Summary of Lectures

Theory of Algorithms, Spring 2023

by:

Padmini Mukkamala

Budapest University of Technology and Economics

Last updated: June 2, 2023

Contents

Color codes:

Definitions

Pseudocode

Correctness/Invariant

Running Time Analysis

Lecture 1

Basics, Order Notations

We will learn three main things during the semester: strategies to divise efficient algorithms (brute force, divide and conquer, dynamic programming, greedy algorithms), graph algorithms (Kruskal's, Dijkstra's etc), data structures and how they effect the efficiency of different algorithms (arrays, lists, all sorts of trees, heaps).
All problems discussed in this lecture will use Arrays of numbers. Design of algorithms has three main steps: the algorithm itself (in words or pseudocode), correctness of the algorithm (we will use invariants), running time (in Order notation). For the running time, we assume that the arrays contain numbers (integers or reals, but bounded to not be too big) and basic manipulation of two numbers (addition, subtraction, multiplication, division, comparision) is one computational step.

Max, Min Search

Input: An array $a[1:n]$ of numbers
Output: The index and the value of the smallest element in the array.
Algorithm in words: we initialize a variable min as $a[1]$. we will read the elements of the array from $a[2]$ to $a[n]$, and compare it with min and update it if min was larger.

Pseudocode:

1. $min = a[1]$ and $pos = 1$
2. for $i = 2$ to n
3. if $a[i] < min$
4. $min = a[i]$ and $pos = i$
5. return (min, pos)

Correctness: for this we will use Invariants

Invariant

An Invariant is a property that is maintained **before the start** of every loop. Invariants work a lot like Induction, there is the Initialization (compare with Base case) and the Maintanence (compare with Induction step) to check.

For this problem, we will establish the invariant that before the i^{th} step of the loop, min will store the smallest element in $a[1 : i-1]$.
Initialization: Here initially when $i = 2$, $min = a[1]$, so it is true.
Maintanence: For any general i, since min contains the smallest element in $a[1 : i-1]$ and then we compare it with $a[i]$ and update it with smaller of the two, the invariant is maintained.
So when all the loops are finished we will have the smallest element stored in min.
Note: the book calls the last step Termination.

Order Notation

When talking about the running time of algorithms, we specify how many computation steps were needed in the **worst-case** for an input of length n, in other words, over all inputs of length n, the one which required the maximum number of steps will be taken as the running time of the algorithm.
For max/min problem, let c_1 be the constant amount of work in updating and checking the counter in every loop, let c_2 be the work done in the comparison of $a[i]$ with min, and let c_3 be the amount of work done when updating the value of min. We can see that the loop will run exactly $n-1$ times. We may not update min everytime, but n is an upper bound for the number of updates. Then we can see that the running time is $\leq c_1(n-1) + c_2(n-1) + c_3 n = (c_1 + c_2 + c_3)n - (c_1 + c_2)$. This is a messy formula and it does not say much when we compare two different algorithms. For example, suppose this, when worked out gives us $2023n - 300$. Further suppose we have another algorithm which works in $\frac{n^2}{10^6}$ time. Which algorithm is better? For this we use Order notation to compare. Basically we strip the function of all but the most dominant term and then we also ignore constants.

Formally, $f(n) \in O(g(n))$ if there exist constants c, n_0 such that, $f(n) \leq cg(n), \forall n \geq n_0$.

For determining lower bounds on the running time we will use Ω notation.

Formally, $f(n) \in \Omega(g(n))$ if there exist constants c, n_0 such that, $f(n) \geq cg(n), \forall n \geq n_0$.

If $f(n) \in O(g(n))$ and also, $f(n) \in \Omega(g(n))$, then we say $f(n) = \Theta(g(n))$.

So, for the Min/Max algorithm, the running time is $O(n)$. Because we check all the elements in all situations, we also see that it is $\Theta(n)$.

Lower bounds

This is in general very hard, but for the max/min problem, we can show that $n-1$ comparisons are needed for a correct algorithm, so this algorithm is optimal because it uses exactly $n-1$ comparisons. This algorithm is also asymptotically optimal . By asymptotically optimal, we mean that in Θ notation, the running time of the algorithm cannot be better than $\Theta(n)$.
To see that $n-1$ comparisons are needed, we notice that in every comparison, the greater of the two numbers can be said to the loser of a match between the numbers. In the end of all comparisons, every other element besides the smallest must have been a loser at least once. But each comparison creates at most 1 loser and we need $n-1$ elements to have lost, so there must be at least $n-1$ comparisons.

Selection sort

Input: An array of n numbers.
Output: The elements of the array appear in increasing order.
Algorithm in words: In the i^{th} iteration, we will only deal with the array $a[i : n]$ and we will find the minimum in this array and swap it with the element $a[i]$.
Pseudocode: We will use $min(a)$ to be the function that returns the minimum value and its position from the array a.

1. for $i = 1$ to $n - 1$
2. $\quad (m, pos) = min(a[i : n])$
3. $\quad (a[pos], a[i]) = (a[i], a[pos])$
4. return a

Correctness: We have already seen the correctness of the min algorithm. We have to now check that the numbers of the array are stored in a sorted order.
Invariant: Before the i^{th} loop, the $i - 1$ smallest elements of the original array appear in $a[1 : i - 1]$ in increasing order.
Initialization: When $i = 1$, then we have 0 smallest elements in an empty output array, so the claim is trivially true.
Maintenance: Given that the smallest $i-1$ elements are in $a[1 : i-1]$, then the i^{th} smallest element will be the smallest element of $a[i : n]$. The algorithm finds this and swaps it with $a[i]$, so at the end of the loop, the i smallest elements will be in $a[1 : i]$ in increasing order.

Running time: Our outer loop is running n times, and in each loop we are running the min finding algorithm, which takes at most cn steps. So in all, the running time is $O(n^2)$.

Binary Search

Here we introduce another method called Divide and Conquer. The idea is to divide the problem into **disjoint** subproblems, solve those, and then use the solution to the two sub problems to solve the main one. Note: if the subproblems are not disjoint, then we get a different algorithm design methodology called Dynamic Programming.
Input: A **sorted** array of n elements $a[1 : n]$, and a value v.
Output: Index i such that $a[i] = v$ or 0 if such an index does not exist.
Algorithm in words: At every stage, we check if v is equal to the middle element of the array. If it is, then we return its index. If v is bigger, then we check in the right half of the array, and if its smaller, then we check the left half.
Pseudocode:

1. $l = 1$, $r = n$, $pos = 0$
2. while $r > l$
3. $\quad x = \lfloor \frac{r+l}{2} \rfloor$
4. $\quad$ if $v == a[x]$
5. $\quad\quad pos = x$
6. $\quad\quad$ return pos
7. $\quad$ else if $v > a[x]$
8. $\quad\quad l = x + 1$
9. $\quad$ else
10. $\quad\quad r = x - 1$
11. return pos

Correctness:

Invariant: Before the i^{th} iteration, $a[l-1] < v < a[r+1]$, where if $l = 1$ or $r = n$, then we assume that the inequality is trivially true. Another way to say this is, if the value v is in the array, then before the i^{th} iteration, the value is either in $a[l:r]$ or not in the array. If it helps, you can also think of $a[0]$ as a new element set to $-\infty$ and $a[n+1]$ a new element set to ∞.
Initialization: For $i = 1$, $l = 1, r = n$, so it is trivially true.
Maintenance: Since a is a sorted array, if we reached the i^{th} iteration, the value is either in $a[l, r]$ or not in the array. Then we compare v with $a[\frac{l+r}{2}]$. The way we update this, the property is still maintained.

Running time: In each iteration, we are doing a constant amount of work in checking v with the middle value and updating the left and right markers. So the question really is how many loops? Since we half the size of the array with every iteration, the number of loops is $\lceil log(n) \rceil$, and so the running time is $O(log(n))$.

Lecture 2

Recurrences, Dynamic Programming

Recursive Algorithms

We also could have written the pseudocode for Binary search as follows:

```
1.  def binarySearch(a,v):
2.      l = length of a
3.      if l == 0 return 0
4.      x = ⌊(1+l)/2⌋
5.      if v == a[x]:
6.          then return x
7.      else if v > a[x]
8.          p = binarySearch(a[x+1 : l])
9.          if p > 0 return(x + p)
10.         else return(p)
11.     else
12.         return(binarySearch(a[1 : x−1]))
```

The previous algorithm was an iterative algorithm while this one is a recursive algorithm.

Iterative Algorithm: is an algorithm which repeats a certain loop some number of times. Computing running time of iterative algorithms requires estimating how many loops the algorithm iterated over, and the steps or cost of each loop. And for correctness we used Invariants.

Recursive Algorithm: is an algorithm which calls itself. Computing running time of recursive algorithms requires solving a recurrence of the sort $T(n) \leq ...T(m)+...$ where m can be $n-a$ or $\frac{n}{a}$ for some constant a. For checking the correctness of the algorithm, we usually use Induction.
A word of caution: Although recursive algorithms are great for understanding problems and divide and conquer techniques, they are in practice much slower than their iterative counterparts. So if you can, you should prefer to design an iterative algorithm instead of a recursive one.

Recurrence for Binary Search is $T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + c$ where c is a constant. We can solve this to get $T(n) = O(log(n))$.

Mergesort

We will give another example of the Divide and Conquer method to give a recursive sorting algorithm, Mergesort. But first we need another algorithm to merge two sorted lists.
Pseudocode for Merge:

1. def Merge(a,b):
2. $c = []$, $i_a = 1, i_b = 1, i = 1$
3. while i_a, i_b are less than or equal to the lengths of a and b respectively.
4. if $a[i_a] < b[i_b]$
5. $c[i] = a[i_a]$, $i = i + 1, i_a = i_a + 1$
6. else $c[i] = b[i_b]$, $i = i + 1, i_b = i_b + 1$
7. if $i_a \leq$ length of a, copy the remaining part of a to c
8. if $i_b \leq$ length of b, copy the remaining part of b to c
9. return(c)

Note: Merge is an iterative algorithm.

Invariant: When merging two sorted arrays, before the i^{th} step, the smallest $i-1$ elements of the two arrays are in the new array area in increasing order.
Initialization: When we have not yet put any elements in the new array area, then this is trivially true.
Maintenance: Suppose the $i-1$ smallest elements are in the new array in increasing order. Then, the i^{th} smallest element must be in the remaining parts of the two arrays. Since these two are sorted, it must be one of the first elements of both and we pick the smaller of the two and place it at the i^{th} position of the new array, so the invariant is still true.

Running time: Merging two arrays of length k, l takes $c(k+l)$ steps. It is custom to take the input size $k+l$ as n, and then we see that merge is $O(n)$.

We now can define a recursive algorithm to sort the array.
Input: An array $a[1:n]$
Output: The elements of the array in a sorted order.
Algorithm in words: At each stage, we split the array into two halves, and sort them recursively. Once we obtain the sorted halves, we will merge them.
Pseudocode:

1. def Mergesort(a) :
2. $n = length(a)$
3. if $n == 1$ return a
4. $m = \lfloor \frac{n+1}{2} \rfloor$
5. return $Merge(Mergesort(a[1:m]), Mergesort(a[m+1:n]))$

Correctness: We use inductive reasoning to see the correctness of recursive algorithms. As a base case, we can consider the array has length 0 or 1. Then they are trivially sorted. Now, assume that it works correctly for inputs of lengths at most n. Now consider an input of length $n+1$. Our algorithm breaks it into two arrays of half the size, which, by inductive hypothesis, are sorted correctly. And we checked the correctness of Merge already, so after Merge, the array a must be sorted.

Running time: Mergesort is a recursive algorithm, so we will write a recurrence for its running time. $T(n) \leq 2T(\lfloor \frac{n}{2} \rfloor) + n$. Solving this, we can see that Merging two arrays using Mergesort takes $O(n \log n)$ steps.

Maximum Subarray

Our last example for Divide and Conquer method will be the maximum subarray problem.
Input: An array $a[1:n]$ of integers (may be negative).
Output: Value m that is the maximum over all $1 \leq l \leq r \leq n$ of the sum, $\sum_{i=l}^{r} a[i]$.
Algorithm in words: For the array a, the maximum subarray is either in the left half, or the right half of a, or it crosses over the middle of a. In the first two cases, we can recursively find the maximum subarray, while in the last case, we can just check in linear time all subarrays crossing the mid point.
Pseudocode:

1. def maxCrossingSubarray(a) :
2. $n = length(a)$
3. if $n == 1$ return a
4. $m = \lfloor \frac{n+1}{2} \rfloor$
5. $sum = 0$, $max = 0$
6. for $i = m$ to 1
7. $sum+ = a[i]$
8. if $sum > max$: $max = sum$
9. for $i = m+1$ to n
10. $sum+ = a[i]$
11. if $sum > max$: $max = sum$
12. return max

Pseudocode for Maximum Subarray:

1. def maxSubarray(a) :
2. $n = length(a)$
3. if $n == 1$ return $a[1]$
4. $m = \lfloor \frac{n+1}{2} \rfloor$
5. return the maximum of (maxSubarray(a[1:m]), maxSubarray(a[m+1:n]), maxCrossingSubarray(a))

Running time: Recurrence for its running time is $T(n) \leq 2T(\lfloor \frac{n}{2} \rfloor) + cn$. Solving this, we get $O(n \log n)$ steps.

Dynamic Programming

A very useful and powerful way of generating iterative algorithms for problems is called Dynamic Programming. The key idea is to start with very small sub problems and solve bigger sub problems using the solution of the previous smaller problems. To avoid recursion and multiple calls, we will always store the solutions of every sub problem solved so far and build the solution matrix or structure. An important thing to note is that this method differs significantly from Divide and Conquer in which the sub problems don't overlap. In Divide and Conquer, we break the problem into two or more smaller, but disjoint sub problems and put them together. In Dynamic programming on the other hand, the sub problems heavily overlap, and it is necessary to maintain a bottom-up storage structure for the sub problems.
There are eight steps we use in an algorithm involving Dynamic Programming.

1. Sub-problems: Definition of sub problems
2. Order: Order of solving the sub problems
3. Start: Base cases
4. Continue: Method for solving later sub problems when the solution to the earlier sub problems (defined according to the order in Step 2) is known.
5. End: Solution of the problem using all sub problems
6. Correctness: we will check individually for base cases, method and final answer, steps 3, 4 and 5.
7. Running Time
8. Optimal Object: This involves finding the part of the input where the optimized solution occurs. At times, it may not be obvious from the optimal value where this optimal value occurs in the input. So we may have to figure out a method of determining this from the values of the sub problems.

Maximum Subarray using Dynamic Programming

We will give an $O(n)$ algorithm for the maximum subarray problem by listing out the eight steps mentioned above.

1. Sub-problems: We let $M[i]$ be the maximum subarray under the condition that $a[i]$ must be the last element of the subarray.
2. Order: from $i = 1$ to n.
3. Start: $M[1] = a[1]$
4. Continue: $M[i] = max\{a[i], M[i-1] + a[i]\}$

5. End: for maximum subarray, we need to find $max\{M[1], M[2], ..., M[n]\}$

6. Correctness: The maximum subarray ending in $a[1]$ is $a[1]$ itself. The maximum subarray ending in $a[i]$ can either be just $a[i]$. But if it is longer, then the subarray obtained by excluding $a[i]$ from it must be maximum in $a[1 : i-1]$, otherwise we could replace it with the maximum one. Finally, the maximum subarray in $a[1 : n]$ must end in some i, and we have computed the maximum of all such values.

7. Running Time: We do a constant amount of work in building each $M[i]$, so the M array takes $c_1 n$ time to build. Then the final answer required finding the maximum element in M which is further $c_2 n$ steps. So the algorithm is $O(n)$.

8. Optimal Object: Let $M[i]$ have the maximum value. To find exactly what part of a has this maximum subarray, we need to start from $a[i]$ and keep adding elements $a[i-1], a[i-2]$... so on until we get the maximum sum.

Knapsack using Dynamic Programming

Input: Two arrays $v[1 : n]$ and $w[1 : n]$ of the corresponding value and weight of n items. We are also given b, a limit on the total allowed weight of a subset.
Output: A subset I of $\{1, 2, ..., n\}$ such that $\sum_{i \in I} w_i \leq b$ and the value $\sum_{i \in I} v_i$ is the maximum possible.
Notice that brute-force method takes $O(2^n)$ time because there are 2^n subsets of an n element set. We will use Dynamic Programming to solve the knapsack problem.

1. Sub-problems: We let $M[i, j]$ be the maximum value of a subset from $\{1, 2, ..., i\}$ such that the total weight of this subset is at most j. Here $0 \leq i \leq n$ and $0 \leq j \leq b$.

2. Order: We first solve the entire row 1, that is $M[0, 0 : b]$, then row 2 with $M[1, 0 : b]$ and so on.

3. Start: $\forall i, j, M[0, j] = M[i, 0] = 0$

4. Continue: $M[i, j] = max\{M[i-1, j], M[i-1, j-w_i] + v_i\}$, here the second term is only if $w_i \leq j$.

5. End: $M[n, b]$

6. Correctness: The maximum value we can take without any elements or with capacity 0 is 0. The maximum value from the first i elements may either contain the i^{th} element or not. If not, then $M[i-1, j]$ will give the value. But if it does contain i, then $M[i, j] - v_i$ must be maximum in the first $i-1$ elements with capacity $j - w_i$, otherwise we could replace it with the maximum from there.

7. Running Time: We do a constant amount of work in building each $M[i, j]$, so the matrix M takes $O(nb)$ time to build. Since $M[n, b]$ is the required answer, this is the running time.

8. Optimal Object: We could make the matrix M also store an additional bit which is 1 for $M[i, j]$ if in the recurrence we picked the i^{th} element, and is 0 otherwise. This way we can find out the subset in $O(n)$ additional time.

Lastly, we note that if we didn't store all the subproblems, but instead just made a recursive call for the Knapsack problem as follows:
$Knapsack(w[1 : n], v[1 : n], b) = \max\{Knapsack(w[1 : n-1], v[1 : n-1], b), Knapsack(w[1 : n-1], v[1 : n-1], b - w_n) + v_n)$
Then the running time of this, if denoted by $T(n)$ will have the recurrence, $T(n) = 2T(n-1) + c$ where c is the constant amount of work done in the additions, subtractions and finding the max. Solving this recurrence, we see that the running time is $O(2^n)$. So it is clear that **not** using the subproblems but computing them again every time is very time intensive. Note that we use the **same** recurrence in our Dynamic Programming algorithm, just that there we **did not** compute the subproblems several times, but stored it in a matrix and computed it only once.

Lecture 3

Graph representation, Depth First Search, Topological ordering in DAGs.

Quick recap: Linked Lists: here objects are arranged in a linear order. Each object has a key (or value) and a pointer (address) to the next object in the list. Insertion of an element in a linked list has $O(1)$ running time, where n is the length of the list. This is because insertions happen at the beginning of the list. Search and deletion (based on key) on the other hand take $O(n)$ time.

Graph Representation

A graph $G(V, E)$ comprises of a set V of vertices and a set E of edges. We will assume we have a simple graph (no multiple edges, loops). In practice, if the vertices have no attributes, then we just store edges and there is no extra memory used for storing vertices. But if the vertices do have attributes like color or weight, then we use an array to store these values. There are two standard ways to store the edges.

Adjacency List: The Adjacency List representation contains an array $Adj[1:n]$ of $|V(G)|$ linked lists. In the case of a directed graph, the list at $Adj[i]$ contains all the vertices j such that $(i, j) \in E(G)$, that is, there is a directed edge in G from i to j. For an undirected graph, this is list of neighbors of i (vertices with an edge with i). Notice that this list is not necessarily sorted.

Note that the Adjacency list uses $O(n+m)$ amount of storage, where $n = |V(G)|$ and $m = |E(G)|$. Some basic operations: Checking if an edge exists ($O(n)$), degree of a vertex (($O(n)$), in-degree of a vertex ($O(n + m)$). Adjacency lists are the preferred storage method unless the graph is known to be very dense. This is the model we will use for a majority of the graph algorithms we discuss.

Adjacency Matrix: The Adjacency Matrix representation contains a matrix $A[1:n, 1:n]$ where $|V(G)| = n$, such that the entry $A[i, j] = 1$ if $(i, j) \in E(G)$, otherwise it is 0. Note that for undirected graphs, the order of indices in (i, j) does not matter, and so $A[i, j] = A[j, i]$.

The Adjacency matrix uses $O(n^2)$ amount of space. Here edge look up is $O(1)$ and degree of a vertex requires $O(n)$ steps. It is a preferred representation of the graph when either the graph is small, or if the graph is dense (many edges).
Every graph algorithm requires an efficient way to explore the graph and judge its structure, connectivity etc. We want to be able to answer simple questions like, is there a path from vertex s to t, what are the vertices in the connected component of s, does the graph have a spanning tree? One graph traversal algorithm that we use very frequently to answer all the questions above is Depth First Search. Many graph algorithms either start with or use a modified version of Depth First Search. An important thing to note is that the DFS algorithm does not have any output, it is just a traversal of the graph. We will use several attributes of the vertices and edges to derive information from the DFS run on the graph.

DFS

DFS. Input: G with $Adj[1:n]$, the Adjacency List for it. Important attributes of the vertices:

- $v.d$: discovery index of the vertex v. Initially, $v.d = *, \forall v$. This denotes the order in which the vertices are discovered, so this list generates a **preordering** of the vertices. **Note:** the book keeps a time stamp, the time stamp also includes times when vertices are finished, so our ordering will be different from the one in the book. Our $v.d$ vector will be the numbers $1:n$ in some order.
- $v.f$: finishing index of the vertex v, also called a **postordering** of the vertices. **Note:** Again our $v.f$ vector will be a permutation of $1:n$ denoting the order in which the vertices were finished in the DFS traversal. The book keeps a time stamp here also, so the numbers in the book will be different.
- $v.color$: this would be the color of each vertex. A vertex is WHITE if it has not been discovered, so then $v.d = *$, it is GRAY if it has been discovered but not finished, so $v.d \neq *$ but $v.f = *$, and finally it is BLACK if its processing has been finished, so $v.d \neq *$ and $v.f \neq *$.
- $v.p$: parent of v in the DFS tree (The book uses π instead of p).

- D: a variable to store the number of new vertices discovered so far
- F: a variable to store the number of vertices whose processing is over. Note that the algorithm stops when all vertices are processed, that is, when $F = n$.
- We don't specifically list it here or in the algorithm, but we can make the DFS algorithm store edge attributes like Tree edge, Back edge, Forward edge and Cross edge in Adj.

This algorithm is from Cormen-Leiserson-Rivest with a few modifications. In particular we don't keep a time stamp, but two variables D, F.

Input: $G = (V, E)$.

DFS(G)

1. **for** each vertex in $u \in V(G)$,
2. $u.color$ =WHITE
3. $u.d = *$, $u.f = *$ and $u.p = *$
4. $D = 0$, $F = 0$
5. **for** each vertex in $u \in V(G)$,
6. **if** $u.color ==$ WHITE (Note: here we could have checked if $u.d == *$ instead)
7. DFS-VISIT(G,u)

DFS-VISIT(G,u)

1. $D = D + 1$
2. $u.d = D$
3. $u.color$ =GRAY
4. **for** each $v \in Adj[u]$
5. **if** $v.color ==$ WHITE (Note: here we could have checked if $v.d == *$ instead)
6. $v.p = u$
7. DFS-VISIT(G,v)
8. $F = F + 1$
9. $u.color$ = BLACK
10. $u.f = F$

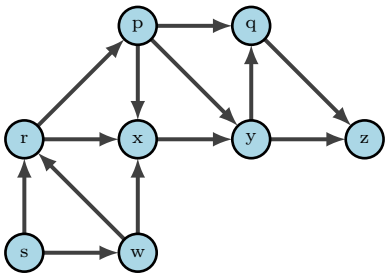
Some key observations about the DFS algorithm:

1. If $\overrightarrow{uv} \in E(G)$ and u has been discovered and v has not, then, v will be discovered at a later time (that is, it cannot happen that v is never discovered), and v will be finished earlier than u. This is because we finish u only when all its neighbors are discovered (so v will be discovered), and also because the recursive call from u to its neighbor only comes back to u when that neighbor has been finished, so all DFS-VISIT calls from u must finish before u is finished. So we can further observe that v must be a descendant of u in the DFS tree.
2. If $uv_1v_2...v_kv$ is a directed path in G and u has been discovered but no other vertex on the path has been discovered, then, v will be discovered at a later time (that is, it cannot happen that v is never discovered), and v will be finished earlier than u. This can be seen by repeatedly applying the previous argument to the path. Again, v will be a descendant of u in the DFS tree.

3. All the discovery edges form a tree, that is why it is called the Depth First Search Tree. A discovery edge is necessarily between a GRAY and a WHITE vertex. If there is a cycle, we would need an edge between two GRAY vertices, which is not possible.

Notice that the DFS algorithm processes every edge exactly once. In the processing of an edge, we update a constant number of attributes. We also note that the DFS-VISIT algorithm is invoked exactly once for each vertex when it is WHITE, and never again because we immediately turn in GRAY. So we can see that the running time of this algorithm is $O(n+m)$, where $n = |V(G)|$ and $m = |E(G)|$.

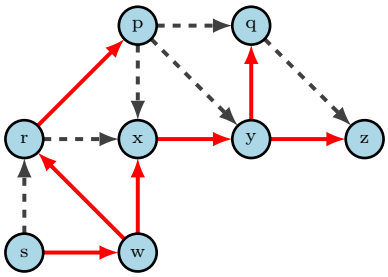
Consider the following graph:



The DFS algorithm when run on the above graph will give the following results:

v	s	p	q	r	x	y	z	w
$v.d$	1	8	6	7	3	4	5	2
$v.f$	8	5	2	6	4	3	1	7
$v.p$	*	r	y	w	w	x	y	s

And the DFS tree will look like:



After running the DFS algorithm, the edges in the graph G can be categorized as follows:

- Tree edges - discovery edges, that is edges which are obtained when we go from a GRAY to a WHITE vertex and discover the WHITE vertex.
- Forward edges - edges $\overrightarrow{uv}$ such that u is an ancestor of v in the DFS tree, that is, there is a directed path from u to v in the DFS tree.
- Back edges - edges $\overrightarrow{uv}$ such that v is an ancestor of u in the DFS tree, that is, there is a directed path from v to u in the DFS tree.
- Cross edges - edges $\overrightarrow{uv}$ such that there is no directed path in the DFS tree from u to v or from v to u.

Lemma: We make the following observations for all edges $\overrightarrow{uv}$ in G:

- If its a tree edge, then $u.d < v.d$ and $u.f > v.f$.

- If its a forward edge, then $u.d < v.d$ and $v.f > v.f$.
- If its a back edge, then $u.d > v.d$ and $v.f > u.f$.
- If its a cross edge, then $u.d > v.d$ and $u.f > v.f$.

In the case of tree, forward and back edges, these follow from the observations about the DFS algorithm made before, because then there is a path or edge between u and v.
If $\overrightarrow{uv}$ is a cross edge, then first we note that $u.d$ cannot be less than $v.d$. Because otherwise, since there is a edge from u to v, v will be a descendent of u. So we conclude that $u.d > v.d$. If $v.f = *$ when u becomes active, then v will be an ancestor of u and we have a directed path from v to u in the DFS tree. Since this is not true, so, v must have finished before we even discovered u, and so, $v.f < u.f$.

We observe that when the DFS algorithm is run for an undirected graph, then the only non-tree edges can be Back-edges. This is because an edge uv between an ancestor u and a descendant v will be discovered at v first because v will be finished first, so it will be a Back-edge. There cannot be any cross edges because there can't be an edge between a finished BLACK vertex and a WHITE undiscovered one.

Lecture 4

DFS vs BFS, DAG, Topological ordering, Shortest paths in DAG.

BFS vs DFS

Recall that the BFS algorithm (done in ITC2) is different from the DFS in the way it explores the graph. It lists every unexplored neighbor of the current active vertex before exploring the next vertex, while the DFS only lists one unexplored neighbor and we immediately begin exploring this neighbor.
We look at some questions about the graph that the BFS and the DFS algorithms can help answer.

Problem	BFS	DFS
Is the (undirected) graph connected	yes	yes
Is there a path from s to t	yes	yes
Determine component of a vertex (undirected graph)	yes	yes
Determine vertices reachable from a fixed vertex (directed graph)	yes	yes
Find a spanning tree	yes	yes
Find the shortest paths from s in an unweighted graph	yes	**NO**

BFS seems to do everything that DFS could and plus some more. So why do we need DFS?
We notice that if the graph had edge weights then the BFS algorithm cannot give the shortest (weighted) paths from a fixed vertex s to all other vertices. In the special case if a directed graph is **acyclic** then we can devise an algorithm for it.

Single Source Shortest Path Problem

The Input for this problem is a graph G given as an adjacency list or matrix, such that all edges have weights. We also have a special designated vertex s. The problem is to find the shortest weighted paths from s to all vertices. We will store this shortest distance as a vertex attribute and we will denote it as $v.t$.

We will follow Cormen-Leiserson-Rivest to define an Initialization routine which we will reuse for different algorithms for the Single Source Shortest Path problem.

INITIALIZE-SINGLE-SOURCE(G,s)

1. **for** each vertex in $v \in V(G)$,
2. $\quad v.t = \infty$
3. $\quad v.p = *$
4. $s.d = 0$

Most graph algorithms for the Single Source Shortest Path Problem use some edge relaxation technique as we will describe below. (The code here is again from Cormen-Leiserson-Rivest). Note that w is the weight of the edge $\overrightarrow{uv}$ and when we are exploring the adjacency list and come across this edge then we will pass the weight attribute to the RELAX function. The basic idea is, if the current discovered distance to v is more than the distance of u along with the weight of the edge $\overrightarrow{uv}$, then we can update the distance to v with this new path.

RELAX(u,v,w)

1. **if** $v.t > u.t + w$,
2. $\quad v.t = u.t + w$,
3. $\quad v.p = u$

Notice: if we replace $>$ with $<$ in the if condition above, then we want to compute the longest paths from $s = v_1$.
With some examples we will see that the order in which the edges are relaxed is critical and will drastically effect our running time.

Suppose we can arrange the vertices of the graph as $s = v_1, ..., v_n$ such that all edges are from a smaller indexed vertex to a larger one, that is all edges $\overrightarrow{v_i v_j}$ are such that $i < j$. Such an ordering of the vertices, so that the edges only go forwards, is called a Topological Ordering of the vertices of the graph.

We notice now that if a graph has such an ordering, then relaxing all outgoing edges of the vertex v_1 then of the vertex v_2 and so on will give the shortest paths from v_1.

DAG-SHORTEST-PATHS(G,s)

1. Obtain $v_1, v_2, ..., v_n$, a topological sorting of the vertices of G
2. INITIALIZE-SINGLE-SOURCE(G,s)
3. **for** i from 1 to n
4. $\quad$ **for** every vertex v in $Adj[v_i]$
5. $\quad\quad w$ = weight of the edge $\overrightarrow{v_i v}$ (this is stored in $Adj[v_i]$)
6. $\quad\quad$ RELAX(v_i, v, w)

We will prove the correctness when $s = v_1$.
Invariant: Before vertex v_i is processed, that is, before the outgoing edges of v_i are relaxed, for all values of k, the attributes $v_k.t$ store the length of the shortest path from v_1 to v_k using only vertices $v_1, ...v_{i-1}$ on the path.

Initialization: When $i = 1$, $v_1.t = 0$ and for every other vertex there is no path from v_1 using no vertices, so the distance is ∞.
Maintanence: Assume that the invariant holds until the vertex v_i vertex. Now the algorithm will relax all outgoing edges of v_i. We know that for every vertex v_k, $v_k.t$ holds the length of the shortest path using vertices $v_1, ..., v_{i-1}$. When trying to find the length of the shortest path using vertices $v_1, ..., v_i$, we notice that either this shortest path uses v_i or it does not. If it does, then $v_i.t + w(v_i, v_k) < v_k.t$. But this is precisely what is updated when we relax all edges outgoing from v_i.
Termination: When v_n is processed, then we have length of the shortest path to all vertices from v_1 with paths using all vertices, which is precisely what we need.

We note without proof that the above algorithm gives the shortest paths from s even if $s \neq v_1$.

If we exclude the part of finding the topological ordering, then after that the algorithm relaxes every edge exactly once. So the running time is $O(n + m)+$ time taken to do the topological sort.

Topological ordering

Directed Acyclic Graph, or DAG for short, is a directed graph with no directed cycles.

Given a DAG, an ordering of the vertices $(v_1, v_2, ..., v_n)$ is said to be a topological ordering if for every edge $\overrightarrow{v_i v_j}$, $i < j$. In particular, there are two vertices s (source) and t (sink) which have only outgoing and incoming edges respectively.

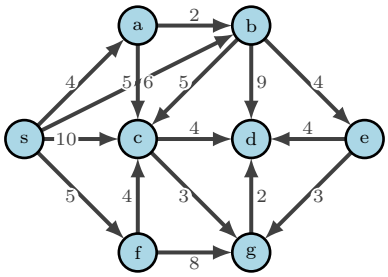
Theorem: A directed graph G has a topological ordering of the vertices if and only if it is acyclic.
Proof sketch: If a directed graph has a topological ordering, say f, then since for every edge $\overrightarrow{uv}$, $f(u) < f(v)$, it cannot have a directed cycle.
On the other hand, if the graph does not have any cycles, then we notice that there must be a vertex which has no incoming edges. Otherwise, we start at any vertex v, and pick a vertex from which it has an incoming edge, and we repeat this until a vertex repeats on this path, thereby giving us a cycle. Let us call this vertex with no incoming edges as v_1. We notice that $G \setminus \{v_1\}$ is also a directed acyclic graph, and so we can repeat this process and pick another vertex here that has no incoming edges. We will call this vertex v_2. We continue this process until we get the required ordering $v_1, v_2, ..., v_n$. We notice that this is a topological ordering of the graph because at any stage i, v_i was a source in the graph $G \setminus \{v_1, ..., v_{i-1}\}$ so all edges outgoing from v_i must be to $v_j, j > i$.

The above algorithm finds all shortest paths from v_1 in $c \cdot (n + m)$ steps, where $n = |V(G)|$ and $m = |E(G)|$.

We also observe that the DFS algorithm can decide if a graph G is acyclic and if yes, provide us with a topological ordering.

Theorem: Consider a directed graph G with the DFS algorithm run from vertex s. G is acyclic if and only if the DFS algorithm does not discover a back-edge. Also, if there were no back-edges, then the reverse of postordering of the vertices (the vertices ordered with decreasing $v.f$ value) gives a topological ordering of the vertices.
Proof sketch: it follows from the properties of all the types of edges discussed earlier, only back edges form directed cycles. Also, only for back edges $\overrightarrow{uv}$, $u.f < v.f$. For every other kind of edge, $u.f > v.f$.

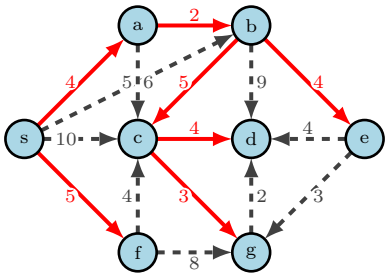
Consider the following graph:



The DFS algorithm when run on the above graph will give the following results:

v	s	a	b	c	d	e	f	g
$v.d$	1	2	3	4	5	7	8	6
$v.f$	8	6	5	3	1	4	7	2
$v.p$	*	s	a	b	c	b	s	c

And we will get the following tree:



The table below has the the distances of the vertices listed below them, and the first column shows the vertex being processed. By processed we mean that all the outgoing edges from this vertex will be relaxed. These edges are further listed in the second column.

Vertex	Edges	s	f	a	b	e	c	g	d
-	-	0	∞	∞	∞	∞	∞	∞	∞
s	sa, sb, sc, sf	0	5	4	5	∞	10	∞	∞
f	fc, fg	0	5	4	5	∞	9	13	∞
a	ab, ac	0	5	4	5	∞	9	13	∞
b	bc, bd, be	0	5	4	5	9	9	13	14
e	ed, eg	0	5	4	5	9	9	12	13
c	cd, cg	0	5	4	5	9	9	12	13
g	gd	0	5	4	5	9	9	12	13

Let $v.t$ denote the minimum distance and $v.T$ the maximum. The following table shows these distances.

v	s	f	a	b	e	c	g	d
$v.t$	0	5	4	5	9	9	12	13
$v.T$	0	5	4	6	10	11	14	16

Lecture 5

Dijkstra's algorithm, Heaps, Heapsort.

We will start with some examples where the relaxing edges once in some fixed order of vertices does not give the shortest paths.

Dijkstra's algorithm

If the graph G can have directed cycles, then our above algorithm will not work. If we add a restriction to the graph that all the edge weights are non-negative, then a greedy approach for relaxing the edges works. This is the Dijkstra's algorithm for SSSP. The key idea here is to maintain a list of distances to all vertices and to always pick the vertex with the least distance and then to relax all outgoing edges from it.

DIJKSTRA(G,s)

1. INITIALIZE-SINGLE-SOURCE(G,s)
2. $P = \phi$ Note: contains the set of processed vertices, initialy empty.
3. $Q = V(G)$ Note: this is just the complement of P.
4. **while** $Q \neq \phi$
5. Remove u the vertex in Q with minimum $u.t$ value.
6. $P = P \cup \{u\}$
7. **for** each vertex $v \in Adj[u]$
8. $w =$ weight of the edge $\overrightarrow{uv}$ (this is stored in $Adj[u]$)
9. RELAX(u, v, w)

Note: Here P contains processed vertices (whose edges have been relaxed), while Q contains vertices yet to be processed.

Let $n = |V(G)|$ and $m = |E(G)$. Suppose the graph is given as an adjacency matrix. We also need to store the sets P and Q. These can be stored as a vertex attribute, or seperately as a linked list.
We notice that at every loop, the algorithm has to find the minimum in the set Q. This is $O(n)$ steps for each loop in both cases, whether it is stored as an attribute or a linked list. So finding the minimum in all the loops together is $O(n^2)$ steps.
Also every edge is relaxed exactly once. For an adjacency matrix this is $O(n^2)$. This gives the total running time of $O(n^2)$.
If the graph were instead represented by an Adjacency list, still the running time will not change because it is storing and finding the minimum that is the bottleneck at $O(n^2)$. This along with $O(m+n)$ for relaxing each edge gives us $O(m+n+n^2) = O(n^2)$ running time.

We need one definition and a notational convention before proving the correctness of Dijkstra's.
Definition: Let $d(u, v)$ denote the shortest path between the vertices u and v.
We will use the notation that for any two vertices, if uv is an edge in the graph, then $w(uv)$ is the weight of the edge as defined by the weight function, but if there is no such edge in the graph, then we define $w(uv) = \infty$.

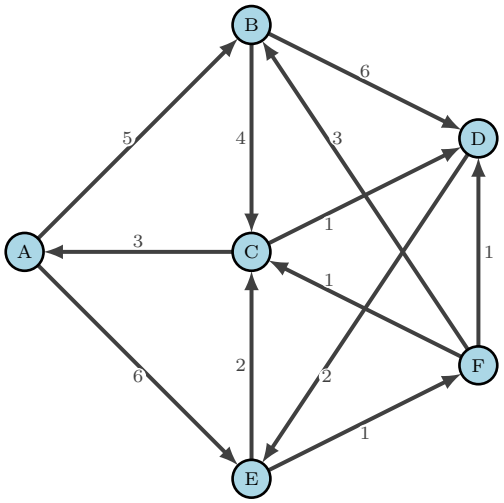
Invariant: At the beginning of every loop,

1. $\forall u \in P$, $u.t = d(s, u)$
2. $\forall u \notin P$, $u.t$ is the shortest distance of a s, u path that besides u uses only vertices in P. In other words it is the length of the shortest path in the subgraph induced on the vertices $P \cup \{u\}$.

Initialization: We initially set $P = \phi$ and Q to be the rest of the vertices. Further $s.t = 0$ and it is ∞ for every other vertex. This all satisfies the Invariant.
Maintanence: Suppose it is true at the beginning of a loop and now the algorithm picks v.
For the first statement, we need to check that $v.t = d(s, v)$. For this, we note that $u.t$ is (by induction) the shortest path to any vertex using only vertices in $P \cup \{u\}$. And v is the vertex with the smallest t value from the vertices in $V(G) \setminus P$. Since any path to v must exit P, the shortest s, v-path must

directly exit P to v, since if it exits P to another vertex u, then the path to v will be longer and it won't be the vertex with the minimum t value in Q. So $v.t = d(s, v)$.
For seeing that the second statement also holds, we need to check that on the graph induced on $P \cup \{u, v\}$, for any vertex $u \notin P$ and $u \neq v$, the updated $u.t$ is the shortest path. We notice that the way we update $u.t$ is by relaxing all edges from v to other vertices. So, $u.t = \min(u.t, v.t + w(vu))$. Since the only new path in this induced subgraph can be through v and we are updating t as the shorter of these two paths, we are making sure that $u.t$ remains the shortest path in $P \cup \{u, v\}$.

From the claim we can see that when the algorithm ends with $P = V(G)$, we will have found the shortest distances to all vertices.
Consider the following graph for the SSSP problem of finding shortest paths from A:



The following table shows the status of Q at every loop of the Dijkstra's algorithm. * indicates that a vertex is no longer in Q, and so has been added to P. The algorithm terminates when all vertices in Q have the same distance, since then the shortest distance cannot be further improved.

$s = A$	B	C	D	E	F
*	5	∞	∞	6	∞
*	*	9	11	6	∞
*	*	8	11	*	7
*	*	8	8	*	*

We will also keep a table to record the final shortest distances,

$s = A$	B	C	D	E	F
0	5	8	8	6	7

and another table to store the parents of vertices, so that we can also give the shortest paths to the vertices. For example the shortest path to D from A is $AEFD$.

$s = A$	B	C	D	E	F
*	A	E	F	A	E

Heaps

An immediate observation we have from the Dijkstra's algorithm implementation provided above is that the running time doesn't so much depend on the relaxing procedure as much as it does on finding the minimum. We only need to find the minimum distance in the set Q, and currently we look at each vertex in Q and look at the distance attribute $v.t$. And this way we spend $O(n)$ time in searching for the minimum because we take $O(n)$ time to go through the elements in Q. So the idea we can have to improve the running time is let Q contain tuples of vertices and their distances, but these are not stored as an array or a linked list, but as a heap.

Heap: A heap is a data structure where the underlying object is an array, but we view it as a complete binary tree. The vertices of the tree are elements of the array. We also require that all the levels except for the last are completely filled. So if there are n elements in the heap, then its depth is $\lceil \log n \rceil$.
We take as a convention that for any element $A[i]$ of the array, $A[\lfloor \frac{i}{2} \rfloor]$ gives the location of its **parent** (note that $A[1]$ does not have a parent). And $A[2i]$ gives the **left child**, while $A[2i+1]$ gives the **right child**.
We note that the heap A has two attributes $A.length$ and $A.heap\text{-}size$. $A.length$ is just the length of the array, while $A.heap\text{-}size$ represents the number of elements in the heap. Note that the heap-size can be strictly less than the length, but not the other way around.

Max-Heap Property: If the heap satisfies the property that the value of any vertex is larger than its children, then the heap is said to have the max-heap property and the heap is called a Max-Heap. Note that in this case, the root contains the maximum value element in the heap.
If we replace greater with smaller in the sentence above, then we get the Min-Heap Property and the heap is accordingly called a Min-Heap.

Building a Heap

Given a heap, suppose we want to insert an element. We will then put it at the end of the array and percolate it up if its big. The UPHEAPIFY procedure does the upwards percolation.

UPHEAPIFY(A,i)

1. **if** $i = 1$ return
2. $p = \lfloor \frac{i}{2} \rfloor$
3. Let $max =$ maximum of $A[i], A[p]$
4. **if** $A[p] \neq max$
5. swap $A[i], A[p]$
6. UPHEAPIFY(A,p)

And now we can Insert into the heap:

INSERT(A,v)

1. $A.heap\text{-}size = A.heap\text{-}size + 1$
2. $A[A.heap\text{-}size] = v$
3. UPHEAPIFY(A,$A.heap\text{-}size$)

Then we can build a heap from an array by inserting one element at a time. Note, we do everything in place and we are not using any new storage space.

BUILD-MAX-HEAP(A)

1. $A.heap\text{-}size = 0$
2. **for** $i = 1$ to n
3. INSERT(A,A[i])

Invariant: We maintain the property that before the i^{th} loop, the first $i-1$ elements satisfy the heap-property.
Initialization: Initially heap-size is 0, so true.
Maintanence: the upheapify procedure maintains the property after the new element is inserted.

We run the upheapify procedure for n nodes, each taking at most $O(\log n)$ time, so the running time is $O(n \log n)$.
Note: Please check the text book for a better running time analysis to see that the heap is actually built in $O(n)$ time.

Assuming we have a heap, we can delete the maximum element by placing the last element in the position of the first and using a heapify procedure which shifts this smaller element down the heap.

DOWNHEAPIFY(A,i)

1. Let max, j = value, index of maximum of $A[i], A^*[2i], A^*[2i+1]$ ($*$ we can only compare if the indices are $\leq$ A.heap-size)
2. **if** $A[i] \neq max$
3. swap $A[i], A[j]$
4. DOWNHEAPIFY(A,j)

Now we can delete an element in the heap as follows:

DELETEMAX(A)

1. swap values of $A[1]$ and $A[A.heap\text{-}size]$
2. $A.heap\text{-}size = A.heap\text{-}size - 1$
3. DOWNHEAPIFY(A,1)

Notice that all the above procedures can easily be modified for the min-heap property.

Deletion running time is $O(\log n)$ as we move once along the height of the heap.

Heapsort

HEAPSORT(A)

1. BUILD-MAX-HEAP(A)
2. **while** $A.heap\text{-}size > 0$
3. DELETEMAX(A)

Deletemax takes $O(\log n)$ time and we repeat it n times, so it is $O(n \log n)$.

To see the correctness, we won't write an invariant but just notice that this proceeds like selection sort, where we choose the maximum and place it at the end. We have already shown the correctness of all other heap procedures. Here since we are picking the maximum of the remaining numbers at every turn and storing it at its appropriate place, we conclude that this sorting algorithm is also correct.

Lecture 6

Sorting Algorithms

Before we get back to sorting algorithms, we will show how to construct a heap in $O(n)$ time.

Building a Heap in $O(n)$ time

Depth and Height: For any element in the heap, we say that its depth is the level it is from the top, with the root being at depth 0, and the leaves at depth $\lfloor \log n \rfloor$). We also say that its height is its level counted from below, so the leaves are at height 0 and the root is at height $\lfloor \log n \rfloor$).

If we analyse our BUILD-MAX-HEAP algorithm, we notice that the amount of work done in inserting any element depends on its depth, since the UPHEAPIFY procedure takes, in the worst case, as many steps as the depth of the element. In a heap with $2^{k+1} - 1$ elements, 2^k elements are leaves, and have depth k. So we will do $k2^k$ amount of work in inserting, and this is $n \log n$. If we want to improve the algorithm, then the first thing we can notice is, instead of building the heap top down, we can build it bottom up. Then the leaves will only use constant amount of work with DOWNHEAPIFY.

LINEAR-BUILD-MAX-HEAP(A)

1. $A.heap\text{-}size = A.length$
2. **for** $i = \lfloor A.length/2 \rfloor$ **downto** 1
3. DOWNHEAPIFY(A,i)

Invariant: at the beginning of a loop, all the elements $A[i+1 : n]$ have the heap property. Initialization, Maintanence: exercise for the reader!

For simplicity, we will assume $n = 2^{k+1} - 1$, that is there are $k+1$ levels with root having height k and the leaves having height 0. The argument is quite similar without this assumption.

For any element of height h, the DOWNHEAPIFY procedure takes h steps. And we notice that there are 2^{k-h} elements of height h. Then the running time of building the heap is $\sum_{h=0}^{k} 2^{k-h}h = 2^k \sum_{h=0}^{k} \frac{h}{2^h}$.

To see how this sum works, let $S = \sum_{h=0}^{k} \frac{h}{2^h}$. Then $2S - S = \sum_{h=0}^{k-1} \frac{1}{2^h} - \frac{k}{2^k} \leq 2$.

Using this, the running time is $\leq 2 \cdot 2^k = O(n)$.

We had so far: Selection sort $O(n^2)$, Bubblesort $O(n^2)$, Mergesort $O(n \log n)$, and last week: Heapsort $O(n \log n)$.

Quicksort

We will follow this exactly as in Cormen-Leiserson-Rivest.

QUICKSORT(A,p,r)

1. **if** $p < r$
2. $\quad q =$ PARTITION(A,p,r)
3. $\quad$ QUICKSORT(A,p,q-1)
4. $\quad$ QUICKSORT(A,q+1,r)

And to sort the entire array, we will call QUICKSORT(A,1,A.length). The key to QUICKSORT is the PARTITION function which moves all elements less than $A[r]$ to positions smaller than q and all bigger elements to positions bigger than q and finally it moves the element $A[r]$ to the position q. It then returns this value q.

PARTITION(A,p,r)

1. $x = A[r]$
2. $i = p - 1$
3. **for** $j = p$ to $r - 1$
4. $\quad$ **if** $A[j] \leq x$
5. $\quad\quad i = i + 1$
6. $\quad\quad$ swap $A[i]$ and $A[j]$
7. swap $A[i + 1]$ and $A[r]$
8. **return** $i + 1$

We will also do the Invariant exactly as in the book. We will prove the correctness of the partition function and the correctness of quicksort follows directly from it.

At the beginning of the every loop for j, for every array value $A[k]$, the following is true:

- If $p \leq k \leq i$, then, $A[k] \leq x$
- If $i + 1 \leq k \leq j - 1$, then, $A[k] > x$
- If $k = r$, then, $A[k] = x$

QUICKSORT is an algorithm with a good average case running time. It is easy to see the worst case running time is $O(n^2)$ if the parition is always imbalanced. On the other hand, for even the best possible input, the partition function creates two problems of half the size, so the best case running time is $O(n \log n)$. We will use without proof the the average case running time of QUICKSORT is $O(n \log n)$

The average case running time of QUICKSORT is $O(n \log n)$. (No proof).

Lower bounds

An important question to consider is whether there exist comparison-based sorting algorithms with better running time than $O(n \log n)$, and the answer to that is no. It is impossible to have a better running time if we sort our array with a single step comprising of comparing two elements of the array.

Theorem: Comparison-based sorting algorithms cannot have a better running time than $O(n \log n)$.

Proof: Note that sorting n distinct numbers is identical to finding the permutation of the numbers that results in a sorted array. So we are searching for one permutation in the space of all permutations of n numbers. Each comparison effectively splits this collection of permutations into two groups (because say we compared $A[i]$ and $A[j]$ elements, we know that in every permutation, either $A[i] < A[j]$ or $A[i] > A[j]$ so they must fall in the two groups). And the outcome of the comparison is that we pick one of the two groups. However nicely we split the existing permutations at any stage, a comparison will get rid of at most half of them. We know that for n numbers there are a total of n permutations. Then after the first comparison, we have at least $\frac{n}{2}$ permutations left. After the second comparison, at least $\frac{n}{4}$, and so on. We want to eventually have exactly 1 permutation left. Then if we made k comparisons, we must have $\frac{n}{2^k} \leq 1$, and this gives us that $k \in \Omega(n \log n)$.

Binsort

If we know that the array $A[1:n]$ has distinct values in the range $1:m$, then we can sort it in $O(m)$ time using Binsort.

BINSORT(A)

1. $m =$ MAX(A)
2. $B[1:m]$ is an array of 0's
3. **for** $i = 1$ to $A.length$
4. $\quad B[A[i]] = 1$
5. $k = 0$
6. **for** $j = 1$ to m
7. $\quad$ **if** $B[j] == 1$
8. $\quad\quad k = k + 1$
9. $\quad\quad A[k] = j$

We will not write an invariant for Binsort. Instead we just note that if a number x is in A, then $B[x]$ will be set to 1 and for every number not in A, the $B[x]$ value will remain 0. In the next step we just write back in A the numbers that have a 1 value in the B-array.

Since $n < m$ and we go through the array A and B once each, so the running time is $O(m)$.

We will also look at a version of Binsort where there can be multiple entries with the same key value. We will also give an algorithm that is a **stable** sorting algorithm.

Stable sorting algorithm: is one in which the order of elements with the same value is not changed during the sort.

BINSORT(A)

1. $m =$ maximum bin number for the elements of A
2. $B[1:m]$ is an array of empty linked lists.

3. **for** $i = 1$ to $A.length$
4. $x =$ the bin number of $A[i]$
5. add $A[i]$ to the end of the list at $B[x]$
6. **for** $j = 1$ to m
7. Add the elements in $B[j]$ to A sequentially

It takes $O(n)$ time to fill the bins, and then $O(n+m)$ time to copy the elements back to A, so its running time is $O(n+m)$. If m is smaller than n, then this is just $O(n)$.

Radixsort

RADIX-SORT(A,d)

1. **for** $i = 1$ to d (where $i = 1$ is the least significant digit)
2. BINSORT(A) with i^{th} least significant digit as the bin number

For decimal numbers, there are 10 bins everytime, so each loop takes $O(n)$ time where the length of A is n. If the numbers in A have d digits, then the total running time is $O(nd)$. In general, if there are k letters in the alphabet, words are d characters long, then sorting n words with radixsort takes $O(ndk)$ time. And if d, k are both contants, then we have a linear running time.

Invariant: At the beginning of the i^{th} loop, the words/numbers truncated to the last $i-1$ digits are sorted.

Initialization: Initially $i - 1 = 0$, so all words/numbers are empty and they are trivially sorted.

Maintanence: The words/numbers truncated to last $i-1$ bits are sorted, and then we use the correctness of bin sort at the i^{th} loop to see that after this loop, the i^{th} bit will be sorted. But because binsort is a stable sort, if we look at all the words having the same i^{th} bit, their order has not changed. So since they were sorted before this loop, so at the end of the i^{th} loop, the words/numbers truncated to the last i bits are also sorted.

The way we have described Radix sort, we have assumed that the words or numbers are all of the same length. If this is not the case, then we can modify our input based on our requirements. If we had words and want lexicographic sorting (as in a dictionary), then we can pad the words at the end with a new dummy characted that we consider to be the first in the alphabet order. If we have numbers and we want to sort them, then it makes more sense to pad them with 0's at the beginning.

Lecture 7

Binary Search Trees, Red-Black Trees

Consider that we have some data we want to store in a data structure. We assume that this data has a key we index it with, where usually keys are distinct numbers. The basic operations we then would like to be able to do with the data structure are: Inserting a key, deleting a key, searching for a key, modifying a key, deciding the maximum and minimum key, sorting with respect to the key value. We will compare all the data structures we have seen so far and see how efficient they are with these operations. (We haven't added sorting to the table below because it usually is an aggregate of n insertions).

Assume that the linked lists are doubly linked lists.

Operation	Linked List	Array	Sorted Linked List	Sorted array	Heaps
INSERT	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(\log n)$
SEARCH (for a key)	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(n)$
DELETE (by key)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
DELETE (by position)	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$
MODIFY (by key)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
MODIFY (by position)	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(\log n)$
MAX/MIN	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(\log n)$

As we can see, all the above data structures have a $O(n)$ performance time for some operations, so the question is really if we can design a better data structure where each of the above operations will be $O(\log n)$. In particular, we see above that if the search is $O(\log n)$ then insert/delete/modify are $O(n)$, and vice versa. So the question really is can be design a data structure so that both search and the insert/delete/modify operations must be $O(\log n)$.

Binary Search Trees

First we recall that a Binary Tree is a data structure where one node is designated as the root (T.root). Every node x has three pointers, one to a parent $x.p$, one to the left child $x.left$ and lastly, one to the right child $x.right$. If a node doesn't have a left or right child or a parent, then the corresponding pointer is a NULL/NIL pointer. The root is the only node that has a NIL pointer for the parent.

A binary tree is called a **Binary Search Tree** if for every node x and for any node y in its left subtree, we have $y.key \leq x.key$. Similarly, for any node y in the right subtree, we must have $x.key \leq y.key$.

Ordered Walks

Given a binary search tree, there are three ways to walk through the entire tree and list the elements, each having its uses.

```
INORDER-TREE-WALK(x)
1. if x ≠ NIL
2.     INORDER-TREE-WALK(x.left)
3.     print(x.key)
4.     INORDER-TREE-WALK(x.right)
```

To get a Inorder tree walk, we call the function INORDER-TREE-WALK(T.root).

Every node in the tree is visited atmost 3 times, one from its parent, once when returning from the recursion of the left child and finally, when returning from the recursion of the right child. At each of these three visits, we do constant amount of work, so the running time is $O(n)$. And because there are n vertices and we do visit them all, the running time is in fact $\Theta(n)$.
The PREORDER and POSTORDER can be obtained with minor modifications to the code as follows:

PREORDER-TREE-WALK(x)

1. **if** $x \neq$ NIL
2. print(x.key)
3. PREORDER-TREE-WALK(x.left)
4. PREORDER-TREE-WALK(x.right)

POSTORDER-TREE-WALK(x)

1. **if** $x \neq$ NIL
2. POSTORDER-TREE-WALK(x.left)
3. POSTORDER-TREE-WALK(x.right)
4. print(x.key)

Observation: We can see that the Inorder of a Binary search tree prints the keys in sorted order. This follows directly from the Binary search tree property.

Basic Operations

Here we will see how to perform Insert, Delete, Search, Modify and Max/Min on Binary Search Trees. But to able to talk about the running time, we will use the height of the BST, denoted by h, where we define the height of the tree as the height of the root vertex.

Height of a vertex is defined as the length of the longest path from the vertex to a leaf.

Depth of a vertex is defined as the length of the path from the vertex to the root.

BST-SEARCH(x,k)

1. **if** $x ==$ NIL or $x.key == k$
2. **return** x
3. **if** $k < x.key$
4. **return** BST-SEARCH(x.left, k)
5. **else return** BST-SEARCH(x.right, k)

Please check the book for an iterative version of the algorithm. Notice that worst case running time of search is through the height of the tree, so $O(h)$.

BST-MINIMUM(x)

1. **while** $x.left \neq$ NIL
2. $x = x.left$
3. **return** x

BST-MAXIMUM(x)

1. **while** $x.right \neq$ NIL
2. $\quad x = x.right$
3. **return** x

Its easy to see these take time $O(h)$.

BST-SUCCESSOR(x)

1. **if** $x.right \neq$ NIL
2. $\quad$ **return** BST-MINIMUM($x.right$)
3. $y = x.p$
4. **while** $y \neq$ NIL and $x == y.right$
5. $\quad x = y$
6. $\quad y = y.p$
7. **return** y

Again the running time is $O(h)$.
For Insertion, we note that, every time a new element is inserted, it would become a leaf of the Tree. This is because, since we have distinct keys, for every node x the key value we want to insert is $> x.key$ or $< x.key$ and based on this we can further check for its place in $x.right$ or $x.left$. This is very much like the SEARCH, only since the new key is not in the tree, it will end in a leaf. First we modify the SEARCH to report the parent also.

BST-SEARCH-P(x,y,k) Note: here y is the parent of x

1. **if** $x ==$ NIL or $x.key == k$
2. $\quad$ **return** (x,y)
3. **if** $k < x.key$
4. $\quad$ **return** BST-SEARCH(x.left,x, k)
5. **else return** BST-SEARCH(x.right,x, k)

Now we can give the INSERT algorithm.

BST-INSERT(T,z)

1. **if** $T.root ==$ NIL
2. $\quad T.root = z$
3. $\quad$ **return**
4. (x, y) = BST-SEARCH-P($T.root$, NIL, $z.key$)
5. **if** $x \neq$ NIL
6. $\quad$ **return**

7. **if** $z.key < y.key$
8. $\quad y.left = z$
9. **else** $y.right = z$

Again the running time is $O(h)$.
Deletion of a node is straight forward if the node is a leaf (then we just set the parent's child pointer to NIL), or if it has one child (then this child will become the child of the parent). But when the node has two children, then our recursive algorithm would be to the successor and replace it with the node and delete the successor.

BST-DELETE(x,z)

1. **if** z doesn't have any children
2. $\quad$ If z was the root, return the empty tree
3. $\quad$ else: Delete z by setting its parent's child pointer to NIL
4. **else if** z has one child,
5. $\quad$ if z was the root, set its child as the new root.
6. $\quad$ else: set $z.p$'s child pointer to z to instead point to the child of z
7. **else if** z has two children,
8. $\quad$ let a = BST-SUCCESSOR(z)
9. $\quad$ set $z.key == a.key$ and copy any other attributes of a to z.
10. $\quad$ BST-DELETE($z.right$,a)

We call BST-DELETE(T.root, z) to delete z.
Here we note that the BST-DELETE only moves down the structure of the tree, so, its running time is $O(h)$. We see that the running time depends quite heavily on the height. With examples, we can see that this can be $O(n)$ where n is the number of elements. So we will try to design a better search tree.

Red-Black Trees

All nodes in Red-Black trees are Binary Search Trees with an additional attribute, all nodes have a colour, either red or black. We have the following constraints:

(NOTE: We think of all NIL pointers as leaves and every node with a key value as an internal node.)

- Root is black
- Every leaf (NIL) is black
- Both children of a red node are black.
- For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Black-height of a node v is defined as the number of black vertices on a path from v to a descendant leaf. (Note, we can consider any path because all such paths have the same number of black nodes).

We note that if the black-height of the root is h, then since every path to a leaf contains h black vertices, the total vertices in the tree must be $2^{h+1} - 1$, since the tree is complete until depth h. Further, since both children of a red vertex are black, no path can be of length more than $2h - 1$ (since root is also black), so

the maximum number of nodes can be at most $2^{2h} - 1$. So we see that the height of a Red-Black tree is $O(\log n)$ if it has n elements. This will hopefully ensure that all our operations are of running time $O(\log n)$.

Theorem: For a red-black tree with n elements, its height is $O(\log n)$.

We note that the operations: SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, PREORDER work like before. Because there we do not modify the tree. The only two operations we must consider here are INSERT and DELETE, and further maintain the Red-black properties listed above.

Rotations

We follow the original BST-INSERT here also and add a new entry. We always make the color of the new vertex red. If its parent was black, then we have not violated any of the properties we required. The only problem is if the parent of this new vertex was also red. This is then handled by what are called rotations.

This can be seen on pg 313, Figure 13.2. Right rotation: Let y be a node with left child x and label subtrees of x as α, β and the right child of y as γ. Then a c**Right rotation** will make y as x's right child. left child of x is α as before. Children of y are β and γ.

Left rotation is this done backwards.

When we insert an element, say z, we trace the tree to a leaf until we find its position, there we want to insert it as a red vertex. The only exception to this is when z becomes the root (its the first key to be inserted in an empty tree). Then we color it black. In all other cases, it has a parent x and we insert z as its child as a red vertex. The following are the possible outcomes then:

- x is black. Then there is nothing left to do, the black-heights are not modified.
- x is red. x cannot be the root, so it must have a parent, say t and let the other child of t be y. Note that t must be black. If y is also red, then we color t as red and both x and y as black and we have shifted the problem a level higher. This is page 320, Figure 13.5 of the book.
- x is red, t its parent is black, but y the other child of t is also black. Here we need at most two rotations and one change of coloring. In this case, the problem is solved here and not shifted to higher levels. This is on page 321, Figure 13.6 of the book.

We observe then that:
(a) Atmost $O(\log n)$ color swaps happen, as the problem moves higher on the tree. (b) Atmost two rotations are needed.

Deletions

Deletions are similar but a little more involved.

For deletions also, we will first follow our original BST-DELETE procedure while keeping the colors of the tree nodes intact even though the key and other attributes change during swap with the successsor. In the end we arrive at a node to delete which either has no children or has at most one child. At this point
If y is red, then we simply delete it as before and it will not violate any properties.
If y is black, then deleting it will reduce the black-height of this path to a leaf. In this case, the node replacing y, we call it x, will be said to have an "extra" black point. Note that x may be the NIL pointer. The rest of the algorithm is about how to push this extra black point above into the tree and eventually make all black-heights same. Let s be the sibling of y.
The color fix-up algorithm does the following:

- x is the root. Then we forget the extra black point. All paths to leaves have the same black height and we are done.
- If either x or y is red, then x when replacing y will become black (or stay black) and we are done.

- If both x, y and s are black, but one s' children is red (where s is the sibling of y), then we rotate to increase the black-height for y and then we color this child black to maintain the black-height on the path from s and this child. Again here we are done.
- If x, y, s and both of s' children are black, then we color s red and push the extra black point up to the parent to y and say that x (replacing y) does not have the extra black point. Here the extra black point has been pushed up, so we will have to repeat this fix-up operation at the parent.
- Is s is red, then we can rotate and recolor to resolve the problem. Here again we are done.

We note here that the fix-up either recolors and pushes the problem upwards or it fixes the problem with utmost three rotations. So fix-up requires at most $O(\log n)$ color changes and at most 3 rotations. So the running time for deletion is $O(\log n)$.

Lecture 8

2-3 Trees, Hash tables

2-3 Trees

This section will not be very detailed because an excellent overview of 2-3 Trees can be found here.
2-3 are search trees where every node can store one or two values and can therefore have 0, 2 or 3 children. When a node has a single value, then as in a binary search tree, all key values in the left subtree are smaller the value of the node, while all key values in the right subtree are bigger than it.
In case of a 3-node, it stores two values, say v_1, v_2, and it has three subtrees coming from it. All elements in the left subtree are smaller than v_1, the ones in middle subtree are bigger than v_1 but smaller than v_3 while the key values in the right subtree are all greater than v_2.
A node storing just one value, and thereby having 2 children (maybe both are NIL pointers) is called a 2-node. while one with 3 children a 3-node.

Uniform depth: An important property of 2-3 trees is that for any vertex v, all the paths leading to a leaf have the same length. Or in other words, the depth of all leaves is the same. So it is a balanced search tree.

Given a 2-3 tree with n stored items, and if h is the height of the tree, then $2^h - 1 \leq n \leq 3^h - 1$.

To see this is true, the minimum number of vertices in a tree of height h is if all its nodes are 2-nodes, and this is $2^h - 1$. The maximum is if all nodes are 3-nodes, and then this number is $2 + 2 \cdot 3 + 2 \cdot 3^2 ... + 2 \cdot 3^{h-1} = 3^h - 1$.

A consequence of this is that the height of a 2-3 tree with n elements is $O(\log n)$.

23T-SEARCH(x,k)

1. **if** x is a 2-node
2. **if** $x ==$ NIL or $x.key == k$
3. **return** x
4. **if** $k < x.key$

```
5.            return 23T-SEARCH(x.left,k)
6.        else return 23T-SEARCH(x.right,k)
7.  else if x is a 3-node
8.        if x == NIL or x.leftkey == k or x.rightkey == k
9.            return x
10.       if k < x.leftkey
11.           return 23T-SEARCH(x.left, k)
12.       if x.leftkey < k < x.rightkey
13.           return 23T-SEARCH(x.middle, k)
14.       else return 23T-SEARCH(x.right, k)
```

Insertion and Deletion

The first step of insertion is exactly like for BST, where we search through the tree until we reach a leaf and found a position for our new key value. If this leaf is a 2-node, then insertion is easy, we just convert it into a 3-node.

If the leaf was instead a 3-node, then inserting this new key value will temporarily make a 4-node. The idea how to deal with a 4-node is to push up the middle key value. This pushing up procedure may continue all the way to the top of the tree, and if it does, then the height of the tree increases by one.

For deletion of a node, we replace it with its successor until we reach a leaf. Wehn we delete this leaf, we will get a **hole**. The algorithm now is to propagate up the hole. If the hole propagates all the way to the root, then the height of the tree reduces by 1.

Since we traverse the height of the tree once in both insertion and deletion, their running time is $O(\log n)$.

Hash tables

In a variety of applications, we find that sorting is not so much needed, but we need data structures with fast insertion, deletion and search. For example health database based on TAJ number, or citizen data in a country using the identity document number etc. In these, it is really not required, for example, to sort the data according to TAJ number or identity number, but instead we should be able to perform fast insertions, deletions and search. Also, we may assume that the keys we encounter are distinct.
In this case we see that the bin-array used in Binsort is probably the most time efficient way to store the data. That is, we create an array $B[1:m]$ where m is the max value we will encounter, and $B[i] = 1$ if key value i is in the data. In this case Insertion, Deletion and Search are all $O(1)$ operations. This method is called *Direct addressing*. But we also note that this uses $O(m)$ space to store the n values, and it might be very wasteful. So we need a data structure which has a smaller size, and yet, it can obtain running times comparable to the method usng B. We will use hash tables and will check their efficiency.

A hash table is essentially an array $T[0:m-1]$, but instead of direct addressing, we will use a hash function to determine the address of keys. Let the universe of keys be U. Then the **hash function** is a function $h : U \to \{0, 1, ..., m-1\}$, such that for any key k, $h(k)$ determines the address of the key in the array T. $h(k)$ is called the **hash value**.

We define the **load factor** α for the hash table as $\frac{n}{m}$, where n is the number of values it stores, while m is size of the hash table.

Ideally, the hash values $h(k)$ for the universe U give us a permutation of $\{0, 1, ..., m-1\}$. But in practice, this is often not the case, and it happens rather frequently that for two different keys k_1 and k_2, $h(k_1) = h(k_2)$. This is called a **collision**. We will see several hash functions and the strategies used for collision resolution.

Chaining

When we resolve collisions with chaining, then each element of the hash table $T[0 : m-1]$ is a linked list, and when we insert an element with key k, we will append it to the beginning of the linked list with address $T[h(k)]$. For searching for a key k, we will search the list $T[h(k)]$, so we can see the running time of insert is $O(1)$, but for search and delete, it depends on the size of the list at the address $h(k)$. The worst case behaviour is $\Theta(n)$ when all elements are at this address.

Simple uniform hashing: when all the elements are equally like to be any slot, independently of the other elements.

With simple uniform hashing, the expected length of the list at any slot is $\frac{n}{m}$, which is the load factor α, and so the average case running time of search and delete is $\Theta(1 + \alpha)$.

So we see that when the number of elements is proportional to the size of the hash table, then the average performance of the hash table is as good as direct addressing.

Open addressing

An **open-address** hash table is one in which there are no linked lists anymore, but the elements occupy the hash table T itself. In open-addressing, when there is a collisionduring insert, we use a **probe** (method/function) to find another empty slot in the hash table. The sequence of probes *depends on the key being inserted.* We will denote the probe sequence with $h(k, i)$ where k is the key and i is the i^{th} probe, $0 \leq i \leq m-1$.

We require that the probe sequence $\{h(k, 0), h(k, 1), ..., h(k, m-1)\}$ should be a permutation of $\{0, 1, ..., m-1\}$ because it does not make sense to probe a cell that already led to a collision.

Insertion: with open-addressing, inserting a key k requires probing $h(k, 0), h(k, 1)...$ until we reach an empty cell and insert it there.

Deletion: for deleting a key k, we again probe $h(k, 0), h(k, 1)...$ until we find the element and then we mark that cell as **DELETED**. Note: it is important that we do not set the cell as EMPTY, but instead as DELETED.

Search: when searching for a key k, we again probe $h(k, 0), h(k, 1)...$ until we find the element. But the important thing to note here is we return NOT FOUND only if we hit a cell that is EMPTY. If we visit a DELETED cell during the probe, then the probe continues to the next address.

We will look at three different probes for collision resolution.

Linear probing

In a linear probe, the probe sequence we use is $h(k), h(k)-1, h(k)-2, ..., h(k)-(m-1)$. Note: all these addresses are considered modulo m, so in case we reach the beginning of the array, we will continue at the end of the array.
It is easy to see that the probe sequence is always a permutation of $\{0, 1, ..., m-1\}$. The problem with Linear probing is that it often leads to clusters.

Quadratic probing

In a quadratic probe, the probe sequence we use is $h(k), h(k)+1^2, h(k)-1^2, h(k)+2^2, h(k)-2^2, ..., h(k)+(\frac{m-1}{2})^2, h(k)-(\frac{m-1}{2})^2$. Note: all these addresses are again considered modulo m.

Theorem: If m is a prime of the form $4k+3$, then there is no n for which $n^2 \equiv -1 \pmod{m}$.

Proof: If there was such a n, then we note that, $-1 \equiv (-1)^{(\frac{m-1}{2})} \equiv (n^2)^{(\frac{m-1}{2})} \equiv n^{m-1} \equiv 1 \pmod{m}$
The last equivalence uses Fermat's little theorem that for any prime p and a number a, $a^{p-1} \equiv 1 \pmod{m}$.

We can use this to show the following theorem:

The quadratic probing sequence is a permutation of $\{0, 1, ..., m-1\}$.

Proof: For $0 \leq i < j \leq \frac{m-1}{2}$, $i^2 \neq j^2 \pmod{m}$, because $i^2 - j^2 = (i-j)(i+j)$ and we know m is a prime and $i+j < m$.
The same reason works for $0 \geq i > j \geq -\frac{m-1}{2}$.
Lastly, if $i^2 \equiv -j^2 \pmod{m}$, then, $(ij^{-1})^2 \equiv -1 \pmod{m}$, which we have just shown is not possible.

Double hashing

In double hashing, we have two has functions h, h', and the probe sequence we use is $h(k), h(k)-h'(k), h(k)-2h'(k), h(k)-3h'(k), ..., h(k)-(m-1)h'(k)$. Note: all these addresses are again considered modulo m.
Here we just note that as long as $h'(k)$ is relatively prime with m, this sequence will be a permutation of $\{0, 1, ..., m\}$. One commonly used function to achieve this is $h'(k) = k \pmod{m-1}$. It is easy to see that this indeed produces a permutation.

Lecture 9

Kruskal's algorithm, Prim's algorithm

Meta algorithm

For the minimum weight spanning tree problem (MST), we are given a graph $G(V, E)$ and a weight function, $w : E \to \mathbb{R}$, we want to find a spanning tree with the smallest total weight. As a side note: here it is not required to have positive weight edges, since the number of edges in a spanning tree are $n-1$, we can add a large constant c to all edge weights and decrease the total tree weight by $c(n-1)$ to get an answer to the original question. But the algorithms we propose would not require even that, since we never condition on the weight of edges to be positive.

We consider the following meta algorithm for MST:

GENERIC-MST(G,w)

1. $A = \phi$
2. **while** A does not form a spanning tree
3. select a cut $(S, V-S)$ of the graph such that no edge of A crosses it
4. add a minimum weight edge crossing this cut to A

Invariant: At the beginning of every loop, the set A consists of edges that are a subset of some minimum weight spanning tree.

Initialization: trivially true.

Maintanence: Let T be a minimum weight spanning tree containing A. Let uv be the new edge added across a cut $(S, V - S)$ where none of the edges in A cross the cut. If uv is in T, then there is nothing left to prove. But if uv is not in T, adding it induces a cycle, which must have another edge, say xy crossing the cut. This cannot be in A and since uv was a minimum weight edge, removing xy and adding uv gives us another minimum weight spanning tree.

The meta algorithm demonstrates that the order of picking the cut across which we take the minimum weight edges does not matter. Any cut will work. We will look at two variations of this, emphasizing the difference in the running times.

Kruskal's algorithm

In Kruskal's algorithm, we always pick the smallest edge in all possible remaining cuts. We do this by ordering edges and we pick an edge if it is in a cut. Correctness of Kruskal's follows from the correctness of the meta algorithm.

KRUSKAL-MST(G,w)

1. $A = \phi$
2. Let E be a data structure containing the edges.
3. **while** A does not form a spanning tree
4. let e = EXTRACT-MIN(E)
5. **if** e is part of a cut, add e to A (since edges are ordered, it is the minimum weight edge of this cut)

For the running time, we must first specify some implementation details. Kruskal's algorithm requires looking at the edges in an increasing weight order. This can either be done by sorting the edges, or by maintaining a min-heap.

We must also decide if the edge we picked is part of a cut or not. We notice now that A divides the graph into components, say $C_1, C_2, ..., C_k$. Then since none of the edges of A are in the cut $(S, V \setminus S)$, we know that each $C_i \subset S$ or $C_i \subset (V \setminus S)$. Then necessarily, an edge from the cut must have both its endpoints in different components. So a natural and fast way to check if the edge is part of a cut, we must keep a data structure storing the component number of each vertex and just compare this number for the endpoints of the edge. The component numbers can be stored as an array $C[1 : n]$, Where $C[i]$ is the component number of vertex v_i. Initially, this array is $C[i] = i$, $\forall 1 \leq i \leq n$.

Running time of Kruskal's:

Sorting the edges: is $O(m \log m)$ which is $O(m \log n)$. Even with a heap implementation, deletemin is $O(\log m)$ and it is repeated m times, so it is essentially the same.

Component check and update: For each edge considered, we check the component numbers of its enpoints from the array C. If they are the same, then we discard the edge and go to the next one. If it is different, then we must merge the two components. This can be done by updating the larger component number to the smaller. Lookup in array is $O(1)$, but if we merge two components, then merge takes as much time as the size of the components, which can be $O(n)$. Since we repeat this step n times (we cannot merge components more time than the number of edges in the spanning tree), so in

total this step takes $O(n^2)$ time.

Total Running Time: $O(m \log n + n^2)$.

Prim's algorithm

In Prim's algorithm, we always expand the same component. We initialize one component with a speacial start vertex $C = \{s\}$. Then at every stage we add the minimum weight edge between C and $V \setminus C$. Correctness of Prim's also follows from the correctness of the meta algorithm.

PRIM-MST(G,w,s)

1. **for** each $u \in G.V$
2. $\quad u.key = \infty$
3. $\quad u.p = NIL$
4. $s.key = 0$
5. $Q = G.V$
6. **while** $Q \neq \phi$
7. $\quad u =$ EXTRACT-MIN(Q)
8. $\quad$ **for** each $v \in Adj[u]$
9. $\quad\quad$ **if** $v \in Q$ and $w(u, v) < v.key$
10. $\quad\quad\quad v.p = u$
11. $\quad\quad\quad v.key = w(u, v)$

We note here that like in Dijkstra's algorithm, we will keep a record of the minimum weight edge leading to every vertex from the current C component to every vertex in Q, the set of vertices not reached yet by the spanning tree. At every stage, we find the minimum in Q, this gives the vertex u with the minimum weight edge $(u.p, u)$ in the cut $(C, Q = V \setminus C)$. Now u is added to C while for every vertex in $v \in Q$, we will update the smallest edge to it by checking it with the weight of uv.
Correctness again follows from the correctness of the meta algorithm.

Running Time of Prim's:

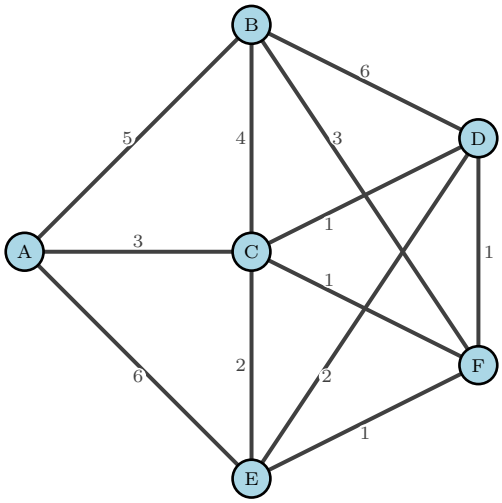
We have a data structure Q that stores the weight of the smallest edge to every vertex. So it has n elements. We need to EXTRACT-MIN once every loop. And we modify the *key* values at most once for every edge.

If we implement Q with an array, then EXTRACT-MIN is $O(n)$ and since it is repeated n times, in total it is $O(n^2)$. The update of values is $O(m)$ because updating a value in the array is $O(1)$. So total running time is $O(n^2)$.

If instead we use a heap for Q, EXTRACT-MIN and UPDATE are both $O(\log n)$ (update is $O(\log n)$ assuming we use pointers to access the value of a vertex in the heap), and since we do EXTRACT-MIN n times and UPDATE at most m times, so the total running time is $O((n+m) \log n)$. This is $O(m \log n)$ because the graph is connected and $m \geq n - 1$. For dense graphs this is worse than using an array.

Summary: Running time of Prim's is $O(n^2)$. If we know the graph is sparse, then we can use a heap to reduce this to $O(m \log n)$ (note: this is worse running time for dense graphs, in which case we prefer to use array with $O(n^2)$ running time).

Consider the following graph where the Prim's algorithm is run from vertex A:



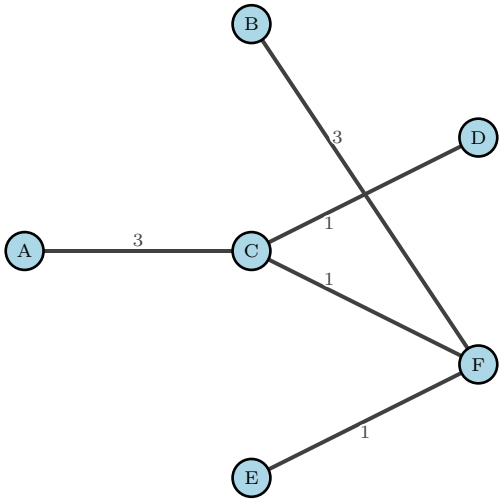
The following table shows the status of Q at every loop of Prim's algorithm. * indicates that a vertex is no longer in Q, and so has been added to C, the component we are growing. The algorithm terminates when there are no more vertices left in Q.

A	B	C	D	E	F
0	∞	∞	∞	∞	∞
*	5	3	∞	6	∞
*	4	*	1	2	1
*	4	*	*	2	1
*	3	*	*	1	*
*	3	*	*	*	*
*	*	*	*	*	*

We will also keep a table to record the parents:

A	B	C	D	E	F
*	F	A	C	F	C

We can use this table to reconstruct the minimum spanning tree:



Kruskal's with UNION-FIND

In Kruskal's algorithm, we are required to maintain components of the graph and at every stage, when we inspect an edge, we need to check the components of the end points, and if they are distinct components, then we need to merge them. These specific operations are required in multiple graph algorithms, and so deserve a special mention.
In all such applications, we require a data structure to store disjoint subsets of a base set, and we need two operations: UNION, where two subsets are merged to one, and FIND, where given an element, we have to find the subset it belongs to. In our previous implementation, the data structure we used was an array. An alternate way is to make each component a tree, where the root of the tree is a representative or identity of the tree, and each vertex has a pointer to its parent in the tree.
FIND: to find the component of any vertex, we must recurse through the parent pointers until we reach the root.
UNION: to merge two subsets, we make the parent of the root of one tree as the root of the other tree.
We can see that if the trees have a very large depth, then this data structure is not very efficient. So there are two easy heuristics which make this more efficient:
UNION BY RANK: we store the depth of the tree also in the root and call it the rank of the tree. The tree with greater rank will become the parent of the other tree.
PATH COMPRESSION: when doing FIND(x), we essentially travel from a vertex to the root of the tree it is in. We can in this stage update all the vertices in this path to directly point to the tree, thereby making all these subtrees shorter.

The worst-case running time using both union by rank and path compression is $O(k\alpha(n))$, where n is the number of elements in the base set, and k is the number of operations performed.

In particular, because we perform at most $2m + n$ operations in Kruskal's, the complexity of Kruskal's becomes $O(m \log n + (2m + n)\alpha(n))$ where the first term was the time required to sort the edges. But since $\log n$ grows much faster than $\alpha(n)$, this is essentially $O(m \log n)$.

Lecture 10

P, NP, coNP, Karp Reduction

Decision Problems

In studying the complexity of problems, we look at the subclass of problems which don't compute something, but instead **decide** a certain property of an input. In this case, given any input, our output is essentially a yes-no answer. In this case, we can think of the collection of all inputs with **yes** answer as a language. So every decision problem is essentially a subset of all possible finite binary strings. The benefit of this is that then we have powerful tools to compare the complexity of different languages, which would be otherwise difficult if we had to worry about the output of the programs.

Language of a Decision Problem: The collection of all inputs which result in a yes answer for a decision problem is said to be the language of the problem.

All problems we have done so far have equivalent decision problems and we would study them while studying complexity of algorithms. Some examples of corresponsing decision problems from the algorithms we did so far, and some others:

SEARCH $= \{(x_1, x_2, ..., x_n, x) | x = x_i \text{ for some } i\}$.

k-INCREASING-SUBSEQUENCE = $\{(x_1, x_2, ..., x_n, k)|$ there is an increasing subsequence of length $k\}$.

st-PATH = $\{(G, s, t)|G$ has a path from s and t $\}$.

st-k-PATH = $\{(G, s, t, k)|G$ has a path of length k between s and t $\}$.

SUDOKU = { a partially filled $n \times n$ sudoku | there is a correct way of filling it }.

k-SPANNING-TREE = $\{(G, k)|G$ has a spanning tree of total weight at most $k\}$.

5CYCLE = $\{G|G$ has a cycle of length 5$\}$.

P, NP

Class P: is the collection of all decision problems which have an algorithm with polynomial running time, i.e., if the input has size n, then there is a k such that the running time is $O(n^k)$.

Class NP: NP stands for **Non-deterministic** Polynomial time. It is the class of all decision problems, which given a (short) proof (WITNESS/GUESS), can verify the proof in polynomial time (VERIFICATION). The non-determinism is the part which generates this short proof. Another way to think of this class is polynomial time verifiable languages.

Difference between Determinism and Non-determinism: To understand the difference between the two classes, its is essential to understand how the algorithms for the two classes will work. A deterministic algorithm search for the answer, while the non-deterministic machine just guesses the answer and verifies it quickly.

Some examples:
SEARCH: a deterministic machine will look through the whole array, a non deterministic machine will guess the position (WITNESS/GUESS) of the element we are trying to find and check at the position if it really is this element (VERIFICATION).

SUDOKU: a deterministic machine will look at all possible ways to fill the sudoku, a non deterministic machine will guess a proper way to fill it (WITNESS/GUESS) and check if it filled it correctly (VERIFICATION).

5CYCLE: a deterministic machine will look at all possible sets of 5 vertices and see if they form a cycle, a non deterministic machine will guess a certain set of 5 vertices (WITNESS/GUESS) and check if it filled it correctly (VERIFICATION).

Witness Theorem: A Language $L \in NP$, if there is a constant c and another language $L' \in P$ of pairs (x, w) such that,
(a) if $(x, w) \in L'$, then, $|w| \leq |x|^c$
(b) $x \in L$ if and only if $\exists w$ such that, $(x, w) \in L'$.

We think of w as the WITNESS or guess or proof for $x \in L$. By saying $L' \in P$, we are saying that the verification is polynomial time. Condition (a) makes sure that $L' \in P$ does not use exponential time in terms of x by cheating with a very large witness w.

Some examples: (Please note that we don't think of numbers as 1 unit anymore, but we are going to be precise about the bits by saying that a number n uses $\log n$ bits. In this way, the adjacency matrix of a graph has size $O(n^2)$ if its unweighted, while an adjacency list has size $O(m \log n)$ because it takes $2 \log n$ bits to write each edge down and there are m edges, and we can upper bound it with $O(n^2 \log n)$.

SUDOKU:
Witness: a way to fill it. Size: $O(n^2 \log n)$
Verification: check every row column and square if it is correctly filled. Time to check: We have to check n rows, n columns, and n squares of side length $\sqrt{n}$. We just check if each contains all numbers, which can be done in polynomial time. So total time is $O(3n\cdot$ time to check one row) which is also polynomial

5CYCLE $= \{G|G$ has a 5 cycle $\}$.
Witness: A sequence of 5 vertices v_1, v_2, v_3, v_4, v_5. Size: $O(5 \log n)$
Verification: check if the necessary edges are there. This we do by looking through the adjacency list five times, so it is polynomial time.

COMPOSITES $= \{x|x$ is not a prime$\}$.
Witness: A divisor. Size: $O(\log x)$ where the input is also $O(\log x)$ long, so it is linear in the size of input.
Verification: Check if the divisor really divides x and is not 1 or x. Long division can be done in polynomial time in terms of $\log x$.

CONNECTED $= \{G|G$ is a connected graph$\}$.
Witness: Either a list of paths between every pair of vertices OR a spanning tree. If the witness is a spanning tree, that is a list of $n-1$ edges, so size is $O(n \log n)$, if it is paths then it is $O(n^3 \log n)$, but in both cases it has polynomial size.
Verification: In the case of paths, we must check if there is a path listed for every pair of vertices and that it is a valid path, this is going through the adjacency list $O(n^3 \log n)$ times, and checking n^2 pairs if they are listed. This is polynomial time.

HAMCYCLE $= \{G|G$ has a Hamiltonian cycle$\}$.
Witness: A permutation of the vertices. Size: $O(n \log n)$
Verification: Check if there are edges between consecutive vertices in the permutation, check if all vertices are there. Again we are looking through the adjacency list n times so it is polynomial time.

3COL $= \{G|G$ has a proper coloring with 3 colors$\}$.
Witness: A coloring of the graph. Size: $O(n)$ as this is just a list of n numbers each in the set $\{1, 2, 3\}$.
Verification: Check if it is proper (check end points of every edge), and check if three colors are used. This can be done in polynomial time.

coNP

Definition: Given any class X of languages, then the class coX is defined as $coX = \{L|\overline{L} \in X\}$.
Note that coX is not the complement of X. In particular, if there is a language L such that $L, \overline{L} \in X$, then it also follows that $L, \overline{L} \in coX$, and so $X \cap coX \neq \phi$.

coNP: this is the class of all languages whose complements are in the class NP.

Note this is not disjoint from NP and not the complement of NP. A useful way to think of this class is all languages such that their complements have a short witness.

Examples: PRIMES, complements of languages above.

Theorem: $P \subseteq NP \cap coNP$.

Proof: We need to show that if $L \in P$, then $L \in NP$ and also $L \in coNP$. For the first, we need to show a polynomial size witness that can be verified in polynomial time. Since $L \in P$, given any input x, we can check if $x \in L$ in polynomial time. Then notice that we can pick anything as witness! Let abc be your favorite movie. Then (x, abc) is also a witness (here abc is a fixed string that we use for all inputs x). To see that it is a good witness, its size is a constant, and to verify, we just

use the polynomial algorithm to check $x \in L$ and ignore the witness and $(x, abc) \in L'$ if and only if $x \in L$.

For $L \in coNP$, we just note that if $L \in P$, then $\overline{L} \in P$, because we just flip the yes-no answers for every input. Then $\overline{L} \in NP$ and so $L \in coNP$.

Karp Reduction

A very powerful technique to show how the complexity of two problems relates to each other is Karp reduction.

Karp Reduction or Polynomial Time reduction: We say that a language A has a Karp-reduction or a polynomial time reduction to another language B, denoted by $A \leq_P B$, if there is a polynomial time computable function f, such that for every string x, $x \in A$ if and only if $f(x) \in B$.

In other words, if there is an input x to the decision problem of A with a yes answer, then $f(x)$ is an input to B with a yes answer, and if x is an input to A with a no answer, then $f(x)$ is an input to B with a no answer.

Note: f may not be an invertible function, so $A \leq_P B$ does not imply $B \leq_P A$!

Why is Karp reduction useful? Well, because if there was a fast way to solve problem B, then we can map every input for A into an input for B and read off the answer, thereby giving a fast algorithm for A. So, in some sense, solving A is at most as complex as solving B. This is also why the less-than sign makes sense, because the complexity of A is in some sense upper bounded by B. We will formalize these in the next few theorems. But first an example:
Example: reduction of $3COL \leq_P 4COL$.

Lecture 11

NP-completeness

For all the languages we defined so far, an important thing to note is that when we talk of their complement, then we do not talk about incorrect or invalid strings. For example,
HAMCYCLE = $\{G | G$ has a Hamiltonian cycle$\}$.

If we look G in the language HAMCYCLE, it is an encoding of a graph as a binary string. But, there are many binary strings which are not encodings of any graphs. So, if we try to be precise in our definition, then the complement of HAMCYCLE should be,
= $\{s | s$ is not an encoding of a graph OR if it is the encoding of some graph G, then G is a graph without a Hamiltonian cycle$\}$.

But in practice, checking if a given string is a proper encoding of a graph is polynomial time task. We have to check if the adjacency matrix has n^2 entries, all 0 or 1 and if undirected, we have to check if the matrix is symmetric, and so on. In case of problems like SUBSET-SUM, the check in the complement involves seeing if the input has proper syntax. Again, this is a polynomial check. So any algorithm that solves these complement problems can easily start with this check and the complexity (P, NP etc) really depends on the special property of a input with correct syntax, like the graph not having a hamiltonian cycle, or there being no subset with the required sum etc. So, in all our discussions, when we talk about complements, we only talk about these properties, with just a one line mention that words with incorrect syntax do not effect our analysis of its complexity.

Properties of Karp Reduction

Karp reductions are very useful because if $A \leq_P B$, then the complexity of B somehow caps the complexity of A. This idea is clarified further with the following theorems:

Theorem: If $A \leq_P B$, then $\overline{A} \leq_P \overline{B}$.

Proof: If $A \leq_P B$, then there is a polynomial time computable function f such that $x \in A$ if and only if $f(x) \in B$. But then, $x \in \overline{A}$ if and only if $f(x) \in \overline{B}$. So f function gives us the required Karp reduction between the complements also.

Theorem: If $B \in P$ and $A \leq_P B$, then $A \in P$.

Proof: Since $A \leq_P B$, there is a polynomial time computable function f such that $x \in A$ if and only if $f(x) \in B$. Also since its a polynomial time computable function, the size of $f(x)$ cannot be bigger than the time required to compute it. So $|f(x)| \leq |x|^c$, for some fixed constant c. We also know that $B \in P$, so there is a polynomial time algorithm which determines, for all inputs of size at most n in $O(n^k)$ time, whether the input is in language B or not (where k is a fixed constant). We will now design a polynomial time algorithm to decide language A. Given any input x, we will compute $f(x)$ and run the algorithm of B. If $|x| = n$, then $|f(x)| \leq n^c$, and so time taken by the algorithm for B is $O((n^c)^k) = O(n^{ck})$.

Theorem: If $B \in NP$ and $A \leq_P B$, then $A \in NP$.

Proof: Since $A \leq_P B$, there is a polynomial time computable function f such that $x \in A$ if and only if $f(x) \in B$. Also since its a polynomial time computable function, the size of $f(x)$ cannot be bigger than the time required to compute it. So $|f(x)| \leq |x|^c$, for some fixed constant c. We also know that $B \in NP$, so there is a polynomial time non-deterministic algorithm which determines, for all inputs of size at most n in $O(n^k)$ time, whether the input is in language B or not (where k is a fixed constant). We will now design a polynomial time non-deterministic algorithm to decide language A. Given any input x, we will compute $f(x)$ and run the non-deterministic algorithm of B. If $|x| = n$, then $|f(x)| \leq n^c$, and so time taken by the algorithm for B is $O((n^c)^k) = O(n^{ck})$.

Alternate proof: Since $A \leq_P B$, there is a polynomial time computable function f such that $x \in A$ if and only if $f(x) \in B$. Also since its a polynomial time computable function, the size of $f(x)$ cannot be bigger than the time required to compute it. So $|f(x)| \leq |x|^c$, for some fixed constant c. We also know that $B \in NP$, so there is a polynomial time verifiable witness language L_B of pairs (x, w), where $|w| \leq |x|^k$ for some fixed k, and every input of size n in L_B can be decided in $O(n^l)$ time, where l is a fixed constant. Then we will give a witness language L_A for A. For every $x \in A$, the witness will be $(x, (f(x), w))$, where w is the witness of $f(x)$ in L_B. This language can be decided by the same algorithm that decides L_B by ignoring the x. If $|x| = n$, then $|f(x)| \leq n^c$ and $|w| \leq (n^c)^k = n^{ck}$, so the time taken is $O((n^{ck})^l) = O(n^{ckl})$, which is still polynomial in n.

Theorem: If $A \leq_P B$ and $B \leq_P C$, then $A \leq_P C$.

Proof: Exercise!

NP-completeness

NP-hard: A language L is called NP-hard, if every language in the class NP has a Karp reduction to it. That is, $\forall L' \in$NP, $L' \leq_P L$.

The terminology comes from the fact these problems are thought to be at least as hard to solve as any problem in the NP class, as solving them fast gives us a fast algorithm for any problem in NP.

NP-complete: A language L is said to be NP-complete if the following two conditions hold:
(a) $L \in$ NP
(b) it is NP-hard

One should think of NP-complete problems as the representatives of the hardest problems there are in the class NP, so that solving any one of them fast gives a fast way for solving all problems in NP. In fact, in our goal to prove or disprove P=NP, these are the problems to look at. This is further clarified with the following theorem.

Theorem: If L is NP-complete and $L \in$ P, then P=NP.

Proof: This follows from the definition of NP-complete and the property of Karp reduction $A \leq_P B$, where if $B \in$ P, then $A \in$ P.

On the outset it seems an outrageously bold statement to prove that any particular language is NP-complete, because for that one has to show a reduction from **every** language in NP to this language, and it is not clear if there is an easy way to list all languages in NP, let alone give a Karp reduction for them. This should motivate one to realize the significance of the following theorem:

Theorem (Cook-Levin): SAT is NP-complete.

SAT = $\{\Phi | \Phi$ is a satisfiable Boolean formula $\}$.

The proof of this theorem involves showing that the computation of any non-deterministic turing machine working in polynomial time can be written down as a polynomial length Boolean formula that has a satifying assignment if and only if the input was in the language of the turing machine. We will skip this proof.

With SAT we have set the stage for a multitude of problems which we will quickly add to our class of NP-complete languages using just a single Karp reduction. We can do so because of the following theorem:

Theorem: Given L_1, an NP-complete language and another language L_2 such that
(a) $L_2 \in$ NP, and
(b) $L_1 \leq_P L_2$,
then, the language L_2 is also NP-complete.

Proof: We use transitivity of Karp reductions. Since any language $L \in$ NP has a Karp reduction to L_1, by transitivity $L \leq_P L_2$.

We have the following Karp reductions, of which we will prove a subset in the class.

SAT $\leq_P$ 3SAT $\leq_P$ 3COL

Using 3COL:
3COL $\leq_P$ kCOL
3COL $\leq_P$ MAX-INDEPENDENT-SET $\leq_P$ MAX-CLIQUE

SAT $\leq_P$ 3SAT $\leq_P$ st-DIRHAMPATH

using st-DIRHAMPATH,
st-DIRHAMPATH $\leq_P$ st-HAMPATH $\leq_P$ HAMPATH $\leq_P$ HAMCYCLE $\leq_P$ HAMPATH $\leq_P$ k-PATH

SAT $\leq_P$ 3SAT $\leq_P$ SUBSET-SUM

using SUBSET-SUM,
SUBSET-SUM $\leq_P$ KNAPSACK
SUBSET-SUM $\leq_P$ PARTITION

Proofs of some of these are in the next section.

Some reductions

Theorem: 3COL $\leq_P$ MAX-INDEPENDENT-SET

Recall that, MAX-INDEPENDENT-SET = $\{(G, k)|G$ has a independent vertex set of size $k\}$.

Proof: Given a graph G, we consider the map $f(G) = (G', n)$, where n is the number of vertices in G, and G' contains three copies of G, say G_1, G_2, G_3. Let vertices of G be $v_1, v_2, \ldots v_n$, then we will say the vertices of G_i will be $v_1^i, v_2^i, \ldots, v_n^i$. G' contains all these three copies with their original edges, but we also include n triangles of the form v_j^1, v_j^2, v_j^3 for all $1 \leq j \leq n$. We claim that G has a 3-coloring if and only if G' has an independent vertex set of size n.

(Proof of IF): If G has a 3-coloring, let the colors required to color G be red, blue and green. Consider the copies of the vertices in the red color class from the copy G_1, the copies of vertices from blue color class from the copy G_2 and the green ones from G_3. We claim that this is an independent vertex set of size n. The copies of red vertices in G_1 are independent since they have no edges in G, similarly for G_2 and G_3, and finally, since we added edges to form triangles only between copies of the same vertex, a red vertex from G_1 will have no edge to a blue vertex from G_2 and so on.

(Proof of ONLY IF): If G' has a independent vertex set of size n, let V_i be the subset of these vertices from G_i. We give a 3 coloring of G by giving a vertex v color i, if the copy of it in the independent vertex set was v^i. Since the size of the independent vertex set is n, and since only one copy of a vertex v can be in this set, we know that **exactly** one copy of every vertex is in this set, so every vertex in G will get a color. Further if uv is an edge in G, then u and v must be in different G_i in the independent vertex set, and so they will get different colors.

Theorem: st-DIR-HAMPATH $\leq_P$ st-HAMPATH
where st-DIRHAMPATH = $\{(G, s, t)|G$ is a directed graph with a directed Hamiltonian path between two designated vertices s and $t\}$, and,
st-HAMPATH = $\{(G, s, t)|G$ is an undirected graph with a Hamiltonian path between two designated vertices s and $t\}$.

Proof: Given a directed graph G, and two vertices s, t, we consider the map $f((G, s, t)) = (G', s, t)$, where G' is the graph obtained by converting every vertex $v \neq s, t$ into a triple v_{in}, v_{mid}, and v_{out}. We will delete the incoming edges of s and out going edges of t and convert the remaining edges into undirected edges. For a directed edge uv in the original graph, we will put an edge in G' from u_{out} to v_{in}. For every v, there is an edge between v_{in} and v_{mid}, and also another edge between v_{mid} and v_{out}. We claim that G has a directed st-Hamiltonian path if and only if G' has a (undirected) st-Hamiltonian path.

(Proof of IF): Any path from $s, u_1, u_2, \ldots, u_k, t$ will look like $s, u_{1,in}, u_{1,mid}, u_{1,out}, u_{2,in}, \ldots, t$ in G'. In particular, a Hamiltonian path in G will give such a Hamiltonian path in G'.

(Proof of ONLY IF): Given a st-Hamiltonian path in G', we claim that for every vertex v, the in, mid and out vertices appear in this order. From s we can only go to an in-vertex. From there we must go to the mid vertex, because otherwise this mid vertex will have to be the last vertex of the Hamiltonian path, but that is t. And from mid there is only one option to go to out. From a out vertex again we can only go to an in-vertex and the property remains. If the order in-mid-out is maintained, then we can see that this easily translates to a directed Hamiltonian path in the original graph.

Theorem: SUBSET-SUM $\leq_P$ PARTITION

Proof: We consider the map $f((s_1, s_2, \ldots, s_n, b)) = (s_1, s_2, \ldots, s_n, s - 2b)$, where $s = \sum_{i=1}^{n} s_i$. We claim that there is a subset summing to b if and only if there is a partition with equal sums.

If there is a subset summing to b, taking that with $s - 2b$ gives us the required partition, since the total sum of numbers in the map is $2s - 2b$. Conversely, if there is a partition, then consider the part containing $s - 2b$. The other numbers in this partition must sum to b, again since the total sum is $2s - 2b$.

Syllabus for Midterm 2 is until here.

Lecture 12

Branch and bound, Approximation algorithms

We will look at several strategies applied to tackle NP-complete problems.

Branch and bound

The brute force algorithm for NP-complete languages requires searching a tree of possible solutions which is exponentially big. Branch and Bound methods try to find efficient ways of exploring these search trees (BRANCH) with easy tools to prune off any subtrees that cannot contain the optimal solution (BOUND).

Example 1: For 3COL, the brute force coloring gives us an algorithm of complexity $O(3^n n^2)$. But we can improve it by first picking a subset (BRANCH), checking if its an independent vertex set (BOUND) and checking if the graph induced on the rest of the vertices is bipartite (BFS). We see that this is $O(2^n n^2)$.

Example 2: Consider the MAX-INDEPENDENT-SET problem which is NP-complete. To check every subset of G is independent takes $O(n^2 2^n)$ time.

(BRANCH) Suppose we pick a vertex and consider maximum independent subsets containing this vertex and those that don't. Those which contain this vertex cannot contain any vertex in its neighborhood. Let the neighborhood of v be $N(v)$, then $MIS(G) = \max(MIS(G \setminus v), MIS(G \setminus (\{v\} \cup N(v))) \cup \{v\})$.

(BOUND) Firstly, we note that finding the graph without all the neighbors of v takes $O(n^2)$ time ($O(m)$ which is bounded by $O(n^2)$). So the recurrence will be $T(n) \leq T(n-1) + T(n-1-d(v)) + O(n^2)$, where $d(v)$ is the degree of v.

But if $d(v) = 0$, then we get $T(n) \leq 2T(n-1) + O(n^2)$, which is as bad as our brute force algorithm.

We can immediately improve upon this with two modifications:

- We always delete the vertex with maximum degree first.
- (BOUND: Improvement) If the maximum degree was 0, we can just return n.

This gives the recurrence $T(n) \leq T(n-1) + T(n-2) + O(n^2)$. Solving this gives us $T(n) = O(\Phi^n)$, where $\Phi = \frac{1+\sqrt{5}}{2} = 1.618$, the golden ratio.

But we can further BOUND the branches of the search tree by realizing that if the maximum degree is 2, then we can find the maximum independent vertex set in linear time. So, we get the following recurrence:

This gives the recurrence $T(n) \leq T(n-1) + T(n-4) + O(n^2)$. Solving this gives us $T(n) = O(\alpha^n)$, where $\alpha = 1.381$ is the solution of the equation $\alpha^4 = \alpha^2 + 1$.

Branch and Bound methods are extensively used in SAT solvers and other NP-complete problems, which though still have exponential running time in worst case, but in practice help us solve SAT formulas with millions of clauses on tens of thousands of variables.

Dynamic Programming

We have seen this extensively, so we will briefly just mention that these give us a polynomial algorithm in the actual value of the variable instead of the size of the input.

Approximation algorithms

Instead of getting the optimal answer for the problem, if we instead just find a suboptimal answer but in polynomial time, then this algorithm is called an approximation algorithm, because we only give an approximation to the optimal value. There are two ways in which we can express the relationship between the optimal and the suboptimal value found by our algorithm: either we look at the difference of the two (additive error), or we say that the optimal is at most a constant times the suboptimal (multiplicative error).

Approximation algorithms - additive error

Let the optimal value be denotes by OPT(n) and the value given by our algorithm $A(n)$. Then when our algorithm has only an additive error, then we mean that, $A(n) - c \leq OPT(n) \leq A(n)$ if our problem required minimizing some quantity, and $A(n) \leq OPT(n) \leq A(n) + c$ if we were maximizing some quantity.

Example 1: Graph edge coloring: We know that the edge chromatic number is $\Delta(G)$ or $\Delta(G) + 1$. It is NP-complete to decide the edge chromatic number (even for max degree 3 graphs, i.e. cubic graphs) of graphs, but the simple algorithm of determining $\Delta(G)$ and reporting $\Delta(G) + 1$ as the edge chromatic number is an approximation algorithm with additive error just 1!

Example 2: Consider the optimization problem of, given a graph G, finding the spanning tree T with the smallest possible $\Delta(T)$. Its corresponding decision problem, MAXDEG-SPANNING-TREE, is NP-complete. But there are polynomial time algorithms which can solve this optimization version with an additive error of just 1 again. So if a graph had a Hamiltonian path, this algorithm would either return that, or a spanning tree with max degree at most 3. We skip the algorithm here.

There are very few problems which have an efficient approximation algorithm with an additive error. The large chunk of existing algorithms fall into the next category.

Approximation algorithms - multiplicative error

Let OPT(n) be the optimal solution of a problem in which a quantity is being minimized. If there is a constant c such that, $A(n) \leq c\text{OPT}(n)$, then the algorithm A is said to be a c-spproximation for the problem.
Example 1: Bin packing: Given positive rational weights $s_1, s_2, \ldots, s_n$, $0 \leq s_i \leq 1$, our task is to find the minimum number of bins these can be packed into given that each bin can carry a maximum weight of 1. For example, $0.4, 0.8, 0.2, 0.1, 0.4, 0.1$.

First Fit Approximation: fit each weight into the first bin it will fit into. Let FF be the number of bins used by our algorithm, and let OPT be the optimal number of bins. Then $\lceil \sum s_i \rceil \leq OPT$. But we notice that we can't have two half empty bins in FF, because then the algorithm would not have chosen the later bin for the elements in it. So then $\frac{1}{2}(FF - 1) \leq \lceil \sum s_i \rceil \leq OPT$.

More careful counting can show $FF \leq 1.7OPT$.

FFD: First Fit Decreasing: First sort the weights into a decreasing sequence. Then follow the FF algorithm. Can be shown that here $FFD \leq \frac{11}{9}OPT + \frac{6}{9}$

Additive approximation: Rothvoss gave a randomized algorithm with at most $OPT + O(\log(OPT))$ bins. Note: For another paper of his, Rothvoss was awarded the Godel prize in 2023 last week!

Example 2: TSP: We give a 2-approximation algorithm for the TSP when the graph edge weights satisfy the traingle inequality $w(uv) + w(vx) \geq w(ux)$. Given a complete graph G on n vertices with edge weights that satisfy triangle inequality, we will find the spanning tree T and walk along it to get a walk that starts and ends in a vertex v. For any vertex that is visited a second time, then skip to the next vertex.

Theorem: There is no constant c such that there is a c-approximation algorithm for TSP.

Proof: Recitation!