CECS 428

Darin Goldstein

1 Amortized analysis: The Pancake Stack

Amortized analysis is the method of determining the *worst case average cost per operation*. Though there are a few different ways of learning amortized analysis, we will focus on the most powerful, the potential method. We assume that we perform n operations numbered 1 through n. We define a potential function $\phi(i)$ (for i between 0 and n) for the data structure[1] that satisfies the following conditions:

1. $\phi(0) = 0$
2. $\phi(i) \geq 0$ for all i

Let $c(i)$ be the actual cost of the ith operation. Then we define the *amortized cost* of the ith operation to be $a(i) = c(i) + \phi(i) - \phi(i-1)$. We now claim that the total amortized cost of all operations is an upper bound for the actual cost. To show this:

$$\sum_{i=1}^{n} a(i) = \sum_{i=1}^{n} c(i) + \sum_{i=1}^{n} (\phi(i) - \phi(i-1)) = \sum_{i=1}^{n} c(i) + \phi(n) - \phi(0) \Rightarrow$$

$$\sum_{i=1}^{n} a(i) \geq \sum_{i=1}^{n} c(i)$$

(Note that because the potential function is defined the way it is, this conclusion is true independent of the value of n.) The potential function intuitively measures how scared you are that you will have to do a bunch of work during the next operation.

1.1 The Quickly Popped Stack

Consider a stack that implements the operations push, pop, and multipop. (Multipop pops all of the elements off the stack all at once.) What is the worst case average amount of time per operation that we do? Define a potential function ϕ that is equal to the number of items stored in the stack. (Verify that

[1] Note that the potential function depends on the configuration of the data structure at time i and *not* the time that has elapsed.

ϕ is legal.) Assume that it takes unit time to push or pop a single item on or off the stack. Let s represent the size of the stack. Then let's analyze the possible operations:

1. Push: $a(i) = c(i) + \Delta\phi = 1 + ((s+1) - s) = 2$
2. Pop: $a(i) = c(i) + \Delta\phi = 1 + ((s-1) - s) = 0$
3. Multipop: $a(i) = c(i) + \Delta\phi = s + (0 - s) = 0$

So it takes constant average worst case time to update this data structure!

2 Amortized analysis: Dynamic tables, Binary counters

2.1 Dynamic tables

Consider database storage. A simple method of allocating memory for an integer database with a size we don't know in advance is to start with a single integer register. In general, when we try to add an integer to the end of the database and discover that we have run out of memory, we simply ask the operating system for double the memory that we are currently using. When the database is $< \frac{1}{4}$ full[2], we cut it in half. That way, our database is always guaranteed to be about half utilized. Of course, if we generate a new table, we have to copy the old values into the new table. So some operations are going to be very expensive, but most do not seem to be...

Assume that it takes unit cost to copy an integer into memory. Let ϕ, the potential of the table, be equal to $2(t - \frac{s}{2})$ if $t \geq \frac{s}{2}$ and $\frac{s}{2} - t$ if $t < \frac{s}{2}$ (where s is the size of the table and t is equal to the number of integers stored in the table). (Verify that ϕ is legal.)

Let's calculate the amortized cost per operation for time i.

1. Assume that we add an integer to the table and it does not cause us to double the table size. Then
$$a(i) = c(i) + \Delta\phi \leq 3$$
2. Assume that we delete an integer from the table and it does not cause us to halve the table size. Then
$$a(i) = c(i) + \Delta\phi \leq 3$$
3. Assume that we add an integer to the table and it does cause us to double the table size. Then the potential of the old data structure was $2(s - \frac{s}{2}) = s$ and the potential of the new data structure is $2(s + 1 - s) = 2$. So we get
$$a(i) = c(i) + \Delta\phi = s + 1 + (2 - s) = 3$$

[2]Why shouldn't we use $\frac{1}{2}$ here? There's a reason. Think about it.

4. Assume that we delete an integer from the table and it does cause a halving of the table size. Then the potential of the old data structure was $(\frac{s}{2} - \frac{s}{4}) = \frac{s}{4}$ and the potential of the new data structure is $\frac{s}{4} - (\frac{s}{4} - 1) = 1$. So we get

$$a(i) = c(i) + \Delta\phi = \frac{s}{4} - 1 + (1 - \frac{s}{4}) = 0$$

So the average worst case effort to update this data structure is constant!

2.2 The binary counter

Assume that we have a binary register initially containing any number and we are allowed to perform the "increment by 1" operation. Assume that it takes unit cost to flip a bit (from 0 to 1 or from 1 to 0). What is the average worst case time to increment the counter n times? Let the potential function ϕ be define to be the number of 1 bits in the register. (Verify that ϕ is legal.) To calculate the amortized cost per increment, let s be the number of 1 low order bits in the register and let t be the total number of 1 bits in the register. Then

$$a(i) = c(i) + \Delta\phi = (s + 1) + ((t - s + 1) - t) = 2$$

So again we get a worst case average constant time per operation!

3 Binomial Heaps

3.1 Application

The Large Hadron Collider (LHC) is the world's largest and most powerful particle collider, built by the European Organization for Nuclear Research (CERN) from 1998 to 2008. Its aim is to allow physicists to test the predictions of different theories of particle physics and high-energy physics, and particularly prove or disprove the existence of the theorized Higgs particle and of the large family of new particles predicted by supersymmetric theories. The Higgs particle was confirmed by data from the LHC in 2013.

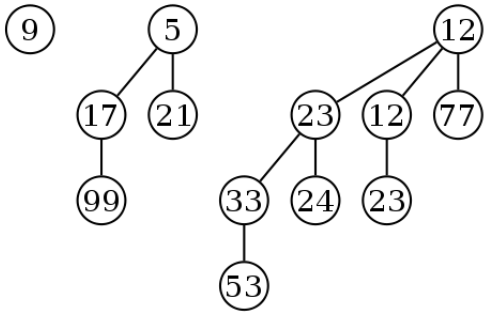
Approximately 600 million times per second, particles collide within the Large Hadron Collider (LHC). By 2012 data from over 300 trillion LHC proton-proton collisions had been analyzed, LHC collision data was being produced at approximately 25 petabytes per year, and the LHC Computing Grid had become the world's largest computing grid (as of 2012), comprising over 170 computing facilities in a worldwide network across 36 countries. The Data Centre processes about one petabyte of data every day - the equivalent of around 210,000 DVDs. The centre hosts 10,000 servers with 90,000 processor cores. Some 6000 changes in the database are performed every second. The Grid runs more than one million jobs per day. At peak rates, 10 gigabytes of data may be transferred from its servers every second.

The basic goal of the grid is to sift through the data produced by the machine and save for later analysis the collision data for those that produce interesting physics.

3.2 The problem

Assume that a person is feeding you distinct numbers (measurements, say) and he will feed you a total of n numbers (where n is large). Every so often, he wants you to output the k smallest numbers you have seen thus far (where $k = o(n)$ is a number he will tell you at that moment) *with the caveat that once you tell him a measurement, he never wants to hear about it again.* If you implement this with a standard heap, you will need to insert all n numbers (for a total time of $\Theta(n \log n)$) and then delete k numbers every so often. Can we do better?

3.3 The structure



A binomial heap B_0 is a single node. B_k is defined to be the heap that occurs when you *link* two B_{k-1} trees together. You choose the root of one of the B_{k-1} trees to be the root of the B_k tree and then make the root of the other B_{k-1} tree the child of the other. (Of course, you can make these max-heaps or min-heaps.) A binomial heap is an ordered collection of binomial trees such that (i) smaller size binomial trees come before larger ones and (ii) there is only a single binomial tree of any given size in a given binomial heap.

We keep track of the B_k-tree that has the minimum/maximum value in it and label it with a *.

There are several operations one can perform on a binomial heap. For example, it is fairly easy to

1. *find the minimum/maximum* node in the heap. Find the heap with the *.
2. *unite* two binomial heaps. To do so, perform the following steps.
   (a) Arrange the binomial trees of each heap together in order of size. (Note that there might be two trees of the same size next to each other when we do so. The next step takes care of this.)
   (b) Starting from the left (the smallest binomial trees first), if there are two trees next to each other with the same size, merge them together by performing the link operation. If you wind up with three binomial trees of the same size, merge the latter two. Continue merging to the right until you are done.

(c) Update the * pointer.

3. *inserting* a node to a binomial heap H. Simply create a single-node binomial heap and unite it with H. Update the * pointer.

4. *decreasing/increasing a key* in a binomial heap. Just decrease/increase the value in the appropriate node in the tree and let it percolate upwards. Update the * pointer if necessary.

5. *delete the node with the minimum/maximum key* from a binomial heap H. To do so, search the tree roots for the minimum/maximum key. Once found, remove that tree from H. Form a new binomial heap H' by simply deleting the root (the minimum node). Now unite H and H'. To *delete an arbitrary key*, just replace the key with the value $\pm\infty$ and let the $\pm\infty$ percolate to the root of the binomial tree. Then delete the node with the minimum key. Update the * pointer as necessary.

We need to show the following things about binomial heaps:

1. The number of nodes in B_k is 2^k.

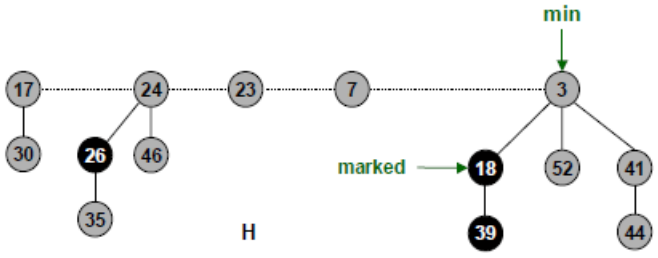
2. The number of binomial trees in a heap of size n is $O(\log n)$.

3. Insert, decrease/increase-key, and delete both take $O(\log n)$ actual time because any arbitrary union operation only takes $O(\log n)$ time.

If we use the potential function $\phi(B) =$ the number of trees in the binomial heap, we can see that insertion takes $O(1)$ amortized time (using essentially the same analysis as the binary counter), an improvement over the $O(\log n)$ regular heap. So all of the insertions in the problem we have above can be done in $O(n)$ amortized time, a great improvement.

4 Fibonacci Heaps I

4.1 Fibonacci heaps



(Fredman and Tarjan 1986) Fibonacci heaps have the following properties: The roots of the trees are kept in a circular doubly linked list. There is a separate pointer that points to the min/max value in the list; in binomial heaps, this was the * node. Each node is either *marked* or not[3]. Each root has a *rank/degree*, defined to be the number of children it has.

Heap type	Find min/max	Insert	Delete min/max	Decrease/Increase key
Regular heap	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Binomial heap	$O(1)$	$O(1)$ amor.	$O(\log n)$	$O(\log n)$
Fibonacci heap	?	?	?	?

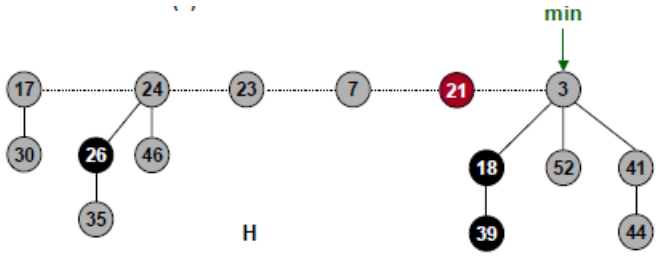
We define a potential function for a Fibonacci heap H to be $\phi(H) = c(t(H) + 2m(H))$ where $t(H)$ represents the number of trees in H, $m(H)$ represents the number of marked nodes in H, and $c > 0$ is a constant to be determined later. We will analyze the amortized cost of each function as we go.

The min/max pointer is updated every time a new tree is added to the circularly linked root list. If a tree with a marked root is added to the circularly linked root list, its root becomes unmarked.

1. To *insert* a node into the heap, create a singleton node and insert it into the doubly linked list to the left of the min pointer (and update the min/max pointer if necessary). Nodes are initially unmarked.

   Amortized cost: $a(i) = c(i) + \Delta\phi = O(1) + c = O(1)$

   Insert the node 21 into the tree to get the following result.

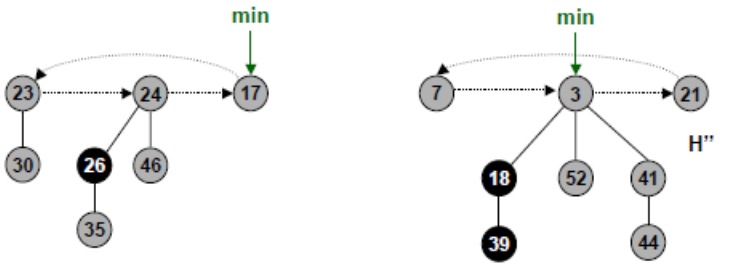


2. To *union* two heaps together, just concatenate the two circularly linked lists together and update the min/max pointer.

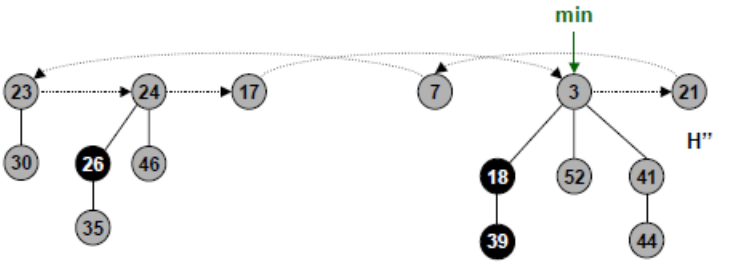
   Amortized cost: $a(i) = c(i) + \Delta\phi = O(1) + 0 = O(1)$

   The union of the following two heaps...

[3]Please be aware that some of the nodes that are marked in the pictures below are not marked correctly.



...is



3. To *delete the min/max*, delete the min/max from the appropriate tree and add the root's children to the tree list. Consolidate the roots of the trees by scanning and merging from left to right so that no two trees have the same degree: To do this, initialize an initially empty array indexed by $0, 1, 2, \ldots$. This array will represent, for a given index i, the tree with root that has i children. Note that there is only allowed to be one of these each at a time. Scan from left to right and, for each root, either the root will be the unique root with its number of children or there will be another already in the array. If there is another already there, make the appropriate (min/max-valued) root a child of the first and continue the check. As you merge, continue to update the min/max pointer.

   Amortized cost: $a(i) = c(i) + \Delta\phi \leq O(D(n) + t(H)) + c(t(H') - t(H))$ where $D(n)$ is the maximum degree of a vertex in a Fibonacci heap of size n (the number of nodes in the heap before the deletion) and H' is the newly created heap structure:

   There is at most $O(D(n))$ work to add the children of the deleted node into the circularly linked list.

   Note that if any nodes become unmarked, then the potential of the data structure goes down; when we calculate the amortized cost, we will therefore ignore any potential root node unmarkings.

   To consolidate the trees, there can be at most $O(D(n)+t(H))$ work: each time two trees are merged together, one of them is never allowed to be

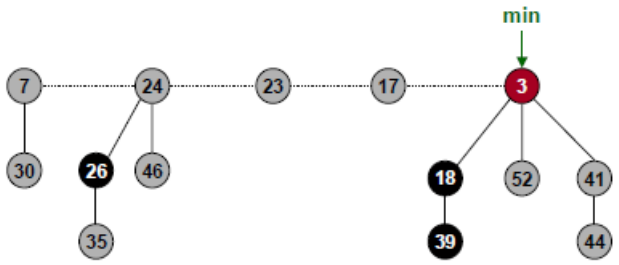
merged again; the size of the circularly linked root list prior to the merge is at most $D(n) + t(H) - 1$.

We claim that $t(H') \leq D(n) + 1$ because no two trees are allowed to have the same degree (and the allowable degrees are $\{0, 1, \ldots, D(n)\}$) so $c(t(H') - t(H)) \leq c(D(n) + 1 - t(H))$. This implies that

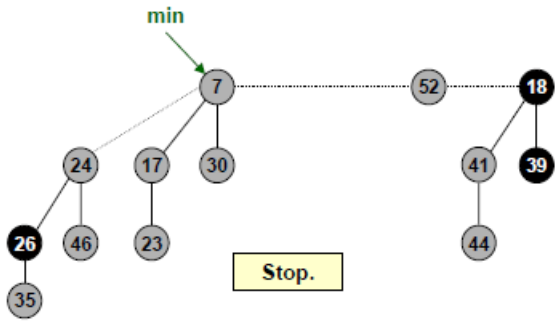
$$a(i) \leq O(D(n)+t(H))+c(t(H')-t(H)) \leq O(D(n)+t(H))+c(D(n)+1-t(H))$$

We will now assume that c is greater than or equal to the constant hidden by O notation so that $a(i) = O(D(n))$. Below, we will figure out a bound on $D(n)$...

Delete the minimum of the following heap...



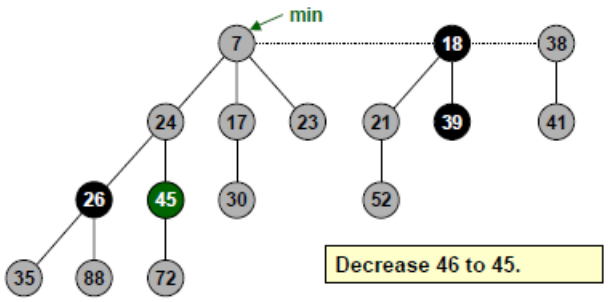
...to get (18 should be unmarked below)



4. To *decrease/increase a given key*, there are 3 cases we need to consider.

   (a) The heap property is not violated. In this case, just decrease/increase the key.

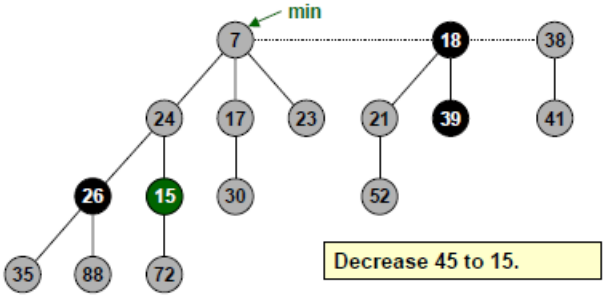
   Amortized cost: $a(i) = c(i) + \Delta\phi = O(1) + 0 = O(1)$



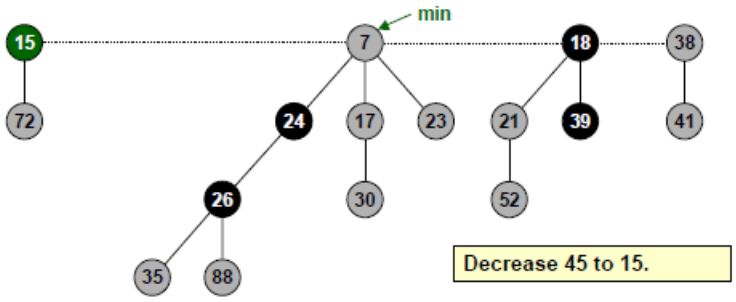
(b) The heap property is violated and the parent of the changed node x is unmarked. Mark x's parent. Cut off the tree rooted at x and add it to the circularly linked root list.

Amortized cost: Once again, we ignore the possibility that x might become unmarked as that will only lower the data structure potential. $a(i) = c(i) + \Delta\phi \leq O(1) + c(1+2) = O(1)$

Consider the following example where the 45 key is decreased to 15. The heap goes from this...



...to this:

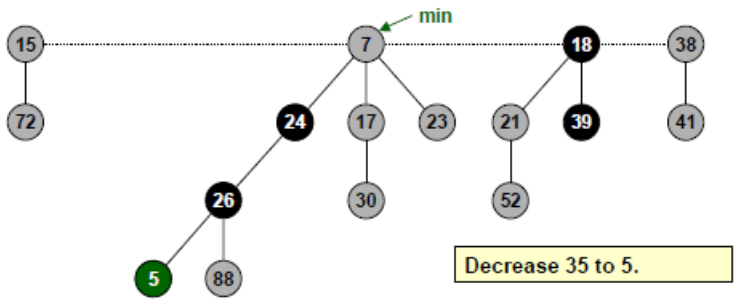


(c) The heap property is violated and the parent of the changed node x is marked. Cut off the tree rooted at x and add it to the circularly linked root list. Let the new value of x be x's former parent $p[x]$. Continue

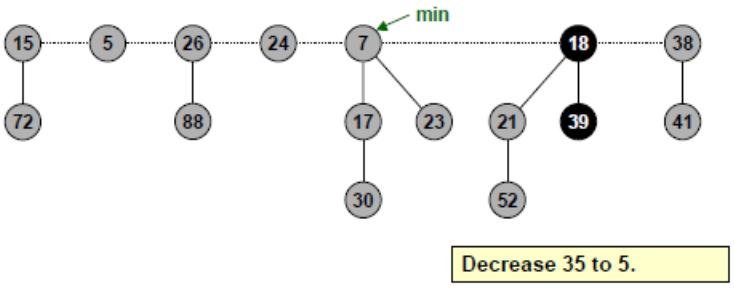
doing the following until x is unmarked: Cut off the tree rooted at x, add it to the circularly linked root list, unmark it (because it becomes a root), and let x become $p[x]$. (We are basically trimming all nodes upward toward the root until we get to one that is not marked.)

Amortized cost: Let r be the number of times we needed to unmark a node. Then $a(i) = c(i) + \Delta\phi = O(r) + cr(1-2)$. Again, we let c be greater than or equal to the constant hidden by the O notation to get $a(i) = O(1)$.

Consider the following example where the 35 key is decreased to 5. The heap goes from this...



...to this:



5. To *delete a key*, decrease/increase its value to $\pm\infty$ and then delete the min/max element.

   Amortized cost: To decrease/increase a key has an amortized cost of $O(1)$ and the amortized cost for delete is $O(D(n))$.

At this point, we can fill in the grid as follows.

Heap type	Find min/max	Insert	Delete min/max	Decrease/Increase key
Regular heap	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Binomial heap	$O(1)$	$O(1)$ amor.	$O(\log n)$	$O(\log n)$
Fibonacci heap (amor.)	$O(1)$	$O(1)$	$O(D(n))$	$O(1)$

5 Fibonacci Heaps II

5.1 Fibonacci rabbits

Assume that there is a breed of immortal rabbits. A mature pair of these rabbits makes a new pair of baby rabbits every month (but it takes the full month to produce them). Baby rabbits mature after a single month. Assume that we go to an initially rabbit-free island and drop a pair of baby rabbits from a helicopter. Let $f(n)$ be the number of pairs of rabbits during the nth month. Then $f(n)$ is the function that represents the **Fibonacci numbers**.

Let $f(0) = 0, f(1) = 1$, and $f(n) = f(n-1) + f(n-2)$. Then

$$f(n) = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^n$$

The proof uses mathematical induction. Before you do anything, find the two solutions to the following equation: $x^2 = x + 1$.

5.2 The final analysis

The goal is now to get some idea of what size $D(n)$ is.

Lemma: Let x be any node in the Fibonacci heap and let $y_1, \ldots, y_k$ be the children of x in the order that they were linked to x. Then degree$(y_i) \geq 0\delta_{i1} + (i-2)(1-\delta_{i1})$ (where δ_{ij} is 1 if $i = j$ and 0 if $i \neq j$).

Proof Obviously, the degree of y_1 is greater than or equal to 0, and the inequality holds if $i = 1$.

Now assume that $i > 1$. Notice that when y_i is linked to x, $y_1, \ldots, y_{i-1}$ are already linked to x. Thus because two nodes are linked only if they have the same degree y_i was linked to x with degree $i-1$. It is a general rule that if two children are removed from a node, then it must become its own root (the first removed child causes the parent to become marked and the next causes it to be removed). Thus at most one child could have been removed from y_i. □

Lemma: In a Fibonacci heap with n nodes, the maximum degree $D(n)$ of any node in any tree is $O(\log n)$.

Proof: Let size(x) be the size of the subtree rooted at x (including the node x itself). Let s_k be the minimum size of a tree rooted at any degree k node; notice that $s_0 = 1$ and $s_1 = 2$. Then we have

$$s_k = 1 + \sum_{i=1}^{k} size(y_i)$$

(where the y_i are the children of the root added in order)

$$1 + \sum_{i=1}^{k} size(y_i) = 2 + \sum_{i=2}^{k} size(y_i) \geq 2 + \sum_{i=2}^{k} s_{deg(y_i)} \geq$$

$$2 + \sum_{i=2}^{k} s_{i-2} = 2 + \sum_{i=0}^{k-2} s_i$$

Recall that f_k is the Fibonacci sequence. Three easy facts to show (in order) are that (i) $f_k = \Omega(\varphi^k)$ (ii) $f_k = 2 + \sum_{i=0}^{k-2} f_i$ for $k \geq 2$ (iii) $\forall k, s_k \geq f_k$. Thus, $s_k = \Omega(\varphi^k)$.

Let x be any node in a Fibonacci heap of size n. Then we have that

$$n \geq size(x) \geq s_{deg(x)} = \Omega(\varphi^{deg(x)})$$

Thus $deg(x) = O(\log_\varphi n)$. But because this was true for any x in any Fibonacci heap of size n, it must also be true of the x with largest possible degree. Thus $D(n) = O(\log n)$. □

5.3 Applications

Consider Prim's algorithm for the minimum spanning tree or Dijkstra's algorithm for single source shortest paths:

1. Maintain a heap for the vertices.
2. Put s in the heap where s is either (i) the start vertex (Dijkstra) or (ii) the initial vertex (Prim) with a key of 0.
3. Repeatedly delete the minimum key in the heap and mark it "scanned." For each neighbor w of the deleted vertex v, if w is not in the heap and not scanned, add it to the heap with key value (i) Dijkstra: $key(v) + |(v, w)|$ or (ii) Prim: $|(v, w)|$. If w is already in the heap, decrease its key value to be the minimum of $key(w)$ and the value given.

Using the classical heap data structure or the binomial heaps, we get a running time of $O(|E| \log |V|)$. Can we do better? Note that decrease key runs very fast. So Dijkstra and Prim's algorithms, given the above, now run in time $O(|E| + |V| \log |V|)$, a major improvement. This is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded nonnegative weights though there are faster known minimum spanning tree algorithms.

Another application of the Fibonacci heap is discrete event simulation. If you are, for example, modeling the action of cars on a network of roads, the best way of doing so is to keep track of the next event that occurs to each car. In other words, a car will continue along its pathway until it either gets near the back of another car or approaches a stop light or stop sign, in which case it will slow down. However, we need not pay attention to the car *until that happens.* Thus, we can calculate using the laws of physics exactly when the next time we need to pay attention to the car is and store that "event" in a heap. The event simulator can just keep track of the next event that needs to be dealt with and assume that everything else is running smoothly. What happens if an event causes new events to occur or old events to be updated such as a new car

appearing on a roadway in front of another car? We can increase a key and/or decrease as necessary to indicate that an event occurs sooner (i.e. that a car needs to slow down sooner than we expect because someone cut him off) or later (i.e. that a car can slow down later than we expect because the car in front turned off the road).

6 Disjoint Sets I

6.1 Description

To represent a set, we simply add a node with the appropriate key value. We let $p[x]$ represent the parent of the node x. If x is a root, then we let $p[x] = x$. For sets, we are really only interested in (i) creating a set (MAKE-SET), (ii) unioning two sets (UNION), and (iii) given an element, determining which set it is in (FIND). (The sets are named using the roots.) We will implement the UNION and FIND operations using two heuristic methods.

Union by rank: The *rank* is an upper bound on the height of a node within the tree: Whenever a new tree is created, the rank of the root is equal to 0. When unioning two sets, if the ranks are different, then the smaller rank becomes a child of the larger rank and no ranks change. When two ranks are equal, we arbitrarily choose one to become the parent and increment its rank by 1.

Path compression: Whenever we conduct a *find* on an element, we make all children on the find path point directly to the root (thereby making the path to the root shorter for all the elements on the path).

- What happens if you use union by rank but not path compression? Build a (non-balanced, non-binary) tree as follows. Starting from a single node, to build stage i, take two copies of stage $i-1$ and union them together. Each time two copies are unioned together, the data structure increases in height by 1. We claim that building a tree with n nodes takes $\Theta(n)$ operations. To see this, note that a recurrence for the number of operations needed to make a n node structure is $T(n) = 2T(n/2)+1 \Rightarrow T(n) = \Theta(n)$.

  Note that we can use the following recurrence to calculate the height of the tree in terms of the number (k) of merge operations: $H(k) = H(k/2) + 1$. If the number of merges we perform is $\Theta(n)$, then the height of the tree is clearly $\Theta(\log_2 n)$ via this recurrence.

  We can now do $m \gg n$ FIND operations on the lowest element to get $\Omega(m \log n)$ behavior.

- What happens if you use path compression but not union by rank? Take the same type of tree as above and call it T_n. Then perform the following actions over and over. Attach T_n to a single node (where the single node becomes the root). Then do a FIND on the lowest node in the resulting structure. Notice that the single node appears as an attachment to the root, but the structure of the rest of the tree remains *exactly the same.*

6.2 Preliminary definitions and lemmas

Let $f^{(i)}(x) = f(f(f(\dots f(x))))$ where the function is applied i times. Define the following functions.

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1 \end{cases}$$

This function grows *extremely* fast and is monotonically increasing with both k and j. Let

$$\alpha(n) = \min\{k : A_k(1) \geq n\}$$

Let's compute $A_4(1)$.

$$A_4(1) = A_3^{(2)}(1) = A_3(A_3(1))$$

$$A_3(1) = A_2^{(2)}(1) = A_2(A_2(1))$$

$$A_2(1) = A_1^{(2)}(1) = A_1(A_1(1))$$

$$A_1(1) = A_0^{(2)}(1) = A_0(A_0(1)) = A_0(2) = 3 \Rightarrow A_2(1) = A_1(3) = 7$$

$$\Rightarrow A_3(1) = A_2(A_2(1)) = A_2(7) = 2047 \Rightarrow A_4(1) = A_3(2047)$$

So how big is this? The Ackermann function grows *much* faster than exponential so this number is larger than 3^{2047}. (The number of particles in the known universe is roughly 10^{82}.) So for all practical purposes $\alpha(n) \leq 4$.

6.3 Applications

1. Equivalence classes: Every time it is found that one element is related to another, union their sets. What would be the data structure you would have to use otherwise? Is it efficient?

2. The offline nearest common ancestor problem in a tree: Given a list of vertex pairs, determine the lowest common ancestor of each pair.

3. The online undirected connected components problem: You are given edges one at a time and told to output the connected components.

4. Minimum spanning tree algorithms: It is easy to tell whether two vertices make a cycle.

5. Fast maze generation

   (a) Create a large array of all the squares in an array.

   (b) Have another array list of all of the adjacent squares in grid and initialize to true (implying is wall).

   (c) Start knocking down walls by randomly choosing from the adjacent square list and making them false (implying no wall).

(d) Once wall knocked down, place two elements into same equivalence class.

(e) Once the start and end element in the same equivalence class - done. Better - all elements in same equivalence class - more detours.

7 Disjoint Sets II

7.1 Analysis

Properties of the rank:

1. The value of $rank[x]$ is initially 0 and increases through time until $x \neq p[x]$; from that point onwards, $rank[x]$ does not change.
2. For all nodes x, $rank[x] \leq rank[p[x]]$ with strict inequality if $x \neq p[x]$. As we follow the path from any node to the root, the node ranks strictly increase.
3. Every node has rank at most $n-1$ where n is the number of elements.

We need to define two functions on nodes x such that x is not a root and $rank[x] \geq 1$.

1. Let the level of a node x be defined to be

$$level(x) = \max\{k : rank[p[x]] \geq A_k(rank[x])\}$$

So $level(x)$ is the greatest k for which A_k when applied to x's rank is no greater than x's parent's rank.

Lemma: $0 \leq level(x) < \alpha(n)$

Proof: Note the following facts:

$$rank[p[x]] \geq rank[x] + 1 = A_0(rank[x]) \Rightarrow 0 \leq level(x)$$

$$A_{\alpha(n)}(rank[x]) \geq A_{\alpha(n)}(1) \geq n > rank[p[x]]$$

$A_y(rank[x])$ is a monotonically increasing function of y and we have by the above that $A_y(rank[x]) > rank[p[x]]$ when $y = \alpha(n)$. Thus, the maximum y for which $A_y(rank[x]) \leq rank[p[x]]$ must be strictly less than $\alpha(n) \Rightarrow level(x) < \alpha(n)$. □

2. The second function is referred to as the iteration.

$$iter(x) = \max\{i : rank[p[x]] \geq A^{(i)}_{level(x)}(rank[x])\}$$

So $iter(x)$ is the largest number of times we can apply $A_{level(x)}$, applied initially to x's rank, before we get to a value greater than x's parent's rank.

Lemma: $1 \leq iter(x) \leq rank[x]$

Proof: Note the following facts:

$$rank[p[x]] \geq A_{level(x)}(rank[x]) = A^{(1)}_{level(x)}(rank[x]) \Rightarrow 1 \leq iter(x)$$

$$A^{(rank[x]+1)}_{level(x)}(rank[x]) = A_{level(x)+1}(rank[x]) > rank[p[x]] \Rightarrow iter(x) \leq rank[x]$$

□

Claim: As long as $level(x)$ remains unchanged, $iter(x)$ must either increase or remain unchanged.

Proof: Assuming that $level(x)$ remains unchanged, when is it possible for $iter(x)$ to decrease? So the goal is to make the inequality false sooner... Either $rank[p[x]]$ must decrease or $rank[x]$ must increase. Because ranks never decrease, $rank[x]$ must increase. However, the rank of a node can only increase if it is a root (i.e. if $p[x] = x$), but $level(x)$ and $iter(x)$ are only defined if x is *not* a root. □

7.2 The potential function

Let the potential of a given node x after q operations be

$$\phi_q(x) = \begin{cases} \alpha(n)rank[x] & \text{if } x \text{ is a root or rank[x]=0} \\ (\alpha(n) - level(x))rank[x] - iter(x) & \text{if } x \text{ is not a root and rank[x]} \geq 1 \end{cases}$$

The potential of the entire data structure is the scaled sum of the potentials of its nodes: $\phi_q = c \sum_{\text{nodes } x} \phi_q(x)$ where c is a positive constant that we will choose later.

Lemma: $0 \leq \phi_q(x) \leq \alpha(n)rank[x]$ for all nodes x.

Proof: Because $level(x) \geq 0$ and $iter(x) \geq 1$, we know that

$$(\alpha(n) - level(x))rank[x] - iter(x) < \alpha(n)rank[x] \Rightarrow \phi_q(x) \leq \alpha(n)rank[x]$$

Because $level < \alpha(n)$ and $iter \leq rank[x]$, we have that

$$(\alpha(n) - level(x))rank[x] - iter(x) \geq 0 \Rightarrow \phi_q(x) \geq 0$$

□

Potential Lemma: Let x be a node that is not a root, and suppose that the qth operation is either a UNION or a FIND. Then

1. $\phi_q(x) \leq \phi_{q-1}(x)$
2. If rank[x]$\geq$ 1 and iter(x) or level(x) changes during the qth operation, then $\phi_q(x) < \phi_{q-1}(x)$.

Proof: Because x is not a root, the qth operation cannot change rank[x].

Note that if rank[x]=0, then $\phi_q(x) = \phi_{q-1}(x) = 0$. So we assume from now on that rank[x]$\geq$ 1.

- Assume that level(x) is unchanged. If both level(x) and iter(x) are unchanged, then $\phi_q(x) = \phi_{q-1}(x)$. If iter(x) increases, then it must increase by at least 1 so $\phi_q(x) < \phi_{q-1}(x)$.

- Assume that level(x) changes. Note that level(x) is monotonically increasing. The term $(\alpha(n) - level(x))rank[x]$ therefore decreases by at least rank[x]. Because level(x) increased, the value of iter(x) might drop but because $1 \leq iter(x) \leq rank[x]$, the drop can be by at most $rank[x] - 1$. Thus, $\phi_q(x) < \phi_{q-1}(x)$ in this case as well.

□

8 Disjoint Sets III

8.0.1 Amortized time per operation

Amortized time to MAKE-SET: $a(q) = c(q) + \Delta\phi = 1 + 0 = O(1)$.

Amortized time to UNION: $c(q) = O(1)$. Assume that the UNION makes node y the parent of node x. The nodes with potentials that may change during the operation are x, y, and x's and y's children prior to the link.

1. x's children: Notice that none of the children of x nor their parent changes rank so no levels nor iters change. Thus, no potentials can change.

2. y's children: By the Potential Lemma, the potential for any node that is not a root cannot increase for a UNION operation. Thus $\Delta\phi \leq 0$ in this case.

3. x: Because x was a root prior to the operation, $\phi_{q-1}(x) = \alpha(n)rank[x]$. If, prior to the operation, rank[x]=0, then clearly the potential does not change because x's rank does not change. Otherwise, x becomes a non-root and $\phi_q(x) = (\alpha(n) - level(x))rank[x] - iter(x) < \alpha(n)rank[x] = \phi_{q-1}(x)$. So x's potential actually decreases; $\Delta\phi \leq 0$ in this case also.

4. y: Because y was a root prior to the operation, $\phi_{q-1}(y) = \alpha(n)rank[y]$. Either y's rank is left alone (leaving a potential change of 0) or it increases by 1 (which increases y's potential by $\alpha(n)$).

Thus, the total potential amortized increase in potential $\Delta\phi$ is $O(\alpha(n))$.

Amortized time to FIND-SET: Assume that there are s nodes on the path up to the root of the set. Then $c(q) = O(s)$.

Note that no node's potential increases due to the cost of the operation (because of the Potential Lemma and the fact that the root's potential doesn't change).

Claim: At least $\max\{0, s - (\alpha(n) + 2)\}$ nodes have their potential decrease by at least 1.

Proof: Let us refer to a node x on the search path that satisfies the following a *special* node: (a) $rank[x] > 0$ and (b) x is followed somewhere higher up on

the path by y such that y is not a root and $level(y) = level(x)$ just before the operation. We claim that there are at most $\alpha(n) + 2$ nodes on the find path that are *not* special: the first node on the find path (if it has rank 0), the last node on the find path (the root because nothing follows it along the path), and the last node w on the path for which $level(w) = k$ (for $k = 0, 1, \ldots, \alpha(n) - 1$).

Notationally, let anything with a prime on it denote its value *after* the FIND-SET operation has been performed and without the prime denotes *before*. Let x be a special node and let y be any node that makes x special.

$$\begin{aligned}
rank[p[y]] &\geq A_{level(y)}(rank[y]) \\
&\quad \text{by the definition of level} \\
&\geq A_{level(y)}(rank[p[x]]) \\
&\quad \text{because } A \text{ and rank are monotonically increasing} \\
&= A_{level(x)}(rank[p[x]]) \\
&\quad \text{because } level(x) = level(y) \\
&\geq A_{level(x)}(A^{(iter(x))}_{level(x)}(rank[x])) \\
&\quad \text{by the definition of iter} \\
&= A^{(iter(x)+1)}_{level(x)}(rank[x])
\end{aligned}$$

We know that $rank[p[y']] = rank[p[x']]$ because x and y will have the same parent after the path compression.

We must have $rank[p[y']] \geq rank[p[y]]$ because nodes must point to the root after the path compression, and $rank[x'] = rank[x]$ because x is not a root.

Thus, after path compression, we will have

$$rank[p[x']] = rank[p[y']] \geq rank[p[y]] \geq A^{(iter(x)+1)}_{level(x)}(rank[x]) = A^{(iter(x)+1)}_{level(x)}(rank[x'])$$

by the above.

Thus, either $iter(x)$ or $level(x)$ must have changed during the qth operation because the values prior to the operation cause a contradiction in the definition of iter. By the Potential Lemma, x's potential must have decreased by at least one. □

Thus, $\Delta\phi \leq -c(s - (\alpha(n) + 2)) \Rightarrow$

$$a(q) = c(q) + \Delta\phi \leq O(s) - c(s - (\alpha(n) + 2))$$

Now we choose c to dominate the constant in the O-notation yielding an asymptotic running time of $O(\alpha(n))$.

9 The good subspace: Simple problems

To take advantage of a good subspace, you might ask the customer who is ordering your algorithm whether they really require the intractable problem to

be solved in its full generality. If the customer can narrow the problem slightly, then it may be that it makes a once intractable problem efficiently solvable. Essentially, you are asking your customer to be input-flexible.

9.1 Clique

9.1.1 Application of the clique problem

In pharmacology, there is a graph based method for searching for a compound that will interact with a specific biochemical target. One of the ways that this is done is to generate a large library of graphs describing known molecules. Then, given a particular target, you generate a *complimentary* graph describing its structure; then you search for graphs in the library that have large cliques in common with the complimentary graph of the target. The resulting list of molecules from the library form a set of likely candidates for a molecule that will produce the desired action by bonding with the target.

9.1.2 Planar cliques

The clique problem asks whether there exists a clique of size k in a given graph $G = (V, E)$.

Definition: A subdivision of a graph is a graph that is formed by subdividing its edges into paths.

Kuratowski's Theorem: A graph is planar if and only if it does not contain a subdivision of a $K_{3,3}$ nor a K_5.

Assume that we restrict our problem to only planar graphs G. Notice that if $k \geq 5$, then Kuratowski's Theorem simply states that the answer is NO. If $k < 5$, then simple searching every subset of k vertices to test for a clique is polynomial in the size of the graph. (There is an algorithm by Hopcroft and Tarjan to test whether any given graph is planar that runs in linear time!)

9.2 The Gray code

The Hamilton cycle question is defined to be: Given a graph G does there exists a simple tour that visits every vertex in G exactly once and returns to its starting point?

Obviously, there is a Hamiltonian cycle in C_n and K_n for any n.

Remember that Q_{n+1} is defined by taking two copies of Q_n and putting a 0 bit on the front of every node in the first copy and a 1 bit on the front of every node in the second copy. Then connect the two copies by connecting all vertices in the first copy with their corresponding vertices in the second copy.

9.2.1 Hamilton circuit application

A **Gray code** is an interesting application of Hamilton circuits. Back in the days, computer equipment wasn't as reliable as it is nowadays. Bits were sometimes measured using analog signals. (In other words, the numbers from 0 to

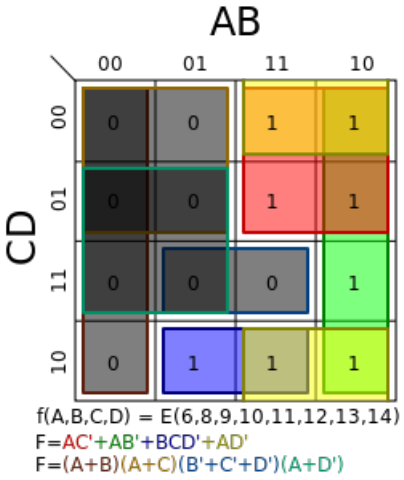
7 would be measured as some number modulo 8.) If we just use the obvious numbering of the analog signals (0 corresponding to 0, 1 corresponding to 1, and so on), then a small error in signal strength might lead to a large error in the bit string representation of the number. For example, do we count the analog signal 3.51 as 3, 011, or 4, 100? Notice that a small error in the signal produces an error in *every* bit of the resulting bit string! Gray came up with a numbering scheme such that a small error in analog signal produces only a single bit of error in the resulting bit string. One possible numbering system is 0,1,3,2,6,7,5,4. Notice that adjacent numbers only differ by one bit. This numbering system corresponds to a Hamilton circuit of the graph Q_3. See if you can determine why...

Claim: A Hamilton cycle in Q_n always exists of $n \geq 2$.

Proof: Induction. The claim is clearly true for $n = 2$. Now assume that there exists a Hamilton cycle in Q_k. Consider the following cycle in Q_{k+1}: Take the Hamilton cycle given by the inductive assumption in the first copy of Q_k but do not traverse the final edge back to the initial vertex; instead, cross the edge that leads to the vertex in the second copy of Q_k. Now take the same path *in reverse* but do not cross the final edge; instead cross back to the first copy of Q_k and you are guaranteed to have made a Hamilton cycle. □

9.2.2 Gray code application

An interesting application of Gray codes are Karnaugh maps for simplifying boolean expressions for the purposes of, for example, designing simple boolean circuits from a potentially complex boolean function. The Karnaugh map (discovered in 1953), also known as the K-map, is a method to simplify boolean algebra expressions. The Karnaugh map reduces the need for extensive calculations by taking advantage of humans' pattern-recognition capability. It also permits the rapid identification and elimination of potential race conditions.



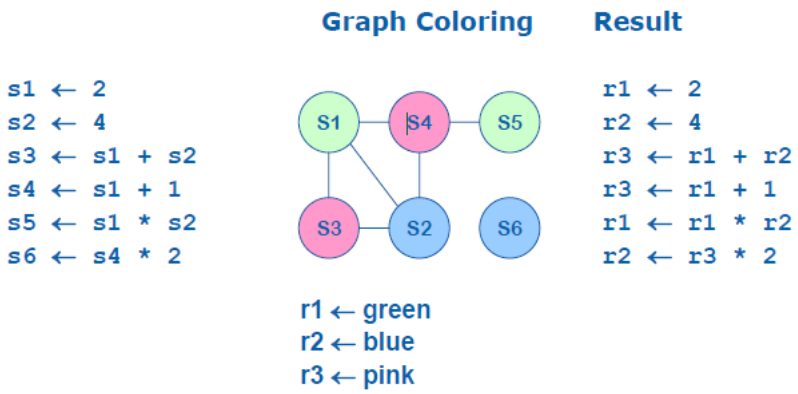
The required boolean results are transferred from a truth table onto a two-dimensional grid where the cells are ordered in Gray code, and each cell represents one combination of input conditions. Optimal groups of 1s or 0s are identified, which represent the terms of a canonical form of the logic in the original truth table. These terms can be used to write a minimal boolean expression representing the required logic.

9.3 Register Allocation

The k-coloring problem is defined as follows: Given an undirected graph G, does there exist a way to color the vertices of G using at most k colors in such a way that the two vertices on every edge are different colors? It is known that the k-coloring problem is NP-complete. (Beware: k is a paramter of the problem and *not* a constant.)

Consider the problem of designing a compiler. A given chip has a certain number of registers that stay "close" to the CPU, say k of them (where k may be around 32). One of the goals of the compiler is to act as a register allocator: an entity that determines which of the program's variables are kept in the registers as the program executes. Note that typically there will be many more variables in a given program than k, but only k can be kept on the CPU at any given time.

When formulated as a coloring problem, each node in the graph represents the live range of a particular value. A variable is defined to be *live* when the variable will be used again before it is overwritten. A *dead* variable is one that either (a) will never be used again or (b) will be overwritten during its next use. A live range is defined as the time when a given temporal variable (defined more thoroughly below) is alive. An edge between two nodes indicates that those two live ranges interfere with each other because their lifetimes overlap. In other words, they are both simultaneously active at some point in time, so they must be assigned to different registers. The resulting graph is thus called an interference graph.



Chaitin (1982) made the following observation: It is possible to solve the register allocation problem if and only if the interference graph is k-colorable. The intuition behind this observation is that we can assign a distinct color to each register; because each temporal variable needs to be assigned a register when it is used, its color is its register assignment. If two adjacent nodes have the same color, then the compiler would be required to store these two variables simultaneously in the same register! If a graph is not k-colorable, then some temporal variables need to spill over into the L1 or L2 cache (or, further, if necessary) and cause a delay when their time comes.

In the top figure above, we start with the code given by the r variables; we then introduce the s variables as temporal variables. (Note that s_5 and s_6 show that we need to introduce a new temporal variable whenever a variable is reassigned.) The variable s_1 is live until s_5 is assigned and after that point, it is no longer used. Thus, s_1 needs to be adjacent to s_2, s_3, s_4. s_2 is live until s_5 as well so it needs to be adjacent to s_3 and s_4. s_3, once assigned, is never used again. s_4 is used once more when assigning s_6 so it is live at the assignment of s_5. Neither s_5 nor s_6, once assigned, will ever be used again.

Even though the problem is likely intractable, there is an obvious fact that helps in certain cases: If there exists a node x in a graph G with fewer than k neighbors, then G is k-colorable if and only if $G - \{x\}$ is k-colorable. This leads to the following algorithm by Chaitin:

- Create a node stack as follows: Choose any node x with fewer than k neighbors. Push x on the stack. Then delete x and all its incident edges.
- Using the stack created above: Pop the next node x from the stack. Assign x a different color than all of the nodes surrounding that have already been colored.

It is clear that inductively, every time a color is assigned, it is a legal color and thus the graph is k-colored legally.

Interesting fact: Most interference graphs are sparse and do *not* look random in that they contain clumps of high-density nodes mixed with low. As of 2005, it was estimated that as many as 95% of all programs written in practice have *chordal* interference graphs; these can be colored optimally in linear time. Even many of the non-chordal graphs respond well to the chordal algorithm for coloring. (An undirected graph is chordal if every cycle with 4 or more nodes has a chord, that is, an edge not part of the cycle that connects two nodes on the cycle.)

10 The good subspace: Satisfiability I

10.1 Introduction

These problems are among the first to be shown to be computationally intractable. The Boolean Satisfiability Problem is as follows: Given a Boolean formula f, does there exist a satisying assignment for f?

An essential circuit design task is to check the functional equivalence of two circuits. Let C_A and C_B denote two circuits, both with inputs $x_1, x_2, \ldots, x_n$ and both with m outputs. The functions implemented by the two circuits are defined as follows: $f_A : \{0,1\}^n \to \{0,1\}^m$ and $f_B : \{0,1\}^n \to \{0,1\}^m$. Let $x \in \{0,1\}^n$ and define $f_A(x) = (f_A^{(1)}(x), f_A^{(2)}(x), \ldots, f_A^{(m)}(x))$ and $f_B(x) = (f_B^{(1)}(x), f_B^{(2)}(x), \ldots, f_B^{(m)}(x))$. Determining whether these two circuits compute the same function is equivalent to determining whether there exists a satisfying assignment to the following boolean formula:

$$\bigvee_{i=1}^{m} (f_A^{(i)}(x) \oplus f_B^{(i)}(x))$$

If there is a way to satisfy this, then there exists a way to make the two formulas different on one of the outputs.

One might think that simplifying the problem to only 3CNF formulas might help: The 3-SAT (3-CNF) Problem is as follows: Given a set of clauses, each of which contains exactly 3 literals, does there exist an assignment to the variables so that at least one literal in each clause is TRUE? Unfortuntately, the 3-SAT problem has also been shown to be intractable.

In the following, we will introduce further specializations that do help.

10.2 2-SAT

2-SAT: Instead of 3 variables per clause, what about 2? Let there be m clauses with variables $x_1, x_2, \ldots, x_n$. Create the graph G in the following way: If there is a clause $A \vee B$, then we place two directed arrows in G, namely $\neg A \to B$ and $\neg B \to A$. (Obviously, these mean that in order to make the entire statement TRUE, if A is FALSE, then B must be TRUE and vice versa.)

10.2.1 Example

$$x_0 \vee x_2, x_0 \vee \neg x_1, x_1 \vee \neg x_2, \neg x_1 \vee \neg x_2, \neg x_0 \vee \neg x_1$$

10.2.2 Algorithm

Claim: For any x_i, if x_i and $\neg x_i$ are in the same connected component, then there cannot be a satisfying assignment for the variables.

Proof: Either x_i must get assigned to TRUE or FALSE. If x_i is assigned to TRUE, then the path from x_i to $\neg x_i$ implies that x_i must be FALSE, a contradiction. If x_i is assigned to FALSE, then then path from $\neg x_i$ to x_i implies that x_i must be TRUE, a contradiction. Thus, x_i cannot be assigned in such a way as to make all implications true and there is no satisfying assignment. □

Claim: If $\forall i, x_i$ and $\neg x_i$ are in different strongly connected components, then there exists a satisfying assignment that can be computed quickly.

Proof: Perform Tarjan's strongly connected component algorithm on G, and then topologically sort the strongly connected components. Choose any strongly

connected component C with out-degree 0 (there must exist one), and let all unassigned literals in C be TRUE; then remove C from the SCC DAG. Continue this until all variables have been assigned.

Is it possible to ever have a path from TRUE to FALSE? Every time we assign a literal to TRUE, is it possible for there to exist a path from that literal to a FALSE one? If not, then we are done.

By the definition of the algorithm and the fact that no variable and its negation can be in the same SCC, we will never simultaneously assign a variable and its negation to the same truth value.

We claim that the algorithm never assigns two literals in the same component to different truth values via this algorithm. It is clearly true after the first TRUE assignment. Assume that it remains true after the kth TRUE assignment. By way of contradiction, assume that the algorithms assigns two literals in the same component to different truth values during the $k+1$th assignment. WLOG say that l_1 is assigned to TRUE and l_2 is assigned to FALSE. Note that by the way the algorithm was constructed, l_2 must have been assigned strictly before l_1. However, if l_1 and l_2 are in the same component, then $\neg l_1$ and $\neg l_2$ are also in the same component. If l_2 was previously assigned to FALSE before l_1, then $\neg l_2$ must previously have been assigned to TRUE before $\neg l_1$. But how could $\neg l_1$ have remained unassigned in the same component as $\neg l_2$ was being assigned to TRUE? The algorithm would not allow it to happen. Thus, it is never possible to assign two literals in the same component to different truth values via this algorithm.

Because we only ever assign TRUE components that point to other TRUE components, we will never get a contradiction.

This algorithm works in linear time! □

11 The good subspace: Satisfiability II

11.1 Linear 3-SAT

Consider a 3-SAT instance with the following special locality property. Suppose there are n variables in the Boolean formula, and that they are numbered $1, 2, \ldots, n$ in such a way that each clause involves variables whose numbers are within ± 10 of each other. Does this restriction allow the problem to be solved more efficiently?

Solution: We will build a function that, given a linear 3-SAT instance with variables x through y (where $x < y$), determines all possible assignments for the variables x through $x+9$ and $y-9$ through y such that there exists a satisfying assignment to the instance given the assignments for these 20 variables. You can think about what gets returned from this function as a list of assignments for 20 variables. There are at most 2^{20} of these total, a constant number.

If $y - x < 200$, just perform a brute force search to determine if top and bottom assignments exist and return them if they do, NULL if they don't.

Define the middle variable to be $m = \lfloor \frac{x+y}{2} \rfloor$. Make the following definitions.

1. Let the set of clauses that contain at least one variable strictly less than $m - 10$ be called s_1.

2. Let the set of clauses that contain at least one variable strictly greater than $m + 10$ be called s_2.

3. Let the set of all other clauses be called s_3.

Note that $s_1 \cup s_2 \cup s_3$ makes up the entire problem instance and also that s_1, s_2, and s_3 are pairwise disjoint. Finally we note that at most 10 variables can overlap between s_1 and s_3 ($m-10$ up to $m-1$) and between s_2 and s_3 ($m+10$ down to $m+1$).

We first claim that the satisfying assignments for the set s_3 can be enumerated in time $O(|s_3|)$. Note that this is the size of the clause and not the number of variables in the clause. The number of variables that can occur in s_3 is bounded above by a constant: 21. Thus, we simply try all 2^{21} possible satisfying assignments and keep track of those that make the entire clause set s_3 TRUE.

We recursively call the function on s_1 and s_2. This yields a solution list for the top 10 and bottom 10 variables only for each of the first and second clause sets. We now match these solutions with the "top and bottom" assignments from the middle that we have already calculated. Even though it seems as if there is a great to deal to check, there is actually at most a constant amount of additional checking to be done. We can now tell which of the assignments of the 20 variables that consist of the top 10 from s_2 and the bottom 10 from s_1 match up to form a possible satisfying assignment for the problem instance as a whole. This list of assignments is then returned.

Let $T(n)$ be the running time for an instance of this problem of size n. Then we get the recursion

$$T(n) = T(|s_1|) + T(|s_2|) + O(|s_3|) + O(1) \Rightarrow$$

$$\exists c' > 0 \text{ such that eventually } T(n) \leq T(|s_1|) + T(|s_2|) + c'|s_3|$$

We claim that $T(n) = O(n) \Rightarrow \exists c > 0$ such that eventually $T(n) \leq cn$. Inductively assume that the result is true for all values strictly less than n. Then we get that

$$T(n) \leq T(|s_1|) + T(|s_2|) + c'|s_3| \leq c|s_1| + c|s_2| + c'|s_3| =$$

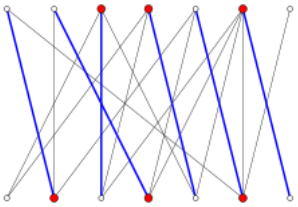
$$cn + (c' - c)|s_3| \leq cn \text{ if } c \geq c'$$

12 The good subspace: Vertex cover in bipartite graphs

In the vertex cover problem, we are given a connected, undirected graph $G = (V, E)$. We would like to find the subset $W \subseteq V$ with the least number of vertices

such that $\{v_1, v_2\} \in E \Rightarrow v_1 \in W \vee v_2 \in W$. A simple application of the vertex cover problem is the placement of police officers in a city or security guards in a museum. You need an authority figure to cover any given street/hallway in the museum. The following will show that if you have some say over the design of the city/museum, you may be able to efficiently guard it using minimal resources.

Konig-Egervary Theorem: In a bipartite, undirected graph, the cardinality of a maximum matching is equal to the minimum cardinality of a vertex cover.



Proof: First, let us observe that the cardinality of *any* vertex cover is greater than or equal to the cardinality of *any* matching. Note that any vertex cover is required to place at least one vertex in every possible match. So we have that the size of the minimum vertex cover must still be greater than or equal to the cardinality of the maximum matching. Thus, it suffices to find a vertex cover of cardinality equal to the maximum matching and we are done.

Let the graph be $G = (V, E)$ and let $L \cup R = V$ such that $L \cap R = \emptyset$ and $\forall \{x, y\} \in E$, x and y are not both in L and not both in R. Create a maximum flow problem as follows: Let edges go from the source to vertices in L and edges go from vertices in R to the sink all have weight 1. Let the weights of the edges E have weight ∞ and point from L to R.

Given a max flow, how do we find a minimum cut? From the source vertex, do a depth-first search along edges in the residual network, and mark all vertices that can be reached this way. The cut consists of all edges that go from a marked to an unmarked vertex. Let the corresponding cut be given by S and T where S represents the vertices on the source side of the minimum cut and T represents the vertices on the sink side.

Let the "potential" vertex cover be

$$(L \cap T) \cup (R \cap S)$$

We now need to prove two separate claims.

Claim: $(L \cap T) \cup (R \cap S)$ forms a vertex cover.

Proof: Let $(x, y) \in E$ be any edge. So $x \in L$ and $y \in R$. Assume that neither x nor y was chosen for the vertex cover. Then $x \in L \Rightarrow x \notin T \Rightarrow x \in S$ and $y \in R \Rightarrow y \notin S$. If $x \in S$, then it is possible to push flow from the source to x. But because every in E has weight ∞, this implies that $y \in S$, a contradiction. $\square$

Claim: $|(L \cap T) \cup (R \cap S)|$ is equal to the size of the maximum matching.

Proof: $|(L \cap T) \cup (R \cap S)|$ is the size of the cut which, by the max-flow, min-cut theorem is equal to the size of the maximum flow which is equal to the size of the maximum matching. $\square$

This completes the proof of the main theorem. □

13 Fixed parameter tractability: 0-1 knapsack problem, Vertex cover problem

To take advantage of fixed paramter tractability, we ask the customer if there are any parameters of the problem that can be considered to be "small" enough so that they can be "essentially" ignored in the running time calculations. For example, if we are given an algorithm that solves a particular problem with two parameters, x and y, and the running time is $O(3^y x^2)$, then this is an exponential time algorithm. However, if the customer declares that all instances for which he is running your algorithm will keep the parameter y to very small values, then your algorithm may be good enough, as it is only exponential in y and not x.

In general, if y is a fixed parameter and x is not, we want a running time that is $O(f(y)x^k)$ for some constant k and some function f (that may be exponential or larger).

13.1 0-1 knapsack problem

A thief robs a bank and can carry a maximum of W pounds in his sack before the sack breaks. Assume that there are n items and that the weights of each item are given by w_i and that each comes with a profit of p_i. What items should we steal?

Note that a recursive relation for $P(i, c)$ (This represents the profit assuming you are only allowed to choose from objects 1 through i and you have capacity c remaining in your sack.) is as follows.

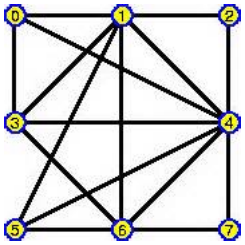
$$P(i,c) = \begin{cases} 0, & \text{if } i = 0 \text{ or } c = 0 \\ P(i-1,c), & \text{if } w_i > c \\ max\{P(i-1,c), P(i-1,c-w_i) + p_i\}, & \text{if } w_i \leq c \text{ and } i > 0 \end{cases}$$

Example:

$Object$	$Weight$	$Profit$
1	3	4
2	5	6
3	5	5
4	1	3
5	4	5

Doing the standard dynamic programming algorithm yields a running time of $O(nW) = O(2^{\log W} n)$. If the value of W is defined to be a "small" parameter, then this becomes a tolerable running time.

13.2 Vertex cover problem



In the vertex cover problem, we are given an undirected graph $G = (V, E)$. We would like to find the subset $W \subseteq V$ with the least number of elements such that $\{v_1, v_2\} \in E \Rightarrow v_1 \in W \vee v_2 \in W$. Assume that you are the president of the company hiring the security guards for the museum and you put an arbitrary cap k on the number of guards you are willing to hire; if the museum requires more guards than this, then the jov becomes unprofitable. We claim that VCP is fixed parameter tractable in k. In other words, if you are guaranteed that your city/museum is guardable by a small force, then it is possible to efficiently determine where they should be placed.

This result isn't obvious. Checking every possible subset of k will yield a running time of at least $\binom{n}{k} = \Omega(n^k)$ and we want $O(f(k)n^{O(1)})$.

VERTEX-COVER(G,C)

1. If $|C| = k$ and G still contains an edge, return $\emptyset$.
2. If G contains no edges, then return C.
3. Pick any edge $\{v, w\}$.
4. Let $G' \equiv G$ with vertex v and all edges incident to v removed.
5. Let $C' \equiv C \cup \{v\}$.
6. If $VERTEX{-}COVER(G', C') \neq \emptyset$, then return $VERTEX{-}COVER(G', C')$.
7. Let $G'' \equiv G$ with vertex w and all edges incident to w removed.
8. Let $C'' \equiv C \cup \{w\}$.
9. If $VERTEX{-}COVER(G'', C'') \neq \emptyset$, then return $VERTEX{-}COVER(G'', C'')$.
10. Return $\emptyset$.

Why does this work? No matter what edge we pick, at least one vertex of the edge must be in the minimal-sized cover; otherwise, there will be an uncovered edge. Essentially, we are checking every possible vertex cover based on this observation. This takes time $O(2^k n)$ because the full recursion tree has depth at most k.

14 Fixed parameter tractability and good subspace: Planar independent set

The independent set problem is: given a planar graph $G = (V, E)$ and a positive integer k, does there exists a set $V' \subseteq V$ such that $|V'| \geq k$ and the vertices in V' are pairwise nonadjacent?

There is an obvious application to scheduling logistics. Let a graph consist of vertices that correspond to jobs that need to be completed. Let there be an edge between any two vertices if and only if the two jobs that the vertices represent require a mutually unsharable resource... Clearly, the maximum number of jobs that can be completed corresponds to the maximum independent set in this graph.

Task	Start	Duration
A	1	5
B	2	2
C	5	3
D	4	7

What happens if we declare that the graph is planar?

Euler's Formula: In any planar graph, $|F| + |V| - |E| = 2$.

Proof: We will use mathematical induction on the number of edges. If the number of edges is 0, then $|V| = |F| = 1$ and the formula is true. Assume that the formula is true for all planar graphs with $k - 1$ edges, and assume that G has $|E| = k$. If G is a tree, then $|V| = k + 1$ and $|F| = 1$ so the formula holds. If G is not a tree, then there is a cycle in G. Choose any edge e in the cycle and remove e from G. This will reduce the number of edges *and faces* of G by exactly 1. By the inductive hypothesis, the formula holds for $G - \{e\}$ and must therefore hold for G as well. □

Claim: In any connected, undirected planar graph with $|V| \geq 3$, $|E| \leq 3|V| - 6$.

Proof: Let's ask the question: In a planar graph G, what is the maximum number of edges that G can possibly have, given a fixed number of vertices such that $|V| \geq 3$. Clearly, if there exists a cycle in G with more than 3 edges, it is possible to add an edge to G without changing the number of vertices; so we can assume that all cycles in G have exactly 3 edges. (Note that this includes cycles that bound the "infinite" face as well.) Thus, the graph with the maximum number of edges given a fixed $|V|$ assumes that all cycles have exactly 3 edges and every face is therefore a triangle.

We can count the number of edge-face pairs (where an edge-face pair is defined to be a pair consisting of an edge and a face where the edge borders the face) in two different ways. Each edge borders exactly 2 faces and each face has exactly 3 edges. Thus, we have $2|E| = 3|F|$. Plugging this into Euler's Formula:

$$\frac{2}{3}|E| + |V| - |E| = 2 \Rightarrow |E| = 3|V| - 6$$

Because this represent the graph with the maximum number of edges given a fixed $|V|$, we must have that $|E| \leq 3|V| - 6$ for every planar graph G. □

Claim: In any connected, undirected planar graph, there exists at least one vertex with degree 5 or less.

Proof: Notice that by the Handshaking Theorem and the above claim, we have that

$$2|E| = \sum_{v \in V} deg(v) \leq 6|V| - 12 \Rightarrow \frac{\sum_{v \in V} deg(v)}{|V|} \leq 6 - \frac{12}{|V|} < 6$$

So the average degree of a planar graph is strictly less than 6. This implies that there must exist some vertex with degree 5 or less. □

Consider the following function for the planar independent set problem on a given graph $G = (V, E)$ with parameter k: $IS(G = (V, E), S)$

1. If G has no edges, then if $|V| + |S| \geq k$ return $V \cup S$ else return $\emptyset$.
2. If $|S| \geq k$, then return S.
3. Let v be the vertex of smallest degree in G. For each $x \in N(v) \cup \{v\}$,
   (a) Construct G' by deleting x and $N(x)$ (and the appropriate edges) from G.
   (b) If $IS(G', S \cup \{x\}) \neq \emptyset$, then return $IS(G', S \cup \{x\})$.
4. Return $\emptyset$.

Pick the vertex $v \in V$ with smallest degree (bounded above by 5, by the above claim). Either v is going into the independent set or one of its at most 5 neighbors will be. In each branching case, delete the corresponding vertex together with all its adjacent edges and vertices from G. Thus, obtain a smaller graph G' in each case, and recursively search for an independent set in each G'. (Obviously, the base cases where the graph is empty or is very small are trivial to handle separately.)

This algorithm must find the maximum cardinality independent set because from the set $\{v\} \cup N(v)$, *exactly* one vertex has to be in an optimal independent set. Since the parameter in each branch decreases by one, we obtain a search tree of size at most 6^k. Using a suitable edge list representation of G, pick a vertex and generating G' can easily be done in linear time in the size of G. Therefore, this algorithm runs in time $O(6^k(|V| + |E|))$ and is fixed parameter tractable in the size of the independent set itself.

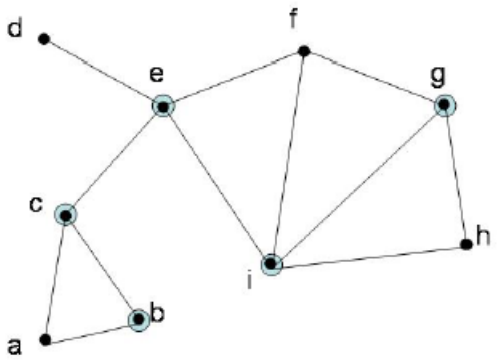
15 Simple approximations

15.1 Vertex cover

The vertex cover problem is as follows: Given an undirected graph $g = (V, E)$, find a minimal-sized subset $C \subseteq V$ such that for every edge $\{v_1, v_2\} \in E$, either $v_1 \in C$ or $v_2 \in C$.

You can think of a *matching* in an undirected graph G as a set of monogamous marriages where two vertices can be married if and only if there exists an edge between them. A *maximal matching* is a matching that cannot be made larger by additional marriages.

It is easy to find a maximal matching. While there are edges remaining to be considered, add any edge to the set of marriages and delete the vertices on the edge and any edges that are attached to those vertices.



Claim: Any maximal matching corresponds to a vertex cover by collecting all married vertices into the cover.

Proof: Is it possible to have an edge that is not covered? If so, that would be a potential marriage. Because the matching is maximal, this is not possible. □

Claim: Any maximal matching is at most twice the size of the optimal vertex cover.

Proof: We will show that any maximal matching is at most twice the size of any vertex cover. Notice that every marriage made by the maximal matching must have at least one representative in any vertex cover. Thus, any vertex cover must have size at least equal to the number of marriages. □

Notice then that the simple greedy algorithm of finding a maximal matching yields a 2-approximation to an extremely difficult problem, vertex cover.

15.2 Scheduling jobs

This is one of the first approximation analyses ever done (Graham, 1966). In this model, there is a set of n jobs and m identical machines. Each job j_i must be processed without interruption for a time $p_i > 0$ on one of the m machines, each of which can process at most one job at a time. How should we schedule the jobs so as to minimize the time spent in the factory?

15.2.1 Example

With 3 machines, schedule the following jobs: 5, 11, 7, 3, 6, 13, 9, 20, 1, 4

15.2.2 Analysis

Consider the following algorithm called *list scheduling* (or LS, for short). We are given a list of the jobs in some arbitrary order. Whenever a machine becomes available, the next job on the list is immediately sent to that machine for processing.

Define the starting time of job j_i using this algorithm as s_i for $i \in [1, n]$. Let the job with the latest completion time be j_x. Then no machine can be idle at any time prior to s_x; otherwise, j_x would have been scheduled there.

Define the completion time of the list scheduling algorithm to be C^{LS} and define the latest completion time of the optimal scheduling to be C^{OPT}. Note the following.

- Note that $C^{OPT} \geq p_x$ because job j_x must complete on some machine.
- Note that $C^{OPT} \geq \frac{1}{m} \sum_{j=1}^{n} p_j$ because there must be at least one machine that takes at least as long as the average machine processing time.
- As we observed above, the start time of j_x must occur at the earliest possible moment. If we remove j_x from the job list, then the average machine time spent on jobs *other than* j_x is $\frac{1}{m} \sum_{i \neq x} p_i$. So some machine must be idle at least as early as that; hence, $s_x \leq \frac{1}{m} \sum_{i \neq x} p_i$.

Now we notice the following string of implications:

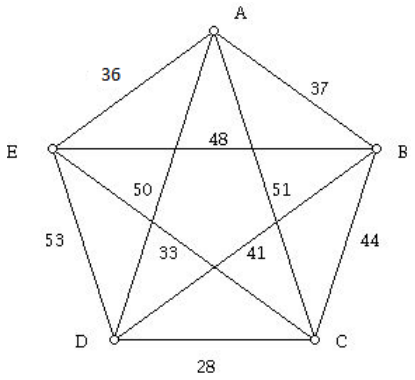
$$C^{LS} = s_x + p_x \leq \frac{1}{m} \sum_{i \neq x} p_i + p_x$$

$$= \frac{1}{m} \sum_{i=1}^{n} p_i + (1 - \frac{1}{m}) p_x$$

$$\leq C^{OPT} + (1 - \frac{1}{m}) C^{OPT} = (2 - \frac{1}{m}) C^{OPT}$$

Thus, list scheduling does at least as well as $2 - \frac{1}{m}$ times the best possible schedule.

16 Approximations: Metric traveling salesman problem



The traveling salesman problem is as follows: A salesman want to visit each of n cities exactly once each, minimizing total distance traveled, and return to the starting point. (Distances between each pair of cities are defined and nonnegative but may theoretically be ∞.)

The traveling salesman problem is known to be NP-complete.

The metric traveling salesman problem is the same as the traveling salesman problem with one additional caveat: The distances involved must obey the metric rules. In other words, (a) $d(c, c) = 0$, (b) for any two cities c_1, c_2, we must have $d(c_1, c_2) = d(c_2, c_1)$, (c) for any three cities c_1, c_2, c_3, we must have that $d(c_1, c_3) \leq d(c_1, c_2) + d(c_2, c_3)$ (triangle inequality).

Consider the following algorithm for solving the MTSP. Create a minimum spanning tree for the graph. Starting at any arbitrary city, perform a DFS of the tree. Any time a city is visited for the first time place it on a list. Once the list is complete, place the original city on the end of the list and the list is now your tour!

Claim: The length of the tour is less than or equal to twice the weight of the minimum spanning tree.

Proof: Consider an ant that moves along the DFS and the distance he travels. If he travels from c_1 to c_2 along the MST/DFS path, then by the triangle inequality, it is clearly at least as efficient to travel directly from c_1 to c_2 (i.e. the tour's path). Thus, the tour has length less than or equal to the total DFS distance which is equal to twice the weight of the MST. □

Claim: The optimal tour distance is greater than or equal to the weight of the minimum spanning tree.

Proof: Consider the optimal tour itself. Deleting a single edge yields a spanning tree which must have weight at least equal to the minimum spanning tree. □

Putting these two claims together shows that the simple greedy algorithm yields a 2-approximation to the MTSP.

17 Approximations: Bin-packing

17.1 Bin packing

You are given a sequence of n items $a_1, a_2, \ldots, a_n$ each with a size $s(a_i) \in (0, 1]$ and are asked to pack them into a minimum number of unit-capacity bins. (Applications: loading trucks subject to weight limitations, packing television commercials into the breaks, cutting the correct amount of some material, say lumber or paper, so as to create a given set of objects)

17.1.1 Example

Consider the following weights: .1, .1, .2, .2, .3, .4, .5, .6, .7, .9

The optimal packing is: .1, .9 : .3, .7 : .4, .6 : .1, .2, .2, .5

17.1.2 Next fit

One of the simplest algorithms for packing is called Next Fit (NF). The NF algorithm functions as follows: If the item to be packed fits into the currently open bin, then pack it in and go on to the next item. Otherwise, close the current bin and open a new bin.

For a given bin-packing algorithm A, let $B(A)$ be defined to be the number of bins that algorithm A uses (assuming the problem instance is clear).

Notice that with the possible exception of the final open bin, if we consider the usage level of two consecutive bins, the average amount of utilized space per bin must be strictly more than one-half via NF. Otherwise, we would have packed the two bins into a single. Thus, defining l_i to be the level of the ith bin utilitized by the NF algorithm,

$$\forall j \in [1..\lfloor B(NF)/2 \rfloor], l_{2j-1} + l_{2j} > 1$$

If we sum all of these equations together, we get that

$$\sum_{i=1}^{2\lfloor B(NF)/2 \rfloor} l_i > \lfloor \frac{B(NF)}{2} \rfloor \Rightarrow$$

The right side is an integer so we get

$$\lceil \sum_{i=1}^{2\lfloor B(NF)/2 \rfloor} l_i \rceil - 1 \geq \lfloor \frac{B(NF)}{2} \rfloor \Rightarrow$$

Notice that $B(OPT) \geq \lceil \sum_i w_i \rceil = \lceil \sum_{i=1}^{B(NF)} l_i \rceil$. The sum of all of the weights is equal to the sum of the levels of all the bins used in the NF algorithm. So we get

$$B(OPT) - 1 \geq \lceil \sum_{i=1}^{B(NF)} l_i \rceil - 1 \geq \lceil \sum_{i=1}^{2\lfloor B(NF)/2 \rfloor} l_i \rceil - 1 \geq \lfloor \frac{B(NF)}{2} \rfloor \geq \frac{B(NF) - 1}{2}$$

Thus, we must have that $B(NF) \leq 2B(OPT) - 1$.

Consider the following sequence of weights (where N is a very large number):

$$(\frac{1}{N+1}, 1 - \frac{1}{2(N+1)}, \frac{1}{N+1}, 1 - \frac{1}{2(N+1)}, \ldots, \frac{1}{N+1}, 1 - \frac{1}{2(N+1)}, \frac{1}{N+1})$$

where the $\frac{1}{N+1}$ is repeated $N+1$ times and the $1 - \frac{1}{2(N+1)}$ is repeated only N times. The optimal arrangement would be to pack all of the small items into the same bin and the rest of the items into their own separate bins for a total of $N+1$ total bins. NF puts every item into its own bin for a total of $2N+1$. Thus, we get that

$$B(NF) = 2N + 1 = 2(N+1) - 1 = 2B(OPT) - 1$$

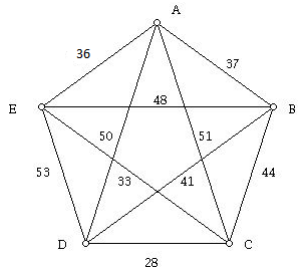
Combining this result with the one above, $B(NF) = 2B(OPT) - 1$.

18 Intermediate approximations: Metric k-center problem

18.1 Definitions

The metric k-center problem is defined on the complete undirected graph $K_n = G = (V, E)$. Let the shortest distance between vertices v_i and v_j in G be d_{ij}. The problem is to find a subset of the vertices S such that $|S| = k$ and so that the longest distance of a node from the nearest node in S is minimized. So imagine each node in the complete graph is a city and edges represent the shortest distance between these cities. We have funds to build exactly k emergency centers. The k-center problem with triangle inequality is to place our k emergency centers such that no one has to go too far to get to their closest center. More formally, the goal is to minimize over all subsets $S \subseteq V$ of size k the following expression:

$$\max_{j \in V} \min_{i \in S} d_{ij}$$



Because we are doing the *metric* k-center problem, we assume that (a) $d_{ii} = 0$ (b) $d_{ij} = d_{ji}$ and (c) $d_{ij} \leq d_{ik} + d_{kj}$.

The metric k-center problem is NP-hard (Kariv and Hakimi, 1979). Applications of this include placement of supply centers for the military, placement of hospitals, data backups, etc.

The *square* of a graph $G = (V, E)$ is defined to be $G^2 = (V, E^2)$ where there exists an edge between two vertices x and y in G^2 if and only if there exists a path of length 1 or 2 between x and y in G.

An *independent set* of a graph is defined to be a set of vertices none of which shares a mutual edge with any other vertex in the set. A *dominating set* of a graph is defined to be a set of vertices such that every vertex in the graph is either in the dominating set or shares an edge with a vertex in the dominating set.

18.2 Algorithm

Consider the following algorithm: Initially, think of the cities as having no edges between them. Order the edges from least weight to most so that $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$ where $m = \binom{n}{2}$. We will add these edges in order to the empty graph, and, at some point, we will have a dominating set of k vertices. After this point, we have our k centers and adding heavier edges will not help: To see why, assume that we have a dominating set of k vertices and we are thinking about adding edge e_i to the graph. There is already a way to get from any city to an emergency center in time at most $w(e_{i-1})$ because the k centers form a dominating set; thus, adding an additional edge of higher weight gains us nothing.

Our problem is that the dominating set problem (the problem of finding a dominating set of size k or smaller in an arbitrary undirected graph) is also NP-complete. This is a problem...

Define G_i to be the graph that consists of adding edges $e_1, \ldots, e_i$ to the set of cities devoid of edges. Now, we can use the following algorithm: Starting from $i = 1, 2, \ldots$, compute a maximal independent set (not a *maximum* independent set, which is NP-hard to compute) L_i for the graph G_i^2. As soon as $|L_i| \leq k$, call the set of vertices in L_i the k centers (and if there are strictly less than k, choose any artbitrary others until there are k).

Let the index of the graph that gets returned by the algorithm above be x (so that G_x^2 is the first graph for which we find a maximal independent set of size less than or equal to k).

(When doing the example, it is best to choose A, D, B to be the maximal independent set until you can't any longer.)

18.3 Analysis

Claim: Let $H = (V, E)$ be an undirected graph, and let I^2 be an independent set of H^2. Let $dom(H)$ denote a minimum cardinality dominating set in H. (Note that $dom(H)$ is NP-hard to compute.) Then $|I^2| \leq |dom(H)|$.

Proof: Let $dom(H)$ be a minimum cardinality dominating set in H. For any vertex $d \in dom(H)$, its neighborhood in H forms a clique in H^2. Thus, H^2

contains at most $|dom(H)|$ cliques spanning all the vertices. But any independent set in H^2 can contain only at most one vertex from any given clique. Thus $|I^2| \leq |dom(H)|$. □

19 Intermediate approximations: Metric k-center problem II

Claim: Let c^* be the cost of an optimal solution to a given instance of the k-center problem on $G = (V, E)$. Then $w(e_x) \leq c^*$.

Proof: Note that it is not possible for every edge in G to have strictly smaller weight than c^*. If so, then there would need to be some node for which the fastest way to the nearest center is not via the edge directly there. However, this violates the triangle inequality.

Let $x' + 1$ be the smallest value such that $w(e_{x'+1}) > c^*$. (Notice that this value must exist because of the above.) If we can show that $x \leq x'$, then we are done because the edges are being added in increasing order. We only need to check that *any* maximal independent set in $G^2_{x'}$ has size less than or equal to k; if this is true, then because x was defined as being the first index where we find G^2_x with an independent set of size $\leq k$, it must have come at least as early as x'.

We claim that $G_{x'}$ has a dominating set with size $\leq k$. Remember that the shortest path from a node to its corresponding center must be via the edge directly there by the triangle inequality. If we have already added edges at least as large as the optimal answer c^* (by the definition of x'), then every node has its optimal direct route to its optimal center and there are exactly k centers.

By the above lemma, we have that for any independent set I^2 of $G^2_{x'}$, $|I^2| \leq |dom(G^2_{x'})| \leq k$. □

Claim: The above algorithm returns a solution with cost at most twice the optimal.

Proof: Note that any maximal independent set in a graph is necessarily a dominating set. So the maximal independent set L_x that was returned by the algorithm is a dominating set in the graph G^2_x. This implies that every vertex in G_x is at most 2 edges away from a center. But by the above lemma, $w(e_x) \leq c^*$ which implies that every edge in G_x is less than or equal to c^*.

So consider the worst case distance from a given vertex v to its center. With our solution, at most two edges need to be traversed, both of which are of length at most c^*; this implies that the worst case distance our solution yields is at most $2c^*$ and thus our algorithm has at most twice the optimal cost. □

19.1 How bad can it get?

Consider the following situation (fixing some small $\epsilon > 0$) where $k = 2$: Let there be a vertex on the left called L and a vertex on the right called R. (These will be the two spots where the facilities *should* be located by an optimal algorithm.)

In the middle, let there be a line of 4 nodes down the center labeled from 1 to 4 (from top to bottom). Let $d(i, R) = 1$ for $i = 1, 2, 3, 4$, let $d(i, L) = 1$ for $i = 1, 2, 3$ and $d(4, L) = 2 - 2\epsilon$. For $i = 1, 2, 3$, let $d(i, i + 1) = 1 - \epsilon$. Let all other distances between distinct nodes be $2 - 2\epsilon$. We claim that this is a metric problem; verify this.

The first 3 edges chosen will be the edges $\{1, 2\}, \{2, 3\}, \{3, 4\}$. Then, we can choose to connect (using an edge of weight 1) either L or R to any vertex in $\{1, 2, 3, 4\}$. WLOG, let us assume that we choose to connect L to 1. Once we do this, we have a dominating set of size 2 in G^2, namely vertices 2 and R. Thus, our algorithm forces us to place our facilities on those vertices. Note that the worst case distance to vertex 4 is $2 - 2\epsilon$, but if the vertices L and R were chosen as facility centers, the furthest distance to any node would be only 1. As $\epsilon \to 0$, this example shows that it is possible for the algorithm described above to approach a worst-case ratio of twice optimal.

20 Randomized approximations: Maximum satisfiability

The Maximum Satisfiability problem (MAXSAT) is the problem where you are given a list of clauses (each clause is a list of literals) and the goal is to return the largest possible number of simultaneously satisfied clauses. (We assume that there are no clauses where a variable and its negation both appear and that a literal appears at most once in any clause.) To solve this problem exactly is NP-complete.

20.1 Example

$$x_1, x_2, \neg x_3 : x_3, \neg x_2 : \neg x_1 : \neg x_3, x_1, \neg x_2, x_4 : x_4 : \neg x_1, \neg x_2, \neg x_3, \neg x_4$$

20.2 Random assignments

Assume that you are given an instance of MAXSAT with n clauses such that every clause has at least k literals. Assign every variable to true with probability $\frac{1}{2}$. Then the probability that a given clause is satisfied is at least $\alpha_k \equiv 1 - 2^{-k}$ and the expected number of satisfied clauses is at least $\alpha_k n = n(1 - 2^{-k})$.

If α_k is large for every clause, then this algorithm functions very well. This happens when k is large. Unfortunately, if there are lots of clauses with a single literal, this yields a performance ratio of only $\frac{1}{2}$.

20.3 Randomized rounding

Lemma: $\forall x \in [0, 1], 1 - \frac{1}{4^x} \geq \frac{3}{4}x$

Proof: Let $f(x) = 1 - \frac{1}{4^x} - \frac{3}{4}x \Rightarrow f'(x) = \frac{\ln 4}{4^x} - \frac{3}{4} \Rightarrow f''(x) = -\frac{(\ln 4)^2}{4^x}$. So because the second derivative is everywhere negative, $f(x)$ is concave down

everywhere. Notice that $f(0) = f(1) = 0$ and $f'(0) > 0$. Thus $f(x) \geq 0$ when $x \in [0, 1]$. □

Lemma: There exists a function $f(x)$ such that $1 - 4^{-x} \leq f(x) \leq 4^{x-1}$.

Proof: All we need to do is show that $\forall x, 1 - 4^{-x} \leq 4^{x-1}$. Let

$$f(x) = 4^{x-1} - (1 - 4^{-x}) \Rightarrow$$

$$f'(x) = 4^{-x-1}(16^x - 4)\ln 4 \Rightarrow$$

$$f''(x) = 4^{-x-1}(16^x + 4)(\ln 4)^2$$

Because the second derivative is everywhere positive, $f(x)$ is concave up. So the function reaches a minimum where the first derivative is equal to 0. This occurs when $x = \frac{1}{2} \Rightarrow f(\frac{1}{2}) = 0$ is a global minimum of this function. Hence $f(x) \geq 0$. □

For the rest of this section, by the above lemma, we can fix some function $f(x)$ such that $1 - 4^{-x} \leq f(x) \leq 4^{x-1}$ everywhere.

Consider the following linear program (where S_r^+ represent the set of positive variables in clause r and similarly for S_r^-; we will assume that $S_r^+ \cap S_r^- = \emptyset$).

Maximize $\sum_{i=1}^{n} z_i$ subject to

1. $\forall r \in [1, n], \sum_{x_i \in S_r^+} x_i + \sum_{x_i \in S_r^-} (1 - x_i) \geq z_r$
2. $\forall r \in [1, m], 0 \leq x_r \leq 1$ and $\forall r \in [1, n], 0 \leq z_r \leq 1$

Randomized rounding will independently set the variable x_i to TRUE with probability equal to the value of $f(x_i)$ that comes out of the linear program.

Lemma: The sum $\sum_{j=1}^{n} z_j$ is an upper bound for the number of satisfied clauses, $C_{max} \Rightarrow \sum_{j=1}^{n} z_j \geq C_{max}$.

Proof: Consider the following linear program: Maximize $\sum_{i=1}^{n} z_i$ subject to

1. $\forall r \in [1, n], \sum_{x_i \in S_r^+} x_i + \sum_{x_i \in S_r^-} (1 - x_i) \geq z_r$
2. $\forall r \in [1, m], x_r \in \{0, 1\}$ and $\forall r \in [1, n], z_r \in \{0, 1\}$

Note that this linear program is strictly more constrained than the program above so that the maximum found in the linear program above will be at least as large as the result in this one. Each z_r value now strictly is 1 if there exists a true literal within clause r and 0 otherwise. It is clear that summing the z values in this linear program results in *exactly* the maximum number of satisfiable clauses, the optimal possible value. □

Now fix some clause c_r. The probability that c_r is *not satisfied* is equal to

$$\prod_{x_i \in S_r^+} (1 - f(x_i)) \prod_{x_i \in S_r^-} f(x_i) \leq \prod_{x_i \in S_r^+} 4^{-x_i} \prod_{x_i \in S_r^-} 4^{x_i - 1} =$$

$$4^{-(\sum_{x_i \in S_r^+} x_i + \sum_{x_i \in S_r^-} (1 - x_i))}$$

Now notice the restriction we put on the rth clause: $\sum_{x_i \in S_r^+} x_i + \sum_{x_i \in S_r^-} (1 - x_i) \geq z_r$. Thus,

$$4^{-(\sum_{x_i \in S_r^+} x_i + \sum_{x_i \in S_r^-} (1-x_i))} \leq 4^{-z_i}$$

So the expected number of satisfied clauses is equal to $E[\sum_{r=1}^{n} I_r]$ where I_r is the indicator random variable that indicates whether the rth clause is satisfied. We get

$$E[\sum_{r=1}^{n} I_r] = \sum_{r=1}^{n} E[I_r] = \sum_{r=1}^{n} P(c_r \text{ satisfied}) \geq$$

$$\sum_{r=1}^{n} 1 - 4^{-z_i} \geq \frac{3}{4} \sum_{r=1}^{n} z_r \geq \frac{3}{4} C_{max}$$

So we get an approximation ratio of $\frac{3}{4}$ via randomized rounding.

20.4 Randomized rounding: the worst case

Consider the following instance of MAXSAT:

$$x_1 \vee x_2, \neg x_1 \vee x_2, x_1 \vee \neg x_2, \neg x_1 \vee \neg x_2$$

Notice that, no matter what we assign to x_1 and x_2, 3 out of the 4 clauses will be TRUE. This transforms into the following linear program: Maximize $\sum_{i=1}^{4} z_i$ subject to

1. $x_1 + x_2 \geq z_1$
2. $(1 - x_1) + x_2 \geq z_2$
3. $x_1 + (1 - x_2) \geq z_3$
4. $(1 - x_1) + (1 - x_2) \geq 4$
5. $0 \leq x_1, x_2 \leq 1$
6. $0 \leq z_1, z_2, z_4, z_4 \leq 1$

By letting $x_1 = x_2 = \frac{1}{2}$, we can let $z_1 = z_2 = z_3 = z_4 = 1$ which is clearly maximal. This is a case, therefore, where the $\frac{3}{4}$ bound is tight.

21 Simple Las Vegas Algorithms: Randomized quicksort

A *Las Vegas* algorithm is an algorithm that uses probability. It will always get the right answer but, depending on the randomness, may take a long time to terminate. These algorithms are sometime known as Robin Hood algorithms because they take from the rich (longer running times) and give to the poor (shorter running times).

Consider quicksort on any instance using a pivot chosen randomly (i.e. assuming all permutations of n distinct numbers are equally likely). We know that the worst case happens when we get back luck and the pivot is chosen as the largest or smallest number remaining in the list and that leads to a running time of $O(n^2)$. But now let's figure out what happens in the average case.

Let X_{ij} be the random indicator variable that counts the number of times x_i and x_j get compared. Note that two elements only get compared when one of them is the pivot, and afterwards, because the pivot is in the correct position, they can never be compared again. Therefore, X_{ij} is an indicator variable for every i and j.

The total number of comparisons during the quicksort algorithm is

$$X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij} \Rightarrow E[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} E[X_{ij}]$$

So assume that $i < j$. Denote the ith smallest element in the array by e_i and the jth smallest element by e_j. If the pivot we choose is between e_i and e_j, then these two will end up in difference sublists and wil never get compared. If the pivot is *exactly* e_i or e_j, then we do compare them. Finally, if the pivot chosen is either larger or smaller than both, then they both end up in the same sublist and we will need to pick another pivot.

This is a kind of dart game for each i and j. If the dart lands between them, you win. If the dart hits one of them, you lose. If the dart lands outside the target, you shoot again. There are $j - i + 1$ numbers between i and j so the probability that you lose the game is $\frac{2}{j-i+1}$ (because you must hit the target eventually). So $E[X_{ij}] = \frac{2}{j-i+1} \Rightarrow$

$$E[X] = \sum_{i=1}^{n} 2(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots + \frac{1}{n-i+1})$$

Let us know consider the integral $\int_1^n \frac{1}{x} dx = \ln n$. If we estimate this integral by rectangles, then we get that

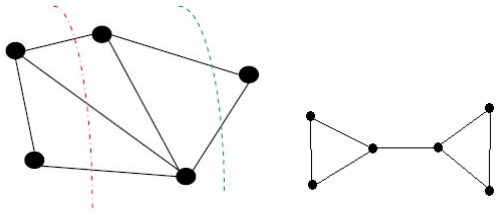
$$\ln n = \int_1^n \frac{1}{x} dx \geq (\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}) \geq (\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots + \frac{1}{n-i+1}) \Rightarrow$$

$$E[X] = \sum_{i=1}^{n} 2(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots + \frac{1}{n-i+1}) \leq 2n \ln n$$

This implies that the *expected* time for quicksort assuming a random pivot is chosen every time is $O(n \ln n)$ (which is not that exciting), but the constant hidden by the O-notation is 2!

22 Simple Monte Carlo Algorithms: Karger's algorithm

The problem of finding a minimum cut in a connected unweighted undirected graph G is to determine the smallest number of edge cuts that it takes to disconnect G. The fastest known deterministic max cut algorithm (which we will not do here) takes time $O(nm \log \frac{n^2}{m})$.



Definition: Let $G = (V, E)$ be a multigraph without self-loops. Given some $\{x, y\} \in E$, we can *contract* the edge $\{x, y\}$ as follows: (a) replace x and y with a new vertex w (b) if $\{v_x, x\} \in E$ for $v_x \neq y$, then create the edge $\{v_x, w\}$; similarly, if $\{v_y, y\} \in E$ for $v_y \neq x$, then create the edge $\{v_y, w\}$ (c) remove all self-loops.

(Karger) Consider the following randomized algorithm: While there are more than 2 nodes in G, choose an edge $\{x, y\}$ uniformly at random from G and contract it. Output the remaining edges as the minimum cut.

Claim: The probability that this algorithm terminates with a minimum cut is at least $\frac{1}{\binom{|V|}{2}}$.

Proof: It is clear that if there are exactly 2 vertices in G, then the output of the algorithm is correct with probability 1.

Let $n = |V|$. Let C be a minimal cut. Note that $|C|$ must be less than or equal to the minimum degree of any vertex in G. Note that by the Handshaking Theorem, $2|E| = \sum_{v \in |V|} deg(v) \geq n|C| \Rightarrow |C| \leq \frac{2|E|}{n}$. Thus, the probability that a randomly selected edge is in C is $\leq \frac{2}{n}$. We perform $n - 2$ removals...

$$P(e_1 \notin C) \geq 1 - \frac{2}{n}, P(e_2 \notin C | e_1 \notin C) \geq 1 - \frac{2}{n-1}, \ldots,$$

$$P(e_{n-2} \notin C | e_1, e_2, \ldots, e_{n-3} \notin C) \geq 1 - \frac{2}{n-(n-3)} = \frac{1}{3}$$

Taking the product of all these, the probability that we get the right answer is lower bounded by $\frac{2}{n(n-1)}$. □

Claim: Given that it is possible to implement a single iteration of Karger's algorithm in time $O(n^2)$ time using appropriate data structures, it is possible to determine the minimum cut in G with any arbitrary probability $\frac{1}{n^c}$ for any constant $c > 0$ in time $O(n^4 \log n)$.

Proof: Run the algorithm $x\binom{n}{2}$ times for some x to be determined. Then the probability that none of the runs yields the minimum cut is

$$1 - (1 - \frac{1}{\binom{n}{2}})^{x\binom{n}{2}} \geq 1 - e^{-x}$$

If we let $x = c \ln n$, then this probability is $\frac{1}{n^c}$. □

23 Online problems: The file cabinet I

23.1 Defining the problem

Suppose we have a filing cabinet containing n labeled but unsorted files. We receive a sequence of requests $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_m)$ to access certain files. Each request is the label of a given file. After a request to access a given file x is given, we must locate the file and remove it from the cabinet. Assume that the only way this can be done is by flipping through the files from the beginning until the file is located. If we need to flip through i files to find the one we want, then we incur a cost of i (and to attempt to find a file that does not exist incurs a cost of n). Once we find a file and remove it, we can replace it anywhere in the cabinet at no cost. On the other hand, we can also move a file without explicitly searching for it: If we decide that we want to move a file from position j to position i (where $i < j$; you assume that you always want to move a file closer to the front, never the back), then it costs $j - i$ to do so.

Our first question will be: "Are these extra movements necessary for optimal performance?" Is it always necessary to do prep work in order to get the best possible performance? Consider the following 3-element list: (x_1, x_2, x_3) (where x_1 is at the front of the cabinet). Assume that the element requests are x_3, x_2, x_3, x_2 and *you know that these requests are coming in this order.* What is the optimal sequence of moves if you know what is coming? The requests can be serviced as follows:

1. Move x_2 to the front (cost 1)
2. Move x_3 to the second position (cost 1)
3. Search for x_3 (cost 2)
4. Search for x_2 (cost 1)
5. Search for x_3 (cost 2)
6. Search for x_2 (cost 1)

This works out to a cost of 8.

Claim: Any algorithm that is not allowed to use paid exchanges must use at least a cost of 9 to perform the searches.

Proof: The first search for x_3 is required to take at least a cost of 3. Now either we move x_3 in front of x_2 or we don't.

- If we move x_3 in front of x_2, then the cost of searching for x_2 becomes 3. The second searches for x_2 and x_3 must take a total of at least cost 3 and therefore the total cost of the requests are at least 9.
- If we do not move x_3 in front of x_2, then the cost of search for x_2 is only at least 2, but then the following search for x_3 is 3 and the last search for x_2 is at least 1; therefore the total cost of the requests are at least 9.

□

23.2 Definition

Definition: Let $A(\sigma)$ be the cost that an algorithm A uses to process an online problem instance σ with initial data size n. We say that A is α-competitive for some constant α iff

$$\exists\beta(n)\forall\sigma A(\sigma) \leq \alpha OPT(\sigma) + \beta(n)$$

where β can grow with n, the initial size of the data, but cannot depend on the online request sequence. (You might think of β as the cost of preprocessing the data before the online part of the problem starts.) α is called the *competitive ratio* for algorithm A. We allow the algorithm OPT to know the request sequence σ in advance; in other words, OPT is offline. The competitive ratio is (basically) the ratio of how the algorithm does against an oblivious adversary as opposed to how an optimal algorithm would do against an oblivious adversary, the ratio of online to offline (because an optimal algorithm would be the one that had the entire problem in advance).

23.3 Lower bound

Recall that an algorithm A is α-competitive for some constant α iff

$$\exists\beta(n)\forall\sigma A(\sigma) \leq \alpha OPT(\sigma) + \beta(n)$$

We will now look to prove a lower bound on the competitive ratio for algorithms that solve the online cabinet problem. Recall that the competitive ratio is the ratio of how well a given algorithm performs as compared with an optimal offline algorithm. Assume that you are given any algorithm A that solves the cabinet problem.

Notice that no matter what A happens to be, there always exists some σ that can force A to choose the very last object in his list every single time. Thus, independent of A, the algorithm can always be forced to use a cost of $|\sigma|n$.

For that value of σ, whatever it is, what kind of behavior can we guarantee for the optimal offline algorithm?

Consider the following set of algorithms. A_π is the algorithm that chooses a permutation π of the n elements and, at the outset, orders the elements in that order. Then A_π just searches for the items as they come with no further movements. If we take the average cost over all of the algorithms in this set,

what is the average cost? Consider the following behavior by some algorithm R: At the beginning of the algorithm, using paid exchanges, R randomizes the ordering of the n elements of the list. This is accomplished in time $O(n^2)$ and is therefore worked into the $\beta(n)$ value. Then R searches for the various elements as they are requested in σ without doing any movements whatsoever. What is the total expected cost of R? The total expected cost is

$$|\sigma|\frac{1}{n}\sum_{i=1}^{n} i = |\sigma|\frac{(n+1)}{2}$$

It is clear that the expected cost of R is the same as the average behavior of the A_π. So the average cost is the same.

This argument implies that there exists some algorithm A_π such that the cost of its operation on σ is at least $|\sigma|\frac{(n+1)}{2}$. So we have that for any algorithm A, there exists some σ such that

$$|\sigma|n = A(\sigma) \leq \alpha_A OPT(\sigma) + O(n^2) \leq \alpha_A|\sigma|\frac{(n+1)}{2} + O(n^2) \Rightarrow$$

$$\alpha_A \geq \frac{2n}{n+1}$$

Thus no online algoritm can ever do better than $\frac{2n}{n+1}$ times the optimal offline algorithm for the file cabinet problem.

23.4 The covering game

Assume that you are playing a game against an opponent. Your opponent writes a bit string of any length on a sheet of paper and gives you an infinite number of magic covering strips. The covering strips will cover up 01 or 11 but *not* 10. You win the game if you can cover every 1 bit in the in the bit string except (maybe, if you need it) a single leftover and lose otherwise.

Claim: It is always possible for you to win this game.

Proof: We give an algorithm for doing so. Working backwards from the end of the string, place coverings on the first run of 1 bits you come to. This will either end with a 01 or you hit the beginning of the string. If it ends with 01, just start again, taking the bit just before the 01 as the end of your new string. If you hit the beginning of the string and the first bit of the string happens to be a 1, this is your single leftover 1 bit. □

24 Online problems: The file cabinet II

24.1 Move to front upper bound

The Move To Front (MTF) algorithm works as follows: After accessing an element, always move it to the front of the file cabinet (which is free). Do not change the relative order of any other items.

The following argument was given by Sandy Irani in 1995.

24.1.1 Definitions

Define the following cost function for a pair of items $x_i \neq x_j$ and any algorithm A:

$$A_{ij}(t,\sigma) \equiv \begin{cases} 1 + \frac{1}{n-1} & \text{if } x_i \text{ appears before } x_j \text{ in } A\text{'s list and } \sigma_t = x_j \\ \frac{1}{n-1} & \text{if } x_j \text{ appears before } x_i \text{ in } A\text{'s list and } \sigma_t = x_j \\ 0 & \text{otherwise} \end{cases}$$

Intuitively, $A_{ij}(t,\sigma)$ is $1 + \frac{1}{n-1}$ if A needs to step over x_i for a request to x_j at time t.

Denote $A(\sigma)$ to be the total cost of servicing the request sequence σ.

Let $A^p_{ij}(t) \equiv \begin{cases} 1 & \text{if } (x_i, x_j) \mapsto (x_j, x_i) \text{ via paid exchange between } \sigma_{t-1} \text{ and } \sigma_t \\ 0 & \text{otherwise} \end{cases}$

24.1.2 The proof

Definition: Say that two items x_i and x_j are in the *wrong order* at some time t if the item that is requested at time t appears after the other (so that the algorithm is charged to step over the offending item).

Claim: $\forall \sigma$ and for any algorithm A,

$$A(\sigma) = \frac{1}{2} \sum_{t=1}^{|\sigma|} \sum_{1 \leq i,j \leq n, i \neq j} [A_{ij}(t,\sigma) + A^p_{ij}(t) + A_{ji}(t,\sigma) + A^p_{ji}(t)]$$

Proof: First, notice that the A^p terms simply sum the paid exchanges that occur during the course of the algorithm. We therefore ignore their effect in the analysis afterwards and assume that no paid exchanges occur.

Assume that $\sigma_t = x_k$. Then we have that

$$\sum_{1 \leq i,j \leq n, i \neq j} A_{ij}(t,\sigma) = \sum_{1 \leq i \leq n, i \neq k} A_{ik}(t,\sigma)$$

In this sum, there is a contribution of $\frac{1}{n-1}$ for every element in the list other than x_k (for a total of exactly 1) and a contribution of exactly 1 for every element x_i such that x_i appears before x_k in the list (i.e. such that x_i is in the wrong order with respect to x_k). This works out to exactly the cost for accessing x_k. Because moving x_k after it has already been searched for is a free operation and the summation for the second term is exactly equal to the first, we are done. □

Definition: For notational convenience, we will let

$$\varphi^A_{\{i,j\}}(t,\sigma) \equiv A_{ij}(t,\sigma) + A^p_{ij}(t) + A_{ji}(t,\sigma) + A^p_{ji}(t)$$

Then we have that

$$A(\sigma) = \frac{1}{2} \sum_{t=1}^{|\sigma|} \sum_{1 \leq i,j \leq n, i \neq j} \varphi^A_{\{i,j\}}(t,\sigma)$$

25 Online problems: The file cabinet III

Fix any two distinct elements, x_i and x_j. Notice that if $\sigma_t \neq x_i$ and $\sigma_t \neq x_j$, then the only charges we incur in $\varphi^A_{\{i,j\}}(t,\sigma)$ are paid exchanges between x_i and x_j. Thus, in the future, we ignore those terms such that $\sigma_t \neq x_i$ and $\sigma_t \neq x_j$. Let τ represent the ordered vector of times such that either $\sigma_t = x_i$ or $\sigma_t = x_j$. In other words if $k \in [1, |\tau|]$, then σ_{τ_k} is either x_i or x_j. We will assume that all paid exchanges that occur in terms we ignore are gathered into the τ terms.

Lemma: For every $k \in [1, |\tau| - 1]$, we either have the following inequality hold *or* MTF has the ordering wrong at time τ_k and correct at time τ_{k+1}:

$$\varphi^{MTF}_{\{i,j\}}(\tau_k,\sigma) + \varphi^{MTF}_{\{i,j\}}(\tau_{k+1},\sigma) \leq \frac{2n}{n+1}\left(\varphi^{OPT}_{\{i,j\}}(\tau_k,\sigma) + \varphi^{OPT}_{\{i,j\}}(\tau_{k+1},\sigma)\right)$$

Proof: Notice that if MTF gets the order of $\{i,j\}$ correct, then because it does no paid exchanges, it has minimum possible φ value. So if both τ_k and τ_{k+1} represent times when MTF has the order correct, then the worst case ratio winds up being 1.

Assume then that τ_{k+1} is a time when MTF has the order wrong: assume WLOG that $\sigma_{\tau_{k+1}} = x_i$ and (x_j, x_i) is the ordering at time τ_{k+1}. Notice that if MTF has the order wrong at time τ_{k+1}, then at time τ_k, we know that $\sigma_{\tau_k} = x_j$. There are two cases to consider.

- OPT has the initial ordering (x_i, x_j) at time τ_k. Between τ_k and τ_{k+1}, OPT is then required to either perform a paid exchange or leave the ordering as is and incur the extra charge for stepping over x_i to get to x_j – either way this yields a total cost of at least $1 + \frac{2}{n-1}$.
- OPT has the initial ordering (x_j, x_i) at time τ_k. Just before τ_k, OPT is again required to either perform a paid exchange (which does not pay off so we assume that he does not do this) or leave the ordering as is and incur the extra charge for stepping over x_j to get to x_i – this yields a total cost of at least $1 + \frac{2}{n-1}$.

Thus, OPT must spend at least $1 + \frac{2}{n-1}$ on this value of k. The worst case for MTF is to get the ordering wrong both times, yielding a total cost of at most $2 + \frac{2}{n-1}$ on this value of k. Thus, the ratio is

$$\frac{\varphi^{MTF}_{\{i,j\}}(\tau_k,\sigma) + \varphi^{MTF}_{\{i,j\}}(\tau_{k+1},\sigma)}{\varphi^{OPT}_{\{i,j\}}(\tau_k,\sigma) + \varphi^{OPT}_{\{i,j\}}(\tau_{k+1},\sigma)} \leq \frac{2 + \frac{2}{n-1}}{1 + \frac{2}{n-1}} = \frac{2n}{n+1}$$

We have covered the cases where MTF has the ordering correct at both times and wrong at time τ_{k+1}. The only case remaining is the case where MTF is correct at time τ_{k+1} and wrong at time τ_k, but that is specifically the case we chose to exclude. □

Claim: $\forall i \neq j \in [1,n], \forall t \in [1, \sigma - 1]$,

$$\sum_{t=1}^{|\sigma|} \varphi^{MTF}_{\{i,j\}}(t,\sigma) \leq O(1) + \frac{2n}{n+1}\sum_{t=1}^{|\sigma|} \varphi^{OPT}_{\{i,j\}}(t,\sigma)$$

Proof: Consider writing the sum for MTF out term by term. Let a term where MTF has the correct ordering be a "0"-term and let a term where MTF has the incorrect ordering be a "1"-term. Letting the terms be a bit sequence, we can see that the previous lemma will "cover" any pairs of terms that are 01 or 11, but not those that are 10. However, because it is always possible to win the covering game, the claim must be true. □

Claim: $\exists \beta(n), \forall \sigma, MTF(\sigma) \leq \beta(n) + \frac{2n}{n+1} OPT(\sigma)$

Proof: By the above claim, we know that

$$\sum_{t=1}^{|\sigma|} \varphi_{\{i,j\}}^{MTF}(t,\sigma) \leq O(1) + \frac{2n}{n+1} \sum_{t=1}^{|\sigma|} \varphi_{\{i,j\}}^{OPT}(t,\sigma) \Rightarrow$$

$$\frac{1}{2} \sum_{1 \leq i,j \leq n, i \neq j} \sum_{t=1}^{|\sigma|} \varphi_{\{i,j\}}^{MTF}(t,\sigma) \leq \frac{1}{2} \sum_{1 \leq i,j \leq n, i \neq j} \left(O(1) + \frac{2n}{n+1} \sum_{t=1}^{|\sigma|} \varphi_{\{i,j\}}^{OPT}(t,\sigma) \right) \Rightarrow$$

$$MTF(\sigma) \leq \beta(n) + \frac{2n}{n+1} OPT(\sigma)$$

□

This implies that the simple algorithm MTF is $\frac{2n}{n+1}$-competitive for the file cabinet problem. In other words, no algorithm is going to be faster than $\frac{2n}{n+1}$ times faster than MTF for any sequence of requests.

But that implies that MTF *meets* the lower bound we found for any algorithm, and is therefore asymptotically optimal for the cabinet problem.

Interesting historical note: The first researchers to examine the online behavior of the file cabinet and MTF were Sleator and Tarjan, the same guys who came up with and analyzed the splay tree. Can you see the similarities between the two data structures and settings?

26 Splay Trees I: Analysis

26.1 Description

A *splay tree* is a binary search tree that adjusts itself whenever necessary. The explicit balance of the tree is never actually calculated anywhere (as opposed to red/black trees, AVL trees, and most other known self-balancing trees) and there may occasionally be operations that take a long time, but over time, the tree behaves exactly as if it were balanced (even though at any particular time, it may be wildly unbalanced).

Inserts, searches, and deletes are performed the same way as a binary search tree. Every time a node is inserted/found, we splay it to the root. (If we have an unsuccessful search, then splay the final node on the search. On a delete, splay the parent of the erased node.) Therefore, the nodes that are accessed the most frequently are closest to the root and the tree becomes somewhat balanced in the process. Let us assume that node R is the node we are accessing. Then

there are 3 cases to consider. Note that in each case, R is moved to the top of the relevant tree. We continue splaying until R has reached the root.

1. Node R's parent is the root. Perform a singular splay: rotate R around its parent.

2. Homogeneous configuration: Node R is the left child of its parent Q, and Q is the left child of its parent P. (or both right) Perform a homogeneous splay. First, rotate Q about P, then R about Q.

3. Heterogeneous configuration: Node R is the right child of its parent Q, and Q is the left of its parent P. (or reversed) Perform a heterogeneous splay. First, rotate R about P, then R about P.

To find out how a splay tree operates, insert the following numbers into an initially empty splay tree: 37,12,65,43,21,11,2,35,27
Delete 43 from the tree. To find out how badly off-balance a splay tree can get, try inserting the numbers from 1 through 10 in order.

26.2 Analysis

To perform an amortized analysis of splay trees, we will need to assign a potential function ϕ to the data structure. For any given node i, let $w(i)$ be an arbitrarily positive weight greater than or equal to 1 assigned to that node. For any node x, let $r(x) = \log_2 S(x)$ (where $S(x)$ represents the sum of the weights of the nodes of the subtree with x as the root). Then we define ϕ of the data structure as being $\sum_{\text{nodes } x} r(x)$. (Verify that ϕ is legal assuming that $\forall i, w(i) \geq 1$.) We will assume that it takes unit time to perform a single rotation.

Lemma For any two real numbers $x, y > 0$,

$$\frac{\log_2(x) + \log_2(y)}{2} \leq \log_2\left(\frac{x+y}{2}\right)$$

Proof: Let $f(x) = \log_2\left(\frac{x+y}{2}\right) - \frac{\log_2(x)+\log_2(y)}{2}$ for constant $y > 0$. Notice that $\frac{d}{dx}f(x) = \frac{x-y}{2x(x+y)}$. So f is decreasing if $x < y$ and increasing for $x > y$. Thus f reaches its minimum when $x = y$. □

Lemma: For any three nodes a, b, c (not even necessarily in the same binary trees),

$$S(a) + S(b) \leq S(c) \Rightarrow r(a) + r(b) \leq 2(r(c) - 1)$$

Proof: By the above lemma,

$$\begin{aligned}
\frac{r(a)+r(b)}{2} &= \frac{\log_2 S(a)+\log_2 S(b)}{2} \\
&\leq \log_2\left(\frac{S(a)+S(b)}{2}\right) \\
&\leq \log_2\left(\frac{S(c)}{2}\right) \\
&= r(c)-1
\end{aligned}$$

□

Consider the case where x is a child of the root. Let the name of the root be y. What is the amortized cost of splaying x upwards? Note that the actual cost of splaying x to the root is 1. Let x' and y' be the new positions of x and y after the rotation[4]. Notice that the potential of all other nodes remains constant.

$$a(i) = c(i) + \Delta\phi = 1 + (r(x') + r(y') - r(x) - r(y))$$

We claim that $r(y') \leq r(x') = r(y)$ (obviously) and so

$$a(i) \leq 1 + r(x') - r(x) \leq 1 + 3(r(x') - r(x))$$

Consider the case where x is in homogeneous position with y and z (z is the parent of y is the parent of x). What is the amortized cost of splaying x upwards? The actual cost is 2. Notice that the potential of all nodes other than x, y, and z remains constant.

$$a(i) = c(i) + \Delta\phi = 2 + (r(x') + r(y') + r(z') - r(x) - r(y) - r(z))$$

Notice that $r(y') \leq r(x') = r(z)$ and $r(x) \leq r(y)$ and so

$$a(i) \leq 2 + r(x') + r(z') - 2r(x)$$

Notice that $S(x) + S(z') \leq S(x')$. By the lemma,

$$a(i) \leq 2 + r(x') - 2r(x) + (2(r(x') - 1) - r(x)) = 3(r(x') - r(x))$$

Finally, consider the case where x is in heterogeneous position with y and z (same situation as above). What is the amortized cost of moving x upwards? The actual cost is 2.

$$a(i) = c(i) + \Delta\phi = 2 + (r(x') + r(y') + r(z') - r(x) - r(y) - r(z))$$

Note that $r(x') = r(z)$ and $r(y) \geq r(x)$. So

$$2 + (r(x') + r(y') + r(z') - r(x) - r(y) - r(z)) \leq 2 + r(y') + r(z') - 2r(x)$$

[4]We will keep the prime notation for simplicity throughout this analysis.

We note that $S(y') + S(z') \leq S(x')$ and again use the lemma to get

$$a(i) \leq 2(r(x') - r(x))$$

Note that for each operation except when x becomes the root, $a(i) \leq 3(r(x') - r(x))$ and at the root $a(i) \leq 3(r(x') - r(x)) + 1$. To get the total amortized cost of a splay operation, we need to sum over all the positions of x. But this is a telescoping sum! So the final result is less than or equal to $1 + 3(r(\text{root}) - r(x)) \leq 3r(\text{root}) + 1$.

Now, we are allowed to make the weights on the nodes anything we like. Let the weight of each node be 1. Then $r(\text{root}) = \log_2 S(\text{root}) = \log_2 n$ and each splay to the root takes only $O(\log n)$ amortized time!

27 Splay Trees II: Information

27.1 Shannon information

If you are sending a character over a line, the amount of information you get from the character should depend on the frequency that the character is sent. Intuitively, the information function should measure the surprise you feel at receiving the given character. The information function I should have the following properties:

1. I should depend on x.
2. $I(x) + I(y) = I(xy)$
3. $I'(x)$ is defined everywhere on $(0, 1]$ from the left hand side (because of $x = 1$)
4. $I'(1)$ is negative and finite.

Let x be any character that is sent with probability strictly less than 1. First, note that by rule 2, we must have that $I(1) = 0$. Then

$$\begin{aligned}
I'(x) &= \lim_{h \to 0^+} \frac{I(x) - I(x-h)}{h} \\
&= \lim_{h \to 0^+} \frac{I(x) - I(x) - I(1 - \frac{h}{x})}{h} \\
&= \lim_{h \to 0^+} -\frac{I(1 - \frac{h}{x})}{h} \\
&= \lim_{h \to 0^+} \frac{1}{x} I'(1 - \frac{h}{x}) \\
&= \frac{I'(1)}{x} \\
\Rightarrow \quad I(x) &= I'(1) \log_2 x + C
\end{aligned}$$

Letting $x = 1$, we get that $C = 0$ so that the amount of information contained in a character that appears with probability x is $I(x) = I'(1) \log_2 x$.

Note that the value of $I'(1)$ changes the base of the log depending on how you measure the information. You can measure information in bits, trits, digits, etc. From now on, we will assume that the base of the log is 2 so that we are always measuring in bits.

The *entropy* of a sequence of bits is the average quantity of information that you get per character in the sequence. We denote it by

$$H(\{x_1, x_2, \ldots, x_n\}) = \sum_{i=1}^{n} x_i I(x_i)$$

What are the best and worst cases for sending information over a line using only two characters? We use the entropy to compare how good a code is: the closer the average codeword size is to the entropy, the better the code.

What does $H(\{1/2, 1/2\})$ represent? How about $H(\{1/4, 1/4, 1/4, 1/4\})$? How about $H(\{1/2, 1/4, 1/4\})$?

A *uniquely decipherable* code is one that has no ambiguity: a given set of characters always has a unique decoding. The following code is not uniquely decipherable: $\{0, 1, 01\}$.

Most and least efficient: What are the minimum and maximum entropy of n-character uniquely decipherable codes? Clearly, the minimum entropy is 0. This entropy arises if one of the characters appears with probability 1 and the rest 0. This corresponds to the "perfect" model, one we always predict with absolute certainty. To find the maximum entropy, note that we are looking to maximize the function $g(p) = -\sum_{i=1}^{n} p_i \log_2 p_i$. Introduce the Lagrangian $f(p, \alpha) = g(p) + \alpha(\sum_{i=1}^{n} p_i - 1) = 0$. So we get that for every i,

$$\frac{\partial f}{\partial p_i} = -\log_2 p_i - \frac{1}{\ln 2} + \alpha = 0 \Rightarrow$$

$$\forall i, j, p_i = p_j \Rightarrow p_i = \frac{1}{n} \Rightarrow H(p) = \log_2 n$$

Notice that this corresponds to the number of bits necessary to distinguish n distinct objects. What does this analyze say about, for example, the standard encoding of characters in an average novel? □

Definition: A *prefix* code is a code such that no codeword is the prefix of any other codeword. For example, the following is *not* a prefix code: $\{01, 10, 101\}$. The following is a prefix code: $\{01, 10, 11\}$.

Claim: Any prefix code is uniquely decipherable, but not every uniquely decipherable code is a prefix code.

Proof: Consider the following code: $\{0, 01, 11\}$. We claim that this is a uniquely decipherable code that is not a prefix code. Obviously, it is not a prefix code. To prove that it is uniquely decipherable, notice that a 1 followed by a 0 implies that there is a word break in between. Where do the other breaks occur? Determine the 10 breaks. Now, for each of those breaks, if the number

of 1's before the break is even, we must have been using the third codeword; otherwise, the second codeword was used during the initial entry into the 1 run. All other codewords are now uniquely determined.

It is clear that the encoding mapping for any prefix code can be represented by a tree where only the leaves correspond to codewords. Note that in the example code above (which is *not* a prefix code), if we create a tree from the codewords, an internal node is used for codeword 0. It is clear that the first codeword in any message is uniquely decipherable because there is a unique leaf that can be reached first and no other codeword is possible other than the one in this leaf. Inductively, the entire message is therefore uniquely decipherable. □

28 Splay Trees III: Kraft Inequality

Kraft inequality for prefix codes: There exists a prefix code for a set of codeword lengths $l_1 = l(x_1), l_2 = l(x_2), \ldots, l_n = l(x_n)$ where $x_i \in C$ if and only if $\sum_{i=1}^{n} 2^{-l(x_i)} \leq 1$.

Proof: First, we show the positive direction. Assume that you are given positive integers $l_1, \ldots, l_n$ such that $\sum_{i=1}^{n} 2^{-l_i} \leq 1$. Our goal will be to show that there exists a prefix code C such that $x_1, \ldots, x_n \in C$. Order the lengths so that WLOG we have $l_1 \leq l_2 \leq \ldots \leq l_n$. Start with a full binary tree of height l_n. Note that this tree has 2^{l_n} leaves.

We will repeatedly perform a pruning of this tree. Starting from the left of the tree, gather $2^{l_n - l_1}$ consecutive leaves together and find their unique lowest common ancestor x_1. Let x_1 be the first codeword node. Now gather the next $2^{l_n - l_2}$ consecutive leaves together and find their unique lowest common ancestor. Note that it cannot possibly be related (i.e. ancestor nor descendant) to x_1. Assign x_2 to this ancestor. Continue this process until x_n has been assigned. We know x_n can be assigned because we assumed that $\sum_{i=1}^{n} 2^{-l_i} \leq 1 \Rightarrow \sum_{i=1}^{n} 2^{l_n - l_i} \leq 2^{l_n}$.

To find the codeword corresponding to the given node, start from the root and traverse the tree downwards until you reach the appropriate x_i. On the way, if you make a left, write down a 0; if you make a right, write down a 1. The binary string that is on the page at the time you reach x_i is its codeword. We claim that no codeword is the prefix of any other codeword: for that to occur, it would need to be the case that x_i is the ancestor of x_j for some i and j, but by construction, this cannot occur.

Now, we show the other direction. Assume that you are given a prefix code $x_1, x_2, \ldots, x_n \in C$. Our goal will be to show that $\sum_{i=1}^{n} 2^{-l(x_i)} \leq 1$. Consider the following process.

1. Flip a fair coin. If you get heads, append a 0 to what is already on the page; 1 otherwise. (Obviously, if there is nothing on the page, just write down the appropriate bit to start off.)

2. Check if what is written on the page is a valid codeword. If so, you win

the game. Otherwise, flip the coin again and append the result. Continue with this process until you either win the game or have a bit string of length longer than the longest possible codeword.

What is the probability of winning this game? The probability of winning this game is equal to the sum of the probabilities that you reach one of the codewords. Note that because the code is a prefix code, the probability of reaching x_i or x_j is the sum of the probabilities of reaching x_i and x_j. (This would not be true if x_i was a prefix of x_j.) This probability, which must be less than or equal to 1 is equal to exactly $\sum_{i=1}^{n} 2^{-l(x_i)}$. □

Kraft inequality for uniquely decipherable codes: There exists a uniquely decipherable code for a set of codeword lengths $l_1 = l(x_1), l_2 = l(x_2), \ldots, l_n = l(x_n)$ where $x_i \in C$ if and only if $\sum_{i=1}^{n} 2^{-l(x_i)} \leq 1$.

Proof: Because every prefix code is uniquely decipherable, it is clearly that given $l_1, l_2, \ldots, l_n$, we can construct a uniquely decipherable code.

Assume that you are given a uniquely decipherable code. Let C be the set of all codewords. Then, if $l(x)$ is the length of codewords x in bits, our goal is to show that $\sum_{x \in C} 2^{-l(x)} \leq 1$.

Note that $\sum_{x \in C} 2^{-l(x)} = \sum_{j=1}^{T} w_j 2^{-j}$ where w_j is the number of codewords of length j in C and T denotes the length of the maximal codeword length in C. Now notice that

$$\left(\sum_{j=1}^{T} w_j 2^{-j} \right)^s = \sum_{k=s}^{Ts} \frac{N_k}{2^k}$$

where $N_K = \sum_{i_1 + \ldots + i_s = k} w_{i_1} \ldots w_{i_s}$ is the total number of messages whose coded representation is of length k.

For an example, consider the code $\{0, 10, 1100, 1101, 1111\}$. Then

$$s = 1 : \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^4} + \frac{1}{2^4} + \frac{1}{2^4} = \frac{1}{2^1} + \frac{1}{2^2} + \frac{3}{2^4}$$

$$s = 2 : (\frac{1}{2^1} + \frac{1}{2^2} + \frac{3}{2^4})^2 =$$

(via FOIL multiplication)

$$\frac{1}{2^2} + \frac{2}{2^3} + \frac{1}{2^4} + \frac{6}{2^5} + \frac{6}{2^6} + \frac{9}{2^8}$$

Note that in particular, given two consecutive choices for codewords, there is only one way to form a bit sequence of length 4, 6 ways to form a bit sequence of length 6, 9 ways to form a bit sequence of length 8, etc.

Because the code is uniquely decipherable, to every sequence of k bits there corresponds at most one possible message; thus, $N_k \leq 2^k \Rightarrow$

$$\sum_{k=s}^{Ts} \frac{N_k}{2^k} \leq \sum_{k=s}^{Ts} 1 = Ts - s + 1 \leq Ts$$

Now

$$\left(\sum_{j=1}^{T} w_j 2^{-j}\right)^s = \sum_{k=s}^{Ts} \frac{N_k}{2^k} \leq Ts \Rightarrow$$

$$\sum_{j=1}^{T} w_j 2^{-j} \leq (Ts)^{\frac{1}{s}}$$

Because this must hold for every s, we can let $s \to \infty \Rightarrow$

$$\sum_{j=1}^{T} w_j 2^{-j} \leq \lim_{s\to\infty} (Ts)^{\frac{1}{s}} = 1$$

and the result follows. □

29 Splay Trees IV: Asymptotic static optimality

Jensen's inequality: Let f be a function such that $\forall x_1, x_2, t$,

$$tf(x_1) + (1-t)f(x_2) \geq f(tx_1 + (1-t)x_2)$$

In other words, f is *concave up*; this is the same thing as stating that f has nonnegative second derivative everywhere in its domain. Then for any weights $t_1, \ldots, t_n$ such that $\sum t_i = 1$ and points $x_1, \ldots, x_n$,

$$\sum t_i f(x_i) \geq f(\sum t_i x_i)$$

with equality iff $x_1 = x_2 = \ldots = x_n$.

Proof: Mathematical induction. It is obviously true if $n = 1, 2$. For $k+1$, we have the following

$$f(\sum_{i=1}^{k+1} t_i x_i) = f(t_1 x_1 + (1-t_1) \sum_{i=2}^{k+1} \frac{t_i}{1-t_1} x_i) \leq$$

$$t_1 f(x_1) + (1-t_1) f(\sum_{i=2}^{k+1} \frac{t_i}{1-t_1} x_i)$$

Now note that $\sum_{i=2}^{k+1} \frac{t_i}{1-t_1} = 1$ and there are exactly k terms in the sum. Thus,

$$t_1 f(x_1) + (1-t_1) f(\sum_{i=2}^{k+1} \frac{t_i}{1-t_1} x_i) \leq t_1 f(x_1) + \sum_{i=2}^{k+1} (1-t_1) \frac{t_i}{1-t_1} f(t_i) =$$

$$\sum_{i=1}^{k+1} t_i f(x_i)$$

Note that the if and only if part of the inequality follows from the induction as well. □

Gibbs' inequality: Let p_i, q_i be any 2 probability distributions (for $1 \leq i \leq n$). Then

$$-\sum_{i=1}^{n} p_i \log_2 p_i \leq -\sum_{i=1}^{n} p_i \log_2 q_i$$

with equality if and only if $p_i = q_i$ for all i.

Proof: The second derivative of $f(x) = -\log_2 x$ is everywhere positive and it is therefore concave up. So we have that by Jensen's inequality

$$\sum_{i=1}^{n} p_i(-\log_2 \frac{q_i}{p_i}) \geq -\log_2(\sum_{i=1}^{n} p_i \frac{q_i}{p_i}) = 0$$

with equality iff $\forall i, j, \frac{q_i}{p_i} = \frac{q_j}{p_j} \Rightarrow \sum_{i=1}^{n} p_i(-\log_2 \frac{q_i}{p_i}) = 0 \Rightarrow \forall i, \frac{q_i}{p_i} = 1 \Rightarrow q_i = p_i$. □

Shannon's Noiseless Coding Theorem Part I: The average length of a codeword of any uniquely decipherable code must be at least as large as the entropy of the distribution of the code. More explicitly, if we let l_i be the length of a codeword x_i that appears with probability p_i, then $\sum p_i l_i \geq \sum -p_i \log_2 p_i$.

Proof: Take $x_1, \ldots, x_n$ and $p_1, \ldots, p_n$ as in the statement, suppose the x_i have been encoded in such a way that the code is uniquely decipherable, and let l_i be length of the codeword for x_i. Then by Kraft's inequality $\sum 2^{-l_i} \leq 1$. Let $C = \frac{1}{\sum 2^{-l_i}}$ so that $C2^{-l_1}, C2^{-l_2}, \ldots, c2^{-l_n}$ is a probability distribution, and can play the role of $\{q_i\}$ in Gibbs' inequality, which yields

$$\sum_{i=1}^{n} p_i \log_2 p_i \geq \sum_{i=1}^{n} p_i \log_2(c2^{-l_i}) = \sum_{i=1}^{n} p_i(\log_2 C - l_i) = \log_2 C - \sum_{i=1}^{n} p_i l_i$$

Now put back the minus signs and remember that $\frac{1}{C} \leq 1 \Rightarrow C \geq 1 \Rightarrow \log_2 C \geq 0 \Rightarrow$

$$\sum_{i=1}^{n} p_i l_i \geq -\sum_{i=1}^{n} p_i \log_2 p_i + \log_2 C \geq -\sum_{i=1}^{n} p_i \log_2 p_i$$

□

29.1 Static optimality and online solutions

Assume that we are given an large alphabet of keys and a sequence of searches for these keys. We will also assume that each key is accessed at least once; otherwise, just remove it from the alphabet. Consider the set of all binary search trees that, at each node (including internal nodes), we hold a record corresponding to the key in the node. There is at least one such binary search tree in this set that is "optimal" in the sense that the total number of nodes accessed is minimized for the given sequences of searches. Aymptotically, how

many node accesses does any given sequence *require* assuming we are allowed to use the best possible binary search tree when we do our searches?

Think of an optimal uniquely decipherable binary search tree as a code for transmitting bits for purposes of sending some key sequence from one place to another. The size of a given key/codeword is clearly equal to its edge depth in the tree. Assume that the keys are numbered $k_1, k_2, \ldots, k_n$ and let the depth of key k_i be $d(k_i)$. Let the frequency that k_i appears in the sequence be $f(k_i)$; thus, the total number of keys in the sequence is $\sum_{i=1}^{n} f(k_i) \equiv T$. Then, Shannon's Noiseless Coding Theorem yields that the asymptotically shortest possible sequence of bits for this code sequence has total length

$$\sum_{i=1}^{n} f(k_i) d_{\text{opt}}(k_i) \geq \sum_{i=1}^{n} -f(k_i) \log_2 \frac{f(k_i)}{T} = \sum_{i=1}^{n} f(k_i) \log_2 \frac{T}{f(k_i)}$$

Unfortunately, the optimal static tree is difficult to find (though not impossible, given the entire sequence of searches) and there is the added problem that, in practice, you cannot possibly count on having the entire message to analyze in advance so that some kind of online solution is necessary. Is there an easier way to transmit the message? Assume that we start with a binary search tree of any kind that contains all of the relevant keys. Think of this binary tree as operating like a splay tree: upon request of a key's codeword, simply transmit the correct codeword (the key's current position in the splay tree) followed by an "end codeword" character (think about doubling each of the bits that we send from the splay tree so that a 010 turns into 001100; then, we can make the "end codeword" character be 01- this only doubles the size of the code); each time a key is searched for in the splay tree, its codeword changes (probably) based on its being splayed to the root. How many bits/operations will this strategy take?

Recall in the analysis of the splay tree that the maximum amortized running time of any splay tree operation on node x was at most $1 + 3(r(\text{root}) - r(x))$ where the weights that we assigned to each node were arbitrary numbers 1 or greater. Assume that we are given any splay tree populated by the keys k_i. Assign the weight of node k_i to be $f(k_i)$ (as opposed to 1). Then note the following facts.

$$r(\text{root}) = \log_2 S(\text{root}) = \log_2 \sum_{i=1}^{n} f(k_i) = \log_2 T$$

$$\forall i, r(k_i) = \log_2 S(k_i) \geq \log_2 f(k_i)$$

These facts imply that for any node k_i, any splay operation (search, in particular) takes amortized time

$$1 + 3(r(\text{root}) - r(k_i)) \leq 1 + 3 \log_2 \frac{T}{f(k_i)} = O(\log_2 \frac{T}{f(k_i)})$$

Thus, the entire sequence of operations takes amortized time at most (because the constant in the O-notation does not vary with any parameter and we send

the "end codeword" T times total)

$$\sim T + \sum_{i=1}^{n} f(k_i) O(\log_2 \frac{T}{f(k_i)}) = \sum_{i=1}^{n} f(k_i) + \sum_{i=1}^{n} f(k_i) O(\log_2 \frac{T}{f(k_i)}) =$$

$$\sum_{i=1}^{n} f(k_i) O(\log_2 \frac{T}{f(k_i)}) = O(\sum_{i=1}^{n} f(k_i) \log_2 \frac{T}{f(k_i)})$$

Comparing this to result above, we conclude that the use of a splay tree for performing *any nontrivial sequence* of searches is at most a constant factor worse than the *best possible* static tree!