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

October 31, 1994

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 1993. 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 (now not so new) 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 notes are in a preliminary form, and were typed in by graduate students taking the course, as well as by yours truly (when I could not twist any student's arm into typing the notes!). 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 72 hour take-home final examination. There was no lecture on the day of the midterm. No scribes were done for the last 2 lectures. The topics for the last two lectures was "Computational Geometry" (convex hulls, closest-pair and point location were covered). These are covered very well in the book by Preparata and Shamos.

Some papers that I thought were very relevant to the topics studied in the class, were covered in a separate Algorithms Reading Group that met for pizza, mexican food and algorithms during the semester. The papers covered in this reading group included: Seidel's all pairs shortest paths algorithm, Seidel's fixed dimension linear programming algorithm, Alon-Seymour-Thomas's proof of the separator theorem for graph-minors, Yellin's work on data structures for set operations, Galil's constant time parallel string matching algorithm, Eppstein-Galil-Italiano-Nissenzweig's work on dynamic graph algorithms, Mehlhorn-Raman-Uhrig's work on lower bounds for set operations, Fellows and Langston's work on graph minors, Luks's work on graph isomorphism, Berkman-Vishkin's work on lowest common ancestors, Callahan-Kosaraju's work on well-separated pair decomposition. I am grateful to Paul Callahan, Bill Gasarch, Simon Hawkin, Dave Mount, Rajeev Raman, Bill Regli and Suleyman Sahinalp for presenting papers in this seminar.

I have editted these notes myself, and many of them have also been proof-read by Bill Gasarch to whom I am very grateful. Bill sat through most of the lectures, and also was the source of excitement during the rather dull lectures. His questions were usually penetrating and kept the lecturer on his toes. I would also like to thank both the Bill's (Gasarch and Pugh) who gave lectures on graph minors and skip-lists respectively. I welcome comments of all kinds, together with correction of errors in these notes. I am most grateful to the students who typed these notes.

Contents

CMSC 651 Advanced Algorithms	Lecture 1
Lecturer: Samir Khuller	Tu. Jan. 19, 1993

Notes by Hsiwei Yu.

1 Overview of Course

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. Towards the end of the semester (if time is available) we will study some special topics, such as randomization and parallel algorithms.

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 $= < b_k b_{k-1} \ldots b_1 >$. Observe that the least significant bit b_1, changes in every step. Bit b_2 however, changes in every alternate step. Bit b_3 changes every 4^{th} step, and so on. Thus the total number of bit operations done are:

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

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

CMSC 651 Advanced Algorithms	Lecture 2
Lecturer: Samir Khuller	Th. Jan. 21, 1993

Notes by Mark Carson.

2 Splay Trees

Splay trees are a powerful data structure, 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.

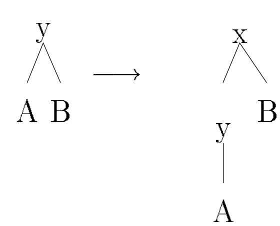
The first of these needs a slight addition:

1a. If k is not found, grab the largest node with key less than k instead. (Then splay this to the top.)

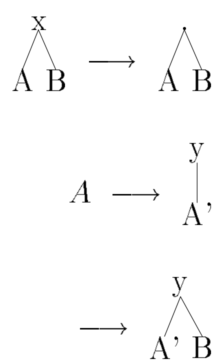
2.1 Use of Splay Operations

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

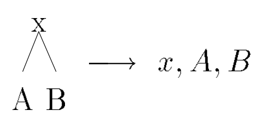
1. $Access(x)$ - Simply splay to bring it to the top, so it becomes the root.
2. $Insert(x)$ - Run $splay(x)$ on the tree to bring y, the largest element less than x, to the top. The insert is then trivial:



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:

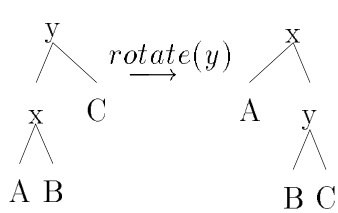
1. Find the item (time dependent on depth of item).

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*:

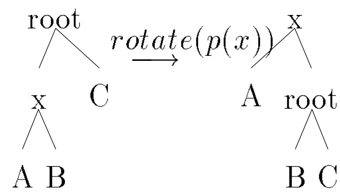


Clearly a rotation can be done in $O(1)$ time. [Note there are both left and right rotations, which are in fact inverses of each other. The sketch above depicts a right rotation going forward and a left rotation going backward. Hence to bring up a node, we do a right rotation if it is a left child, and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

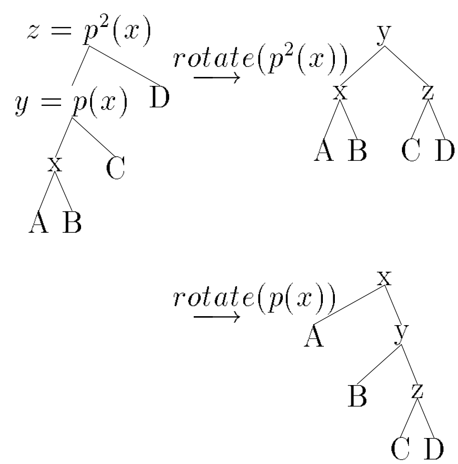
A splay is then done with a (carefully-selected) series of rotations. Let $p(x)$ be the parent of a node x. Here is the splay algorithm:

Splay Algorithm:

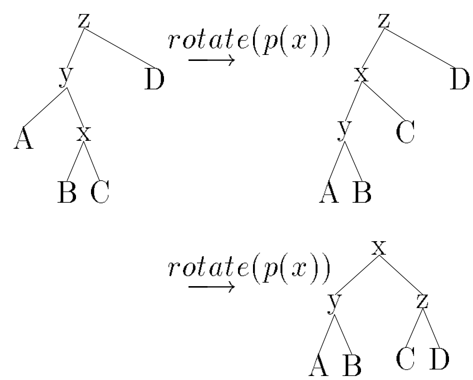
> **while** $x \neq root$ **do**
>
> > **if** $p(x) = root$ **then** $rotate(p(x))$



> > **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.

2.3 Amortized Time for Splay

Theorem 2.1 *The amortized time of a splay operation is $O(\log n)$.*

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

Definition 2.2 *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*

Let s be the splay tree.

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

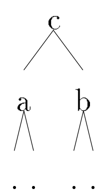
Thus we have:

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

Clearly, the better balanced the tree is, the lower the potential Φ is.
We will need the following lemmas to bound changes in Φ.

Lemma 2.3 *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 $r(c) > 1+min(r(a), r(b))$.



$\square$

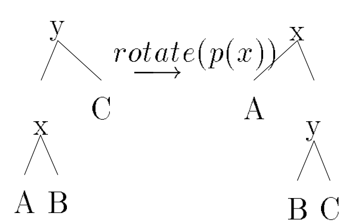
We apply this to

Lemma 2.4 (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

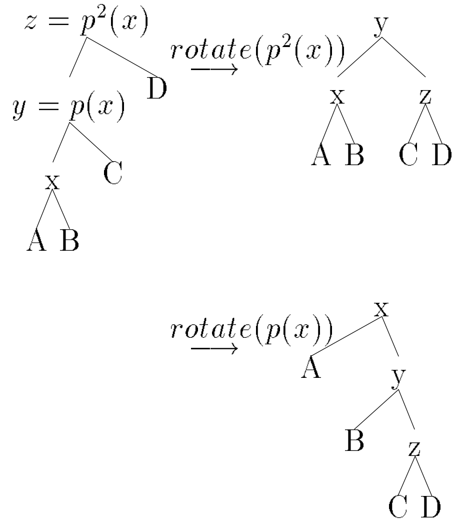


so that clearly $r'(x) = r(y)$. But then since only x and y change rank in s',

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

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

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

or

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

In the first case, since

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

clearly

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

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

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

□

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

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

where at is the amortized time, rt the real time. In this case, $rt = 1$, since a splay step can be done in constant time.

[Note: Here we have scaled the time factor to say a splay step takes one time unit. If instead we say a splay takes c time units for some constant c, we change the potential function Φ to be

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

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

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

For the second case, we have

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

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

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

But the latter series telescopes, so [tossing out the extra 1],

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

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

$$at < 3 \log n$$

as desired.

2.4 Additional notes

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

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

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

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

CMSC 651 Advanced Algorithms	Lecture 3
Lecturer: Samir Khuller	Tu. Jan. 26, 1993

Notes by King-Ip Lin

3 Heaps

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 3.1 (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.

3.1 Binomial heaps

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

Definition 3.2 (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

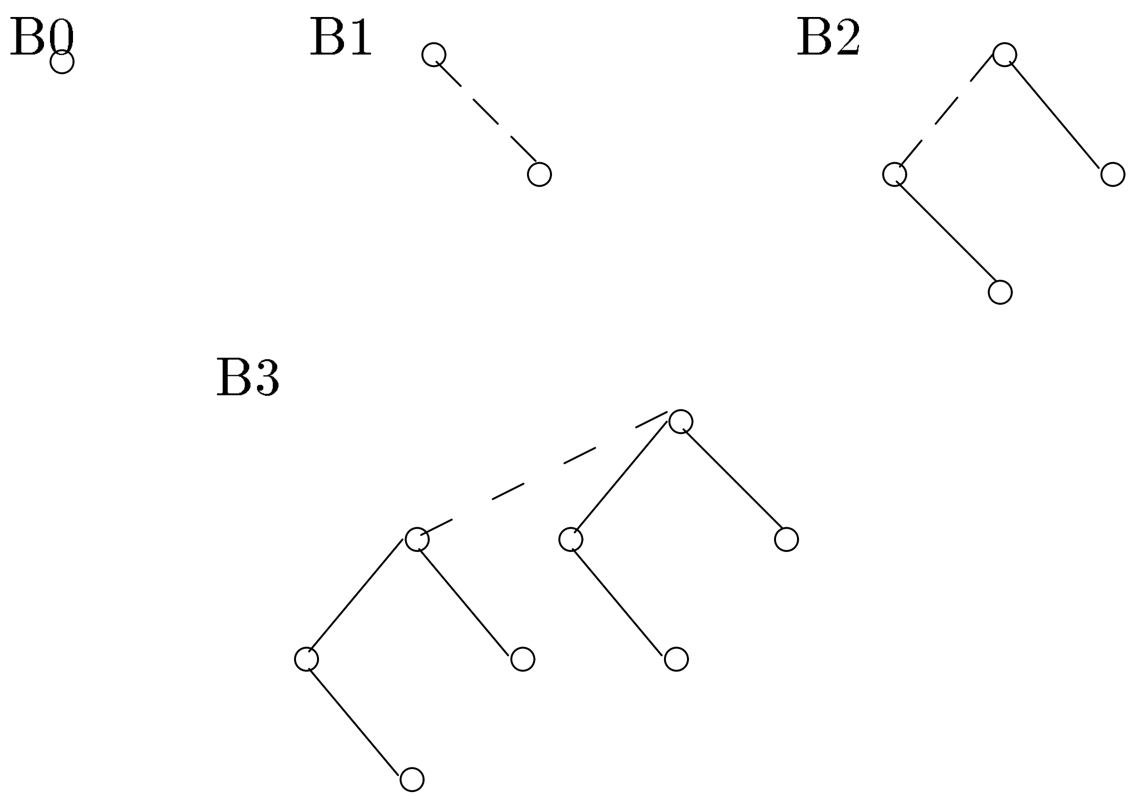


Figure 1: Examples of binomial heaps

- 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 3.3 (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 : $O(\log n)$

3.2 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:

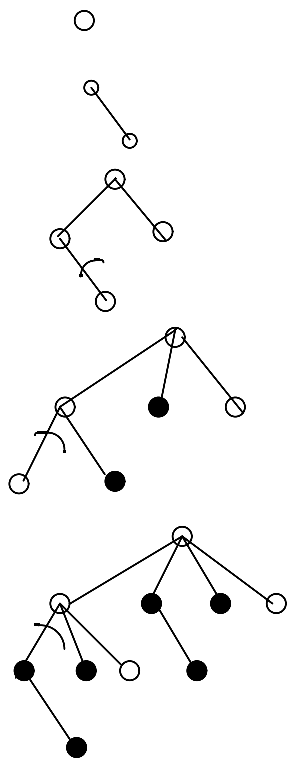
- whenever a node is being cut
  - mark the parent of the cut node in the original tree
  - put \$2 on that node
- when a second child of that node is lost (by that time that node will have \$4), recursively cut that node from its parent, use the \$4 to pay for it:
  - \$1 for the cut
  - \$1 for new 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 2: Minimum size B_i' trees

CMSC 651 Advanced Algorithms	Lecture 4
Lecturer: Samir Khuller	Thu. Jan. 28, 1993

Notes by Patchanee Ittarat.

4 F-heap

4.1 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,

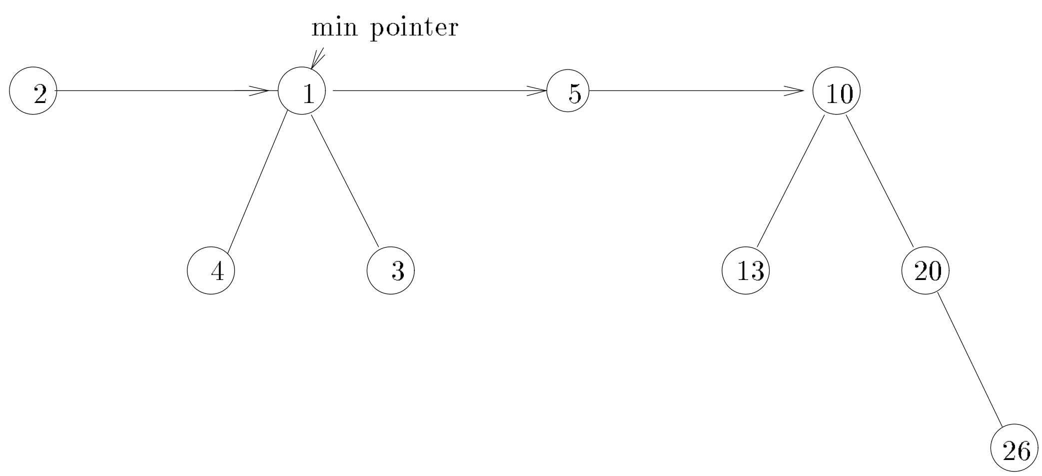


Figure 3: Example of F-heap

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.

4.2 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.

Example: How does mark strategy work?

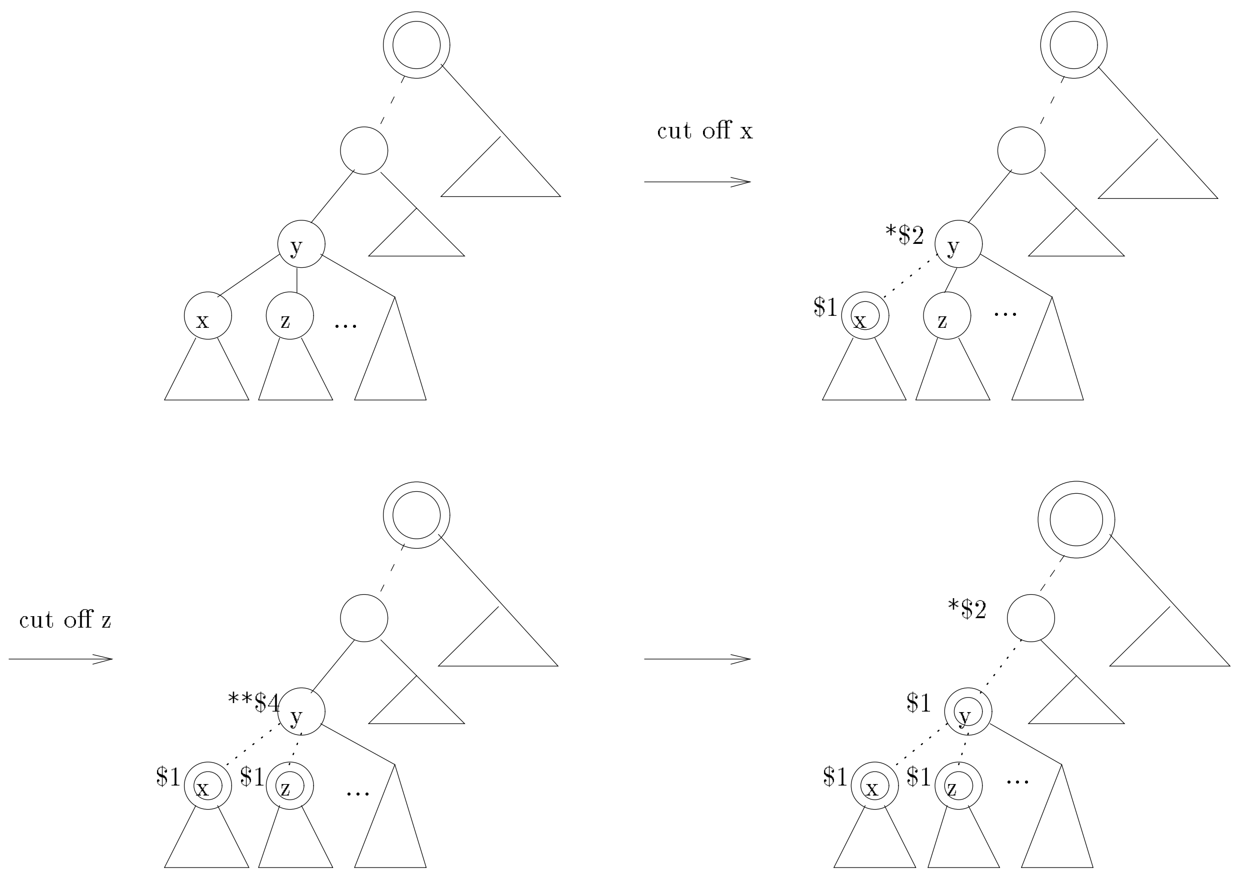


Figure 4: Marking Strategy

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)?

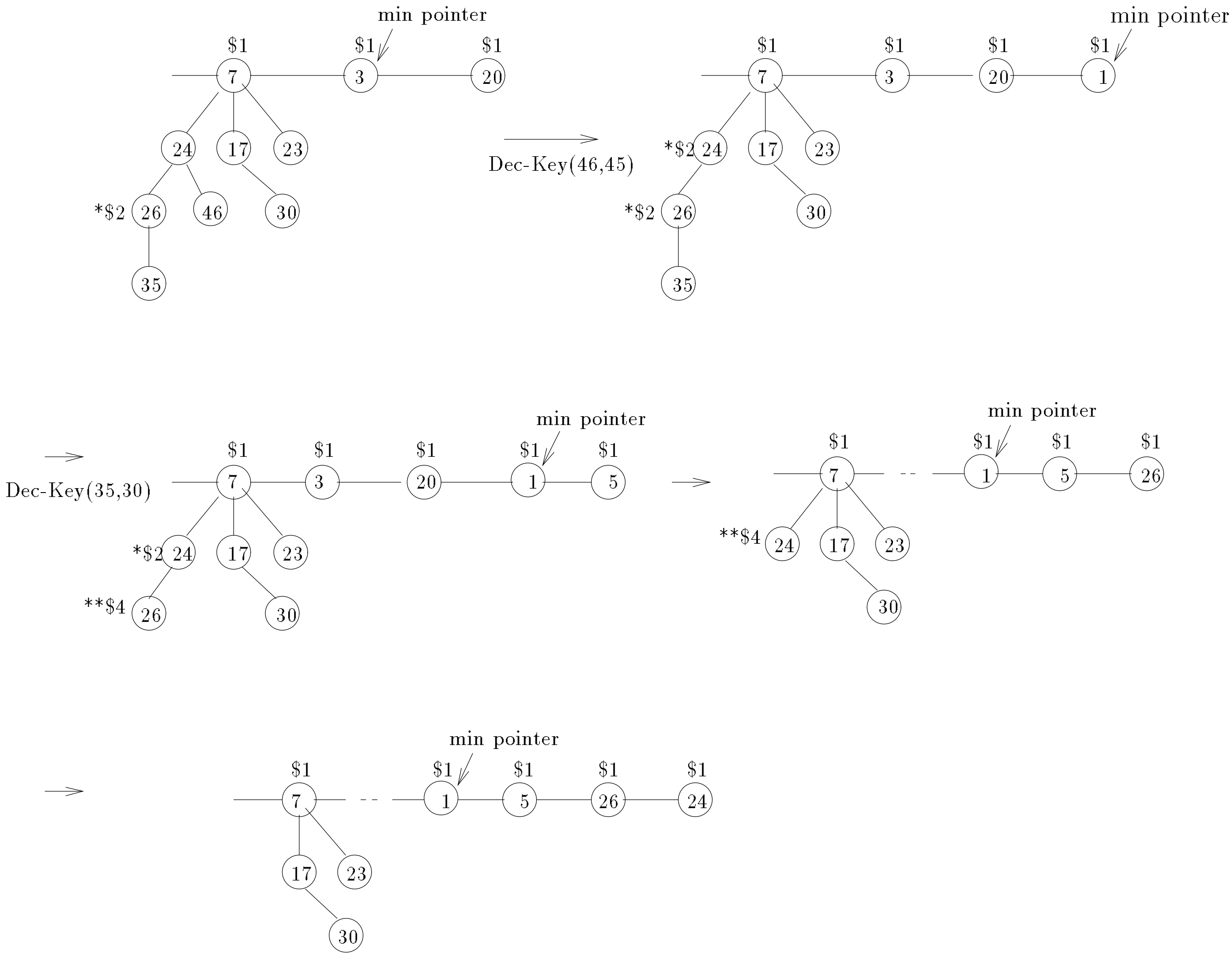


Figure 5: Example

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

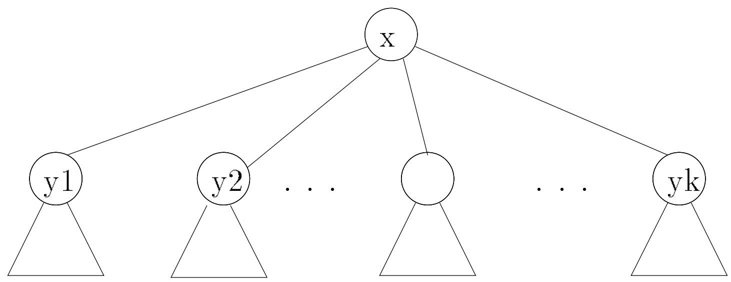


Figure 6: Example

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

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

These numbers are called Fibonacci numbers.

Property 4.1 $F_{k+2} = 1 + \sum_{i=1}^{k} F_i$

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

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

$\square$

Property 4.2 $F_{k+2} \geq \phi^k$

Proof:
[By induction]

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

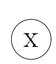
Assume $F_{k+2} \geq \phi^k$ for all $k < n$

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

$\square$

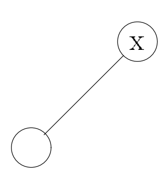
Theorem 4.1 *x is a root of any subtree. $size(x) \geq F_{k+2} \geq \phi^k$ where k is a degree of x*

Proof:
[By induction]
$k = 0$:



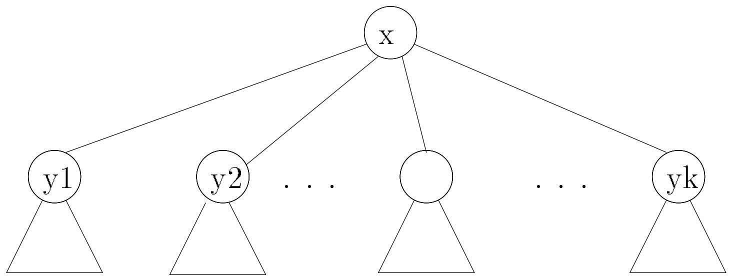
$size(x) = 1 \geq 1 \geq 1$

$k = 1$:



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

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 $\log_\phi size(x)$. $\square$

5 Maintaining Disjoint Set's

Another data structure that is useful in Kruskal's algorithm is to maintain disjoint sets. (It will be useful in solving other problems as well, such as in homeworks and exams!)

5.1 Disjoint set operations:

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

The name of a set is given by its smallest item.

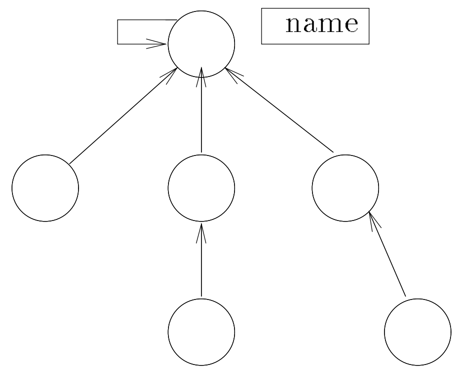


Figure 7: Data Structure

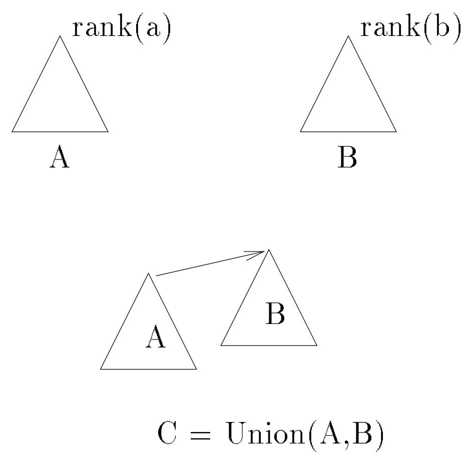


Figure 8: Union

5.2 Data structure:

- tree. See figure 5.

Find - worst case cost is $O(n)$.
Union - worst case cost is $O(1)$.
For m Find operations, total time is $O(m \cdot n)$.

To improve total time:

1. Balance a tree in some sense.

2. Make a tree shallower.

3. Hook the smaller tree to the bigger one.

Why will these improve a tree?

- Since the amount of work depends on the height of a tree.

Let rank be an upper bound on the height of a tree.

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

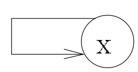


Figure 9: Rank 0 node

Union(A,B) - see figure 6.

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

5.3 Union by rank

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

Property 5.1 *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, now let's consider a Union operation in figure 6.

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 Find operation, we can make a tree shallower by path compression method.

5.4 Union by rank and path compression

- n Union's and m Find's cost total time in $O(m \log^* n)$.

where $\log^* n = \{min(i) | \log^{(i)} n \leq 1\}$, for example,

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

Theorem 5.1 *m Find's cost $O(m + n \log n)$. If $m > n \log n$, then this is optimal.*

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.
2. $Rank(p(x))$ is non-decreasing.
3. $Rank(p(x)) > rank(x)$.

Notes by Matos Gilberto.

5.5 Better Upper Bounds on the Disjoint-Set Union Operations

Three statements hold for this data structure at all times ($p(x)$ is the parent of x):

- Rank of x goes up as long as x is a root.
- Rank of $p(x)$ is bigger than rank of x.
- Rank of $p(x)$ is always nondecreasing

Also rank of any root is lesser than or equal $\log_2 n$, where n is the number of its descendants.

We want to prove that for a sequence of n UNION and m FIND operations the upper bound on the total execution time will be $O(m + n\log_2 n)$

The UNION operation is done by rank of the root, and the FIND operation performs the path compression while the set is being identified.

Before the find operation is performed on some node, that node points to his parent in the tree, and following the pointers through subsequent parents leads to the root of the tree. Algorithm will pass the path twice, 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.

While the operation of finding a path to the node is being performed each of the nodes on the path except the root and its child will be issued a bill for the work that has been done on its path compression. The operation pays only for the work on the root and its child.

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 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

Total work for the n unions and m find operations will be $O(m)$ for the charge for the find operations, and the sum of bills issued to all nodes will be upper bounded by $n\log_2 n$

Lemma 5.2 *There are at most $\frac{n}{2^r}$ nodes of rank r.*

Proof:

When the node gets rank r it has 2^r descendants, and it is the root of a tree. After some union operation this node will no longer be root, and may start to loose some descendants, but its rank will not decrease. Assume that every descendant of a root node gets a stamp when the root increases its rank. Subsequent stamps that the nodes get can be only with higher values of the rank of the root, because rank of root can increase after some union, or the root will become child of a node with a higher rank. So for every node of rank r there are at least 2^r nodes with its stamp for rank r. □

We will try to prove there is an even tighter upper bound for the cost of m find operations. That function is $O(m \log^* n)$ where $\log^* n$ is the number of log's that are required to reduce n to 1. (This is a proof done by Hopcroft and Ullman.)

For this we introduce the fast growing function F which is defined as

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

Another necessary function is $G(n)$ where $G(n) = \min\{k \text{ such that} F(k) \geq n\}$. This function is equal to the $\log^* n$ function.

5.5.1 Concept of Blocks

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

- $B(0)$ 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 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 billings a node can expect to 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. 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 n unions and m finds, 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.

CMSC 651 Advanced Algorithms	Lecture 6
Lecturer: Samir Khuller	Thurs. Feb 4, 1993

Notes by Robert Bennet.

6 The Skip List Data Structure

Skip lists, like array and linked list structures, support the following dictionary operations:

- Search()
- Insert()
- Delete()

But unlike arrays (sorted) which support these operations in $O(\log_2 n)$, $O(n)$ and $O(n)$ time and linked lists which support these operations in $O(n)$, $O(1)$ (non-sorted) and $O(n)$ time respectively, skip lists supports them in $O(\log_2 n)$ randomized time. (The expected time for a single operation is $O(\log n)$.)

The structure of a skip list is that of a sorted linked list with "express lanes" that skip 2-ahead, 4-ahead until there is a lane that skips $\frac{n}{2}$ ahead.

The height of an element in the skip list is defined as the number of stops at that element there are. This height can be determined probabilistically as the number of consecutive "coin-flips" until a head is encountered.

Given this information, an algorithm to search for an element can be given:

Let startlevel $= \log_2 n$:

1. **Search**(x, L).
2. $p = L \rightarrow$ header.
3. for i=startlevel downto 1 do
4. while $(p \rightarrow next[i].key < x)$
5. do $p = p \rightarrow next[i]$ /* p is largest element $< x$ */

For insertion, we keep a pointer to source level i pointer that goes over or to the element being searched for; call this over$[i]$. It should be noted that these algorithms use multi-pointer data structures. The algorithm to insert into a skip list is as follows:

1. **Insert** (x, L)
2. $p = L \rightarrow header$

3. for i=startlevel downto 1 do
4.   while($p \rightarrow next[i].key < x$)
5.   do $p = p \rightarrow next[i]$
6.   over[i] = p
7. ℓ = random level()
8. e=new-node(ℓ)
9. for i=1 to ℓ do
10. insert e after over[i] at level i

For deleting from a skip list change lines $5 \dots 9$ to:

1. (5) $e = p- > next[\ell]$
2. (6) Verify $e- > key == x$
3. (7) $\ell = e- > level$

Then change "insert" in line 10 to "delete" and add a free(e) call.

With skip lists, we want to handle the average probabilistic performance on worst case input.

6.1 Analysis

The analysis must be done oblivious to the element position. We will also start our search path backwards; meaning that the element will only be entered from above. Let, $C(\ell)$ = cost to climb up ℓ levels in an infinite list + the number of elements at the top level.

Thus, $C(\ell) = 1 + \frac{1}{2}C(\ell) + \frac{1}{2}C(\ell - 1)$, or $C(\ell) = 2\ell$, where 2ℓ is the expected cost of moving.

Since the algorithm depends on random variables, the lists cannot be re- distributed, which makes lists "non-random".

6.2 Probabilistic Analysis

Let, $C(\ell)$ = number of coin flips needed to see heads. Assume this is true for $C(1..\ell)$ and we will prove for $C(\ell + 1)$, which turns out to be a binomial distribution. These bounds are exact for a search at the end of the list.

Notes by Wlod Glazek.

7 Minimum Spanning Trees and Shortest Paths

Given a graph $G = (V, E)$ and a weight function $w(u, v)$ we wish to find a spanning tree $T \subseteq E$ such that its total weight $\sum_{(u,v)\in T} w(u, v)$ is minimized. We call the problem of determining the tree T the minimum-spanning tree problem and the tree itself an MST.

We will present now two approaches to finding MST. The first is Prim's method and the second is Yao's method. Before we present Prim's algorithm, we briefly review Dijkstra's algorithm.

7.1 Dijkstra's algorithm

Recall that Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted graph $G = (V, E)$ for the case in which all edge weights are nonnegative. The algorithm maintains a Fibonacci heap H of all the nodes that are not currently in the Shortest Paths Tree (SPT) and a table $d[v]$ of estimated distances from the source s for each v. It repeatedly selects $v \in H$ with the minimum shortest path estimate, inserts it into SPT and updates shortest path estimates for all nodes adjacent to selected node v. Once a node v is in SPT, the estimate $d[v]$ contains its real shortest distance from the source s.

Dijkstra's algorithm

Step 1: for all $v \in V$ except source vertex s initialize $d[v] := \infty$, $d[s] := 0$.

Step 2: put all $v \in V$ in the Fibonacci heap H.

Step 3: while heap H is not empty do

Step 4: u:=EXTRACT-MIN(H). DELETE-MIN(H).

Step 5: for all $w \in Adj[u]$ do

Step 6: if $d[w] > d[u] + w(u, w)$ then DECREASE-KEY($w, d[w] - d[u] - w(u, w)$)

The running time of this algorithm is $O(m + n \log_2 n)$.

This is because we do $n - 1$ DELETE-MIN operations, that cost $\log n$ time each. We do only $O(m)$ DECREASE-KEY operations (for each vertex the work done is proportional to its degree).

Note: $O(m+n \log n)$ is *optimal* for any implementation of Dijkstra's algorithm (by reducing sorting to it). This is something worth thinking about. It is possible that one can do shortest paths faster than $O(n \log n)$ for graphs that have $O(n)$ edges. For example, Frederickson has shown that for planar graphs we can compute shortest paths in $O(n\sqrt{\log n})$ time. (This is highly non-trivial.) Can we extend this to any graph with $O(n)$ edges ? (This is still open.)

7.2 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. This satisfies the property of the MST. Tarjan describes generic red-blue rules that let us put edges in and out of the MST. All MST algorithms are essentially an application of the red-blue rules in a particular order. 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 first study Yao's algorithm, and then the more recent algorithm 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, and that is the best known (also uses F-heaps).

7.3 Yao's algorithm

Yao's algorithm starts with a collection of singleton vertex sets placed in a priority queue and repeatedly merges pairs of sets in a round-robin fashion until only one set is left. Specifically, each vertex is initially in its own set. 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 (say to vertex u), and merges itself with u. We remove both v and u from the queue and put the connected pair at the end of the queue. We continue doing this until all the vertices, initially in the queue, are processed. Notice that we now have a collection of connected components (a forest of trees) that are in the queue. Each 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.)

We can use the UNION-FIND structure for keeping track of connected components. This will give us a running time of $O(m \log^* n)$ per phase and with the bound on the number of phases this takes $O(m \log^* n \log n)$ time.

Homework: How do we implement a single phase in linear time ? Notice that we do not really need UNION-FIND to implement a phase !

The main thing that slows down the algorithm is the fact 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 breaking the adjacency list of each vertex $Adj(v)$ 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 $w(e) < w(e')$. 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. Then we recurse on each portion to break the sets, and we get 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, clearly takes $O(m \log k)$ time.

Let T be a set of edges in the final MST, VS be a collection of vertex sets that form connected components and ES be collection of edge sets incident to each connected component.

Yao's algorithm

Step 1: $T := \emptyset$,$VS := \emptyset$,$ES := \emptyset$,

Step 2: for each $v \in V$ do

Step 3: add v to VS

Step 4: add $E(v)$ to ES

Step 5: divide $E(v)$ into k groups E_v^1, E_v^2,..., E_v^k

Step 6: while $|VS| > 1$ do

Step 7: take a vertex set W from VS

Step 8: for each vertex $v \in W$ do

Step 9: FIND cheapest outgoing edge from v

Step 10: Among the edges pick cheapest edge (w, w') in $E(W)$

Step 11: Let $w' \in W'$

Step 12: Replace W and W' by UNION(W,W')

Step 13: Replace $E(W)$ and $E(W')$ by UNION($E(W)$,$E(W')$)

Step 14: add (w, w') to T

Step 15: output T

Now, the search for the cheapest outgoing edge is simpler. For every v we only scan through the pieces not scanned so far. We take the first encountered edge as the cheapest one. 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 $O(\frac{m}{k}\log^* n \log n) + O(m \log k)$. If we take $k = \log n$ then we get $O(m \log^* n) + O(m \log\log n)$, which is better than the cost of Prim's algorithm.

Notes by Annette Evangelisti.

8 Fredman-Tarjan MST Algorithm

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 $n-1$ times (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 in a round-robin fashion 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 (component). From the set of edges between two different trees we simply maintain the lowest weight edge and discard all other edges. A linear time implementation of CLEANUP will be described later.

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, again stopping only when the heap gets too large, or 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 now condense every tree into a single supervertex (basically by doing CLEANUP; this condensation is thus done implicitly) and begin a new pass 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 phase, we start with a collection of *old trees*. The pass connects these trees to form new larger trees that become the *old trees* for the next phase. We *unmark* all the trees, create an empty heap. We 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 empty the heap and mark the tree. The F-heap has the set of all trees that are adjacent to the current tree (tree to grow).

(See GROWTREES in Handout.)

Procedure GROWTREES

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)

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 merges with another tree. Assume there exist T trees at the start and m is the number of edges initially. Then

$$k = 2^{2m/T}$$

where k increases as the number of trees decreases. Notice that k is essentially 2^d, where d is the average degree of a vertex in the super-graph. But one phase is upperbounded by

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

This is so because we perform at most T DELETE-MIN operations (each one reduces the number of trees), and $\log k$ is the upperbound on a single heap operation. The time for initialization and CLEANUP etc is $O(m)$.

Consider the effect of a pass that begins with T trees and $m' \leq m$ edges (some edges may have been discarded). Each tree remaining after the pass has more than $k > 2^{\frac{2m}{T}}$ edges with at least one endpoint in T. If a tree stopped after colliding with the initially grown tree, then the merged tree has the property that it has at least k neighbors. 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 ! However, if we consider the multigraph formed by considering each tree as a single vertex we can argue that each tree (vertex) must have degree more than k. Thus we obtain $T'k \leq \sum_{T_i} d(T_i) = 2m'$. 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 ?

Let $\beta(m,n) = \min\{i | \log^{(i)} n \leq \frac{m}{n}\}$. It is clear that the number of rounds is at most $\beta(m,n)$. This gives us an algorithm with total running time $O(m\beta(m,n))$.

Procedure CLEANUP

1. Resets marks on all trees to false.
2. Recomputes tree[V] for all vertices V.
3. Removes all edges that go between vertices that are in the same tree. (i.e. remove (u,v) if tree[u] = tree[v]).
4. Of all edges between T_i and T_j, retain only the cheapest edge.

After each phase of growtrees cleanup keeps only the cheapest edge between two trees and deletes all edges that go between vertices in the same tree.

We number all trees consecutively from one. We sort all the edges lexicographically on the numbers of the trees containing their endpoints, using a two-pass radix sort. We then scan the sorted list saving only the appropriate edges. After CLEANUP, we construct a list for each old tree T of the edges with one endpoint in T. (Each edge is on two lists.) This can be done in $O(m)$ time.

Notes by Michael Tan

9 Matchings

In this lecture we will examine the problem of finding a maximum matching. We will do this by examining the particular case of finding a maximum matching in a bipartite graph.

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 $|M|$.

A **bipartite** graph $G = (U, V, E)$ has $E \subseteq U \times V$.

Aside: We can test if a given graph is bipartite in O($E|$) time. Here is how: Do BFS on the graph. Let every vertex discovered on an even level be in set U, and every vertex discovered on an odd level be in set V. As BFS runs, make sure that no "even" vertex has an edge to another "even" vertex, and that no "odd" vertex has an edge to another "odd" vertex.

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.

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:

9.1 Hall's Theorem

Theorem 9.1 (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. □

Before proceeding with the algorithm, we meed 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 $|M| + 1$.

9.2 Berge's Theorem

Theorem 9.2 (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). □

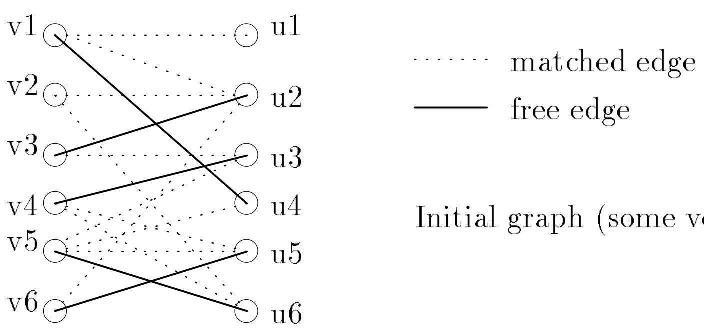
Armed with this theorem, we can outline a primitive algorithm to solve the maximum matching problem.

Simple Matching Algorithm [Edmonds]:

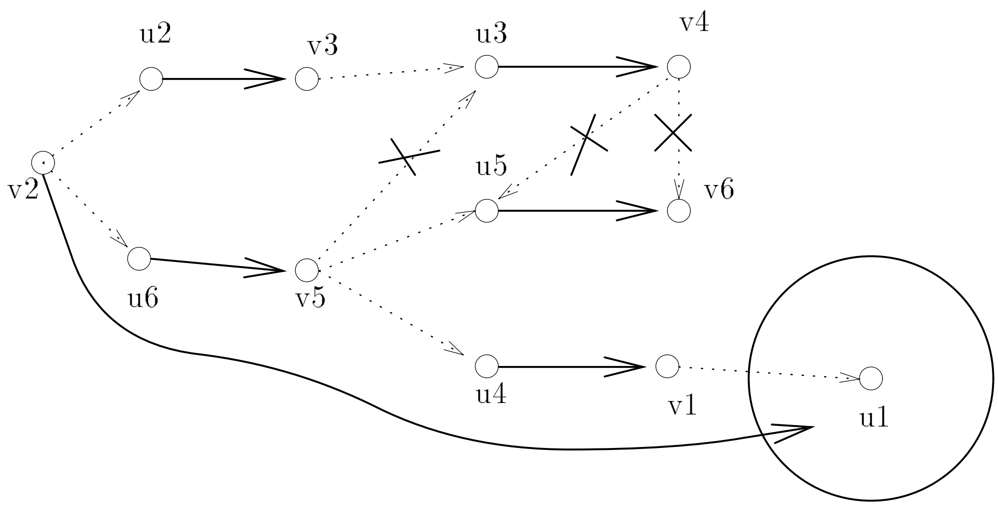
Step 1: Start with $M = \emptyset$.

Step 2: Search for an augmenting path.

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



Initial graph (some vertices already matched)



BFS tree used to find an augmenting path from v2 to u1

Figure 10: Sample execution of Simple Matching Algorithm

Step 4: Go to 2.

Here is an example:

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 following lecture, we will learn the Hopcroft-Karp O($\sqrt{|V|}|E|$) algorithm for maximum matching on a bipartite graph. In 1981, Micali-Vazirani extended this algorithm to general graphs (keeping the same running time)!

Notes by Samir Khuller.

10 Hopcroft-Karp Matching Algorithm

We present the Hopcroft-Karp matching algorithm along with a proof of Hall's theorem in this lecture. (This theorem can be proven by induction as well, unfortunately that does not yield an efficient algorithm.)

Theorem 10.1 (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. 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 $|N(S)| < |S|$. □

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 10.2 (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.)*

The main idea behind the Hopcroft-Karp algorithm is to augment along a disjoint set of shortest augmenting paths *simultaneously*. (For example, if the shortest augmenting path has length k then in a single phase they obtain a *maximal* set S of 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. This can be done in linear time. 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 now prove the following lemmas.

Lemma 10.3 *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 like Edmonds algorithm that simply searches for an augmenting path). 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 some 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 known as *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. 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. We then 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 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).

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. $\square$

From the following lemma it is clear that the augmenting paths in phase i will be strictly longer in phase $(i+1)$. After phase i, any augmenting path that has the same length as the paths in phase i cannot intersect the paths in phase i – this gives a contradiction to the maximality of the paths found in phase i.

Lemma 10.4 *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*

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

Proof:

Let $N = (M \oplus P) \oplus P'$. Thus $N \oplus M = P \oplus P'$. Consider $N \oplus M$. This is a collection of cycles and paths. 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.

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

This yields the required inequality. □

Theorem 10.5 *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 $\sqrt{n} + 1$. (The paths are always odd in length and always increase 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 path we have:

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

Thus k is upper bounded by $\sqrt{n}$. 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. □

Notes by Joseph Naft.

11 Assignment Problem

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

An algorithm which solves the Assignment Problem is due to Kuhn and Munkres, and is called the Hungarian Method in honor of Egervary. 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.) For example, there exists a feasible vertex labeling such that

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

and

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

We define the **Equality Subgraph**, $G_l(X, Y, (x_i, y_j))$, 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 11.1 *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,

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

□

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 lables. We also ensure that edges from our current matching do not leave the equality subgraph. After adding edges to the equality subgraphs, 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.

The Kuhn-Munkres Algorithm:

Step 1: Build an Equality Subgraph, G_l.

Step 2: Find a maximum matching in G_l (not necessarily a perfect matching). If it is a perfect matching, according to the theorem above, we are done.

Step 3: Let S = the set of free nodes in X
T = all nodes in Y encountered in the search for an augmenting path.

Step 4: Grow Hungarian trees from S, revise the labeling, l, *adding edges to* G_l to find a perfect matching, and adding vertices to S and T as they are encountered in the search, as described below. Return to Step 2.

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 increase the size of the matching which is already a maximum matching. 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. As the labels in S are decreased, edges 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 these nodes to S and T, which yields a larger Hungarian Tree. If the node from $\overline{T}$ node 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).

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 12
Lecturer: Samir Khuller	Th. Feb. 25, 1993

Notes by Sibel Adali.

12 Stable Marriages Problem.

We're given a complete bipartite graph consisting of n boys and n girls. Each girl (boy) makes a preference list of the boys (girls), such that the first element in the list shows the boy (girl) she (he) likes the most and the last element is the last boy (girl) in the world she (he) would like to be with. The preference lists of the girls and the boys are required to contain all the individuals from the opposite sex. A perfect matching on this graph is called a marriage and the matched pairs are called couples. We can formalize this problem as follows: given a complete bipartite graph $G = (G \cup B, E)$ and preference lists $b_i = [\sigma(b^i, 1), \sigma(b^i, 2), \ldots, \sigma(b^i), n)]$, $g_j = [\sigma(g^j, 1), \sigma(g^j, 2), \ldots, \sigma(g^j, n)]$ for all $b_i \in B, g_j \in G$, find a perfect matching that is not *unstable*.

A marriage M is called *unstable* iff the following is true: Let Jane and Paul, Mary and John be two couples in M. If Jane prefers John to Paul and John prefers Jane to Mary according to their preference lists. Then, this marriage is unstable.

Paul $\leftrightarrow$ Jane	John = [... Jane ... Mary ...]
John $\leftrightarrow$ Mary	Jane = [... John ... Paul ...]

In other words a marriage M is called unstable, iff there exists a pair that is not matched in M who prefer each other to their current partners.
The algorithm to find a stable marriage is given by Gale & Shapley and it works as follows:

Algorithm for finding a stable marriage

Step 1: Every girl proposes to the first boy in her preference list.

Step 2: Every boy who receives proposals checks his preference list and gets engaged to the girl having the highest preference among the ones who proposed to him.

Step 3: In the next round, all the girls who are not engaged propose to the next boy in their preference lists.

Step 4: If the boy is already engaged with another girl, he checks if he got a proposal from a girl with higher preference, if so (as most boys do (!)) he breaks his engagement with his former fiancée and he gets engaged to the girl who just proposed to him. In the next round, the girls who lost their fiancés in the competition, will propose to the boy that comes just after their lost fiancés in their preference lists.

Step 5: Last two steps are repeated until everybody is matched.

The above algorithm will always terminate: because, when a girl proposes to the last boy in her list all the other boys are engaged (to the other $n-1$ girls), so the boy who is not engaged has to accept her proposal.

Proposition 12.1 The marriage found by the above algorithm is *stable*.
Proof:

Assume by way of contradiction that the marriage is unstable, that is there is a pair John and Jane as explained above. Since John comes before Paul in Jane's preference list, Jane must have proposed to John before Paul. Since John rejected Jane, he must be engaged with a girl that comes before Jane in his preference list. John will break his engagement only if a girl that comes before his current partner proposes to him, therefore Mary must come before Jane in his list. Hence, the marriage is stable. $\square$

It is easy to see that the above algorithm runs in $O(n^2)$ steps.

13 Network Flow - Maximum Flow Problem

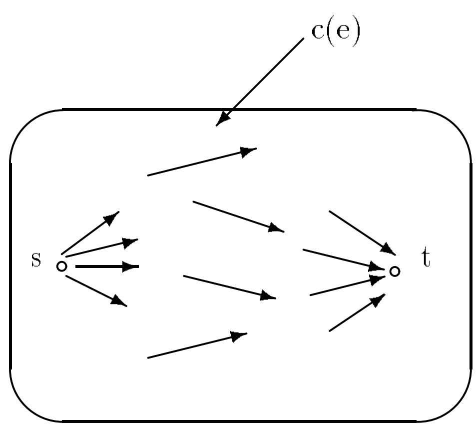
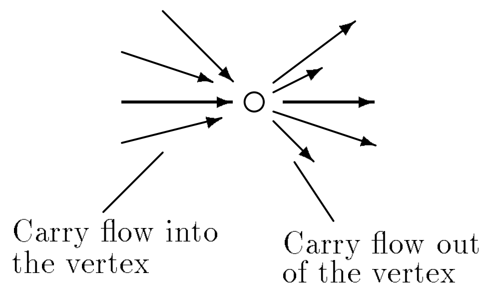


Figure 11: Network flow problem

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 $c : E \to \Re^+$, find the maximum flow from s to t.

Flow is a function $f : E \to \Re^+$ that has the following properties:

1. **(Skew Symmetry)** $f(u, v) = -f(v, u)$



Flow Conservation

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) $\Sigma_{v \in V} f(u,v)$

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 13.1 (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 $s \in A, t \in B$.

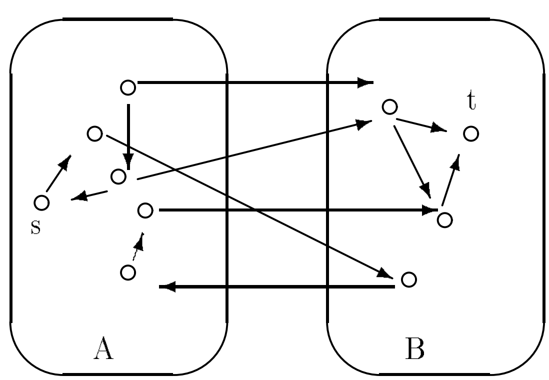


Figure 12: An (s,t) Cut

Definition 13.2 (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). Then, the capacity of the minimum capacity cut is an upper bound on the value of the maximum flow.

Definition 13.3 (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 13.4

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. $|f + f'| = |f| + |f'|$.
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.

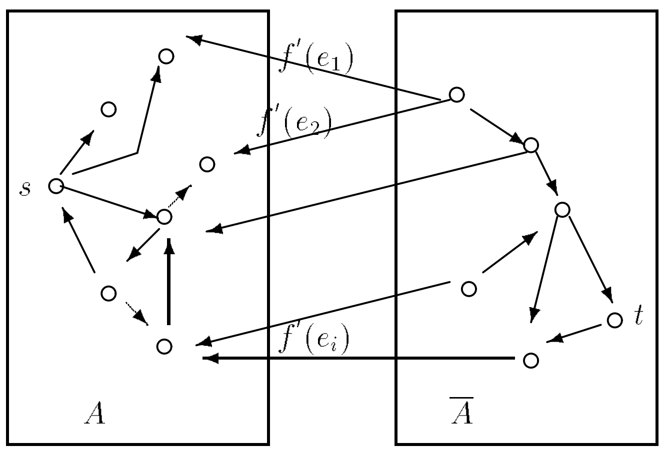


Figure 13: The Residual Graph G_f

Theorem 13.5 (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.

[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 $0 - (-6) = 6$.

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 given in figure 13. There is no directed path from s to t in G_f, since if there was this would be an augmenting path. Let $A = \{v | v$ is reachable from s 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. $\square$

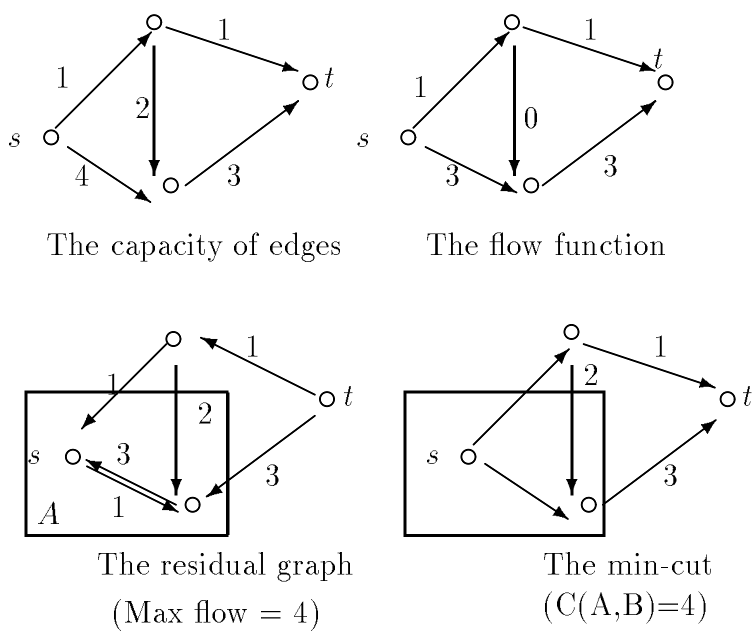


Figure 14: An example

CMSC 651 Advanced Algorithms	Lecture 13
Lecturer: Samir Khuller	March 2, 1993

Notes by Andrew Vakhutinsky.

14 The Max Flow Problem

Definition 14.1 (Flow Function) *Given a directed flow graph $G = (V, E, s, t, c)$ with source s, sink t and capacity function $c: E \rightarrow R^+$. Function $f: E \rightarrow R^+$ is called a flow if it has the following three properties:*

1. *skew symmetry* $f(u, v) = -f(v, u)$
2. *flow conservation* $\sum_{v \in V} f(u, v) = 0 \;\; \forall u \neq s, t$
3. *capacity constraint* $f(u, v) \leq c(u, v)$

Value of flow f is defined as $|f| \stackrel{\text{def}}{=} \sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, t)$

Max Flow Problem: maximize $|f|$

Definition 14.2 (Flow Cut) *In a given flow graph $G = (V, E, s, t, c)$, a cut(A, B) is a partition of the vertex set V such that $A \cap B = \emptyset$, $A \cup B = V$, $s \in A$, $t \in B$.*
Capacity of a cut is defined as $c(A, B) \stackrel{def}{=} \sum_{u \in A, v \in B} c(u, v)$

Definition 14.3 (Residual Graph) *A flow graph $G_f = (V, E, s, t, c')$ is called a residual graph with respect to flow graph G and flow function f if for all $e \in E$ $c'(e) = c(e) - f(e)$. In this case we write simply $c' = c - f$.*

Zero capacity edges are usually considered eliminated from G_f.

Lemma 14.4

1. *f' is a flow in G_f iff $f + f'$ is a flow in G.*
2. *f' is a max flow in G_f iff $f + f'$ is a max flow in G.*
3. $|f + f'| = |f| + |f'|$
4. *If f is a flow in G and f^* is the max flow in G then $f^* - f$ is the max flow in G_f.*

Definition 14.5 (Augmenting Path) *A path P from s to t in flow graph G is called augmenting if all its edges are not saturated, that is $\forall e \in P$ $f(e) < c(e)$. Residual capacity of an augmenting path P will be called a value* $\max_{e \in P}(c(e) - f(e))$

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.

14.1 Max Flow - Min Cut Theorem

Theorem 14.6 (Max Flow - Min Cut Theorem) *The Following three statements are equivalent.*

(a) *f is a max flow in G*

(b) *$\exists$ cut(A,B) such that $c(A,B) = |f|$*

(c) *there are no augmenting paths in G_f*

Proof:

(a)$\Rightarrow$(c) is trivial.

(b)$\Rightarrow$(a) is also trivial if one notices $|f| = f(A,B) \leq c(A,B)$ where $f(A,B) \stackrel{def}{=} \sum_{u\in A, v\in B} f(u,v)$

(c)$\Rightarrow$(b). Let $W = \{v | v \text{ is reacheable from } s \text{ in } G_f\}$ and consider a cut $(W, \overline{W})$ in G where $\overline{W} = V - W$. Then

$$\begin{aligned}
\forall (v,u) \in E \text{ and } v \in W, u \in \overline{W}\ f(u,v) &= c(u,v) \\
\forall (u,v) \in E \text{ and } u \in \overline{W}, v \in W\ \ f(u,v) &= 0
\end{aligned}$$

It implies $c(W,\overline{W}) = f(W,\overline{W}) = |f|$. □

A "Naive" Max Flow Algorithm:

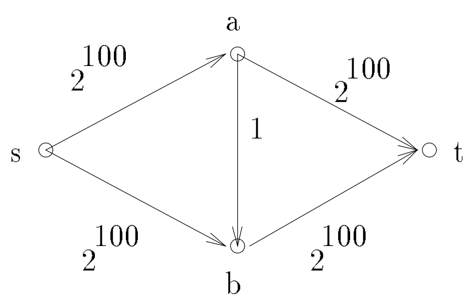
while (there is an augmenting path P in G) **do**
    pick up an augmenting path P;
    $c(P) \leftarrow \min_{e\in P} c(e)$;
    send flow amount $c(P)$ along P;
    update flow value $|f| \leftarrow |f| + c(P)$;

Analysis: On each iteration, sending some flow along an augmenting path, we increase the flow value. If all edge capacities are integral, the algorithm will terminate but its running time is $O(m\,|f^*|)$ in the worst case (where $|f^*|$ is the value of the max-flow).

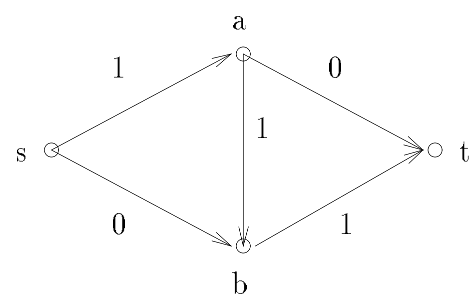
A worst case example. Consider a flow graph as shown on the Fig. 15. Using augmenting paths (s,a,b,t) and (s,b,a,t) alternatively at odd and even iterations respectively (fig.1(b-c)), it requires total $|f^*|$ iterations to construct the max flow.

If all capacities are not integral (rational), flow algorithm might never terminate.

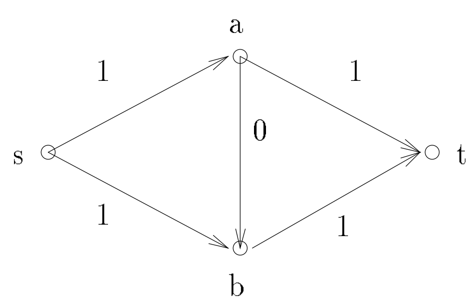
Example. Consider a graph on Fig. 16 where all edges except (a,d), (b,e) and (c,f) are unbounded (have comparatively large capacities) and $c(a,d) = 1$, $c(b,e) = R$ and $c(c,f) = R^2$. Value R is chosen such that $R = \frac{\sqrt{5}-1}{2}$ and, clearly, $R^{n+2} = R^n - R^{n+1}$. If augmenting paths are selected as shown on Fig. 16 by dotted lines, residual capacities of the edges (a,d), (b,e) and (c,f) will remain 0, R^{3k+1} and R^{3k+2} after every $(3k+1)$st iteration $(k = 0,1,2,\ldots)$. Thus, the algorithm will never terminate.



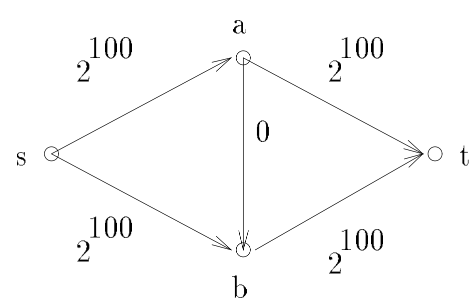
(a) Initial flow graph with capacities



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



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



(d) Flow in the graph after the final iteration

Figure 15: Worst Case Example

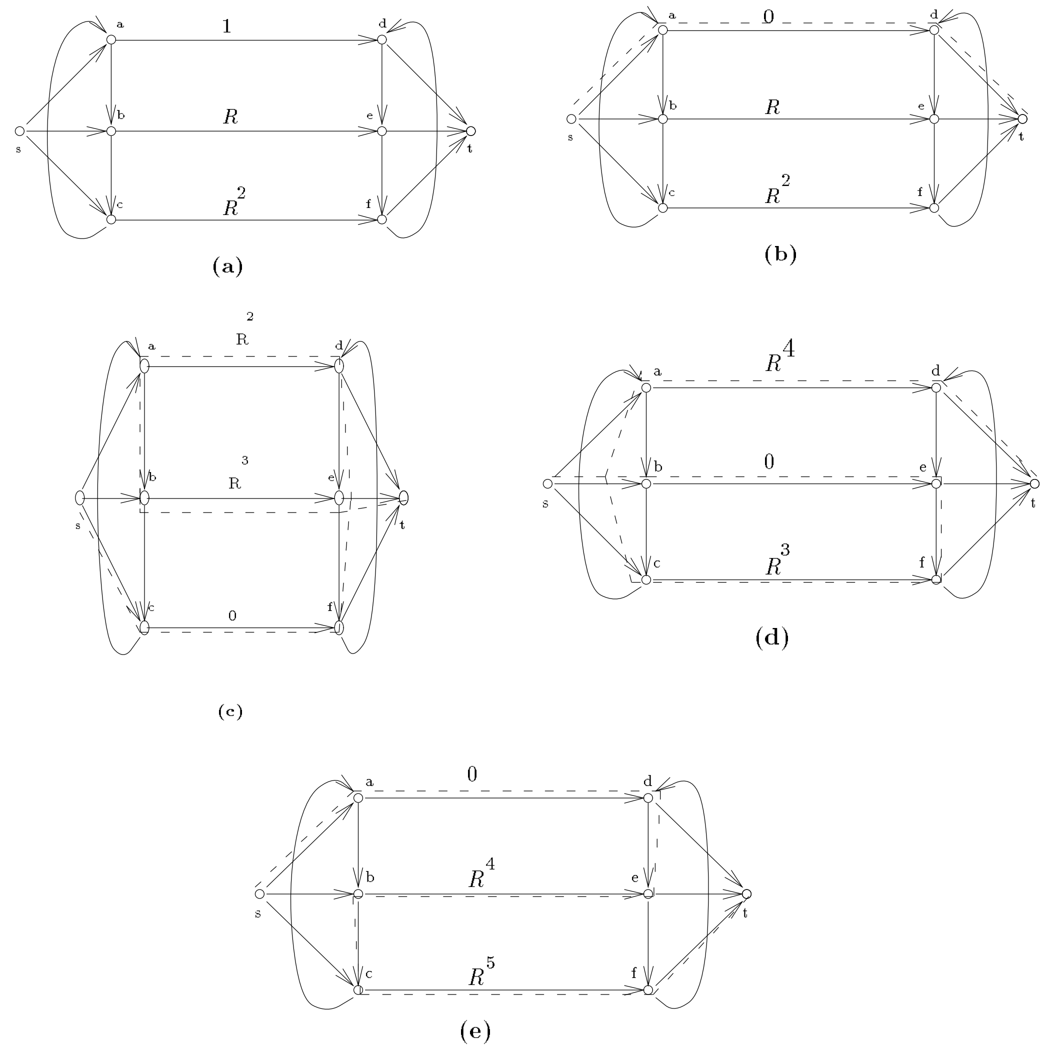


Figure 16: Non-terminating Example

14.2 Polynomial Time Algorithms (Edmonds-Karp)

Assumption: Capacities are integers.

1st Edmonds-Karp Algorithm:

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

Analysis: First, prove that if the max flow in a graph G is f^*, the highest capacity of an augmenting path is $\geq \frac{|f^*|}{m}$.

Consider a decomposition of the flow onto saturated flow paths passing each edge in the same direction. Such a decomposition does exist. Indeed, if there are two paths $P_1 = (P_1', (u,v), P_1'')$ and $P_2 = (P_2', (v,u), P_2'')$ passing the edge (u,v) in opposite directions with corresponding flow amounts $f_1(u,v) > f_2(v,u)$, then they can be replaced by three paths (P_1', P_2'') , $(P_1', (u,v), P_1'')$ and (P_2', P_1'') with flow amounts f_2, $f_1 - f_2$ and f_2 respectively and the edge (u,v) will not be passed in opposite directions. The maximum number of these paths is m since each augmenting flow contains at least one saturated edge. Hence, there is one augmenting path with a capacity at least $\frac{|f^*|}{m}$. □

Thus flow amount remained after the first iteration is

$$\text{Rem}\,(1) \leq |f^*|(1 - 1/m);$$

amount of flow pushed on the second iteration is

$$\text{Push}\,(2) \geq (|f^*|\frac{m-1}{m})\frac{1}{m}\ .$$

Finally, amount of flow remained after the k-th iteration is

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

Solving equation

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

obtain the number of iterations $k \sim m \log |f^*|$.

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 $O(\,(m^2 + mn \log n) \log |f^*|)$.

2nd Edmonds-Karp Algorithm:

while (there is an augmenting path $s-t$ in G) **do**
pick up the shortest augmenting path (counting only the number of edges);
use this path to augment the flow;

Analysis: Consider a layered graph L_G obtained after BFS in G_f from s. There are at most m augmenting paths in the graph L_G, each of them can be found in time $O(m)$. When all the augmenting paths are exhausted in L_G, consider a new one which will have a greater number of layers. Since the maximum number of layers is n, the overall time complexity of the algorithm is $O(nm^2)$.

History

Ford-Fulkerson (1956)
Edmonds-Karp (1969) $O(nm^2)$
Dinic (1970) $O(n^2m)$
Karzanov (1974) $O(n^3)$
MKM (1977)$O(n^3)$

Notes by Marios Leventopoulos

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

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

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

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

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

Step 1: $f = 0$

Step 2: Construct G_f

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

Step 4: $f \leftarrow f + g$

Step 5: goto step 2 (next phase)

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

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

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

Step 1: Pick a vertex v with minimum throughput,

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

Step 3: Repeat until t is disconnected from s

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

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

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

To summarize we have used the following techniques:

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

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

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

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

Notes by Samir Khuller.

16 Pre-flow Push Method

Please read Chapter 27.4 (pp. 605 – 614) from [CLR]. The book explains the pre-flow push method very well.

There will be no lecture on Mar 11. We will have an Examination instead.

CMSC 651 Advanced Algorithms	Lecture 16
Lecturer: Samir Khuller	Tu. March 23, 1993

Notes by Marsha Chechik.

17 Planar Graphs

A graph is said to be *embeddable in the plane*, or *planar*, if it can be drawn in the plane so that its edges intersect only at their ends.

17.1 Euler's Formula

There is a simple formula relating the numbers of vertices (V), edges (E) and faces (F) in a connected planar graph - Euler's formula:

$$V - E + F = 2.$$

Proof:
Simply by induction. □

To remember this formula, think of a cycle (n vertices, n edges, 2 faces) or of a tree (n vertices, $n-1$ edges, 1 face).
From this formula, we can prove that a planar graph has a linear number of edges ($E \leq 3V - 6$).
Planar graphs are useful since they appear a lot in the context of VLSI design.

Theorem 17.1 (Separator Property (Lipton - Tarjan)) *Given a planar graph G=(V,E), there exists disjoint vertex partition $V = A \cup B \cup C$, such that*

1. $|C| = O(\sqrt{n})$. *Actually,* $|C| \leq 2\sqrt{2}\sqrt{n}$.
2. *There are no edges between vertices in A and vertices in B. Therefore, removal of C makes the graph disconnected. C is called a* separator.
3. $|A|, |B| \leq \frac{2}{3}n$.

Note (Alon, Seymour, Thomas): Computing the separator set of a family with excluded minor H, takes $O(|H|^{1.5}\sqrt{n}n)$.
Examples:

1. See Fig. 17. This is the grid from the midterm. Removal of the middle row ($\sqrt{n}$ elements) makes the graph disconnected.
2. Of course, A does not have to be connected. Rather, it could contain many small graphs. See Fig. 18.

This theorem is used for divide-and-conquer algorithms.

Figure 17: Application of the separator property.

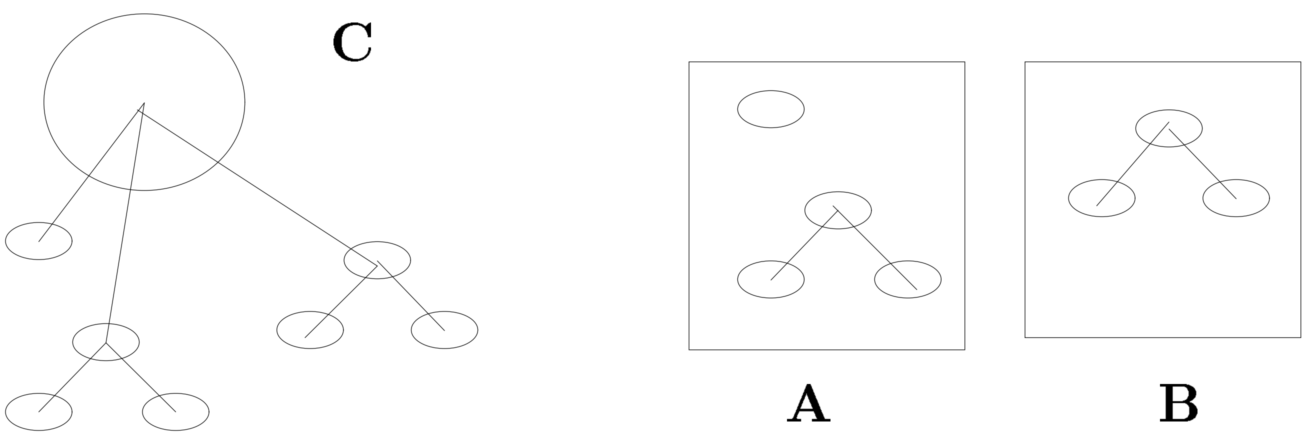


Figure 18: Multiple graphs in A.

17.2 Smallest Non-Planar Graphs

Theorem 17.2 *The smallest non-planar graphs are K_5 and $K_{3,3}$.*

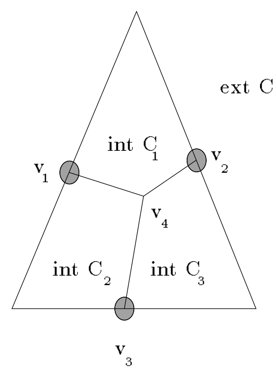


Figure 19: Embedding of K_5

Proof:

We will show that K_5 is non-planar. The proof is by contradiction. Let G be a planar graph corresponding to K_5. Let the vertices be called by v_1, v_2, v_3, v_4, and v_5. Since G is complete, any two of the vertices are joined by an edge. Assume, without the loss of generality, that after inserting vertex v_4, the graph looks like Fig. 19. Now v_5 must lie in one of the four regions $extC$, $intC_1$, $intC_2$, and $intC_3$. If $v_5 \in extC$ then the edge (v_4v_5) must cross C at some point. This contradicts the assumption that G is a planar graph. Cases where $v_5 \in intC_i$, i = 1, 2, 3, can be treated similarly. Nonplanarity of $K_{3,3}$ can be proven the same way. (A simple proof of non-planarity of K_5 follows by Euler's formula as was observed by Marsha Chechik.) $\square$

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.

17.3 Bridges

Definition: Consider a cycle C in G. Edges e and e' are said to be in the same bridge if there is a path joining them that does not use any vertex in C.

By this definition, edges form equivalence classes that are called bridges. A trivial bridge is a singleton edge.

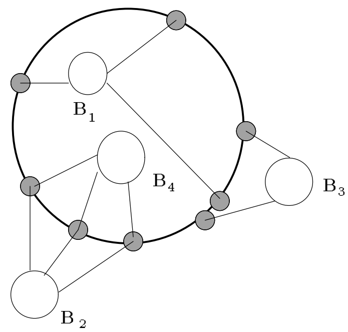


Figure 20: Bridges relative to the cycle C.

Definition: A common vertex between the cycle C and one of the bridges with respect to C, is called *a vertex of attachment.*

Definition: Two bridges *avoid* each other with respect to the cycle if all vertices of attachment are on the same segment. In Fig. 20, 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 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. 20). Therefore, two bridges either

1. Avoid each other
2. Overlap (don't avoid each other).
   - Skew: A vertex of attachment of one bridge lies on a segment between vertices of attachment of the other bridge (see Fig. 21).
   - 3-equivalent (like B_2 and B_4 in Fig. 20)

It can be shown that other cases reduce to the above cases. For example, a 4-equivalent graph (see Fig.22) can be classified as a skew, with vertices u and v belonging to one bridge, and vertices u' and v' belonging to the other.

17.4 Kuratowski's Theorem

Consider non-planar graphs that do not contain K_5 or $K_{3,3}$ as a homeomorph. 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.

The following theorem is claimed without proof.

Theorem 17.3 *Such a graph G is 3-connected.*

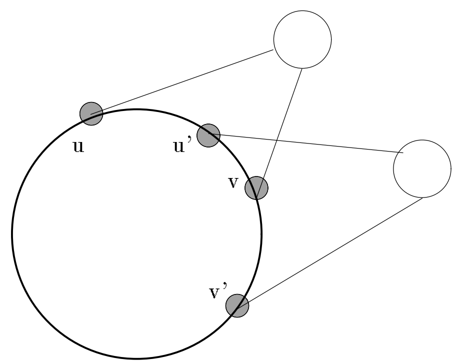


Figure 21: Two bridges that are skew.

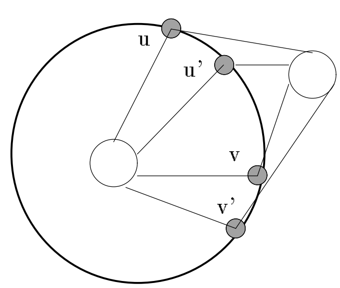


Figure 22: Two bridges with 4 attachment points.

Theorem 17.4 (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 $G' = G - (u, v)$ is planar. Take a planar embedding of G', and take a cycle C passing through u and v such that G' has the largest number of edges in the interior of C. (Such a cycle must exist due to the 3-connectivity of G.)

Claims:

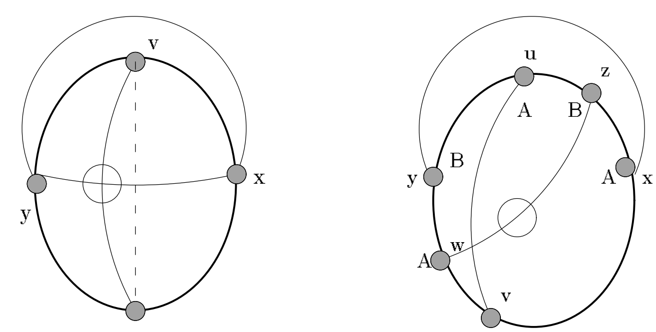
1. Every external bridge with respect to C has exactly two attachment points. Otherwise, a path in this bridge going between two attachment points, could be added to C to obtain more edges inside C.

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 u and v could be connected while keeping the graph planar.

4. There is at least one internal bridge, to prevent an internal edge between u and v.

5. There is at least one internal and one external bridge that prevent 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 23: Possible vertex-edge layouts.

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

1. See Fig. 23(a). This graph contains a homeomorph of K_5. Notice that every vertices are connected with an edge.

2. See Fig. 23(b). This graph contains a homeomorph of $K_{3,3}$. To see that, let $A = \{u, x, w\}$ and $B = \{v, y, z\}$. Then, there is an edge between every $v_1 \in A$ and $v_2 \in B$.

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

Definition: Graph G contains H as a *minor* if the exists a set of edges which can be thrown away or contracted, to obtain H.

Note: The Kuratowski's theorem can be extended to showing that every non-planar graph contains K_5 and $K_{3,3}$ minors.

Notes by Michael Tan and Samir Khuller.

18 Graph Minor Theorem and other CS Collectibles

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

Definition 18.1 (minor) *We say that H is a* **minor** *of G ($H \leq G$) if H can be obtained from G by edge removal, vertex removal and edge contraction.*

Definition 18.2 (closed under minor) *Given a class of graphs $\mathcal{C}$, and graphs G and H, $\mathcal{C}$ is closed under taking minors if $G \in \mathcal{C}$ and $H \leq G$ implies $H \in \mathcal{C}$.*

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

Theorem 18.3 (Graph Minor Theorem Corollary) [*Robertson−Seymour*] *If $\mathcal{C}$ is any class of graphs closed under taking minors, then there exists a finite obstruction set $H_1, \ldots, H_l$ such that $G \in \mathcal{C} \leftrightarrow (\forall i)[H_i \not\leq G]$.*

The proof of this theorem is highly non-trivial and is omitted.

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

Theorem 18.4 *Testing $H \leq G$, for a fixed H, can be done in time $O(n^3)$, if n is the number of vertices in G.*

Notice that the running time does not indicate the dependence on the size of H. In fact, the dependence on H is huge! There is an embarassingly large constant that is hidden in the big-O notation. This algorithm is far from practical for even very small sizes of H.

Some results of the above statements:

1. If $\mathcal{C}$ ={all planar graphs}, we can test $G \in \mathcal{C}$ in O(n^3) time. (To do this, test if K_5 and $K_{3,3}$ are minors of G. Each test takes O(n^3) time.)

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

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

18.1 Towards an algorithm for testing planarity in genus k

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

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

Theorem 18.5 *Given a graph G and an oracle that determines planarity on a surface with genus k, we can determine if G is planar in genus k and find such a planar embedding in polynomial time.*

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

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

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

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

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

18.2 Applications:

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

Algorithm 1 [Fellows]:

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

```
                 . VC={}, (u,v) is uncovered
                / \
      VC = {u} /   \  VC = {v}
             .      .
            / \   / \
   VC={u,v4}/   \ /   \  VC = {v, v2}
           etc.
        . . . . . . . . .
```

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

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

Notes by Vadim Kagan.

19 Graph Coloring

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

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

Theorem 19.1 (6-colorability) *If $G = (V, E)$ is planar,then G is 6-colorable.*

Proof:

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

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

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

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

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

Theorem 19.2 (5-colorability of planar graphs) *Every planar graph is 5-colorable.*

Proof:

We will use the same approach as for the 6-colorability. Proof will again be on induction on the number of vertices in the graph. The base case is trivial. For inductive case, we again find the minimum degree vertex. When we delete it from the graph, the resulting graph is

5-colorable by induction hypothesis. This time, though, putting the deleted vertex back in the graph is more complicated than in 6-colorability case. Let us denote the vertex that we are trying to put back in the graph by v. If v has ≤ 4 neighbours, putting it back is easy since one color is left unused and we just assign v this color. Suppose v has 5 neighbours (since v was the minimum-degree vertex in a planar graph, its degree is ≤ 5) and all of them have different colors (if any two of v's neighbours have the same color, there is one free color and we can assign v this color). To put v back in G we have to change colors somehow (see Fig. 24). How do we change colors? Let us concentrate on the subgraph of nodes of colors 2 and 4. Suppose colors 2 and 4 belong to different connected components. Then flip colors 2 and 4 in the component that vertex 4 belongs to, i.e., change color of all 4-colored nodes to color 2 and nodes with color 2 to color 4. Since colors 2 and 4 are in the different connected components, we can do this without violating the coloring (in the resulting graph, all the adjacent nodes will still have different colors). But now we have a free color - namely, color 4 - that we can use for v.
What if colors 2 and 4 are in the same component? In this case, there is a path P of alternating colors (as shown on Fig. 24).

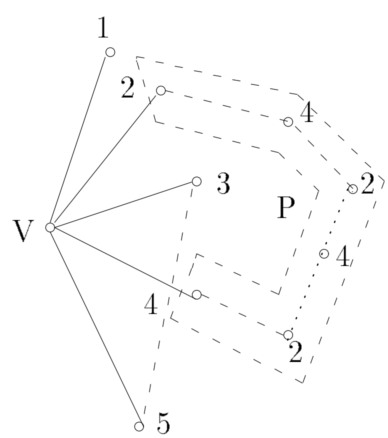


Figure 24: Graph with alternating colors path

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

20 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?

In the algorithms described in this section, the notion of *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. 25).

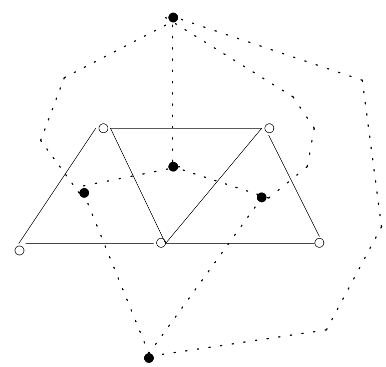


Figure 25: Graph with corresponding dual

For every face in G put a vertex in G^D; if two faces f_1, f_2 share an edge in G, put an edge between vertices corresponding to f_1, f_2 in G^D (if two faces share two edges, add two edges to G^D). Note that $G^{DD} = G, |V^D| = F, |E^D| = E$, where F is the number of faces in G.

20.1 Two algorithms

Uppermost Path Algorithm

1. Take the uppermost path (see Fig. 26).
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.

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

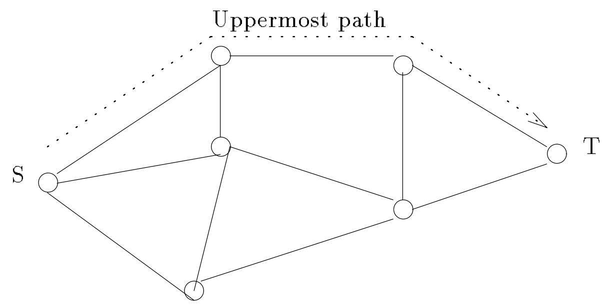


Figure 26: Uppermost path

for min-capacity edge finding. To reflect the decreasing residual capacities, keep a counter that shows by how much we decrease keys of edges in the heap. For example, if counter for a particular heap is -1, it means that the key of every edge in the heap is to be decreased by 1, i.e. an edge with key 5 should be interpreted as having key 4. When adding an edge with key 5 to this heap, the key should be increased to 6.

Hassin's algorithm

Given a graph G, construct a dual graph G^D and add to it two new nodes $s^\star$ and $t^\star$, both on the infinite face (see Fig. 27).

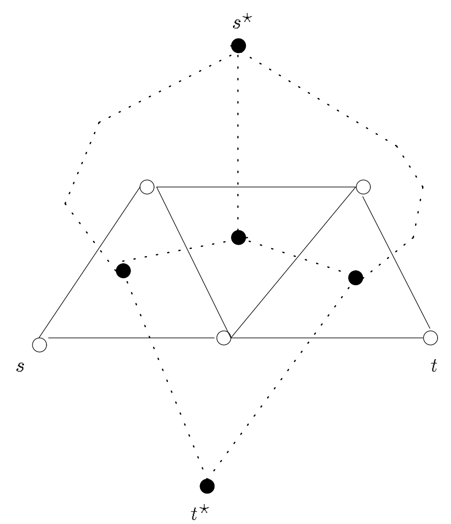


Figure 27: $G^{D\star}$

Node $s^\star$ is added at the top of G^D and every node in G^D corresponding to an edge on the top of G is connected to $s^\star$. Node $t^\star$ is similarly placed and connected at the bottom of

G^D. 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 $G^{D\star}$, weight of each edge is equal to the capacity of the corresponding edge in G, so the min-cut in G corresponds to the shortest path between $s^\star$ and $t^\star$ in $G^{D\star}$. There exists a number of algorithms for shortest path finding. Frederickson's algorithm, for example, has running time $O(n\sqrt{\log n})$.

Now that we have found the value of maxflow, the problem is to find the flow through each individual edge.

To do that, let us label each node v_i in $G^{D\star}$ with d_i, v_i's distance from $s^\star$ (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 through it in such a way that the largest-labeled of the two "adjacent" to e nodes in $G^{D\star}$ is on the right (see Fig. 28).

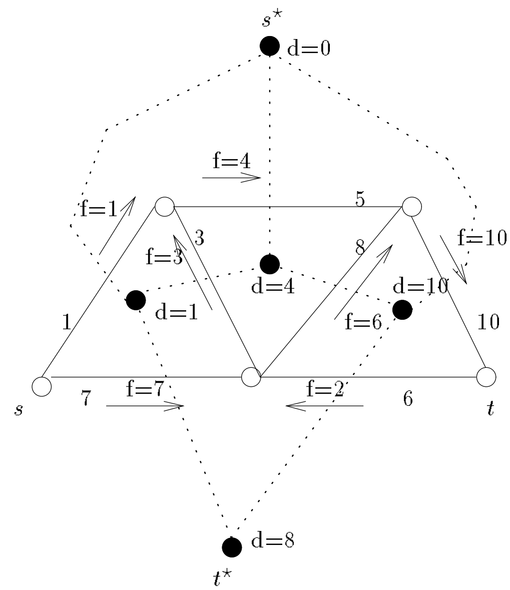


Figure 28: Labeled graph

If d_i and d_j are labels for a pair of adjacent to e faces, the value of the flow through e is defined to be equal to the difference between the larger and the smaller of d_i and d_j. Let us prove that thus 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 target).
Suppose there is an edge in G having capacity C (as on Fig. 29), such that the amount of flow f through it is greater than C: $d_i - d_j > C$ (assuming, without loss of generality, that $d_i > d_j$).

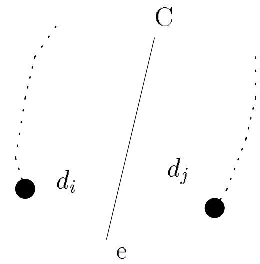


Figure 29:

Then, we can decrease d_i (which is the cost of the shortest path to this node from $s^\star$) by following the shortest path to v_j and then going from d_j to d_i across e. In this case, the label for $v_i = d_j + C < d_i$. This contradicts the fact that the original d_i was already the cost of the shortest path, and therefore no such edge e exists.

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 target), let us do the following. We will go around in some (say, counterclockwise) direction and add together flows through all the edges adjacent to the vertex (see Figure 30). Note that, for edge e_i, $d_i - d_{i+1}$ gives us negative value if e_i

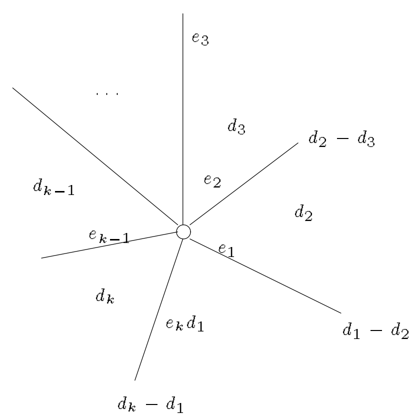


Figure 30:

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 $d_1 - d_2 + d_2 - d_3 + \ldots + d_{k-1} - d_k + d_k - d_1 = 0$, from which it follows that flow entering the node is equal to the amount of flow leaving it.

Notes by Maria Chechik & Suleyman Sahinalp

21 On Line vs. Off Line: A Measure for Quality Evaluation

Suppose you have a dynamic problem. An algorithm for solving a dynamic problem which uses information about future is called an off-line algorithm. But computer scientists, are human beings without the capability of predicting the future. Hence their algorithms are usually on-line (i.e. they make their decisions without any information about the future).

21.1 K-Server Problem

In a fraternity party there are several robots which are responding to requests of the frat-members. Whenever someone requests a drink, one of the robots is sent to him. The problem here is to decide which robot is "best suited" for the request; moreover to find a measure for being "best suited".
If you knew the sequence of the location of requests, then you'd try to minimize the total distance your robots would travel until the end of the party. (It is an interesting exercise to solve this problem.) As you don't have this information you try to achieve some kind of a measure for assuring you (as the robot controller) didn't perform "that badly".

Conjecture 21.1 *If you have k robots, then there exists an on-line decision algorithm such that*

$$\textit{total cost of on-line alg} \leq k \cdot \textit{total cost of off-line alg} + C$$

where C is some constant.

Example:
In this example the requests come from the vertices of an equilateral triangle and there are only two servers. Suppose the worst thing happens and the requests always come from the vertex where there is no server. If the cost of moving one server from one vertex to a neighboring vertex is 1 then at each request the cost of the service will be 1. So in the worst case the on-line algorithm will have a cost of 1 per service whatever you do to minimize the cost.
To analyse the performance of an on-line algorithm, we will run the on-line algorithm on a sequence provided by an adversary, and then compare the cost of the on-line algorithm to the cost of the adversary (who may know the sequence before answering the queries).

For example, assume you have two robots, and there are three possible places where a service is requested: vertices A, B, and C. Let your robots be in the vertices A and B. The adversary will request service for vertex C. If you move the robot at vertex B (hence your servers will be at vertices A and C), the next request will come from B. If you now move your robot at vertex C (hence you have robots at A and B), the next request will come from C and so on. In this scheme you would do k moves for a sequence of k requests but after finishing your job the adversary will say: "you fool, if you had moved the robot at A in the beginning, then you would have servers at vertices B and C and you wouldn't have to do anything else for the rest of the sequence".
We can actually show that for *any* on-line algorithm, there is an adversary that can force the on-line algorithm to spend at least twice as much as the offline algorithm. Consider any on-line algorithm A. The adversary generates a sequence of requests $r_1, r_2, \ldots, r_n$ such that each request is made to a vertex where there is no robot (thus the on-line algorithm pays a cost of n). It is easy to see that the offline algorithm can do a "lookahead" and move the server that is not requested in the immediate future. Thus for every two requests, it can make sure it does not need to pay for more than one move. This can actually be extended to a *lower bound* of k for the k-Server problem. In other words, any on-line algorithm can be forced to pay a cost that is k times the cost of the off-line algorithm (that knows the future).

21.2 K-servers on a straight line

Suppose you again have k servers on a line. The naive approach is to move the closest server to the request. But what does our adversary do? It will make a request near the initial position of one server (which moves), and then requests the original position of the server. So our server would be running back and forth to the same two points although the adversary will move another server in the first request, and for all future requests it would not have to make a move.
On-line strategy: We move the closest server to the request but we also move the server on the other side of the request, towards the location of the request by the same amount. If the request is to the left (right) of the leftmost (rightmost) server, then only one server moves.
We will analyse the algorithm as a game. There are two copies of the server's. One copy (the s_i servers) are the servers of the online algorithm, the other copy is the adversary's servers (a_i). In each move, the adversary generates a request, moves a server to service the request and then asks the on-line algorithm to service the request.
We prove that this algorithm does well by using a potential function, where

- $\Phi(t-1)$ is potential before the t^{th} request,
- $\Phi'(t)$ is potential after adversary's t^{th} service but before your t^{th} service,
- $\Phi(t)$ is the potential after your t^{th} service.

The potential function satisfies the following properties:

1. $\Phi'(t) - \Phi(t-1) < k\times$ (cost for adversary to satisfy the t^{th} request)

2. $\Phi'(t) - \Phi(t) >$ (cost for you to satisfy the t^{th} request)

3. $\Phi(t) > 0$

Now we define the potential function as follows:
$\Phi(t) := k \times$ Weight of Minimum Matching$in G' + \sum_{i<j} \text{dist}(s_i, s_j)$.
The minimum matching is defined in the bipartite graph G' where

- $A \equiv$ set of the placements of your servers
- $S \equiv$ set of the placements of adversarys servers
- edge weights$(a_i,\ s_j) \equiv$ distance between those two servers

The second term is obtained by simply keeping the online algorithm's servers in sorted order from left to right.
Why does the potential function work: It is easy to see that those three properties are satisfied by our potential function:

1. the weight of the minimum matching will change at most the cost of the move by the adversary and distance between your servers do not change. Hence the first property follows.

2. if the server you're moving is the rightmost server and if the move is directed outwards then the second term of the potential function will be increased by $(k-1)d$ however the first term will be decreasing by kd, hence the inequality is valid. (Notice that the first term decreases because there is already one of the adversary's server's at the request point.) If the server you're moving is the leftmost server and the motion is directed outwards then the second term will increase by at most $(k-1)d$, though the first term will decrease by kd, hence the inequality is again valid. Otherwise, you will be moving two servers towards each other and by this move the total decrease in the second term will be $2d$ (the cost for service): two servers are $2d$ closer and the increase in distance by the first server is exactly same with the decrease in distance by the second server. In the worst case the size of the matching won't change but again because of the second term our inequality is satisfied.

3. Obvious.

Since the potential is always positive, we know that

$$\text{Initial Potential} + \text{Total Increase} \geq \text{Total Decrease}.$$

Notice that the total decrease in potential is an upper-bound on our algorithm's cost. The Total Increae in potential is upper-bounded by $k\times$ the adversary's cost. Therefore,

$$(\text{cost of your moves}) < k \times (\text{total cost of adversary's moves}) + \text{Initial Potential}.$$

Notes by Samir Khuller.

22 NP-Completeness

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

1. SAT to CLIQUE.
2. CLIQUE to MULTIPROCESSOR SCHEDULING.
3. SAT to DISJOINT CONNECTING PATHS.

The first reduction is taken from Chapter 36.5 [CLR] pp. 946–949.
The second reduction goes as follows.
MULTIPROCESSOR SCHEDULING: Given a DAG representing precedence constraints, and a set of jobs J all of unit length. Is there a schedule that will schedule all the jobs on M parallel machines (all of the same speed) in T time units ?
Essentially, we can execute upto M jobs in each time unit, and we have to maintain all the precedence constraints and finish all the jobs by time T.
Reduction: Given a graph $G = (V, E)$ and an integer k we wish to produce a set of jobs J, a DAG as well as parameters M, T such that there will be a way to schedule the jobs if and only if G contains a clique on k vertices.
Let n, m be the number of vertices and edges in G respectively.
We are going to have a set of jobs $\{v_1, \ldots, v_n, e_1, \ldots, e_m\}$ corresponding to each vertex/edge. We put an edge from v_i to e_j (in the DAG) if e_j is incident on v_i in G. Hence all the vertices have to be scheduled before the edges they are incident to. We are going to set $T = 3$.
Intuitively, we want the k vertices that form the clique to be put in time slot 1. This makes $C(k, 2)$ edges available for time slot 2 along with the remaining $n-k$ vertices. The remaining edges $m - C(k, 2)$ go in slot 3. To ensure that no more than k vertices are picked in the first slot, we add more jobs. There will be jobs in 3 sets that are added, namely B, C, D. We make each job in B a prerequisite for each job in C, and each job in C a prerequisite for each job in D. Thus all the B jobs have to go in time 1, the C jobs in time 2, and the D jobs in time 3.
The sizes of the sets B, C, D are chosen as follows:

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

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

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

DISJOINT CONNECTING PATHS:

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

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

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

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

Proofs of this construction are left to the reader.

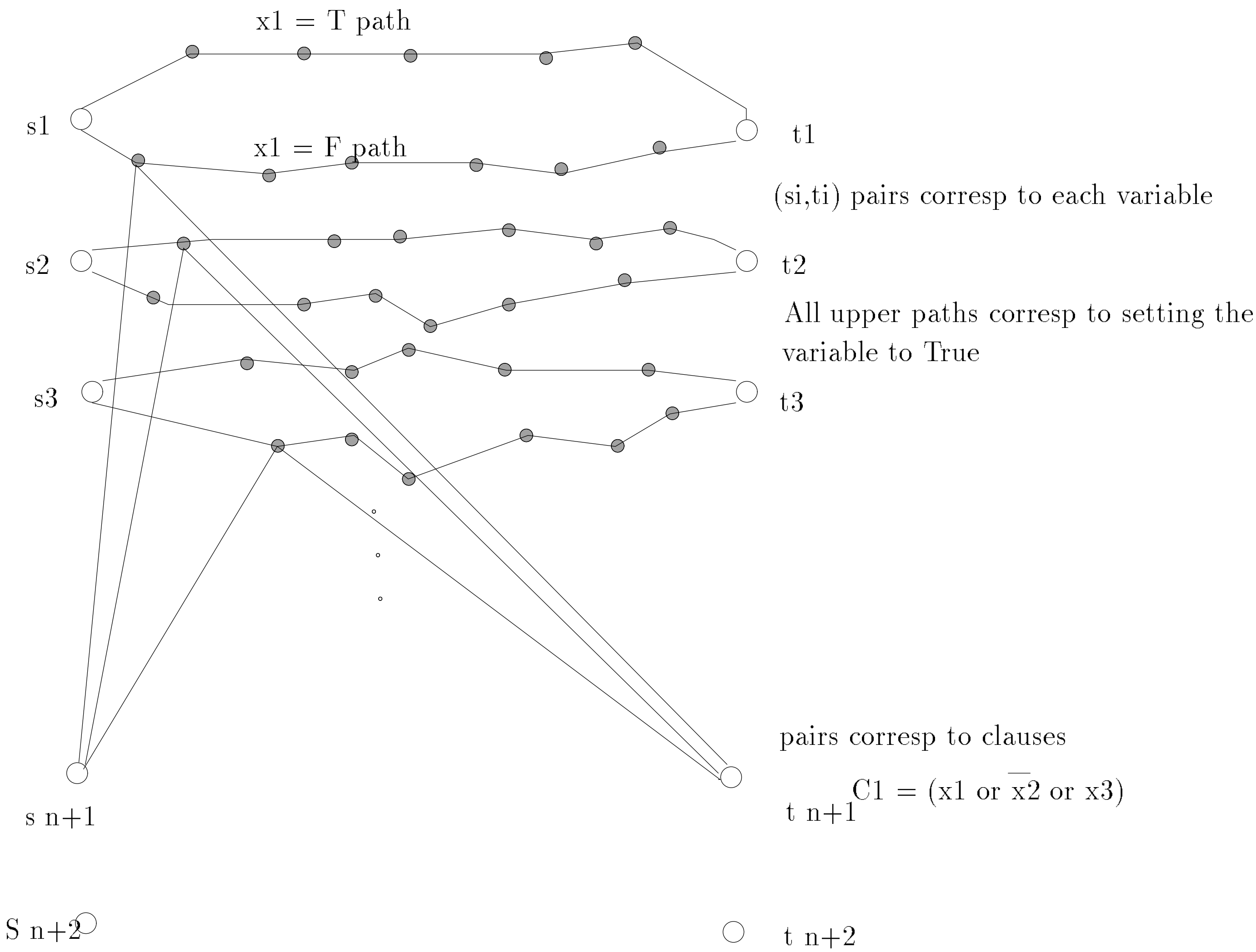


Figure 31: Graph corresponding to formula

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

Notes by Samir Khuller.

23 More on NP-Completeness

The reductions we will study today are:

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

I will outline the proof for 3SAT to 3COLORING. The other proofs may be found in Garey and Johnson.
We are given a 3SAT formula with clauses $C_1, \ldots, C_m$ and n variables $x_1, \ldots, x_n$. We are going to construct a graph that is 3 colorable if and only if the formula is satisfiable.
We have a clique on three vertices. Each node has to get a different color. W.l.o.g, we can assume that the three vertices get the colors $T, F, *$. (T denotes "true" F denotes "false".)
For each variable x_i we have a pair of vertices X_i and $\overline{X_i}$ that are connected to each other by an edge and also connected to the vertex $*$. This forces these two vertices to be assigned colors different from each other, and in fact there are only two possible colorings for these two vertices. If X_i gets the color T, then $\overline{X_i}$ gets the color F and we say that variable x_i is true. Notice that each variable can choose a color with no interference from the other variables. These vertices are now connected to the gadgets that denote clauses, and ensure that each clause is true.
We now need to construct an "or" gate for each clause. We want this gadget to be 3 colorable if and only if one of the inputs to the gate is T (true). We construct a "gadget" for *each* clause in the formula. For simplicity, let us assume that each clause has exactly three literals. If all three inputs to this gadget are F, then there is no way to color the vertices. The vertices below the input must all get color $*$. The vertices on the bottom line now cannot be colored. On the other hand if any input is T, then the vertex below the T input can be given color F and the vertex below it (on the bottom line) gets color $*$, and the graph can be three colored.
If the formula is satisfiable, it is really easy to use this assignmenmt to the variables to produce a three coloring. We color each vertex X_i with T if x_i is True, and with F otherwise. (The vertex $\overline{X_i}$ is colored appropriately.) We now know that each clause has a true input. This lets us color each OR gate (since when an input it true, we can color the gadget). On the other hand, if the graph has a three coloring we can use the coloring to obtain a satisfying assignment (verify that this is indeed the case).

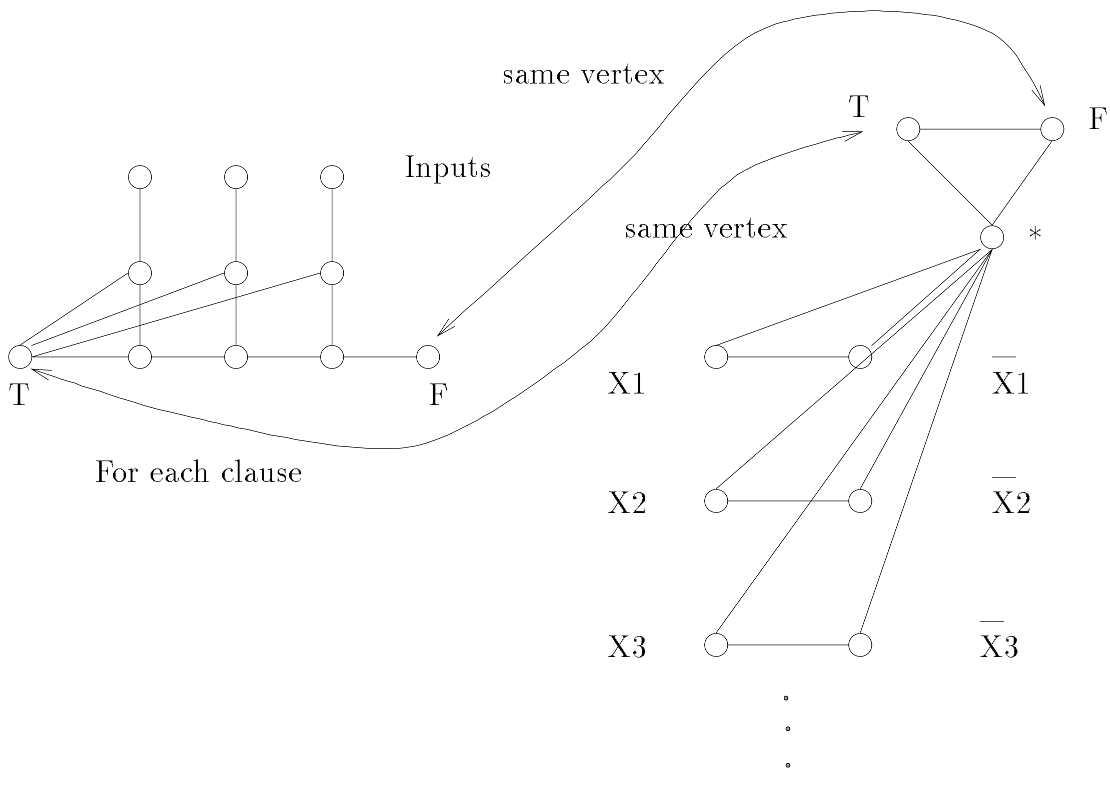


Figure 32: Gadget for OR gate

Notes by Samir Khuller.

24 Approximation Algorithms

Please read Chapter 37 (pp. 964–974) from [CLR] for Vertex Cover (unweighted) and Traveling Salesman Problem.

Notes by Samir Khuller.

25 Approximation Algorithms

Please read Chapter 37 (pp. 974–978) from [CLR] for Set Cover algorithm. Weighted Vertex Cover will be covered in the next lecture.

CMSC 651 Advanced Algorithms	Lecture 24
Lecturer: Samir Khuller	Tu. Apr. 20, 1993

Notes by Yachai Limpiyakorn.

26 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 Vertex Cover $C \subseteq V$, and $w : V \rightarrow R^{+}$

We will apply the greedy algorithm for the WVC problem.

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.

Algorithm MGA:
Input: Graph $G(V, E)$ and weight function w on V
Output: Vertex Cover C

1. $\forall v \in V$ do $W(v) \leftarrow w(v)$;
2. $\forall v \in V$ do $D(v) \leftarrow d(v)$;
3. $\forall e \in E$ do $EC(v) \leftarrow 0$;
4. $C \leftarrow \emptyset$
5. while $E \neq \emptyset$ do begin
6. Pick a vertex $v \in V$ for which $\frac{W(v)}{D(v)}$ is minimized;
7. $C \leftarrow C + v$;
8. $\forall e = (u, v) \in E$ do begin
9. $E \leftarrow E - e$; update $D(u)$
10. $W(u) \leftarrow W(u) - \frac{W(v)}{D(v)}$;
11. $EC(e) \leftarrow \frac{W(v)}{D(v)}$;
12. end;

13. $W(v) \leftarrow 0$;

14. end;

15. return C

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 edge cost $EC(e)$ reflects the cost of covering the edge e.
The algorithm assigns costs to the edges in a manner which guarantees that each vertex in the cover partitions its weight amongst the incident edges, and each edge gets assigned the same weight from both its end-points. Thus, the weight of the cover being produced is at most twice the net cost of the edges. Under any such choice of the edge cost function, it can be easily seen that an optimal cover must have weight at least as large as the total of the edge costs.

Observing that, at all times during the execution of the algorithm, the following invariants hold :

$$\forall e \in E :: EC(e) \geq 0$$

since the only modification to the edge costs is the addition of a positive value.

$$\forall v \in V :: W(v) \geq 0$$

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).

$$\forall v \in V :: w(v) = W(v) + \sum_{u \in N(v)} EC(u, v)$$

where $N(v)$ denotes the set of neighbors of v in the *original* graph

The algorithm terminates with the facts that,

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

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

Eq.(1) holds since $\forall v \in C, W(v) = 0$.
Eq.(2) holds due to INV2 and INV3.

From the above facts, we can derive the following lemmata to relate the weight of MGA's output to the book-keeping variables of edge costs.

Lemma 26.1

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

Proof:
Observe that by eq.(1)

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

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

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

$\square$

The next step is to relate the edge costs to the value of the optimal solution.

Lemma 26.2

$$\sum_{e \in E} EC(e) \leq c^* = w(C^*)$$

where

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

Proof:
Observe that

$$\sum_{e \in E} EC(e) \leq \sum_{v \in C^*} \sum_{u \in N(v)} EC(u, v) \leq \sum_{v \in C^*} w(v)$$

As C^* is a VC, the second expression must count each edge at least once. From eq.(1), we now have the desired result. $\square$

Putting together these two lemmata, we then have

$$w(C) \leq 2 \sum_{e \in E} EC(e) \leq 2w(C^*),$$

that is, the weight of C is at most twice optimal.

Algorithm MGA runs in time $O(m \log n)$ and has $R_{MGA} = 2$ which is the best possible bound.

Packing Functions

A packing function p is defined as follows:

$$p : E \rightarrow R^+$$

such that

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

Note that zero packing is a valid packing by setting packing value of every edge to be zero.

Example (see Fig. 33)

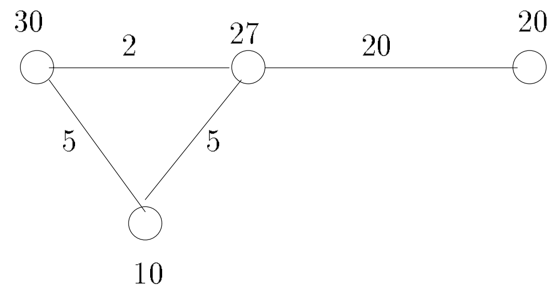


Figure 33:

$$\sum_{e \in E} p(e) = P = 32$$

Observing that

$$P \leq c^*$$

By considering

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

This is true because some edges may be counted twice in the third expression.

We claim that :

$$cost(algo) \leq 2P \leq 2c^*$$

Since $EC(u,v)$ is a valid packing, following the previous proof and substituting $EC(u,v)$ by $p(u,v)$, then we will get the desired result.

Maximal packing : packing which cannot increase $p(e)$ for any e.

Example (see Fig. 34)
If we pick full capacity nodes in maximal packing; that is, a node v is in the VC if $\sum_{u \in N(v)} p(u,v) = w(v)$, and results in

$$P \leq c^* \leq 2P.$$

This set of vertices forms a cover since for each edge at least one endpoint has its constraint met with equality.

Why "Packing" ?
Packing is the dual problem to Vertex Cover problem in Linear Programming.

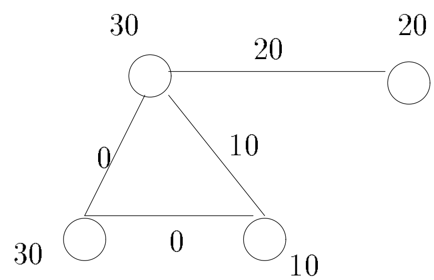


Figure 34:

Vertex Cover Problem in LP
Define function $X(v) = 0, 1$ if $v \notin VC$ and $v \in VC$ respectively.

We want to minimize $\sum_{v \in V} x(v)w(v)$ such that $x(u) + x(v) \geq 1$ for all edges (u, v).

Clearly, this is an Integer Linear Program (solving these is NP-hard). We can relax the integer constraints and make a regular Linear Program from it. For minimization problems, the relaxed Linear Program solution is no more than the solution to the Integer Linear Program; that is $c^*_{ILP} \geq c^*_{LP}$, and c^*_{LP} in turn will be lower bounded by any solution of its DUAL LP, in which here we want to maximize $\sum_{e \in E} p(e)$ such that $\forall v \in V \sum_{u \in N(v)} p(u, v) \leq w(v)$.

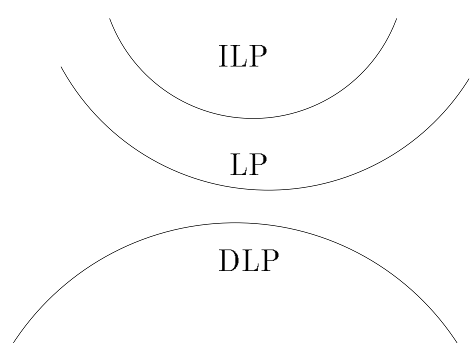


Figure 35:

If we can prove that $cost(algo) \leq kc_{DLP}$ then we get

$$cost(algo) \leq kc^*_{DLP} \leq kc^*_{LP} \leq kc^*_{ILP}.$$

So the entire difficulty is in proving the first inequality. The rest follow automatically. In the Vertex Cover Problem, we prove this by showing that the cost of our solution is no more than twice a packing (which is a feasible dual solution).

CMSC 651 Advanced Algorithms	Lecture 25
Lecturer: Samir Khuller	Thu April 22, 1993

Notes by Marga Alisjahbana.

27 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.

27.1 Approximation Algorithm

Algorithm :

1. Construct a new graph $H = (S, E_S)$, which is a complete graph. The edges of H have weight $w(i, j)$ = minimal weight path from i to j in the original graph G.
2. Find a minimum spanning tree MST_H in H.
3. Construct a subgraph G_S of G by replacing each edge in MST_H by its corresponding shortest path in G.
4. Find a minimal spanning tree MST_S of G_S.
5. Construct a Steiner tree T_H from MST_S by deleting edges in MST_S if necessary so that all leaves are in S (these are redundant edges, and can be created in the previous step).

Running time :

1. step 1 : $O(|S||V|^2)$
2. step 2 : $O(|S|^2)$
3. step 3 : $O(|V|)$
4. step 4 : $O(|V|^2)$
5. step 5 : $O(|V|)$

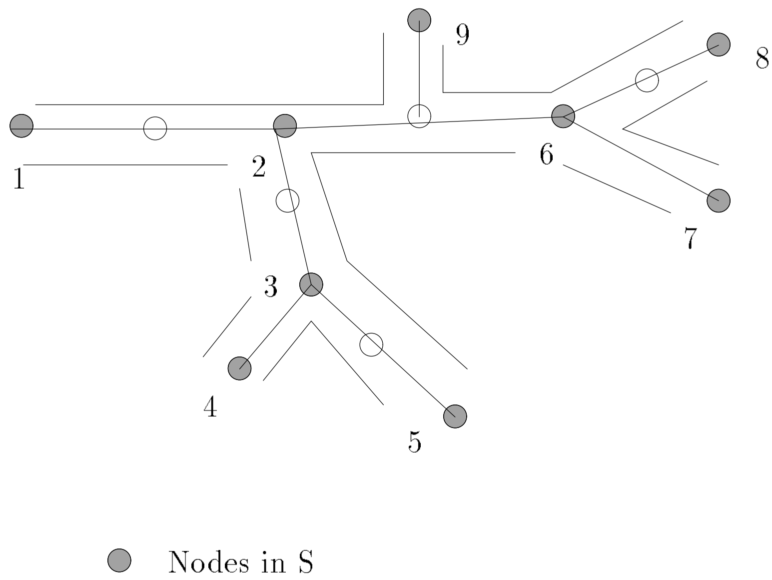


Figure 36: Optimal Steiner Tree

So worst case time complexity is $O(|S||V|^2)$.
Will will show that this algorithm produces a solution that is at most twice the optimal.
More formally: weight(our solution) $\leq weight(MST_H) \leq 2 \times$ weight(optimal solution)
To prove this, consider the optimal solution, i.e., the minimal Steiner tree T_opt.
Do a DFS on T_opt and number the points in S in order of the DFS. (Give each vertex a number the first time it is encountered.) Traverse the tree in same order of the DFS then return to starting vertex. Each edge will be traversed exactly twice, so weight of all edges traversed is $2 \times weight(T_opt)$. Let $d(i, i+1)$ be the length of the edges traversed during the DFS in going from vertex i to $i+1$. (Thus there is a path $P(i, i+1)$ from i to $i+1$ of length $d(i, i+1)$.) Also notice that the sum of the lengths of all such paths $\sum_{i=1}^{|S|} d(i, i+1) = 2 \cdot MST_S$. (Assume that vertex $|S|+1$ is vertex 1.)
We will show that H contains a spanning tree of weight $\leq 2 \times w(T_opt)$. This shows that the weight of a mininal spanning tree in H is no more than $2 \times w(T_opt)$. Our steiner tree solution is upperbounded in cost by the weight of this tree. If we follow the points in S, in graph H, in the same order of their above DFS numbering, we see that the weight of an edge between points i and $i+1$, in H, cannot be more than the length of the path between the points in MST_S during the DFS traversal (i.e., $d(i, i+1)$). So using this path we can obtain a spanning tree in H (which is actually a Hamilton Path in H) with weight $\leq 2 \cdot w(T_opt)$.
Figure 37 shows that the worst case performance of 2 is achievable by this algorithm.

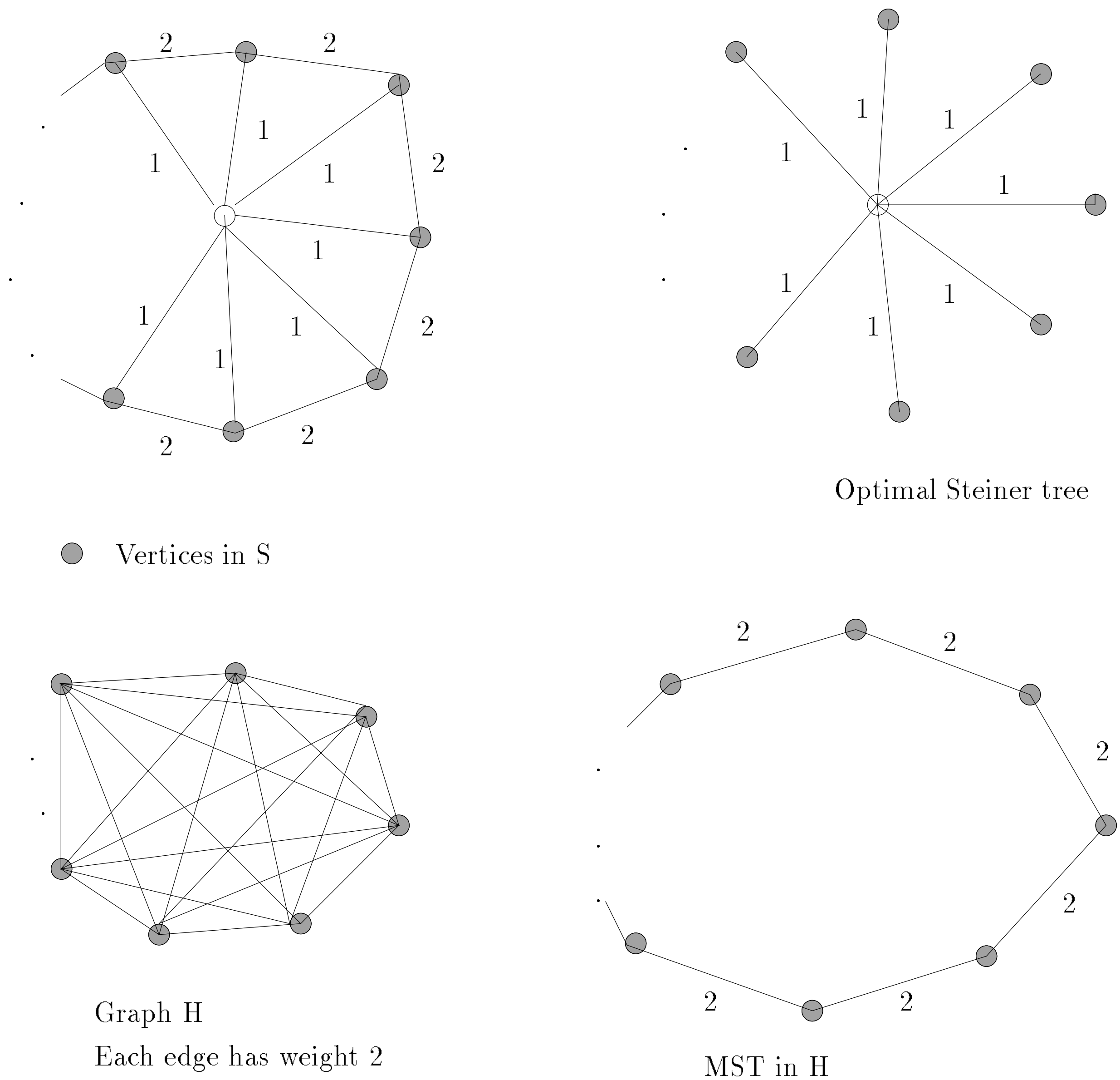


Figure 37: Worst Case ratio is achieved here

27.2 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.

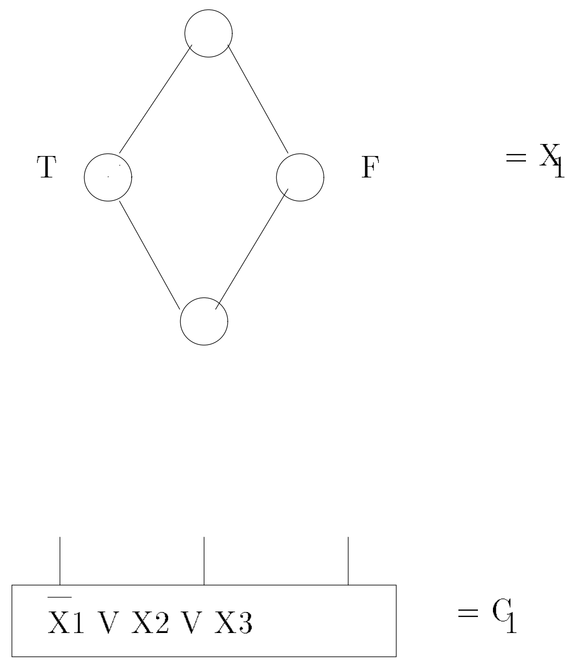


Figure 38: Gadget for a variable

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

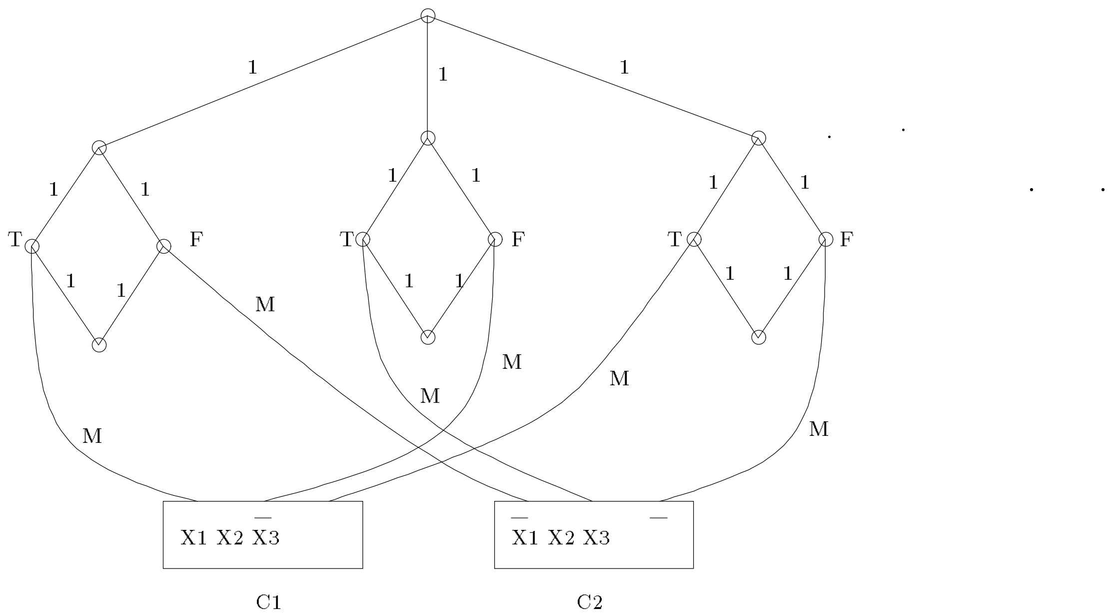


Figure 39: Reduction from 3-SAT.

take T, else take F). Now take the path via that node to connect top and bottom nodes of a variable, this gives $2n$ edges of weight 1.

- If 3-SAT is satisfied then each clause has (at least) 1 literal that is "1", and thus connects to the T or F node chosen in (2) of the corresponding variable. Take this connecting edge in each clause; we thus have m edges of weight M, giving a total weight of mM. So, altogether weight of the Steiner tree is $mM + 3n$.

If there exists a Steiner tree of weight $mM + 3n$ then 3-SAT is satisfiable. To prove this we note that the Steiner tree must have the following properties:

- It must contain the n edges connecting to the root vertex. There are no other edges connected nodes corresponding to different variables.
- It must contain one edge from each clause node, giving a weight of mM. Since M is big, we cannot afford to pick two edges of weight M from a single clause node. If it contains more than one edge from a clause (e.g. 2 edges from one clause) the weight would be $mM + M > mM + 3n$. Thus we must have exactly one edge from each clause in the Steiner tree.
- For each variable must have 2 more edges via the T or the F node.

Now say we set the variables to true according to whether their T or F node is in the Steiner tree (if T then $x = 1$, if F then $x = 1$.) We see that in order to have a Steiner tree of size

$mM + 3n$, all clause nodes must have an edge connected to one of the variables (i.e. to the T or F node of that variable that is in the Steiner tree), and thus a literal that is assigned a "1", making 3-SAT satisfied.

Notes by Chung-Yeung Lee

28 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 approximate algorithm which gives $First - Fit(I) \leq 2OPT(I)$.

28.1 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 exiting non-empty bins. We shall give a simple analysis that shows that $First - Fit(I) \leq 2OPT(I)$. In Fact, we have

Theorem 28.1 *$First - Fit(I) \leq 1.7OPT(I) + C$ 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. Therefore, if c_i denotes the contents in bin i and k is the no of bins used, we have

$$\sum_{i=1}^{k} c_i \geq k/2.$$

Hence,

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

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

28.2 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 28.2 *(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.)

28.3 Approximate Schemes for bin-packing problems

In the 1980's, two approximate schemes were proposed. They are

1. (Vega and Lueker, 1981) $\forall \epsilon > 0$, there exists an Algorithm A_e such that

$$A_e(I) \leq (1+\epsilon)OPT(I) + 1$$

   , where A_e 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_e such that

$$A_e(I) \leq OPT(I) + O(lg^2(OPT(I))$$

   , where A_e runs in time polynomial in n and $1/\epsilon$.(n=total no. of items.) They also guarantee that $A_e(I) \leq (1+\epsilon)OPT(I) + 1$.

3. It is conjectured that the term $O(lg^2(OPT(I))$ can be improved to a constant.

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.

28.3.1 Restricted Bin Packing

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

1. Each item has size $\geq \delta$.
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 $(1 \leq i \leq m)$, with $\sum_{i=1}^{m} n_i = n$.

For constant δ and m, the above can be solved in polynomial time (actually in $O(n+f(m,\delta))$). Our overall stretegy 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 28.3 *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.

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

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

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

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

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

Next, we shall introduce the grouping scheme for RBP. Consider 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

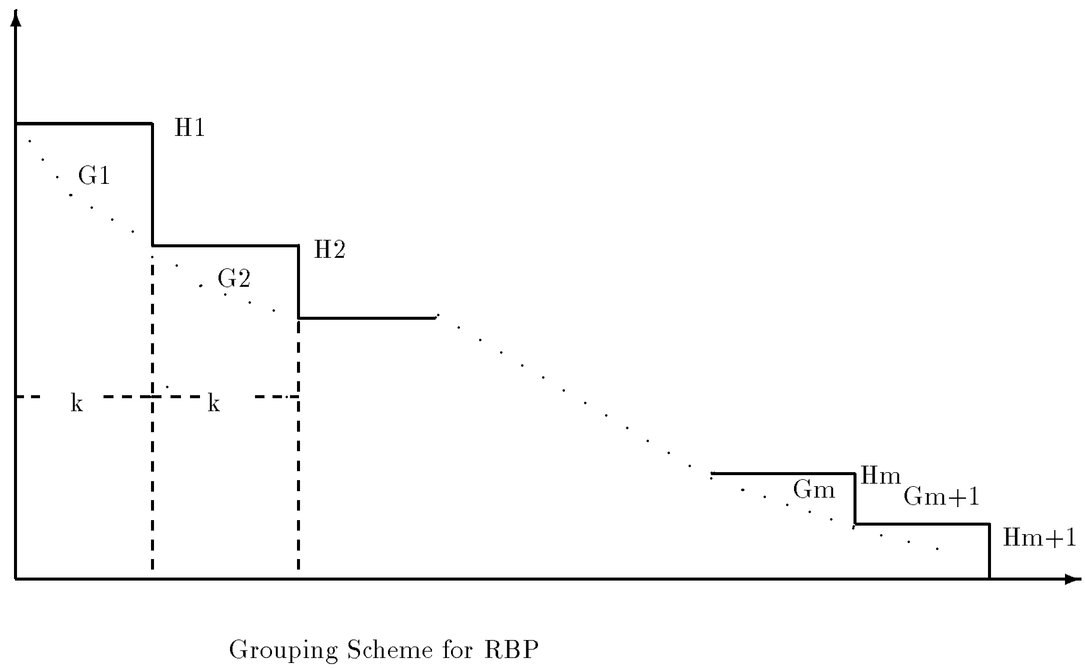


Figure 40: Grouping scheme

- size of any item in $H_i \geq$ size of any items in $G_i \forall i$.
- size of any item in $G_i \geq$ size of any items in $H_{i+1} \forall i$.

The following diagram illustrates the ideas:
We then define J_{low} be the instance consisting of items in $H_2, .., H_{m+1}$ and J_{high} be the instance consists of items in $G_1, H_2, ..., H_{m+1}$. Our goal is to show

$$\mathrm{OPT}(J_{\text{low}}) \leq \mathrm{OPT}(J) \leq \mathrm{OPT}(J_{\text{low}}) + k,$$

The first inequality is trivial, since from $\mathrm{OPT}(J)$ we can always get a solution for J_{low}. We shall continue to prove the other inequality next time.

CMSC 651 Advanced Algorithms	Lecture 27
Lecturer: Samir Khuller	Thu. Apr. 29, 1993

Notes by Gisli R. Hjaltason.

At the end of last lecture, we present the inequality

$$\mathrm{OPT}(J_{\mathrm{low}}) \leq \mathrm{OPT}(J) \leq \mathrm{OPT}(J_{\mathrm{low}}) + k,$$

for the instances J and J_{low} of RBP, with grouping factor k. It remains to prove the latter inequality. Remember that using the $\mathrm{OPT}(J_{\mathrm{low}})$ solution we can pack all the items in $G_2, \ldots, G_{m+1}$ (since we over allocated space forr 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

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

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

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

To summarize, we have:

Theorem 28.4 *Let I be an instance of BP(n), and let J_{low} be the instance of RBP(m, δ), where $\delta = \epsilon/2$ and $m = \lfloor \frac{n}{k} \rfloor$, obtained from I by grouping items in decreasing order of their values into groups of $k = \lfloor \frac{\epsilon^2 n}{2} \rfloor$ items, discarding the first group. The following inequality relates the optimal solutions of the two instances:*

$$\mathrm{OPT}(J_{\mathrm{low}}) \leq (1+\epsilon)\mathrm{OPT}(I) + 1.$$

Now 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 $T^{(1)}, T^{(2)}, \ldots, T^{(n)}$ 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

$$\forall j = 1, \ldots, m, \sum_{i=1}^{p} x_i T_j^{(i)} = n_j;$$

$$\forall i = 1, \ldots, p, x_i \geq 0, x_i integer.$$

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 28.5 *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 procede 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 in theorem 28.4.

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 $(1+\epsilon)\mathrm{OPT}(I)+1$ buckets, which is the bound established in theorem 28.4. 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 polynomial but not fully polynomial.

Karmarkar and Karp came up with a an algorithm for $\mathrm{RBP}(\delta,m)$ that is polynomial in $\frac{1}{\delta}$ and m as well as n. Instead of using integer linear programming, they relaxed the condition of the x_i's being integers, which results in a regular linear programming problem. The optimal solution to the linear programming problem may assign fractional values to the x_i's, but it was shown that by appropriate rounding of the values, a solution close to optimal for the RBP is obtained. Since polynomial algorithms (for example, the ellipsoid algorithm) exist for linear programming, the approximation scheme for BP based on this result is fully polynomial.