

# COMP 251 Study guide

Francis Piche

February 14, 2018

# Contents

<b>1</b>	<b>Disclaimer</b>	<b>6</b>
<b>2</b>	<b>Preliminaries</b>	<b>6</b>
<b>I</b>	<b>Recursive Algorithms</b>	<b>6</b>
<b>3</b>	<b>Divide + Conquer Algorithms</b>	<b>7</b>
3.1	MergeSort . . . . .	7
3.2	Binary Search . . . . .	8
3.3	Run Time of Divide + Conquer in General . . . . .	8
3.4	Aside on Recurrences: Domain Transformation . . . . .	9
<b>4</b>	<b>Master Theorem</b>	<b>10</b>
4.1	Tree Method to Prove Master Theorem . . . . .	13
<b>5</b>	<b>Multiplication</b>	<b>14</b>
5.1	Grade School Multiplication . . . . .	14
5.2	Russian Peasant Multiplication . . . . .	14
5.3	Divide + Conquer Multiplication . . . . .	14
5.4	Fast Fourier Transforms . . . . .	15
5.5	Multiplying Matrices . . . . .	16
5.6	Fast Exponentiation . . . . .	17
<b>6</b>	<b>The Median Problem</b>	<b>18</b>
6.1	The Selection Problem . . . . .	18
6.2	Median of Medians . . . . .	19
<b>7</b>	<b>Finding the Closest Pair of Points in the Plane</b>	<b>20</b>
7.1	Exhaustive Search . . . . .	20
7.2	2-D case . . . . .	20
7.3	Widening the Bottleneck . . . . .	21
7.4	The Finished Algorithm . . . . .	22
7.5	The Runtime (Enhanced) . . . . .	22
<b>II</b>	<b>Graph Algorithms</b>	<b>22</b>

<b>8</b>	<b>Theorems About Undirected Graphs</b>	<b>23</b>
8.1	Handshaking Lemma . . . . .	23
8.2	Leaf Existence . . . . .	23
8.3	Number of edges in a Tree . . . . .	24
8.4	Halls Theorem . . . . .	24
<b>9</b>	<b>Breadth First Search</b>	<b>25</b>
9.1	Generic Search Algorithm . . . . .	25
9.1.1	Revised Generic Search Algorithm . . . . .	25
9.1.2	The Running Time . . . . .	26
9.1.3	Validity . . . . .	26
9.2	Search Trees . . . . .	27
9.3	Choices of Bags . . . . .	27
9.4	BFS Trees . . . . .	27
9.4.1	Structure . . . . .	27
9.4.2	BFS on Bipartite Graphs . . . . .	28
<b>10</b>	<b>Depth First Search</b>	<b>28</b>
10.1	DFS Trees . . . . .	28
10.2	Recursive DFS . . . . .	29
10.3	Ancestral Edges . . . . .	29
10.4	Previsit and Postvisit . . . . .	30
10.5	Directed Graph BFS Tree Structure . . . . .	31
10.6	Example: Directed Acyclic Graphs . . . . .	31
10.7	Example: Topological Ordering . . . . .	32
<b>III</b>	<b>Greedy Algorithms</b>	<b>32</b>
<b>11</b>	<b>Scheduling</b>	<b>33</b>
11.1	Task Scheduling . . . . .	33
11.1.1	Running Time . . . . .	34
11.2	Class Scheduling . . . . .	34
11.2.1	First Start . . . . .	34
11.2.2	Shortest-Duration . . . . .	34
11.2.3	Minimum Conflict . . . . .	35
11.2.4	Last Start . . . . .	35

<b>12 The Shortest Path Problem</b>	<b>37</b>
12.1 Dijkstra's Shortest Path Algorithm . . . . .	37
12.1.1 Special Case, All Arcs Have Distance 1 . . . . .	37
12.1.2 Shortest Path Graph . . . . .	37
12.1.3 Shortest Path Tree . . . . .	38
12.1.4 The Running Time . . . . .	39
<b>13 Huffman Codes</b>	<b>39</b>
13.1 Data Encoding . . . . .	39
13.1.1 Morse Code . . . . .	40
13.2 Prefix Codes . . . . .	40
13.3 Binary Tree Representations . . . . .	40
13.3.1 Letter to Leaf Assignment . . . . .	41
13.3.2 Tree Shape . . . . .	42
13.4 The Key Formula . . . . .	42
13.5 The Algorithm . . . . .	43
13.5.1 Proof Of Correctness . . . . .	43
13.5.2 Running Time . . . . .	43
<b>14 Minimum Spanning Tree Problem</b>	<b>44</b>
14.1 Kruskal's Algorithm . . . . .	44
14.2 Prim's Algorithm . . . . .	44
14.3 Boruvka's Algorithm . . . . .	44
14.4 Running Times . . . . .	44
14.4.1 Kruskal's Running Time . . . . .	44
14.4.2 Prim's Running Time . . . . .	45
14.4.3 Boruvka's Running Time . . . . .	45
14.5 Proof That They All Work . . . . .	45
14.6 The Cycle Property . . . . .	46
14.7 The Reverse Delete Algorithm . . . . .	47
14.7.1 Runtime of Reverse Delete . . . . .	47
14.7.2 Proof of Reverse Delete . . . . .	47
<b>15 The Clustering Problem</b>	<b>47</b>
15.1 Maximum Spacing Clustering . . . . .	48
15.2 Reverse-Delete Clustering Algorithm . . . . .	49
15.2.1 Proof Of Reverse-Delete Clustering . . . . .	49

<b>16 The Set Cover Problem</b>	<b>50</b>
16.1 The Greedy Set Cover Algorithm . . . . .	50
16.2 Approximation Algorithms . . . . .	51
16.3 Proof that Greedy Set Cover Almost Works . . . . .	52
16.4 Running Time . . . . .	53
<b>17 The Hitting Set Problem</b>	<b>53</b>

## 1 Disclaimer

The material of this document was transcribed from Prof. Adrian Vetta's lecture recordings for COMP251 in the winter of 2018. Extra notes, clarifications or interpretations added by me may not be correct. All images are taken directly from Prof. Adrian Vetta's lecture slides. I claim no ownership over these images or the content taken from the slides.

## 2 Preliminaries

In this course an algorithm is considered **good** if it:

- Works
- Runs in polynomial time. Meaning it runs, in  $O(n^k)$  time. Where  $n$  is (always) the size of the problem. (Number of elements in a list to be sorted etc.)
- Scales multiplicatively with computational power. (If your computer is twice as fast, the problem is solved at least twice as fast)

A bad algorithm is one that:

- Doesn't always work
- Runs in exponential time or greater. Meaning:  $O(k^n)$  time.
- Does not scale well with computational power. (Your computer is twice as fast, but barely any performance boost).

## Part I

# Recursive Algorithms

I won't be going into detail on the specifics of things like how recursion works, MergeSort, BinarySearch, solving recurrences, Big  $O$ , etc. as it's considered prerequisite material. If you need some review, my COMP250 study guide is still publicly available.

## 3 Divide + Conquer Algorithms

Examples:

- MergeSort
- BinarySearch

### 3.1 MergeSort

The MergeSort algorithm involves splitting a list of  $n$  elements in half, sorting each half recursively, and merging the sorted lists back into one. It takes time  $T(\frac{n}{2})$  to sort the list of half size, and time  $O(n)$  to merge the list back together. So the recurrence relation for MergeSort is given by:

$$T(n) = 2T(\frac{n}{2}) + cn$$

where  $c$  is some constant.

**Theorem 1.** MergeSort runs in time  $O(n \log(n))$ .

*Proof.* Add **dummy numbers** (extra "padding" to the list), until  $n$  is a power of two.  $n = 2^k$ . We can do this because  $O()$  gives an **upper bound**, and adding numbers will make our solution take longer than the real one. Doing this will make solving the recurrence easier.

Unwinding the formula:

$$\begin{aligned} T(n) &= 2(2(T(\frac{n}{4}) + c\frac{n}{2}) + cn) \\ &= 2^2(T(\frac{n}{4}) + 2cn) \\ &= 2^3(T(\frac{n}{8}) + 3cn) \\ &= 2^4(T(\frac{n}{16}) + 4cn) \end{aligned}$$

Notice we have a pattern emerging.

$$= 2^k(T(1)) + kcn$$

Recall  $2^k = n$ , so  $k = \log_2(n)$  and  $T(1) = 1$  so:

$$= n + n \log_2(n)$$

Which is  $O(n \log n)$ . □

## 3.2 Binary Search

Binary search involves splitting your sorted list into two, and searching that half. So our recurrence is given by:

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

where  $c$  represents the constant work (comparisons, setting new bounds etc.)

**Theorem 2.** Binary Search is  $O(\log_2(n))$ .

*Proof.* Again we add dummy numbers so that  $n$  is a power of two.  $n = 2^k$

We begin with our recurrence:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c \\ &= T\left(\frac{n}{4}\right) + c + c \\ &= T\left(\frac{n}{8}\right) + c + c + c \\ &= T\left(\frac{n}{2^k}\right) + kc \\ &= T(1) + \log_2(n) \end{aligned}$$

since  $k = \log_2(n)$  which is  $O(\log_2(n))$ . □

## 3.3 Run Time of Divide + Conquer in General

Divide and Conquer is a technique of solving problems that involves taking one large problem of size  $n$ , and breaking it down into  $a$  smaller problems of size  $\frac{n}{b}$ , and solving those problems recursively. They are then combined to produce a solution in time poly-time:  $O(n^d)$ .

So the run-time of a divide and conquer algorithm is:

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

In the case of MergeSort,  $a = 2$ ,  $b = 2$ ,  $d = 1$ .

In the case of BinarySearch,  $a = 1$ ,  $b = 2$ ,  $d = 0$ .



### 3.4 Aside on Recurrences: Domain Transformation

Note that the recurrence for MergeSort is really:

$$T'(n) \leq T'(\lfloor n/2 \rfloor) + T'(\lceil n/2 \rceil) + cn$$

Which we simplified by adding dummy entries. However, we can also say this:

$$T'(n) \leq 2T'(\frac{n}{2} + 1) + cn$$

But the +1 doesn't fit with our previous method.

We'll use **domain transformation** to solve this, starting with:

$$\begin{aligned} T(n) &= T'(n + 2) \\ &\leq T'(\frac{n+2}{2} + 1) + c(n+2) \end{aligned}$$

plugging in our expression from above

$$\leq T'(\frac{n+2}{2} + 1) + c'(n)$$

absorbing the +2 into  $c$ .

$$= T'(\frac{n}{2} + 2) + c'(n)$$

simplifying the fraction.

$$= T(\frac{n}{2}) + c'n$$

from our domain transformation at the beginning. Solving this the usual way, we get:

$$T(n) = O(n \log(n))$$

But again from our domain transformation:

$$T(n) = T'(n + 2)$$

, so

$$T'(n) = T(n - 2) = O(n \log(n))$$

So we've shown that  $T'(n)$  has the same upper bound as  $T(n)$ .

## 4 Master Theorem

**Theorem 3.** If  $T(n) = aT(n/b) + O(n^d)$  for constants  $a > 0$ ,  $b > 1$ ,  $d \geq 0$ , then:

$$\begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log(n)) & \text{if } a = b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

These cases are just a few that occur often in practice when dealing with divide + conquer algorithms.

*Proof.* First we'll need two things. One is the geometric series, and the other is a law of logarithms. Professor Vetta proved them in class, and honestly I doubt you'd be asked to prove them on an exam, but it's good proof practice to go through them so I'll do it here.

$$\sum_{k=0}^l x^k = \frac{1 - x^{l+1}}{1 - x}$$

Proof:

Starting with:

$$(1 - x) \sum_{k=0}^l x^k$$

We can expand it out:

$$= \sum_{k=0}^l x^k - \sum_{k=0}^l x^{k+1}$$

Simplifying the sigma notation:

$$= \sum_{k=0}^l x^k - \sum_{k=1}^{l+1} x^k$$

All terms will cancel except:

$$= x^0 - x^{l+1} = 1 - x^{l+1}$$

Divide through by  $1 - x$

$$= \frac{1 - x^{l+1}}{1 - x}$$

Our second fact to derive is this law of logs:

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

Using the power rule of logarithms:

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

similarly,

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

so,

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

Now we're ready for the proof.

Assume  $n$  is a power of  $b$ , and split up the problem into all its chunks.

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

(this is just if you'd "unwind" the whole recursion down to its simplest form like we did in the MergeSort/Binary Search proofs.)

Each term is the amount of work it will take at each level of the recursion.

Notice you can factor out:

$$\begin{aligned} &= n^d \left(1 + a\left(\frac{1}{b}\right)^d + a^2\left(\frac{1}{b^2}\right)^d + \dots + a^l\left(\frac{1}{b^l}\right)^d\right) \\ &= n^d \left(1 + \left(\frac{a}{b}\right)^d + \left(\frac{a}{b}\right)^{2d} + \dots + \left(\frac{a}{b}\right)^{ld}\right) \end{aligned}$$

That looks like a geometric series! So let's look at the cases:

Case 1:  $a < b^d$

Applying the geometric series formula:

$$\begin{aligned}
&= n^d \sum_{k=0}^l \left(\frac{a}{b^d}\right)^k \\
&= n^d \frac{1 - \left(\frac{a}{b^d}\right)^{l+1}}{1 - \frac{a}{b^d}}
\end{aligned}$$

we can remove the  $\frac{a}{b^d}^{l+1}$  term with this inequality (since the term doesn't depend on  $n$ ):

$$\leq n^d \frac{1}{1 - \frac{a}{b^d}}$$

which is  $O(n^d)$ .

Case 2:  $a = b^d$

Since  $\frac{a}{b^d} = 1$ :

$$= n^d(1 + 1 + 1 + \dots + 1)$$

There are  $l + 1$  terms, but we said  $n$  was a power of  $b$ , ( $n = b^l$ ) so,  $l = \log_b(n)$ , thus:

$$= n^d(\log_b(n) + 1)$$

which is  $O(n^d \log_b(n))$

Case 3:  $a > b^d$

Again from geometric series, and multiplying through by -1:

$$n^d \frac{\left(\frac{a}{b^d}\right)^{l+1} - 1}{\frac{a}{b^d} - 1}$$

Again this inequality holds:

$$\leq n^d \frac{\left(\frac{a}{b^d}\right)^{l+1}}{\frac{a}{b^d} - 1}$$

Which is  $O(n^d \left(\frac{a}{b^d}\right)^l)$  which we can simplify:

$$\left(\frac{n}{b^l}\right)^d a^l$$

but  $n = b^l$ , so:

$$\begin{aligned} &= (1)a^l \\ &= a^{\log_b(n)} \end{aligned}$$

now by our second fact:

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

which is  $O(n^{\log_b(a)})$

□

It's **much** more important to understand the proof than it is to memorize the theorem.

## 4.1 Tree Method to Prove Master Theorem

A more intuitive way to think of the proof is with a *Recursion Tree*.

The root node of the tree has label  $n$ , and each node has  $a$  children (except the leaves).  $a$  is called the *branching factor*. Each child is labelled  $\frac{n}{b^d}$  where  $d$  is the depth. The labels represent the size of the sub problems.

The number of nodes at each level is  $a^d$ .

Case 1 is when the root level "dominates" all other levels, so the running time is just  $O(f(n))$  where  $f(n)$  is the amount of work at the root level.

Case 2 is when all levels are roughly the same weight. So the total running time is just  $O(f(n)l)$  where  $l$  is the number of levels.

Case 3 is when the leaves dominate, so the running time is  $O(a^l)$  since the leaves each take time  $O(1)$ , and there are  $a^l$  of them.

## 5 Multiplication

### 5.1 Grade School Multiplication

This takes  $n^2$  multiplications when you multiply two  $n$ -digit numbers. so the runtime is  $\Omega(n^2)$

### 5.2 Russian Peasant Multiplication

Super weird looking algorithm but it works!

---

```
Mult(x,y){  
  if x = 1 then output y  
  if x is odd then output y + Mult(floor(x/2),2y)  
  if x is even then output Mult(x/2, 2y)  
}
```

---

This actually comes from if you take the binary representation of  $x$ : say  $x = 46_{10}$  then  $x = 101110_2$ . The bits that are 1's will have the  $y$  added step, and the zero bits will just have the doubling step. Weird right?

Notice that this means the number of steps is just the number of bits in  $x$ . The number of digits in the result will be at most  $2n$ , so if we need to then add these, we add at most  $n$  numbers of  $2n$  digits so takes time  $O(n^2)$

### 5.3 Divide + Conquer Multiplication

Notice that a number  $x$  can be written as:

$$x = x_n x_{n-1} \dots x_{\frac{n}{2}+1} x_{\frac{n}{2}} \dots x_2 x_1$$

where the  $x_i$  are the digits.

Then we have:

$$x = 10^{\frac{n}{2}} x_L + x_R$$

where  $n$  is the number of digits,  $x_L$  is the first  $\frac{n}{2}$  digits, and  $x_R$  is the last  $\frac{n}{2}$

So by expanding:

$$xy = (10^n x_L y_R + 10^{\frac{n}{2}}(x_L y_R + x_R y_L) + x_R y_R)$$

Notice that this now involves four products of  $\frac{n}{2}$  digit numbers. So the recursion is:

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

We have  $a = 4, b = 2, d = 1$ , which is case 3 of the master theorem.

Which means the running time is:

$$O(n^{\log_2(4)})$$

which simplifies to:

$$O(n^2)$$

Thanks to Gauss, we can actually use this fact:

$$x_L y_R + x_R y_L = x_R y_R + x_L y_L - (x_R - x_L)(y_R - y_L)$$

which is actually only 3 unique products. (adding is cheap)

So our new running time is:

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

which is case 3 of the master theorem, so

$$\begin{aligned} O(n^{\log_2(3)}) \\ = O(n^{1.59}) \end{aligned}$$

## 5.4 Fast Fourier Transforms

These are  $O(n \log(n))$  for multiplying n-bit numbers. They'll be studied more in-depth at the end of the course (time-permitting).

## 5.5 Multiplying Matrices

There are  $n$  multiplications to calculate each entry of the result matrix, and there are  $n^2$  entries, so  $O(n^3)$

Using divide + conquer, divide into 4 sub-matrices:

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ \dots & \dots & \dots & \dots & \dots \\ x_{d1} & x_{d2} & x_{d3} & \dots & x_{dn} \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

So if we let:

$$x = \begin{bmatrix} A & B \\ C & D \end{bmatrix} y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

then:

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

So multiplying involves eight products with  $\frac{n}{2} \times \frac{n}{2}$  and the recurrence is:

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

which is Case 3 of the master theorem, so runtime is  $O(n^{\log_2 8})$  which is  $O(n^3)$ , no improvement.

There actually is a trick to do better.

Claim:

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

is the same as:

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

where:

$$S_1 = (B - D)(G + H)$$

$$S_2 = (A + D)(E + H)$$



$$S_3 = (A - C)(E + F)$$

$$S_4 = (A + B)H$$

$$S_5 = A(F - H)$$

$$S_6 = D(G - E)$$

$$S_7 = (C + D)E$$

which is only 7 products! (The additions are negligible)

So we have:

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

Which is Case 3 of the master theorem, so  $O(n^{\log_2(7)})$  which is  $O(n^{2.81})$

## 5.6 Fast Exponentiation

Method of taking exponents in a fast way, since doing:

$$x * x * x * x \dots * x$$

is super slow.

---

```
FastExt(x,n){
  if n=1 output x
  else
    if n is even output FastExp(x, floor(n/2))^2
    if n is odd output FastExp(x, floor(n/2))^2*x
}
```

---

So our recurrence looks like:

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

(since we're halving the problem, and doing some constant work at each step)

This is Case 2 of the Master Theorem, so the runtime is  $O(\log_2 n)$

## 6 The Median Problem

### 6.1 The Selection Problem

Want to find the  $k$ th smallest number in a set  $S$ .

Select( $S, k$ )

If  $|S| = 1$  then output  $x_1$ .

Else:

Set  $S_L$  = all numbers less than  $x_1$

Set  $S_R$  = all numbers greater than  $x_1$

If  $|S_L| = k - 1$  then output  $x_1$  (since if you have  $k-1$  things smaller than  $x_1$ , that can only mean  $x_1$  is the  $k$ th smallest element)

If  $|S_L| > k - 1$  then output Select( $S_L, k$ ) (since that means the  $k$ th smallest element must be within that set)

If  $|S_L| < k - 1$  then output Select( $S_R, k-1-|S_L|$ )(-1 since you know its not  $x_1$ , and -  $|S_L|$  since you know its not in any of those, so you want the  $k-1-|S_L|$ -th element of  $S_R$ .)

The runtime of this algorithm is almost entirely dependent on the choices of pivots, since if you get a "bad" pivot every time, then you would recurse on a set of size  $n-1$ .

$$T(n) = (n - 1) + T(n - 1)$$

.

.

.

$$= \frac{1}{2}(n(n + 1))$$

which is  $O(n^2)$ .

We could instead choose our pivot randomly.

The pivot would separate the list into sizes from  $\frac{n}{4}$  to  $\frac{3n}{4}$  with probability  $\frac{1}{2}$ , and so the pivot would be good half the time. So the expected running time is:

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

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

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

which satisfies Case 1 of the master theorem which is  $O(n)$ .

But what if we want to be **certain** that the worst case will never happen?

## 6.2 Median of Medians

Divide the set  $S$  into groups of size 5. Sort each group and find the median of each group. If you were to find the median of these medians, there would always be less than  $\frac{7}{10}n$  elements in your two groups, which is pretty good. The reason this comes up is:

There's  $\frac{n}{5}$  groups overall. Imagine the everything was sorted. Each group of 5 is sorted, and the groups are sorted by their medians. So there's  $\leq \frac{n}{5} * \frac{1}{2} = \frac{n}{10}$  groups to the left of the median of medians. There's 3 elements above than the median in its own group, so there's  $\leq \frac{3n}{10}$  elements smaller (to the top left) than the median, which means there's  $\leq \frac{7n}{10}$  elements larger (to the bottom right) than the median.

So the max size of the sets is  $\frac{7n}{10}$

Finding the median of the medians is done recursively, by partitioning into 5 groups, and putting a recursive call on finding the pivot.

So the recursive formula is:

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

Notice the Master Theorem doesn't apply here, instead we need to use the recursion tree method.

First our problem of size  $n$  is broken into two problems, one of size  $\frac{7n}{10}$  and the other size  $\frac{2n}{10}$ . Continuing down recursively, we actually get one side of

the tree ending before the other. Namely, the  $\frac{7n}{10}$  side will reach the leaves later than the  $\frac{3n}{10}$  side.

However, up until the point that this end is reached, we're doing  $(\frac{9}{10})^l n$  work at each level. Beyond this, the work needed at each level only decreases, so it's  $\leq (\frac{9}{10})^l n$ . These terms are geometrically decreasing, so the first term dominates, and we get  $O(n)$ .

## 7 Finding the Closest Pair of Points in the Plane

How fast can we solve this?

### 7.1 Exhaustive Search

Calculate the distance between every pair of points, choose the shortest pairwise distance.  $O(n^2)$ . Is there a faster algorithm?

In one-Dimension, notice that the closest pair of points needs to be next to each other on the line. So we only need to find how far each **pair** is. ( $n - 1$  distances to calculate).

### 7.2 2-D case

Simply taking the closest in their x-coordinate (or y coordinate) doesn't work since they could be close in x but very far in y.

A divide + conquer approach is to separate the points into two groups of size  $\frac{n}{2}$ , so we want our dividing line to pass through the median x-coordinate.

We can now recursively search for the closest pairs in each group.

But what if the closest pair is **between** the two groups?

So we have to check to see if there's a better solution with an endpoint in each group. How can we do this efficiently? (This is the bottleneck step).

### 7.3 Widening the Bottleneck

Notice that by solving the subproblems recursively we can find the smallest distance between two points in both the left and right subproblems call this  $\delta$ . So we know that if a better solution exists, it will be within  $\delta$  from the dividing line.

This seems much better! But what if all the points are within  $\delta$  of the dividing line? Well then this doesn't help much.

There's actually a trick we can do.

We can break up the area into squares of size  $\frac{\delta}{2}$ , and no two points will lie in the same square. This is because if two points are in the same square, then there are on the same side of the dividing line. These points are within  $\delta \frac{\sqrt{2}}{2}$  (by construction of the boxes) from each other, but this is  $< \delta$ , so this contradicts the minimality of  $\delta$ .

We can now use this fact to derive another fact:

Suppose there's a point on either side of the dividing line with distance less than  $\delta$ . We can prove that there will be at most 10 points between them in the y-ordering. (Within the area filled with boxes).

#### Proof

Since the squares are of size  $\frac{\delta}{2}$ , then the two points are either on the same row, or one is within two rows above the other. (or else it would be further than  $\delta$ ) Now, since there can only be one point per-box, there's at most 10 points between them. (count the boxes for yourself!)

Now recall the 1-D case, we can now just look at every pairwise distance on a group where the points are at most 11 apart (rather than the ones that are next to each other as before). So you need to find the distance between a given point, and the next 11 distances.

So at most  $11n$  distances to calculate.

## 7.4 The Finished Algorithm

- Find the point with the median x-coordinate
- Partition using this point
- Recursively find the closest pair of points in each half
- Find the closest pair within the small range given by  $\delta$ , by checking the nearest 11 points (in the y-ordering) for each point.
- Among the three pairs found, (left, right, crossing) output the closest pair.

## 7.5 The Runtime (Enhanced)

Two subproblems of size  $\frac{n}{2}$ , and the work at each level is: finding the median  $O(n)$ , partitioning  $O(n)$ , making the smaller group (within  $\delta$  of dividing line)  $O(n)$ , applying the 1-D algorithm  $O(n)$ . So our recurrence looks like:

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

which is case 2 of the master theorem, so  $n \log(n)$ .

## Part II

# Graph Algorithms

For a review of basic graph terminology, see my COMP250 study guide. And I'm going to assume you took MATH240 and know the basics of graph theory from there. Lecture 7 had a review of these basics, but I won't include them here. I will only go over the theorems that were proved.

## 8 Theorems About Undirected Graphs

### 8.1 Handshaking Lemma

**Theorem 4.** In an undirected graph, there are an even number of vertices with odd degree.

*Proof.* We start with:

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

Since each edge is double counted when summing the degrees. (Each edge  $(u, v)$  contributes 1 degree to  $u$  and 1 degree to  $v$ )

This is the same as:

$$= \sum_{v \in \text{Odd}} \deg(v) + \sum_{v \in \text{Even}} \deg(v)$$

(the sum of the vertices of even degree plus the odd degree ones)

Rearranging we get:

$$\sum_{v \in \text{Odd}} \deg(v) = 2 \mid |E| - \sum_{v \in \text{Even}} \deg(v)$$

Now we know at the sum of the even degrees is even, and we know that  $2x$  is even for any  $x$ . So the right hand side is always even. Therefore the left hand side must be even. But for the sum of odd numbers to be even, there must be an even number of odd terms.

□

### 8.2 Leaf Existence

**Theorem 5.** Lemma: A tree  $T$  with  $n \geq 2$  vertices has at least one leaf.

*Proof.* A tree is connected, which means there's no vertices with degree 0. A leaf is a vertex with degree 1, so to get a contradiction, assume every vertex has degree  $\geq 2$ .

Take the longest path  $P = v_1, v_2, v_3 \dots v_{l-1}, v_l$

But every vertex has degree greater than 2, so  $v_l$  has a neighbour  $x \neq v_{l-1}$  so  $v_l$  forms an edge with something in the path which would create a cycle and thus be a contradiction (since all trees have no cycles). If the neighbour was not on the path, then  $P$  was not the longest path.  $\square$

### 8.3 Number of edges in a Tree

**Theorem 6.** A tree with  $n$  vertices has  $n - 1$  edges

*Proof.* By induction:

Base Case:

A tree on one vertex has zero edges

linebreak Induction Step:

Assume that any tree on  $n - 1$  vertices has  $n - 2$  edges. Take a tree with  $n \geq 2$  vertices. By the previous lemma, there exists a leaf vertex  $v$ . Let  $T' = T - v$ . Then  $T'$  is a tree on  $n - 1$  vertices, which has  $n - 2$  edges by the induction hypothesis. Adding back  $v$ , we get that  $T$  is a tree on  $n - 1$  vertices with  $n - 2$  vertices.  $\square$

### 8.4 Halls Theorem

**Theorem 7.** A bipartite graph, with  $|X| = |Y|$  contains a perfect matching  $\Leftrightarrow \forall S \subseteq X, |\Gamma(S)| \geq |S|$

*Proof.*  $(\Rightarrow)$

If there is a set  $S \subseteq X$  with  $|\Gamma(S)| < |S|$ , then the graph cannot have a perfect matching, since there would not be enough things in the neighbourhood for the things in  $S$  to match to.

$(\Leftarrow)$

Take a maximum cardinality matching  $M$  in the graph. If  $M$  is perfect we're done, if not, then there exists a vertex  $x_0$  who is not matched in  $X$ . If Halls condition holds, then  $x_0$  has a neighbour  $y_0$ . Suppose  $y_0$  is matched to  $x_1$ . Again if Halls condition holds, then  $x_0, x_1$  have another neighbour say  $y_1$ .

We now repeat this process until eventually it terminates (it will since we



have a finite number of vertices). It will terminate when we reach  $x_k$  who is unmatched.

We now create an  $m$ -alternating path from  $y_k$  to  $x_0$ . This path is  $m$ -augmenting, so we augment, and receive a larger matching. Contradiction,  $M$  was not maximal.

□

## 9 Breadth First Search

### 9.1 Generic Search Algorithm

---

```
Put root into bag
while bag not-empty
  remove v from the bag
  if v is unmarked
    mark v
    for each arc(v,w)
      put w into the bag
```

---

A vertex is discovered when it is marked. Notice that there can be multiple copies of a vertex in the bag, and that this actually won't affect our performance.

#### 9.1.1 Revised Generic Search Algorithm

Instead of adding vertices, we'll add arcs.

---

```
Put (*,r) into a bag
while bag not-empty
  remove (u,v) from the bag
  if v is unmarked
    mark v
    set p(v) to u //keep track of "predecessor" of v
    for each arc(v,w)
      put (v,w) into the bag
```

---

Keeping track of the predecessor will be useful later.

### 9.1.2 The Running Time

We look at each arc out of  $v$  only once, when  $v$  is first marked. The arc is then added to the bag once, and removed once. So, we get a runtime proportional to the number of arcs.

$$\Rightarrow O(m)$$

### 9.1.3 Validity

**Theorem 8.** Let  $G$  be a connected, undirected graph. Then the search algorithm finds every vertex in  $G$ .

*Proof.* We need to show that every vertex  $v$  is marked by the algorithm. We will use induction on the length of the smallest path from the vertex to the root.

Base Case:

$k = 0$  then  $v$  is the root, and only the root exists, so trivially true.

Induction Step:

Assume true for a path of length  $k - 1$  from the root. Now assume there is a path  $P$  with  $k$  edges from  $v$  to  $r$ . So let

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

Then there is a path:

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

So by the induction hypothesis,  $u$  is marked. Then after we mark  $u$ , all edges incident to  $u$  would have been added, so  $(u, v)$  would have been added. And so later,  $(u, v)$  would be removed and  $v$  would be marked.  $\square$

We can prove that for directed graphs, every vertex that has a directed path from  $r$  is marked in the same way.

## 9.2 Search Trees

**Theorem 9.** The predecessor edges made by the search algorithm on a connected, undirected graph  $G$  is a tree rooted at  $r$ .

*Proof.* By induction on the number of marked vertices,  $k$ .

Base Case:  $k = 1$

Induction Step:

Assume true for the first  $k - 1$  vertices. Let  $v$  be the  $k$ th vertex to be marked. Assume  $v$  was marked when we removed the edge  $(u, v)$ . This means that  $u$  is the predecessor of  $v$ . But  $(u, v)$  was added to the bag when we marked  $u$ , so  $u$  must be in the set of the first  $k - 1$  vertices to be marked. Thus, by the induction hypothesis, when we add the edge  $(p(v), v) = (u, v)$ , we are adding a leaf, so the new graph formed is still a tree.  $\square$

## 9.3 Choices of Bags

We can use a Queue to get BFS, if we use a Stack we get DFS, if we use a Priority Queue, we get minimum spanning tree.

## 9.4 BFS Trees

The edges are added to the queue in order of their distance from  $r$ . The vertices are marked in order of their distance from  $r$ .

**Theorem 10.** For any vertex  $v$ , the path from  $v$  to  $r$  given by the search tree  $T$  of predecessor edges is a shortest path.

*Proof.* Left as exercise.  $\square$

### 9.4.1 Structure

The structure of these trees can be broken down into "layers", where each layer is the set of vertices at a given distance from the root.

Any vertex  $v \in S_l$  is at distance  $l$  from  $r$  in  $T$ , and the same is true in the whole graph  $G$ .

This implies that for every edge in the graph that is not in the tree  $(u, v)$ ,  $u$  and  $v$  are either in the same layer or in adjacent layers. If this was not the case, say  $u$  was in  $S_3$  and  $v$  was in  $S_6$ , then we could get from the root to  $v$  in less than 6 steps.

### 9.4.2 BFS on Bipartite Graphs

**Theorem 11.** A graph  $G$  is bipartite  $\Leftrightarrow$  it contains no odd length cycles.

*Proof.*  $\Rightarrow$

Assume  $G$  contains an odd length cycle  $C$ .

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

Without loss of generality we can assume  $v_0 \in Y$ , therefore  $v_1 \in X$ , and so on. We eventually get down to  $v_{2k} \in X$  but it is a cycle so  $v_{2k}$  forms an edge with  $v_0$ , and we said  $v_0 \in Y$ , which means  $v_{2k} \in Y$ , it can't be both in  $X$  and in  $Y$ , contradiction.

$\Leftarrow$

Assume  $G$  has no odd length cycles. Choose a root vertex  $r$ , and run BFS. Let  $X$  be the set of all odd layers of the BFS tree. Let  $Y$  be the set of all even layers of the BFS tree. Since every edge in the graph goes to either adjacent layers or the same layer, we know that if we have no edges in the same layer, then we'll have that every edge goes from  $X$  to  $Y$ .

Assume there's 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. Let  $P$  be the path from  $u$  to  $z$  in the tree, and let  $Q$  be the path from  $v$  to  $z$  in the tree. The length of  $P$  is the same as  $Q$  since  $u$  and  $v$  are in the same layer. But then the cycle

$$C = P \cup Q \cup (u, v)$$

has an odd number of edges. So  $(u, v)$  cannot exist.  $\square$

## 10 Depth First Search

We use the generic search algorithm using a stack.

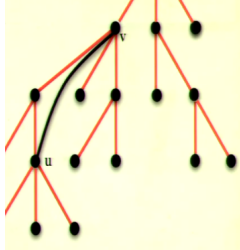
### 10.1 DFS Trees

The DFS tree is much different than the BFS tree. DFS partitions the edges of an undirected graph into two types:

Tree Edges: Predecessor edges in the DFS tree at  $T$

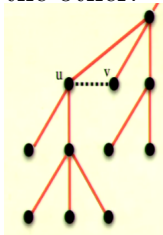
Back Edges: Edges where one endpoint is the ancestor of the other endpoint

in  $T$ .



Here  $(u,v)$  is a back edge.

We cannot have Cross Edges: Where neither endpoint is an ancestor of the other.



## 10.2 Recursive DFS

We can also do DFS recursively:

---

```
RecursiveDFS(r)
  mark r
  for each edge (r,v)
    if v is unmarked
      set p(v) = r
      RecursiveDFS(v)
```

---

## 10.3 Ancestral Edges

**Theorem 12.** Let  $T$  be a DFS tree in an undirected graph  $G$ . Then for every edge  $(u,v)$  either  $u$  is an ancestor of  $v$  in  $T$  or  $v$  is an ancestor of  $u$ .

*Proof.* Wlog assume  $u$  is marked before  $v$ . At the time  $u$  is marked, the algorithm will recurse on each arc incident to  $u$ .

Case 1:  $v$  is unmarked when the  $\text{RecursiveDFS}(u)$  examines  $(u, v)$ .  
 Then the parent of  $v$  is then  $u$ , and so  $(u, v)$  is an ancestral tree edge.  
 Case 2:  $v$  is already marked. But  $v$  was marked after  $u$ , so it was marked during  $\text{RecursiveDFS}(u)$ . So we have a series of vertices

$$\{u = w_0, w_1 \dots w_{l-1}, w_l = v\}$$

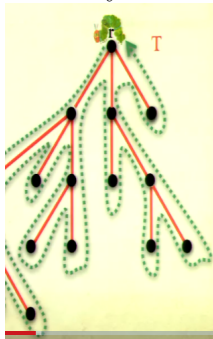
where  $p(w_k) = w_{k-1}$  (the parent of each vertex is the previous vertex). This means that  $u$  is an ancestor of  $v$ , so  $(u, v)$  is a back edge.

□

**Corollary** Every non-tree edge is a back edge.

## 10.4 Previsit and Postvisit

The way DFS explores the vertices of a graph is given by this picture:



We can add a "clock" that will keep track of the order in which the vertices were visited.

$Pre(v)$  is the time at which we arrive at a subtree rooted at  $v$ .  $Post(v)$  is the time at which we leave a subtree rooted at  $v$ .

So we can represent each vertex by an interval of time. If we take the interval for every vertex, we get what's called a **Laminar Family**. Meaning, every interval is either completely disjoint, or completely overlapping.



If we draw an edge between each interval and the smallest interval that contains it, we actually build up the DFS tree again!

## 10.5 Directed Graph BFS Tree Structure

Now we can have four types of edges:

Tree arcs: Same as before

Forward Arcs: Arcs  $(u, v)$  where  $u$  is an ancestor of  $v$

Backward Arcs: Arcs  $(u, v)$  where  $v$  is an ancestor of  $u$

Cross Arcs: Non-Ancestral arcs  $(u, v)$  where  $u$  is marked after  $v$ . Note that the other way around is not possible because after visiting  $v$ , we must visit all descendants of it, before moving back up the tree and going down the other branch containing  $u$ .

We still have intervals. In a tree arc  $(u, v)$ , the interval of  $v$  is contained in the interval of  $u$ . In a forward arc, the same is true. In a backward arc however, the interval of  $u$  is contained in the interval of  $v$ . In cross arcs, the intervals of  $u$  and  $v$  are disjoint.

So we have this list of properties:

For tree arcs:

$$post(v) < post(u)$$

For forward arcs:

$$post(v) < post(u)$$

For backward Arcs:

$$post(u) < post(v)$$

For cross arcs:

$$post(u) < post(v)$$

So the only different one is for backward arcs.

## 10.6 Example: Directed Acyclic Graphs

How can we determine if a graph is acyclic?

**Theorem 13.** A directed graph  $G$  is acyclic  $\Leftrightarrow$  DFS produces no backward arcs.

*Proof.*  $\Rightarrow$

Suppose DFS gives a backward arc  $(u, v)$ . By definition, then  $u$  is a descendant of  $v$  in the DFS tree  $T$ . Then there exists a path:

$$P = \{v = v_0, v_1, \dots, v_k = u\}$$

which means  $P \cup (u, v)$  is a directed cycle in  $G$ .

$\Leftarrow$

Assume DFS gives no backward arcs. Suppose there's a directed cycle:

$$C = \{v_0, v_1, \dots, v_k, v_0\}$$

Since there's no backward arcs we have that:

$$post(v_0) > post(v_1) > \dots > post(v_k) > post(v_0)$$

But  $post(v_0)$  can't be greater than itself.

□

**Corollary** There is a linear time algorithm to test whether or not a graph is acyclic. Just run DFS and check if any arc is a backward arc.

## 10.7 Example: Topological Ordering

A topological ordering is when the vertices of a graph can be horizontally ordered such that every arc is from right to left.

**Theorem 14.** A directed graph  $G$  has a topological ordering  $\Leftrightarrow$  DFS produces no backward arcs.

*Proof.*  $\Rightarrow$  If DFS produces a backward arc then  $G$  contains a cycle  $C$ . Let the cycle:

$$C = \{v_0, v_1, \dots, v_k, v_0\}$$

where, wlog,  $v_0$  is the leftmost vertex of the cycle in the order. But then  $v_0, v_1$  goes from left to right, which is not allowed.

$\Leftarrow$

Assume DFS gives no backward arcs. Then for every arc  $(u, v)$  we have:

$$post(u) > post(v)$$

so simply order the vertices by their post numbers.

□



## Part III

# Greedy Algorithms

## 11 Scheduling

### 11.1 Task Scheduling

A firm can process 1 task a time. The job of customer  $i$  takes  $t_i$  time. We want to minimize the sum of the waiting times. Any job cannot be started until the previous one is finished.

First, we sort the jobs by length, shortest to longest. Simply schedule them in that order.

We must now prove this works.

**Theorem 15.** The greedy algorithm outputs an optimal schedule.

*Proof.* We will use an exchange argument.

Let the greedy algorithm schedule in the order:  $\{1, 2, \dots, n\}$

Assume there's a better schedule  $S$ . Then there must be a pair of jobs  $i$  and  $j$  such that:

Job  $i$  is scheduled immediately before job  $j$  by schedule  $S$

Job  $i$  is longer than job  $j$

If we don't have this property, then it's sorted. The waiting time of job  $i$  is currently when job  $i$  finishes, and the waiting time of job  $j$  is when job  $j$  finishes.

Swap jobs  $i$  and  $j$ . Everything else stays the same. Specifically, the waiting time of every unchanged job stays the same.

Now the new waiting time of job  $j$  is better than both of the old ones, and the waiting time of job  $i$  is the same as the waiting time of the old job  $j$ .



So this configuration is better, and that contradicts the assumption that  $S$  was an optimal schedule.  $\square$

### 11.1.1 Running Time

All we did was sort, so it's  $O(n \log(n))$

## 11.2 Class Scheduling

There is one classroom. There's a set  $I = \{1, 2, \dots, n\}$  of classes that want to use the room class  $i$  has a start time  $s_i$  and a finish time  $f_i$ . The goal is to book as many classes as possible.

This is also known as the interval selection problem.

### 11.2.1 First Start

Select the class that starts earliest, iterate on the remaining classes that do not conflict with this one.

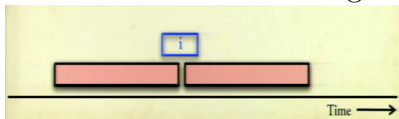
This doesn't work because the first class might also be the longest.



### 11.2.2 Shortest-Duration

Select the shortest class first, then iterate.

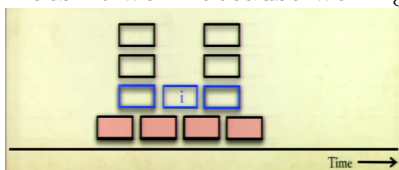
Doesn't work because might just be in an unlucky position.



### 11.2.3 Minimum Conflict

Select the class that conflicts with the fewest number of classes first. Then iterate.

Doesn't work because we might get a configuration like this:



Here the optimal solution has 4 classes, but we chose the configuration that has only 3, we chose  $i$  since  $i$  conflicts with only 2, whereas the red ones all conflict with 3 or 4. Next time we iterate, we just have two stacks of 3, so we can choose any from the left or right, and we're stuck with having at most 3 classes in our solution.

### 11.2.4 Last Start

Select the class that starts last, and iterate on the classes that do not conflict with this selection.

This one works!

It is symmetric to selecting the class that finishes first, then iterating on the classes that don't conflict with this selection.

Here's the pseudocode:

---

**FirstFinish(I)**

Let class1 be the class with the earliest finish time

Let X be the set of classes that clash with class1

output  $\{1\} \cup \text{FirstFinish}(I \setminus X)$

---

**Lemma** There is some optimal solution that selects Class 1.

**Proof**

Recall the classes are indexed such that  $f_1 \leq f_2 \leq \dots \leq f_n$ .

Take an optimal schedule  $S$  and assume Class 1 is not in it. Let  $i$  be the lowest index class in  $S$ .

We claim we can replace  $i$  with 1. We know that  $f_i$  is before  $s_j$  for any  $j \in S$ . But  $f_1$  is before  $s_j$ , so we know Class 1 doesn't conflict with any class in  $S - \{i\}$ .

So this ordering is at least as good as before, and contains class 1.

**Theorem 16.** The first-finish algorithm outputs an optimal schedule.

*Proof.* By induction on the cardinality of the optimal solution  $|opt(I)|$

Base Case:

Let the solution have size 1, then this is trivially the optimal solution.

Induction Step:

Assume true for size  $k$ . Let the optimal solution have size  $k + 1$ . First finish outputs  $\{1\} \cup FirstFinish(I - X)$ . But by the lemma, 1 is part of some optimal solution,  $S^*$ .

This means  $S^* - \{1\}$  is an optimal solution with size  $k$ . So by the induction hypothesis, we have an optimal solution.  $\square$

There are at most  $n$  iterations of the algorithm, and it takes  $n$  time to find the class that finishes earliest in each iteration. So  $O(n^2)$ , but you could get it down to  $O(n \log n)$  if you're careful about implementation.

## 12 The Shortest Path Problem

If every arc  $a$  has a length associated to it (a weight),  $l_a$ , then the length of a path  $P$  is:

$$l(P) = \sum_{a \in P} l_a$$

How do we then find the shortest path from  $s$  to every other vertex in the graph.

It turns out, we can find all the shortest paths in one go!

### 12.1 Dijkstra's Shortest Path Algorithm

Initially set the distance from the first vertex  $s$  to itself to be 0, and set every other vertex to have distance  $\infty$ .

Now take the vertex  $v$  with the smallest distance label. Since we just have 0 and the rest are  $\infty$ , it's obvious to choose  $s$ . Now look at every arc coming out of this vertex.

For each arc  $(s, v)$  coming out of the  $s$ , check to see if the distance from  $s$  to  $v$  is less than what  $v$  is marked as.  $v$  is currently marked as infinity, so yes it is. We then update the predecessor arc of  $v$  to be  $(s, v)$ .

Now go back and find the vertex with the smallest distance, add it to the set of vertices we're done with, and repeat from here.

This algorithm is honestly super confusing and you should probably look at videos and try lots of examples on paper.

#### 12.1.1 Special Case, All Arcs Have Distance 1

In this case, we actually get exactly Breadth First Search! Try it out ;).

#### 12.1.2 Shortest Path Graph

Let  $S^k$  be the set of vertices in  $S$  at the end of the  $k$ th iteration, where  $S$  is the set of vertices we're "done with".

Let  $T^k$  be the set of arcs in  $T$  at the end of the  $k$ th iteration, where  $T$  is the set of arcs we fix to be in our final result.

Notice that all arcs in  $T^k$  are between vertices in  $S^k$  because when we add a vertex to  $S$ , we add the arc between it and its predecessor (another vertex in  $S$ ), to  $T$ . This means that  $G^k = (S^k, T^k)$  is a directed graph.

Finally,  $G^n$  is the final output of the algorithm.

### 12.1.3 Shortest Path Tree

**Theorem 17.** The Graph  $G^k$  is a directed tree rooted at  $s$ .

*Proof.* Base Case:  $k=1$

$S^1$  only contains  $s$ , and  $T^1$  is empty. One vertex is a trivial tree.

Induction Step:

Assume true for  $G^{k-1}$ . Let  $v_k$  be the vertex added to  $S$  at the  $k$ th iteration. So  $S^k = S^{k-1} \cup \{v_k\}$ . This means that  $T^k = T^{k-1} \cup (\text{pred}(v_k), v_k)$ . So  $v_k$  has in-degree 1, and out-degree 0, which means  $v_k$  is a leaf. So  $G^k$  is still a directed tree rooted at  $s$ .  $\square$

**Theorem 18.**  $G^k$  gives the true shortest path distances from  $s$  to every vertex in  $S^k$ .

*Proof.* Base Case:  $k = 1$

Trivially true. The label,  $d(s)$  is 0, which is the shortest path distance,  $d^*(s)$

Induction Step:

Assume true for  $G^{k-1}$ . That is,  $d^{k-1}(v) = d^*(v) \forall v \in S^{k-1}$

Let  $v_k$  be the vertex added to  $S$  in the  $k$ th iteration. Take the shortest path  $P$  from  $s$  to  $v_k$  that uses as many arcs in common to  $G^k$  as possible. (basically this path follows the tree, then eventually jumps out to get to  $v_k$  and we're assuming this is faster than just following the tree).

Let  $x$  be the last vertex of  $G^{k-1}$  in  $P$ . Let  $y \notin S^k$  be the vertex after  $x$  in  $P$ . If this  $y$  doesn't exist, (there's no vertex after  $x$ ) then we're done, since

that means  $P \subseteq G^k$ . Assume it does exist.

Since  $y$  is on the shortest path from  $s$  to  $v_k$  and the arc-lengths are non-negative, each sub-path is also a shortest path. So the path from  $s$  to  $y$  is shorter than the path from  $s$  to  $v_k$ .

Since we're assuming  $P$  is the optimal path:

$$d^*(y) \leq d^*(v_k) < d^k(v_k)$$

But since  $x \in S^{k-1}$  we have:

$$d^k(y) \leq d^{k-1}(x) + l(x, y)$$

(basically meaning that the distance from  $s$  to  $y$  is at most the distance from  $x$  to  $y$ .) And by our induction hypothesis:

$$= d^*(x) + l(x, y)$$

and by our assumption:

$$= d^*(y)$$

So we've now proved:

$$d^k(y) \leq d^*(y) < d^k(v_k)$$

Which contradicts our choice of  $v_k$ , since  $d^k(y) < d^k(v_k)$ . □

#### 12.1.4 The Running Time

There are  $n$  iterations, there are at most  $n$  distance updates at each iteration. So at most  $O(n^2)$  but again we can improve it to  $O(m \log n)$  using a heap.

## 13 Huffman Codes

### 13.1 Data Encoding

Suppose we want to encode the alphabet in binary. How many bits do we need to encode every letter?

Five bits since  $2^5 \geq 26$

How do we measure the quality of an encoding? A natural measure would be the length of the encoding. But what if some letters are used very often? We would want these to have a smaller size.

Let  $f_i$  be the frequency at which a letter  $i$  appears in the alphabet. Then:

$$cost = \sum_{i \in A} l_i f_i$$

### 13.1.1 Morse Code

Morse code follows this idea. It uses less bits for the frequently used letters, and less bits for the less common ones. But there's problem with it. It cannot be binary because it's ambiguous whether 101 means 101 or 1, 0, 1. So Morse code is actually ternary. It uses pauses to signify the end of a letter.

How can we get around this?

## 13.2 Prefix Codes

A coding system is prefix-free if no codeword is a prefix of another codeword. Morse is not prefix free, since in 1101, 1 means t, 11 means m, 110 means g and 1101 means q. So t is a prefix of m is a prefix of g is a prefix of q.

## 13.3 Binary Tree Representations

We can use a binary tree  $T$  to represent a prefix-free binary code. Each left edge has label 0 and each right edge as label 1.

The leaf vertices are the letters of the alphabet. The codeword for a letter are the labels on the path from root to leaf.

**Theorem 19.** A binary coding system is prefix-free  $\Leftrightarrow$  it has a binary tree representation.

*Proof.* ( $\Leftarrow$ )

In a binary tree representation the letters are at the leaves. This means that the path  $P_x$  from the root to leaf  $x$  and the path  $P_y$  from the root to a leaf



$y$  must diverge at some point.

So the codeword for  $x$  cannot be a prefix of the codeword for  $y$ .

( $\Rightarrow$ )

Given a binary coding system, we can define a binary tree recursively. A letter whose code word started with a 0 is placed in the left subtree. Otherwise it is placed in the right subtree. Then just recurse on the next letter.  $\square$

Observe that the cost of the tree is:

$$cost(T) = \sum_{i \in A} f_i d_i(T)$$

where  $d_i$  is the depth of the node  $i$ .

### Proof

We have the definition of cost:

$$cost(T) = \sum_{i \in A} f_i l_i(T)$$

The length of the word is just the sum of the edges in the word

$$= \sum_{i \in A} \sum_{e \in P_i} 1$$

Which is exactly the same as the depth in the tree.

$$= \sum_{i \in A} f_i d_i(T)$$

### 13.3.1 Letter to Leaf Assignment

How should we assign letter to leaves? The least frequent letters should be at the deepest leaf. So, we can just sort all the frequencies, and start adding each least frequent letter to the deepest leaf.

### 13.3.2 Tree Shape

But what should the shape of the tree be?

Let  $n_e = \sum_{i \in A: e \in P_i} f_i$  be the number of letters (weighted by frequency) whose root-leaf paths use edge  $e$  in  $T$ . (How many letters use this edge)

$$\text{cost}(T) = \sum_{e \in T} n_e$$

#### Proof

We start with our first observation:

$$\text{cost}(T) = \sum_{i \in A} f_i d_i(T)$$

The depth is just the sum of the edges in the path from root to the vertex.

$$= \sum_{i \in A} f_i \sum_{e \in P_i} 1$$

Changing the order of summation:

$$= \sum_{e \in T} \sum_{i \in A: e \in P_i} f_i$$

Which is exactly our definition.

$$= \sum_{e \in T} n_e$$

### 13.4 The Key Formula

The key to designing a good coding system is the following formula:

**Theorem 20.** Let  $\hat{T}$  be the tree formed from  $T$  by removing a pair of sibling-leaves  $a$  and  $b$  and labelling its parent by  $z$  where  $f_z = f_a + f_b$  then:

$$\text{cost}(T) = \text{cost}(\hat{T}) + f_a + f_b$$

Observation 1 is telling us that the least frequent letters should be siblings, and observation 2 tells us how to find the optimal shape of the tree.

## 13.5 The Algorithm

---

```
Huffman(A,f)
  if A has two letters then
    encode one letter with 0 and the other with 1
  else
    let a and b be the most infrequent letters
    merge a and b into a new node z with frequency  $z = a + b$ 
    recurse on the new set
    create the tree by adding a and b as children of z in the
      completed tree
```

---

### 13.5.1 Proof Of Correctness

**Theorem 21.** The Huffman Coding Algorithm gives the minimum cost encoding.

*Proof.* By Induction on the size of  $A$ .

Base Case:  $|A| = 2$

Each letter has codeword length 1.

Induction Step:

Assume works for  $|A| = k$ . Take  $|A| = k + 1$ . Let  $a$  and  $b$  be the least frequent letters. Then  $a$  and  $b$  are siblings in the optimal solution, and for any  $\hat{T}$ :

$$\text{cost}(T) = \text{cost}(\hat{T}) + f_a + f_b$$

so the best choice of  $\hat{T}$  is the optimal solution for  $\hat{A}$ , but by the induction hypothesis, this is what we did in the last step of the algorithm.  $\square$

### 13.5.2 Running Time

There are  $n-2$  iterations, each iteration takes  $O(n)$  to find the two least frequent letters and update the alphabet. So,  $O(n^2)$ . Again, with heaps we can get it to  $O(n \log(n))$ .

## 14 Minimum Spanning Tree Problem

Given a graph, each edge  $e$  has a cost  $c_e$ , where all edge costs are distinct.

So the cost of a tree  $T$  is:

$$c(T) = \sum_{e \in T} c_e$$

### 14.1 Kruskal's Algorithm

Sort the edges  $\{e_1, e_2, \dots, e_m\}$  by cost, least to greatest.

Set  $T = \phi$

For each  $i = \{1, 2, \dots, m\}$

Let  $e_i = (u, v)$

if  $u$  and  $v$  are in different components of the tree, then add this edge to  $T$ .

### 14.2 Prim's Algorithm

Set  $T = \{a\}$

If  $V(T) \neq V(G)$  then

Let  $e$  be the minimum cost edge in  $\delta(T)$  (edges leaving a vertex in  $T$ )

Add this edge to  $T$ . (and its vertices)

### 14.3 Boruvka's Algorithm

Set  $T = \phi$

If  $T$  has more than one component  $\{S_1, S_2, \dots, S_l\}$  then

For  $i = \{1, 2, \dots, l\}$  let  $e_i$  be the minimum cost edge in  $\delta(S_i)$

Add all of these edges to the tree.

### 14.4 Running Times

#### 14.4.1 Kruskal's Running Time

It takes  $O(m \log m)$  to sort the edges, and there's  $m$  iterations of the loop. Within the loop we have to search the tree to see if  $u$  and  $v$  are in different components. This takes time  $O(n)$ .

So we have:

$$O(m \log m + mn) = O(mn)$$

#### 14.4.2 Prim's Running Time

We have  $n$  iterations of the loop. Within the loop, we have to exhaustively search for the minimum edge in time  $O(m)$ .

So:

$$O(mn)$$

#### 14.4.3 Boruvka's Running Time

We have at most  $n$  components, finding the minimum edge takes  $O(m)$ , and there are  $\leq \log n$  iterations.

So:

$$O(mn \log n)$$

### 14.5 Proof That They All Work

First, notice that for a chicken to cross a road and get to a chicken coop, it must cross the road an odd number of times.

We'll also need this fact:

**Theorem 22.** The Cut Property of a minimum spanning tree is this: Assume the edge costs are distinct. If  $e$  is the cheapest edge in some cut  $\delta(S)$  then  $e$  is in the minimum spanning tree.

*Proof.* Let  $e = (u, v)$  be the cheapest edge in a cut  $\delta(S)$  recall that  $\delta(S)$  is the edges leaving a subset of vertices  $S$ .

Let  $T^*$  be a minimum spanning tree, and to get a contradiction, assume  $e \notin T^*$ .

Since  $T^*$  is a spanning tree there is a unique path  $P$  in  $T^*$  from  $u$  to  $v$ .

Observation. If  $\hat{e} \in P$  then  $(T^* \setminus \hat{e}) \cup e$  is a spanning tree. (Basically we can replace an edge from the path joining  $u, v$  with  $u, v$  itself).

If a chicken is walking along  $P$  must cross the cut  $\delta(S)$ .

Observation. There is at least one edge  $\hat{e} \in P \cap \delta(S)$ .

We know from the beginning that  $e$  is the lowest cost edge, so

$$\begin{aligned} c_{\hat{e}} &> c_e \\ \Rightarrow (T^* - \hat{e}) \cup e \end{aligned}$$

is a cheaper spanning tree than  $T^*$ . This contradicts the assumption that  $T^*$  was the minimum cost tree.

□

This proves all our algorithms simultaneously.

In the case of Prim's algorithm, we literally added edges based on if they were the minimum edge in the cut  $\delta(T)$ , so it directly uses this theorem.

In Baruvka's algorithm, we add edges from the cut  $\delta(S)$  for each component, so again using the theorem.

In Kruskal's algorithm, we add edges if  $u, v$  are in different components. Let  $S$  be one of the two. Then  $e_i$  is the cheapest edge in  $\delta(S)$  since we look at edges by order of cost. So this one also works.

## 14.6 The Cycle Property

**Theorem 23.** Assume distinct edge costs. If  $e$  is the most expensive edge in some cycle  $C$ , the  $e$  is not in the MST.

*Proof.* Let  $e = (u, v)$  be the most expensive edge in the cycle  $C$ .

So  $P = C - e$  is a path from  $u$  to  $v$ .

Assume for a contradiction that  $e$  is in the MST  $T^*$ .

Let  $(S, V - S)$  be the cut introduced by  $T^* - e$ .

Observation 1: If  $\hat{e} \in \delta(S)$  then  $(T^* - e) \cup \hat{e}$  is a spanning tree. (basically we can replace  $e$  with  $\hat{e}$  and get a spanning tree, since  $\hat{e}$  also joins the two sets.)

Observation 2: There is at least one edge  $\hat{e} \in P \cap \delta(S)$  (there's an edge that crosses the cut that is also part of the path.)

But  $c_{\hat{e}} < c_e$  so we can replace  $e$  by  $\hat{e}$  which contradicts the fact that  $T^*$  was the MST.  $\square$

## 14.7 The Reverse Delete Algorithm

Sort the edges by cost, most expensive to cheapest.

For each edge:

If  $G \setminus \{e_i\}$  is connected then set  $G = G \setminus \{e_i\}$

So basically take the most expensive edge, and if the graph is still connected without it, then throw it away.

### 14.7.1 Runtime of Reverse Delete

There are  $m$  iterations, and at each one need to check that graph is still connected (using BFS or DFS) in  $O(m)$ . So the running time is  $O(m^2)$

### 14.7.2 Proof of Reverse Delete

First notice that  $G \setminus \{e_i\}$  being connected means there was a cycle including  $e_i$ , and since we've sorted in reverse order,  $e_i$  is the most expensive edge, so by the cycle property, this algorithm works.

## 15 The Clustering Problem

Given a collection of objects,  $O$  we want to partition the objects into a set of clusters  $\{S_1, \dots, S_k\}$ . A "good" clustering has similar objects in the same

clusters.

We represent the problem by a weighted graph  $G$ .

There is a vertex for each object  $O$ , and an edge between each pair of objects.

The weight  $d_{ij} \geq 0$  of an edge represents the dissimilarity of object  $i$  and object  $j$ .

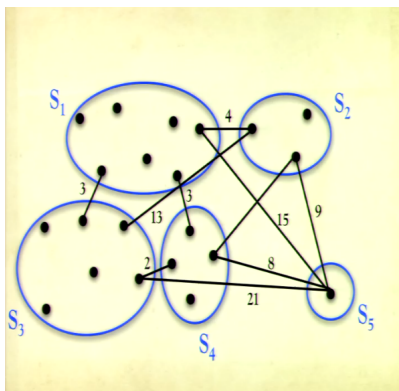
The quality of a clustering has no optimal definition. (Depends on application)

## 15.1 Maximum Spacing Clustering

Maximize the distances between the clusters. In other words, partition the vertices into  $k$  clusters so that the minimum distance between two vertices in different clusters is maximized.

Given a clustering  $\{S_1, S_2, \dots, S_k\}$  we define the distance between two clusters as:

$$d(S_l, S_m) = \min_{i \in S_l, j \in S_m} d_{ij}$$



So here we just want to maximize the minimum black line (here it's 2) so the quality is (2).



## 15.2 Reverse-Delete Clustering Algorithm

Sort the edges by cost, highest to lowest.

For each edge:

If  $G \setminus \{e_i\}$  has  $k$  components or less, then set  $G = G \setminus \{e_i\}$

Notice, this is exactly the MST problem, where in MST,  $k = 1$

### 15.2.1 Proof Of Reverse-Delete Clustering

First, observe this:

**Theorem 24.** A connected graph contains a spanning tree as a subgraph

*Proof.* Simply grow a BFS tree from any root vertex.  $\square$

Next, observe this fact:

**Theorem 25.** We can remove an edge, and the number of components increases by at most 1.

*Proof.* Originally,  $u, v$  are in the same component  $S_1$ .  $S_1$  contains a spanning tree  $T$ .

Case 1:  $e$  is not in  $T$ . Then  $S_1$  remains a component after deletion of  $e$

Case 2:  $e$  is in  $T$  for every spanning tree in  $S_1$ . Then  $S_1$  is broken into two components on the deletion of  $e$ .  $\square$

Now our algorithm:

*Proof.* Let  $e_l$  be the edge whose deletion causes the number of components to increase from  $k - 1$  to  $k$ .

This means that the algorithm deleted all the edges up to  $e_l$ .

When we delete  $e_l$  we have the clustering  $S = \{S_1, \dots, S_k\}$

But this means that only the edges up to  $e_l$  can cross between the clusters. Since we organized these to be largest to smallest, that means  $e_l$  is the

shortest edge between two clusters. So the quality is determined by  $e_l$ .

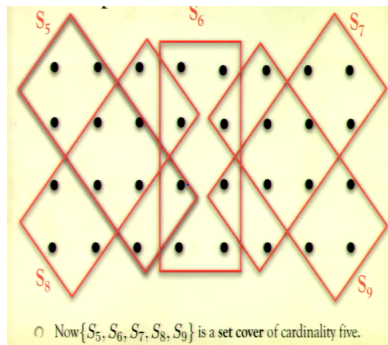
We now need to show that this is the optimal solution.

Any other clustering  $S^* = \{S_1^*, \dots, S_k^*\}$  with  $k$  components must separate the endpoints of at least one edge with an endpoint in the edges up to  $e_l$  from below. (edges smaller than  $e_l$ ).

But then we'll have separated two clusters by an edge shorter than  $e_l$  which is worse than  $S$ .  $\square$

## 16 The Set Cover Problem

Given a collection of  $n$  items,  $I$ , and another collection of sets  $S = \{S_1, \dots, S_m\}$  we want to find the smallest collection of sets in  $S$  that contain all of the elements of  $I$ .

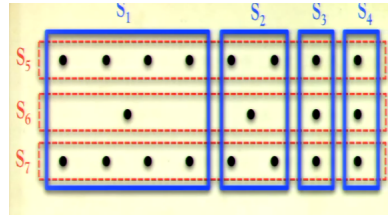


### 16.1 The Greedy Set Cover Algorithm

Repeat until  $I = \phi$

Let  $\hat{S} = \operatorname{argmax} |S \cap I|$  (pick the set that covers the most items in  $I$ )

Set  $S = S - \{\hat{S}\}$  and  $I = I - \hat{S}$



This doesn't work!  
close.

But we'll see that it's pretty

## 16.2 Approximation Algorithms

An algorithm  $A$  is an  $\alpha$ -approximation algorithm if for any instance  $I$ :

It runs in time  $\text{poly}(|I|)$

It always outputs a feasible solution  $S$ .

It always guarantees:  $\text{cost}(S) \leq \alpha * \text{OPT}$  where  $\text{OPT}$  is the optimal solution, and  $\alpha$  is the desired approximation. (here we're looking at a minimization problem)

The greedy set cover algorithm is an approximation algorithm. So we want to find  $\alpha$  for it, so we know how good it actually is.

### Observation

If the optimal set cover has cardinality  $k$  then for any  $X \subseteq I$ , then there is some set  $S$  that covers at least  $\frac{1}{k} |X|$  items of  $X$ .

**Proof** Let the optimal solution be  $\{S_1^*, \dots, S_k^*\}$

Let the sets  $\{S_1^*, \dots, S_k^*\}$  cover every item in  $I$ .

So they cover every item of any subset  $X$  of  $I$ . So since  $X$  is covered by  $k$  sets, there must be some set that covers greater than one  $k$ th fraction of  $X$

### 16.3 Proof that Greedy Set Cover Almost Works

**Theorem 26.** If the optimal set cover has cardinality  $k$  then the greedy algorithm finds a solution of cardinality at most  $k \ln(n)$ .

*Proof.* wlog let the greedy algorithm output  $\{S_1, \dots, S_T\}$  and let the optimal solution be  $\{S_1^*, \dots, S_k^*\}$ .

We want to show that  $T \leq k \ln(n)$

Let  $I_t$  be the uncovered items in the start of step  $t$  For example,  $I_1$  is just  $I$ .

Since  $I_t \subseteq I$ , by the observation before, there's a set that covers at least  $\frac{1}{k} |I|$  items in  $I_t$

This means in step  $t$  it must pick a set the covers at least  $\frac{1}{k} |I_t|$  items in  $I_t$ .

$$\begin{aligned} \Rightarrow |I_{t+1}| &\leq |I_t| - \frac{1}{k} |I_t| \\ &= \left(1 - \frac{1}{k}\right) |I_t| \\ &\quad \cdot \\ &\quad \cdot \\ &\quad \cdot \\ &\leq \left(1 - \frac{1}{k}\right) \left(1 - \frac{1}{k}\right) \dots \left(1 - \frac{1}{k}\right) |I_1| \end{aligned}$$

We iterate  $t$  times, so

$$= \left(1 - \frac{1}{k}\right)^t |I_1|$$

And since  $I_1$  is just  $n$ :

$$= \left(1 - \frac{1}{k}\right)^t n$$

Key fact:  $1 - x < e^{-x} \forall x \neq 0$

So if we let  $\frac{1}{k} = x$ , then we have:

$$|I_{t+1}| < \left(e^{-\frac{1}{k}}\right)^t n$$

$$= e^{-\frac{t}{k}} n$$

Now setting  $t = k \ln(n)$ , then:

$$| I_{t+1} | < (e^{-\frac{k \ln(n)}{k}})^t n$$

$$= e^{-\ln(n)} n$$

$$= 1$$

$$\Rightarrow | I_t | = | I_{k \ln(n)+1} | < 1$$

which means that it's empty, which means we're done at step  $t$ . So there was  $t = k \ln(n)$  steps, we finished, meaning there's at most  $k \ln(n)$  sets that the algorithm picked.  $\square$

So this is a  $\log(n)$  – *approximation* algorithm for this problem. This is actually a bad approximation, but it's the best we've come up with unless we solve P=NP.

## 16.4 Running Time

There are  $n$  iterations, and there's at most  $n$  distance updates at each iteration, so  $O(n^2)$

## 17 The Hitting Set Problem

Given a collection  $S$  of  $m$  elements, and a collection  $I$  of  $\{I_1, \dots, I_n\}$  sets that are subsets of  $S$ .

We want to select as few elements as possible such that there is at least one element selected in every set.

In other words, we want to find the smallest set  $X$  of elements such that every set  $I$  is "hit".

It was left as an exercise to show that this is exactly the same as the set cover problem.