COMP 251: Algorithms & Data Structures

Owen Lewis

Winter 2018

Contents

1 Overview of Graph Theory

1.1 Definitions

A graph $G = (V, E)$ is a set V of vertices (a.k.a. nodes) and a set E of edges (denoting vertex pairs). We set $n = |V|$, and $m = |E|$. A graph is said to be *undirected* when for any edge $(u, v) \in E$ there exists an edge $(v, u) \in E$ for some nodes u, and v. A graph is said to be *directed* if it is not undirected. In other words, the edge set of a directed graph consists of ordered pairs where the edge set of an undirected graph consists of unordered pairs.

A *walk* is a set of vertices $\{v_0, v_1, \ldots, v_\ell\}$ such that $(v_i, v_{i+1}) \in E$, $\forall\ 0 \leq i \leq \ell$. A walk where $v_0 = v_\ell$ is said to be a *circuit* or a *closed walk*. A circuit where every edge in the graph is used exactly once is known as an *Eulerian circuit*. A *cycle* is a walk $\{v_0, v_1, \ldots, v_\ell\}$ such that every vertex is distinct except $v_0 = v_\ell$. A cycle where every vertex of the graph is used exactly once is known as a *Hamiltonian cycle*. A walk where every vertex is distinct is said to be a *path*.

A graph is said to be *connected* if for each $u, v \in V$ there exists a walk from u to v. A graph is said to be *disconnected* id it is not connected. Each connected subgraph of a graph is called a *component*. A connected graph therefore has exactly one component.

A connected component with no cycles is called a *tree*. A graph whose components are all trees is said to be a *forrest*. A tree is said to be *spanning* if it contains every vertex in the graph. A vertex in a tree with at most one neighbour is called a *leaf*.

A *matching* is a set of vertex-disjoint edges i.e. each edge is incident to at most one other edge in a matching. A matching is said to be *perfect* if every vertex is incident to exactly one edge in the matching.

A *clique* is a set of pairwise adjacent vertices. In *independent set* (a.k.a. a *stable set*) is set of pairwise non-adjacent vertices.

A *bipartite graph* is a graph such that the vertex set V can be partitioned as $V = X \cup Y$ where each edge has one node in X and the other node in Y. Note that X and Y are necessarily independent sets.

1.2 Some Theorems for Undirected Graphs

Theorem: (Handshaking Lemma) Let $G = (V, E)$ be an undirected graph, let $\Gamma(v) := \{u : (u, v) \in E\}$ be the set of neighbours of a node v, and let the *degree* $\deg(v)$ of a vertex v equal the cardinality of $\Gamma(v)$. Then there are an even number of vertices with odd degree.
Proof: First note that since we're double-counting the number of pairs where

(v, e) is an edge incident to v

$$2 \cdot |E| = \sum_{v \in V} \deg(v)$$

Since the degree of a vertex is either even or odd, we can partition V into a set of odd-degree vertices $\mathcal{O}$, and a set of even-degree vertices $\mathcal{E}$. This gives us

$$\sum_{v \in V} \deg(v) = \sum_{v \in \mathcal{O}} \deg(v) + \sum_{v \in \mathcal{E}} \deg(v)$$

which implies

$$\sum_{v \in \mathcal{O}} \deg(v) = 2 \cdot |E| - \sum_{v \in \mathcal{E}} \deg(v)$$

since both the $2 \cdot |E|$ term is even (obvious) and the $\sum_{v \in \mathcal{E}} \deg(v)$ term is even (sum of even numbers) then the $\sum_{v \in \mathcal{O}} \deg(v)$ term must also be even. □

Theorem: (Euler's Theorem) If G is an undirected graph then G contains an Eulerian circuit if and only if every vertex has even degree.
Proof: Easy proof by induction

Lemma: A tree T with $n \geq 2$ vertices has at least one leaf vertex.
Proof: Trees are connected so there exists no vertices with degree 0 when $n \geq 2$. Suppose each vertex has degree of at least 2. Then consider the longest path $P \subseteq T$, $P = \{v_1, v_2, \ldots, v_{\ell-1}, v_\ell\}$. Since $\deg(v_\ell) \geq 2$, $\exists$ a neighbour (of v_ℓ) $x \in P$ with $x \neq v_{\ell-1}$. If $x = v_{\ell+1}$ then P is not the longest path, a contradiction. Therefore, for P to be the longest path, x must be somewhere else in P, but this creates a cycle, another contradiction. Thus there must exist at least one node v such that $0 < \deg(v) < 2$ – a leaf. □

Theorem: A tree with n vertices has exactly $n - 1$ edges.
Proof: Simple proof by induction.
Base case: A tree with one vertex trivially has 0 edges.
Induction Hypothesis: Assume any tree with $n - 1$ vertices has $n - 2$ edges.
Inductive Step: Take a tree with $n \geq 2$ vertices. By the previous lemma this tree contains a leaf vertex v. This implies that $T \setminus \{v\}$ is a tree with $n - 1$ vertices and by the induction hypothesis $T \setminus \{v\}$ is a tree with $n - 2$ edges, which imples that T is a tree with $n - 1$ edges. □

Theorem: (Hall's Theorem) Let $G = (X \cup Y, E)$ with $|X| = |Y|$ be a bipartite graph. G contains a perfect matching if and only if $\forall \ B \subseteq X$, $|\Gamma(B)| \geq |B|$ (Hall's condition).
Proof: Firstly, the ($\Rightarrow$) direction is fairly obvious. If $B \subseteq X$ with $\Gamma(B) < |B|$ then the graph can't have a perfect matching. The ($\Leftarrow$) direction is a bit trickier. Suppose Hall's condition is satisfied. Then, take the maximum cardinality

matching M is the graph. If M is perfect then we are done. Otherwise there must exist an unmached vertex b_0.

- Since Hall's condition holds, we have $|\Gamma(\{b_0\})| \geq |\{b_0\}| = 1$ so b_0 must have at least one neighbour s_0.

- Suppose s_0 is matched in M to b_1.

- Since Hall's condition holds, we have $|\Gamma(\{b_0, b_1\})| \geq |\{b_0, b_1\}| = 2$ so $\{b_0, b_1\}$ must have at least one neighbour $s_1 \neq s_0$.

- Suppose s_1 is matched in M to b_2.

- Since Hall's condition holds, we have $|\Gamma(\{b_0, b_1, b_2\})| \geq |\{b_0, b_1, b_2\}| = 3$ so $\{b_0, b_1, b_2\}$ must have at least one neighbour $s_2 \notin \{s_0, s_1\}$.

- ...

we repeat this argument as long as we can. Since the graph contains a finite number of vertices this process must terminate, but it can only terminate when we reach an unmatched node s_k. Using the edges we've formed in M we can create a path P from b_0 to s_k that alternates between using non-matching edges and using matching edges. Swapping the matching edges with the non-matching edges gives us one more matching edge (as we have an odd number of edges.) This is still a valid matching as the internal nodes of P are still incident to exactly one matching edge. Also, the end nodes, b_0 and s_k were previously unmatched but are now incident to exactly one edge in the new matching. Thus M isn't the maximum capacity matching – a contradiction. □

1.3 """"Data Structures"""" for Representing Graphs

1.3.1 Adjacency Matrices

For a graph, an *adjacency matrix* M is a matrix suxh that

1. There is a row for each vertex

2. There is a column for each vertex

3. The $ij - th$ entry is defined as $M_{ij} = \begin{cases} 1, (i, j) \in E \\ 0, (i, j) \notin E \end{cases}$

Note that in an undirected graph the matric is symmetric around the diagonal because $(i, j) \equiv (j, i)$. Of course this is not necessarily true of directed graphs.

1.3.2 Adjacency Lists

An *adjacency list* of an undirected graph is such that for each vertex v of V we store a list of its neighbours. For a directed graph we have two lists: one in which we store the in-neighbours of v and one in which we store the out-neighbours of v.

1.3.3 Adjacency Matrices vs. Adjacency Lists

The main difference between the two is the amount of storage required to implement them.

- An adjacency matrix requires we store $\Theta(n^2)$ numbers

- An adjacency list requires we store $\Theta(m)$ numbers

In any graph $m = O(n^2)$. This means that for a sparse graph adjacency lists are highly favourable in terms of space complexity.

Verifying whether an edge exists, however, is much faster in an adjacency matrix – when using the array representation of a matrix it takes $O(1)$ time, where verifying the existance of an edge takes $O(\log n)$ time for an ordered adjacency list (using binary search), and $O(n)$ time if the adjacency list is not ordered (using sequential search).

2 Divide & Conquer

A *divide and conquer* algorithm ideally breaks up a problem of size n into smaller sub-problems such that:

- There are exactly a sub-problems

- Each sub-problem has a size of at most $\frac{1}{b} \cdot n$

- Once solved, the solutions to the sub-problems must be combined in $O(n^d)$ time to produce a solution to the original problem

Therefore the time-complexity of a divide and conquer algorithm satisfies a recurrence relation given by

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$$

2.1 The Master Theorem

Lemma 1: $\displaystyle\sum_{k=0}^{\ell} \tau^k = \frac{1 - \tau^{\ell+1}}{1 - \tau}$, for any $\tau \neq 1$.

Proof:

$$(1 - \tau)\sum_{k=0}^{\ell} \tau^k = \sum_{k=0}^{\ell} \tau^k - \sum_{k=1}^{\ell+1} \tau^k$$
$$= \tau^0 - \tau^{\ell+1}$$
$$= 1 - \tau^{\ell+1}$$
$$\sum_{k=0}^{\ell} \tau^k = \frac{1 - \tau^{\ell+1}}{1 - \tau}$$

$\square$

Lemma 2: $x^{\log_b y} = y^{\log_b x}$ for any base $b \in \mathbb{R}$.
Proof: From the power rule of logarithms we have

$$\log_b x \cdot \log_b y = \log_b \left(y^{\log_b x}\right)$$

similarily, we have

$$\log_b x \cdot \log_b y = \log_b \left(x^{\log_b y}\right)$$

therefore

$$\log_b \left(x^{\log_b y}\right) = \log_b \left(y^{\log_b x}\right)$$

thus

$$x^{\log_b y} = y^{\log_b x}$$

$\square$

Theorem: (The Master Theorem) If a recurrence relation is of the form $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$, for constants $a > 0$, $b > 1$, and $d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & \text{, if } a < b^d \text{ [Case I]} \\ O(n^d \cdot \log n) & \text{, if } a = b^d \text{ [Case II]} \\ O(n^{\log_b a}) & \text{, if } a > b^d \text{ [Case III]} \end{cases}$$

Proof: By adding dummy numbers, we may assume that n is a power of b, i.e. $n = b^\ell$, for some $\ell \in \mathbb{N}_0$. Then

$$T(n) = n^d + a\left(\frac{n}{b}\right)^d + a^2\left(\frac{n}{b^2}\right)^d + \cdots + a^\ell\left(\frac{n}{b^\ell}\right)^d$$

simlifying, we get

$$T(n) = n^d\left(1 + \frac{a}{b^d} + \left(\frac{a}{b^d}\right)^2 + \cdots + \left(\frac{a}{b^d}\right)^\ell\right)$$

we now have three cases:

Case I: $\dfrac{a}{b^d} < 1$

Set $\tau := \dfrac{a}{b^d}$ then we have

$$T(n) = n^d \sum_{k=0}^{\ell} \tau^k$$

applying lemma 1 we get

$$T(n) = n^d\left(\frac{1 - \tau^{\ell+1}}{1 - \tau}\right) \leq n^d\left(\frac{1}{1 - \tau}\right)$$

but $\dfrac{1}{1 - \tau}$ is a constant so $T(n) \leq n^d$ and thus

$$T(n) = O(n^d)$$

Case II: $\dfrac{a}{b^d} = 1$

Then we have that

$$T(n) = n^d\left(1 + 1 + 1^2 + \cdots + 1^\ell\right) = n^d\left(\ell + 1\right)$$

but $n = b^\ell \Rightarrow \ell = \log_b n$, thus

$$T(n) = O(n^d \cdot \log n)$$

8

Case III: $\dfrac{a}{b^d} > 1$

Set $\tau := \dfrac{a}{b^d}$ then we have

$$T(n) = n^d \sum_{k=0}^{\ell} \tau^k$$

applying lemma 1 we get

$$T(n) = n^d \left(\frac{\tau^{\ell+1} - 1}{\tau - 1} \right) \leq n^d \left(\frac{\tau^{\ell+1}}{\tau - 1} \right)$$

but since $\tau - 1$ is a constant we get

$$\begin{aligned} T(n) &= n^d O(\tau^{\ell+1}) \\ &= O(n^d \tau^{\ell+1}) \\ &= O(n^d \tau^{\ell}) \\ &= O\left(\left(\frac{a}{b^d} \right)^{\ell} n^d \right) \\ &= O\left(\left(\frac{n}{b^{\ell}} \right)^d a^{\ell} \right) \end{aligned}$$

but $n = b^{\ell}$ so

$$T(n) = O(a^{\ell})$$

and $\ell = \log_b a$ so

$$T(n) = O(a^{\log_b n})$$

and applying lemma 2 gives

$$T(n) = O(n^{\log_b a})$$

This completes the proof. $\qquad\qquad\square$

2.2 MergeSort

MergeSort is an algorithm to sort n numbers into non-decreasing order.

MergeSort$(x_1, x_2, \ldots, x_n)$
    if $n = 1$
        return x_1
    else
        return Merge(MergeSort$(x_1, \ldots, x_{\lfloor \frac{n}{2} \rfloor})$, MergeSort$(x_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, x_n)$)

Where the Merge function is a linear time algorithm combining two sorted lists (by comparing the first element in each list and moving the smaller element of

the two into our new list.)

MergeSort is correct!

- It calls itself on smaller instances until the division process terminates when it reaches a base case where each list has size 1

- MergeSort works trivially on the base cases

- This sort of strong induction-type proof works for just about all divide and conquer algorithms

MergeSort is efficient!

- The recurrence relation that we can construct to model the running time of MergeSort is given by

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

  as we break the problem into two sub-problems of size $\frac{n}{2}$, plus a linear time sub-routine to combine the solutions (Merge.)

- This recurrence relation is in the proper form to use the Master Theorem and indeed we're in its Case II.

- By the master theorem the running time of MergeSort is $O(n \cdot \log n)$.

- The running time can also be proved by unwinding the recurrence.

2.3   Binary Search

BInary Search is an algorithm to search for whether a key k exists in a given sorted list.

BinarySearch($a_1, a_2, a_n; k$)
    while $n > 0$
        if $a_{\lfloor \frac{n}{2} \rfloor} = k$
            return true
        if $a_{\lfloor \frac{n}{2} \rfloor} > k$
            return BinarySearch($a_1, \ldots, a_{\lceil \frac{n}{2} \rceil - 1}; k$)
        if $a_{\lfloor \frac{n}{2} \rfloor} < k$
            return BinarySearch($a_{\lceil \frac{n}{2} \rceil + 1}, \ldots, a_n; k$)
    return false

Binary Search works!

- It works for the same strong induction argument as MergeSort

Binary Search is efficient!

- The recurrence relation that we can construct to model the running time of Binary Search is given by

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

  as we break the problem into one sub-problem of size $\frac{n}{2}$.

- This recurrence relation is case II of the master theorem.

- By the master theorem the running time of MergeSort is $O(\log n)$.

- Again, the running time can also be proved by unwinding the recurrence.

2.4 Fast Multiplication

In COMP 250 we found an $O(n^2)$ algorithm to multiply 2 n-digit numbers (in any base). Let's try to do better using divide and conquer. Consider the example:

Let $\mathbf{x} := x_n x_{n-1} \ldots x_{\frac{n}{2}+1} x_{\frac{n}{2}} \ldots x_2 x_1$
Let $\mathbf{y} := y_n y_{n-1} \ldots y_{\frac{n}{2}+1} y_{\frac{n}{2}} \ldots y_2 y_1$

Then $\mathbf{x} = 10^{\frac{n}{2}} \mathbf{x}_L + \mathbf{x}_R$, where $\mathbf{x}_L = x_n \ldots x_{\frac{n}{2}+1}$, and $\mathbf{x}_R = x_{\frac{n}{2}} \ldots x_1$
Similarily $\mathbf{y} = 10^{\frac{n}{2}} \mathbf{y}_L + \mathbf{y}_R$, where $\mathbf{y}_L = y_n \ldots y_{\frac{n}{2}+1}$, and $\mathbf{y}_R = y_{\frac{n}{2}} \ldots y_1$

Thus

$$\mathbf{x} \cdot \mathbf{y} = (10^{\frac{n}{2}} \mathbf{x}_L + \mathbf{x}_R)(10^{\frac{n}{2}} \mathbf{y}_L + \mathbf{y}_R)$$
$$= 10^n \cdot \mathbf{x}_L \mathbf{y}_L + 10^{\frac{n}{2}} (\mathbf{x}_L \mathbf{y}_R + \mathbf{x}_R \mathbf{y}_L) + \mathbf{x}_R \mathbf{y}_R$$

So we've found a divide and conquer algorithm to multiply two n-digit numbers. Here we're breaking the problem down into four sub-problems of size $\frac{n}{2}$, with a few additions thrown in there (can be done in linear time). Therefore, the recurrence for this algorithm is given by

$$T(n) = 4 \cdot T\left(\frac{n}{2}\right) + O(n)$$

then, by the master theorem, the running time of this algorithm is $O(n^2)$. Hmmm. This isn't better. Well thanks to our good friend Carl "G-Money" Gauss, all is not lost. Gauss was studying the product of complex numbers

$$(a + bi)(c + di) = ac - bd + (bc + ad)i$$

which seemingly involved 4 multiplications, when he noticed that actually we can do it with only 3. Note that

$$(bc + ad) = (a + b)(c + d)$$

11

Now we can use this exact same trick but we'll replace the i with $10^{\frac{n}{2}}$. This gives

$$(\mathbf{x}_L \mathbf{y}_R + \mathbf{x}_R \mathbf{y}_L) = (\mathbf{x}_R + \mathbf{x}_L)(\mathbf{y}_R + \mathbf{y}_L) - \mathbf{x}_R \mathbf{y}_R - \mathbf{x}_L \mathbf{y}_L$$

So now we can break our problem into only 3 sub-problems! Our new recurrence is

$$T(n) = 3 \cdot T\left(\frac{n}{2}\right) + O(n)$$

so finally, by the master theorem, $T(n) = O(n^{\log_2 3}) \approx O(n^{1.59})$

2.5 Fast Matrix Multiplication

Let's try to efficiently multiply two $n \times n$ matrices. Let

$$X := \begin{pmatrix} A & B \\ C & D \end{pmatrix}, \text{ and } Y := \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

where X, and Y are two $n \times n$ matrices and $A, B, \ldots H$ are eight $\frac{n}{2} \times \frac{n}{2}$ matrices. Then their product Z is given by

$$Z = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

Therefore multiplying two $n \times n$ matrices involves eight products of $\frac{n}{2} \times \frac{n}{2}$ matrices, and n^2 additions. Therefore the recurrence relation is given by

$$T(n) = 8 \cdot T\left(\frac{n}{2}\right) + O(n^2)$$

By the master theorem, this then takes $O(n^3)$ time. That's ok but again, we can do better. To do better however, we need a little algebraic trick. To do this, we need to define 7 matrices, $S_1, \ldots, S_7$ given by

$$\begin{aligned} S_1 &= (B - D) \cdot (G + H) & S_2 &= (A + D) \cdot (E + H) \\ S_3 &= (A - C) \cdot (E + F) & S_4 &= (A + B) \cdot H \\ S_5 &= A \cdot (F - H) & S_6 &= D \cdot (G - E) \\ S_7 &= (C + D) \cdot E \end{aligned}$$

then the product of X, and Y is given by

$$Z = \begin{pmatrix} S_1 + S_2 - S_4 + S_6 & S_4 + S_5 \\ S_6 + S_7 & S_2 - S_3 + S_5 - S_7 \end{pmatrix}$$

This means that we only need to do 7 multiplications! Therefore our new recurrence is

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + O(n^2)$$

and thus by the master theorem $T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})$

12

2.6 Fast Exponentiation

Consider the following algorithm to quickly calculate x^n:

FastExp(x, n)
    if $n = 1$
        return x_1
    else if n is even
        return FastExp$(x, \lfloor \frac{n}{2} \rfloor)^2$
    else if n is odd
        return $x \cdot$FastExp$(x, \lfloor \frac{n}{2} \rfloor)^2$

Assuming that n is a power of two, we get that recurrence is given by

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

and thus by the master theorem this algorithm has time complexity of $O(\log n)$.

2.7 The Selection Problem

Suppose we're given a list of number and we want to find the kth smallest number in that list. We can use the following divide and conquer algorithm:

Select$(\mathcal{S}, k)$
    if $|\mathcal{S}| = 1$
        return x_1
    else
        set $\mathcal{S}_L := \{x_i \in \mathcal{S} : x_i < x_1\}$
        set $\mathcal{S}_R := \{x_i \in \mathcal{S} : x_i \geq x_1\}$

        if $|\mathcal{S}_L| = k - 1$
            return x_1
        else if $|\mathcal{S}_L| > k - 1$
            return Select$(\mathcal{S}_L, k)$
        else if $|\mathcal{S}_L| < k - 1$
            return Select$(\mathcal{S}_R, k - 1 - |\mathcal{S}_L|)$

Well.. this actually sucks because it runs in $\Omega(n^2)$ time. Note that it would actually be faster to just sort the list and then pull the kth element (this can of course be done in $O(n \cdot \log n)$ time.) One idea that we can exploit to improve this is to use a random pivot rather than deterministically selecting x_1 as a pivot.

We will define a *good pivot* to be such neither $\mathcal{S}_L$, nor $\mathcal{S}_R$ contain more than $\frac{3}{4}$ of $\mathcal{S}$, and a *bad pivot* to be otherwise. Note that by this definition the probability that a pivot will be either good or bad (respectively) is 0.5.

For randomized algorithms we're always interested in the *expected runtime* of the algorithm rather than the worst case runtime. The recurrence relation, $\bar{T}(n)$, for the expected runtime for the randomized version of the selection is then given by

$$\bar{T}(n) \leq \frac{1}{2} \cdot \bar{T}\left(\frac{3n}{4}\right) + \frac{1}{2} \cdot \bar{T}(n) + O(n)$$

where the $\frac{1}{2} \cdot \bar{T}\left(\frac{3n}{4}\right)$ term represents the recursive call given a good pivot, and the $\frac{1}{2} \cdot \bar{T}(n)$ term represents the recursive call on a bad pivot. It's less than or equal to because we will only ever use one of those terms at once but not both. We can clean this recurrence up a bit to find the expected runtime.

$$\bar{T}(n) \leq \frac{1}{2} \cdot \bar{T}\left(\frac{3n}{4}\right) + \frac{1}{2} \cdot \bar{T}(n) + O(n)$$
$$\frac{1}{2} \cdot \bar{T}(n) \leq \frac{1}{2} \cdot \bar{T}\left(\frac{3n}{4}\right) + O(n)$$
$$\bar{T}(n) \leq \bar{T}\left(\frac{3n}{4}\right) + O(n)$$

Applying the master theorem to that recurrence gives $\bar{T}(n) = O(n)$

This is good but let's try to do better, i.e. let's see if we can find a deterministic linear time algorithm. It turns out we can but it requires a little trick called the *median of medians*.

It's best to illustrate this concept with an example. Suppose we have a list, $\mathcal{S} = \{x_1, x_2, \ldots, x_n\}$, from which we want to find the median of medians. To do this we partition the list into $\left\lceil \frac{n}{5} \right\rceil$ groups of cardinality 5. Denote each such group by $G_1, \ldots, G_{\lceil \frac{n}{5} \rceil}$ Then we'll sort each group and let z_i be the median of each group G_i. Let m be the median of the set $\mathcal{Z} = \{z_1, \ldots, z_{\lceil \frac{n}{5} \rceil}\}$. Predictably, we call m the *median of medians*. This is significant because m will always be a good pivot. This is because there will always be at least $\frac{3n}{10} - 1$ numbers in $\mathcal{S}$ less than m, and $\frac{3n}{10} - 1$ numbers in $\mathcal{S}$ at least as large as m. Therefore $|\mathcal{S}_R| \leq \frac{7n}{10}$, and $|\mathcal{S}_L| \leq \frac{7n}{10}$.

Now that we have a deterministic way to guarntee we have a good pivot, all that's left is to bake it into a selection algorithm. Luckily, our algorithm won't vary too much since all we have to do is add a few steps at the beginning.

DetSelect$(\mathcal{S}, k)$

    if $|\mathcal{S}| = 1$

        return x_1

    else

        partition $\mathcal{S}$ into $\lceil \frac{n}{5} \rceil$ groups of 5

        for j in range $\left(1, \lceil \frac{n}{5} \rceil \right)$

            let z_j be the median of the group G_j.

        let $\mathcal{Z} = \{z_1, \ldots, z_{\lceil \frac{n}{5} \rceil}\}$

        set $m := \text{DetSelect}\left(\mathcal{Z}, \lceil \frac{n}{10} \rceil \right)$

        set $\mathcal{S}_L := \{x_i \in \mathcal{S} : x_i < m\}$

        set $\mathcal{S}_R := \{x_i \in \mathcal{S} : x_i \geq m\}$

        if $|\mathcal{S}_L| = k - 1$

            return m

        if $|\mathcal{S}_L| > k - 1$

            return DetSelect$(\mathcal{S}_L, k)$

        if $|\mathcal{S}_L| < k - 1$

            return DetSelect$(\mathcal{S}_R, k - 1 - |\mathcal{S}_L|)$

The recursive formula for the running time of this algorithm is

$$T(n) \leq T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + O(n)$$

where the $T\left(\dfrac{n}{5}\right)$ term is the time to find the median of medians, the $T\left(\dfrac{7n}{10}\right)$ term is pivoting based on the median of medians. Unfortunately, this doesn't work with the master theorem and we can't simplify it down to a form that does (unlike with the randomized algorithm, we need both of the $T(\ldots)$ terms.) Since it doesn't work with the master theorem, we need to use the recusion tree method. From the recursion tree method, $T(n) = O(n)$. So there it is.. a linear time deterministic algorithm to select the kth smallest element from a list!

2.8 The Closest Pair of Points in a Plane

Suppose we're given a set $\mathcal{P}$ of n points in a 2D plane, $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ and want to find the closest 2. For each $i \leq i \leq n$, let $p_i = (x_i, y_i)$.

The easiest (correct) solution to think of is an exhaustive search where we calculate every distance between any two points and return the smallest pair. There are $\binom{n}{2}$ possible pairs of points, and calculating the distance between any two can be done in constant time, so this means the runtime is $O(n^2)$. This is

okay, it's certainly polynomial-time efficient, but let's do it faster.

First, perhaps it will be handy to consider the 1D case (the closest two pairs of points on a line.) Note that in this case, the closest pair of point must be adjacent in their x-ordering. Therefore, assuming the list is already sorted according to their x-ordering, all we need to do is calculate $n-1$ pairwise distances, which can be done in $O(n)$ time. Of course if the list is not sorted we have to do that first, so the closest pair can be found in $O(n \cdot \log n)$ time. If we try to mimic that idea in the 2D case we come up with the following algorithm:

Pair2D($\mathcal{P}$)

    sort the pairs by their x-coordinate
    sort the pairs by their y-coordinate

    find the closest pair of points with respect to their x-coordinates
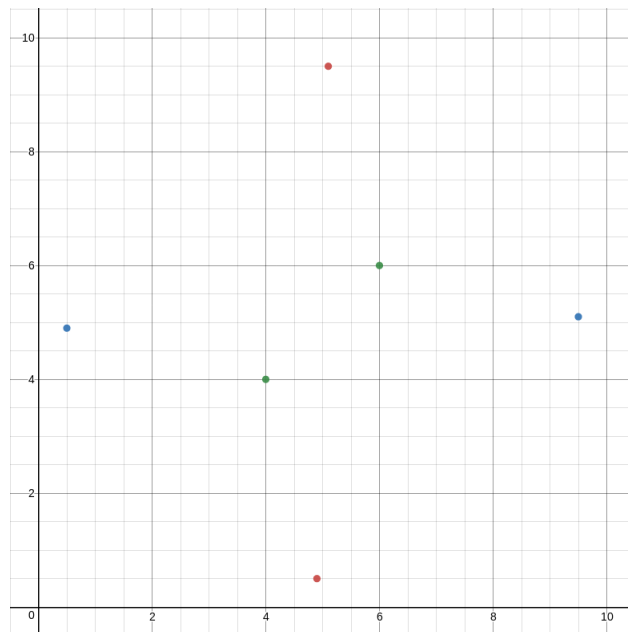   find the closest pair of points with respect to their y-coordinates

    return the pair from the above 2 with the smallest distance between them

---

Sorting the lists takes $O(n \cdot \log n)$ time, and finding the closest points takes $O(n)$ time, so this algorithm take $O(n \cdot \log n)$ time. Does it work though? Nope. Consider this example:

16

It's clear here that the two green points are the closest together, but the algorithm would have returned either the pair of red points or the pair of blue points.

Let's try to come up with a divide and conquer algorithm to solve this problem. We can start by partitioning $\mathcal{P}$ into two sets, $\mathcal{P}_L$, and $\mathcal{P}_R$, of cardinality $\frac{n}{2}$. This gives us three possible outcomes:

1. The closest pair of points is in $\mathcal{P}_L$

2. The closest pair of points is in $\mathcal{P}_R$

3. The closest pair of points has one point in $\mathcal{P}_L$, and the other point in $\mathcal{P}_R$

To summarize this idea so far, we have the algorithm

DCPair2D($\mathcal{P}$)
    find the point q with the median x-coordinate
    partition $\mathcal{P}$ into $\mathcal{P}_L$, and $\mathcal{P}_R$

    find the closest pair of points in $\mathcal{P}_L$
    find the closest pair of points in $\mathcal{P}_R$
    find the closest pair of points with one point in $\mathcal{P}_L$ and the other in $\mathcal{P}_R$

    return the pair from the above 3 with the smallest distance between them

---

The recursive formula for this algorithm is given by

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n^2)$$

The $O(n^2)$ term comes from the work needed to compare all $\frac{n}{2}$ points in $\mathcal{P}_L$ with all $\frac{n}{2}$ points in $\mathcal{P}_R$. $\left(\frac{n}{2} \times \frac{n}{2} = \frac{n^2}{4}$ comparisons.$\right)$ Therefore, by the master theorem, the running time is $O(n^2)$.

This isn't any better than exhaustive search because there's a bottleneck step where we compare the points in $\mathcal{P}_L$ with those in $\mathcal{P}_R$. Indeed we're pretty much just doing exhaustive search but on only $\frac{n}{2}$ elements. With this in mind, let's try to slim this step down. Note that since we've solved both the left and right sub-problems, there's a minimum pairwise distance δ, given by

$$\delta = \min(\delta_L, \delta_R)$$

for which the distance between the closest two points will not exceed, even if the two points are not in the same partition. Therefore, when looking for pairs of points with one point in each partition, we only need to look for points within δ of the dividing line! From now on we will denote the set of points within δ of

17

the dividing line by $\mathcal{P}_M$. This trick does work, but not trivially, so it has to be proved.

Lemma: Let $p_i \in \mathcal{P}_L$, and let $p_j \in \mathcal{P}_R$. If $d(p_i, p_j) \leq \delta$ then $\{p_i, p_j\} \subseteq \mathcal{P}_M$.

Proof: WLOG assume that $p_i \notin \mathcal{P}_M$. Then $d(p_i, p_j) > \delta$. Similarily, if $p_j \notin \mathcal{R}_M$. Then $d(p_i, p_j) > \delta$. Thus if the pair of points is not within one of the two original sub-problems, it must be contained within $\mathcal{P}_M$. $\square$

Let's implement this trick in the last algorithm we wrote up.

DCPair2D($\mathcal{P}$)

    find the point q with the median x-coordinate
    partition $\mathcal{P}$ into $\mathcal{P}_L$, and $\mathcal{P}_R$

    find the closest pair of points in $\mathcal{P}_L$
    find the closest pair of points in $\mathcal{P}_R$
    find the closest pair of points in $\mathcal{P}_M$

    return the pair from the above 3 with the smallest distance between them

---

There's still a problem. $\mathcal{P}_M$ can possibly contain all (or most) of the points! This means that our algorithm can *still* take $\Omega(n^2)$ time to measure the distances between the points in $\mathcal{P}_L \cap \mathcal{P}_M$, and the points in $\mathcal{P}_R \cap \mathcal{P}_M$. Once again, however, this was fortunately not all for nothing. Since $\mathcal{P}_M$ covers only a narrow band of the x-axis, we'll consider the area of the plane within δ of the dividing line (i.e. the area pf $\mathcal{P}_M$.) Suppose we divide this area into small squares of size $\frac{\delta}{2}$ then we have the following lemma:

Lemma: No two points from $\mathcal{P}$ will lie within the same square of width $\frac{\delta}{2}$.

Proof: WLOG take two points in a square in $\mathcal{P}_L$. Then their pairwise distance d is

$$d \leq \sqrt{\left(\frac{\delta}{2}\right)^2 + \left(\frac{\delta}{2}\right)^2} = \frac{\delta}{\sqrt{2}} < \delta$$

which is a contradiction as δ is the smalles distance between two points. $\square$

Theorem: Let $p_i \in \mathcal{P}_L$, and let $p_j \in \mathcal{P}_R$. If $d(p_i, p_j) \leq \delta$ then p_i, and p_j have at most 10 points between them in $\mathcal{P}_M$ with respect to their y-ordering.

Proof: WLOG suppose that p_i is below p_j in the y-order. Then p_j is in the same row of squares as p_i, or it is somewhere in the next two higher rows, or else they won't be within δ of eachother (a contradiction). Therefore we are working

in a rectangle that is at most three rows of squares high and four columns of squares wide. By the previous lemma we can have at most one point in any square and therefore in our 4×3 rectangle there can be at most 12 points. Thus, by choosing the the points on either extreme of the y-ordering of our rectangle, there can be at most 10 points between. □

This result implies that the idea underlying the algorithm from the 1D case can be applied after all! We just have to tweak it slightly so as to look for points that are 11 apart rather than just adjacent, i.e. we'll have to calculate at most $11n$ pairwise distances rather than at most $n-1$. Thus we either find a pair of points in $\mathcal{P}_M$ closer than δ or we don't, but if we don't then we can conclude that no such pair of points exists, and we can do so in $O(n)$ time!

Let's update our algorithm to reflect this.

DCPair2D($\mathcal{P}$)

  find the point q with the median x-coordinate
  partition $\mathcal{P}$ into $\mathcal{P}_L$, and $\mathcal{P}_R$

  find the closest pair of points in $\mathcal{P}_L$
  find the closest pair of points in $\mathcal{P}_R$
  find the closest pair of points in $\mathcal{P}_M$ using the (updated) 1D algorithm

  return the pair from the above 3 with the smallest distance between them

---

Now that we've gotten the bottleneck step down to $O(n)$, the new recurrence relation for this algorithm is

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

where the $O(n)$ term comes from finding the median with respect to the x-coordinates (using algorithm from previous lecture), partitioning $\mathcal{P}$, finding $\mathcal{P}_M$ and applying the 1D algorithm. Thus, by the master theorem, the time complexity of this algorithm is $O(n \cdot \log n)$. (If you think about it, the new most expensive step is just sorting the points in $\mathcal{P}_M$ with respect to the y-coordinates! If you don't think that's hella cool then you have no soul.)

Like most divide and conquer algorithms, this one is correct and the proof follows immediately from strong induction, where the proof that the base cases are correct follows from the proofs of the above various lemmas and theorems.

3 Graph Algorithms

3.1 The Generic Search Algorithm

Suppose we want to search a graph to determine whether there exists a path between a root node r and some other node in the graph. Then we get the following algorithm:

Search(r)
    put r into a bag
        while the bag is not empty
            remove a node v from the bag
            if v is unmarked
                mark v
                for each arc (v, w)
                    put w into the bag

Where we say a new vertex (and therefore a path to that new vertex) is discovered when it's marked. We can write this algorithm in a bit more 'formal' pseudo-code as such:

Search(r)
    set $\mathcal{B} := \{r\}$
        while $\mathcal{B} \neq \emptyset$
            let $v \in \mathcal{B}$
            set $\mathcal{B} \leftarrow \mathcal{B} \setminus \{v\}$
            if v is unmarked
                mark v
                for each arc (v, w)
                    set $\mathcal{B} \leftarrow \mathcal{B} \cup \{w\}$

Note that this algorithm allows multiple copies of the same node to be put into the bag. This *could* pose a problem as it would require our algorithm execute superfluous steps to terminate. We can get around this by putting edges into the bag instead of nodes. Let's update our pseudo-code to reflect this:

```
Search(r)
    set B := {(r, *)}
        while B ≠ ∅
            let (u, v) ∈ B
            set B ← B \ {(u, v)}
            if v is unmarked
                mark v
                set p(v) ← u
                for each arc (v, w)
                    set B ← B ∪ {(v, w)}
```

Where $(r, *)$ is some pseudo-edge and $p(v)$ is the parent of v.

3.1.1 Time Complexity

We search every edge out of v only once (when u is first marked). The edge is added to the bag once and removed from the bag once. Thus the time complexity is $O(m)$ (where m is the cardinality of the edge set.)

3.1.2 Validity

Theorem: Let G be a connected, undirected graph. The generic search algorithn finds every vertex in G.

Proof: We must show that every vertex is marked by the algorithm. We will do this by induction on the smallest number k of edges in a path from the vertex to the root.
Base Case: $k = 0$. Then v is the root node r, but r is marked by the algorithm.
Induction Hypothesis: Assume any vertex v that has a path of $k - 1$ or fewer edges to the root r is marked.
Inductive Step: Assume there exists a path P with k edges from v to r. i.e.

$$P = \{v = v_k, v_{k-1}, \ldots, v_1, v_0 = r\}$$

Then there is a path Q with $k - 1$ edges from $u = v_{k-1}$ to r. i.e.

$$Q = \{u = v_{k-1}, \ldots, v_1, v_0 = r\}$$

By our induction hypothesis u is marked. After we mark u we pace the edges incident to it in the bag. Therefore the edge (u, v) is added to the bag. Thus when (u, v) is removed from the bag we will mark v if it is not marked already. $\qquad\square$

3.1.3 Search Trees

Note that after running this algorithm, each non-root vertex has exactly one predecessor.

Theorem: Let G be a connected, undirected graph. Then the predecessor edges form a tree rooted at r.

Proof: We will prove this by induction on the number k of marked vertices.
Base Case: $k = 1$. The root vertex r is the first one marked. Trivially, this forms a tree.
Induction Hypothesis: Assume the predecessor edges for the first $k - 1$ marked vertices form a tree rooted at r.
Inductive Step: Let v be the kth vertex to be marked. Assume that v was marked when the edge (u, v) was removed. Then $u = p(v)$. But (u, v) was added to the bag when we marked vertex u. Therefore u is in the set S of the first $k - 1$ vertices to be marked. By the induction hypothesis, the predecessor edges for S form a tree T rooted at r. Thus $T \cup (p(v), v)$ is a tree rooted at r on the first k marked vertices. $\square$

Theorem: For any vertex v, the path from v to r given by the search tree T of predecessor edges is a shortest path.

3.1.4 The Bag

The bag is just shorthand for "some data structure". In this algorithm, the bag can be any data structure we want but our choice has some important consequences. Specifically, for the graph search algorithm we really want to use one of three data structures, each of which will give a distinct algorithm:

1. Queue: The generic search algorithm becomes Breadth-First Search

2. Stack: The generic searcg algorithm becomes Depth-First Search

3. Priority Queue: The generic search algorithm becomes a Minimum Spanning Tree algorithm

3.2 Breadth-First Search

Breadth-First search is an implementation of the generic search algorithm using a queue as the data structure (the bag.) Because we use a queue, edges are added in the order of their distance from r. The vertices are then marked in order of their distance from r.

3.2.1 Breadth-First Search & Bipartite Graphs

Theorem: Let G be a graph. Then G is bipartite if and only if it contains no odd-length cycles.

Proof: ($\Rightarrow$) Assume G contains as a subgraph an odd-length cycle C. Let

$$C := \{v_0, v_1, \ldots, v_{2k}\}$$

WLOG assume $v_0 \in Y$. Then

$$v_0 \in Y \Rightarrow v_1 \in X \Rightarrow v_2 \in Y \Rightarrow v_3 \in X \ldots \Rightarrow v_{2k} \in Y$$

but $(v_0, v_{2k}) \in E$. Thus $v_0 \in Y \Rightarrow v_{2k} \in X$, a contradiction. Thus G is not bipartite.

($\Leftarrow$) Assume G contains no odd-length cycles. Then select an arbitrary root vertex r and run the BFS algorithm. We want to show that if G contains no odd-length cycles we can find two sets X, and Y such that each edge goes only between X and Y. Recall that for every edge (u, v) we have that u, and v are either in the same layer or are in adjacent layers. Now set

$$X := \bigcup_{\ell \text{ odd}} S_\ell, \text{ and, } Y := \bigcup_{\ell \text{ odd}} S_\ell$$

where S is a set of nodes in the layer of a BFS-generated search tree, and ℓ is the level of the node form the root. We claim that this choice of X, and Y gives a bipartite graph. Suppose that for each non-tree edge (u, v), that u, and v are in adjacent layers. Assume that there exists a non-tree edge (u, v) with u, and v in the same layer. Let z be the closest common ancestor of u, and v in the search tree T from u to v; let Q be the part in T from v to z. Since u, and v are in the same layer $|P| = |Q|$. But then the cycle $C = P \cup Q \cup (u, v)$ has an odd number of edges, so (u, v) can't exist, a contradiction. $\qquad\square$

3.3 Depth-First Search

Running the generic search algorithm but using a stack as our bag gives depth-first search. An interesting property of depth-first search is that it partitions the edges of an undirected graph into two types:

1. Tree edges: Predecessor edges in the DFS tree T

2. Back edges: Edges where one endpoint is an ancestor of the other endpoint in T

we cannot have edges of the following type:

3. Cross edges: Edges where neither endpoint is an ancestor of the other in T

We can also define depth-first search recursively as such:

RecursiveDFS(r)
    mark r
    for each edge (r, v)
        if v is unmarked
            set $p(v) \leftarrow r$
            RecursiveDFS(v)

3.3.1 Ancestral Edges Theorem

Theorem: Let T be a depth-first search-generated tree in an undirected graph G. Then for each edge (u, v), either u is an ancestor of v in T, or v is an ancestor of u.

Proof: WLOG assume that u is marked before v. Consider the time u is marked during RecursiveDFS(u). In RecursiveDFS(u), the algorithm examines every arc incident to u. This leads to two cases:
Case 1: v is unmarked when RecursiveDFS(u) examines (u, v). Then RecursiveDFS(u) sets $p(v) \leftarrow u$. Therefore (u, v) is a tree edge.
Case 2: v is marked when the algorithm examines (u, v). But v must have been marked after u, so it was marked during RecursiveDFS(u). Thus we have a series of vertices $\{u = w_0, w_1, \ldots, w_{\ell-1}, w_\ell = v\}$ where $p(w_k) = w_{k-1}$. Therefore u is an ancestor of v in T and is thus a back edge. $\square$

Corollary: Let T be a DFS tree in an undirected graph G. Then every non-tree edge is a back edge.

3.3.2 Previsit & Postvisit

We can add a clock to the RecursiveDFS(r) algorithm to record when we visit a vertex for the first time (previsit) and when we visit a vertex for the last time (postvisit). Let's update our recursive algorithm to include this.
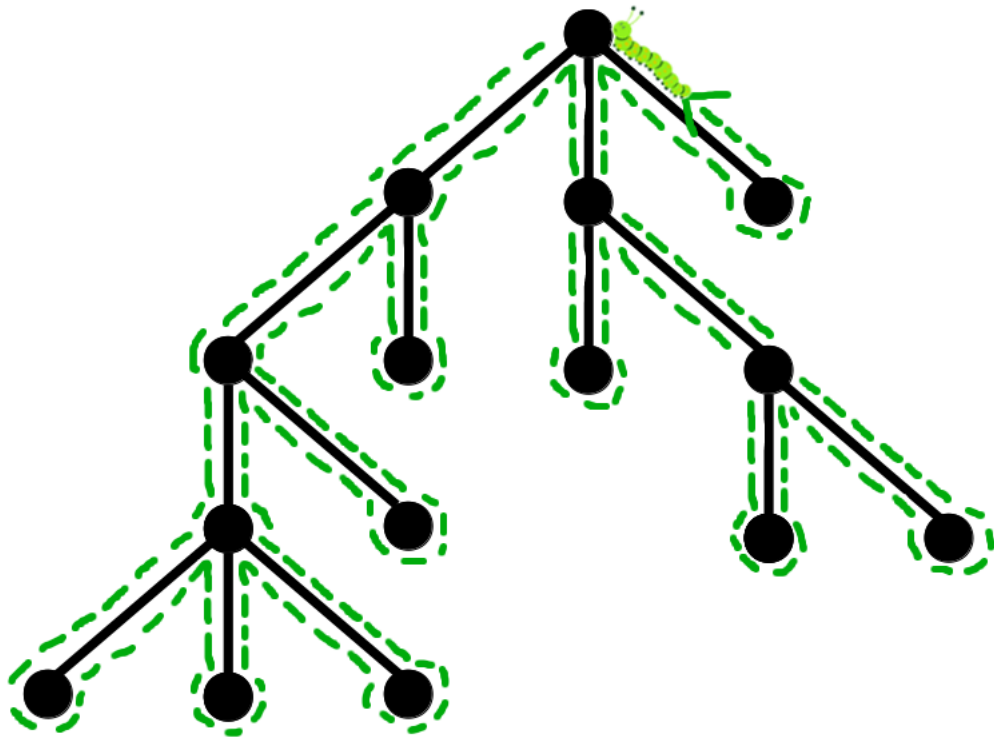
```
clock ← 0
RecursiveDFS(r)
     mark r
     pre(r) ← clock
     clock++
     for each edge (r, v)
          if v is unmarked
               set p(v) ← r
               RecursiveDFS(v)
     post(r) ← clock
     clock++
```
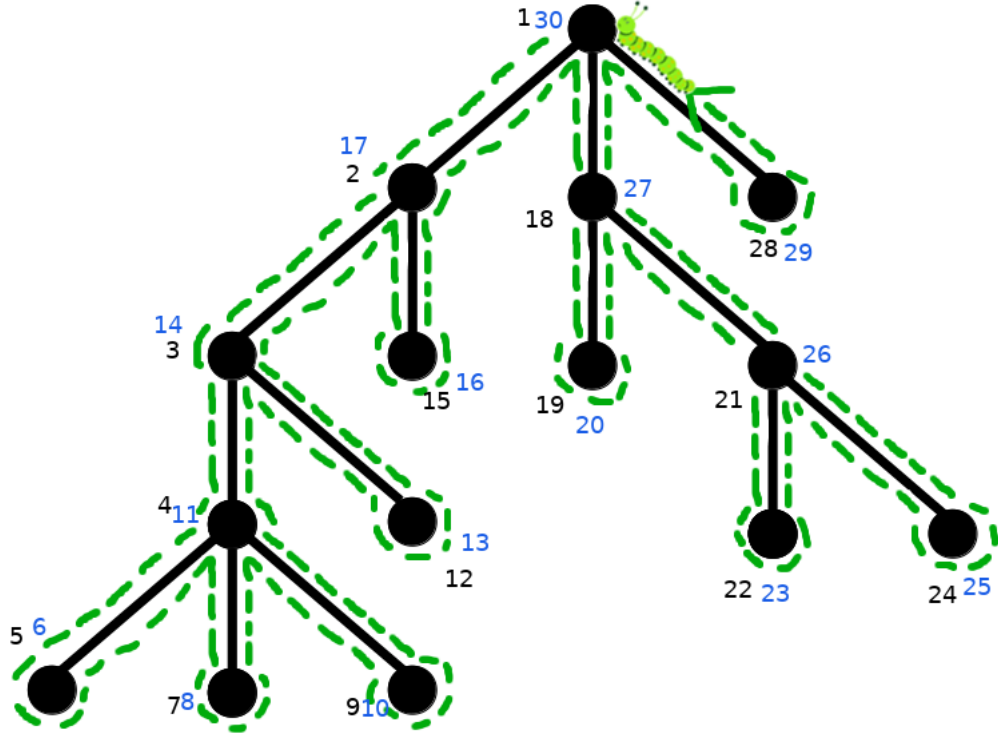
To visualize this imagine that a caterpillar is crawling along the graph like so:

The clock records the first time the caterpillar visits a node and the last time the caterpillar visits a node. Then, let the black number be the clock on the previsit, and let the blue number be the clock on the postvisit. We get this

25

Note that pre(v) is the time that the caterpillar arrives at the subtree rooted at v, and post(v) is the time when the caterpillar leaves that subtree.

Depth-first search expores one subtree of the tree at-a-time. This means that we can represent each subtree as an interval over $\mathbb{R}$. More strongly, the set of intervals is laminar (for every pair of intervals, I_1, I_2, either I_1, and I_2 are disjoint, or one is a subset of the other.)

3.3.3 Postvisit Times

For an arc (u, v) the following always hold:

1. Tree arcs: post(v) < post(u)

2. Forward arcs: post(v) < post(u)

3. Backward arcs: post(u) < post(v)

4. Cross arcs: post(v) < post(u)

Note that only for backward arcs is post(u) < post(v). This is a useful result.

3.3.4 Directed Acyclic Graphs

Theorem: A directed graph G is acyclic if and only if depth-first search produces no backward arcs.

Proof: ($\Rightarrow$) Let DFS give a backward arc (u, v). By definition u is a descendant of v in the DFS tree T. Therefore there exists a path P, given by $P = \{v = v_0, v_1, \ldots, v_k = u\} \in T$. But then $P \cup (u, v)$ is a directed cycle in T, a contradiction.
($\Leftarrow$) Assume DFS gives no backward arcs. Suppose there is a directed cycle $C = \{v_0, v_1, \ldots, v_k, v_0\} \in T$. Then, as there are no backward arcs we have that $\text{post}(v_0) > \text{post}(v_1) > \text{post}(v_2) > \ldots > \text{post}(v_k) > \text{post}(v_0)$, a contradiction. $\square$

Corollary: We have an $O(m)$ algorithm to test whether the graph is acyclic.

Proof: Run DFS and check for backward arcs.

3.3.5 Topological Orderings

A directed graph has a "Topological Ordering" if the vertices can be ordered such that every arc is from right to left.

Theorem: A directed graph G has a topological ordering if and only if DFS produces no backward arcs.

Proof: ($\Rightarrow$) If DFS produces a backward arc then G contains a cycle C. Let the cycle $C = \{v_0, v_1, \ldots, v_k, v_0\}$, where WLOG v_0 is the leftmost vertex of the cycle in the order. But then the arc (v_0, v_1) goes from left to right, a contradiction.
($\Leftarrow$) Assume DFS gives no backward arcs. Then for every arc (u, v), we have that $\text{post}(u) > \text{post}(v)$. Thus, if we place each vertex v at coordinate $\text{post}(v)$ on the x-axis, then every arc goes from right to left and hence G has a topological ordering. $\square$

Corollary: We have an $O(m)$ algorithm to test whether or not we have a topological ordering.

Corollary: A directed graph is acyclic if and only if it has a topological ordering.

3.3.6 Disconnected Graphs & Strongly Connected Components

For disconnected undirected graphs, a depth-first search from a vertex r will find the entire component containing the vertex r. Therefore, if we want to discover the entire graph, not just the component containing r, we must run DFS from another vertex (in a different component).

In a directed graph, a subgraph S is said to be strongly connected if for every vertex pair $u, v \in S$, there exists a directed path from v to u and vice versa. DFS can be used to decompose a directed graph into its strongly connected components very quickly.

Theorem: There is an $O(m)$ algorithm to find stromngly connected components of directed graphs.

4 Greedy Algorithms

Greedy algorithms are a class of algorithms characterized by how they make locally optimal choices at each step. They have a few properties:

- They're fast!

- They're usually very simple!

- They're easy to code!

- They rarely work!

Thus for this topic in the course the goal is to understand some basic (working) greedy algorithms along with some basic techniques and when such techniques can succesfully be applied.

4.1 The Task Scheduling Problem

Suppose a firm receives a set J of n job orders from some customers, but the firm can only process one order at once. Suppose the job of customer i will take t_i units of time to complete. Assume each customer wants their job done as quickly as possible (and will pay accordingly.)

Assume the firm processes the jobs in order $\{1, 2, \ldots, n\}$. Then the wating time of customer ℓ will be

$$w_\ell = \sum_{i=1}^{\ell} t_i$$

Under the assumption that the customers will pay according to time, the firm wants to minimize the sum of the waiting times. That is, they want to minimize

$$\sum_{\ell=1}^{n} w_\ell = \sum_{\ell=1}^{n} \sum_{i=1}^{\ell} t_i$$

We'll use a greedy approach to solve this problem:

Schedule(J)
      sort the list of orders in non-decreasing order of their completion time
      schedule the jobs in this order

Unfortunately, unlike divide and conquer algorithms that generally work, and where most of the proofs follow directly from strong induction (requiring only the base cases be verified), greedy algorithms don't work in general and thus have to be proved in more creative ways. Let's now prove the task scheduling algorithm works.

Proof: Let the algorithm schedule the tasks in the order $\{1, 2, \ldots, n\}$. Assume there is a better schedule $\mathcal{S}$. Then there must be a pair of jobs i, j such that

1. Job i is (WLOG) scheduled immediately before job j by $\mathcal{S}$.

2. Job i is longer than job j. In other words, $t_i > t_j$.

This implies that exchanging jobs i, and j leads to a more optimal schedule $\hat{\mathcal{S}}$. Observe that after exchanging the two jobs the waiting time for all the other jobs remains the same (draw a picture if you need to convince yourself.) That is,

$$\hat{w}_k = w_k, \forall k \neq i, j$$

but obviously

$$\hat{w}_i + \hat{w}_j < w_i + w_j$$

a contradiction. Thus $\mathcal{S}$ was the optimal solution. $\square$

The runtime of this algorithm is $O(n \log n)$ as the only real step is sorting the list of jobs by length.

4.2 The Interval Selection Problem (Class Scheduling)

Suppose there is one classroom and as set $I = \{1, 2, \ldots, n\}$ of n classes that want to use the room. Suppose each class $i \in I$ has a start time s_i, and a finish time f_i. Our goal is to book as many classes into that room as possible (in a certain period of time) but no more than one class can use the room at once. Let's try using a greedy algorithm to solve this problem. There's a couple different possibilities we can try:

1. First start: Choose the class that starts first and iterate over the remaining non-conflicting classes

2. Shortest duration: Select the class that is the shortest and then iterate over the remaining non-conflicting classes

3. Minimum Conflict: Find the course that conflicts with the fewest other courses and iterate over the remaining non-conflicting classes

4. First finish: Choose the class that finishes first and iterate over the remaining non-conflicting courses.
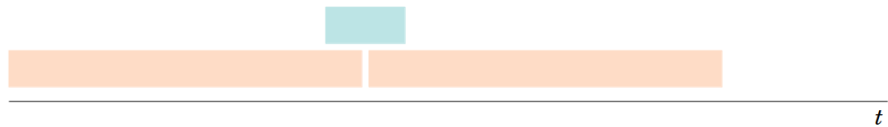
But do any of these actually work? Let's see.

Does first start work? Nope. Consider the example:
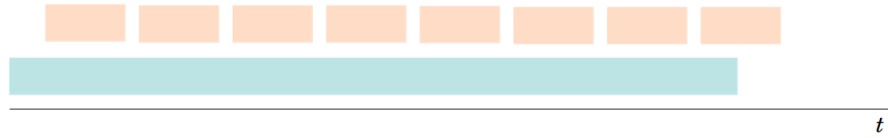
Note that that algorithm would select the blue interval where the eight orange intervals would be the optimal solution.

How about shortest duration? Nope. Consider the following:

This algorithm would select the blue interval where the two orange intervals would be the optimal solution.

What about minimum conflict? Nope.
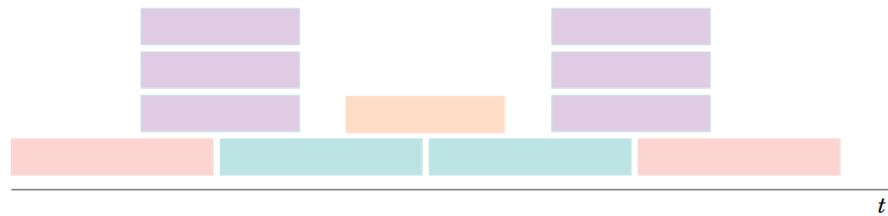
This algorithm would select the orange interval because it only conflicts with the two blue intervals, and then two of the purple intervals, but the blue intervals plus the red intervals along the bottom is the optimal solution even though the red intervals each conflict with the three purple intervals, and the blue intervals each conflict with four intervals.

First finish? Yep this one works. First we'll write down the pseudo-code algorithm and then we'll prove that it works.

FinishFirst(I)

    sort the classes by their finish time in non-decreasing order

    initialize the set $S := \emptyset$ to be the set of scheduled classes

    while $I \neq \emptyset$

        let i be the lowest index class in I

        set $S \leftarrow S \cup \{i\}$ and set $I \leftarrow I \setminus \{i\}$

        for each class $j \in I$

            if $s_i < f_i$

                set $I \leftarrow I \setminus \{i\}$

    return S

Before we prove the algorithm we need some help from a lemma

Lemma: Let c be the class with the earliest finish time. Then there is an optimal solution that contains c.

Proof: Recall the classes are indexed such that $f_1 \leq f_2 \leq \cdots \leq f_n$. Then take an optimal solution $\mathcal{S}$ and assume that $c \in \mathcal{S}$. Let i be the lowest index in $\mathcal{S}$. We claim that $\mathcal{S} \setminus \{i\} \cup \{c\}$ is a feasible allocation of maximum size. This follows as $f_i \leq s_j$ for any $j \in \mathcal{S} \setminus \{i\}$. But $f_1 \leq f_i \leq s_j$ so c doesn't conflict with any class in $\mathcal{S} \setminus \{i\}$. Therefore $\mathcal{S} \setminus \{i\} \cup \{c\}$ is feasible with cardinality equal to $|\mathcal{S}|$. □

We can now prove that the algorithm works.

Proof: We'll proceed by induction on the cardinality of the optimal solution.
Base Case: Let $|\mathrm{opt}(I)| = 1$. Then the algorithm outputs c and this is trivially an optimal solution.
Induction Hypothesis: If $|\mathrm{opt}(I)| = k$ then the algorithm outputs an optimal solution.
Inductive Step: Let $|\mathrm{opt}(I)| = k + 1$. Then the algorithm outputs $\{c\} \cup$ FirstFinish($I \setminus k$). But by the previous lemma there exists an optimal solution $\hat{\mathcal{S}}$ containing c. Thus $\hat{\mathcal{S}} \setminus \{I\}$ is an optimal solution for the subproblem $I \setminus X \Rightarrow |\mathrm{opt}(I)| = k$. But then by our induction hypothesis, the algorithm gives an optimal solution to the sub-problem $I \setminus X \Rightarrow |\mathrm{FirstFinish}(I \setminus X)| = k \Rightarrow |\{c\} \cup \mathrm{FirstFinish}(I \setminus X)| = k + 1$. Thus the algorithm outputs an optimal solution. □

There are at most n iterations, and it takes $O(n)$ time to find a class that finishes first. Therefore the runtime is $O(n^2)$. With better implementation, we can get the runtime down to $O(n \log n)$.

4.3 Dijkstra's Shortest Path Alorithm

Given a directed graph $G = (V, A)$ with a source vertex $s \in V$, let each arc arc $a \in A$ have a non-negative length ℓ_a. Then the length of a path $P \subseteq G$ is

$$\ell(P) = \sum_{a \in P} \ell_a$$

Our goal is to find the shortest path from s to every other vertex $v \in V$. We can do this using *Dijkstra's Shortest Path Algorithm.*

Dijkstra(s, G)
    set $\mathcal{S} := \emptyset$
    set $d(s) := 0$, and set $d(v) := \infty$, $\forall v \in V \setminus \{s\}$
    while $\mathcal{S} \neq V$
        if $v = \text{argmin}_{u \in V \setminus \mathcal{S}}$
            set $\mathcal{S} \leftarrow \mathcal{S} \cup \{v\}$
        for each arc (v, w)
            if $d(w) > d(v) + \ell_{vw}$ and $w \in V \setminus S$
                set $d(w) = d(v) + \ell_{vw}$

This is indeed a greedy algorithm because it myopically selects the vertex with the smallest label. Note this algorithm only calculates the shortest path distances, but it's easy to tweak it to keep track of the actual paths themselves:

Dijkstra(s, G)
    set $\mathcal{S} := \emptyset$
    set $\mathcal{T} := \emptyset$
    set $\text{pred}(s) \leftarrow \text{NULL}$, $\text{pred}(v) \leftarrow \text{--}$
    set $d(s) := 0$, and set $d(v) := \infty$, $\forall v \in V \setminus \{s\}$
    while $\mathcal{S} \neq V$
        if $v = \text{argmin}_{u \in V \setminus \mathcal{S}}$
            set $\mathcal{S} \leftarrow \mathcal{S} \cup \{v\}$
            set $\mathcal{T} \leftarrow \mathcal{T} \cup (\text{pred}(u, v))$
        for each arc (v, w)
            if $d(w) > d(v) + \ell_{vw}$ and $w \in V \setminus S$
                set $d(w) = d(v) + \ell_{vw}$
                set $\text{pred}(w) \leftarrow v$

Note that this is just a generalization of Breadth-First search!

To show that the algorithm works first observe that $\mathcal{S}$ is a set of vertices and $\mathcal{T}$ is a set of arcs. Then let $\mathcal{S}^k$ be the set of vertices in $\mathcal{S}$ at the end of the

k^{th} iteration, and let $\mathcal{T}^k$ be the set of arcs in $\mathcal{T}$ at the end of the kth iteration. Furthermore observe that the arcs in $\mathcal{T}^k$ are between the nodes in $\mathcal{S}^k$. Therefore $\mathcal{G}^k = (\mathcal{S}^k, \mathcal{T}^k)$ is just a directed graph. Note that $\mathcal{G} = \mathcal{G}^n$ is the final outcome of the algorithm. Next consider the following lemma.

Lemma: The graph $\mathcal{G}^k$ is a directed tree rooted at s.

Proof: We will proceed by induction on k.
Base Case: $k = 1$. Then $\mathcal{S}^1$ contains only s, and $\mathcal{T}^1 = \emptyset$ as $\text{pred}(s) = \text{NULL}$. Therefore it's trivially an arborescence.
Induction hypothesis: $\mathcal{G}^{k-1} = (\mathcal{S}^{k-1}, \mathcal{T}^{k-1})$ is an arborescence rooted at s.
Inductive Step: Consider $\mathcal{G}^k = (\mathcal{S}^k, \mathcal{T}^k)$. Let v_k be the vertex added to $\mathcal{S}$ in the kth iteration. Then $\mathcal{S}^k = \mathcal{S}^{k-1} \cup \{v_k\} \Rightarrow \mathcal{T} = \mathcal{T}^{k-1} \cup (\text{pred}^{k-1}(v_k), v_k)$. Therefore v_k has in-degree exactly 1 in G^k, and out-degree exactly 0. Then by our induction hypothesis $\mathcal{G}^{k-1}$ is an arborescence rooted at s. Furthermore by definition, $\text{pred}^{k-1}(v_k) \in \mathcal{S}^{k-1} \Rightarrow \mathcal{G}^k = (\mathcal{S}^k, \mathcal{T}^k)$ is an arborescence rooted at s with v_k as a leaf. $\qquad\square$

Now we just need to show that the paths in the outputted arborescence are the shortest paths in the graph and we'll have proved the algorithm.

Theorem: The graph $\mathcal{G}^k = (\mathcal{S}^k, \mathcal{T}^k)$ gives the shortest distances from s to every vertex in $\mathcal{S}^k$.

Proof: Again, we'll proceed by induction on k.
Base Case: $k = 1$. Then this is trivially true as $d^1(s) = 0 = d^*(s)$.
Induction Hypothesis: Assume $\mathcal{G}^{k-1} = (\mathcal{S}^{k-1}, \mathcal{T}^{k-1})$ gives the shortest distance from s to every vertex in $\mathcal{S}^{k-1}$. i.e. $d^{k-1}(v) = d^*(v), \forall v \in \mathcal{S}^{k-1}$.
Inductive Step: Consider the graph $\mathcal{G}^k = (\mathcal{S}^k, \mathcal{T}^k)$. Let v_k be the vertex added to $\mathcal{S}$ in the kth iteration. Take the sortest path P from s to v_k with as many arcs in common with $\mathcal{G}^k$ as possible. Let x be the last vertex of $\mathcal{G}^{k-1}$ in P, and let $y \notin \mathcal{S}^{k-1}$ be the vertex after x in P. Such a y exists or we are done. Since y is on the shortest path from s to v_k and the arc lengths are non-negative, we have that $d^*(y) \leq d^*(v_k) < d^k(v_k)$. We have a strict inequality or we'd be done. But since $x \in \mathcal{S}^{k-1}$ we have that $d^k(y) \leq d^{k-1}(x) + \ell(x, y) = d^*(x) + \ell(x, y) = d^*(y)$. And therefore $d^k(y) \leq d^*(y) < d^k(v_k)$, which contradicts our choice of v_k as $d^k(y) < d^k(v_k)$. $\qquad\square$

There are n iterations and there are at most n distance updates in each iteration. Therefore the runtime is $O(n^2)$. We can implement this algorithm faster, in $O(m \log n)$ time, using a heap.

5 Dynamic Programming

6 Network Flows

7   Data Structures

7.1   Priority Queues & Heaps

7.2   Hashing

7.3   String Matching

7.4   Binary Search Trees

7.5   Data Structure for Disjoint Sets

7.6   Segment Trees