Weighted Vertex Cover Algorithm

Input: Graph G(V,E), weight function w.
Output: Vertex cover C.
1. $C \leftarrow \emptyset$.
2. **while** $E \neq \emptyset$ **do begin**
    Pick a random edge $(u, v) \in E$.
    Flip a biased coin:
        choose u with probability $\frac{w(v)}{w(u)+w(v)}$;
        choose v with probability $\frac{w(u)}{w(u)+w(v)}$.
    Denote by z the chosen vertex.
    Delete from E all edges covered by z.
end
3. **return** C

In the case of unweighted problem, endpoints u and v are picked with the same probabilty 1/2. It is easy to see that the approximation factor for this algorithm applied to the unweighted problem is 2:

> **Basic Idea.** Fix an optimal solution C^*. The randomized algorithm yields an approximate solution C where $|C|$ is the number of the edges considered. Suppose that whenever the randomized algorithm adds a vertex from C^* to C, a bell rings. Since each edge is covered by C^*, bell rings for each edge with probability 1/2. Therefore $C^* \geq E(\text{number of bell rings}) = |C|/2$.

In the general (weighted) case we can get the same estimate. Let C be the vertex cover picked by the algorithm and C^* be an optimal vertex cover. Define a random variable:

$$X_u = \begin{cases} w(u) & \text{if } u \in C; \\ 0 & \text{otherwise.} \end{cases}$$

We compute the expected weight of the cover $w(C) = \sum_{v \in V} X_v$:

$$E(w(C)) = E\left(\sum_{v \in V} X_v\right) = \sum_{v \in V} E(X_v) = \sum_{v \in V} e_v,$$

where we denote $e_v = E(X_v)$. We claim that the approximation factor is 2. This is equivalent to the following

Let C^* be an optimal vertex cover. Let $C' = C \cap C^*$. Thus $w(C') = \sum_{v \in C^*} X_v$ and $E(w(C')) = \sum_{v \in C^*} e_v \leq w(C^*)$.

Theorem 40.1 $E(w(C)) = \sum_{v \in V} e_v \leq 2E(w(C')) \leq 2w(C^*)$.

Proof. Think of e_v as money put on vertex v. We want to distribute all money from vertices to edges so that each edge gets equal amount of money from its both endpoints. The edge now returns all its money to a vertex that is in C^* (at least one such vertex exists). Each vertex in C^* has at most double the amount of money it started with, and vertices not in C^* have no money. Thus the total amount of money at vertices is at most $2\sum_{v \in C^*} e_v = 2E(w(C'))$.

We now show how to do the distribution of the e_v values so that each edge gets the same amount of money from each end point. Define a new random variable:

$$X_{u,v} = \begin{cases} w(u) & \text{if } u \text{ is picked upon processing the critical edge } (u, v); \\ 0 & \text{otherwise.} \end{cases}$$

An edge is called *critical* if it is picked by the randomized algorithm. We have

$$X_u = \sum_{v \in V} X_{u,v}, \quad E(X_u) = E\left(\sum_{v \in V} X_{u,v}\right) = \sum_{v \in V} E(X_{u,v}).$$

Let $E(X_{u,v})$ be the amount of money transferred from vertex u to edge (u,v). We have $E(X_{u,v}) = E(X_{v,u})$, since

$$E(X_{u,v}) = w(u)\frac{w(v)}{w(u)+w(v)}\Pr((u,v) \text{ is critical}),$$

$$E(X_{v,u}) = w(v)\frac{w(u)}{w(u)+w(v)}\Pr((u,v) \text{ is critical}).$$

This tells us that a distribution scheme with the desired properties exists. We can pay at least the same amount of money to each edge, if initially the money is only in C^* vertices and each vertex $v \in C^*$ has $2e_v$ money units. Therefore $2\sum_{v \in C^*} e_v \geq \sum_{v \in V} e_v$ which justifies our claim.

CMSC 651 Advanced Algorithms	Lecture 25
Lecturer: Samir Khuller	Thu April 22, 1993

Notes by Marga Alisjahbana.

41 Steiner Tree Problem

We are given an undirected graph $G = (V, E)$, with edge weights $w : E \rightarrow R+$ and a special subset $S \subseteq V$. A Steiner tree is a tree that spans all vertices in S, and is allowed to use the vertices in $V - S$ (called steiner vertices) at will. The problem of finding a minimum weight Steiner tree has been shown to be NP-complete. We will present a fast algorithm for finding a Steiner tree in an undirected graph that has an approximation factor of $2(1 - \frac{1}{|S|})$, where $|S|$ is the cardinality of of S.

41.1 Approximation Algorithm

Algorithm :

1. Construct a new graph $H = (S, E_S)$, which is a complete graph. The edges of H have weight $w(i,j)$ = minimal weight path from i to j in the original graph G.
2. Find a minimum spanning tree MST_H in H.
3. Construct a subgraph G_S of G by replacing each edge in MST_H by its corresponding shortest path in G.
4. Find a minimal spanning tree MST_S of G_S.
5. Construct a Steiner tree T_H from MST_S by deleting edges in MST_S if necessary so that all leaves are in S (these are redundant edges, and can be created in the previous step).

Running time :

1. step 1 : $O(|S||V|^2)$
2. step 2 : $O(|S|^2)$
3. step 3 : $O(|V|)$
4. step 4 : $O(|V|^2)$
5. step 5 : $O(|V|)$

So worst case time complexity is $O(|S||V|^2)$.

Will will show that this algorithm produces a solution that is at most twice the optimal. More formally: weight(our solution) $\leq weight(MST_H) \leq 2 \times$ weight(optimal solution)

To prove this, consider the optimal solution, i.e., the minimal Steiner tree T_opt.

Do a DFS on T_opt and number the points in S in order of the DFS. (Give each vertex a number the first time it is encountered.) Traverse the tree in same order of the DFS then return to starting vertex. Each edge will be traversed exactly twice, so weight of all edges traversed is $2 \times weight(T_opt)$. Let $d(i, i+1)$ be the length of the edges traversed during the DFS in going from vertex i to $i+1$. (Thus there is a path $P(i, i+1)$ from i to $i+1$ of length $d(i, i+1)$.) Also notice that the sum of the lengths of all such paths $\sum_{i=1}^{|S|} d(i, i+1) = 2 \cdot MST_S$. (Assume that vertex $|S|+1$ is vertex 1.)

We will show that H contains a spanning tree of weight $\leq 2 \times w(T_opt)$. This shows that the weight of a mininal spanning tree in H is no more than $2 \times w(T_opt)$. Our steiner tree solution is upperbounded in cost by the weight of this tree. If we follow the points in S, in graph H, in the same order of their above DFS numbering, we see that the weight of an edge between points i and $i+1$, in H, cannot be more than the length of the path between the points in MST_S during the DFS traversal (i.e., $d(i, i+1)$). So using this path we can obtain a spanning tree in H (which is actually a Hamilton Path in H) with weight $\leq 2 \cdot w(T_opt)$.

Figure 33 shows that the worst case performance of 2 is achievable by this algorithm.

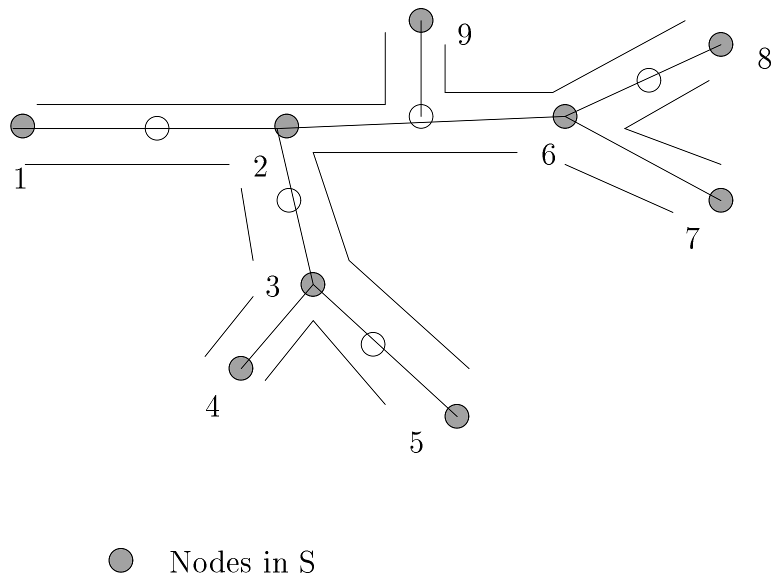


Figure 32: Optimal Steiner Tree

41.2 Steiner Tree is NP-complete

We now prove that the Steiner Tree problem is NP-complete, by a reduction from 3-SAT to Steiner Tree problem

Construct a graph from an instance of 3-SAT as follows:

Build a gadget for each variable consisting of 4 vertices and 4 edges, each edge has weight 1, and every clause is represented by a node.

If a literal in a clause is negated then attach clause gadget to F of corresponding variable in graph, else attach to T. Do this for all literals in all clauses and give weight M (where $M \geq 3n$) to each edge. Finally add a root vertex on top that is connected to every variable gadget's upper vertex. The points in S are defined as: the root vertex, the top and bottom vertices of each variable, and all clause vertices.

We will show that the graph above contains a Steiner tree of weight $mM + 3n$ if and only if the 3-SAT problem is satisfied.

If 3-SAT is satisfiable then there exists a Steiner tree of weight $mM + 3n$ We obtain the Steiner tree as follows:

- Take all edges connecting to the topmost root vertex, n edges of weight 1.
- Choose the T or F node of a variable that makes that variable "1" (e.g. if $x = 1$ then take T, else take F). Now take the path via that node to connect top and bottom nodes of a variable, this gives $2n$ edges of weight 1.
- If 3-SAT is satisfied then each clause has (at least) 1 literal that is "1", and thus connects to the T or F node chosen in (2) of the corresponding variable. Take this connecting edge in each clause; we thus have m edges of weight M, giving a total weight of mM. So, altogether weight of the Steiner tree is $mM + 3n$.

If there exists a Steiner tree of weight $mM + 3n$ then 3-SAT is satisfiable. To prove this we note that the Steiner tree must have the following properties:

- It must contain the n edges connecting to the root vertex. There are no other edges connected nodes corresponding to different variables.

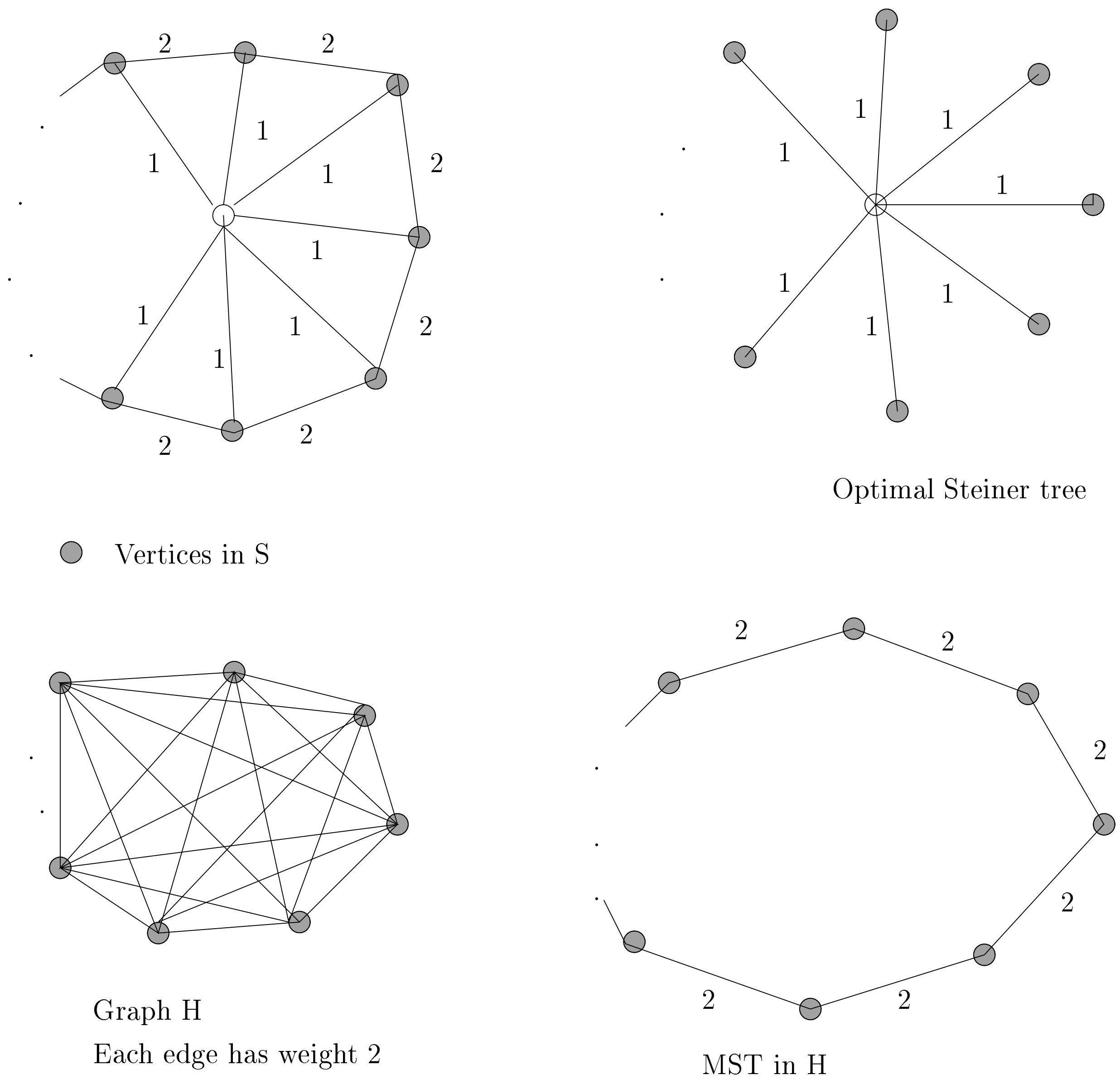


Figure 33: Worst Case ratio is achieved here

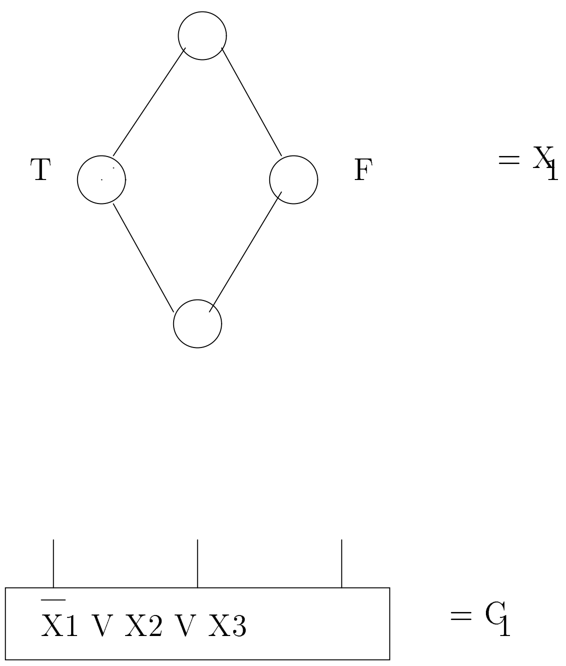


Figure 34: Gadget for a variable

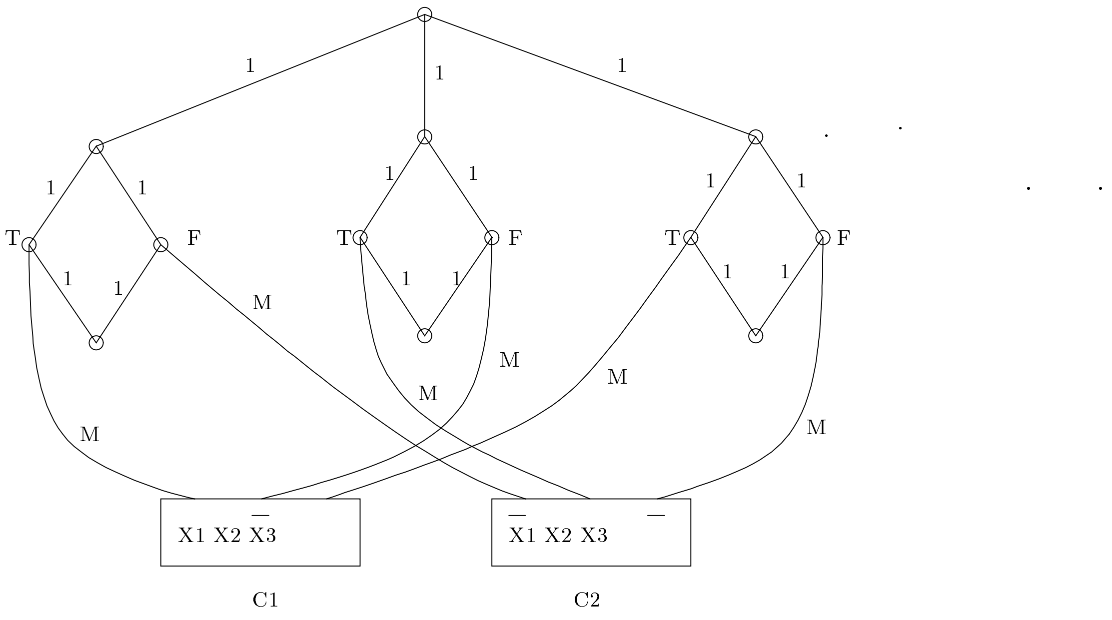


Figure 35: Reduction from 3-SAT.

- It must contain one edge from each clause node, giving a weight of mM. Since M is big, we cannot afford to pick two edges of weight M from a single clause node. If it contains more than one edge from a clause (e.g. 2 edges from one clause) the weight would be $mM + M > mM + 3n$. Thus we must have exactly one edge from each clause in the Steiner tree.
- For each variable must have 2 more edges via the T or the F node.

Now say we set the variables to true according to whether their T or F node is in the Steiner tree (if T then $x = 1$, if F then $x = 1$.) We see that in order to have a Steiner tree of size $mM + 3n$, all clause nodes must have an edge connected to one of the variables (i.e. to the T or F node of that variable that is in the Steiner tree), and thus a literal that is assigned a "1", making 3-SAT satisfied.

CMSC 858K Advanced Algorithms	Lectures 10 and 11
Lecturer: Samir Khuller	Mar 2 and 4, 1999

Notes by Alexandros Labrinidis and Samir Khuller

42 Traveling Salesperson and Chinese Postman Problem

In this lecture some routing problems were presented. Generally, in routing problems we are given a graph G (typically undirected, but can also be directed or even mixed), and we are asked to find the best route on the graph based on a completeness criterion (i.e. visit all vertices/edges) and a weight/distance function which we are trying to minimize.

42.1 Traveling Salesperson Problem

In the Traveling Salesperson Problem (TSP for short), a salesperson has to visit n cities. If we model the problem as a complete graph $G = (V, E)$ with n vertices, we can say that the salesperson wishes to make a tour, visiting each city exactly once and finishing in the same city s/he started from. There is a cost $c_{i,j}$ associated with traveling from city i to city j and the salesperson wishes to minimize that cost.

Definition 42.1 (Euclidean TSP) *Given a set of points in the plane, $S = \{s_1, s_2, \ldots, s_n\}$, and a Euclidean distance function $d_{i,j}$, find the shortest tour that visits all the points.*

Euclidean TSP was proven $\mathcal{NP}$*-hard*, so we only hope to approximate it, if we want to have an algorithm with polynomial time complexity. In fact, recently Arora and Mitchell independently announced a $(1 + \epsilon)$ approximation.

Heuristics:

- We can take the greedy approach and always try to visit the next unvisited node that is closest to our current position.

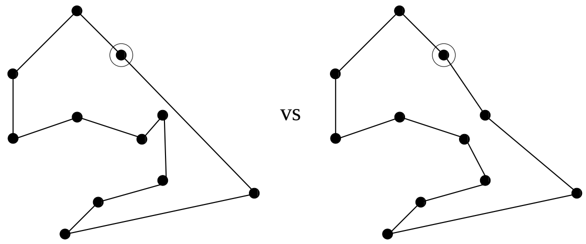


Figure 36: Greedy heuristic

- Because the triangular inequality holds, we can always eliminate *"crossing tours"* and get better tours.

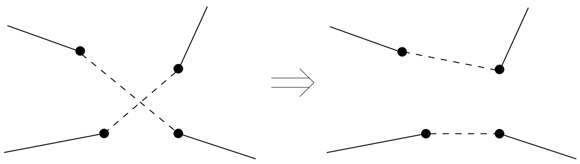


Figure 37: Eliminating Crossing tours

- We can simply focus on an incremental algorithm which, based on an existing tour, tries to find the best possible "augmentation" to insert an "new" unvisited city to the tour.

Minimum Spanning Tree Another idea for finding a solution/tour for the *TSP* is to find a Minimum Spanning Tree for the given graph. After we've found the *MST* we can simply take a depth-first traversal of the tree. This guarantees the completion criterion for the TSP, but the cost of the resulting path can be as bad as twice the cost of MST. An improvement to this is trying to make "shortcuts" whenever possible (in other words we want to try and not revisit any node, advancing to the next unvisited node in the traversal without going through the "back" steps). This is feasible because of triangular inequality and clearly improves the total cost.

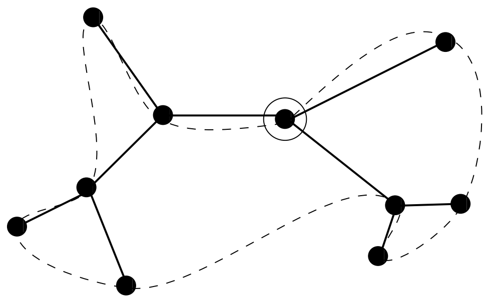


Figure 38: Minimum Spanning Tree

We have $c_{MST} < c_{OPT}$, since if we take out one edge of the optimal tour then we would have a tree that spans the entire graph (which might not be of minimum cost).

Q: How to convert spanning trees into an optimal tour? Basically, we need to make the graph *Eulerian*, in other words guarantee that $\forall v \in V, degree(v)$ is even (or in the case of directed graphs: *in-degree*(v) = *out-degree*(v)). This would imply that for every node we visit we can always exit that node so that we don't ever get "stuck" at one place. So, since we have to visit every node at least once (and thus don't mind visiting the same vertex more than once), augmenting an MST to become an Eulerian graph should be a reasonable solution to the TSP.

Claim: *There is an even number of odd degree vertices in any graph.*

We know that $2 \times |E| = \sum_{\forall v \in V} d(v)$. If we partition V into two disjoint sets V_O and V_E, so that V_O contains all the vertices with odd degree, and V_E contains all the vertices with even degree, we have:

$$2 \times |E| = \sum_{\forall v \in V_O} d(v) + \sum_{\forall v \in V_E} d(v)$$

Since both $2 \times |E|$ and $\sum_{\forall v \in V_E} d(v)$ are even, we have that $\sum_{\forall v \in V_O} d(v)$ should also be even, which implies that the number of vertices with odd degree is an even number. $\square$

One idea that can help us transform an MST into a Eulerian graph, is to "pair up" the odd degree vertices (by adding more edges to the graph - we can do this since we are given a distance metric) and thus "force" all vertices to have even degree. To decide on the "pairs" we can simply find a minimum-weight maximum-matching on the distance graph of the odd degree nodes.

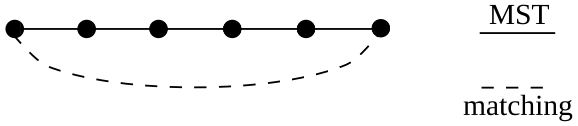


Figure 39: Matching *vs* MST size

We want to quantify how good an approximation we can achieve with this solution. However, there is no way to reasonably bound the weight of the produced matching with respect to the weight of the *MST*, since in the worst case they can even be equal (see figure 39).

However, we have more luck when we try to compare the size of the matching against the size of the optimal tour.

Claim: $w(m_S) \leq \frac{1}{2}w(OPT)$

Say that matching$_A$ and matching$_B$ are the two different matchings that can be derived from the minimum-weight maximum-matching on the distance graph of the odd degree nodes. $w(m_A)$ and $w(m_B)$ are their respective sizes.

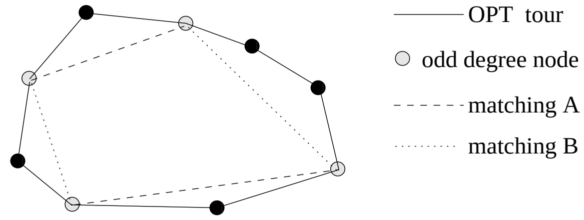


Figure 40: Matching *vs* OPT size

Since triangular inequality holds, we have that:

$$w(m_A) + w(m_B) \leq w(OPT)$$

where $w(OPT)$ is the size of the optimal tour.

If we pick the smallest of the two matchings (named m_S) than we have that $2 \times w(m_S) \leq w(m_A)+w(m_B)$, which in turn gives us $w(m_S) \leq \frac{1}{2}w(OPT)$. $\square$

So the overall size of our solution is:

$$w(MST) + w(m_S) \leq w(OPT) + \frac{1}{2}w(OPT) = 1.5w(OPT)$$

and therefore $\alpha = 1.5$.

42.2 Traveling Salesperson Path Problem

One variation of the TSP is the Traveling Salesperson Path Problem. In this case, instead of trying to find the best tour, we simply want to find the best *path* that would cover all the cities. There are three obvious sub-variations:

- The path can start at any city and end at any city.
- The first point of the path is fixed, but the end can be any other point in the graph.
- Both endpoints are fixed and given to us.

We will consider the third variants here. Let s and t be two specified vertices in G. We consider the problem of finding a path with s and t as its two ends, that visits all vertices of G. One can solve this problem in a manner similar to Christofides' algorithm for TSP, by starting with $MST(G)$, adding a suitable matching, and then finding an Eulerian walk of the resulting graph. We do not get a $\frac{3}{2}$ approximation ratio since in the TSPP the optimal solution is a path, and not a tour. The bound of $\frac{1}{2}OPT$ on the weight of a minimum-weight perfect matching of a subset of the vertices, which holds for TSP (tour), does not hold here.

We obtain a $\frac{5}{3}$ approximation ratio as follows.

Theorem 42.2 *There exists a polynomial-time algorithm for TSPP between given end vertices s and t, that finds solutions S_1 and S_2 which satisfy the following equations:*

$$l(S_1) \leq 2\ MST(G) - l(s,t)$$

$$
\begin{aligned}
&\leq \; 2\ OPT - l(s,t), \\
l(S_2) &\leq \; MST(G) + \frac{1}{2}(OPT + l(s,t)) \\
&\leq \; \frac{3}{2}\ OPT + \frac{1}{2}\ l(s,t).
\end{aligned}
$$

Proof:
We "double" the edges of $MST(G)$ except for those on the unique $s-t$ path on it. The result is a connected multigraph whose vertex degrees are all even except for those of s and t. We now find an Eulerian walk between s and t on this multigraph and turn it into a Hamiltonian path between s and t without increasing the weight by shortcutting and applying the triangle inequality. We call it S_1. The length of S_1 is at most $2MST(G) - l(s,t)$, which is at most $2OPT - l(s,t)$.

To obtain S_2, we adopt the following strategy. Consider adding the edge (s,t) to OPT, and making it a cycle. The length of this cycle is $OPT + l(s,t)$. The cycle can be decomposed into two matchings between any even-size subset of vertices, and the length of the smaller matching is at most $\frac{1}{2}(OPT + l(s,t))$. We add to $MST(G)$ a minimum-weight matching of vertices selected based on their degree in $MST(G)$ (odd degree vertices of $V \setminus \{s,t\}$ and even degree vertices of $\{s,t\}$ in $MST(G)$), and output an Eulerian walk S_2 from s to t in the resulting graph. This Eulerian walk can be converted into a Hamiltonian path by shortcutting. Using the triangle inequality, we obtain that $l(S_2)$ is at most $MST(G) + \frac{1}{2}(OPT + l(s,t)) \leq \frac{3}{2}OPT + \frac{1}{2}l(s,t)$.
□

Corollary 42.3 *The shorter of the paths S_1 and S_2 is at most $\frac{5}{3}OPT$.*

Proof:
Observe that if $l(s,t) > \frac{1}{3}OPT$, then $l(S_1) \leq \frac{5}{3}OPT$. Otherwise, $l(s,t) \leq \frac{1}{3}OPT$, in which case $l(S_2) \leq \frac{5}{3}OPT$.
□

42.3 Chinese Postman Problem

Given:

- a connected graph $G = (V, E)$
- a cost function $c : E \rightarrow Z^+$

Find: a walk in which we traverse every edge at least once, such that the total cost of the walk (with multiple traversals of an edge charged multiply) is as small as possible.

For simple undirected graphs there is a polynomial time algorithm to solve the problem optimaly. For mixed graphs (were we have both undirected and directed edges) the problem was proven to be $\mathcal{NP}$*-complete*, but there are approximation algorithms.

We will first review the algorithm for undirected graphs and then give some more information about the two algorithms for mixed graphs.

Undirected graphs We observe that if $G = (V, E)$ is an Eulerian graph (every vertex has even degree), then it is easy to find an optimal solution, since each edge will be traversed exactly once in an Euler tour. In any tour that starts and ends at the same vertex, each node is entered and left exactly the same number of times. Hence, for the case that the graph is not Eulerian, we are looking for the lowest-weight set of edges whose addition to G will make each vertex have even degree.

The solution to this is to simply find a minimum weight perfect matching, m, on the set of odd degree vertices of G. If $V_O \subset V$ is the set of vertices with odd degree, we know that $|V_O|$ is even. Let $G_O = (V_O, E_O)$ be the complete graph on the vertices of V_O, where the weight for each edge $e_{i,j} \in E_O$ is the weight of the shortest path on G between vertex i and vertex j. Find a minimum weight perfect matching on G_O, and add the paths corresponding to the edges in the matching to G. This creates a new multigraph G', where each vertex has even degree. Find an Euler tour in this multigraph. This can be done in linear time (linear in the size of G').

To prove that this is optimal, let $w(G)$ be the total weight of the edges in G. Let $w(m)$ be the total weight of the matching on G_O. We will now prove that *any* tour has cost at least $w(G) + w(m)$ (which is the cost of our solution). Let T be any Chinese Postman tour. This tour must enter and leave each vertex exactly the same number of times. Let's create multiple copies of an edge when it is traversed more than once. The resulting multigraph T' has the property that each vertex has even degree. So, we have that $\ell(T) = w(T') = w(G) + w(G'')$, where G'' is the subgraph obtained by removing E from T'.

If v is a a vertex having odd degree in G, then it must have odd degree in G''. If it has even degree in G, then it must have even degree in G''. Clearly, if V_O is the set of odd degree vertices in G, then G'' is a possible solution to the problem of finding a minimum weight subgraph where the vertices in V_O have odd degrees and the other vertices have even degrees. The minimum weight perfect matching problem solves this problem optimally. Hence $w(m) \leq w(G'')$ (since we found an optimal solution to the matching problem), and $\ell(T) = w(G) + w(G'') \geq w(G) + w(m)$, implying that the solution we found is optimal.

CMSC 858K Advanced Algorithms	Lecture 18 April 10, 1997

Notes by Alexander Dekhtyar

43 Vehicle Routing Problems:Stacker Crane

This part summarizes results presented by Frederickson, Hecht and Kim (SICOMP 78)

Problem. We are given two sets of points $\{A_1, \ldots, A_n\}$ (we will call them *pegs*) and $\{B_1, \ldots, B_n\}$ (we will call them *slots*). Without loss of generality we can assume that all A_i and B_j correspond to some points on the plane. Each point A_i is thought to contain a peg that has to be deleivered to a corresponding slot B_i (for all $1 \le i \le n$). The crane operator has to deleiver all the pegs to appropriate slots. A crane can only hold one peg at a time. The goal is to produce an optimal tour that would pass through all peg and slot points and deleiver all pegs to the designated slots.

TSP can be reduced to the Stacker Crane problem. Consider an instance of the Stacker Crane problem where A_i coincides with (or lies very close to) B_i for all $1 \le i \le n$. The optimal Stacker Crane tour then will also be an optimal TSP tour.

In what follows we present two approximation algorithms for SCP and show that both will give a 2-approximation. Then we will show that for each instance of the problem the better of the two solutions will give us a 1.8-approximation.

The two algorithms will employ two polar ideas. One of them will work gell in the situation when the distances between each peg and its designated slot are relatively short compared to the distances that the crane will have to travel between the delieveries. Second algorithm will be based on the assumption that the distances that the crane has to cover delivering the pegs exceed by far the distances that the crane has to travel between the deliveries.

43.1 Algorithm 1

We will call a route between any peg A_i and corresponding to it slot B_i a *delivery route*. Generally speaking every Stacker Crane route consists of 2 parts:

1. all delivery routes
2. a tour between different delivery routes

The main assumption this algorithm employs is that the delivery routes are short compared the length of the tour that connects these routes. If that is the case - we would like very much to find the best possible tour connecting the delivery routes. The price we might wind up paying for such tour is that we might be forced to traverse delivery routes more than once. But if delivery routes are short compared to the length of a tour connecting them, it is a reasonable price to pay. As a “good” connecting tour we will be looking for a “reasonably” short TSP tour between all delivery routes.

1. We construct a complete graph G', with the set of nodes $V = \{v_1, \ldots, v_n\}$. Each v_i corresponds to the delivery route A_i to B_i.
2. For each pair of vercices $v_i, v_j \in V$ we set $d(v_i, v_j) = \min\{d(A_i, A_j), d(B_i, B_j), d(A_i, B_j), d(B_i, A_j)\}$.

The algorithm will do the following:
Algorithm 1.

1. Construct G'.
2. Solve TSP on G'.
3. Let (v_i, v_j) be the edge of a computed in G' TSP tour. Let $d(v_i, v_j) = d(D_i, D_j)$, where $D_i \in \{A_i, B_i\}, D_j \in \{A_j, B_j\}$. The part of the Stacker Crane tour corresponding to it will be: $A_i \to B_i \to D_i \to D_j \to A_j$.

4. Using the conversion from (3) compute and return the Stacker Crane tour.

Let us now see how well this algorithm does. Let T be the cost of computed TSP tour on G'. Let C_A be the cost of all deliveries ($C_A = \sum_{i=1}^{n} d(A_i, B_i)$). We show that the total cost of the Stacker Crane tour (SCT) computed by *Algorithm 1* is:

$$SCT \leq T + 2C_A$$

Let us look at the two tours SCT_1 and SCT_2: one is the Stacker Crane tour T followed in counterclockwise direction, the other is the Stacker Crane tour T followed in clockwise direction. We can see that $SCT_1 + SCT_2 = 2T + 4C_A$ which will give us $SCT = min(SCT_1, SCT_2) \leq T + 2C_A$.

From previous material we know that we can expect $T \leq 1.5T^*$ where T^* is the cost of an optimum TSP tour on G'. Let SCT^* be the optimum stacker crane tour. It must be that $C_A \leq SCT^*$. Also, it must be that $T^* \leq SCT^* - C_A$ (i.e, optimum Stacker Crane tour save the deliveries part must be sufficient to connect all delivery routes). Finally the combining with the previously computed bound for SCT we get:

$$SCT \leq T + 2C_A \leq 1.5T^* + 2C_A \leq 1.5(STC^* - C_A) + 2C_A \leq 1.5STC^* + 0.5C_A \leq 2STC^*$$

i.e. *Algorithm 1* gives us a 2-approximation.

As you can see - the reason why this algorithm gives us 2-approximation is because we bound C_A with the size of the optimal tour. However if C_A is much smaller than STC^* then this algorithm will give a better solution.

43.2 Algorithm 2

Second algorithm, in contrast with the first one will be designed under the assumption that the length of all delivery routes (C_A in our adopted in previous subsection notation) is relatively big (greater than the length of the tour between the delivery routes).

In this case we do not want to risk travelling along the delivery route "emptyhanded" -as this might be a large distance. We would prefer to find such a tour that would take us from the slot to the peg for the next delivery. Now the tour connecting delivery routes might be bigger, but we will travel along each delivery route only once instead of 2 times as it is expected in Algorithm 1 On the other hand we still want a the connecting tour to be reasonably small.

We will find such a tour in two steps. First we will establish minimum-weight perfect matching on the bi-partite graph $G_1 =< \{A_1, \ldots, A_n\}, \{B_1, \ldots, B_n\}, \{(A_i, B_j) | i \neq j\}$. This matching might still not be sufficient to establish a tour (it could be that the perfect matching breaks G into a number of cycles). We will then consider a complete graph of such cycles and we will find its minimum spanning tree. This will give us a Stacker Crance route with the important property that it travels along every delivery route only once (in the direction from peg to slot).

Algorithm 2.

1. Construct bipartite graph $G_1 =< \{A_1, \ldots, A_n\}, \{B_1, \ldots, B_n\}, \{(A_i, B_j) | i \neq j\}$.
2. Find a minimum weight perfect matching M on graph G_1 (notice that A_i has no edge to B_i in G_1 for all $1 \leq i \leq n$ and therefore, A_i cannot be matched to B_i).
3. If M is a tour connecting all delivery routes, than $M + C_A$ is our Stacker Crane route. Otherwise,
4. M breaks G_1 into a number of cycles. Construct (complete) graph G_2 where the nodes are *all* cycles $c_1, \ldots, c_k$, M induces on G_1 and $d(c_i, c_j) = \min_{C_i \in c_i; C_j \in c_j}(d(C_i, C_j))$.
5. Find minimum spanning tree $ST(G_2)$. The final Stacker Crane route will be as follows:
   (a) Start at some cycle c of G_2 (choose the starting point among the pegs in that cycle).
   (b) While inside the cycle, do deliveries in clockwise direction using matching M to choose next delivery route, *until* a vertex that induced an edge in $ST(G_2)$ is reached.
   (c) Follow that vertex. Goto step 2. Stop when all deliveries are made.

The figure below illustrates the Stacker Crane route found by Algorithm 2.

Let us now find out how good (or bad) this algorithm is. As usual, let STC denote the length of the Stacker crane route found by Algorithm 2, and let STC^* be the optimum Stacker Crane route. Let $w(M)$ denote the weight of matching M.

It is easy to see that $w(M)+C_A \leq STC^*$. In fact, any Stacker Crane route (including optimal) has to do all the deliveries (that is covered by C_A) and has to connect all the delivery routes with each other somehow. This connection of delivery routes induces a perfect matching on graph G_1 But M is a mimnimum weight matching in this graph, so, we have to conclude that the inequality above holds.

Let now $w(ST)$ be the wieght of minimum spanning tree of G_2. ¿From the decription of the Stacker Crane route created by Algorithm 2 we see that:

$$STC \leq w(M) + C_A + 2w(ST)$$

Optimal tour STC^* consits of two parts: delivery routes and tour connecting delivery routes. This second part can be thought of as a spanning tree on the graph where each cycle is just one delivery route. It is clear that our spanning tree $ST(G_2)$ is not longer than this tour in optimal route. Therefore, we also obtain: $w(ST) \leq STC^* - C_A$.

Combining our knowledge about $w(M)$ and $w(ST)$ together we obtain the bound on STC:

$$STC \leq w(M) + C_A + 2w(ST) \leq STC^* + 2(STC^* - C_A) = 3STC^* - 2C_A$$

When $C_A \geq 0.5STC^*$ (and this is exactly the case for which we designed Algorithm 2) we obtain

$$STC \leq 2STC^*$$

43.3 Combining both results

In order to find a good Stacker Crane route we may run *both* Algorithm 1 and Algorithm 2 and then choose a better solution. Surprisingly (?) this simple combination of two methods allows us to prove a better approximation bound.

Indeed, we know that our solution STC will be bounded by the following :

$$STC \leq 1.5STC^* + 0.5C_A \quad \text{(Algorithm 1)}$$
$$STC \leq 3STC^* - 2C_A \quad \text{(Algorithm 2)}$$

Therefore we can always assume that

$$STC \leq \min(1.5STC^* + 0.5C_A, 3STC^* - 2C_A)$$

The worst possible case will be if

$$1.5STC^* + 0.5C_A = 3STC^* - 2C_A$$

i.e.,

$$C_A = 0.6STC^*$$

Plugging it in we get $STC \leq 1.8STC^*$, which is a better than a 2-approximation obtained by each single algorithm.

CMSC 858K Advanced Algorithms	Lecture 19
Lecturer: Samir Khuller	Apr 15 , 1997

Notes by Zhexuan Song and Samir Khuller.

44 Vehicle Routing Problems: Capacity Bounded Vehicles

Read "Algorithms for Robot Grasp and Delivery" by Chalasani, Motwani and Rao. [CMR]

K-Delivery TSP Given: n source points and n sink points in some metric space, with exactly one pegs placed at each source (all pegs are identical), and a vehicle of capacity k,

Find: A minimum length route for the vehicle to deliver exactly one peg to each sink.

The capacity k may be unbounded. This problem is easily seen to be NP-hard via a reduction to TSP: pick $k = 1$ and put sink very close to source. In the rest of the notes, we give a 2.5-approximation algorithm for $k = 1$, then we will discuss a 9.5-approximation algorithm in [CMR] for arbitrary k, after some improvement, the ratio will finally be 6.5. Then we will talk about algorithms for unbounded k.

We will refer to the sources as pegs or red points and the sinks as slots or blue points. Also we will assume for convenience that n is a multiple of k.

45 Unit Capacity Problem (1-delivery TSP)

Definition 45.1 (Unit capacity problem) Given an edge-weighted graph G with n pegs and slots, find the optimal bipartite tour starting and ending at a designated peg s and visiting all vertices. A tour called bipartite if every pair of adjacent points in it must be peg and slot.

Anily and Hassin have shown a 2.5-approximation algorithm. Suppose A is a minimum-weight perfect matching. T is a TSP tour of pegs. The final route consists of visiting the red vertices in the sequence specified by the tour T, using the matching edges in A to deliver a part to a slot and return to peg (or go to the next peg in the T). So

$$\text{Our cost} = T + 2A.$$

T is a tour of red vertices, Using Christofides' heuristic, we can find a 1.5-approximation tour of pegs. Since any OPT tour must cover all red vertices. So we have

$$T \leq 1.5 \text{ OPT (red vertices)} \leq 1.5\text{OPT}$$

Each OPT solution looks like s (peg) $\rightarrow$ slot $\rightarrow$ peg $\rightarrow$ slot $\rightarrow \cdots \rightarrow$ slot $\rightarrow$ s (peg). We can easily divide the tour into two perfect matching M_1 and M_2. Since A is a minimum-weight perfect matching, we have $M_1, M_2 \geq A$. So

$$2 \times A \leq M_1 + M_2 = \text{OPT}$$

and

$$\text{Our cost} = T + 2 \times A \leq 1.5 \text{ OPT} + \text{OPT} = 2.5 \text{ OPT}.$$

In fact, now the best algorithm is 2-approximation. We can find it in [CMR]. The central idea of this algorithm is try to find a bipartite spanning tree T where each red vertex has degree at most 2. Then T has exactly one red vertex s of degree 1. If T is rooted at s then every red vertex has exactly one child. Clearly, if we traverse this tree T in depth-first order (in fact any order), the sequence of vertices visited are of alternating color. Clearly, the OPT bipartite tour contains a bipartite spanning tree where all red vertices have degree at most 2. Therefore the weight of the minimum-weight bipartite spanning tree whose red vertices have degree at most 2 is a lower bound on OPT. Then the problem changes to how to find such a tree. Here, they use some new tool called "matroid" to achieve the result.

46 K-capacity problem

When k is arbitrary, we will not get a 2.5-approximation solution by using above algorithm. For example,

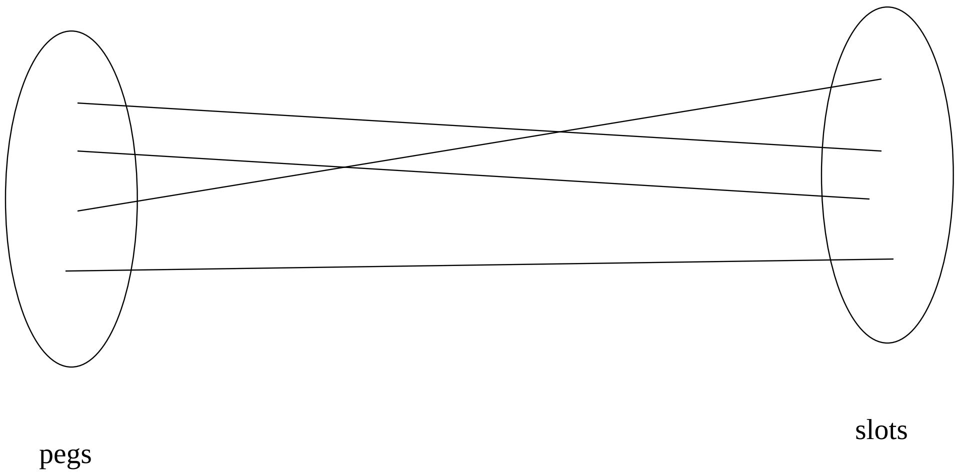


Figure 41: An example of k-capacity problem

We have n pegs and n slots. All the pegs are very close, and so are the slots. At the same time, the distance between slots and pegs is quite large. One good solution should be: first pick k pegs then go to other side and put them down, then go back and pick another k pegs and so on. The cost should be $n/k \times 2 \times M$ where M is the distance between pegs and slots. But if we use the algorithm above, T is almost 0 and A is $n \times M$. So the approximation ratio is (OPT $\times$ a linear function of k). It is a bad solution.

What we do here is try to find a constant approximation algorithm.

9.5-approximation algorithm First, we will talk about the algorithm in [CMR] whose approximation ratio is 9.5.

Let C_k denote the optimal k-Delivery tour, and let C_r and C_b denote the optimal tours on the red and blue points respectively. Let A denote the weight of the minimum-weight perfect matching in the bipartite graph with red vertices on one side and blue vertices on the other. To keep the notation simple, we will often use the same symbol to denote a graph and its weight, and the context will make it clear which one is intended.

Lemma 46.1 $A \leq C_1/2$ *and* $C_1 \leq 2k \times C_k$

Proof:

First part has been proved above, we will not prove second part here because we will improve it later. □

We use Christofides' heuristic to obtain a 1.5-approximation tour T_r of the red vertices and a 1.5-approximation tour T_b of the blue vertices. We omit the proof of the following lemma:

Lemma 46.2 $T_r(T_b) \leq 1.5\times$ $OPT(T_r(T_b)) \leq 1.5 \times C_k$

First we break up T_r and T_b into paths of k vertices each, by deleting edges appropriately. It will be convenient to view each k-path as a "super-node" in the following. Now we have a bipartite graph with n/k super-nodes on each side. Then we try to find a minimum-weight perfect matching in the graph. Suppose this matching called M.

Lemma 46.3 $M \leq A/k$

Proof:

We overlay the minimum-weight perfect matching of cost A on this graph. Note that each super-node now has degree exactly k, and that there may be several edges between two given super-nodes. Think about problem 1 in our homework 1, "Every d-regular bipartite graph always has a perfect matching". If we delete this perfect matching, we left with a $(k-1)$-regular bipartite graph. Repeat the process and finally we partition A into k perfect matching. Among these matching, at lease one with weight less than A/k. Since M is the minimum-weight matching, This proves the lemma. $\square$

After finding out the matching M, we can partition the graph into n/k sub-graphs and label them H_1, H_2, $\cdots H_{n/k}$, each consisting of a red super-node connected via an edge of M to a blue super-node. We can traverse the sub-graph in sequence as follows. Within each sub-graph H_i, first visit all the red vertices, then use the edge of M to go to the blue side and visit all the blue vertices. Then return to the red side, and go to the red vertex that is connect via an edge e of T_b to the next sub-graph H_{i+1}, and use the edge e to go to H_{i+1} (or H_1 if $i = n/k$).

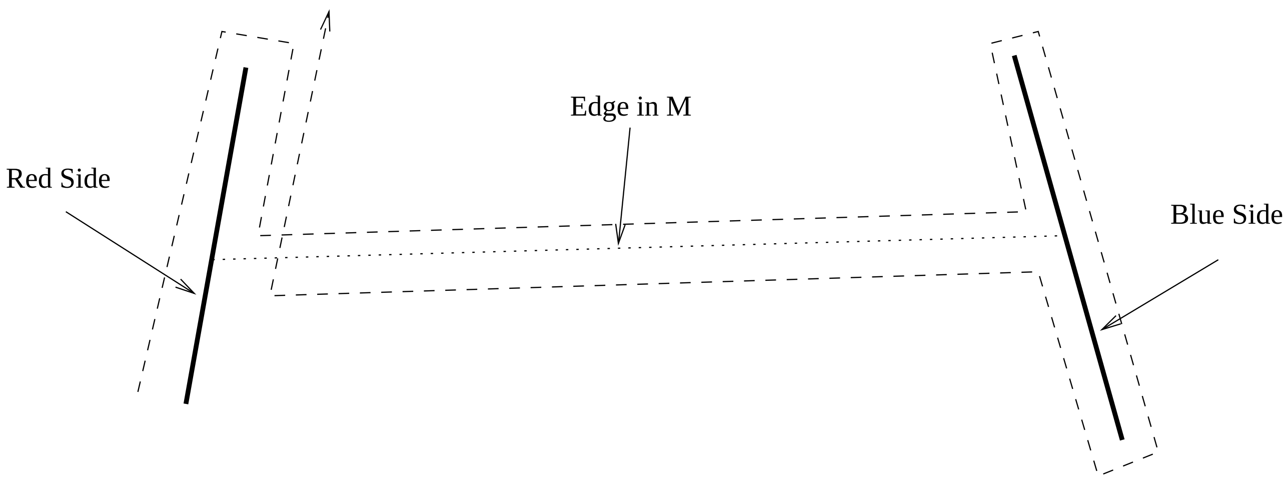


Figure 42: Vehicle route in H_i

We claim that this tour T is within a constant factor of the optimal k-delivery tour. From the above graph, we can find that each edge of T_r is charged no more than 3 times, each edge of T_b is charged no more than twice and each edge of M at most twice. Thus we obtain that

$$T \le 3T_r + 2T_b + 2M \le 3 \times 1.5C_k + 2 \times 1.5C_k + (2/k) \times A \le 9.5C_k$$

Some improvement After carefully observing the algorithm, we have the following modification.

Lemma 46.4 $C_1 \le k \times C_k$

Proof:

Here we omit the detail proof and just give out a demonstration. Suppose we have a car with capacity k and a bicycle with capacity 1. Look at the OPT tour of the car. It looks like first pick some parts and then drop them and then pick some and drop.

Since this is a k-delivery tour, every time the car can pick at most k pegs before it drops them in slots. Now we use a bicycle to simulate the car. The bicycle will pick pegs when it pass a peg point and put it into the first empty slot it meets. Since the capacity of a car is k, there must be at least n/k peg-slot change. That means that at one round, bicycle can at least pick n/k pegs. So bicycle can finish all the work in k rounds. That proves the lemma. $\square$

Now observe Figure 2 about the route of a vehicle in H_i, red side of H_i has been divided into two parts by an edge in M, the upper side has been visited three times and the lower part only once. If the vehicle

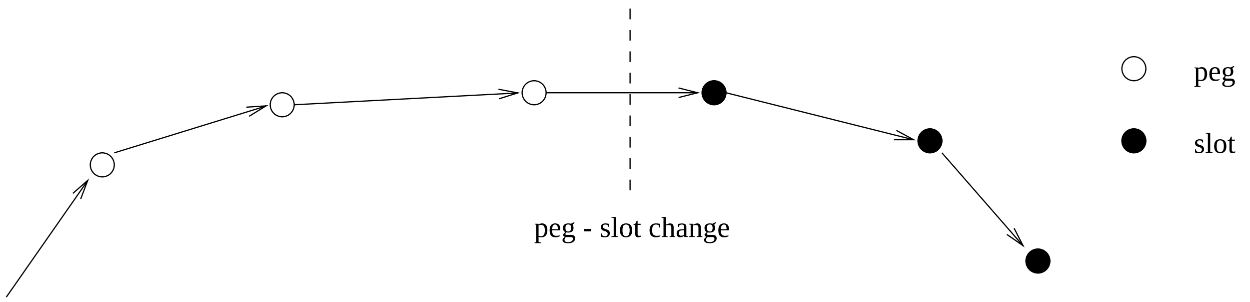


Figure 43: Tour of a k-capacity car

travels in reversed direction, then low part will be three times and upper part will be once. So put these two tour together, total cost should be:

$$\text{Total cost} \leq 4 \times T_r + 4 \times T_b + 4 \times M$$

Pick the one with smaller cost, we have:

$$T \leq 2T_r + 2T_b + 2M \leq 2 \times 1.5C_k + 2 \times 1.5C_k + (2/k) \times A \leq 7C_k$$

More improvement Now we will present another algorithm that improves the approximation ratio to 6.5. A wonderful idea can be found inside it.

1. Find tours T_r and T_b as in the above algorithm.
2. Break T_r and T_b into paths containing $k/2$ vertices each. (Not k.)
3. View each segment of T_r and T_b as a super-node and construct the auxiliary bipartite graph as before.
4. Find a minimum-weight perfect matching M in this graph. Mark the red segments of T_r sequentially as R_1, R_2, $\cdots$ around the cycle. Each edge in M matches a segment in T_r to a segment in T_b. Let the segment matched to R_i called B_i.
5. The vehicle starts at the beginning of segment R_1 and proceeds along the T_b, pick the pegs in its path. When the vehicle reaches R_2, it will travel as above algorithm — visit B_i sometimes and put parts into slots and come back. Find that every time the vehicle encounters the edge in M, there are always at least $k/2$ parts in the vehicle. That is no back trace needed in T_r. This save quit a lot of time.
6. Finally, when the vehicle returns to the starting location, it is carrying $k/2$ pegs that have to be delivered to B_1. It goes over R1 again to reach edge in M, crosses over to B_1 and delivers the pegs to B_1.

Figure 4 demonstrates the algorithm.

The algorithm above generates a valid vehicle routing without violating the capacity constraints of the vehicle due to the following reason. When it is on a segment of R_i that precedes M on T_r, it is carrying $k/2$ pegs from $R_i - 1$ and the other pegs that have been collected from R_i. Since R_i has at most $k/2$ pegs, the total number of pegs that it is carrying does not exceed k. On reaching edge in M, the vehicle goes over to B_i and delivers the $k/2$ pegs that were collected from $R_i - 1$. Therefore when it returns to R_i and resumes its journey, it reaches the end of R_i with $k/2$ pegs that were on R_i.

The reason that the algorithm gives a better approximation ratio is as follows. Observe the graph, we will find that each edge in Tr only be visited once (except for segment R_1) instead of twice. This decreases the length traveled. But the cost of M is now more since there are twice as many segments as before in each of T_r and T_b (because the segment have only $k/2$ vertices each).

Now the cost of the algorithm is:

$$T \leq T_r + 2T_b + 2M + R_1 \leq 1.5C_k + 2 \times 1.5C_k + (2/k) \times 2A + \mathrm{O}(k/n) \leq 6.5C_k + \mathrm{O}(k/n) \leq 6.5C_k$$

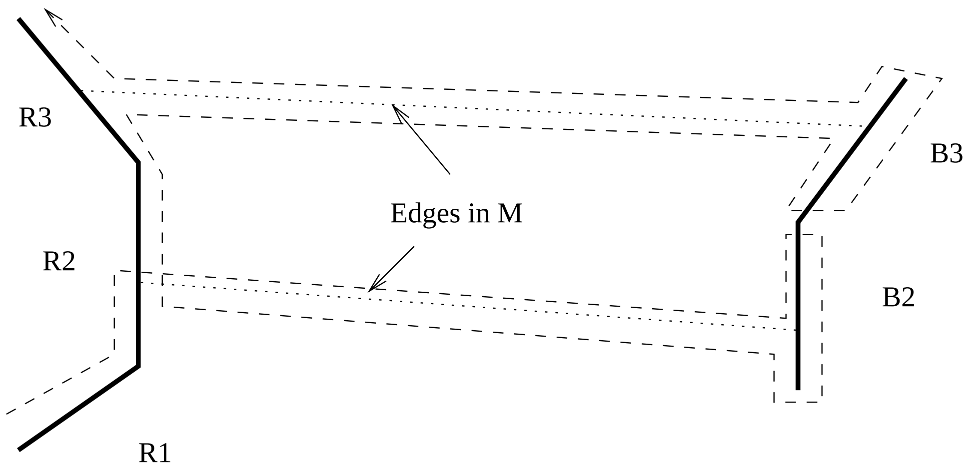


Figure 44: Vehicle route in new algorithm

If we can drop the pegs on the road and deliver them later, we can have a modification of the algorithm that obtained a ratio of a little more than 5.

The algorithm is a modification of the algorithm in the previous part that obtained a ratio of 6.5. Instead of delivering immediately the pegs to the red points in B_i when we cross edge in M, we do the following. The vehicle crosses the edge and leaves $k/2$ pegs there to be delivered to the blue points at a later time. Once the tour along the red cycle is completed, the vehicle switches to the blue tour, and delivers the pegs, but this time picking them up at intermediate points on the blue tour as it reaches them. In all, the vehicle travels around each of the cycle T_r and T_b once, twice around M and once extra on the segments R_1 and B_1. Our algorithm now obtains a tour with a ration of 5+O(k/n).

47 Unbounded capacity problem

Definition 47.1 (Unbounded capacity problem) Given an edge-weighted graph G with n sources and n sinks, find an optimal tour that starts at s and visits all vertices and returns to s in a red-dominant order. An ordering of vertices is red-dominant if in each prefix of the ordering, there are at least as many red vertices as blue vertices.

Easily, we have a 3-approximation algorithm here. First use Christofides' heuristic to find a tour T of all nodes. Then in the first round, we pick up all the pegs and in the second round, we drop them one by one. Total cost is $2T \leq 2 \times 1.5\text{OPT} = 3\text{OPT}$.

This algorithm is simple, so the result is not so good. In fact there is a 2-approximation algorithm in [CMR].

CMSC 858K Advanced Algorithms	Lecture 22
Lecturer: William Gasarch	April 24, 1997

Notes by Yannis Kotidis.

48 Lower Bounds on Approximations

In this lecture we study some recent results on the hardness of approximating NP-hard problems. According to these results, trying to approximate many problems within a certain approximation ratio is NP-hard. This implies that no polynomial-time algorithm achieves that approximation ratio, unless $NP = P$. Graph Coloring is such an example. The problem can be described as follows:

COLORING: Given a graph G, assign a color to each vertex, such that no two adjacent vertices have the same color.

The objective function in this case is the number of colors in a proper coloring. The optimal coloring scheme is the one that uses the minimum number of colors. This number is called the *chromatic number* of graph G and is denoted $x(G)$. Many years back it was shown that a $\frac{4}{3}$-approximation of the COLORING problem is NP-hard : (Garey and Johnson's book shows better hardness results for this problem.) This is just to give you an idea about some basic tools to show these kinds of results.

Theorem 48.1 *If there exists a polynomial-time algorithm A for COLORING such that :*

$$x(G) \leq A(G) < \frac{4}{3}x(G)$$

then $P = NP$.

Proof:
Assume $A \in P$ exists. We will use A to write a polynomial-time algorithm for solving 3-COLORING which is known to be NP-complete.

Algorithm for 3-COLORING:

1. Input G.
2. Create H $= n$ copies of G. Connect all nodes in different copies so that any two nodes in two different parts are connected.
3. Compute $A(H)$ in polynomial-time.
4. If $A(H) < 4n$ output(YES)
   otherwise output(NO).

Its easy to see that because of the way we construct H, if G is 3-Colorable then $x(H) = 3n$, otherwise $x(H) \geq 4n$. This gap is used by the above algorithm to determine if G is 3-Colorable:

- If G is 3-Colorable $\Rightarrow x(H) = 3n \Rightarrow 3n \leq A(H) < 4n$.
- If G is not 3-Colorable $\Rightarrow 4n \leq A(H)$.

$\square$

48.1 Probabilistically Checkable Proofs

Prior to presenting the framework of Probabilistically Checkable Proofs (PCP) we will first give a new (equivalent to the standard one) definition of class NP:

Definition 48.2 $A \in NP$ if $\exists R \in P$, polynomial p such that:

- $x \in A \Rightarrow (\exists y, |y| \leq p(|x|))[R(x,y)]$
- $x \notin A \Rightarrow (\forall y, |y| \leq p(|x|))[\sim R(x,y)]$

In other words $A \in NP$ if there exists a Turing Machine R, such that given an instance $x \in A$, there exists an evidence y of polynomial size, such that R accepts (x,y). On the other hand, if $A \notin NP$ then any proof y will be rejected by the machine.

Starting from this definition we will do two modifications to the Turing Machine that we use :

1. Instead of having $R \in P$ we will use *randomized-P*. This means that we will allow the Machine to flip coins as it processes the input strings. In that case instead of asking if the machine accepts or rejects its input we will be concerned with the probability that either case happens.
2. We will allow R to do indirect accessing on y. This will allow the machine to use extremely long proofs y (like 2^{n^k}), under the restriction that it can query a limited number of bits of y each time.

Definition 48.3 (Probabilistic Checkable Proofs - PCP) $A \in PCP(r(n), q(n))$ if :

- $\exists$ a coin-flipping Turing Machine $M(x,y)$ that makes $O(r(n))$ coin-flips and $O(q(n))$ queries to y,
- $\exists k \in N$ on input (x,y) such that $|x| = n, |y| = 2^{n^k}$

  and

- $x \in A \Rightarrow (\exists y, |y| = 2^{n^k})[Prob(M(x,y) \text{ accepts}) = 1]$
- $x \notin A \Rightarrow (\forall y, |y| = 2^{n^k})[Prob(M(x,y) \text{ accepts}) < \frac{1}{4}]$

Comparing the two definition we see that instead of receiving the entire proof and conducting a deterministic polynomial-time computation (as in the case of NP), the machine (*verifier*) may toss coins and query the proof only at locations of its choice. Potentially, this allows the verifier to utilize a very long proof of super-polynomial length or alternatively inspect very few bits of a polynomially long proof.

Using the PCP-framework we can simplify our definition of NP to: $NP = PCP(0, poly(n))$. For SAT it is straightforward to show that $SAT \in PCP(0,n)$: a valid proof of length n could be the assignments that satisfy the given, n-variable, formula. For 10-COL it is also easy to see that 10-$COL \in PCP(0,4n) = PCP(0,n)$: y can encode the color assignments using 4 bits per color. Nevertheless all these examples do not use the probabilistic power of the verifier. A more interesting example is that $SAT \in PCP(\log n, 1)$, meaning that $NP = PCP(\log n, 1)$ (*PCP Theorem* [AS92,ALMSS92]).

48.2 Example 1

We will now show an example that uses randomness for $\overline{GI}$[6] We will build a machine $M((H_0, H_1), y)$ that takes as input 2 graphs H_0, H_1 and the evidence y and by coin-flipping and querying y is accepts its input[7] if $(H_0, H_1) \in \overline{GI}$.

[6] We remind that two graphs are isomorphic, if we can redraw one of them, so that it looks exactly like the other.

[7] for a "nice" y

Machine $M((H_0, H_1), y)$

1. Flip n coins[8], Get $S_r = 0100011011\ldots01$.
   Match 0 to H_0, 1 to H_1. Get $S_x = H_0H_1H_0\ldots H_0H_1$ from S_r
2. Randomly permute labels of each graph in S_x, and get $S_f = F_1F_2F_3\ldots F_{n-1}F_n$
3. For each graph F_k in S_f find the location i_k of that graph in y
4. Ask about $i_1 \ldots i_n$ bits in y.
   If the bits that you get are the same with those in S_r ACCEPT, else REJECT.

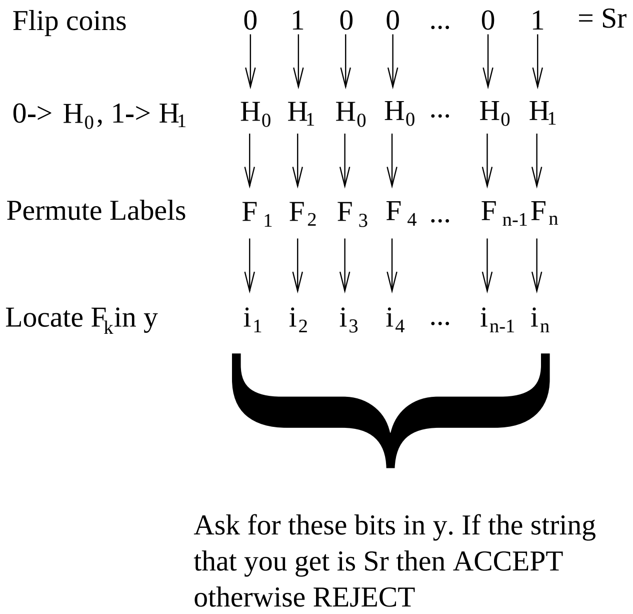


Figure 45: A coin-flipping TM for $\overline{GI}$

Description For a value of n, there are $2^{\binom{n}{2}}$ possible graphs: $G_1, G_2, \ldots, G_{2^{\binom{n}{2}}}$. Evidence y will be exactly $2^{\binom{n}{2}}$ bits long, utilizing exactly one bit for each G_i. Let's assume that H_0 and H_1 are not isomorphic. We must show that there exists a "nice" y such that $Prob(M((H_0, H_1), y)$ accepts$) = 1$. Look at the following way to construct the evidence:

$$i^{th}\text{bit of}y = \begin{cases} 0 & \text{if } G_i \text{ isomorphic to } H_0 \\ 1 & \text{if } G_i \text{ isomorphic to } H_1 \\ 0 & \text{otherwise} \end{cases}$$

Given this evidence, if the graphs are not isomorphic, when querying the i_k bit of y, we will always get a bit of the same value with the k^{th} bit of S_r.

If the graphs H_0 and H_1 are isomorphic then the probability that by querying the $i_1, \ldots i_n$ bits of y we will get S_r is too small: Suppose that we are given a string y. The probability that the n examined bits of y match the random string S_r is less than $\frac{1}{2^n}$. Therefore :

- if $(H_0, H_1) \in \overline{GI} \Rightarrow \exists y : Prob(M(x, y)$ accepts$) = 1$
- if $(H_0, H_1) \notin \overline{GI} \Rightarrow \forall y : Prob(M(x, y)$ accepts$) < \frac{1}{2^n}$

[8] suppose that a coin flip returns 0 or 1

In the first step the machine makes n random guesses. At step 2, for each graph we do a random permutation of its labels. Since each graph has n vertices there are $n!$ possible permutations. Clearly these can be encoded in $\log n! = O(n \log n)$ bits, thus the total number of coin flips is $O(n^2 \log n)$. Finally at step 4 the machine asks n queries. Therefore $\overline{GI} \in PCP(n^2 \log n, n)$.

48.3 Example 2

We will give a definition for Max-Set that is a generalization of the one given in the textbook (MAX-k-FUNCTIONAL-SAT pp. 436–437):

Definition 48.4 (Max-Set) Given $F = f_1, f_2, \ldots, f_m$ where all f_i are formulas of any form (not necessarily in CNF), we want to find an assignment to the variables that satisfy the maximum number of them.

The objective function is the fraction of functions that evaluate to 1. We will show that no .99-approximation poly-time algorithm exists for Max-Set.

Theorem 48.5 *If $\exists A \in P$ for Max-Set such that:*

$$\forall \textit{set of formulas } F\text{: } .99OPT(F) \leq A(F) \leq OPT(F)$$

where $OPT(F)$ is the maximum number of formulas in F that can be satisfied by any assignment, then $P = NP$

Proof:
We already saw that $SAT \in PCP(\log n, 1)$. Imagine a coin-flipping machine M that flips $O(\log n)$ coins and then asks a constant number of questions on its input y. Figure 46 shows the tree that is defined by

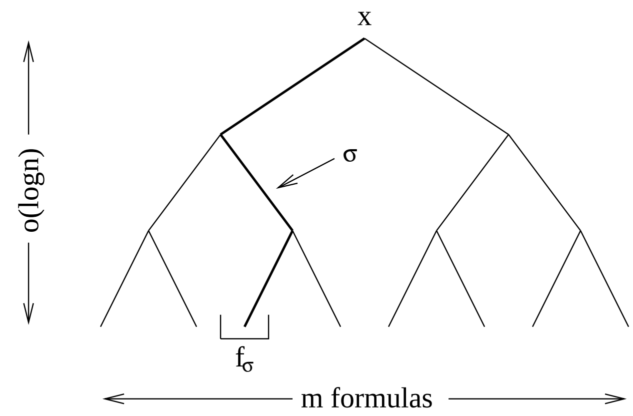


Figure 46: Different paths for $\log n$ coin-flips

all possible paths that we can get by flipping $o(\log n)$ coins. At the leaf-node of a path σ we get a set of queries that are asked by M on y. Since the number of queries is constant we can write a truth table, figure exactly which combinations are accepted by M and feed them to a formula f_σ. Since the height of that tree is $o(\log n)$ the number of all formulas f_σ that we can get, m, is polynomial to n.

The key-idea is that since this tree is of polynomial-size we can write an algorithm that *simulates* all these random choices of M and builds that tree. The key-point is that :

$$\begin{array}{lll} \text{if } x \in SAT & \Rightarrow & \exists \text{ a way (a “nice” y) to answer all queries} \\ & \Rightarrow & \cup f_\sigma \in SAT \\ \text{if } x \notin SAT & \Rightarrow & \text{You can't possibly satisfy more than } \frac{1}{4}m \text{ of } f_\sigma\text{'s.} \end{array}$$

We will use that gap between these two cases to built an Algorithm for SAT that runs in poly-time:

Algorithm for SAT

1. Input(x)
2. Find formulas f_σ
3. Run algorithm A on $(f_1, f_2, \ldots, f_m)$, get approximation α
4. If $\alpha \geq .99m$ output(YES)
5. otherwise $(\alpha < \frac{m}{4})$ output(NO)

$\square$

CMSC 858K Advanced Algorithms	Lecture 23
Lecturer: William L. Gasarch	Apr 29, 1997

Notes by Dmitry Zotkin.

49 Lower Bounds on Approximations (contd)

Topics:

- Application of PCP theorem resulting in proof of inapproximability of $CLIQUE$.
- Sketch of a proof that $SAT \in PCP(\log n, \log n)$.

Definition 49.1 ($PCP(r(n), q(n))$ language) The language $L \in PCP(r(n), q(n))$ if $\exists$ polynomial time, coin-flipping Turing machine M capable of doing random access input queries on two inputs (x, y) and $\exists k \in N$:

- $\forall x, y : |x| = n, |y| = 2^{n^k}$ during the run of M(x,y)
  - at most $O(r(n))$ coins will be flipped,
  - at most $O(q(n))$ bit-queries to y will be made;
- if $x \in L$, then $\exists y : [Prob(M(x, y)\ accepts) = 1]$
- if $x \notin L$, then $\forall y : [Prob(M(x, y)\ accepts) < \frac{1}{4}]$

(The constant $\frac{1}{4}$ here is selected arbitrarily and can be made as small as desired).

In other words, if $x \in L$, then y is a proof (an evidence of membership). If $x \notin L$, then *any* y will be rejected with some probability, which can be made as close to 1 as desired after several runs of M. Another important thing is that y is allowed to have exponential length, but the machine examines only $O(q(n))$ bits of y to make a decision.

Theorem 49.2 $NP \subseteq PCP(\log n, 1)$.

This is a quite remarkable theorem. Usually NP is defined as a class of languages for which verification of membership can be done in polynomial time using a polynomial length evidence y (for example, a satisfying assignment for SAT problem). The theorem says that to verify membership for any language $L \in NP$ one can use a really long evidence, flip $O(\log n)$ coins, access only $O(1)$ bits of y and be pretty sure that the membership decision is correct. The sketch of a proof of a weaker statement that $NP \subseteq PCP(\log n, \log n)$ will be given later.

49.1 Inapproximability of $CLIQUE$

Now the theorem above will be applied to prove that $CLIQUE$ cannot be approximated in polynomial time within *any* constant factor. Remember that proofs of inapproximability are based on *gap-preserving reductions* (the decision problem Z in NP is reduced to optimization problem Z^* in such a way that solutions to Z^* are substantially different (and therefore it is possible to distinguish between them by applying polynomial-time approximation algorithm A to Z^*) depending on answer for Z; therefore, if such algorithm A exists, Z can be decided upon in polynomial time.

Theorem 49.3 *Assuming $P \neq NP$, it is not possible to approximate $CLIQUE$ within $\alpha = \frac{1}{4}$.*

Proof:

Assume that there exists a polynomial-time algorithm that approximates $CLIQUE$ within $\frac{1}{4}$. Then the following algorithm can be used to solve SAT in polynomial time.

First, input ϕ - the instance of SAT problem. Then, use the PCP-verifier for SAT to construct graph G (see below for the actual procedure) and run the approximation algorithm for $CLIQUE$ on G. If the resulting approximation of $CLIQUE$ is more than $\frac{1}{4}n^k$ in size, then the answer to SAT is *yes*; otherwise, the answer is *no*.

To reduce SAT, consider the algorithm of the probabilistic Turing machine $M(x, y)$ for SAT. (Since $NP \subseteq PCP(\log n, 1)$ and $SAT \in NP$, $M(x, y)$ exists). Run $M(\phi, \theta)$ for all possible sequences of coin flips. Each sequence of coin flips defines $O(1)$ positions in θ to which queries will be made; look on all possible answers to these queries. We can think of this procedure as a construction of a tree; each leaf node can be described by a sequence of flips leading to it's parent (e.g. 100100011) and answers to queries (e.g. TFTTTFFFFTFT). Note that each node in the tree has $O(1)$ sons and the height of the tree is equal to the number of coin flips (ie. $O(\log n)$); therefore, there is only a polynomial number of leaves in the tree.

To construct graph G, make every leaf a node. Define two nodes to be *consistent* if the answers to bits that are common to both nodes are the same. (E.g. if in node A_1 the sequence of coin flips define that bits 17, 42 and 7 will be queried and answers to these queries are T, T and F respectively and in node A_2 queries are made to bits 7, 14 and 40 with answers T, T, T, then these node are inconsistent because of different answers for bit 7. If two nodes don't have any common queries, they are consistent). Connect two *accepting* nodes with an edge if they have *different* coin flip sequences and are consistent. Now, observe that if ϕ is satisfiable, then, for each sequence of coin flips, there is a way of answering queries in a consistent manner, which means that the graph G has a large clique of the size $2^{O(\log n)} = n^k$; if ϕ is not satisfiable, then at most $\frac{1}{4}$ of nodes can form a clique. Therefore, existance of polynomial-time approximation of $CLIQUE$ with $\alpha = \frac{1}{4}$ would allow us to solve SAT in polynomial time. □

Since the constant $\frac{1}{4}$ in definition of PCP can be reduced by running the machine several times, it can be shown that approximation of CLIQUE within *any* constant factor gives a polynomial-time algorithm for SAT.

49.2 PCP-verifier for SAT

Now, the proof of the fact that $SAT \in PCP(\log n, \log n)$ will be sketched. The first part of the proof (reduction of SAT to problem to be used as an input to PCP-verifier) will be completely omitted, and only the second part (the verifier itself) will be shown.

Given a formula ϕ - an instance of SAT - the verifier should produce a decision whether ϕ belongs to SAT or not. There exist a long and complex reduction of SAT to the following problem:

- given polynomial $p(x_1, ..., x_l, A(x_1, ..., x_l))$,
- given prime q,
- whether $\exists A(x_1, ..., x_l) : Z_q^l \Rightarrow Z_q$ such that

$$\sum_{x_1=0}^{1} \sum_{x_2=0}^{1} ... \sum_{x_l=0}^{1} p(x_1, ..., x_l, A(x_1, ..., x_l)) \equiv 0 \bmod q$$

(and A satisfy other conditions, which won't be explored here).

The function A is used to map variables to values. If we have n variables we can encode them using $\log n$ bits, hence $l = \log n$ and these bits are used to specify a variable.

In the reduction above, $l \approx \log n$, Z_q denotes the numeric field modulo q, p depends on ϕ and therefore is known, and if A exists, then ϕ is satisfiable; otherwise, it is not.

Now, given ϕ reduced into the form above, and given some string θ which is claimed to be the proof that ϕ is satisfiable, PCP-verifier need to make $O(\log n)$ coin flips, access only $O(\log n)$ bits of θ and accept or reject this proof so that probability of false acceptance is less than $\frac{1}{4}$. The evidence used by the verifier is θ which records all possible values of $A(a_1, ..., a_l)$ and all possible sums of the form

$$\sum_{x_1=0}^{1} \dots \sum_{x_k=0}^{1} p(x_1, \dots, x_k, a_{k+1}, \dots, a_l, A(x_1, \dots, x_k, a_{k+1}, \dots, a_l))$$

for all $k \in [0, l]$ and all possible $a_i \in [0, q-1]$. The following procedure will do the job; basically, what it does is repeatedly checking the consistency of data in θ. If A exists, there will be θ where data are consistent (and it will pass all checks independent of the sequence of coin flips); otherwise, every possible θ will fail with some probability.

Here is what should be done. Define $S_{i,\xi}$ to be

$$S_{i,\xi} = \sum_{x_1=0}^{1} \dots \sum_{x_{i-1}=0}^{1} p(x_1, \dots, x_{i-1}, \xi, a_{i+1}, \dots, a_l, A(x_1, \dots, x_{i-1}, \xi, a_{i+1}, \dots a_l))$$

First, ask θ about the values of $S_{l,0}$ and $S_{l,1}$ and check whether they sum to 0 (mod q). Then, flip coins and randomly choose $a_l \in [0, q-1]$. Ask for the values of S_{l,a_l}, $S_{l-1,0}$ and $S_{l-1,1}$ (keeping chosen a_l in it's place in last two queries) and check whether $S_{l-1,0} + S_{l-1,1} = S_{l,a_l}$. Select randomly a_{l-1} and continue this process.

Eventually, all queried values will sum up to the $S' = p(a_1, \dots, a_l, A(a_1, \dots, a_l))$. Now a query should be made to θ to retrieve the value of $A(a_1, \dots, a_l)$. Since p is known, the verifier can compute the "true" value of the S' using $A(a_1, \dots, a_l)$ and check whether they are in agreement. This is a biggest consistency check done against θ, and while a correct θ (ie. θ with correct A encoded in the case that ϕ is satisfiable) will pass it, it is very unlikely that any θ will be accepted when ϕ is not satisfiable since verifier knows polynomial p which θ must fit into, and therefore θ can't turn verifier into thinking that such A exist.

Notes by Laura Bright.

50 Linear Programming: The Use of Duality

The Primal-Dual Method is a useful tool for solving combinatorial optimization problems. In this lecture we first study Duality and see how it can be used to design an algorithm to solve the Assignment problem.
Dual Problem Motivation

The primal problem is defined as follows:

Maximize

$$\sum_{j=1}^{n} c_j x_j$$

subject to:

$$\forall i \in (1,\ldots,m) \sum_{j=1}^{n} a_{ij} x_j \leq b_i$$

$$x_j \geq 0$$

To formulate the dual, for each constraint introduce a new variable y_i, multiply constraint i by that variable, and add all the constraints.

$$\sum_{i=1}^{m} (\sum_{j=1}^{n} a_{ij} x_j) y_i \leq \sum_{i=1}^{m} b_i y_i$$

Consider the coefficient of each x_j.

$$\sum_{i=1}^{m} a_{ij} y_i$$

If this coefficient is at least c_j

$$c_j \leq \sum_{i=1}^{m} a_{ij} y_i$$

then we can use $\sum_{i=1}^{m} b_i y_i$ as an upper bound on the maximum value of the primal LP.

$$\sum_{j=1}^{n} c_j x_j \leq \sum_{j=1}^{n} (\sum_{i=1}^{m} a_{ij} y_i) x_j \leq \sum_{i=1}^{m} b_i y_i$$

We want to derive the best upper bound for the dual so we wish to minimize this quantity.
Dual:

Minimize

$$\sum_{i=1}^{m} b_i y_i$$

subject to:

$$\forall j \in (1,\ldots,n) \sum_{i=1}^{m} a_{ij} y_i \geq c_j$$

$$y_i \geq 0$$

The constraint that each y_i be positive is to preserve the inequalities of the primal problem. The constraints of the primal problem could also be equality constraints, and in this case we can drop the requirement that the y_i values be positive. It is worth noting that if you take the dual of the dual problem, you will get back the original primal problem.

We saw in the previous lecture that using the simplex method, you can find the optimal solution to a primal problem if one exists. While the simplex certifies its own optimality, the dual solution can be used to certify the optimality of any given primal solution. The *Strong Duality Theorem* proves this.

Theorem 50.1 (Strong Duality Theorem)
If the primal LP has an optimal solution x^*, then the dual has an optimal solution y^* such that:

$$\sum_{j=1}^{n} c_j x_j^* = \sum_{i=1}^{m} b_i y_i^*.$$

Proof:

To prove the theorem, we only need to find a (feasible) solution y^* that satisfies the constraints of the Dual LP, and satisfies the above equation with equality. We solve the primal program by the simplex method, and introduce m slack variables in the process.

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j \quad (i = 1,\ldots,m)$$

Assume that when the simplex algorithm terminates, the equation defining z reads as:

$$z = z^* + \sum_{k=1}^{n+m} \overline{c}_k x_k.$$

Since we have reached optimality, we know that each $\overline{c}_k$ is a nonpositive number (in fact, it is 0 for each basic variable). In addition z^* is the value of the objective function at optimality, hence $z^* = \sum_{j=1}^{n} c_j x_j^*$. To produce y^* we pull a rabbit out of a hat ! Define $y_i^* = -\overline{c}_{n+i} \quad (i = 1,\ldots,m)$.

To show that y^* is an optimal dual feasible solution, we first show that it is feasible for the Dual LP, and then establish the strong duality condition.

From the equation for z we have:

$$\sum_{j=1}^{n} c_j x_j = z^* + \sum_{k=1}^{n} \overline{c}_k x_k - \sum_{i=1}^{m} y_i^* (b_i - \sum_{j=1}^{n} a_{ij} x_j).$$

Rewriting it, we get

$$\sum_{j=1}^{n} c_j x_j = (z^* - \sum_{i=1}^{m} b_i y_i^*) + \sum_{j=1}^{n} (\overline{c}_j + \sum_{i=1}^{m} a_{ij} y_i^*) x_j.$$

Since this holds for all values of x_i, we obtain:

$$z^* = \sum_{i=1}^{m} b_i y_i^*$$

(this *establishes the equality*) and

$$c_j = \overline{c}_j + \sum_{i=1}^{m} a_{ij} y_i^* \quad (j = 1,\ldots,n).$$

Since $\bar{c}_k \leq 0$, we have

$$y_i^* \geq 0 \quad (i = 1, \ldots, m).$$

$$\sum_{i=1}^{m} a_{ij} y_i^* \geq c_j \quad (j = 1, \ldots, n)$$

This *establishes the feasibility* of y^*.

□

We have shown that the dual solution can be used to verify the optimality of a primal solution. Now we will show how this is done using an example. F irst we introduce *Complementary Slackness Conditions.*

Complementary Slackness Conditions:

Theorem 50.2 *Necessary and Sufficient conditions for x^* and y^* to be optimal solutions to the primal and dual are as follows.*

$$\sum_{i=1}^{m} a_{ij} y_i^* = c_j \textit{ or } x_j^* = 0 \textit{ (or both) for } j = 1, \ldots, n$$

$$\sum_{j=1}^{n} a_{ij} x_j^* = b_i \textit{ or } y_i^* = 0 \textit{ (or both) for } i = 1, \ldots, m$$

In other words, if a variable is non-zero then the corresponding equation in the dual is met with equality, and vice versa.
Proof:
We know that

$$c_j x_j^* \leq (\sum_{i=1}^{m} a_{ij} y_i^*) x_j^* \qquad (j = 1, \ldots, n)$$

$$(\sum_{j=1}^{n} a_{ij} x_j^*) y_i^* \leq b_i y_i^* \qquad (i = 1, \ldots, m)$$

We know that at optimality, the equations are met with equality. Thus for any value of j, either $x_j^* = 0$ or $\sum_{i=1}^{m} a_{ij} y_i^* = c_j$. Similarly, for any value of i, either $y_i^* = 0$ or $\sum_{j=1}^{n} a_{ij} x_j^* = b_i$. □

The following example illustrates how complementary slackness conditions can be used to certify the optimality of a given solution.
Consider the primal problem:
Maximize

$$18x_1 - 7x_2 + 12x_3 + 5x_4 + 8x_6$$

subject to:

$$2x_1 - 6x_2 + 2x_3 + 7x_4 + 3x_5 + 8x_6 \leq 1$$

$$-3x_1 - x_2 + 4x_3 - 3x_4 + x_5 + 2x_6 \leq -2$$

$$8x_1 - 3x_2 + 5x_3 - 2x_4 + 2x_6 \leq 4$$

$$4x_1 + 8x_3 + 7x_4 - x_5 + 3x_6 \leq 1$$

$$5x_1 + 2x_2 - 3x_3 + 6x_4 - 2x_5 - x_6 \leq 5$$

$$x_1, x_2, x_3, x_4, x_5, x_6 \geq 0$$

We claim that the solution

$$x_1^* = 2, x_2^* = 4, x_3^* = 0, x_4^* = 0, x_5^* = 7, x_6^* = 0$$

is an optimal solution to the problem.

According to the first complementary slackness condition, for every strictly positive x value, the corresponding dual constraint should be met with equality. Since the values x_1, x_2, and x_5 are positive, the first, second, and fifth dual constraints must be met with equality. Similarly, from the second complementary slackness condition, we know that for every strictly positive y value, the corresponding primal constraint should be met with equality. By substituting the given x values in the primal equations, we see that the second and fifth equations are not met with equality. Therefore y_2 and y_5 must be equal to 0. The complementary slackness conditions therefore give us the following:

$$2y_1^* - 3y_2^* + 8y_3^* + 4y_4^* + 5y_5^* = 18$$

$$-6y_1^* - y_2^* - 3y_3^* + 2y_5^* = -7$$

$$3y_1^* + y_2^* - y_4^* - 2y_5^* = 0$$

$$y_2^* = 0$$

$$y_5^* = 0$$

Solving this system of equations gives us $(\frac{1}{3}, 0, \frac{5}{3}, 1, 0)$. By formulating the dual and substituting these y values, it is easy to verify that this solution satisfies the constraints of the dual, and therefore our primal solution must be optimal.

Note that this method of verifying solutions only works if the system of equations has a unique solution. If it does not, it means that the dual is unbounded, and the primal problem is therefore infeasible. Similarly, if the primal problem is unbounded, the dual is infeasible.

Assignment Problem

We can now revisit the weighted matching problem we studied earlier in the semester to see how the solution to this problem can be derived using the primal-dual method. The Assignment Problem is, given a complete bipartite graph with a weight w_e assigned to each edge, to find a maximum weighted perfect matching. Stated as a linear program, the problem is as follows:

Maximize

$$\sum w_e x_e \quad 1 \geq x_e \geq 0$$

subject to:

$$\forall u \in U \sum_{e=(u,*)} x_e = 1$$

$$\forall v \in V \sum_{e=(*,v)} x_e = 1$$

The dual to this problem is:

Minimize

$$\sum_{u \in U} Y_u + \sum_{v \in V} Y_v$$

subject to:

$$Y_u + Y_v \geq w(u,v) \quad \forall edges(u,v)$$

Since all the constraints of the primal are met with equality, the second complementary slackness condition is automatically satisfied. By the definition of complementary slackness, if we have a feasible primal solution that satisfies the first complementary slackness condition, we are done.

In any feasible solution to the primal, every vertex has some edge e such that $x_e > 0$. By the first complementary slackness condition, if $x_j^* > 0$ then the corresponding dual constraint must be met with equality. When we have a feasible primal solution such that $Y_u + Y_v = w(u, v)$ for all edges $e = (u, v)$ such that $x_e > 0$, we have an optimal solution to both problems. This suggests that only the edges in the "equality subgraph" (edges for which the dual constraints are met with equality) should be considered when we want to add edges to the matching. Also notice that the number of unmatched vertices is our "measure of infeasibility" – as this decreases, we approach feasibility.

The labeling function l on the vertices that we used earlier is nothing but the dual variables.

$$l(u_i) + l(v_j) \geq w(u_i, v_j)$$

Start with a feasible labeling and compute the equality subgraph G_l which includes all the vertices of the original graph G but only edges (x_i, y_j) which have weights such that $w(x_i, y_j) = l(x_i) + l(y_j)$.

If G_l is a perfect matching, we have an optimal solution. Otherwise, we revise the labels to improve the quality of the matching.

The algorithm finds a maximum matching in the equality subgraph. We increase the labels of some vertices and decrease the labels of others. The total sum of the labels is dropping, so we are decreasing our dual solution. If we increase the size of the matching, we are one step closer to the optimal solution. When an optimal labeling is found, we have an optimal solution to the dual, and we therefore have a maximum weighted matching in G.

CMSC 858K Advanced Algorithms	Lecture 25
Lecturer: Samir Khuller	May 6, 1997

Notes by Josh Burdick.

51 Using duality to analyze (and design) approximation algorithms

This material is covered in Chapter 4 (which was written by Goemans and Williamson) of the *Approximation Algorithms* book edited by Hochbaum. (The original paper appeared in SICOMP 1995.)

51.1 Using duality to bound OPT-IP

Duality was first studied as a way of proving optimality of solutions to an LP instance. It also played a role in the minimum-weight bipartite perfect matching algorithm, where the labeling function is essentially the dual problem, although the concept of a "dual problem" was not formulated at that time.

Assume there is some integer program (IP) that you wish to solve; assume without loss of generality that you wish to minimize the value of the objective function. Let OPT-IP denote the value of the objective function at the optimal solution.

Note that the value of the LP relaxation of this problem can be no more than OPT-IP, since the optimal IP solution is also a feasible LP solution – in general, OPT-LP is less than OPT-IP. By the Weak Duality theorem of linear programming, any feasible dual solution is a lower bound on OPT-LP (and thus OPT-IP).

To summarize, we have that

any solution to the *dual* LP	$\leq$	OPT-DLP = OPT-LP	$\leq$	OPT-IP

(This is assuming that the primal IP is a minimization problem; if it were a maximization problem, all the inequalities would be reversed.)

Using this to bound approximation factors Our main goal will be to produce a feasible solution whose cost is at most α times the value of the dual feasible solution that we can find, which is also a lower bound on the cost of the optimum solution.

The Steiner tree problem The Steiner tree problem is: Given an undirected graph with weights on the edges, and a subset T of the vertices, find a minimum-weight tree that connects all the vertices in T. (The tree needn't connect all the vertices in V, but it optionally can use vertices from $V - T$, if that will help to connect the vertices in T.)

The problem is NP-complete. There was a sequence of results, with improving approximation ratios:

- Kou, Markowsky and Berman discovered a 2-approximation (this was homework 3, problem 3)
- Takahashi and Matsuyama proved that a simple greedy algorithm also gives a 2 approximation.
- Zelikovsky obtained an $\frac{11}{6} \approx 1.833$ approximation ratio by considering three vertices at a time, and determining whether or not Steiner nodes would help to connect just those three. (Steiner nodes are vertices in $V - T$.)
- Berman and Ramaiyer improved this bound to $\frac{16}{9} \approx 1.777$ with a more complicated algorithm which considers more vertices at a time.
- Karpiniski and Zelikovsky later found a bound of 1.644.
- Agrawal, Klein, and Ravi studied a generalization of the Steiner tree problem, the pairwise connection problem. Here, given some sets of vertices $\{s_1, s_2, ...s_n\}$ and $\{t_1, t_2, ...t_n\}$, the goal is to pick edges so that s_i is connected to t_i for all i such that $1 \leq i \leq n$.

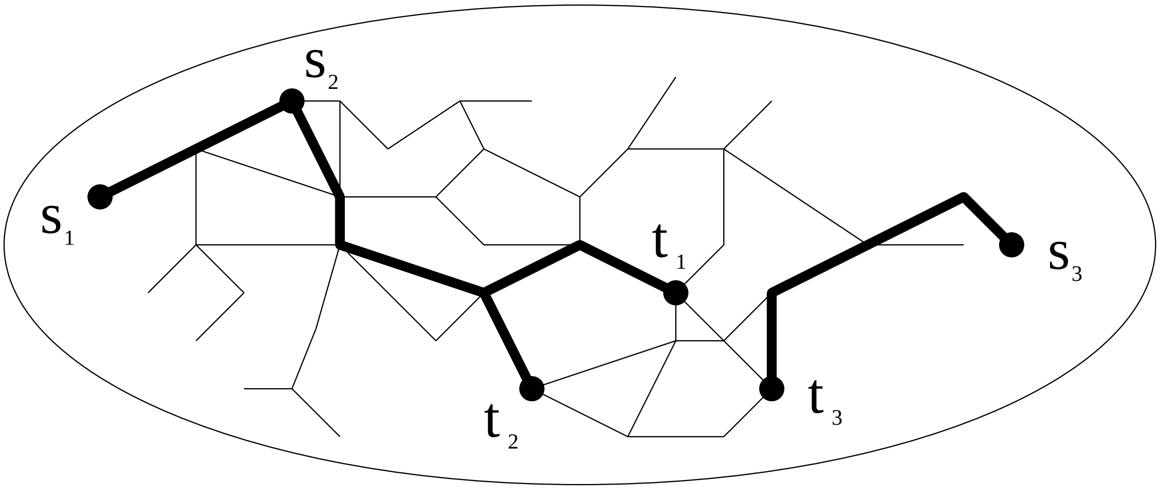


Figure 47: The pairwise connection problem.

Goemans and Williamson found that the generalized Steiner tree problem, and a variety of other problems (mostly network-design problems), could all be stated and approximated in a unified framework.

51.2 The "proper function" framework

Now we return to Goemans and Williamson's general approximation framework. We are given a graph $G = (V, E)$, with weights on the edges. We will have indicator variable x_e for each edge, in the usual way. Also, for any subset of vertices S, let $\delta(S)$ be the set of all edges from S to $V - S$. Let f be a function from subsets (also, in this context, called "cuts") of the vertices V to $\{0, 1\}$.

A problem is expressible in this framework if you can write it as an integer programming problem:

$$\text{minimize} \sum w_e \cdot x_e$$

subject to the constraints

$$\sum_{e \in \delta(S)} x_e \geq f(S)$$

This looks a lot like the usual integer programming formulation. The main difference is the use of the function f. Intuitively, you want to pick f to be 1 whenever you want to ensure that there's at least one edge crossing some cut S.

In the dual problem, we have a variable for every cut S of V. The dual problem is

$$\text{maximize} \sum y_S \cdot f(S)$$

subject to the constraints

$$\sum_{S : e \in \delta(S)} y_S \leq w_e$$

There are $2^{|V|}$ different cuts, and therefore $2^{|V|}$ different dual variables y_S. Fortunately, most of them are zero, and we need only keep track of the nonzero dual variables.

The function f is called a "$\{0, 1\}$ proper function" (to be defined formally later).

51.3 Several problems, phrased in terms of proper functions

Many problems related to network design can be stated in these terms.

Generalized Steiner tree We are given an undirected, weighted graph, and two sets of vertices, $\{s_1, s_2, ...s_n\}$ and $\{t_1, t_2, ...t_n\}$. The goal is to find a minimum-weight subset of edges such that s_i is connected to t_i, for $1 \leq i \leq n$.

To solve this, let $f(S) = 1$ if for some i, $s_i \in S$ and $t_i \notin S$ (or vice versa); otherwise let $f(S) = 0$. This forces s_i to be connected to t_i.

Steiner tree Let T be the set of terminal nodes (that is, nodes which are required to be connected.) Let S be a cut of V. Define

$$
\begin{aligned}
f(S) &= 1 \text{ if } S \neq \emptyset,\ S \cap T \neq T, \text{and } S \subset V \\
&= 0 \text{ otherwise}
\end{aligned}
$$

If a cut S contains at least one of the terminal nodes, but not all of them, then there should be at least one edge across the cut, to connect with the terminal nodes not in S.

Minimum-weight perfect matching Let S be a cut of V. Let $f(S) = 1$ iff $|S|$ is odd; otherwise let $f(S) = 0$. This forces each tree in the forest to have even size. Once we have these even components, we can convert it into a matching of the same size.

51.4 The greedy algorithm

Here is Goemans and Williamson's greedy algorithm to 2-approximate problems stated in the proper function framework:

1. Initialize the set of connected components $C = V$.

2. Increase all of the y_Ss that haven't hit a constraint, all at once. If a constraint hits equality, pick the edge corresponding to that constraint. A new connected component S' has been created, so add S' to C, and create a new variable $y_{S'}$ for it. $y_{S'}$ starts at 0.

3. Repeat step 2 until enough edges are picked.

4. Afterwards, throw away any unneeded edges. To do this, let A and B be the connected components at each end of some edge. If $f(A) = 0$ and $f(B) = 0$, you can discard that edge.

At the outset, there are $|V|$ seperate connected components, each consisting of one vertex. Each connected component has a variable y_S associated with it, which grows as the algorithm progresses. If a constraint is hit, that y_S stops growing. Later, of course, the vertices in y_S may be added to some new connected component S'.

The cost of the solution found is no more than two times the cost of some dual solution; by the arguments at the start of the lecture, that means that the cost of the solution found is within a factor of two of optimal. (The proof will be given next lecture.)

An example run of the algorithm Here is a small example of how this algorithm runs. The problem being solved is minimum perfect matching. The vertices are on a plane, and the weights between them are Euclidean distances.

The primal problem is to meet the constraints, and find a minimum perfect matching, using a minimum-weight set of edges. The dual variables have been described by Jünger and Pulleyblank as "moats", or regions surrounding the connected components. They note that in the case of the minimum-weight perfect matching problem in the Euclidean plane, the dual problem

> "...attempts to find a packing of non-overlapping moats that maximizes the sum of the width of the moats around odd-sized sets of vertices. The algorithm... can thus be interpreted as growing odd moats at the same speed until two moats collide, therefore adding the corresponding edge, and repeating the process until all components have even size." (Hochbaum, p. 173.)

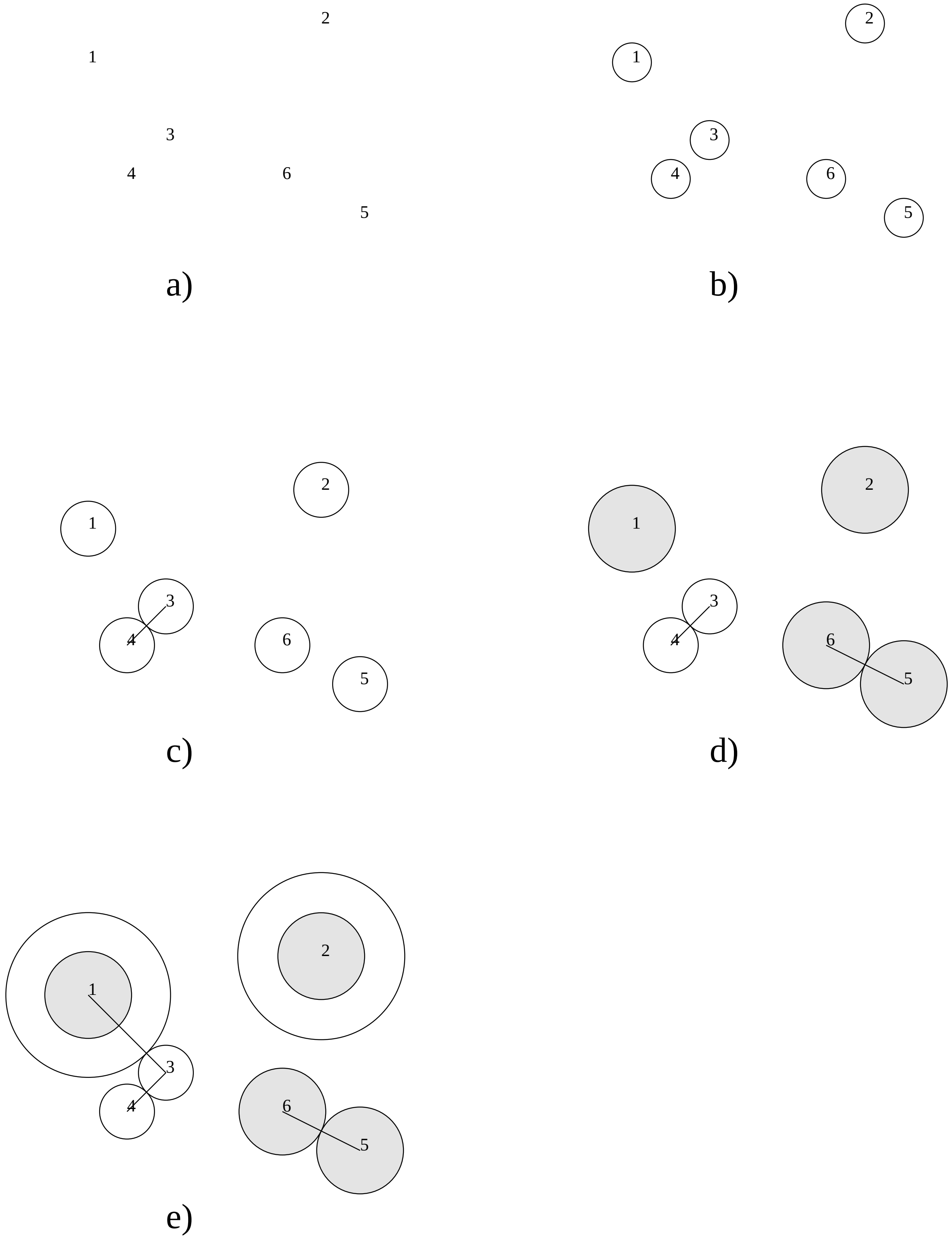


Figure 48: The greedy algorithm for minimum-weight perfect matching in the Euclidean plane (example from class.)

The diagram on p. 174 of Hochbaum illustrates this; the example from class was similar.

Fig.$\tilde{3}$ a) shows six points in the Euclidean plane. The values of the dual variables $y_{\{1\}}$, $y_{\{2\}}$, $y_{\{3\}}$, $y_{\{4\}}$, $y_{\{5\}}$, and $y_{\{6\}}$ are all initialized at zero.

In b), the dual variables have increased some, but no constraint has yet been hit.

In c), dual variables $y_{\{3\}}$ and $y_{\{4\}}$ have hit a constraint. So, they are connected with an edge. A new connected component has been formed. We needn't create a new dual variable, $y_{\{3,4\}}$, because that variable will always be zero.

In d), the dual variables $y_{\{1\}}$, $y_{\{2\}}$, $y_{\{5\}}$, and $y_{\{6\}}$ have all increased, and an edge is added from 5 to 6. Vertices 5 and 6 are now "inactive." At this point, only vertices 1 and 2 are growing.

In e), vertex 1 gets connected to vertices 3 and 4. The dual variable $y_{\{1,3,4\}}$ is formed, and will keep increasing.

When no dual variables are active anymore (that is, when every connected component has an even number of vertices), this phase will stop. The algorithm will now delete as many edges as it can.

CMSC 858K Advanced Algorithms	Lecture 26
Lecturer: Samir Khuller	May 13, 1997

Notes by Thomas Vossen.

52 A General Approximation Technique for Constrained Forest Problems

Today's lecture notes continue with the general framework for approximation algorithms that was started last time. We discuss the algorithm of Goemans and Williamson in more detail, and proof its correctness and approximation bound. The material covered here can also be found in Goemans and Williamson's article (from handout).

52.1 Constrained Forest Problems

Following the terminology of Goemans and Williamson, we call the problems to which the framework can be applied *constraint forest problems.* Given a graph $G = (V, E)$, a function $f : 2^V \rightarrow \{0, 1\}$ and a non-negative cost function $c : E \rightarrow \mathcal{Q}_+$, the associated constrained forest problem is then defined by the following integer program (IP):

$$
\begin{array}{lll}
\text{Min} & \sum_{e\in E} c_e x_e & \\
\text{subject to:} & & \\
 & \sum_{e\in\delta(S)} x_e \geq f(S) & S \subset V, S \neq \emptyset \\
 & x_e \in \{0,1\} & e \in E
\end{array}
$$

where $\delta(S)$ denotes the set of edges having exactly one endpoint in S. Any solution to integer program corresponds to a set of edges $\{e \in E \mid x_e = 1\}$. Observe that this set of edges can always be turned into a forest by arbitrarily deleting an edge from a cycle (By definition of $\delta(S)$, this does not affect feasibility).

We only consider *proper* functions f here, that is, functions $f : 2^V \rightarrow \{0, 1\}$ for which the following properties hold:

1. **Symmetry** $f(S) = f(V - S)$ for all $S \subseteq V$;
2. **Disjointness** If A and B are disjoint, then $f(A) = f(B) = 0$ implies $f(A \cup B) = 0$.

Furthermore, it is assumed that $f(V) = 0$. We have already seen several examples of graph problems that can be modeled as proper constraint forest problems. Some additional examples are the following :

- **Minimum Spanning Tree problem** Define $f(S) = 1$ for all subsets S.
- **Shortest $s - t$ Path problem** Define $f(S) = 1$ *iff* $|S \cap (s, t)| = 1$.

The greedy algorithm of Goemans and Williamson can be seen as a primal-dual algorithm, in which an approximate solution to the primal and to the LP-relaxation of the dual are constructed simultaneously, such that the primal solution value is at most a constant times the dual solution value. More precisely, we define (LP) to be the linear programming relaxation of (IP) by relaxing the integrality condition $x_e \in \{0, 1\}$ to $x_e \geq 0$, and we define the dual (D) of (LP) as :

$$
\begin{array}{lll}
\text{Max} & \sum_{S\subset V} f(S) y_s & \\
\text{subject to:} & & \\
 & \sum_{S:e\in\delta(S)} y_S \leq c_e & e \in E \\
 & y_s \geq 0 & S \subset V, S \neq \emptyset
\end{array}
$$

In the remainder we describe how the greedy algorithm constructs a feasible solution that is at most $(2 - \frac{2}{|A|})$ times the value of a (implicitly constructed) dual solution, with $A = \{v \in V \mid f(\{v\}) = 1\}$. Hence, it follows (see last time's notes) that the algorithm obtains an approximation factor of $2 - \frac{2}{|A|}$. On a sidenote, we mention that the algorithm runs in $O(n^2 \log n)$ time.

Input : An undirected graph $G = (V, E)$, edge costs $c_e \geq 0$, and a proper function f
Output: A forest F' and a value LB

1 $F \leftarrow \emptyset$
2 *Comment : Implicitly set* $y_S \leftarrow 0$ *for all* $S \subset V$
3 $LB \leftarrow 0$
4 $\mathcal{C} \leftarrow \{\{v\} : v \in V\}$
5 **for all** $v \in V$
6 $\quad d(v) \leftarrow 0$
7 **while** $\exists\, C \in \mathcal{C} : f(C) = 1$
8 $\quad$ Find edge $e = (i, j)$ with $i \in C_p \in \mathcal{C}$, $j \in C_q \in \mathcal{C}$, $C_p \neq C_q$ that minimizes $\epsilon = \frac{c_e - d(i) - d(j)}{f(C_p) + f(C_q)}$
9 $\quad F \leftarrow F \cup \{e\}$
10 $\quad$ **for all** $v \in C_r \in \mathcal{C}$ **do** $d(v) \leftarrow d(v) + \epsilon \cdot f(C_r)$
11 $\quad$ *Comment : Implicitly set* $y_C \leftarrow y_C + \epsilon \cdot f(C_r)$ *for all* $C \in \mathcal{C}$
12 $\quad LB \leftarrow LB + \epsilon \sum_{C \in \mathcal{C}} f(C)$
13 $\quad \mathcal{C} \leftarrow \mathcal{C} \cup \{C_p \cup C_q\} - \{C_p\} - \{C_q\}$
14 $F' \leftarrow \{e \in F :$ For some connected component N of $(V, F - \{e\})$, $f(N) = 1\}$

Figure 49: The algorithm of Goemans and Williamson.

52.2 Description of the Algorithm

The main algorithm is shown in Figure 49. The basic idea behind the algorithm is to build a set of edges F' that corresponds to a feasible solution of (IP). Generally speaking, the algorithm does this as follows:

- Maintain a forest of edges F (F is empty initially);
- Add an edge to F such that two disconnected components of F get merged. Repeat this step until $f(C) = 0$ for all connected components C in F;
- Remove 'unnecessary' edges from F to obtain the final solution F'. An edges e is not necessary if it can be removed from F such that $f(C) = 0$ for all components C of $F - \{e\}$. In words, this means that the removal of e does not affect the validity of the constraints in (IP).

Note that since $f(V) = 0$, the number of iterations is at most $|V| - 1$.

The central part of the algorithm is of course the way we choose an edge at each iteration. This choice is guided by the implicit dual solution we wish to construct: simply stated, we can say that the edge that is selected is such that at each iteration,

1. the dual solution y is maximized, while at the same time
2. y remains a feasible solution to (D).

To be more precise, we define an *active component* to be any component C of F for which $f(C) = 1$. Intuitively, only the dual variables y_C connected with an active component C can increase the dual solution value. Now, the algorithm proceeds greedily. All dual variables y_C associated with active components C are increased uniformly until a dual constraint is met with equality. When this happens, the dual variables cannot be increased further without violating the dual's feasibility, and we select the edge associated with the dual constraint that has become tight.

The next step is to show that the increase ϵ of the dual variables is equal to the formula given in step 8. For this, we first make two observations :

- $d(i) = \sum_{S: i \in S} y_S$ throughout the algorithm for each vertex i. This can easily be shown by induction.
- $\sum_{S: e \in \delta(S)} y_S = d(i) + d(j)$ for all edges $e = (i, j)$ such that i and j are in different components. This follows from the previous observation and from the definition of $\delta(S)$.

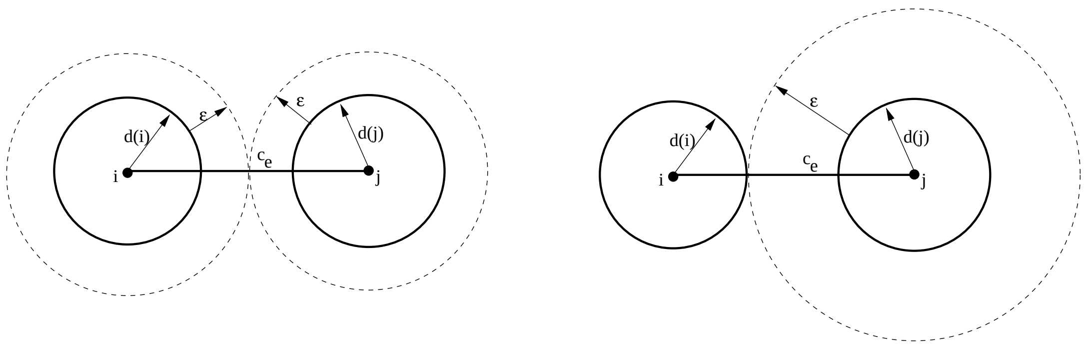


Figure 50: Determination of the dual increase ϵ. The thick radii corresponds to components in F, the dotted radii to a possible increase of the dual variables. In the figure on the left, both components are active. In the figure on the right, only one component is active.

Now, consider any edge $e = (i,j)$ that connects two distinct components C_p and C_q in F. If we increase the dual variables y_C by ϵ, $\sum_{e\in\delta(S)} y_S$ will increase by $\epsilon \cdot f(C_p) + \epsilon \cdot f(C_q)$. Hence, we can reformulate the dual constraint associated with e as

$$d(i) + d(j) + \epsilon \cdot f(C_p) + \epsilon \cdot f(C_q) \leq c_e$$

Then, it follows that the largest possible increase of ϵ that does not violate the feasibility of the dual is exactly the one given in step 8. The relation between $d(i), d(j), f(C_p, f(C_q)$ and ϵ is illustrated in Figure 50.

We also mention that the algorithm enforces the complementary slackness conditions of the primal (see notes lecture 24), that is, for all $e \in E$ we have $e \in F \Rightarrow c_e = \sum_{S:e\in\delta(S)} y_S$. This immediately follows from the preceding discussion, since we only add e to F if the corresponding dual constraint is met with equality.

52.3 Analysis

At this point, we still need to prove three things : the feasibility of the final primal and dual solutions, and the actual approximation bound. We first prove the feasibility of the final dual solution.

Theorem 52.1 *The dual solution y constructed by the algorithm is feasible for (D).*

Proof:
By induction. It is evident that y is feasible initially, since all the dual variables y_S are 0. By the preceding discussion, we have that the dual constraint associated with an edge $e = (i,j)$ holds throughout the algorithm if i and j are in different components. In case i and j are in the same component, the left hand side $\sum_{S:e\in\delta(S)} y_S$ of the dual constraint does not increase, and therefore the constraint continues to hold. $\square$

Now, we show that the algorithm produces a feasible solution to the integer program (IP). The idea behind the proof is to show that if $f(C) = 0$ for all connected components C in F', the solution corresponding to F' is feasible. For this to apply however, we first need to show that $f(C) = 0$ for all connected components C in F'. This is captured in the following lemma's.

Lemma 52.2 *If $f(S) = 0$ and $f(B) = 0$ for some $B \subseteq S$, then $f(S - B) = 0$.*

Proof:
By the symmetry property of f, $f(V - S) = f(S) = 0$. By the disjointness property, $f((V - S) \cup B) = 0$. Then, by the symmetry property again, we have $f(S - B) = f((V - S) \cup B) = 0$ $\square$

Lemma 52.3 *For each connected component N of F',$f(N) = 0$.*

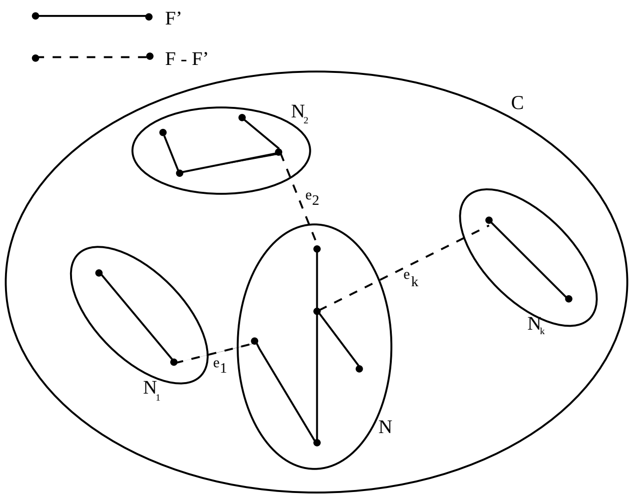


Figure 51: Illustration of Lemma 1.3.

Proof:
By construction of F', we have that $N \subseteq S$ for some component C of F. Let $e_1, \ldots, e_k$ be edges of F that have exactly one endpoint in N. These are the edges in C that were removed in the final step of the algorithm. Let N_i and $C - N_i$ be the two components that were generated by removing e_i from C, with $N \subseteq C - N_i$(see Figure 51). Since e_i was removed, we have that $f(N_i) = 0$. Also, the sets $N, N_1, \ldots, N_k$ form a partition of C. So, by the disjointness property of f we then have $f(C - N) = f(\bigcup_{i=1}^k N_i) = 0$. Because $f(C) = 0$, the previous lemma then implies that $f(N) = 0$. □

Theorem 52.4 *The solution associated with F' is a feasible solution to (IP).*

Proof:
By contradiction. Suppose that some constraint is violated, i.e., $\sum_{e \in \delta(S)} x_e = 0$ for some S such that $f(S) = 1$. This means that there is no edge in F' that has exactly one endpoint in S, so for all connected components N_i in F', either $N \cap S = \emptyset$ or $N \cap S = N$. So, $S = N_{i_1} \cup \ldots \cup N_{i_k}$ for some $i_1, \ldots, i_k$ However, by the previous lemma $f(N) = 0$, and then by the disjointness property $f(S) = 0$ too. But this contradicts our assumption, which completes the proof. □

Finally, we show that the algorithm obtains an approximation factor of $2 - \frac{2}{|A|}$. To do this, we only need to prove that the value of the primal solution associated with F' is at most $2 - \frac{2}{|A|}$ times the value of the implicitly constructed dual solution y. This is done in the following theorem.

Theorem 52.5 *The algorithm produces a set of edges F' and a dual solution y such that*

$$\sum_{e \in F'} c_e \leq (2 - \frac{2}{|A|}) \sum_{S \subset V} y_s$$

Proof:
We have that

$$\begin{aligned} \textstyle\sum_{e \in F'} c_e &= \textstyle\sum_{e \in F'} \sum_{S: e \in \delta(S)} y_s && \text{by the primal complementary slackness conditions} \\ &= \textstyle\sum_{S \subset V} y_s \cdot |F' \cap \delta(S)| && \text{by exchanging the summations} \end{aligned}$$

Now, we show by induction on the number of iterations that

$$\sum_{S \subset V} y_s \cdot |F' \cap \delta(S)| \leq (2 - \frac{2}{|A|}) \sum_{S \subset V} y_s$$

It is easy to see that this inequality holds initially, because then all y_s are 0. The next step is then to prove that the inequality remains valid from one iteration to another. This is done by showing that at each iteration, the increase in the left hand side of the equation is at most the increase in the right hand side. If we let $\mathbf{C}$ be the set of components at the beginning of the iteration and $\mathbf{C}^1 = \{C \in \mathbf{C}: f(C) = 1\}$ the set of active components, this amounts to proving

$$\sum_{C \in \mathbf{C}^1} \epsilon \cdot |F' \cap \delta(C)| \leq (2 - \frac{2}{|A|})\epsilon \cdot |\mathbf{C}^1|$$

To prove this, we first observe that $F' \cap \delta(C)$ consists of those edges in F' that have exactly one endpoint in C, so if we looked at a component C as being a vertex, $|F' \cap \delta(C)|$ would be its degree. The inequality then states that the average degree of the active components is at most $(2 - \frac{2}{|A|})$ (divide both sides by ϵ and $|\mathbf{C}^1|$).

To see this, we construct a graph H whose vertices are the components $C \in \mathbf{C}$. Its edges are $e \in F' \cap \delta(C)$ for all C, that is, those edges of F' that connect components in $\mathbf{C}$. Isolated vertices that correspond to inactive components are removed. Note that since F' is a forest, H is a forest too.

The key observation to make about the graph H is that it has no leaf that corresponds to an inactive component (so all inactive components have a degree of at least 2). To see this, we suppose otherwise, and let v be a leaf, C_v its associated inactive component, and e the edge incident on v. Now, by removing e, the component N in H that contains v is partioned into components $\{v\}$ and $N - \{v\}$. Let $C_{N-\{v\}}$ be the union of the components assiocated with $N - \{v\}$. We know that $f(C_v) = 0$ and since there are no edges leaving $N - \{v\}$, we also have $f(C_{N-\{v\}}) = 0$. But this means that e was removed at final step of the algorithm, so we have a contradiction.

We now finalize the proof. Let d_v be the degree of a vertex v in H, and let N_a, N_i be the sets of vertices in H corresponding to active resp. inactive components. Given the observations above, we then have

$$\sum_{v \in N_a} d_v = \sum_{v \in N_a \cup N_i} d_v - \sum_{v \in N_i} d_v \leq 2(|N_a| + |N_i| - 1) - 2|N_i| = 2|N_a| - 2$$

The inequality holds since in any forest, the number of edges is less than the number of vertices. Multiplying by ϵ and substituting N_a by $\mathbf{C}^1$, we then get

$$\sum_{C \in \mathbf{C}^1} \epsilon \cdot |F' \cap \delta(C)| \leq 2\epsilon(|\mathbf{C}^1| - 1) \leq (2 - \frac{2}{|A|})\epsilon \cdot |\mathbf{C}^1|$$

because the number of active components at any iteration will be at most $|A|$. $\square$

CMSC 651 Advanced Algorithms	Lecture 19
Lecturer: Samir Khuller	Th. April 1, 1993

Notes by Maria Chechik & Suleyman Sahinalp

53 On Line vs. Off Line: A Measure for Quality Evaluation

Suppose you have a dynamic problem. An algorithm for solving a dynamic problem which uses information about future is called an off-line algorithm. But computer scientists, are human beings without the capability of predicting the future. Hence their algorithms are usually on-line (i.e. they make their decisions without any information about the future).

53.1 K-Server Problem

In a fraternity party there are several robots which are responding to requests of the frat-members. Whenever someone requests a drink, one of the robots is sent to him. The problem here is to decide which robot is "best suited" for the request; moreover to find a measure for being "best suited".
If you knew the sequence of the location of requests, then you'd try to minimize the total distance your robots would travel until the end of the party. (It is an interesting exercise to solve this problem.) As you don't have this information you try to achieve some kind of a measure for assuring you (as the robot controller) didn't perform "that badly".

Conjecture 53.1 *If you have k robots, then there exists an on-line decision algorithm such that*

$$\textit{total cost of on-line alg} \leq k \cdot \textit{total cost of off-line alg} + C$$

where C is some constant.

Example:
In this example the requests come from the vertices of an equilateral triangle and there are only two servers. Suppose the worst thing happens and the requests always come from the vertex where there is no server. If the cost of moving one server from one vertex to a neighboring vertex is 1 then at each request the cost of the service will be 1. So in the worst case the on-line algorithm will have a cost of 1 per service whatever you do to minimize the cost.
To analyse the performance of an on-line algorithm, we will run the on-line algorithm on a sequence provided by an adversary, and then compare the cost of the on-line algorithm to the cost of the adversary (who may know the sequence before answering the queries).
For example, assume you have two robots, and there are three possible places where a service is requested: vertices A, B, and C. Let your robots be in the vertices A and B. The adversary will request service for vertex C. If you move the robot at vertex B (hence your servers will be at vertices A and C), the next request will come from B. If you now move your robot at vertex C (hence you have robots at A and B), the next request will come from C and so on.
In this scheme you would do k moves for a sequence of k requests but after finishing your job the adversary will say: "you fool, if you had moved the robot at A in the beginning, then you would have servers at vertices B and C and you wouldn't have to do anything else for the rest of the sequence".
We can actually show that for *any* on-line algorithm, there is an adversary that can force the on-line algorithm to spend at least twice as much as the offline algorithm. Consider any on-line algorithm A. The adversary generates a sequence of requests $r_1, r_2, \ldots, r_n$ such that each request is made to a vertex where there is no robot (thus the on-line algorithm pays a cost of n). It is easy to see that the offline algorithm can do a "lookahead" and move the server that is not requested in the immediate future. Thus for every two requests, it can make sure it does not need to pay for more than one move. This can actually be extended to a *lower bound* of k for the k-Server problem. In other words, any on-line algorithm can be forced to pay a cost that is k times the cost of the off-line algorithm (that knows the future).

53.2 K-servers on a straight line

Suppose you again have k servers on a line. The naive approach is to move the closest server to the request. But what does our adversary do? It will make a request near the initial position of one server (which moves), and then requests the original position of the server. So our server would be running back and forth to the same two points although the adversary will move another server in the first request, and for all future requests it would not have to make a move.

On-line strategy: We move the closest server to the request but we also move the server on the other side of the request, towards the location of the request by the same amount. If the request is to the left (right) of the leftmost (rightmost) server, then only one server moves.

We will analyse the algorithm as a game. There are two copies of the server's. One copy (the s_i servers) are the servers of the online algorithm, the other copy is the adversary's servers (a_i). In each move, the adversary generates a request, moves a server to service the request and then asks the on-line algorithm to service the request.

We prove that this algorithm does well by using a potential function, where

- $\Phi(t-1)$ is potential before the t^{th} request,
- $\Phi'(t)$ is potential after adversary's t^{th} service but before your t^{th} service,
- $\Phi(t)$ is the potential after your t^{th} service.

The potential function satisfies the following properties:

1. $\Phi'(t) - \Phi(t-1) < k\times$ (cost for adversary to satisfy the t^{th} request)
2. $\Phi'(t) - \Phi(t) >$ (cost for you to satisfy the t^{th} request)
3. $\Phi(t) > 0$

Now we define the potential function as follows:

$\Phi(t) := k \times \text{Weight of Minimum Matching} in G' + \sum_{i<j} \text{dist}(s_i, s_j)$.

The minimum matching is defined in the bipartite graph G' where

- $A \equiv$ set of the placements of your servers
- $S \equiv$ set of the placements of adversarys servers
- edge weights(a_i, s_j) $\equiv$ distance between those two servers

The second term is obtained by simply keeping the online algorithm's servers in sorted order from left to right.

Why does the potential function work: It is easy to see that those three properties are satisfied by our potential function:

1. the weight of the minimum matching will change at most the cost of the move by the adversary and distance between your servers do not change. Hence the first property follows.
2. if the server you're moving is the rightmost server and if the move is directed outwards then the second term of the potential function will be increased by $(k-1)d$ however the first term will be decreasing by kd, hence the inequality is valid. (Notice that the first term decreases because there is already one of the adversary's server's at the request point.) If the server you're moving is the leftmost server and the motion is directed outwards then the second term will increase by at most $(k-1)d$, though the first term will decrease by kd, hence the inequality is again valid. Otherwise, you will be moving two servers towards each other and by this move the total decrease in the second term will be $2d$ (the cost for service): two servers are $2d$ closer and the increase in distance by the first server is exactly same with the decrease in distance by the second server. In the worst case the size of the matching won't change but again because of the second term our inequality is satisfied.
3. Obvious.

Since the potential is always positive, we know that

$$\text{Initial Potential} + \text{Total Increase} \geq \text{Total Decrease}.$$

Notice that the total decrease in potential is an upper-bound on our algorithm's cost. The Total Increae in potential is upper-bounded by $k\times$ the adversary's cost. Therefore,

$$(\text{cost of your moves}) < k \times (\text{total cost of adversary's moves}) + \text{Initial Potential}.$$

References

[1] B. Awerbuch, A. Baratz, and D. Peleg, Cost-sensitive analysis of communication protocols, Proc. of 9th Symp. on Principles of Distributed Computing (PODC), pp. 177–187, (1990).

[2] I. Althöfer, G. Das, D. Dobkin, D. Joseph, and J. Soares, *On sparce spanners of weighted graphs* Discrete and Computational Geometry, 9, pp. 81–100, (1993)

[3] Bondy and Murty, *Graph Theory*

[4] J. Cong, A. B. Kahng, G. Robins, M. Sarrafzadeh and C. K. Wong, Provably good performance-driven global routing, *IEEE Transactions on CAD*, pp. 739-752, (1992).

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to algorithms*, The MIT Press and McGraw-Hill, (1990).

[6] M. L. Fredman and R. E. Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, Journal of the ACM, 34(3), pp. 596–615, (1987).

[7] M. Fujii, T. Kasami and K. Ninomiya, *Optimal sequencing of two equivalent processors"*, SIAM Journal on Applied Math, 17(4), pp. 784–789, (1969).

[8] H. N. Gabow, Z. Galil, T. Spencer and R. E. Tarjan, *Efficient algorithms for finding minimum spanning trees in undirected and directed graphs*, Combinatorica, 6 (2), pp. 109–122, (1986).

[9] Goldberg-Tarjan, *A new approach to the maximum flow problem*, Journal of the ACM 35, pp. 921–940, (1988).

[10] R. Hassin, *Maximum flows in (s,t) planar networks*, Information Processing Letters, 13, p. 107, (1981).

[11] J. E. Hopcroft and R. M. Karp, *An* $n^{2.5}$ *algorithm for maximum matching in bipartite graphs*, SIAM Journal on Computing, 2(4), pp. 225–231, (1973).

[12] N. Karmarkar and R. M. Karp, *An efficient approximation scheme for the one-dimensional bin packing problem*, Proc. 23rd Annual ACM Symp. on Foundations of Computer Science, pp.312–320, (1982).

[13] S. Khuller and J. Naor, *Flow in planar graphs: a survey of recent results*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 9, pp. 59–84, (1993).

[14] S. Khuller, B. Raghavachari, and N. E. Young, *Balancing minimum spanning and shortest path trees*, 1993 Symposium on Discrete Algorithms, (1993).

[15] D. R. Karger, P. N. Klein and R. E. Tarjan, A randomized linear time algorithm to find minimum spanning trees, *Journal of the ACM*, Vol 42(2) pp. 321–328, (1995).

[16] L. Kou, G. Markowsky and L. Berman, *A fast algorithm for steiner trees*, Acta Informatica, 15, pp. 141–145, (1981).

[17] V. M. Malhotra, M. P. Kumar, and S. N. Maheshwari, *An* $O(n^3)$ *algorithm for finding maximum flows in networks*, Information Processing Letters, 7, pp. 277–278, (1978).

[18] Nishizeki and Chiba, *Planar graphs: Theory and algorithms*, Annals of Discrete Math, Vol 32, North-Holland Mathematical Studies.

[19] Papadimitriou and Steigltiz, *Combinatorial Optimization: algorithms and complexity*, Prentice-Hall, (1982).

[20] Sleator and Tarjan, *Self-adjusting binary trees*, Proc. 15th Annual ACM Symp. on Theory of Computing, pp.235–245, (1983).

[21] R. E. Tarjan, *Data structures and network algorithms*, CBMS-NSF Regional Conference Series in Applied Mathematics, SIAM, (1983).

[22] W. Fernandez de la Vega and G. S. Lueker, *Bin packing can be solved within* $1 + \epsilon$ *in linear time*, Combinatorica, 1 pp. 349–355, (1981).

[23] A. C. Yao, *An O(—E— log log —V—) algorithm for finding minimum spanning trees*, Information Processing Letters, 4 (1) pp 21–23, (1975).