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

April 19, 1996

Preface

These are my lecture notes from **CMSC 651: Design and Analysis of Algorithms**, a one semester course that I taught at University of Maryland in the Spring of 1995. The course covers core material in 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. 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.

The course was a 15 week course, with 2 lectures per week. These notes consist of 27 lectures. There was one midterm in-class examination and one final examination. There was no lecture on the day of the midterm. No scribe was done for the guest lecture by Dr. R. Ravi.

Contents

CMSC 651 Advanced Algorithms	Lecture 1
Lecturer: Samir Khuller	Th. Jan. 19, 1995

Original notes by Hsiwei Yu.

1 Overview of Course

Read [5] Chapter 18.1–18.3 for general amortization stuff.

The course will cover many different topics. We will start out by studying various data structures together with techniques for analyzing their performance. We will then study the applications of these data structures to various graph algorithms, such as minimum spanning trees, max-flow, matching etc. 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: 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 $= < b_k b_{k-1} \dots b_1 >$. 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:

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

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:

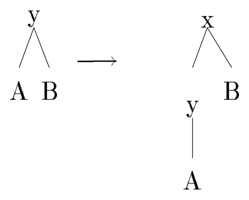
1. Search through out the tree to find node with key k.
2. Do a series of rotations (a splay) to bring it to the top.

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.)

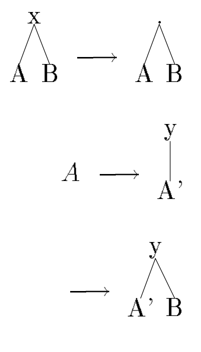
2.1 Use of Splay Operations

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

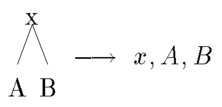
1. *Access*(x) - Simply splay to bring it to the top, so it becomes the root.
2. *Insert*(x) - Run *splay*(x) on the tree to bring y, the largest element less than x, to the top. The insert is then trivial:



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

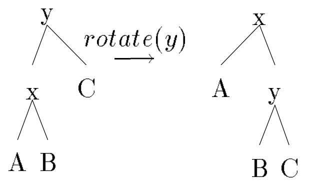
How much work does a splay operation require? We must:

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*:

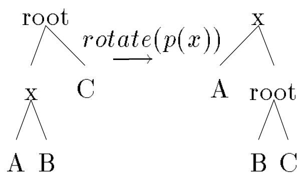


Clearly a rotation can be done in $O(1)$ time. (Note there are both left and right rotations, which are in fact inverses of each other. The sketch above depicts a right rotation going forward and a left rotation going backward. Hence to bring up a node, we do a right rotation if it is a left child, and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

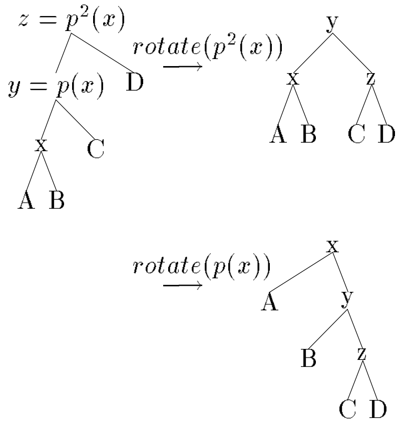
A splay is then done with a (carefully-selected) series of rotations. Let $p(x)$ be the parent of a node x. Here is the splay algorithm:

Splay Algorithm:

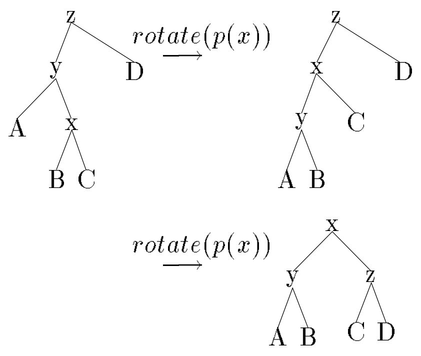
```
while $x \neq root$ do
        if $p(x) = root$ then $rotate(p(x))$
```



```
        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:

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

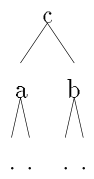
Clearly, the better balanced the tree is, the lower the potential Φ is. (You may want to work out the potential of various trees to convince yourself of this.)

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 $r(c) > 1 + min(r(a), r(b))$.



□

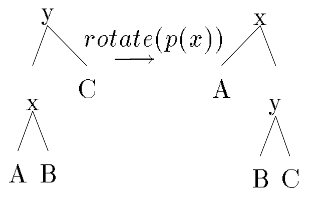
We apply this to

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

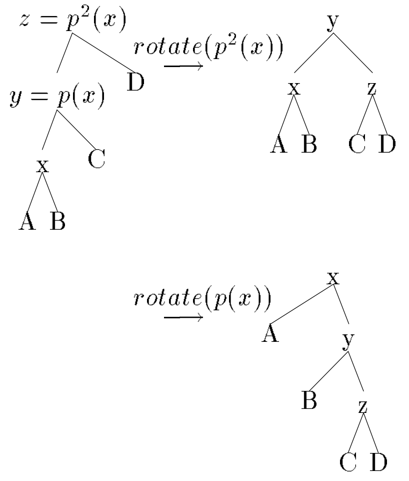


so that clearly $r'(x) = r(y)$. But then since only x and y change rank in s',

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

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

$$\begin{aligned} r''(x) &= r(x) \\ r''(y) &= r'(x)(= r(z)) \\ r''(z) &= r'(z) \end{aligned}$$

so that

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

Hence, either we have

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

or

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

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

$$at < 1 + 3(\textit{ final rank of } x - \textit{ initial rank of } x)$$

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

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

CMSC 651 Advanced Algorithms	Lecture 4
Lecturer: Samir Khuller	Tu. Jan 31, 1995

Original notes by Matos Gilberto and Patchanee Ittarat.

4 Maintaining Disjoint Set's

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

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 [5] 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:

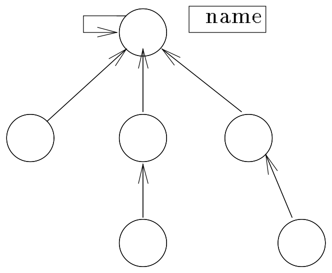


Figure 1: Data Structure

Find - worst case cost is $O(n)$.
Union - worst case cost is $O(1)$.
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.

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

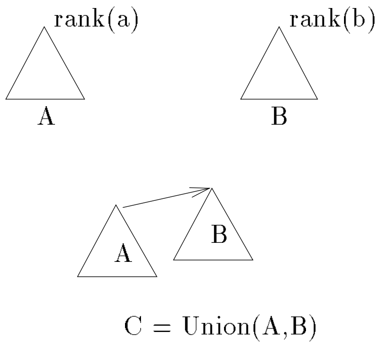


Figure 2: Union

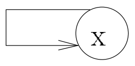


Figure 3: Rank 0 node

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]
$rank(x) = 0 \Rightarrow x$ has no descendants.
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 *$rank(a) < rank(b)$:*

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

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

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

□

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 the root is being searched for.

Path compression takes two passes – first to find the root, and second time to change the pointers of all nodes on the path to the root so that they point directly to the root. So 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 one 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 a 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 (due to the change of government!).

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 loose some 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. $\square$

We introduce the fast growing function F which is defined as (in class I defined $F(1) = 1$ but the proof is essentially the same).

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. Now 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 because the rank of the parent only goes up. This means that the find operation is going to pay for the work on that node every time. So any node will first 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

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

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 alpha 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 disjoint set data structures on the pointer machine model.

Notes by Samir Khuller.

5 Minimum Spanning Trees

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

Given a graph $G = (V, E)$ and a weight function $w : E \longrightarrow R^+$ 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.

5.1 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?).

5.2 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 queue. (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);
    $T, VS \leftarrow \emptyset$;
    $\forall v : \ell[v] \leftarrow 1$;
    while $|VS| > 1$ do
        Let $W$ be the first set in the queue VS;
        For each $v \in W$ do;
            small$[v] \leftarrow \infty$;
            while small$[v] = \infty$ and $\ell[v] \leq k$ do
                For each edge $e = (v, v')$ in $E_v^{\ell[v]}$ do
                    If $find(v') = find(v)$ then delete $e$ from $E_v^{\ell[v]}$
                      else small$[v] \leftarrow$ min (small$[v]$, $w(e)$)
                If small$[v] = \infty$ then $\ell[v] \leftarrow \ell[v] + 1$
            end-while
        end-for
        Let the lowest weight edge out of $W$ be $(w, w')$ where $w \in W$ and $w' \in W'$;
        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 $O(m \log \log n)$.

CMSC 651 Advanced Algorithms	Lecture 6
Lecturer: Samir Khuller	Tu. 7 Feb., 1995

Notes by Samir Khuller.

6 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 7
Lecturer: Samir Khuller	Th. Feb. 9, 1995

Original notes by King-Ip Lin.

7 Heaps

This material is from [5] Chapter 20 and 21. Please read the book for a comprehensive description.

We will use heaps to implement MST and Shortest Path algorithms since they provide a quick way of computing the minimum element in a set.

Definition 7.1 (d-Heaps) *A d-heap is a tree which the following properties hold:*

1. *Each node have a key attached to it*
2. *Every internal node (except parents of leaves) have exactly d children*
3. [Heap-order property] *For any node x, $key(parent(x)) \leq key(x)$*

Basic operations on heaps

Insert Attach the new item to the rightmost unfilled parent of leaves and restore the heap-order-property by pushing the new key upwards.

Delete Swap element to be deleted to the rightmost child. Remove the rightmost child and restore the heap-order property of the swapped node.

Search Except for the minimum (at the top of the tree), not well supported

Time for insert and delete = height of tree = $\log n$. Please see Tarjan's book for more details on heaps.

7.1 Binomial heaps

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

Definition 7.2 (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 child of root (degree of root) : k

In order to store n nodes in binomial tress when $n \neq 2^k$, first write n in binary notation, then for every 1 bit in the notation, 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 7.3 (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 is still 2 $B_i$ tree do
            join the two $B_i$ tree 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 formed i binomial trees, from B_0 to B_{i-1}

```
Step 1: Find the minimum root (Assume its in $B_k$)
Step 2: Break $B_k$, forming $k$ smaller trees
Step 3: while there is still at least 2 $B_i$ tree do
            join the two $B_i$ tree to form a $B_{i+1}$ tree
            $i \leftarrow i+1$
```

Note that at each stage there will be at most 3 B_i trees, thus for each i only 1 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
```

2 and 3 is to be grouped and called DECREASE_KEY(x)
Time for insertion and deletion : $O(\log n)$

7.2 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 the F-heap :

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 1 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: results may not be a true binomial tree)

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

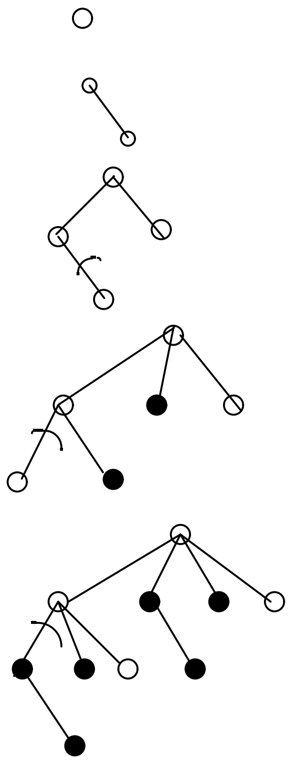
Step 4: Put \$1 to the new root

Step 5: Pay \$1 for the cut

Problem with the above algorithm: Can result in trees where the root have disportionly large number of child (i.e. not enough internal nodes).
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

CMSC 651 Advanced Algorithms	Lecture 8
Lecturer: Samir Khuller	Tu. Feb 14, 1995

Original notes by Patchanee Ittarat.

8 F-heap

8.1 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 for any k, i.e., we can have any number of B_k at one time,
- trees are not true binomial trees.

For example,

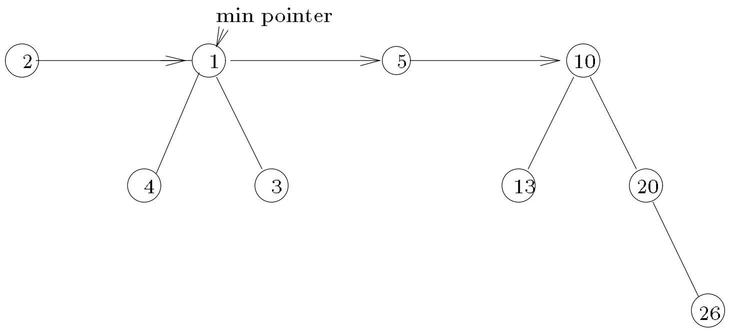


Figure 5: Example of F-heap

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.

8.2 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.

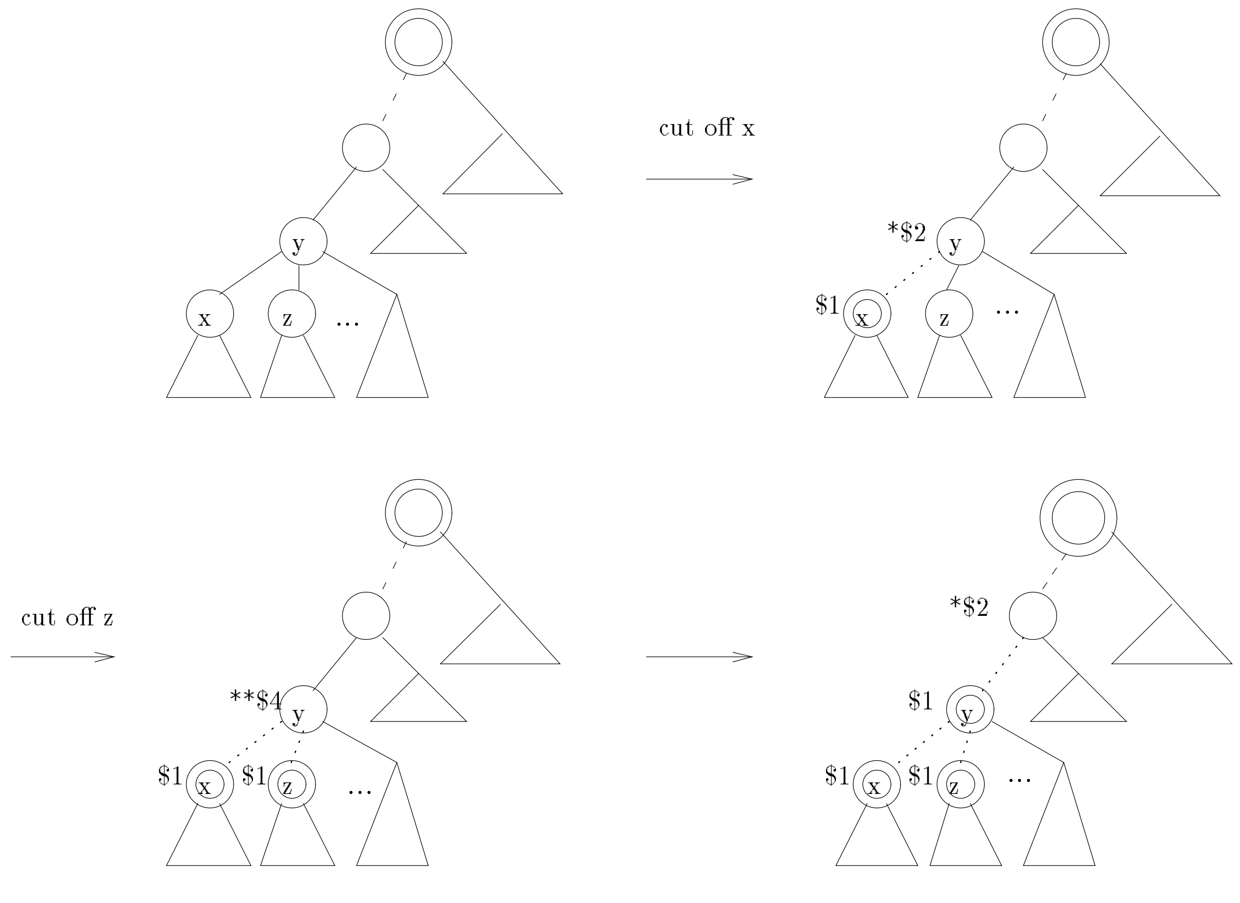


Figure 6: Marking Strategy

Example: How does mark strategy work?

The cost of Decrease-Key operation = cost of cutting link + \$3. We can see that no extra dollars needed when the parent is cut off recursively since when the cut off is needed, 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)?

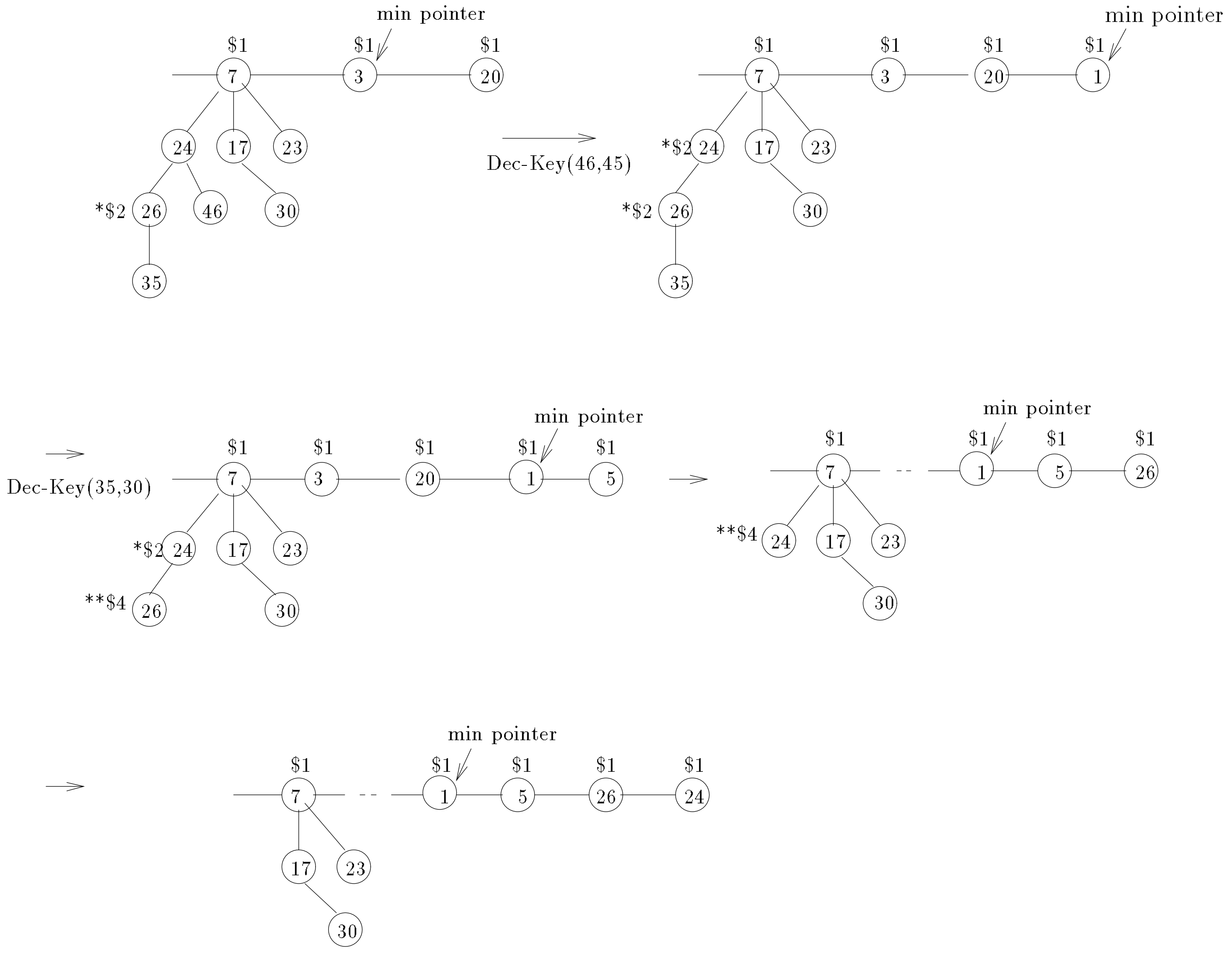


Figure 7: Example

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

We want to prove that y_i has some subtrees.

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 y_i.

And since y_i has lost at most 1 child, so now y_i has at least degree $i-2$.

Let's define

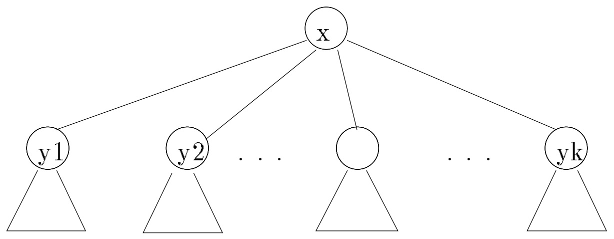


Figure 8: Example

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

These numbers are called Fibonacci numbers.

Property 8.1 $F_{k+2} = 1 + \sum_{i=1}^{k} F_i$

Proof:
[By induction]

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

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

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

□

Property 8.2 $F_{k+2} \geq \phi^k$

Proof:
[By induction]

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

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

$$\begin{aligned} k = n: \qquad F_{n+2} &= F_{n+1} + F_n \\ &\geq \phi^{n-1} + \phi^{n-2} \\ &\geq \frac{1+\phi}{\phi^2} \cdot \phi^n \\ &\geq \phi^n \end{aligned}$$

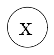
□

Theorem 8.1 *x is a root of any subtree. $size(x) \geq F_{k+2} \geq \phi^k$ where k is a degree of x*

Proof:

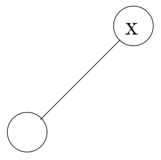
[By induction]

$k = 0$:



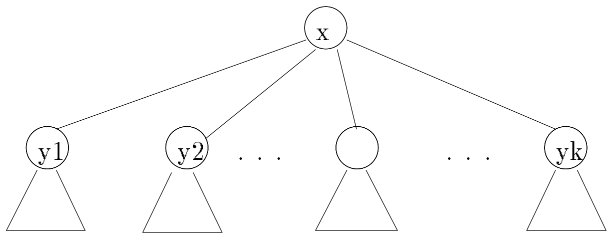
$size(x) = 1 \geq 1 \geq 1$

$k = 1$:



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

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



$k = n + 1$:

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

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

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

Notes by Samir Khuller.

9 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 $\forall v \; \text{dist}_T(r,v) \leq (1+\epsilon)\text{dist}(r,v)$. 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.

9.1 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 9.1 *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 ($0 < i \leq k$), 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 9.2

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

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 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 10.1 *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 10.2 *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 10.3 *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 $w(C-e) \le t \cdot w(e)$. Assume that e_{max} is the heaviest weight edge on the cycle. Then $w(C) \le (t+1) \cdot w(e) \le (t+1) \cdot w(e_{max})$. 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 10.4 *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 $w(E(v)) \le \frac{2w(MST)}{(t-1)}$.

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 $E(v)$).

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

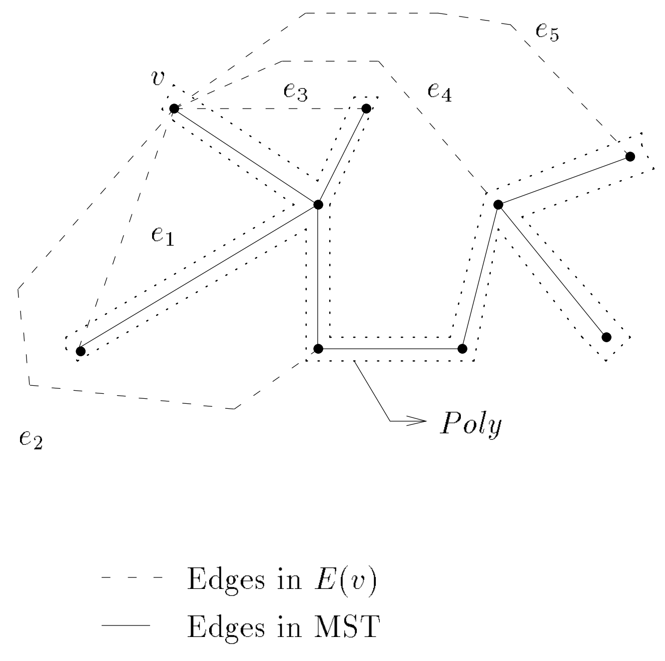


Figure 9: Figure to illustrate polygon, MST and $E(v)$.

Theorem 10.5

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

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.

CMSC 651 Advanced Algorithms	Lecture 10
Lecturer: Samir Khuller	Tu., Feb. 21, 1995

Original notes by Michael Tan.

11 Matchings

Read: Chapter 9 [21]. Papadimitriou and Steiglitz [19] also has a nice description of matching.

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.
□

For non-bipartite graphs, this theorem does not work. Tutte proved the following (you can read the Graph Theory book by Bondy and Murty for a proof) theorem 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 (edges) which begins at a free vertex and whose edges alternate between matched and unmatched edges.

An **augmenting path** is an alternating path which starts and ends with unmatched edges (and thus starts and ends with free vertices).

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 $M \oplus P = (M \cup P) - (M \cap P)$ 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, half of which are from M and half of which are 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.

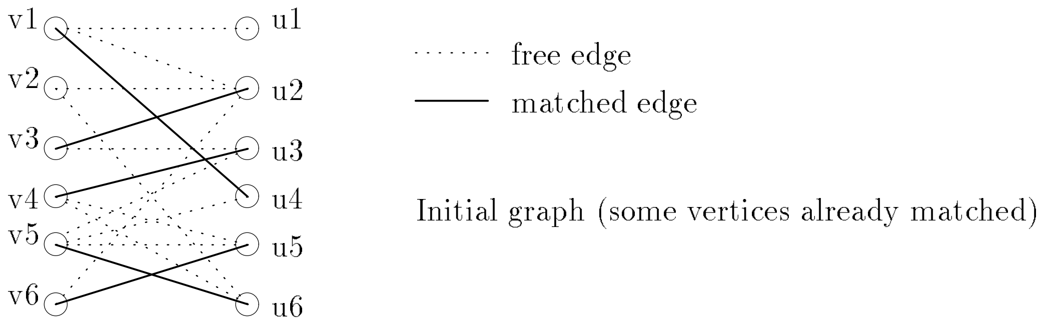
Step 3: Increase M by 1 (using the augmenting path).

Step 4: Go to 2.

We will discuss the search for an augmenting path in bipartite graphs via a simple example (see Fig. 10).

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.



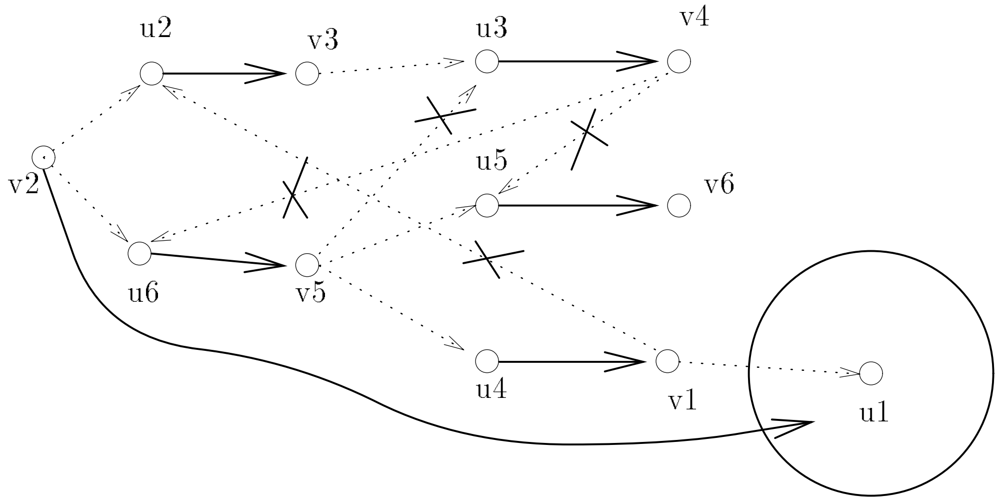


Figure 10: Sample execution of Simple Matching Algorithm.

CMSC 651 Advanced Algorithms	Lecture 11
Lecturer: Samir Khuller	Th. Feb. 23, 1995

Notes by Samir Khuller.

12 Hopcroft-Karp Matching Algorithm

The original paper is by Hopcroft and Karp [11].

We present the Hopcroft-Karp matching algorithm along with a proof of Hall's theorem (I didnt get around to doing this during the lecture, but we'll finish this in the next lecture).

Theorem 12.1 (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 $|N(S)| < |S|$. □

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 12.2 *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 12.3 *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. $\square$

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.

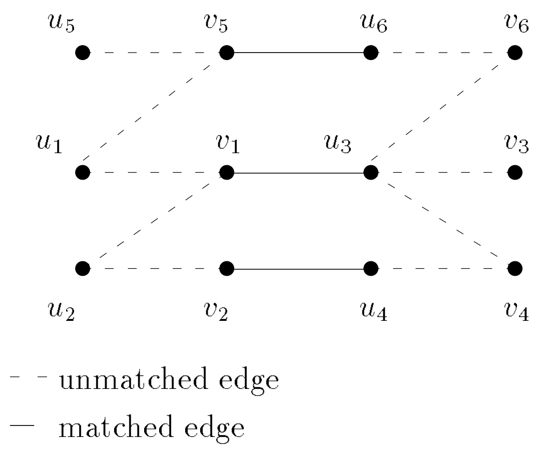


Figure 11: Sample execution of Topological Erase.

Theorem 12.4 *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:

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

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 12
Lecturer: Samir Khuller	Tu. Feb. 28, 1995

Notes by Samir Khuller.

13 Two Processor Scheduling

Most of this lecture was spent in wrapping up odds and ends from the previous lecture. 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.

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).

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. 13). 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.

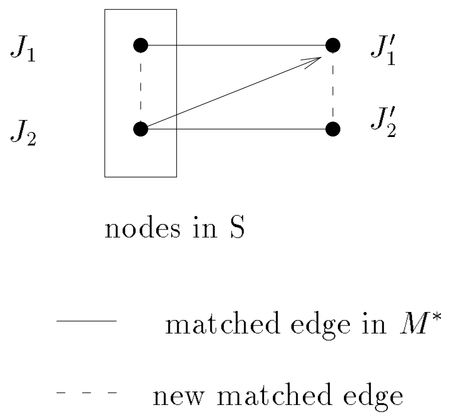


Figure 12: Changing the schedule.

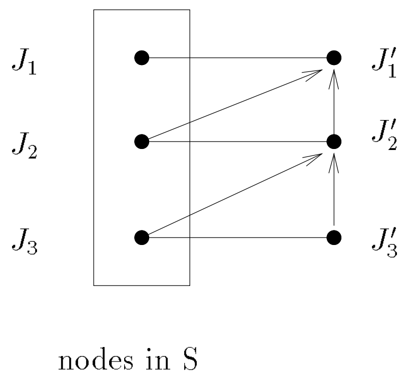


Figure 13: Continuing the proof.

CMSC 651 Advanced Algorithms	Lecture 13
Lecturer: Samir Khuller	Th. Mar. 2, 1995

Notes by Samir Khuller.

14 Assignment Problem

See [3] for a nice description of the Hungarian Method.

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,

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

□

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.

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. 14).

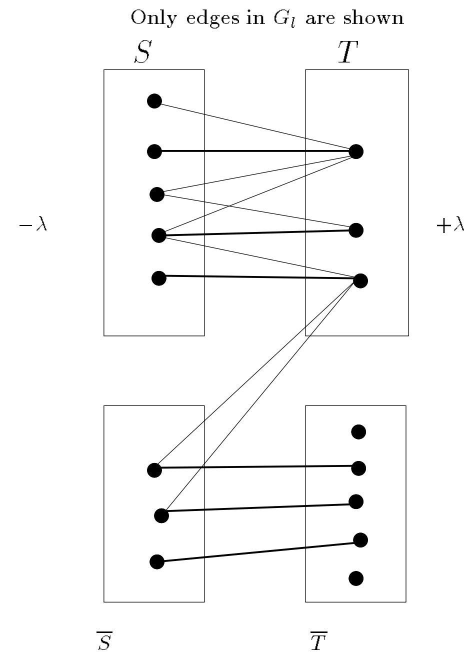


Figure 14: Sets S and T as maintained by the algorithm.

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.

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 14
Lecturer: Samir Khuller	Th. Mar. 9, 1995

Original notes by Sibel Adali and Andrew Vakhutinsky.

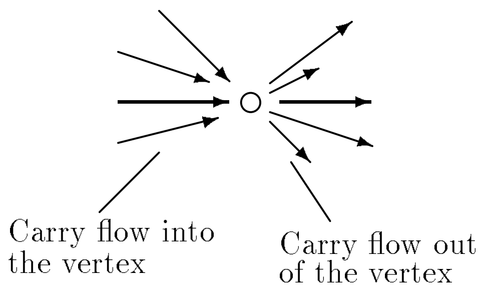
15 Network Flow - Maximum Flow Problem

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 $0 - (-6) = 6$.

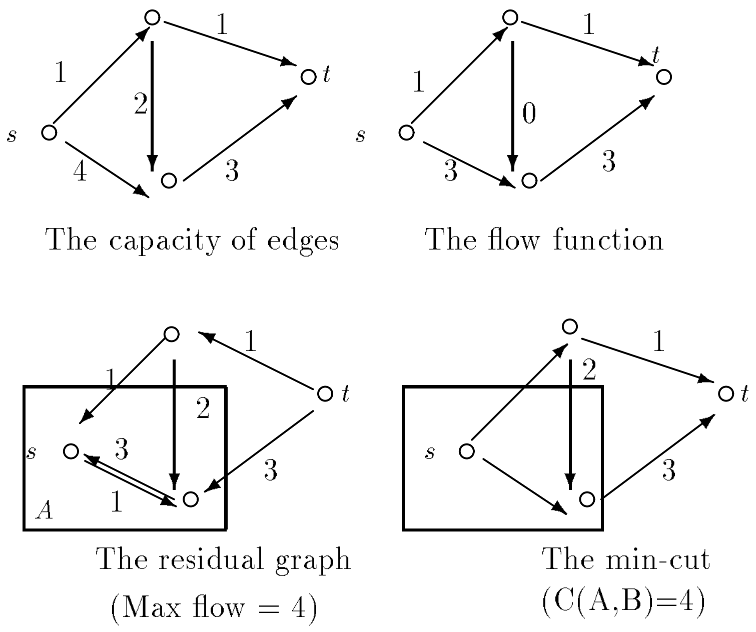


Figure 15: An example

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 $s \in A, t \in B$.*

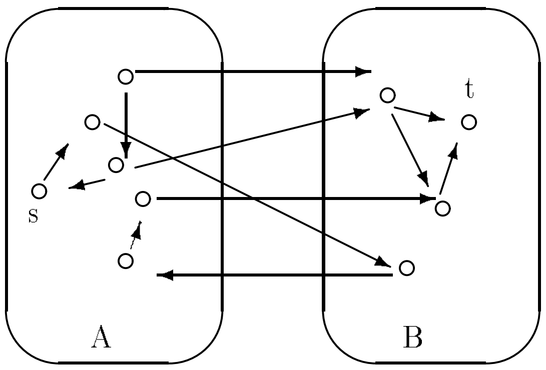


Figure 16: An (s, t) Cut

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. 17. 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. 18 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. 18 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.

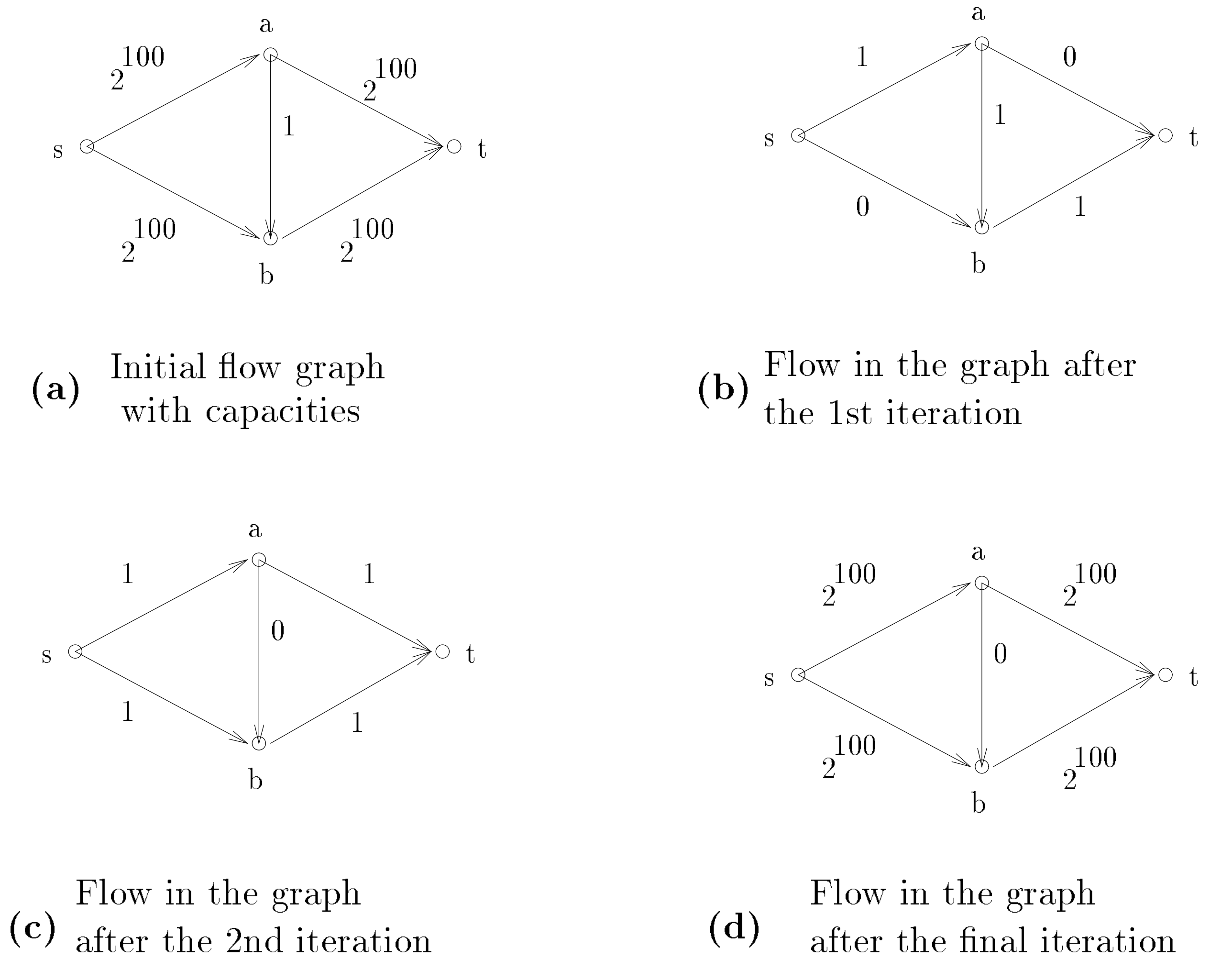
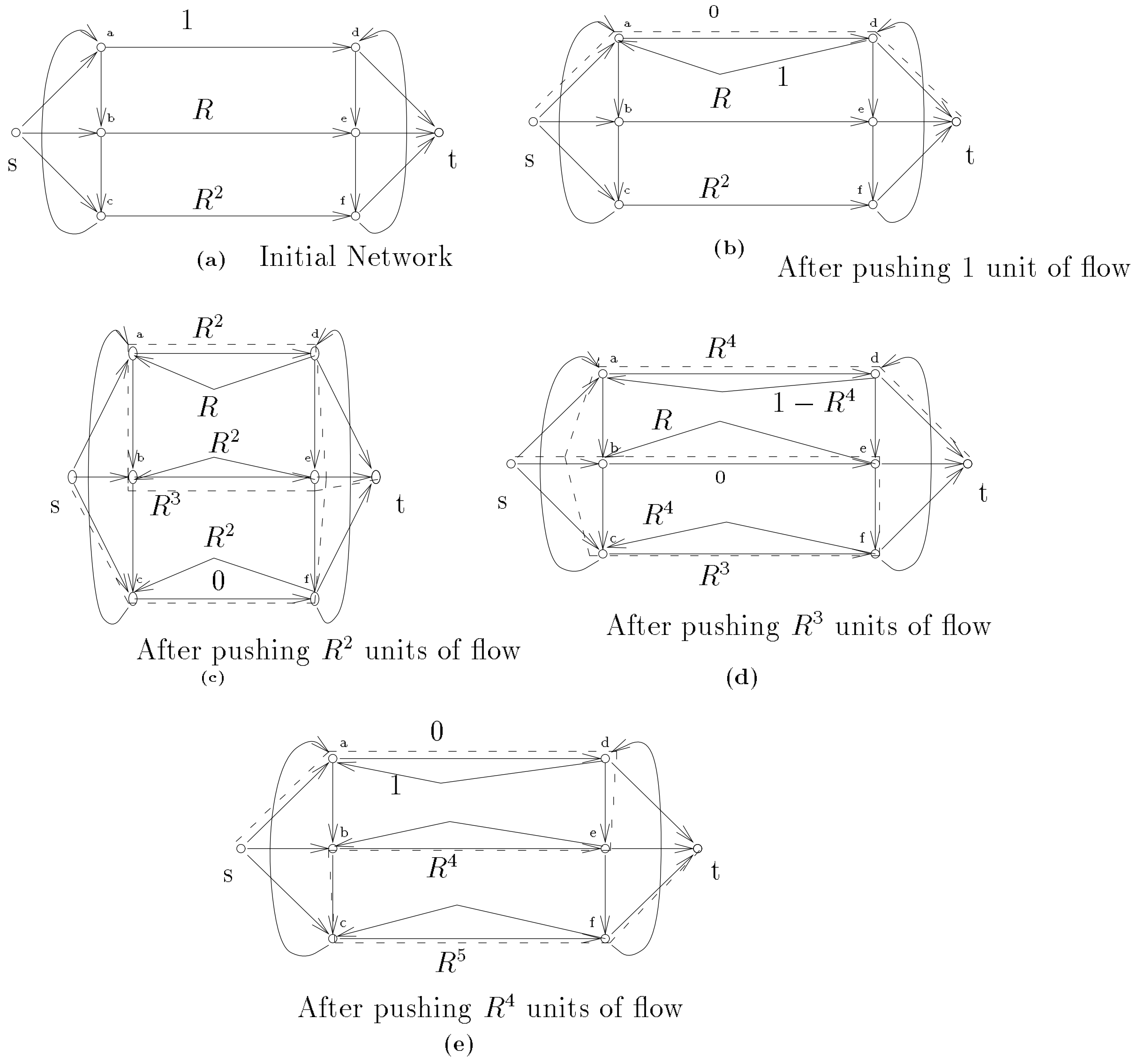

a
s
t
b
2^{100}
1
0
(a) Initial flow graph with capacities
(b) Flow in the graph after the 1st iteration
(c) Flow in the graph after the 2nd iteration
(d) Flow in the graph after the final iteration

Figure 17: Worst Case Example



1
R
R^2
a
b
c
d
e
f
s
t
(a) Initial Network
0
1
R
R^2
(b) After pushing 1 unit of flow
R^2
R
R^2
R^3
R^2
0
After pushing R^2 units of flow
(c)
R^4
$1 - R^4$
R
0
R^4
R^3
After pushing R^3 units of flow
(d)
0
1
R^4
R^5
After pushing R^4 units of flow
(e)

Figure 18: Non-terminating Example

CMSC 651 Advanced Algorithms	Lecture 15
Lecturer: Samir Khuller	March 14, 1995

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 $\geq \frac{|f|}{m}$.

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 $\frac{|f^*|}{m}$ 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

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

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

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

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?

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

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

$$k \geq m \log |f^*|.$$

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 $O(m + n \log n)$ (HW 4), 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) $O(n^2 m)$
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 | Lectures 16, 17, 18 and 19
Lecturer: Samir Khuller | March 28, 30, Apr 4 and 6 1995

Notes by Samir Khuller.

17 The Max Flow Problem

Lecture 16

We reviewed the mid-term exam.

Lecture 17

We covered the $O(n^3)$ max flow algorithm. The specific implementation that we studied is due to Malhotra, Kumar and Maheshwari [17]. The main idea is to build the layered network from the residual graph and to find a *blocking flow* in each iteration. We showed how a blocking flow can be found in $O(n^2)$ time. [See handout from the book by Papadimitriou and Steiglitz "Combinatorial Optimization: Algorithms and Complexity".]

Lecture 18

We studied the preflow-push method. This is described in detail in Chapter 27 [5].

Lecture 19

We will study the "lift-to-front" heuristic that can be used to implement the preflow push method in $O(n^3)$ time. This is also described in Chapter 27 in [5].

CMSC 651 Advanced Algorithms	Lecture 20
Lecturer: Samir Khuller	Tu. April 11, 1995

Original notes by Vadim Kagan.

18 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]).

18.1 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.

18.2 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 18.1 (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.

18.3 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. 19). 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.

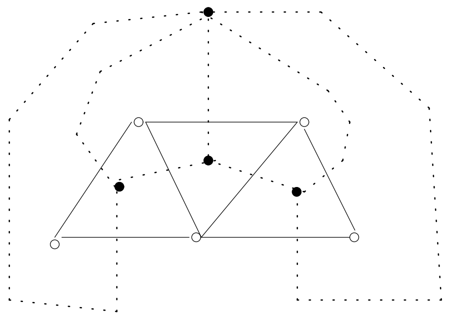


Figure 19: Graph with corresponding dual

18.4 Two algorithms

Uppermost Path Algorithm (Ford and Fulkerson)

1. Take the uppermost path (see Fig. 20).
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.

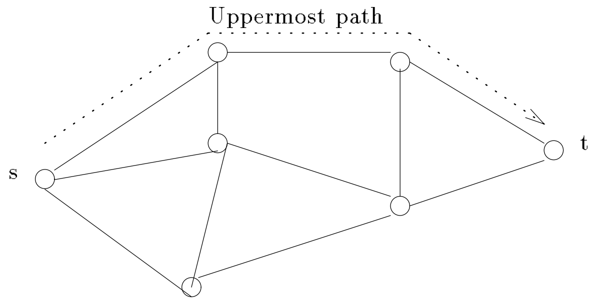


Figure 20: Uppermost path

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

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 $t^\star$.

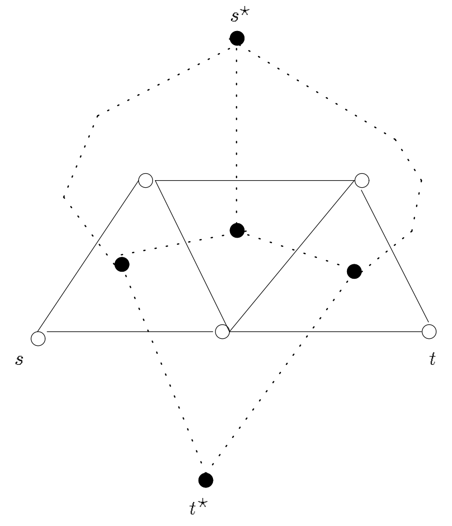


Figure 21: $G^{D\star}$

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. 22).

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. 23), 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 24). 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.

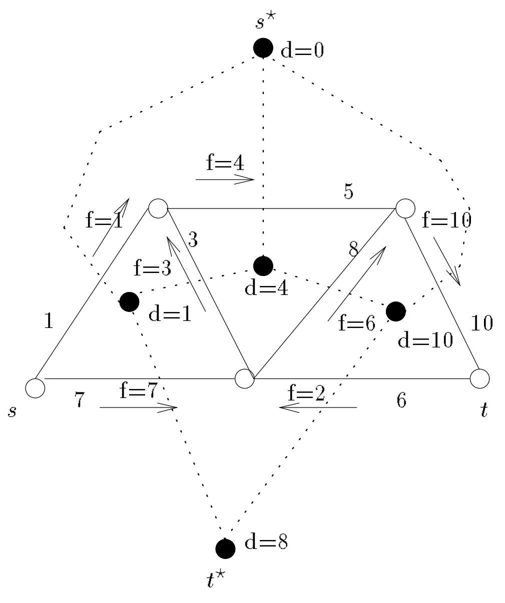


Figure 22: Labeled graph

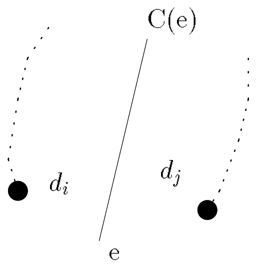


Figure 23:

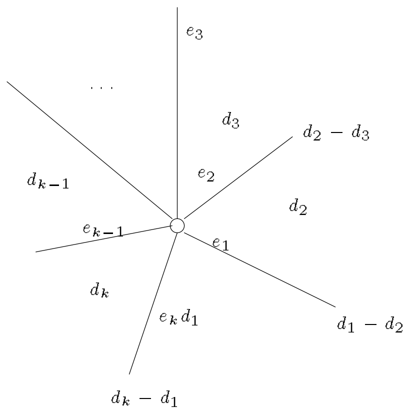


Figure 24:

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

Original notes by Marsha Chechik.

19 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 19.1 *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.

19.1 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.*

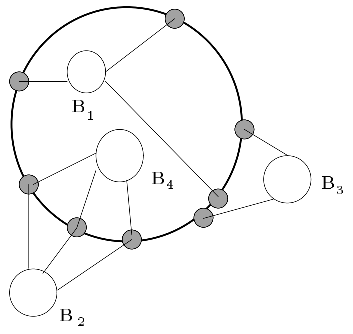
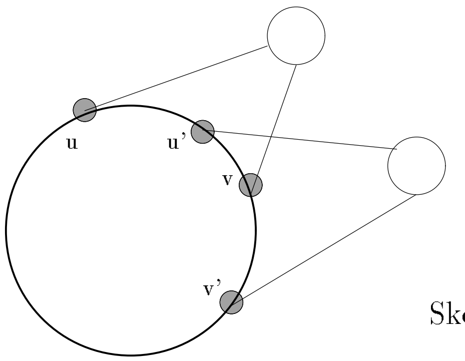


Figure 25: Bridges relative to the cycle C.

Definition: Two bridges *avoid* each other with respect to the cycle if all vertices of attachment are on the same segment. In Fig. 25, 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. 25). 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. 26).



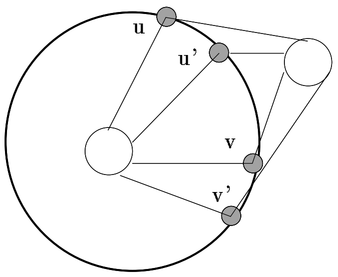


Figure 26:

- 3-equivalent (like B_2 and B_4 in Fig. 25)

It can be shown that other cases reduce to the above cases. For example, a 4-equivalent graph (see Fig.26) can be classified as a skew, with vertices u and v belonging to one bridge, and vertices u' and v' belonging to the other.

19.2 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 19.2 *A minimal non-planar graph that does not contain a homeomorph of K_5 or $K_{3,3}$ is 3-connected.*

Theorem 19.3 (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:

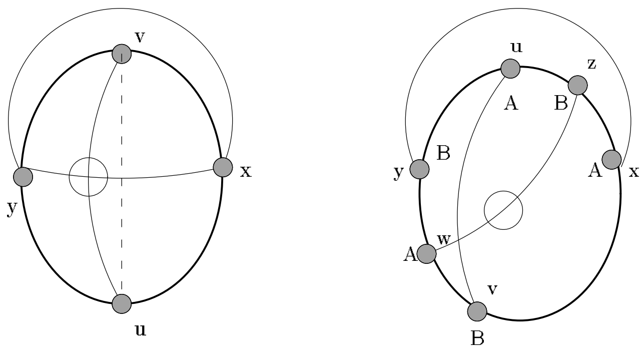
1. Every external bridge with respect to C has exactly two attachment points. Otherwise, a path in this bridge going between two attachment points, could be added to C to obtain more edges inside C.

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 27: Possible vertex-edge layouts.

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

1. See Fig. 27(a). This graph contains a homeomorph of K_5. Notice that every vertices are connected with an edge.

2. See Fig. 27(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. □

CMSC 651 Advanced Algorithms Lecture 22
Lecturer: Samir Khuller Tu. Apr. 18, 1995

Notes by Samir Khuller.

20 NP-Completeness

Please read Chapter 36 [5] for definitions of P, NP, NP-hard, NP-complete, polynomial time reductions, SAT etc. You need to understand the concept of "decision problems", non-deterministic Turing Machines, and the general idea behind how COOK's Theorem works.

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

Notes by Samir Khuller.

21 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.

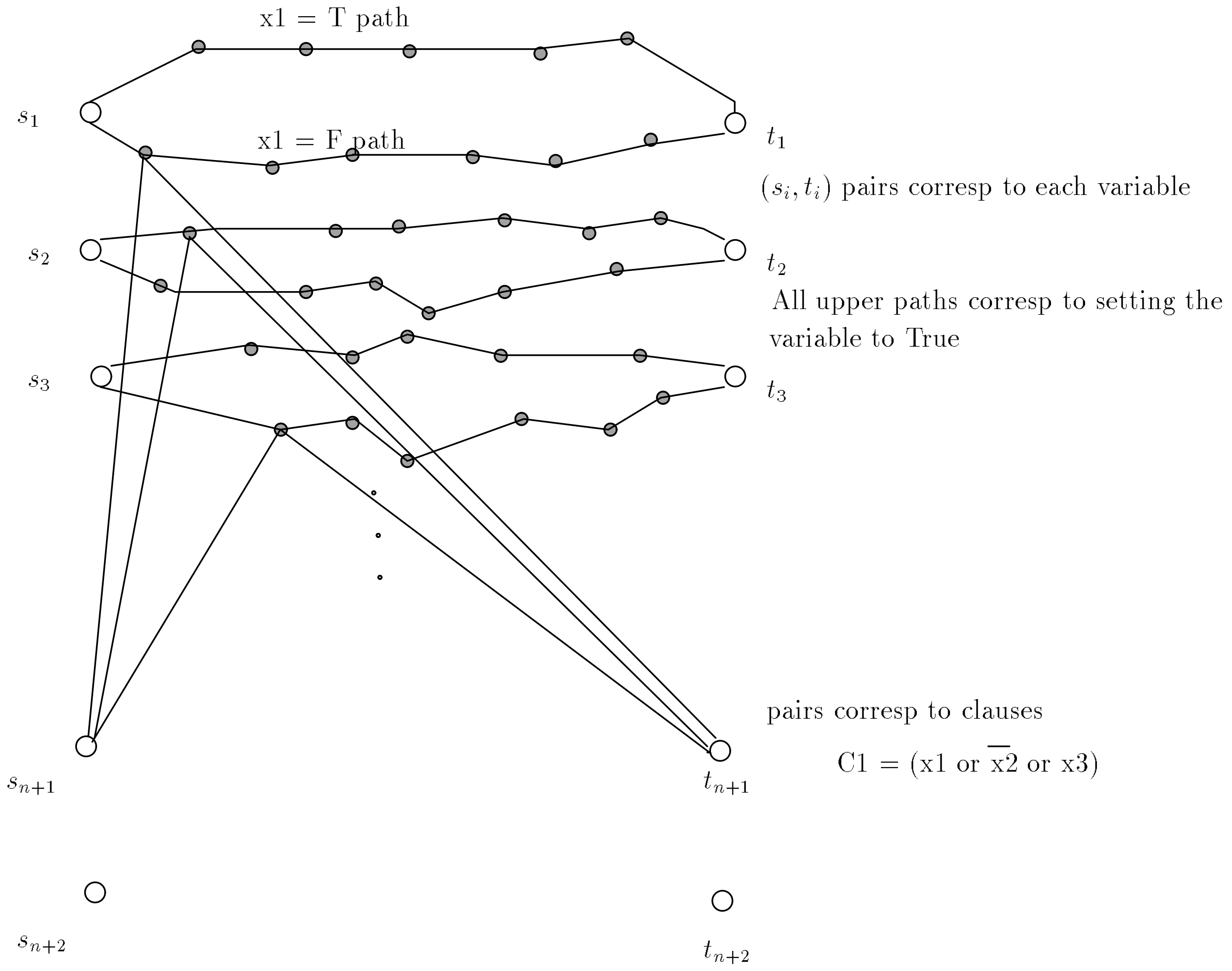

x1 = T path
x1 = F path
s_1
t_1
(s_i, t_i) pairs corresp to each variable
s_2
t_2
All upper paths corresp to setting the variable to True
s_3
t_3
pairs corresp to clauses
C1 = (x1 or $\overline{\text{x2}}$ or x3)
s_{n+1}
t_{n+1}
s_{n+2}
t_{n+2}

Figure 28: Graph corresponding to formula

CMSC 651 Advanced Algorithms	Lecture 24
Lecturer: Samir Khuller	Tu. Apr. 25, 1995

Notes by Samir Khuller.

22 Approximation Algorithms

Reading Homework: Please read Chapter 37.1 and 37.2 from [5] for the Vertex Cover algorithm and the Traveling Salesman Problem.

Please read Chapter 37 (pp. 974–978) from [5] for Set Cover algorithm that was covered in class.

It is easy to show that the Set Cover problem is NP-hard, by a reduction from the Vertex Cover problem, which we know is NP-complete. (In fact, Set Cover is a generalization of Vertex Cover. In other words, Vertex Cover can be viewed as a special case of Set Cover with the restriction that each element (edge) occurs in exactly two sets.)

CMSC 651 Advanced Algorithms	Lecture 25
Lecturer: Samir Khuller	Th. Apr. 27, 1995

Notes by Samir Khuller.

23 Unweighted Vertex Cover

GOAL: Given a graph $G = (V, E)$, find a minimum cardinality vertex cover C, where $C \subseteq V$ such that for each edge, at least one endpoint is in C.

For the unweighted case, it is very easy to get an approximation factor of 2 for the vertex cover problem. We first observe that a maximum matching $|M^*|$ gives us a lower bound on the size of any vertex cover. Next observe that if we find a maximum matching, and pick all the matched vertices in the cover, then the size of the cover is exactly $2|M^*|$ and this is at most 2 OPT-VC, since OPT-VC $\geq |M^*|$. The set of matched vertices forms a vertex cover, since if there was an edge between two unmatched vertices it would imply that the matching was not a maximum matching. In fact, it is worth pointing out that all we need is a maximal matching for the upper bound and not a maximum matching. (The matched vertices in a maximal matching also form a vertex cover.) If we can find a maximal matching that is significantly smaller than a maximum matching then we may obtain a smaller vertex cover!

When the graph is bipartite, one can convert the matching into a vertex cover by picking exactly one vertex from each matched edge (this was done in the midterm solutions). This produces an optimal solution in bipartite graphs.

24 Weighted Vertex Cover

GOAL: Given a graph $G = (V, E)$, find a minimum weight vertex cover

$$w(C) = \sum_{v \in C} w(v),$$

where $C \subseteq V$ is a vertex cover, and $w : V \rightarrow R^+$

We first introduce the concept of a packing function and then see why it is useful.

Packing Functions:

A packing $p : E \rightarrow \Re$ is a non-negative function defined as follows:

$$\forall v, \sum_{u \in N(v)} p(u, v) \leq w(v).$$

We will refer to this as the *packing condition* for vertex v.

For an example of a packing function see Fig. 29.

Lemma 24.1 *Let C^* be the minimum weight vertex cover.*

$$w(C^*) = \sum_{v \in C^*} w(v) \geq \sum_{e \in E} p(e).$$

Proof:

$$w(C^*) = \sum_{v \in C^*} w(v) \geq \sum_{v \in C^*} (\sum_{u \in N(v)} p(u, v)) \geq \sum_{e \in E} p(e).$$

This is true because some edges may be counted twice in the third expression, but each edge is counted at least once since C^* is a vertex cover. □

Maximal packing: is a packing for which we cannot increase $p(e)$ for any e without violating the packing condition for some vertex. In other words, for each edge at least one end point has its packing condition met with equality.

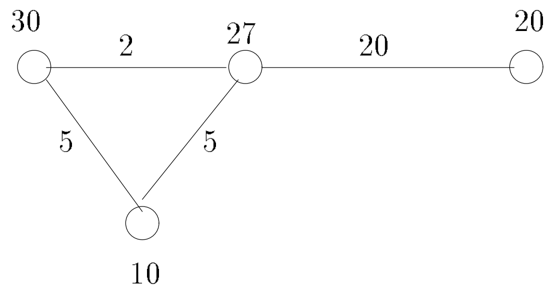


Figure 29:

We now have to somehow(?) cough up a vertex cover. We will use a maximal packing for this (ofcourse, we still need to address the issue of finding a maximal packing). One thing at a time....

Pick those nodes to be in the vertex cover that have their packing condition met with equality. In a maximal packing each edge will have at least one end point in the cover, and thus this set of nodes will form a valid vertex cover. Vertex v is in the VC if $\sum_{u \in N(v)} p(u,v) = w(v)$.,

Let C be the cover induced by the maximal packing.

Lemma 24.2

$$w(C) \leq 2 \sum_{e \in E} p(e)$$

Proof:

$$w(C) = \sum_{v \in C} w(v) = \sum_{v \in C} \sum_{u \in N(v)} p(u,v).$$

Since each edge in E is counted at most twice in the last expression, we have

$$w(C) \leq 2 \sum_{e \in E} p(e).$$

Another way to view the proof, is to have each vertex chosen in the cover distribute its weight as bills to the edges that are incident on it (each bill is the packing value of the edge). Since each edge gets at most two bills, we obtain an upper bound on the weight of the cover as twice the total sum of the packing values. □

Combining with the first lemma, we conclude

$$w(C) \leq 2w(C^*).$$

There are *many different* algorithms for finding a maximal packing. (Just greedily increasing the packing values for the edges also works. The algorithm presented here was obtained before it was realised that any maximal packing will work.) We review a method that can be viewed as a modification to the simple greedy algorithm.

IDEA : At each stage, select the vertex that minimizes the ratio between the *current* weight $W(v)$ and the *current* degree $D(v)$ of the vertex v.

```
proc Modified-Greedy;
    $\forall v \in V$ do $W(v) \leftarrow w(v)$
    $\forall v \in V$ do $D(v) \leftarrow d(v)$
    $\forall e \in E$ do $p(e) \leftarrow 0$
    $C \leftarrow \emptyset$
    while $E \neq \emptyset$ do
        Pick a vertex $v \in V$ for which $\frac{W(v)}{D(v)}$ is minimized
        $C \leftarrow C + v$
        For all $e = (v, u) \in E$ do
            $p(e) \leftarrow \frac{W(v)}{D(v)}$
            $E \leftarrow E - e$; update $D(u)$
            $W(u) \leftarrow W(u) - p(e)$
            Delete $v$ from $G$
        end-for
    end-while
    return $C$
end proc;
```

In this algorithm, each time a vertex is placed in the cover, each of its neighbors has its weight reduced by an amount equal to the ratio of the selected vertex's *current* weight and degree. The packing values of all edges incident to vertex v, that was chosen, are defined as $\frac{W(v)}{D(v)}$.

Observe that at all times during the execution of the algorithm, the following invariant holds:

$$\forall e \in E \quad p(e) \geq 0.$$

The current weight of a vertex is reduced only when its neighbor is selected. Since the selected vertex has a smaller weight to degree ratio, then the result of subtraction must be non-negative (make sure that you understand why this is the case).

$$\forall v \in V \quad w(v) = W(v) + \sum_{u \in N(v)} p(u, v).$$

The algorithm terminates with a maximal packing because each time we delete v (together with its incident egdes), its packing condition is met with equality and no further increase can be done on the packing value of any edge incident to v.

$$\forall v \in C, w(v) = \sum_{u \in N(v)} p(u, v) \tag{1}$$

$$\forall v \notin C, w(v) \geq \sum_{u \in N(v)} p(u, v) \tag{2}$$

In fact, we can use the above method to get an r-approximation algorithm for the set cover problem whenever we have the property that each element occurs in at most r sets. For certain situations this may be better than the Greedy Set Cover algorithm.

CMSC 651 Advanced Algorithms	Lecture 26
Lecturer: Samir Khuller	Tu May 2, 1995

Notes by Samir Khuller.

25 Traveling Salesperson Problem

Chapter 37.2 [5] discusses the traveling salesperson problem in detail, and describes an approximation algorithm that guarantees a factor 2 approximation. Read that subsection before continuing with these notes.

We will discuss Christofides heuristic that guarantees an approximation factor of 1.5 for TSP. The main idea is to "convert" the MST to an Eulerian graph without "doubling" all the edges. Let S be the subset of vertices that have odd degree (observe that $|S|$ is even). Now find a perfect matching M among the nodes of S and add these edges to the MST. The union of the MST and the matching yields an Eulerian graph. Find an Euler tour and "shortcut" it as in the "doubling" the MST method.

We now prove that

$$weight(MST) + weight(M) \leq 1.5 \text{ OPT-TSP} .$$

The weight of the MST is clearly at most the weight of the optimal TSP tour. We now argue that the matching M has weight at most $\frac{1}{2}$ OPT-TSP. Consider an optimal TSP tour. Mark the vertices of S on the tour. The tour induces two distinct perfect matchings: match the odd nodes of S to the right, or match the odd nodes of S to the left. These two matchings add up in weight to at most the length of the optimal tour. Clearly, one of them must have length at most half the optimal tour length. Since we found a minimum perfect matching on the nodes in S, the length of our matching is no longer. This finishes the proof.

26 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. This algorithm was developed by Kou, Markowsky and Berman [16].

26.1 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).

The worst case time complexity is clearly polynomial.

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 in S a number the first time it is encountered.) Traverse the tree in same order of the DFS then return to starting vertex.

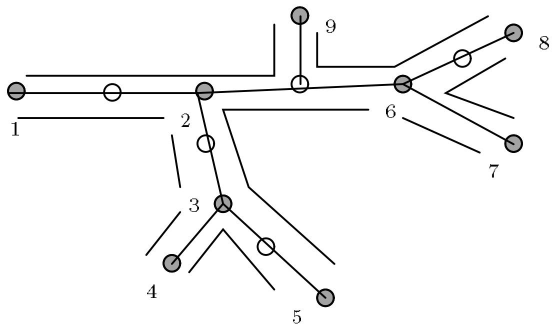


Figure 30: Optimal Steiner Tree

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 $\sum_{i=1}^{|S|} d(i, i+1) = 2 \times weight(T_{opt})$. (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 31 shows that the worst case performance of 2 is achievable by this algorithm.

26.2 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 $mM + 3n$.

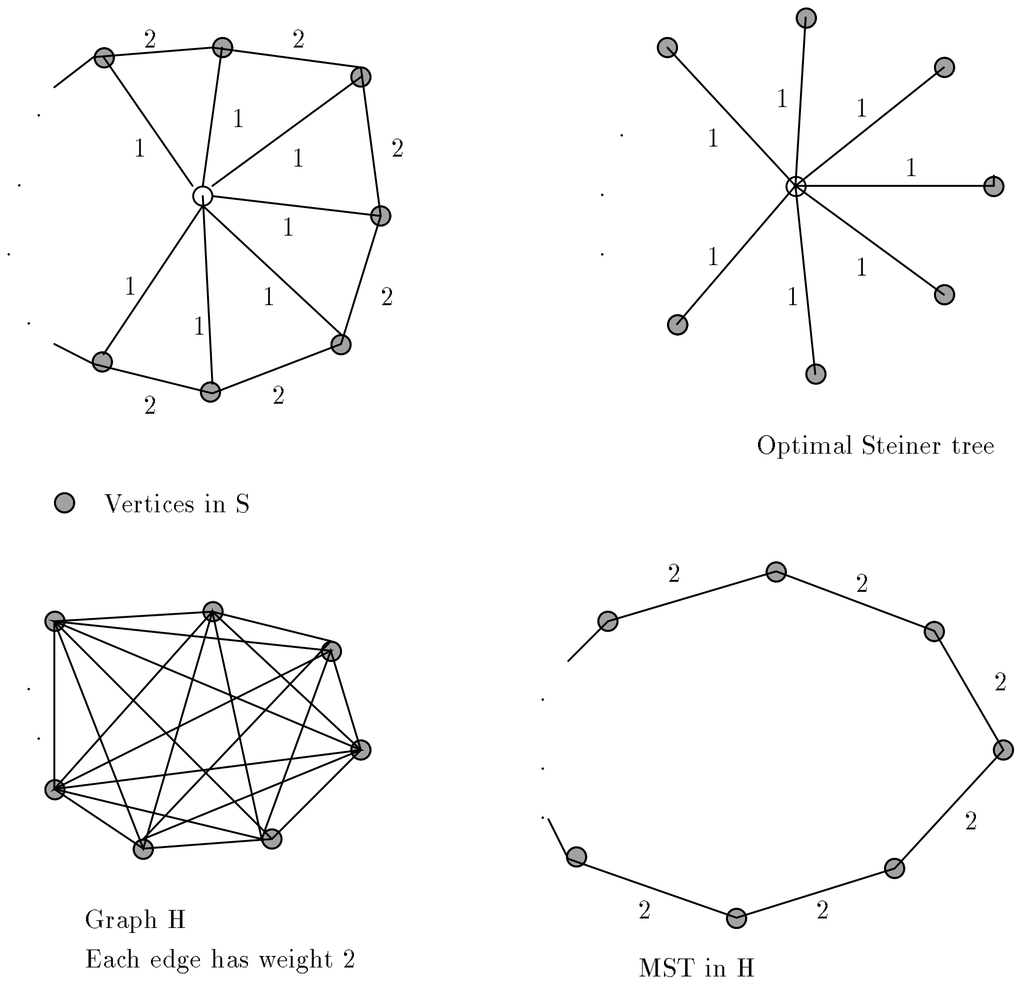


Figure 31: Worst Case ratio is achieved here

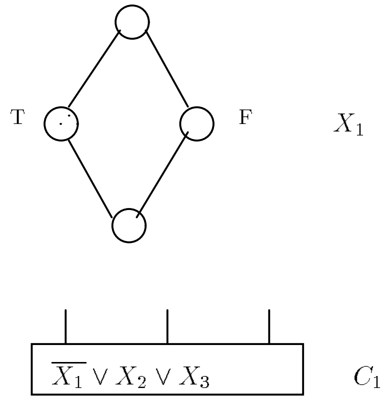
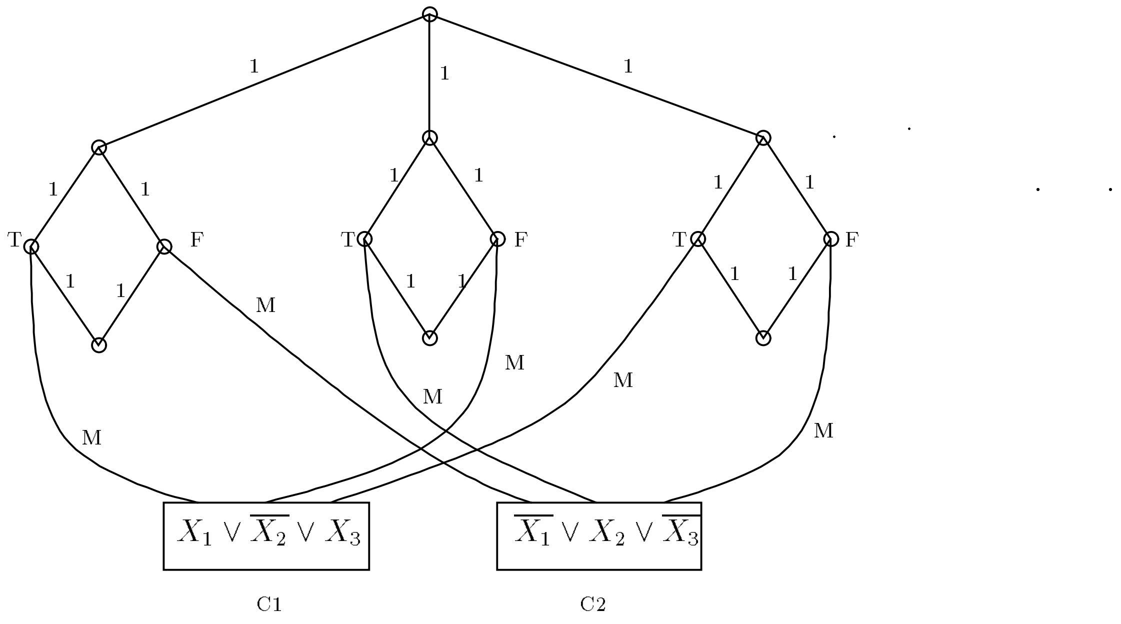


Figure 32: Gadget for a variable



1
1
1
1
1
1
1
1
1
T
F
T
F
T
F
1
1
1
1
1
1
M
M
M
M
M
M
$X_1 \vee \overline{X_2} \vee X_3$
$\overline{X_1} \vee X_2 \vee \overline{X_3}$
C1
C2

Figure 33: Reduction from 3-SAT.

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 connecting nodes corresponding to different variables.
- It must contain exactly 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 651 Advanced Algorithms	Lecture 27
Lecturer: Samir Khuller	Th. May 4, 1995

Original notes by Chung-Yeung Lee and Gisli Hjaltason.

27 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.

27.1 First-Fit

The strategy for First-Fit is that when packing an item, we shall put it into the lowest number bin that it will fit in. We start a new bin only when the item cannot fit into any existing non-empty bins. We shall give a simple analysis that shows that $First - Fit(I) \leq 2OPT(I)$. The folllowing more general result is known:

Theorem 27.1 *$First - Fit(I) \leq 1.7OPT(I) + C$ and the ratio is best possible (by First-Fit).*

The proof is complicated and therefore omitted here. Instead we shall prove that $First - Fit(I) \leq 2OPT(I)$.
Proof:
The main observation is that at most 1 bin is less than half of its capacity. In fact, the contents of this bin and any other bin add up to at least 1 (else we would have packed them together). The remaining $k-2$ bins are all at least half full. Therefore, if c_i denotes the contents in bin i and k is the no of bins used, we have

$$\sum_{i=1}^{k} c_i \geq \frac{1}{2}(k-2) + 1 \geq k/2.$$

Hence,

$$OPT(I) \geq \sum_{i=1}^{k} c_i \geq k/2.$$

Therefore $2OPT(I) \geq First - Fit(I)$. □

27.2 First-Fit Decreasing

A variant of First-fit is the First-Fit Decreasing heristics. Here, we first sort all the items in decreasing order of size and then apply the First-Fit algorithm.

Theorem 27.2 *(1973) FFD(I) $\leq$ 11/9 OPT(I).*

Remarks:

1. The known proof is very long and therefore is omitted.
2. The following instance shows that first fit decreasing is better than first fit. Consider the case where we have
    - $6m$ pieces of A, each of size $1/7 + \epsilon$.

- $6m$ pieces of B, each of size $1/3 + \epsilon$.
- $6m$ pieces of C, each of size $1/2 + \epsilon$.

First Fit will require $10m$ bins while First fit Decreasing requires $6m$ bins only. Note that the ratio is 5/3. This also shows that First-Fit does as badly as a factor of 5/3. (There are other examples to show that actually it does as badly as 1.7.)

27.3 Approximate Schemes for bin-packing problems

In the 1980's, two approximate schemes were proposed [22, 12]. They are

1. (Vega and Lueker) $\forall \epsilon > 0$, there exists an Algorithm A_ϵ such that

$$A_\epsilon(I) \le (1+\epsilon)OPT(I) + 1$$

where A_ϵ runs in time polynomial in n, but exponential in $1/\epsilon$ (n=total no. of items).

2. (Karmarkar and Karp) $\forall \epsilon > 0$, there exists an Algorithm A_ϵ such that

$$A_\epsilon(I) \le OPT(I) + O(lg^2(OPT(I))$$

where A_ϵ runs in time polynomial in n and $1/\epsilon$ (n=total no. of items). They also guarantee that $A_\epsilon(I) \le (1+\epsilon)OPT(I) + O(\frac{1}{\epsilon^2})$.

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.

27.3.1 Restricted Bin Packing

We consider the following restricted version of bin packing problem (RBP). We require that

1. Each item has size $\ge \delta$.
2. The size of the items takes only one of the m distinct values $v_1, v_2, ..., v_m$. That is we have n_i items of size v_i $(1 \le i \le m)$, 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 27.3 *Let J be the instance of RBP 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 no. of items with size $< \delta$ is large enough (otherwise I needs only β bins) so that we use β' bins.

$$S \ge (1-\delta)(\beta' - 1)$$

$$\beta' \le 1 + \frac{S}{1-\delta}$$

$$\beta' \le 1 + \frac{OPT(I)}{1-\delta}$$

$$\beta' \le 1 + (1+2\delta)OPT(I)$$

as $(1-\delta)^{-1} \leq 1 + 2\delta$ for small δ. □

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

$$k = \lfloor \frac{\epsilon^2 n'}{2} \rfloor.$$

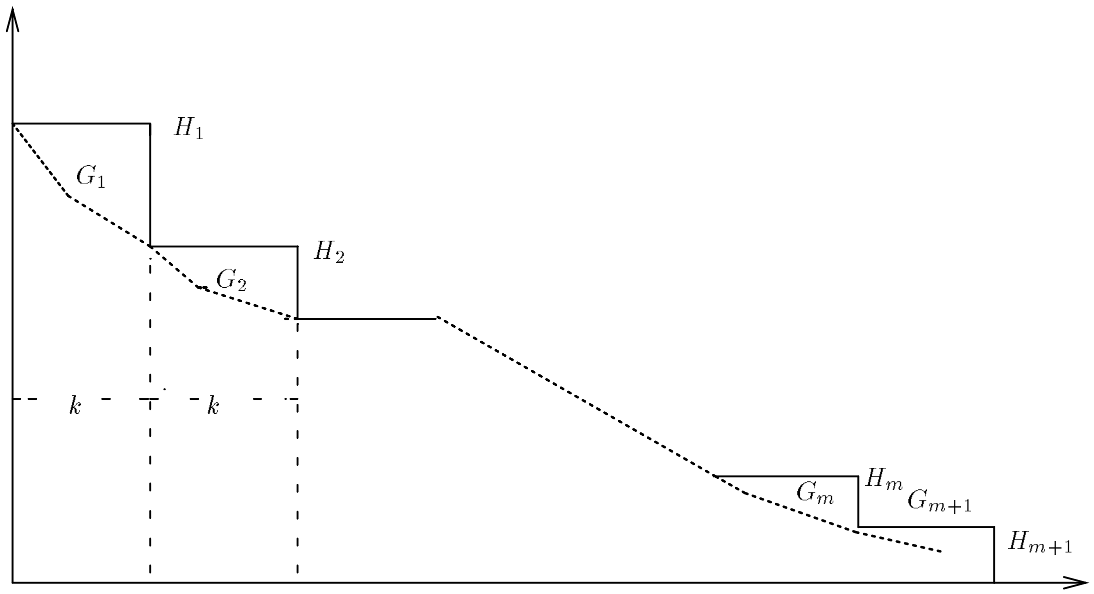
Then, we have m+1 groups $G_1, .., G_{m+1}$, where

$$m = \lfloor \frac{n'}{k} \rfloor.$$

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 illustrates the ideas:



Grouping Scheme for RBP

Figure 34: Grouping scheme

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 (J_{low}) is an instance of a Restricted Bin Packing Problem we can solve it optimally, and then add the items in G_1 in at most k extra bins. Directly from this inequality, and using the definition of k, we have

$$Soln(J) \leq \text{OPT}(J_{\text{low}}) + k \leq \text{OPT}(J) + k \leq \text{OPT}(J) + \frac{\epsilon^2 n'}{2}.$$

Choosing $\delta = \epsilon/2$, we get that

$$\mathrm{OPT}(J) \geq \sum_{i=1}^{n'} s_i \geq n'\frac{\epsilon}{2},$$

so we have

$$\mathrm{OPT}(J) + \frac{\epsilon^2 n'}{2} \leq \mathrm{OPT}(J) + \epsilon\mathrm{OPT}(J) = (1+\epsilon)\mathrm{OPT}(J).$$

By applying Theorem 27.3, using $\beta \leq (1+\epsilon)\mathrm{OPT}(J)$ and the fact that $2\delta = \epsilon$, we know that the number of bins needed for the items of I is at most

$$\max\{(1+\epsilon)\mathrm{OPT}(J), (1+\epsilon)\mathrm{OPT}(I)+1\} \leq (1+\epsilon)\mathrm{OPT}(I)+1.$$

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 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 different bin types. A solution to the RBP can now be stated as having x_i bins of type $T^{(i)}$. 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 27.4 *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 RBP(δ, n') by getting rid of all elements in I smaller than $\delta = \epsilon/2$.

Step 2: Obtain J_{low} from J, using the parameters k and m established earlier.

Step 3: Find an optimal packing for J_{low} by solving the corresponding integer linear programming problem.

Step 4: Pack the k items in G_1 using at most k bins.

Step 5: Pack the remaining items of J as the corresponding (larger) items of J_{low} were packed in step 3.

Step 6: 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 $\frac{1}{\epsilon}$.

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).