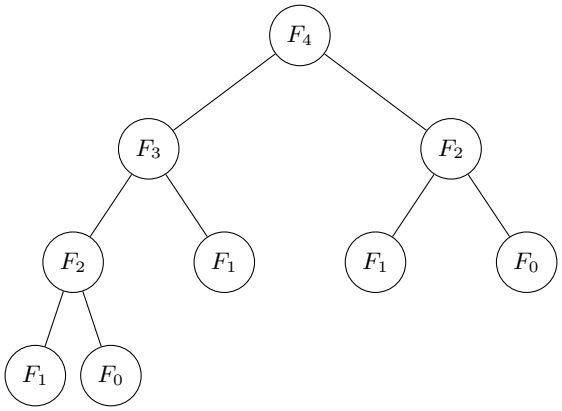
CS 170 Lecture Notes

Lecture Notes

Semester Fall 2025



Alexander Lu

Instructor: John Wright, Sanjam Garg

Contents

Chapter 1

Lecture 1 - Algorithms

1.1 Efficiency

The efficiency of integer arithmetic is important. The decimal system expreses integers in base 10, this form of writing is unlimited unlike Roman integers. Once we are able to add and multiply numbers for single digit integers a and b, we can add and multiply any two integers together.

To study efficiency, we first look at the Fibonacci Sequence. The fibonacci sequence F_n may be defined as:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

For any F_n, it will be double that of F_{n-2} but less than double of F_{n-1}. This hints at an exponential growth.

$$2^{\frac{n}{2}} < F_n < 2^n$$

The fibonacci sequence grows approximate to $2^{0.694n}$. The naive implementation follows the recursive algorithm exactly.

1.1.1 Algorithm Design

To design an algorithm, we have to keep 3 things in mind:

- Is the algorithm crrect? Does it produce the right outputs?
- How long does the algorithm take? How efficient is the algorithm?
- Can we do better?

Efficiency of an algorithm may vary a lot depending on our platform. Thus, we need a hardware independent method of measuring the efficiency of an algorithm. One simple way to measure efficiency is just to measure the **number of basic computer steps**.

Definition 1.1.1: $T(n)$

$T(n)$ is the number of basic computer steps required to solve a problem of size n.

In the case of our naive fibonacci algorithm:

$$T(n) = \begin{cases} c & \text{if } n = 0 \text{ or } n = 1 \\ T(n-1) + T(n-2) + c & \text{if } n > 1 \end{cases}$$

Where c is some constant independent of n. We can see that this algorithm is highly inefficient, exponential as we've shown earlier.

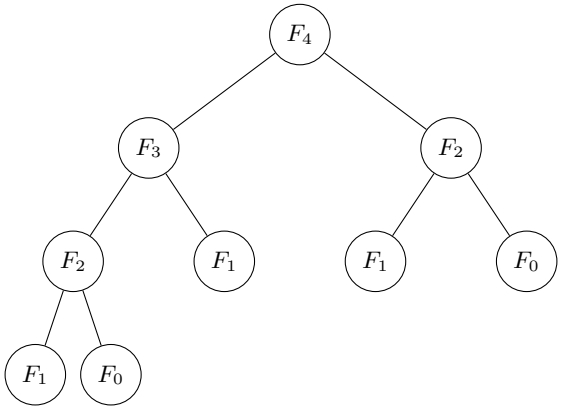


Figure 1.1: Naive Fibonacci Call Tree

The more efficient algorithm is to do this iteratively:

```
Fib(n):
    if n = 0 return 0
    if n = 1 return 1
    create an array F[0...n] # This is constant work (n-2)*c
    F[0] = 0, F[1] = 1
    for i = 2 to n:
        F[i] = F[i-1] + F[i-2] # This is constant work (n-2)*c
    return F[n]
```

The above algorithm is actually a flavor of **Dynamic Programming**, which is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable when the subproblems overlap, meaning that the same subproblems are solved multiple times. We have reduced the algorithm from exponential time down to linear time.

Another important thing to note is that addition is not necessarily a constant time operation. With larger numbers, the time for addition actually scales linearly with the number of bits that the number has. With large numbers, the algorithm is actually quadratic.

1.1.2 Asymptotic Analysis

We want an appropriate rough approximation that gives us a good sense of how performant an algorithm is:

- We do not care about constants
- We only care about the highest growing term.

Example.

> If our running time is $n^3 - n^2 + 7n$, then the quadratic and linear terms do not matter. We say that the algorithm is cubic time.

The standard way to express algorithm efficiency is Big-O notation.

Definition 1.1.2: Big-O Notation

Let $f(n)$ and $g(n)$ be functions that map from $\mathbb{Z}^+$ to $\mathbb{R}^+$. We say that $f(n)$ is $O(g(n))$ if there exist positive constants c and n_0 such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$.

Example.

> Suppose that we have $f(n) = an^3 + bn^2$. We know that $f(n)$ is $O(n^3)$. Because $f(n) < (a+b)n^3$ for sufficiently large n.

The iterative fibonacci algorithm that we came up with has an overall running time of $O(n^2)$.

1.1.3 Arithmetic

Going back to integer addition and multiplication, we know that to add two numbers, we may simply add them digit by digit while keeping track of carries. This operation is $O(n)$, where n is the number of bits. In fact, this algorithm is optimal, because we have to read all the digits of the number at least once.

When performing multiplication, we perform it digit by digit as well

$$\text{Multiplication}(x, y) = \sum_{i=0}^{n-1} x_i \cdot y \cdot 10^i$$

Since we peform multiplication digit by digit, this algorithm is $O(n^2)$. However, this algorithm is not optimal, we can do better.

Proposition 1.1.3

Multiplication using Divide and Conquer Approach
Divide and conquer is an algorithm that:

- Break up the problem into smaller subproblems

- Solve the subproblems recursively.
- Assemble the subsolutions into the final solution

We may develop the following algorithm for binary integers. Let x and y be n-digit numbers. We also assume that $n = 2^k$ for some k. We can break x into two parts, x_L and x_R, and similarly for y_L and y_R. This is the dividing part of our divide and conquer algorithm. We can see that:

$$x = x_L \cdot 2^{n/2} + x_R, \quad y = y_L \cdot 2^{n/2} + y_R$$

Thus:

$$x \times y = x_L y_L 2^n + (x_L y_R + x_R y_L) 2^{n/2} + x_R y_R$$

Each of $x_L y_R$, $x_R y_L$, $x_L y_L$ and $x_R y_R$ are multiplications of $n/2$ digit numbers. And these problems may also be solved by divide and conquer.

Suppose n denotes the number of bits in x and y.

```
MUL(x, y)
    if n = 1 return x * y
    x_L, x_R = x[:n/2], x[n/2:]
    y_L, y_R = y[:n/2], y[n/2:]
    p1, p2, p3, p4 = MUL(x_L, y_L), MUL(x_L, y_R),
    MUL(x_R, y_L), MUL(x_R, y_R)
    return p1 * 2^n + (p2 + p3) * 2^(n/2) + p4
```

For this algorithm, we can see that the time complexity may be calculated as:

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

The tree for this algorithm looks like the following:

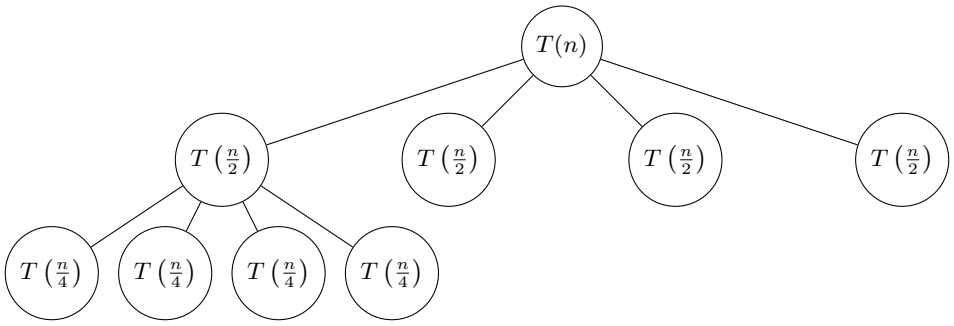


Figure 1.2: Naive Mul Call Tree

We can calculate the running time of this algorithm by looking at the time complexity of each level

$$O(n) + 4 * O(n/2) + ... + 4^{\log n} * O(n/2^{\log n})$$

Now note the following facts:

Fact 1.1.4

- for $1 + \alpha + \alpha^2 + \cdots + \alpha^n$ where $\alpha > 1$, then we know that this is O(last term)
- $2^{\log_2 n} = n$

Thus, we may just focus on our last term:

$$O\left(4^{\log n} \cdot n / 2^{\log n}\right) = O\left(\frac{4^{\log_2 n} \cdot n}{2^{\log_2 n}}\right) = O\left(\frac{2^{2 \cdot \log_2 n} \cdot n}{2^{\log_2 n}}\right) = O\left(\frac{n^2 \cdot n}{n}\right) = O(n^2)$$

However, we may actually tweak our formula for fining the product to reduce the complexity by a slight bit. Note that we may express the sum $x_L y_R + x_R y_L$ as:

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

Hence, we have reduced the total number of product terms we need to calculate from 4 to 3. The expression now becomes:

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

So the total time complexity now is:

$$O(n) + 3O(n/2) + 3^2 O(n/4) + \cdots + 3^{\log n} O(n/2^{\log n})$$

Once again performing our analysis:

$$3^{\log n} \cdot O(n/2^{\log n}) = \frac{3^{\log n} \cdot n}{2^{\log n}} = O\left(3^{\log_2 n}\right) = O\left(n^{\log_2 3}\right)$$

Chapter 2

Lecture 2 - Divide and Conquer

2.1 Divide & Conquer

We start with a problem of size n, and want to divide it into b smaller subproblems that we can tackle more easily. We can then assemble solutions of smaller sub problems into the solution to the original problem. The recurrence relation for D&Q problems is typically as such:

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

Where a, b, and d are constants. n^d represents the time that it takes to split the problem, or the total work down in each recursive call, like adding the subproblems, or splitting them. In the previous lecture, we looked at the problem of calculating the product of two integers. To do this, we split each integer x and y into their left and right parts respectively. In that problem, we have reduced the problem of finding the product of n bit numbers into products of $\frac{n}{2}$ bit numbers. Thus, we had $a = 4$ different subproblems, and reduced the problem size by a factor of $b = 2$, and had $d = 1$ which was linear work each layer.

Our fix last class was reducing the number of subproblems needed from 4 subproblems, down to 3 subproblems.

2.1.1 Recurrence Relations

We expand on the facts taught last lecture.

For a geometric progression $1, \alpha, \alpha^2, \ldots, \alpha^k$. Now, we have three cases depending on α:

$$1 + \alpha + \alpha^2 + \ldots + \alpha^k \to \begin{cases} O(1), & \alpha < 1, \\ O(k), & \alpha = 1, \\ O(\alpha^k), & \alpha > 1 \end{cases}$$

Secondly, we have $2^{\log_2 n} = n$.

Now we explore a way of solving the recurrence relation using something called the **Master's**

Theorem.

Theorem 2.1.1: Master's Theorem

For $a > 0$, $b > 1$, $d \geq 0$ if $T(n) = aT\left(\lceil \frac{n}{b} \rceil\right) + n^d$, then:

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

At each level of our divide and conquering, we divide the problem size by b, and introduce a smaller problems. At the final level of $\log_b n$, we have the smallest subproblems and we have $a^{\log_b n}$ subproblems. Each subproblem can be interpreted to have work k^d where k is the size of the problem. We can look at this in terms of levels on the recurrence tree.

$$\underbrace{1 \cdot cn^d}_{\text{level } 0} + \underbrace{a \cdot c\left(\frac{n}{b}\right)^d}_{\text{level } 1} + \underbrace{a^2 \cdot c\left(\frac{n}{b^2}\right)^d}_{\text{level } 2} + \ldots + \underbrace{a^{\log_b n} \cdot c\left(\frac{n}{b^{\log_b n}}\right)^d}_{\text{level } \log_b n}$$

This is equal to:

$$cn^d \cdot [1 + \frac{a}{b^d} + \left(\frac{a}{b^d}\right)^2 + \cdots + \left(\frac{a}{b^d}\right)^{\log_b n}]$$

Now we analyze the cases:

- **Case 1:** If $\frac{a}{b^d} < 1$, then we know that $d > \log_b a$, which implies $O(n^d)$.
- **Case 2:** If $\frac{a}{b^d} = 1$, then we know that $d = \log_b a$, which implies $O(n^d \log n)$.
- **Case 3:** If $\frac{a}{b^d} > 1$, then we know that $d < \log_b a$, which implies $O(n^d \cdot \frac{a^{\log_b n}}{(b^{\log_b n})^d}) = O(a^{\log_b n})$.

Intuitively, d represents the work that it takes to divide and combine the problems, and $\log_b a$ represents the growth of the number of subproblems. Thus, we can think of the overall complexity as being driven by the larger of these two factors.

2.2 Matrix Multiplication

We have two $n \times n$ matrices A and B. We want to output matrix $C = AB$. The matrix multiplication formula for C_{ij} is given by $C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$. Assuming each multiplcation is a $O(1)$ operation, then it takes $O(n)$ time to compute each entry of C. This is because we have n multiplications to compute, and also summing them up.

Since there are n^2 entrices in C, the overall time complexity is $O(n \cdot n^2) = O(n^3)$.

We can create a divide and conquer algorithm as such:

Definition 2.2.1: 1969 Strassen divide and conquer algorithm

We express matrices X and Y as:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Where $A, \ldots, H$ are all $\frac{n}{2} \times \frac{n}{2}$ matrices. Then we have:

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

The recurrence relation is:

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

Where $a = 8$, $b = 2$, and $d = 2$. The work done at each level is n^2 because have to add n^2 terms together. We can now apply the masters theorem, and we see that $d < \log_b a = 3$. Hence, this algorithm is $O(n^{\log_2 8}) = O(n^3)$, which is no better than the naive algorithm.

A little bit dissapointing, but we have established a framework for divide and conquer. What's left is finding a method to decrease the value of a. There is a more efficient algorithm that decreases a from 8 to 7.

$$X \times Y = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_3 - P_5 - P_7 \end{bmatrix}$$

Where

- $P_1 = A(F - H)$
- $P_2 = (A + B)H$
- $P_3 = (C + D)E$
- $P_4 = D(G - E)$
- $P_5 = (A + D)(E + H)$
- $P_6 = (B - D)(G + H)$
- $P_7 = (A - C)(E + F)$

2.3 Median Finding

Our input is a list of n numbers that are unsorted. We want to find the $\lceil \frac{n}{2} \rceil$-th smallest number in the list.

The naive algorithm is the first sort the list and then take the middle element. This algorithm is $O(n \log n)$.

We can do better by considering a more general or harder problem. We introduce the *selection problem*. Where we find the k-th smallest number in a unsorted list in S. We can then use the *selection* algorithm to find the median by setting $k = \lceil \frac{n}{2} \rceil$.

```
SELECT(S, k):
    Select a pivot v from s
    Split S into three parts S_L, S_V and S_R
        S_L = all elements in S such that a < V
        S_V = all elements in S such that a = V
        S_R = all elements in S such that a > V
    If k <= |S_L|:
        // Recur on the left part
        return SELECT(S_L, k)
    If k > |S_L| + |S_V|:
        // Recur on the right part
        return SELECT(S_R, k - |S_L| - |S_V|)
    // If we get here, then k is in S_V
    return v
```

This algorithm is no longer a deterministic algorithm, as the the random choice of the pivot can lead to different efficiencies. The worst case situation is that the pivot always uses the smallest or largest element in S, leading to an unbalanced partition, and a runtime of $O(n^2)$. The ultra good case is that the pivot is actually the k-th smallest element. In which case, the runtime is $O(n)$. The good case is when the size of the left and right partition are roughly equal.

Chapter 3

Lecture 3 - Divide & Conquer Part 2

3.1 Median of Medians

We need a new measurement for the median searching algorithm that we devised last lecture, as the runtiem of the algorithm is not deterministic. Hence, we introduce the concept of a **Expected number of steps taken** T(N).

$$T(N) = \sum_t tP(t)$$

The running time does not depend solely on the input, but also the randomness/choice of the pivot that we choose. In our median finding algorithm, if our input collection was sorted, there is a probability of 0.5 that our pivot choice occurs in the middle section of the array. In this ideal case, the largest size of S_L or S_R is $\frac{3n}{4}$. We thus say v is a good pivot if it lies between the $\frac{n}{4}$-th smallest element and the $\frac{3n}{4}$-th smallest element.

Claim: Good Pivot

The probability of choosing a good pivot is $\frac{1}{2}$. And if v is a good picot, then $\max\{S_L, S_R\} \leq \frac{3n}{4}$. So the size of the problem is reduced to at most $\frac{3n}{4}$.

There are $\frac{n}{2}$ good pivots. In this case, This leads to a recurrence relation of:

$$T(N) = T\left(\frac{3N}{4}\right) + O(N) = O(N)$$

Now:

$$
\begin{aligned}
T(n) &= \mathbb{E}(\#\text{ of steps to compute select on input S of size n}) \\
&= \mathbb{E}(\#\text{ of steps to reduce list to size} \leq \frac{3n}{4}) + \mathbb{E}\{\#\text{ of steps to compute select on input list of size} \leq \frac{3n}{4}\} \\
&\leq \mathbb{E}(\#\text{ of pivots to get a good pivot}) \cdot cn + T\left(\frac{3n}{4}\right) \\
&= 2cn + T\left(\frac{3n}{4}\right) \\
&= O(n)
\end{aligned}
$$

Note that we computed $E = \mathbb{E}(\#\text{ of pivots to get a good pivot})$ as follows:

$$
\begin{aligned}
E &= \sum_{i=1}^{\infty} i \cdot P(\text{get good pivot on i-th try}) \\
&= \sum_{i=1}^{\infty} i \cdot \left(\frac{1}{2}\right)^i \\
&= \frac{1}{\left(1 - \frac{1}{2}\right)^2} = 2
\end{aligned}
$$

We now introduce a deterministic algorithm for median finding. Instead of sampling the pivot at uniform, we design a `Pick Pivot(S)` algorithm that takes the set S and splits it. This algorithm is called **Median of medians**

`PICK PIVOT(S)`:

```
1. Split S into n/5 groups for 5 elements each
2. Find the median of each list separately B=(B_1, B_2, ..., B_(n/5))
3. Find the median of B recursively SELECT(B, n/10)
```

Steps 1 and 2 of `PICK PIVOT(S)` take $O(n)$ time. Step 3 takes $T(n/5)$ time. Thus, the overall time complexity of `PICK PIVOT(S)` is $T(n/5) + O(n)$.

The whole purpose of defining `PICK PIVOT(S)` is to ensure that we always get a good pivot. Moreover, the algorithm itself to find the median of medians is also a linear time algorithm if we apply the master theorem. To show that we always select a good pivot, we can draw a grid representing the subarrays of length-5 that we created: Draw a grid representing the subarrays of length-5 that we created:

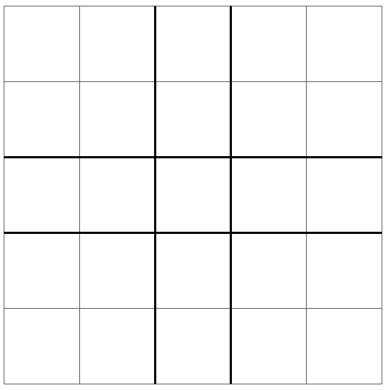


Figure 3.1: Median of Medians Grid

The center element represents the median of medians. The two thick horizontal lines represent the elements that are greater than or equal to the median of medians, and the two thick vertical lines represent the elements that are less than or equal to the median of medians. We can see that there are at least 3 elements in each row and column that are greater than or equal to the median of medians, and at least 3 elements in each row and column that are less than or equal to the median of medians. Thus, we have at least $3 \cdot \frac{n}{10} = \frac{3n}{10}$ elements that are greater than or equal to the median of medians, and at least $\frac{3n}{10}$ elements that are less than or equal to the median of medians. Hence, we have at most $\frac{7n}{10}$ elements on either side of the pivot.

If we then apply this to our median finding algorithm, we have the following recurrence relation:

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

This type of recurrence relationship is new to us, but we may rewrite it more generally as:

$$T(n) = T(an) + T(bn) + cn$$

To analyze this, we draw the recurrence tree:

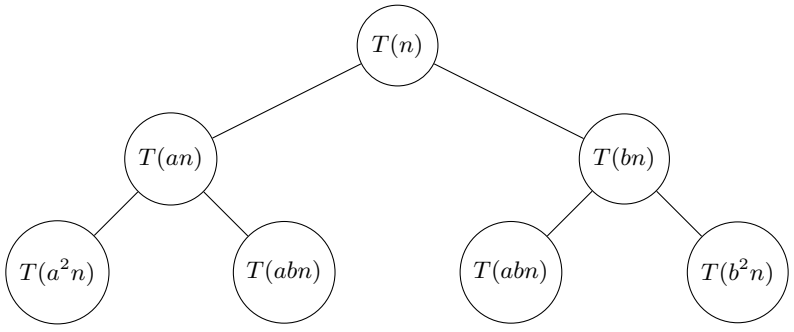


Figure 3.2: General Recurrence Tree

We can see that the work done at each level is cn. At the first layer, the total work done is cn. At the second layer, it is $(a+b)n$, and the third layer is $(a^2+ab+b^2)n = (a+b)^2n$.

We can see that the work done at each layer is $n(a+b)^k$ where k is the layer number. Now we may express $T(n)$ as

$$T(n) = \sum_{k=0}^{h} n(a+b)^k = O(n)$$

Since $a + b < 1$, we know that this is a decreasing geometric series, and thus the last term dominates, giving us $O(n)$.

3.2 Exponentiation

Given an input of integer n, we want to output 2^n. The naieve algorithm is just to perform n multiplications, which is $O(nM(n))$, where $M(n)$ is the time complexity of multiplying two multiplications. We can also apply devide and conquer onto this algorithm. To do this, we divide the exponent by 2 each time as:

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ (a^{n/2})^2 & \text{if } n \text{ is even} \\ a \cdot (a^{(n-1)/2})^2 & \text{if } n \text{ is odd} \end{cases}$$

The algorithm may be expressed as:

```
EXP(a, n):
    if n = 0 return 1
    if n is even:
        half = EXP(a, n/2)
        return half * half
    else:
        half = EXP(a, (n-1)/2)
        return a * half * half
```

This algorithm requires $\log n$ multiplications.

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

3.3 Fibonacci Revisited

We can apply the exponential algorithm to calculate Fibonacci numbers. This is because we can express the last two terms of the fibonacci sequence as a $2-tuple$. As such, we can multiply the $2-tuple$ by a matrix to get the next term.

Theorem 3.3.1: Fibonacci Matrix Exponentiation

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix}$$

Now all we have to do to calculate F_n is to calculate the matrix exponentiation of

the matrix $A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ to the power of $n-1$. We only require $\log n$ multiplications of the matrix. Like earlier, this fibo algorithm now gives us $O(M(n)) = O(n^{\log_2 3})$ time complexity.

Once again, we remember that $2F_{n-2} \leq F_N \leq 2F_{n-1}$.

Chapter 4

Lecture 4 - Closest Pair and Graphs

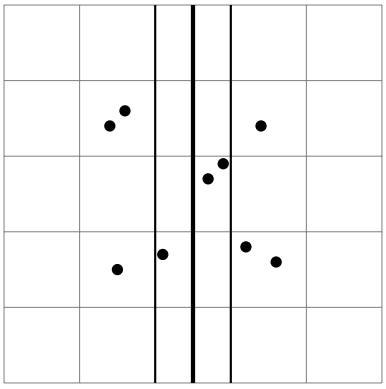
4.1 Closest Pair Problem

We introduce the **Closest Pair Problem**. We have n points in a plane $P = \{p_1, \ldots, p_n\}$. We want to output the distance between the closest pair of points. We define the distance $\Delta(p_1, p_j)$ between two points as $\sqrt{(x_1 - x_i)^2 + (y_1 - y_j)^2}$.

The naive algorithm is is to calculate the distance between every pair of points, and store the minimum distance. This algorithm runs in $O(n^2)$ time.

To find a more optimal solution, we attempt to use divide-and-conquer to do so. Intuitively, we may attempt to divide the plane into two sides down a specific point, and then find the closest pair on each side. However, this does not work, as the closest pair may be split across the two sides. Furthermore, the act of checking the points across the two sides and merging the solutions appear to be a $O(n^2)$ problem.

We must handle points that cross over the left and right divisions. To do this, we define $d = \min\{d_L, d_R\}$, where d_L is the closest pair on the left side, and d_R is the closest pair on the right side. We then only need to check points that are within d distance from the



dividing line. We define a vertical strip of width $2d$ centered at the dividing line. We then only need to check points that are within this strip. We have some properties

Claim: Closest Pair Properties

- if $p \in L$ and $q \in R$ where L and R are the left and right sides of the strip, and $\Delta(p, q) < d$, then $|p_x - mid_x| < d$. and $|q_x - mid_x| < d$.
- Any tile that is $\frac{d}{2} \times \frac{d}{2}$ that is entirely in L or R can have at most 1 point.
- If $p \in$ strip and $p_y \leq q_y$ then there are at most 7 points on the strip such that $p_y \leq q_y$ and $\Delta(p, q)$ could be less than d.

Proof. We prove the properties:

- if p is outside this strip $|p_x - mid_x| \geq d$, then $\Delta(p_x, q_x) > \Delta(p, (mid_x, p_y)) \geq d$, a contradiction.
- Prove by contradiction. Suppose that there are 2 points p and q in the sametile, then $\Delta(p, q) \leq \sqrt{(\frac{d}{2})^2 + (\frac{d}{2})^2} = \frac{d}{\sqrt{2}} < d$. This is a contradiction, as we assumed d to be the closest distance of two points in the left and right split.
- The algorithm will start from the point with the greatest y value and then proceed down, so we only need to check the points below p. We can draw a tile of size $d \times d$ around p. We can see that there are at most 8 tiles of size $\frac{d}{2} \times \frac{d}{2}$ that intersect with this tile. Each tile can have at most 1 point, and one of the tiles contains p. Thus, there are at most 7 other points that we need to check.

□

We define a preprocessing function to help us write `CLOSEST`.

```
PREPROCESS(P):
    P_x = sort(P, key=lambda p: p.x)
    P_y = sort(P, key=lambda p: p.y)
    return P_x, P_y
```

Now we can write the CLOSEST function:

```
CLOSEST(P_x, P_y):
    // Base Case:
    if |P_x| <= 3:
        return brute_force(P_x)

    // Step 1: Divide
    mid = |P_x| // 2
    mid_x = P_x[mid].x
    L_x, R_x = P_x[:mid + 1], P_x[mid + 1:]
    L_y, R_y = [], []
    for p in P_y:
        if p.x <= mid_x:
            L_y.append(p)
        else:
            R_y.append(p)

    // Step 2: Recurse
    d_L = CLOSEST(L_x, L_y)
    d_R = CLOSEST(R_x, R_y)

    //Step 3: Conquer
    d = min(d_L, d_R)
    strip = []
    for p in P_y:
        if |p.x - mid.x| < d:
            strip.append(p)
    k = |strip|
    for i in range(k):
        for j = i + 1 to min(i + 7, k - 1):
            d = min(d, dist(strip[i], strip[j]))
    return d
```

The time complexity of step 1 is $O(n)$, as we are just splitting the array in half. Step 2 is $2T(n/2)$. Step 3 is also $O(n)$, as we are just iterating through the strip and checking at most 7 points for each point in the strip. Thus, the overall recurrence relation is:

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

Solving this recurrence relation using the master's theorem gets us $\Theta(n \log n)$ time complexity. Furthermore, the preprocessing algorithm is only ran once, and takes $O(n \log n)$ time. Thus, the overall time complexity of the closest pair algorithm is $\Theta(n \log n)$.

4.2 Graphs

A graph G is a pair of two sets (V, E) where V is the set of vertices and E is the set of edges. For convention, we denote $|V| = n$ and $|E| = m$. We know that $m < n^2$ as there are at most $\binom{n}{2}$ edges in a simple graph.

4.2.1 Graph Representations

Graphs may be undirected or directed in nature. A directed graph has a direction attached to each edge, whereas an undirected graph is a bidirectional graph. There are two popular ways of representing graphs.

Definition 4.2.1: Adjacency Matrix

An adjacency matrix is a $n \times n$ matrix A where $A_{ij} = 1$ if there is an edge between nodes i and j. Otherwise, $A_{ij} = 0$.

$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Definition 4.2.2: Adjacency List

An adjacency list is a list of lists where the i-th list denotes all the edges to neighbors of the i-th node.

$$1 \to 2, 4$$
$$2 \to 1, 3, 5$$
$$3 \to 2, 4$$
$$4 \to 1, 3, 5$$
$$5 \to 2, 4$$

We may compare the two representations:

	Adjacency Matrix	Adjacency List
Space Complexity	$O(n^2)$	$O(n+m)$
Time Complexity to check if (u, v) inE	O(1)	$O(\deg u)$
List all neighbors of node u	$O(n)$	$O(\deg u)$

Table 4.1: Graph Representation Comparison

4.3 Connectivity in Graphs

The main questions that we want to ask are:

- Is there a path from u to v?
- Is the graph connected?
- What are the connected components of the graph?

Chapter 5

Lecture 5 - Graphs

Graphs capture pairwise relationships between objects. A common problem in graphs is the issue of connectivity, that is, can we go from node $u \to v$.

5.1 Explore and Depth First Search

We consider the algorithm `EXPLORE(G, v)`, where our input is a graph G and a vertex v. We want to output a vector visited where visited(u) is set to true if u is reachable from v.

```
EXPLORE(G, v):
visited(v) = true
for each (v, u) in E:
    if not visited(u):
        EXPLORE(G, u)
```

Theorem 5.1.1: EXPLORE

EXPLORE(G, v) visits exactly the vertices u that have a path $v \to u$ in G.

Proof. We want to show that if a vertex u is visited, then $\exists$ a path from $v \to u$. If $\exists$ a path $v \to u$, then u is visited. Assume that u is not visited, then there exists some vertex in the path from $v \to u$ such that it is the first one that is not visited. Call this vertex z. The the vertex y before z, must have been visited, and thus z must have been visited, a contradiction. □

We want to develop an algorithm that can traverse all vertices in a graph, not just the ones reachable fro some node.

```
DFS(V):
    forall u in V:
        visited(u) = false
```

```
for each v in V:
    if not visited(v):
        EXPLORE(G, v)
```

Theorem 5.1.2: Depth First Search

The runtime of DFS is $O(n+m)$.

Proof. To analyze the runtime of depth first search, we note the following:

- Explore on each vertex v is invoked exactly once. Order of $O(n)$.
- The runtime of explore on input v is $O(1+\deg v)$

Combining the two facts, we see that the overall runtime can be expressed as a sum over the explore calls on each vertex:

$$\sum_{v \in V} O(1+\deg v) = O(n+2m) = O(n+m) = O(|V|+|E|)$$

Note to self: There is a $2m$ term because we techically visit each edge twice. □

It is important to note that the runtime of DFS depends on the underlying graph representation we are using. If we use an adjacency matrix, then the overall running time os $O(n^2)$, as we have to check each entry in the matrix. If we use an adjacency list, then the overall running time is $O(n+m)$.

5.2 Connected Components

How may we adpat DFS to fnd the number of connected components in a graph? And how do we label each vertex with the connected component that they belong to? The simplest way is to introduce a counter into our DFS algorithm, that increments each time we discover a new connected component.

```
CONNECTED COMPONENTS(G):
    forall u in V:
        visited(u) = false
    cc = 0
    for each v in V:
        if not visited(v):
            cc = cc + 1
            EXPLORE(G, v)
    return cc
```

5.2.1 DFS over directed graphs

DFS works similarly in directed graphs. The only difference is that we follow edges that go out of vertices. There are different sort of edges that we may encounter in a directed graph. There are tree edges, back edges, forward edges, and cross edges.

Definition 5.2.1: Edge types

- **Tree edges:** Edges that are used in the DFS tree.
- **Back edges:** Edges that point from a node to one of its ancestors in the DFS tree.
- **Forward edges:** Edges that point from a node to one of its descendants in the DFS tree.
- **Cross edges:** Edges that go between two nodes that are not ancestor/descendant of each other.

We can actually classify these edges by just keeping track of the time in which we visit and leave a vertex: We define the `PREVISIT(v)` and the `POSTVISIT(v)` algorithms for a graph traversal:

```
init time = 1
PREVISIT(v):
    pre[v] = clock
    clock = clock + 1

POSTVISIT(v):
    post[v] = clock
    clock = clock + 1

EXPLORE(G, v):
    visited[v] = true;
    PREVISIT(v)
    for each (v, u) in E:
        if not visited[u]:
            EXPLORE(G, u)
    POSTVISIT(v)
```

For every vertex $v \in V$ we have a tuple $[\text{pre}[v], \text{post}[v]]$. We make some observations:

- For any pair of vertices $u, v \in V$, the intervals $[\text{pre}[u], \text{post}[u]]$ and $[\text{pre}[v], \text{post}[v]]$ are either disjoint or one is contained in the other.
  - If the intervals are disjoint then u and v do not share a descent-ancestor relatinship. Otherwise, they are ancestors and descendants of each other.
- G has a cycle if and only if G has a back edge.

  Proof. Suppose that $u \to v$ is a back edge. Then it must mean that v is a vertex that has already been visited, so v is an ancestor of u. Hence, we have a path from $v \to u$, and a edge from $u \to v$, so we have a cycle.

  Conversely, suppose that G has a cycle $v_0 \to v_1 \to \cdots v_i \to \cdots \to v_k \to v_0$. Let

v_i be the first vertex to be visited. Then, our algorithm would visit all the vertices in the cycle, and when it reaches the vertex v_{i-1}, it would see that there is an edge from $v_{i-1} \to v_i$. But since there is already a path from $v_i \to v_{i-1}$, this edge is a back edge. □

- (u, v) is a back edge if and only if post$[u] <$ post$[v]$.

  Proof. We have several cases to consider:

  - if $[_u]_u$ is contained in $[_v]_v$, then u is a descendant of v. Hence, (u, v) is a back edge. We also have post$[u] <$ post$[v]$.
  - if $[_v]_v$ is contained in $[_u]_u$, then v is a descendant of u. Hence, (u, v) is a forward edge. We also have post$[u] >$ post$[v]$.
  - if $[_u]_u$ and $[_v]_v$ are disjoint, and the interval of u comes before v, then (u, v) is a cross edge. We also have post$[u] >$ post$[v]$.
  - if $[_u]_u$ and $[_v]_v$ are disjoint, and the interval of v comes before u, this case is not possible, as we would have visited u before v.

  □

5.3 Directed Acyclic Graphs (DAGs)

A directed acyclic graph (DAG) is a directed graph with no cycles. DAGs are useful to schedule dependency graphs, where certain events/objects requires each other.

5.3.1 Topological Sort(Linearization)

Definition 5.3.1: Topological Sort

A topological sort of a directed graph $G = (V, E)$ is an ordering of the vertices $v_1, v_2, \ldots, v_n$ such that if $(v_i, v_j) \in E$, then $i < j$.

The naive algorithm is to just start with any source node (a node with in-degree 0). If there is not a source node, then we know the graph is cyclic. We take the node, record it in our topological sort, and remove the node and all it's edges from the graph. We keep repeating this process, until we detect a cycle, or there are no nodes left. This algorithm runs in $O(n(n+m))$ time, as we may have to search through all the nodes to find a source node.

We make an observation. Sicne the graph is acyclic, we know that there exists no back-edges, thus for all $(u, v) \in E$, we know that post(u) > post(v). Thus, we just have to sort the vertices in the order of decrease post(v). This is the same as reverse the postorder traversal of the DFS. This algorithm runs in $O(n+m)$ time. Hence, topological sort reduces to DFS.

Chapter 6

Lecture 6 - Connectivity in Directed Graphs

In a directed graph, connectivity is more hard to identify. If vertex A has a directed edge to B, then A and B are connected, but B and A are not, as we cannot get to A from B.

Definition 6.0.1: Strongly Connected

u is strongly-connected to v if $\exists$ a path from u to v and there $\exists$ a path from v to u.

For vertices that are strongly connected to each other, we may combine them into one super-vertex.

Fact 6.0.2

Every graph G can be decomposed as a DAG of strongly connected components (SCCs).

Intuitively, each strongly connected component is composed of cycles. Every cycle that is in the graph thus belongs in a strongly connected component by definition. Furthermore, in a directed graph, any vertex that is involved in a cycle is part of a strongly connected component. We want to create an algorithm that takes in a directed graph G and decompose G into a DAG of strongly connected Components.

6.1 Decomposing Directed Graphs

Input: Directed Graph $G = (V, E)$
Output: DAG of strongly connected components of G
We use the following intution:

- Find a vertex v that is in a sink SCC (a SCC with no outgoing edges). This ensures that our algorithm does not escape the SCC.
- Run `EXPLORE(v)`

- Remove the visited vertices and recurse.

As the algoritm removes vertices, new sink SCCs may be created. Steps 2 and 3 are not difficult to implement. The magic is in step 1.

Recall: In a DFS traversal, the vertex with the highest post value is in a source SCC (a SCC with no incoming edges). This is because if it wasn't in the sink SCC, then it must have been reachable from some other vertex u, and our DFS would've found it earlier on. Our idea is to take the graph G and flip the direction of all the edges. We call this new edge G_R. We can then find the source SCC in G_R, which would then be a vertex in a sink SCC of G.

- $G \to G_R$ with edges reversed.
- Let $\text{post}_R[v]$ be the post time of v in G_R.
- Run DFS on G a second time in decreasing $\text{post}_R[v]$ order.

```
V = decreasing post_R order // O(n + m) time
SCC(G):
    for all v in V:
        visited[v] = false
    count = 0
    ccnum[1 .. n] = 0

    Explore(G, v)
        visted[v] = true
        ccnum[v] = count
        for each (v, u) in E:
            if not visited[u]:
                EXPLORE(G, u)

    for each v in V:
        if not visited[v]:
            EXPLORE(G, v)
            count = count + 1
```

Theorem 6.1.1: SCC Algorithm

The SCC algorithm runs in $O(n + m)$ time and correctly computes the strongly connected components of G.

6.2 Breadth First Search (BFS)

DFS is really good at finding if a graph is connected. However, it does not do a good job of telling us which path between two vertices is the best (shortest).

Input: Graph $G = (V, E)$, directed or undirected vertex, $s \in V$
Output: For each vertex v reachable from s, `dist[u]` is set to the distance from u to v.

The BFS explores the graph levels at a time rather than depths. The algorithm traverses the closer nodes first and the farther nodes later. While DFS uses the stack data structure, BFS uses a queue (the stack is implied through the function calls). We define the function as such:

```
BFS(G, s):
    for all v in V:
        dist[v] = infinity
    queue = [s]
    dist[s] = 0
    while queue is not empty:
        u = queue.pop()
        for (u, v) in E:
            if (dist[v] == infinity):
                dist[v] = dist[u] + 1;
                queue.push(v)
```

Consider the following graph:

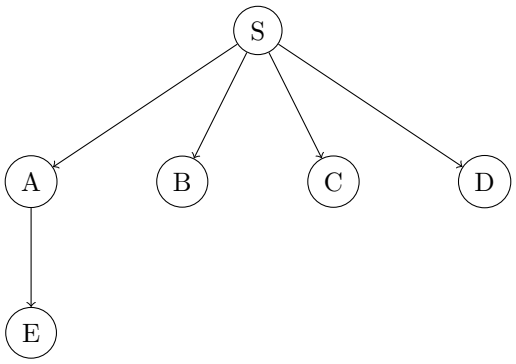


Figure 6.1: Example Graph for BFS

We can track the queue of the BFS algorithm as follows:

Step	Queue	dist
0	[S]	S:0, A:∞, B :∞, C:∞, D:∞, E:∞
1	[A, B, C, D]	S:0, A:1, B:1, C:1, D:1, E:∞
2	[B, C, D, E]	S:0, A:1, B:1, C:1, D:1, E:2
3	[C, D, E]	S:0, A:1, B:1, C:1, D:1, E:2
4	[D, E]	S:0, A:1, B:1, C:1, D:1, E:2
5	[E]	S:0, A:1, B:1, C:1, D:1, E:2
6	[]	S:0, A:1, B:1, C:1, D:1, E:2

Table 6.1: BFS Queue and Distance Tracking

We now proof the correctness of BFS through induction:

Proof. $\forall d = 0, 1, \ldots$, there exists a moment where:

- All nodes at distance $\leq d$ from s have been visited and have their dist set correctly.
- All other nodes have dist set to ∞.
- The queue contains exactly the nodes at distance d.

Base Case: $d = 0$. Only s is in the queue and no other nodes have been visited. Dist is set correctly.
Inductive Step: Assume the claim is true for $d - 1$. Using the 3 properties, we can see that when we pop the first node u in the queue, it must be at distance d. We then visit all its neighbors. If a neighbor v has not been visited, then it must be at distance $d + 1$, and we set its dist to $d+1$ and add it to the queue. After we finish visiting all nodes at distance d, the queue contains all nodes at distance $d + 1$. Thus, the claim is true for d. □

6.3 Shortest Path Problem

Suppose G is now a weighted graph. We want to find the shortest path in G.

6.3.1 G^{extended}

Let each edge $e \in E$ has a weight l_e. Suppose we transform $G \to G^{\text{extended}}$ by replacing each edge in G by l_e edges of weight 1 each.

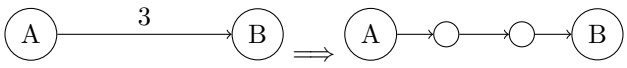


Figure 6.2: Graph Extension Example

We can run BFS on G^{extended} to find the shortest path from s to t. However, this is inefficient, as G^{extended} may be very large if we have large weights.

6.3.2 Dijkstra's Algorithm

We don't want to visit the graph one weight increment at a time, we just want to skip to the next closest vertex directly. Hence, we want to use a priority queue to keep track of which vertex to visit next.

Proposition 6.3.1

Use a priority queue where elements are processed based on the priority of their distance, rather than first-in-last-out. Each element in a priority queue may be represented as a tuple (v, k).

We define the following operations:

- `INSERT(v, k)`: Insert element v with priority k into the queue.

- `DECREASE_KEY(v, k)`: Decrease the priority of element v to k.
- `DELETE_MIN()`: Remove and return the element with the smallest priority.
- `MAKE_QUEUE()`: Builds a queue given a set $\{e, \text{key}\}$ of elements and their priorities. Can potentially be faster than insertion one by one.

All operations unless otherwise stated run in $O(\log n)$ time.

The algorithm we are going to derive is called **Dijkstra's Algorithm**. The algorithm is as follows:
Dijkstra(G, s):
Input: A weighted graph $G = (V, E)$ with weights $l_e \geq 0$ for all $e \in E$, and a source vertex $s \in V$.
Output: For each vertex v reachable from s, dist$[v]$ is set to the length of the shortest path from s to v.

Chapter 7

Lecture 7: Dijkstra's Algorithm

Let $G = (V, E)$ be a graph, let l be a mapping from $E \mapsto \mathbb{R}$ that denotes the weights of edges, and let s be a source vertex. We define `DIJKSTRA(G, l, s)` as such:

```
DIJKSTRA(G, l, s)
    for all u in v:
        dist[u] = inf
        prev[u] = null

    dist[s] = 0
    H = makequeue(V) // Using dist-vals as keys
    while H is non-empty:
        u = DELETE_MIN(H)
        for all (u,v) in E:
            if (dist[v] > dist[u] + l(u,v)):
                dist[v] = dist[u] + l(u,v)
                prev[v] = u
                DECREASE_KEY(v, dist[v])
```

Dijkstra is basically the same version as BFS, but we use a priority queue to keep track of the next closest node in our traversal.

To analyze the runtime of Dijkstra, we note the following:

- There are a total of $|V|$ nodes in H.
- there is a total of $O(|V|)$ calls to `DELETE_MIN`.
- There are total of $O(|E|)$ calls to `DECREASE_KEY`, which happens when we process every edge.

Hence we have $O(V + E)$ inserts and deletes, where each operation takes roughly $O(\log V)$ time to accomplish. Hence our overall running time is $O((|V| + |E|) \log |V|)$.

7.1 Proof of Correctness

Theorem 7.1.1

At the end of each iteration of the while loop, $\exists R, d$ such that:

1. (a) $\forall v \in R$, the distance $s \to v \leq d$
   (b) $\forall v \in V \setminus R$, the distance $s \to v > d$.
2. For all $v \in V$. dist$[v]$ is set to the minimu distance via traversing through vertices in R.

R is the set of all vertices that we have visited. d is the distance of the last vertex we visited.

Proof. We prove via induction.
Base Case: Let $R = \{s\}$ and $d = 0$. 1a and 1b are obviously true. Moreover, by the end of the first while loop iteration, all vertices v such that $(u, v) \in E$ have $dist[v] = l(u, v)$, which means 2 is also satisfied.
Inductive Step: Assume that the claim is true for iteration i. Now for iteration $i + 1$, we note that there is already a set R of vertices that we traversed, and a set $V \setminus R$ of vertices we did not traverse yet. We know there exists a vertex $v^* \in V \setminus R$ such that the distance from $s \to v^*$ is the minimum among all vertices in $V \setminus R$. We set $d^* = \text{dist}[v^*]$ and add v^* to R to form R^*. We now need to show that 1a, 1b, and 2 are still satisfied.

We claim that the shortest path from s to v^* does not take any vertices in $V \setminus R \setminus \{v^2\}$. It's not possible for the shortest path to go from s, to the set $V \setminus R$, then to v^*. This is because if it were true, then the first such vertex that we encounter in $V \setminus R$ would have a distance larger than d^*, a contradiction. Moreover, every other vertex in $V \setminus R$ is further away from s than v^*. Thus, the shortest path from s to v^* must go through vertices in R only. Hence, 1a and 1b are satisfied.

Now that we updated R to R^*, we need to show that 2 holds. There are two possible paths in this case: Either we reach arbitrary vertex v through R only, or we reach v through R^*, which means we go through v^*. In the first case, the distance is still the same as before, and in the second case, the distance is $\text{dist}[v^*] + l(v^*, v)$. We update dist$[v]$ to be the minimum of these two values. Thus, 2 is satisfied.

□

7.2 Shortest paths in presence of negative edges

Negative edges do not make sense in undirected graphs, as we may just repeatedly travel on an edge to keep reducing the total distance. Furthermore, negative weights may not make sense in directed graphs, as we may have negative cycles that allows us to keep reducing our weight. Hence, whenever we talk about shortest path with negative edge weights, it'll always be under the context that the graph is directed and with no negative cycles.

7.2.1 UPDATE(u,v)

We define the function `UPDATE(u,v)` as follows:

```
UPDATE(u, v)
    if (dist[v] > dist[u] + l(u,v)):
        dist[v] = dist[u] + l(u,v)
        prev[v] = u
```

There are some properties of `UPDATE` that we need to note:

Claim: Properties of UPDATE

1. If
   - dist$[u]$ is set to the correct shortest distance
   - The shortest path from $s \to v$ goes immediately after u.

   then calling `UPDATE(u,v)` sets dist$[v]$ to the correct shortest distance.
2. Calling `UPDATE(u,v)` "never hurts". We can never set dist$[v]$ to a value larger than the current shortest distance.
3. Shortest path from $s \to u$ for any u has at most $|V| - 1$ edges. Otherwise, there is a cycle that we can remove.
4. Say $s \to u_1 \to u_2 \cdots \to u_k \to v$ is the shortest path from $s \to v$. If we call `UPDATE`(s, u_1), `UPDATE`(u_1, u_2), ..., `UPDATE`(u_k, v) in this order, then dist$[v]$ is set to the correct shortest distance. The shortest path from $s \to t$ provides the shortest path for all intermediate vertices.

We define `SHORTEST_PATH(G, l, s)` as follows:

- Input: Graph $G = (V, E)$ directed with edge lengths $\{l_e : e \in E\}$ with no negative cycles, with vertex s.
- Output: for all $u \in V$ dist$[u]$ is set tot he shortest distance from $s \to u$.

```
SHORTEST-PATH(G, l, s)
    for all u in V:
        dist[u] = infinity
        prev[u] = null
    dist[s] = 0

    for i = 1 to |V| - 1:
        for all (u, v) in E:
            UPDATE(u, v)
```

Obviously, the running time of this algorithm is $O(|V||E|)$. One optimization to perform in this algorithm is to stop early if we go through an entire iteration of the outer loop without any updates. This means that we have already found the shortest path for all vertices.

7.2.2 How can we detect a negative cycle?

We can just run the loop one extra time to see if the distances change at all. If they do, then we know there is a negative cycle to exploit.

7.2.3 How can we optimize this algorithm?

Suppose we have no negative cycles. THen either we have no negative edges, or we have a DAG. We can build a more efficient algorithm for these two cases. The first case is just Dijkstra's algorithm. The second case is to just run a topological sort on the DAG, and then run `UPDATE(u,v)` for each edge in the topological order. This runs in $O(n+m)$ time.
`SHORTEST_PATH_DAG(G, l, s)`
Input: A directed acyclic graph $G = (V, E)$ with edge lengths $\{l_e : e \in E\}$, and a source vertex $s \in V$.
Output: For each vertex v reachable from s, dist$[v]$ is set to the length of the shortest path from s to v.

```
SHORTEST-PATH-DAG(G, l, s)
    for all u in V:
        dist[u] = infinity
        prev[u] = null
    dist[s] = 0

    G_T = topological sort of G
    For each u in G_T:
        for all (u,v) in E:
            UPDATE(u,v)
```

The key improvement now is that we only have to go through each edge once as opposed to $|V|-1$ times. This is because in a DAG, the shortest path from $s \to v$ may only go through the vertices that come before v in the topological order. Hence, we only need to go through each edge once. The overall running time is $O(n+m)$.

For graphs, DFS is used for connectivity and cycle detection, and BFS is used for shortest paths.

Chapter 8

Lecture 8: Greedy Algorithms

- **Goal:** Optimize some task involving multiple decision steps.
- **Greedy:** We take whatever seems optimal right now; hopefully the final solution is also optimal.
- **When?** When the locally optimal steps ensures a globally optimal solution.

8.1 Interval Scheduling

- **Input:** n jobs with start and end times $[s_i, t_i]$.
- **Task:** Schedule as many non-overlaping tasks as possible.

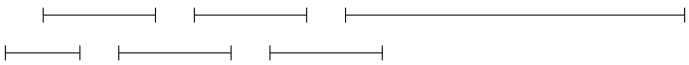


Figure 8.1: Interval Scheduling Example

We have several possible strategies

Strategies	Good idea?
Shortest first	No, the next smallest event could overlap two otherwise disjoint events
First start time	No, the first job could be very long and overlap may other smaller disjoint jobs
Last finish time	This mitigates the issue in the first two strategies.

Table 8.1: Interval Scheduling Strategies

This gives us the idea that always scheduling the job that finishes first gives us the most flexibility that we do not overlap with other jobs. It doesn't matter if we pick any other job other than the first job that ends because we'll always be able to fit in the same number of jobs after that.

Claim: Greedy Choice Property

Picking the job that finishes first is optimal

Proof. Suppose that our greedy algorithm returns us $[s_1, t_1], \ldots, [s_r, t_r]$ and the optimal solution is $[s'_1, t'_1], \ldots [s'_l, t'_l]$. We want to show that the greedy solution is the optimal solution, i.e. $r = l$.

Claim 1: $r \leq l$, this follows from the fact that l is the maximum number of jobs that can be scheduled.

Claim 2: $\forall i = 0, \ldots, r$, $H_i = [s_1, t_1], \ldots, [s_i][t_i], [s'_{i+1}, t'_{i+1}], \ldots, [s'_l, t'_l]$ is also an optimal solution. We prove this via induction. Note that H_0 is just the optimal solution. Assume that H_i is optimal. To show that H_{i+1} is optimal, we want to show that:

$$[s_1, t_1], \ldots, [s_{i+1}][t_{i+1}], [s'_{i+2}, t'_{i+2}], \ldots, [s'_l, t'_l]$$

is also optimal. To prove this, we must show that H_{i+1} must have the same number of jobs as H_i, and also the jobs in H_{i+1} are not overlapping. The first part is trivial, as we just replaced one job with another job. For the second part, we note that by the induction hypothesis, all other intervals are not overlapping with each other. Hence, we just have to show that $[s_{i+1}, t_{i+1}]$ does not overlap with $[s_i, t_1 i]$ and $[s'_{i+2}, t'_{i+2}]$. We see that because of how the greedy algorithm performs, we know that $[s_{i+1}, t_{i+1}]$ must finish before $[s'_{i+2}, t'_{i+2}]$ starts, as t_{i+1} must have finished before or at the same time as $[s'_{i+1}, t'_{i+1}]$ as it must finish before t'_{i+1}. Moreover, it's easy to see that $[s_{i+1}, t_{i+1}]$ does not overlap with $[s_i, t_i]$ as $[s_{i+1}, t_{i+1}]$ must start after t_i finishes. Thus, H_{i+1} is also optimal.

Claim 3: $r = l$. By induction, we have that H_r is optimal. But we argue that the greedy algorithm must have stopped at $r = l$. If $r < l$, then we can add another job to the greedy solution, which contradicts the fact that the greedy algorithm has stopped. □

We now imlement the interval scheduling algorithm:

```
INTERVAL-SCHEDULING(J)
    sort J by finish time
    A = []
    lastfinish = -infinity
    for j = 1 to n:
        if (lastfinish < sj)
            A.append(J[j])
            lastfinish = tj
    return A
```

This algorithm runs in $O(n \log n)$ time, as we have to sort the jobs by finish time first. The rest of the algorithm completes in linear time.

8.2 Compression (Huffman Codes)

- Input: Encode some text with T letters from an alphabet Γ with n letters and frequences $f_i : i \in \Gamma$.

$$\text{cost}(T) = \sum_i f_i \times \text{number of bits to represent } i \in T$$

Each letter in the alphabet Γ is represented by a binary string. We want to minimize the cost of representing the text T. Additionally, we want the encoding to be **prefix-free**, meaning that no encoding is a prefix of another encoding. This ensures that we can decode the text unambiguously.

8.2.1 Prefix-freeness

We want no code word to be a prefix of another codeword. We can represent the codewords as a binary tree, where each left edge represents a 0 and each right edge represents a 1. Furthermore, this tree must be a full binary tree, so any node either has 0 or 2 children. The codewords are represented by the leaves of the tree.

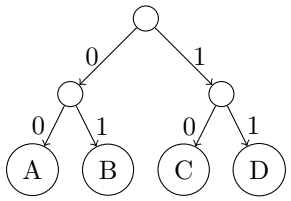


Figure 8.2: Prefix-free Code Example

In the above example, we have the following encodings:

- A: 00
- B: 01
- C: 10
- D: 11

We must come up with a greedy algorithm that allows us to minimize the cost of encoding the text T.

8.2.2 Greedy Huffman Encoding

We consider some possible greedy choices:

Strategies	Good idea?
Heaviest Codeword First	No, we want to make the heaviest codeword as short as possible.
Lightest Codeword First	Yes, we want to make the lightest codeword as long as possible.

Table 8.2: Huffman Encoding Strategies

Claim: Algorithmic Intuition

We list symbols in increasing order of frequency $f_1, \ldots, f_n$.

- Find the lowest two frequences f_i, f_j.
- Merge them into a new symbol with frequency $f_i + f_j$.
- Solve problem until there are only $n - 1$ symbols remaining.

We write the algorithm as follows:

```
HUFFMAN-CODE(f)
    H = makequeue(i, f_i for i in Gamma)
    For i = 1 to n - 1:
        (i1, f1) = DELETE_MIN(H)
        (i2, f2) = DELETE_MIN(H)
        newi = new symbol
        newf = f1 + f2
        INSERT(H, (newi, newf))
        add edges from newi to i1 and i2
    return H
```

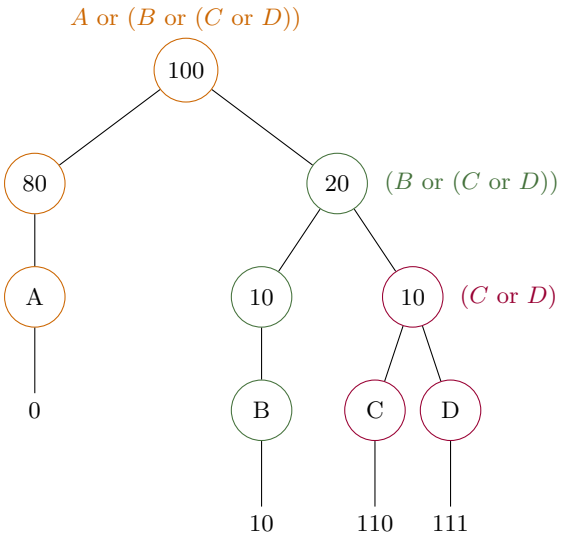


Figure 8.3: Example Huffman Tree

We now prove the correctness of the algorithm:

Claim

There exists an optimal tree T where f_i, f_j (the lowest two frequencies) are the deepest siblings.

Proof. Let O be an optimal tree. Let f_v be the deepest node in O, and f_u must have a sibling f_v. If f_i and f_j are somewhere else, then we can swap f_i with f_v and f_j with f_u to form tree M. This does not increase the cost of the tree, as $f_i, f_j \leq f_u, f_v$. Hence, we have found an optimal tree where f_i, f_j are the deepest siblings. Hence $\text{cost}(M) \leq \text{cost}(O)$. This is only possible when $\text{cost}(M) = \text{cost}(O)$.

□

We now prove the optimality of the greedy algorithm:

Lemma 8.2.1: Optimality of Huffman

The greedy algorithm produces an optimal tree.

Proof. We prove this via induction.
Base Case: $n = 2$. The greedy algorithm must merge the two nodes, which is obviously optimal.
Inductive Step: Assume that the greedy algorithm produces an optimal tree for n nodes. By the claim earlier, we know there exists an optimal solution tree O_{n+1} where the lowest two frequencies f_i and f_j are the deepest siblings. We know that $\text{cost}(O_{n+1}) = \text{cost}(O_n) + f_i + f_j$, where O_n is the optimal tree for the n nodes after merging f_i and f_j. Our Huffman algorithm places f_i and f_j as siblings and merges them. Our algorithm essentially calculates H_n on the other symbols including the symbol (f_i, f_j). We see that $\text{cost}(H_{n+1}) = \text{cost}(H_n) + f_i + f_j$. By the induction hypothesis, we know that $\text{cost}(H_n) = \text{cost}(O_n)$. Hence, $\text{cost}(H_{n+1}) = \text{cost}(O_{n+1})$. Thus, the greedy algorithm produces an optimal tree. □

Chapter 9

Lecture 9: Greedy Graphs

To show that a local optiaml choices leads to a global optimal solution, we need to show that the local optimal solutions applied iteratively leads to a global optimal solution. We need to show two properties:

- **Greedy Choice Property:** A global optimal solution can be arrived at by making a series of locally optimal (greedy) choices.
- **Optimal Substructure:** An optimal solution to the problem contains within it optimal solutions to subproblems.

Hence, in a proof for a greedy algorithm, we can start from an optimal solution and show that we can transform it into the greedy solution without increasing the cost.

We now consider the problem of **Minimum Spanning Tree**.

9.1 Minimum Spanning Tree (MST) - Kruskal and Prims

9.1.1 Tree and Graph properties

Definition 9.1.1: Tree

A **tree** is an undirected graph that is connected and acyclic.

There are several properties that we note about connected graphs:

Proposition 9.1.2

- **Property 1:** Removing a cycle edge e in a connected graph does not disconnect it.
- **Property 2:** A tree with n nodes has $n-1$ edges.
- **Property 3:** A connected graph on n nodes and $n-1$ edges is a tree.

Proof. To prove property 1, we prove that for all $u, v \in V$, in the graph $G' = (V, E-e)$, the graph is still connected. Take any path $u \to v \in G$. There are two cases:

- **Case I:** e is not on the path from u to v. Then the same path from u to v in G exists in G'.
- **Case II:** e is on the path from $u \to v$. However, because e is a cycle edge, there exists another path between the two endpoints of e. We may then construct a new path from u to the first endpoint of e, and then take the path to the other endpoint of e (essentially going around the cycle), and then go to v. This path exists in G'.

We now prove property 2. We note that if we start with an graph with n nodes and no edges, we initially have n connected components. Each time we add an edge that does not form a cycle, we reduce the number of connected components by 1. Once we have added $n-1$ such edges, the number of connected components becomes 1, which means the graph is connected without a cycle. Hence, the graph is a tree.

We now prove property 3. We are given that the graph is connected, we must show that it is acyclic. Assume for the sake of contradiction that G has a cycle. Then by property 1, we may remove an edge in the cycle so that the graph would still remain connected and have $n-2$ edges. Since we have $n-2$ edges and n nodes, the graph must have at least 2 connected components, a contradiction. □

9.1.2 Minimum Spanning Tree

Definition 9.1.3: Spanning Tree

A spanning tree $T = (V, E')$ of a graph $G = (V, E)$ where $E' \subseteq E$is a tree such that $weight(T) = \sum_{e \in E'} w_e$ is minimized.

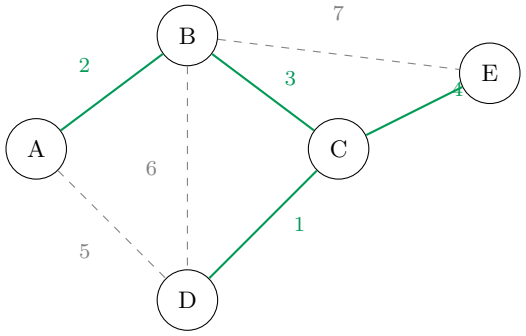


Figure 9.1: Spanning Tree

The greedy approach to finding a minimum spanning tree is as follows: Add the least weight edge that does not intrudce a cycle, and interate. Our main theorem is:

Theorem 9.1.4: MST Theorem

1. Let $X \subseteq E$ be part of some MST of G.
2. Let $S \subseteq V$ be a set such that there are no edges in X from S to $V - S$
3. Let $e \in E$ be the lightest edge from S to $V - S$.

Then $X \cup \{e\}$ is part of some MST of G.

To visualize property 2 consider:

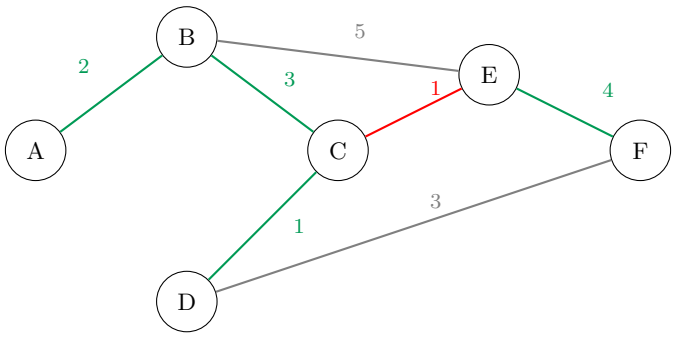


Figure 9.2: S is the vertices A,B,C,D and V-S is the vertices E,F. The red edge is the lightest edge from S to V-S. The green edges are part of X. There are no edges in X from S to V-S.

We now prove the MST theorem:

Proof. Case I: Suppose that $e \in T$, where T is some MST that contains X. Then we are done, as $X \cup \{e\} \subseteq T$.

Case II: Suppose that $e \notin T$. We may perform an exchange argument. Suppose that some other edge f crosses S and $V - S$ and $w_e \leq w_f$, and $f \in T$.
Part 1: Consider $T' = T - \{f\} + \{e\}$. We see that $\text{cost}(T') = \text{cost}(T) - w_f + w_e$. Since $w_e \leq w_f$, we have that $\text{cost}(T') \leq \text{cost}(T)$. Since, T is an MST, $\text{cost}(T') = \text{cost}(T)$.

Part 2: It remains to show that T' is a tree. We know that T is a tree. By property 1, removing f from $T + e$, shows us that T' is still connected. Moreover, T' has $n - 1$ edges, as we added and removed an edge. Since T' is connected and has $n - 1$ edges, by property 3, T' is a tree.

Part 3: We show that $X + e \subseteq T'$. We see that $f \notin X$, hence, $X + e \subseteq T + e - f = T'$. (Key point is that X is a subset of $T - f$, as we assume that there are no edges in X from S to $V - S$).

Thus, T' is an MST that contains $X \cup \{e\}$ □

We now implement Kruskal's algorithm:

9.2 Kruskal's Algorithm

Definition 9.2.1: Kruskal's Algorithm

- Go over edges in order of increasing weights
- Add edge if it doesn't introduce a cycle and skip otherwise.

Claim: Kruskal's Algorithm is correct

Kruskal's finds returns an MST.

Proof. We perform induction ove the number of added edges.

Base Case: 0 edges have been added. The empty set is part of some MST.

Inductive Step: Assume that after k edges have been added, the set of edges X is part of some MST. We want to show that after adding the $k+1$-th edge e, $X \cup \{e\}$ is part of some MST. We see that e is the lighest edge that does not form a cycle with X. Let S be the set of vertices reachable from some vertex in e using only edges in X. We see that there are no edges in X from S to $V - S$, as otherwise, adding e would form a cycle. By the MST theorem, we have that $X \cup \{e\}$ is part of some MST. □

9.2.1 Union-Find Data Structure

To implement Kruskal's algorithm efficiently, we need a data structure that efficiently keeps track of connected components and also finds if an edge introduces a cycle.

Definition 9.2.2: Union-Find Data Structure

- `Makeset(x)`: Creates a singleton set containing element x.
- `Find(x)`: Find the set x belongs to.
- `Union(x,y)`: Merge the sets containing x and y.

Whenever we add a new edge, we connect the connected component that the new edge connects into one new connected component. Every time we consider an edge, we see if the endpoints of the edge already are connected.

To implement the Union Find Data Structure, we store each set as a tree labeled by its root. We define:

- $\pi(x)$: parent of x.
- rank(x): height of the tree rooted at x.

Now for the implementation:

```
MAKESET(x)
    pi(x) = x
    rank(x) = 0
FIND(x)
    if (x!=pi(x)):
        FIND(pi(x))
    else:
        return x
UNION(x,y)
    rx = FIND(x)
    ry = FIND(y)
    if (rx == ry):
        return
    if (rank(rx) > rank(ry)):
        pi(ry) = rx
    else if (rank(ry) > rank(rx)):
        pi(rx) = ry
    else:
        pi(ry) = rx
        rank(rx) = rank(rx) + 1
```

For the implementation of `UNION` we want to make the tree as shallow as possible so that `FIND` runs quickly. Hence, we always attach the shorter tree to the taller tree. This ensures that the height of the tree is at most $\log |V|$. This is because every time we increase the height of a tree, we must have two trees of the same height. Hence, a tree of height h must have at least 2^h nodes. Thus, the height of the tree is at most $\log |V|$. Since `FIND` runs in time proportional to the height of the tree, `FIND` and `UNION` both run in $O(\log |V|)$ time.

Claim: Union-Find Analysis

`MAKESET` runs in $O(1)$ time, and `FIND` and `UNION` both run in $O(\log |V|)$ time.

Proof. `MAKESET` obviously runs in $O(1)$ time. We claim that if $\text{rank}(x) = r$, then x has $\geq 2^r$ nodes in the tree rooted at r. We prove this via induction on r.

Base Case: $r = 0$. Then x is a singleton, which has 1 node.

Inductive Step: Assume that the claim is true for r. The only case for r to increase is when we perform a union of two trees of rank r. By the induction hypothesis, each tree has at least 2^r nodes. Hence, the new tree has at least $2^r + 2^r = 2^{r+1}$ nodes. Thus, the claim is true.

Hence, the max rank of any root is $\leq \log n$. □

The union-find data structure could actually be further optimized with path compression,

where every time we call find or union, we automatically set the parent of children nodes to the root of the set.

9.2.2 Kruskal's Algorithm Implementation

```
KRUSKAL(G)
    for all v in V:
        Makeset(v)
    sort edges E by weight
    T = []
    for all (u,v) in E:
        if (Find(u) != Find(v)):
            T.append((u,v))
            Union(u,v)
    return T
```

9.2.3 Kruskal's Algorithm Analysis

We see that the sorting the edges takes $|E| \log |E| = |E| \log |V|$ time. We perform at most $2|E|$ finds and $|V| - 1$ unions. Find and union both run in $\log |V|$ time, and makeset runs in $O(1)$ time. Hence, the overall running time is $O((|V| + |E|) \log |V|)$.

9.3 Prim's Algorithm

Prim's algorithm runs analogous to Dijkstra's algorithm. We start form some vertex s and grow the MST one edge at a time. At each step, we add the lightest edge that connects a vertex in the MST under construction to the rest of the graph. This uses the cut property that we proved earlier, that the lighest weight edge crossing a cut must be in some MST.

Definition 9.3.1: Prim's Algorithm

- Start from some vertex s.
- Grow the MST one edge at a time.
- At each step, add the lightest edge that connects a vertex in the MST under construction to the rest of the graph.

```
PRIM(G,s)
    for all v in V:
        cost(v) = infinity
        prev(v) = null
    key(s) = 0
    for all v in V - {s}:
```

```
        INSERT(Q,v,cost(v))
    while (Q not empty):
        v = DELETE_MIN(Q)
        for all (u,v) in E:
            if (cost(u) > w(u,v)):
                prev(u) = v
                cost(u) = w(u,v)
                Decrease_Key(Q,u,cost(u))
    pi = []
    for all v in V - {s}:
        pi.append((prev(v), v))
    return pi
```

We see that there are $O(|V|)$ inserts and deletes, and $O(|E|)$ decrease keys. Each operation takes $O(\log |V|)$ time. Hence, the overall running time is $O((|V| + |E|) \log |V|)$.

Chapter 10

Lecture 10: Horn-SAT and Dynamic Programming

10.1 Horn-SAT

Let $x_1, x_2, \ldots, x_n$ be n boolean variables (can be TRUE/FALSE). A clause c_i is either $(\bar{X}_1, \vee \bar{X}_2 \vee \ldots \vee \bar{X}_k)$ or $(X_1 \wedge X_2 \wedge \ldots \wedge X_k \rightarrow X_{k+1})$. We're given an input of clauses $c_1, \ldots c_m$ We want to check if $F = c_1 \wedge c_2 \wedge \ldots \wedge c_m$ is satisfiable. This is called the Horn-SAT problem.
This problem can be solved greedily as such:

- Start by assigning all variables to FALSE
- Switch a variable to TRUE if it must be set so in any satisfying assignment.

```
HORN(F)
    for all x_i:
        x_i = FALSE
    While there exists an unsatisfiable implication clause C,
        set the right hand varialbe in C to TRUE
    if (all pure negative clauses are satisfied):
        return assignment x_i ... x_n
    else:
        return UNSATISFIABLE
```

The runtime of the algorithm is $O(n|F|)$, where $|F|$ is the size of the input formula, and n is the number of variables. This is because we may have to set each variable to TRUE at most once, and each time we set a variable to TRUE, we may have to check all the clauses to see if they are satisfied. Hence, the overall running time is $O(n|F|)$.

10.1.1 Correctness of Horn-SAT

We first prove a lemma.

Lemma 10.1.1: Horn-SAT Invariant

If `HORN(F)` sets a variable x = TRUE, then this is TRUE in all satisfying assignments of F.

Proof. If a variable is ever set to TRUE, it must be because there is an implication clause C that is unsatisfied. This means that all the variables on the left hand side of C are FALSE, and the variable on the right hand side is FALSE. We prove the lemma by induction:

Base Case: The first variable x that is set to TRUE must be TRUE in all satisfying assignments. This is because all the variables on the left hand side of C are FALSE, so the variable on the right hand side must be TRUE in any satisfying assignment.

Inductive Step: Assume that the first k variables set to TRUE are TRUE in all satisfying assignments. Consider the $k+1$-th variable x_{k+1} that is set to TRUE. The only reason this variable may be set to true is if the implication clause C that sets x_{k+1} to TRUE is unsatisfied. This means that all the varialbes on the left hand side of C are TRUE, and x_{k+1} is FALSE. By the induction hypothesis, all the variables on the left hand side of C are TRUE in all satisfying assignments. Hence, x_{k+1} must be TRUE in all satisfying assignments. Thus, by induction, the lemma is true. □

Theorem 10.1.2: Horn-SAT Correctness

If F is satisfiable, then the algorithm returns a satisfying assignment.

Proof. **Case 1:** `HORN(F)` outputs as assignment. Then that assignment passed both the implication clauses and the prue negative clauses, and thus is satisfying.

Case 2: `HORN(F)` outputs UNSATISFIABLE. By our Lemma, we only set a variable to TRUE if this is required in every satisfying assignment. The pure-negative clauses must be unsatisfied. Hence, F has no satisfying assignment. □

10.1.2 Improving the Runtime of Horn-SAT

The idea to optimize Horn-SAT is that we don't need to recompute F every time a varialbe is set to TRUE. We may map any HORN-SAT problem to a graph. This graph van be described by the vertex set V and the edge set E:

- The vertex set V is the set of variables $\{x_1, x_2, \ldots, x_n\}$, and all the clauses $\{c_1, c_2, \ldots, c_m\}$.
- We create an edge from a variable to a clause if the variable is on the left hand side of the clause.

Thde idea is that if a clause has in-degree 0, then it is unsatisfied, and we must set the variable on the right hand side of the clause to TRUE. We may then remove all edges from this variable to any clauses it is on the left hand side of. This may cause other clauses to have in-degree 0, and we may repeat this process until there are no clauses with in-degree 0. We may implement this as follows:

```
HORN-IMPROVED(F)
    Construct graph G=(V,E) from F
    Q = []
    find nodes of in degree 0 and add them to Q, set the variable to TRUE
    while (Q not empty):
        remove x from q
        Delete edges coming out of x
        For any node with in degree - 0, add it to Q and set varialbe to TRUE
```

The number of edges is $O(|F|)$, and there may be up to n vertices in the queue. Hence, the overall running time is $O(|F| + n)$.

10.2 Dynamic Programming

We provide a high-level framework of how Dynammic Programming works:

- Define appropriate subproblems
- Write a recurrence relation
- Deterine order of computation of subproblems, which induces a DAG structure.

Small sub-problems are solved first, this allows us to solve larger sub-problems without recomputing the smaller sub-problems. We create a DAG structure of the subproblems, and a topological sort of those subproblems gives us the order of computation.

10.2.1 Longest Path in a DAG

- **Input:** $G = (V, E)$ a DAG
- **Output:** The longest path in G.

Step 1: Defining the Subproblems:

The largest challenge of dynammic programming is identifying the subproblems. The recurrence relation tells us how to combine subproblems to solve larger problems.

We define the subproblems as follows.

$$L(v) = \text{length of longest path from some source to } v$$

Our solution is simply $\max_{v \in V} L(v)$.

Step 2: Connecting the Subproblems - Recurrece Relation:

Now to combine the subproblems, we create our recurrence relation as follows:

$$L(v) = \max_{(u,v) \in E} L(u) + 1$$

If v has no incoming edges, then $L(v) = 0$. A naive recursive algorithm would look like the following.

```
LONGEST-PATH(G)
    If there are no incoming edges to v:
        return 0
    else:
        return max_{(u,v) in E} LONGEST-PATH(u) + 1
```

Correctness-wise, this algorithm would work. However, the running time could be exponential, as we may recompute the same subproblems multiple times.

Step 3: Order of Computation - Topological Sort:
We introduce the idea of memoization, where we don't recompute subproblems that have already been computed. For this problem, we may accomplish this by performing a topological sort of the DAG. We then compute $L(v)$ in the order of the topological sort. This ensures that when we compute $L(v)$, all the $L(u)$ for $(u, v) \in E$ have already been computed. The algorithm is as follows:

```
LONGEST-PATH-DAG(G)
    Topologically sort G to get ordering v_1, v_2, ..., v_n
    for i = 1 to n:
        if (v_i has no incoming edges):
            L(v_i) = 0
        else:
            L(v_i) = max_{(u,v_i) in E} L(u) + 1
    return max_{v in V} L(v)
```

The run-time of this algorithm is $O(|V|+|E|)$, as we perform a topological sort in $O(|V|+|E|)$ time, and then we compute $L(v)$ for each vertex v. Computing $L(v)$ takes time proportional to the in-degree of v. Hence, the overall running time is $O(|V| + |E|)$.

10.2.2 Longest Increasing Subsequence (LIS)

- **Input:** $A = [a_1, a_2, \ldots, a_n]$ a sequence of n numbers.
- **Output:** The length of the longest increasing subsequence of A.

This problem just reduces to finding the longest path in a DAG. We create a graph $G = (V, E)$ as follows:

- $V = \{1, 2, \ldots, n\}$
- $E = \{(i, j) | i < j \text{ and } a_i < a_j\}$

We see that G is a DAG, as if there was a cycle, then there would be an edge (j, i) for some $j > i$, which is a contradiction.

10.2.3 Edit Distance

- **Input:** Two strings $A = a_1 a_2 \ldots a_n$ and $B = b_1 b_2 \ldots b$.
- **Output:** The minimum number of operations to convert A to B. The allowed operations are:

- Insertion of a character
- Deletion of a character
- Substitution of a character

Step 1: Defining the Subproblems: We define a subproblem for this as the number of operations to convert the first i letters of a word into the first j letters of another word. We define:

$$d(i,j) = \text{Minimum number of operations to convert } a_1a_2\dots a_i \text{ to } b_1b_2\dots b_j$$

By default, we know that $d(0,j) = j$ and $d(i,0) = i$. These will serve as our base cases.

Step 2: Connecting the Subproblems - Recurrence Relation:
We now create our recurrence relation. We see that there are 3 cases, either we want to add a character, delete a character, or substitute a character. We see that:

- In the case of a deletion, we must have deleted a_i. Hence, $d(i,j) = d(i-1,j) + 1$.
- In the case of an insertion, we must have inserted b_j. Hence, $d(i,j) = d(i,j-1) + 1$.
- In the case of a substitution, we must have substituted a_i for b_j. Hence, $d(i,j) = d(i-1,j-1) + 1$.

Step 3: Order of Computation - DAG Structure:
We see that $d(i,j)$ depends on $d(i-1,j)$, $d(i,j-1)$, and $d(i-1,j-1)$. Hence, we may create a DAG structure where there is an edge from $d(i-1,j)$ to $d(i,j)$, an edge from $d(i,j-1)$ to $d(i,j)$, and an edge from $d(i-1,j-1)$ to $d(i,j)$. We see that this is a DAG, as all edges go from a node with smaller $i+j$ to a node with larger $i+j$. We may then perform a topological sort of this DAG, and compute $d(i,j)$ in the order of the topological sort. The algorithm is as follows:

```
EDIT-DISTANCE(A,B)
    n = length(A)
    m = length(B)
    for i = 0 to n:
        d(i,0) = i
    for j = 0 to m:
        d(0,j) = j
    for i = 1 to n:
        for j = 1 to m:
            d(i,j) = min(d(i-1,j) + 1, d(i,j-1) + 1, d(i-1,j-1) + 1)
    return d(n,m)
```

The running time of this algorithm is $O(nm)$, as we compute $d(i,j)$ for all $1 \le i \le n$ and $1 \le j \le m$. Each computation takes $O(1)$ time.

Chapter 11

Lecture 11: Dynamic Programming II

Dynamic programming defines a set of sub-problems that when assembled, gives the full solution. Subproblems may be represented in a DAG structure, and a topological sort of the DAG gives the order of computation of the subproblems.

The runtime of a dynamic programming solution corresponds to the number of subproblems times the time to compute each subproblem.

11.1 Knapsack Problem

- **Input:** Total weight capacity of a knapsack W, and a list of n items where item i has weight w_i and value v_i.
- **Output:** Set of items that will maximize total value with total weight $\leq W$.

There are two variations of this problem, **knapsack with repetition** and **knapsack without repetition**. In knapsack with repetition, we may take an item multiple times, whereas in knapsack without repetition, we may only take an item once.

The knapsack problem might seem like it could be solved greedily, but it cannot. This is because taking the item with the highest value/weight ratio may not lead to the optimal solution, as we might have deadweight loss. (unused weight)

11.1.1 Knapsack with replacement

We define our DP solution as follows:

1. **Subproblem:** let $K(C)$ be the max value that can be packed into a knapsack of capacity C.
2. **Base case:** $K(0) = 0$.
3. **Recurrence relation:** $K(C) = \max_{i:C \geq w_i} K(C - w_i) + v_i$.

4. **Representing as a DAG:** $K(C)$ depends on $K(C - w_i)$ for all i such that $C \geq w_i$. Hence, we may create a DAG where there is an edge from $K(C - w_i)$ to $K(C)$ for all i such that $C \geq w_i$. This is a DAG, as all edges go from a node with smaller capacity to a node with larger capacity.

```
KNAPSACK-WITH-REPLACEMENT(W,w,v)
    K(0) = 0
    for C = 1 to W:
        K(C) = max_{i:w_i <= C} K(C - w_i)
    return K(W)
```

The running time of the algorithm is $O(nW)$, as we compute $K(C)$ for all $1 \leq C \leq W$. Each computation takes $O(n)$ time.

11.1.2 Knapsack without replacement

Now that we cannot have duplicate items, we must modify our subproblem to account for what values we already have in our knapsack. We don't know what items have already been added. We want to add this information into our subproblem, somehow not including all the possible items that can fit in a knapsack of capacity C. We my define our DP solution as follows:

1. **Subproblem:** let $K(C, j)$ be the max value that can be packed into a knapsack of capacity C using only items $1, 2, \ldots, j$.

2. **Base case:** $K(C, 0) = 0$ for all C, and $K(0, j) = 0$ for all j.

3. **Recurrence relation:**

$$K(C, j) = \begin{cases} K(C, j-1) & \text{if } w_j > C \\ \max(K(C, j-1), K(C - w_j, j-1) + v_j) & \text{if } w_j \leq C \end{cases}$$

   At each step, either we don't take item j, in which case the max value is $K(C, j-1)$, or we take item j, in which case the max value is $K(C - w_j, j-1) + v_j$. w_j could also not fit in the knapsack, in which case we cannot take item j.

4. **Representing as a DAG:** $K(C, j)$ depends on exactly two subproblems, $K(C, j-1)$ and $K(C - w_j, j-1)$.

```
KNAPSACK-WITHOUT-REPLACEMENT(W,w,v)
    for C = 0 to W:
        K(C,0) = 0
    for j = 0 to n:
        K(0,j) = 0
    for C = 1 to W:
        for j = 1 to n:
            if (w_j > C):
                K(C,j) = K(C,j-1)
```

```
            else:
                K(C,j) = max(K(C,j-1), K(C-w_j, j-1) + v_j)
    return K(W,n)
```

The running time of the algorithm is $O(nW)$, as we compute $K(C, j)$ for all $1 \leq C \leq W$ and $1 \leq j \leq n$. Each computation takes $O(1)$ time.

We can save space further by not storing the entire K table, but only the last two columns. This is because $K(C, j)$ only depends on $K(C, j-1)$ and $K(C - w_j, j-1)$. Hence, we only need to store the last column to compute the current column. This reduces the space complexity from $O(nW)$ to $O(W)$.

11.1.3 Memoization

Memoization is a top-down approach to dynamic programming. We start with the full problem, and recursively break it down into subproblems. Whenever we compute a subproblem, we store the result in a table. If we ever need to compute the same subproblem again, we simply look up the result in the table instead of recomputing it. This is useful when the number of subproblems is small, but the number of ways to reach each subproblem is large.

In our knapsack problem, if the gap between the weights are too small, then we may compute meaningless/useless subproblems a lot. Memoization avoids that.

11.2 Chain Matrix Multiplication

- **Input:** A sequence of matrices $A_1, A_2, \ldots, A_n$ where A_i has dimensions $p_{i-1} \times p_i$.
- **Output:** The best parenthesization of the product that minimizes the number of scalar multiplications.

We define our DP solution as follows:

1. **Subproblem:** let $M(i, j)$ be the minimum number of multiplications needed to multiply $A_i \times A_{i+1} \times \ldots \times A_j$.
2. **Base Case:** $M(i, i) = 0$ for all i.
3. **Recurrence Relation:**

$$M(i, j) = \min_{i \leq k \leq j} M(i, k) + M(k+1, j) + p_{i-1} p_k p_j$$

   This captures all the possible parenthesizations of the product.

4. **Representing as a DAG:** We see that we're not concerned with subproblems where $j < i$. We want our DAG to always have smaller subproblems point to larger subproblems. We see that $M(i, j)$ depends on $M(i, k)$ and $M(k+1, j)$ for all $i \leq k \leq j$. We see that $M(i, k)$ has $k - i < j - i$ and $M(k+1, j)$ has $j - (k+1) < j - i$. Hence, we may create a DAG where there is an edge from $M(i, k)$ to $M(i, j)$ for all $i \leq k < j$, and an edge from $M(k+1, j)$ to $M(i, j)$ for all $i \leq k < j$. This is a DAG, as all edges go from a node with smaller $j - i$ to a node with larger $j - i$.

The algorithm is:

```
MATRIX-CHAIN-MULTIPLICATION(p)
    n = length(p) - 1
    for i = 1 to n:
        M(i,i) = 0
    for l = 2 to n - 1: // l is the chain length
        for i = 1 to n-s:
            j = i + s
            M(i,j) = min_{i <= k < j} {M(i,k) + M(k+1,j)} + p_{i-1}p_kp_j
    return M(1,n)
```

The running time of the algorithm is $O(n^3)$, as we compute $M(i, j)$ for all $1 \leq i \leq j \leq n$.

Chapter 12

Lecture 12: Dynammic Programming III

The DAG shortest path problem we solved in previous lectures is at its core a DP solution. We relaxed edges in topological order.

12.1 SSSP problem revisited

- **Input:** A weighted graph with possibly negative edge weights without negative cycles, and a source vertex v.
- **Output:** For every v, set dist$[v]$ to the shortest path from $s \to v$.

A natural intuition for the SSSP problem might be as follows

1. **Subproblem**: dist$[v]$ = shortest path from $s \to v$.
2. **Recurrence Relationship**: dist$[v] = \min_{(u,v)\in E}\{\text{dist}(s,u) + w_{u,v}\}$
   (a) **Base Case**: dist$[s] = 0$.

However, the failure condition for this is that we cannot find a DAG structure for the DP solution. Consider a directed graph with a cycle, then we would have cyclic dependencies. To resolve this, we may introduce a parameter into the graph structure to destroy this cyclic property.

12.1.1 Coping with cycles in SSSP

We define:

$$\text{dist}(v,k) = \text{shortest distance from } s \to v \text{ taking at most } k \text{ edges}$$

Furthermore, our base case is now:

$$\text{dist}(s,0) = 0 \quad \text{dist}(v,0) = \infty \quad \forall v \neq s$$

We can now formulate a better recurrence relation for our problem that should hopefully avoid the cycle issue. To solve for $\text{dist}(v, k)$ we can look at all the incoming edges into v and take the path that takes at most $k-1$ edges to get to those neighbors from s, and any paths that to v that takes less than k paths.

$$\text{dist}(v, k) = \min\{\text{dist}(v, k-1), \min_{u,v \in E}\{\text{dist}(u, k-1) + w_{u,v}\}\}$$

Now, we just need to compute for all v:

$$\text{dist}(v, 0); \text{dist}(v, 1) \cdots \text{dist}(v, n-1)$$

This effectively creates n copies of our original graph, with edges between the copies denoting the transitions between subproblems. This effectively blocks out any of the cycles in our old graph as we are no longer concerned with them by how we defined our new subproblem structure. **Changing the subproblem definition would change the underlying subproblem graph structure. We want to change it so that it resembles a DAG.**

Algorithm:

```
For v in V dist(v,0) = infty
dist(s,0) = 0
for all k = 1 ... n - 1:
    for all v in V:
        dist(v,k) = dist(v,k-1)
        for all (u,v) in E:
            if dist(v,k) > dist(u, k - 1) + w_{u,v}:
                dist(v,k) = dist(u, k - 1) + w_{u,v}
```

The overall runtime of this algorithm is $O(n(n+m)) = O(nm)$. Performance may be slightly slower than Bellman-Ford because Bellman-Ford may skip certain edges that DP processes.

12.2 Single Source Reliable Shortest Path

- **Input:** A weighted graph $G = (V, E)$ with weights for each edge w_e. A source s. and a bound B.
- **Output:** For every v, the shortest path from s to v taking at most B edges.

The algorithm covered before can be extended to solve this problem

12.3 All pairs shortest path (Floyd-Warshall)

- **Input:** A weighted graph $G = (V, E)$ with weights for each edge w_e.
- **Output:** For every $u, v, \text{dist}(u, v) =$ shortest path from u to v.

The naive way to do this is to just run Bellman-Ford or DP variant for every $u \in V$. This takes $O(n \times nm) = O(n^2 m) \approx O(n^4)$. This is quite expensive!
A better solution would be to take advantage of our DP solution with caches the relevant distances. We can make our subproblems even more general. Define:

$$\text{dist}(u, v, k) = \text{shortest path from } u \to v \text{ taking at most } k \text{ edges}$$

However, this takes at least $O(n^3)$ time, because we have $O(n^3)$ possible subproblems to solve. More specifically, this DP modification still gives us $O(n^2 m)$ performance. We may modify the definition of our subproblems to imrpove the DP performance. Consider the subproblem:

$$\text{dist}(u, v, k) = \text{shortest path from } u \to v \text{ that only takes vertices in } \{1 \dots k\}$$

This is similar to the knapsack problem, where we are trying to solve the problem with a restriction on the elements that we may interact with. Our base case can be defined as:

$$\text{dist}(u, v, 0) = w_{u,v}, \infty \text{ otherwise}$$

Proposition 12.3.1

On the shortest path from $u \to v$ no vertex w happens twice.

Proof. If w is visited twice, then there is a cycle on w in the path from $u \to v$. We can remove this cycle to get a shorter path, a contradiction. □

We are now prepared to write our recurrence relationship. $\text{dist}(u, v, k)$ can either involve k or not. If this distance does not involve k, then we just have $\text{dist}(u, v, k-1)$. Otherwise, there is some shortest path from $u \to v$ that goes through k. Hence, there is a shortest path from $u \to k$ using only vertices in $\{1 \dots k-1\}$ and a shortest path from $k \to v$ using only vertices in $\{1 \dots k-1\}$. Hence, we just need to take the minimum of the two cases.

$$\text{dist}(u, v, k) = \min\{\text{dist}(u, v, k-1), \text{dist}(u, k, k-1) + \text{dist}(k, v, k-1)\}$$

Now the in-degree of each subproblem is only 3 instead of the degree of a given vertex. Because there are $O(n^3)$ subproblems, and each subproblem only relies on 3 subproblems, this new algorithm has a runtime-complexity of $O(n^3)$.

12.4 Travelling Salesman Problem

- n cities and distance d_{ij} $i \neq j$
- Find minimum distance path from city 1 back to city 1 visitng each city exactly once.

From CS70, we know that this problem reduces down to finding a Hamiltonian path of the graph G, and then finding a minimum path. The brute force solution is just to consider every path from 1 back to itself that visits all vertices. There could be up to $n!$ different

permutations for these hamiltonian paths. Using DP, we can construct a n^n runtime solution, which is asymptotically the same as $n!$.

We define:

$$C(j) = \text{cost of the minimum path from } i \to j \text{ that visits all nodes exactly once}$$

This subproblem does not expose enough information for us to work with it. Consider:

$C(j,k) = \text{cost of the minimum path from } i \to j \text{ that visits all nodes in } 1 \dots k \text{ exactly once}$

This subproblem still does not expose enough information. The set $1 \dots k$ is too rigid of a set. Now consider:

$$C(j,S) = \text{cost of the minimum path from } i \to j \text{ that visits all nodes in } S \text{ exactly once}$$

Where S is an arbitrary set.

Chapter 13

Lecture 13: Dynamic Programming

We revisit the traveling salesman problem. To simplify the problem, we start from city 1 but we can end at any other city while visiting all the vities.

13.1 Traveling Salesman

13.1.1 Defiing a subproblem

We define our subproblem as:

$$C(S, j) = \text{least cost path from } 1 \to j \text{ that visits all nodes in } S \text{ once.}$$

j is part of S. Hence, our base case would be when $|S| = 1$. Hence, we have:

$$C(\{1\}, 1) = 0 \quad C(S, 1) = \infty$$

The second condition is because we cannot visit 1 twice, as we start at 1. This behavior is not allowed, so we set it to infinity so as to not disrupt our recurrence relation. We see that our recurrence location look like:

$$C(S, j) = \min_{i \in S, i \neq j} [C(S \setminus \{j\}, i) + d_{ij}]$$

13.1.2 Solving the solution

Our solution right now only finds the shortest path from $1 \to j$ for any j given any set of vertices. The real solution to the TS problem is:

$$\min_j \{C(\{1, \ldots, n\}, j) + d_{j1}\}$$

```
TS()
    for s = 2 to n
        for all subsets S of sizes and containing 1:
            C(S,1) = infty
            for all j in S, j != 1
                C(S,j) = minimum of i in S i!=j of C(S-j, i) + dij
    return min_j {C({1,...,n}, j) + d_{j1}}
```

The runtime of this is $O(n^2 2^n)$, as can be seen from stacking the loops.

13.2 Independent Set

Definition 13.2.1: Independent Set

Given a grahh $G = (V, E)$, an independent set is a set $I \subseteq V$ such that $\forall u, v \in I$ we have that $(u, v) \notin E$.

- **Input:** Graph $G = (V, E)$.
- **Output:** $\text{Ind}(G) = \max_{I \subseteq V}\{|I| : I \text{ is an independent set}\}$

This problem is hard, so we'll consider a simpler problem.

13.2.1 Maximal Independent Set for Tree

- **Input:** Tree $G = (V, E)$.
- **Output:** $\text{Ind}(G) = \max_{I \subseteq V}\{|I| : I \text{ is an independent set}\}$

The intuition is that for creating I, we cannot put direct parents and childrens together. We define our subproblem as:

$$I(v) = \text{Size of the maximal independent set of the sub-tree rooted at } v$$

1If v happens to be the root of the tree, then $I(v)$ is simply our solution.

The base case occurs when v has no children. Then we have that $I(v) = 1$. If v is not a root node, we may define an recurrence relation for $I(v)$:

$$I(v) = \max\{\sum_{u \in C(v)} I(u), 1 + \sum_{u \in G(v)} I(w)\}$$

Where $C(v)$ is the set of the children of v, and $G(v)$ is the set of grand children of v.

```
\underline{Ind(G):}
    Topologically sort V such that p(v) > v for all v != s
    for all v in V, if p(v) != v then add v to C(p(v))
                    if p(p(v)) != p(v) then add v to G(p(p(v)))
    for all v in V if C(v) = null then I(v) = 1
    for all v in V
```

```
        I(v) = max{sum_{w in C(v)} I(w), 1 + sum_{w in G(v)} I(w)}
    output I(r)
```

Topological sort takes $O(n)$ work and setting grand parents and parents data also takes $O(n)$ work. Establishing base case also takes $O(n)$ time and the reccurence also takes $O(n)$ work. This algorithm takes $O(n)$ work in total.

13.3 Examples Where DP Works but Greedy Doesn't

- **Input:** $x_1, \ldots, x_n; V$
- **Goal:** Possible to make change of v using the given coins? Find minimum # of coins needed.

The greedy solution is not optimal for all choices. Hence, we must rely on dynammic programming. We define the subproblem as:

$$K(v) = \text{min number of coins needed to give change for } v, \text{ set to } \infty \text{ if not possible}$$

For our recurrence relation, we have:

$$K(v) = \min\{\min_{i:X_i \leq v}\{K(v-c)+1\}, \infty\}$$

Our base case is simple, it is just $K(0) = 0$.

13.4 Task Scheduling Problem with a Twist

- n jobs with start and end times, with a weight w_i for each job
- Schedule non-overlapping tasks such that the weight of the tasks is maximized.

When we previously had to solve the problem by considering only scheduing as many on-overlapping tasks as possible, we used a greedy solution. However, we now have to keep track of the weights of the jobs.

13.4.1 Solution

1. We first sort the jobs by their finish times.
2. We define $p(i) =$ the last job that can be scheduled if the i job has been scheduled.
3. Define OPT(i) =optimal weight obtained by scheduling jobs $1, \ldots, i$ only.

Now to define the recurrence relation, we note that:

$$\text{OPT}(i) = \max(\text{OPT}(i-1), w_i + \text{OPT}(p(i)))$$

Our base case is $OPT(0) = 0$ or $OPT(1) = w_1$.

13.5 Review of DP

We have now encounted many forms of DP problems:

- Longest Path in DAG: explicit DAG
- Edit distance: grid Graph
- Matrix chain Multiplication: Interval subproblems
- All-Pair-Shortest-Path: DP with "allowed intermediate vertices (states)"
- Traveling Salesman: Subset subproblems. Arbitrary subsets.
- Knapsack with and without replacement (coin denomination): Subproblems based on capacity
- Independent Set: DP on a tree structure.
- Weighted Task Scheduling:

Chapter 14

Lecture 14: Linear Programming

Linear programming is a mathematical way of optimizing for the best outcome in a linear model (maximizing products or minimizing costs). In a linear programming problem, we have:

- A Goal
- Decision Variables
- Objective
- Constraints.

In a linear program, all of our objectives and contraints are linear systems.
Consider the following problem. Suppose that Professor John has a bakery where donuts are \$5 and cakes are \$25. We have 200 units of flour, 300 units of sugar, and 500 units of eggs. Moreover. Each donuts take 2 flour, 2 sugar, and 7 eggs, while each cake takes 5 flour, 9 sugar, 12 eggs. We see that our linear program can be described as the following:

- **Decision variables**: x number of donuts and y number of cakes
- **Objective**: maximize $5x + 25y$
- **Constraints**:

$$x \geq 0, y \geq 0$$
$$2x + 5y \leq 200$$
$$2x + 9y \leq 300$$
$$7x + 12y \leq 500$$

However, the problem is that optimization over a system like this works in the real numbers. That means the optimal solution may involve a fraction of a donut or a cake, which is what we do not. Hence, we want to add an additional constraint:

$$x \text{ and } y \text{ are integers}$$

However, this is not allowed, as this contraint is not linear! Adding such a constraint gives us an integer linear program, which there currently does not exist an algorithm that can solve an integer linear program efficiently. We just have to cope with the fact that we could have fractional outputs, and then fix it when we do get a fractional output.

14.1 Classroom allocation

We have a set of courses and classrooms, which can be represented as vertices. Class c can be assigned room r iff $(c, r) \in E$. We see that this graph is a bipartite graph.

Goal: Maximize the number of courses assigned to rooms.

We want to cast this problem as a Linear Program. For each class c and room r, we assign a variable $X_{c,r}$ if $(c, r) \in E$. Our objective is:

$$\text{maximize} \sum_{(c,r)\in E} X_{c,r}$$

Our constrants are:

- for all (c, r), $X_{c,r} \geq 0$, $X_{c,r} \leq 1$
- for all rooms r: $\sum_{c:(c,r)\in E} X_{c,r} \leq 1$
- for all classes c: $\sum_{r:(c,r)\in E} X_{c,r} \leq 1$

We can plug our constraints into a LP program, and we actually know that we will get integer solutions. However, suppose that we didn't, then we just have to figure out what to do with any fractions that we get.

14.2 How to find optimal solutions?

We can actually view this very nicely, by plotting our objective and constraints on a cartesian graph. A constraint such as $x = 3$ creates a halfspace.

Definition 14.2.1: Halfspace

A constraint that separates the space we have into halves.

Definition 14.2.2: Convexity

A set of points $S \subseteq \mathbb{R}^d$ is **convex** if $\forall x, y \in S$, the line segment $x \to y$ is in S.

Definition 14.2.3: Polytope

A **polytope** is a geometric body with all edges flat.

Definition 14.2.4: Feasible region

The **feasible region** is the set of points satisfying all constraints. It has some properties:

- The feasible region is *convex*
- The feasible region is also an "intersection" of halfspaces. It is a *polytope*.

To obtain the optimum from a feasible region, we consider **level sets** of the objective function.

Definition 14.2.5: level set

A **level set** of an objective function is the set of all variables of the funciton such that the function evaluates to a constant c. Level sets are parallel to each other, so given one level set, we may generate all other level sets.

We may take a level set, and just slide it along the feasible region until we hit a optimum value.

Fact 14.2.6

For all linear programs, there is an optimal solution at a vertex.

14.3 General LP

The input to any LP problem is a set of constraints and an objective. m is the number of constraints in a linear program, while n is the number of variables. We can convert LPs with $\geq$ constraints into $\leq$ constraints by negating and vice versa. Additionally, a minimizing objective is the same as maximizing the negation.

- **Variables:** $x_1, \ldots, x_n$
- **Constraints:** $a_{k1}x_1 + a_{k2}x_2 + \cdots + a_{kn}x_n \leq b_k$ for kth constraint
- **Objective:** Maximize $c_1x_1 + \cdots + c_nx_n$

We may represent a LP system as a matrix. All the xs form a column vector, and all the bs form a column vector, and all the cs form a column vector. All of our a_is form a matrix:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \quad c = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} \quad A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

Our problem becomes maximizing c^Tx subject to $Ax \leq b$ and $x \geq 0$.

Definition 14.3.1: vertex

A **vertex** is a point x in the polytope $x \in$ feasible set which lies at the intersection of n hyperplanes in n dimensions.

There is always an optimal solution at a vertex. This suggests our first brute force search algorithm for linear programming, which is just checking all of the vertices.

Fact 14.3.2

Suppose LP has n variables and m constraints. Then the number of vertices $\leq \binom{m}{n}$. We just have to count the number of ways we can chose n constraints that intersect from m constraints.

14.4 Linear Programming Algorithm 1: "Try all vertices"

Given m constraints $C = \{\sum a_{ij}x_j \leq b_j\}$. For each subset $S \subseteq C$ of size n:

1. Solve for the point of intersections x^* using gaussian elimination.
2. Check if the intersection point x^* is a feasible point.
3. If so, compute the vaue of x^*

Output the best vertex found.

This algorithm is correct, but it's time complexity is dominated by the number of vertices, which is approximately $\binom{m}{n} \approx m^n$ which is exponentially large in n.

14.5 Linear Programming Algorithm 2: Simplex

Definition 14.5.1: Neighboring Vertex

A **Neighboring Vertex** of a vertex v is a vertex that shares all constraints as the v, but 1.

Fact 14.5.2

Suppose we have n variables, m constraints, then our number of neighbors $\leq nm$

1. Start at some vertex x^*.
2. Look at all neighboring vertices to x^*.
3. Find neighbor with best value y^*
4. If y^* has better value than x^* repeat algorithm for y^*

5. Otherwise, x^* is optimal

The sad fact is that simplex runs in exponential time still. In real life, simplex is very fast.

Theorem 14.5.3

Linear Programs can be solved in polynomial time. Look at the **ellipsoid algorithm** by Kachiyan 1979. Look at the **Interior points** algorithm by Karmarker 1984.

Chapter 15

Lecture 15: Maximum Flow

Back in the cold war, the US was curious about how much logistic capability that USSR had by shipping resources from sources to destination nodes on the eastern front. The US was able to identify a bottleneck throughput on the soviet transport chains. Harris and Ross identified a cut that bottlenecked the Russian logistic network at a flow of 163000 tons.

This problem was solved greedily, using the **Ford-Fulkerson algorithm**

15.1 Maximum Flow

Think of a graph with water pipes, which each edge having a certain capacity that limits the flow in that pipe.

- **Input:** A directed graph $G = (V, E)$, one source vertex s, one sink vertex t, and for each edge $e \in E$, a capacity $c_e \in \mathbb{Z}^+$
- **Output:** Route the maximum amount of flow from s to t. For every edge, what is the flow f_e on that edge?

Definition 15.1.1: Flow

A **flow** assigs a number f_e to each directed edge $e \in E$ such that:

- **Nonnegativity:** $f_e \geq 0$
- **Capacity:** $f_e \leq c_e$ for all $e \in E$
- **Flow conservation:** For all vertices $v \in V \setminus \{s, t\}$, we have that $\sum_{u:(u,v)\in E} f_{u,v} = \sum_{w:(v,w)\in E} f_{v,w}$. This means that the flow into a vertex is equal to the flow out of that vertex.

Definition 15.1.2: Size of flow

The **size** of a flow f is the total quantity sent from s to t. It is the total flow out of the source vertex s, or the total flow into the sink vertex t.

$$\sum_{u:(s,u)\in E} f_{s,u} = \sum_{v:(v,t)\in E} f_{v,t}$$

Definition 15.1.3: Maximum Flow Problem

The **maximum flow problem** is the problem of finding a flow f such that the size of f is maximized.

We can solve the maximum flow problem with a linear program. However, there is a more efficient way to solve the maximum flow problem, which is the Ford-Fulkerson algorithm.

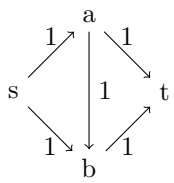
15.2 Solving Maximum Flow

15.2.1 A naive algorithm

We can solve the maximum flow problem by just trying to send as much flow as possible from s to t.

- We find a path P from s to t that has not be saturated yet.
- Send more flow along P until we cannot send any more flow.
- Repeat until we cannot find any more paths from s to t

However, this algorithm may fail, consider the following example:



In this case, we can send 2 units of flow from s to t, but if we started with the path $s \to a \to b \to t$, we would only send 1 unit of flow. Hence, we need to be more careful about how we send flow. We could mitigate this however, if we send another flow along the path $s \to b \to\to at$. This cancels out the flow in the middle, and we can still send 2 units of flow. However, this is not a very efficient algorithm, as we may have to send flow back and forth between the two paths.

We keep track of a residual graph, which is a graph that keeps track of how much flow is already used along an edge. If we use an edge $u \to v$, then there is a back-edge $v \to u$ that has a capacity equal to the flow that we have already sent along $u \to v$. The residual graph is defined as follows:

Definition 15.2.1: Residual Graph

The **residual graph** G_f of a flow f is a directed graph with the same vertices as G, and edges defined as follows:

- For each edge $e = (u, v) \in E$, if $f_e < c_e$, then we have a forward edge (u, v) with capacity $c_e - f_e$.
- For each edge $e = (u, v) \in E$, if $f_e > 0$, then we have a backward edge (v, u) with capacity f_e.

To solve the problem, we try to find another path from $s \to t$, but not on our original graph, but in our residual graph. The back-edges that we added before may now serve as forward edges in a new path from $s \to t$ in the residual graph. This implements the canceling out behavior that we saw before, so now we may undo flow that we have already sent.

15.3 Ford-Fulkerson algorithm

- Find a path P from s to t in the residual graph which is not yet saturated. This is an **augmenting path**.
- send more flow along P
- Repeat until no more augmenting paths can be found.

The flow of a path that we add is equal to the minimum capacity along that path. This is because we cannot send more flow than the minimum capacity along the path. Additionally, we know that our algorithm terminates if there exists some cut of the graph such that all edges going from the source side to the sink side are saturated. This is because if there exists such a cut, then there cannot exist any more augmenting paths from s to t.

15.3.1 Analysis of Ford-Fulkerson

Definition 15.3.1: $s - t$ cut

A $s - t$ **cut** is a partition of $V = L \cup R$ of the vertices such that $s \in L$ and $t \in R$.

Definition 15.3.2: Capacity of a cut

The **capacity** of a cut (L, R) is defined as the sum of the capacities of all edges going from L to R.

$$\text{capacity}(L, R) = \sum_{(u,v) \in E, u \in L, v \in R} c_{u,v}$$

Theorem 15.3.3: Cut-Flow Inequality

For any flow f and any $s-t$ cut (L, R), the size of the flow is at most the capacity of the cut.

$$\text{size}(f) \leq \text{capacity}(L, R)$$

Definition 15.3.4: Minimum Cut

A **minimum cut** is a cut (L, R) such that the capacity of the cut is minimized. This min-cut is the bottle-neck mentioned earlier.

Combining these ideas, we see that the size of any flow is at most the capacity of the minimum cut. Hence, if we can find a flow whose size is equal to the capacity of some cut, then we have found the maximum flow.

Theorem 15.3.5: Max-Flow = Min-Cut

The maximum size of a flow in a flow network is equal to the minimum capacity of an $s-t$ cut.

Proof. We only need to show the "$\geq$" direction, as the other direction is given by the cut-flow inequality. We run the Ford-Fulkerson algorithm on G, let f be the flow outputted by the algorithm. By the time the algorithm terminates, there are no more $s \to t$ paths in the residual graph G_f. We define:

$$L = \{\text{vertices reachable from } s \text{ in } G_f\}, \quad R = V \setminus L$$

We know that $s \in L$ and $t \in R$.

Fact 15.3.6

In G_f, every $L \in R$ edge has capacity 0.

Proof. Suppose not. Then there exists some edge (u, v) with capacity > 0. Then we may reach v from s through u, a contradiction as $v \in R$. □

Consider the original graph G with the same cut. We see that every $L \to R$ must have flow $f_e = c_e$.

Fact 15.3.7

In G, every $L \to R$ edge has flow $f_e = c_e$.

Proof. The capacity of the edge in G_f is $c_e - f_e = 0$. □

Fact 15.3.8

In G every $R \to L$ edge has flow $f_e = 0$.

Proof. Look at the reverse edge. We see that the capacity of the edge in G_f is $f_e = 0$. □

Now we claim:

Claim: $\text{size}(f) = \text{capacity}(L, R)$

The size of the flow f is equal to the capacity of the cut (L, R).

Proof. The size of the flow is:

$$\text{size}(f) = \sum_{u:(s,u)\in E} f_{s,u} = \sum_{u\in L}\sum_{v\in R} f_{u,v} - \sum_{v\in R}\sum_{u\in L} f_{v,u}$$

The first term is equal to the capacity of the cut, as all $L \to R$ edges are saturated. The second term is 0, as all $R \to L$ edges have 0 flow. Hence, the size of the flow is equal to the capacity of the cut. □

We know that the max flow is greater than or equal to the size of f, which is equal to the capacity of the cut. Hence, we have that:

$$\min_{(L,R)} \text{capacity}(L, R) \leq \max_{f} \text{size}(f) \geq \text{capacity}(L, R) \geq \min_{(L,R)} \text{capacity}(L, R)$$

So, all inequalities are equalities, and we have proven the theorem. □

This also doubles as a proof of correctness for the Ford-Fulkerson algorithm, as when the algorithm terminates, we have found a flow whose size is equal to the capacity of some cut. So Ford-Fulkerson outputs the maximum flow.

15.3.2 Ford Fulkerson runtime analysis

Each iteration of the Ford-Fulkerson algorithm requires finding an augmenting path, which can be done with DFS or BFS in $O(m + n)$ time. However, the number of augmenting paths is bounded by U, where U is the maximum flow. This is because each augmenting path sends at least 1 unit of flow. Hence, the runtime of Ford-Fulkerson is $O((m + n)U)$. This is not polynomial time, as U may be exponential in the size of the input. U may be exponential because the capacities can be very large.However, if all capacities are integers, then Ford-Fulkerson runs in pseudo-polynomial time.

Fact 15.3.9

If all the capacities are integral, then the Max-flow is integral.

Proof. Ford-Fulkerson only adds integer flows along paths, so the final flow is integral. □

Chapter 16

Duality

16.1 Bipartite Perfect Matching

- **Input:** Bipartite (undirected) graph $G = (L, R, E)$ with $|L| = |R| = n$
- **Output:** Perfect matching from L to R

Definition 16.1.1: Perfect Match

A **perfect matching** is a matching of the vertices in L with vertices in R such that each vertex is connected to another vertex from another set.

This problem is useful because it allows us to solve problems concerning pairing (can we always pair up resources in two sets?). We will solve this problem by converting the problem of bipartite matching to the problem of max-flow, and apply Ford-Fulkerson.

16.1.1 Solving Bipartite Perfect Matching

We first convert our graph G to a graph G' by adding a source node that connects to all vertices in L, and a destination node that connects to all vertices in R. Then then transform all edges into directed edges going left to right.

Theorem 16.1.2: Bipartite Perfect Matching Theorem

A perfect matching exists in a bipartite graph if and only if the maximum flow in the corresponding flow graph G' is equal to n.

Proof. We prove both directions

- ($\rightarrow$) Let M be a perfect matching on G. We place one unit of flow on every edge in M, every edge $s \rightarrow v$ and every edge $v \rightarrow t$. This is a flow of size n.
- ($\leftarrow$) Let f be a flow of size n in the flow graph G'. We know that if all capacities are integral, then our max-flow is integral. Hence, we know that f is an integral flow of size n in G' (all flow values are 0 or 1). We also know that s has n edges coming out of itself. Hence, all edges going out of s to every node in L must be saturated. Hence,

all $u \in L$ must have an out-going edge with 1 unit of flow, so every $v \in R$ has 1 unit of flow on an incoming edge (as they only have one outgoing edge each). Hence, the edges that have 1 unit of flow between L and R forms a perfect matching.

□

Hence, perfect matching is a reduction from the maximum flow problem.

Last chapter, we saw that we could prove a flow was optimal by showing there exists a cut of the same value. The fact that there are dual representations of a

16.2 Duality in LPs

For any general LP problem, we can show that a solution is optimal by constructing upper bounds on the objective function. For instance, in a maximization problem, we find the lowest upper-bound on any satisfying assignment of the variables that obeys the constraints. If we can find an upper-bound that is equal to the objective value of a feasible solution, then we have shown that the feasible solution is optimal. Can we find an algorithm for this?

16.2.1 Solving for Duality

We take all of our LP constraints and form a linear combination with them with coefficients $y_1, \ldots, y_n$. The original LP is known as the primal LP, and the new LP we form is known as the dual LP. The optimal value of the dual LP provides a bound on the optimal value of the primal LP.

$$\text{Primal LP Opt} \leq \text{Dual LP Opt}$$

In fact, we can show that this is a strict equality. If the constraints for primal LP forms a matrix, then the corresponding matrix for the constraints of the dual LP can be obtained by transposing the primal matrix and switching the roles of the objective coefficients and the right-hand side constants.

Theorem 16.2.1: Weak Duality

All feasible solutions s to primal LP $\leq$ all feasible solutions y to dual LP

Proof.

$$c^T x \leq y^T A x \leq y^T b = b^T y$$

□

As a corrollary of weak duality, we can see that:

Corollary 16.2.2

Primal LP Opt $\leq$ Dual LP Opt

We also see that there is a strong duality.

Theorem 16.2.3: Strong duality

If the **Primal LP** has an optimal solution that is bounded, then the **Dual LP** also has an optimal solution, and the optimal values of the primal and dual LPs are equal. This means that the duality gap is zero.

16.3 Two-player Zero-sum games

- **Input:** A **pay off matrix** M with m rows and n columns. Each row denotes the strategies that the row player can take, and the each column denotes the strategies that the column player can take.
- **Row player:** Pick a row
- **Column player:** Pick a column

Below is an example pay off matrix for the rock paper scissors game:

$$M = \begin{bmatrix} 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{bmatrix}$$

For a given pair of strategies (r, c), the row player wins $A_{r,c}$, and the column player wins $-A_{r,c}$. There are many different tpes of strategies.

- **Pure strategy:** Player always a single row/column. NOT GOOD LMAO
- **Mixed Strategy:** Probability distribution over pure strategies. Best strategy for rock paper scissors. *The average score is always 0 no matter what the column player does, if our strategy is a uniform distribution.*

Example.

Consider the game represented by A where:

$$A = \begin{bmatrix} 3 & -1 \\ -2 & 1 \end{bmatrix}$$

The row player announces a mixed strategy $P = (p_1, p_2)$ a probability distribution, and the column player also responds with a mixed strategy $Q = (q_1, q_2)$. We can now compute the average score of the two players as follows:

Definition 16.3.1: Average Score

Row player's average score is $\text{Score}(p, q) = 3p_1q_1 - 1p_1q_2 - 2p_2q_1 + p_2q_2$

The column player's best strategy is the minimize $\text{Score}(p, q)$. For the column player, it always makes sense to pick the column that minimizes their expected loss, so a mixed strategy is not optimal. The column player wants to pick his distributions such that the score earned from each column is minimized.

Definition 16.3.2: Column Player's Strategy

The best strategy of the column player is to pick q_1 and q_2 such that they minimize their expected loss.

$$\min_{q_1, q_2} \text{Score}(p, q)$$

We see that the row player's optimal strategy is to pick p_1 and p_2 such that they maximize their expected score. This is just an LP Problem

Definition 16.3.3: Row Player's Strategy

We calculate $\max_{p_1, p_2}\{\min\{3p_1 - 2p_2, -p_1 + p_2\}\}$

But how does this relate to the concept of duality in linear programming? Consider the game where the column player is now the player that goes first. We see that the optimal strategies and values correspond to the primal and dual solutions of a linear program, which is the connection between game theory and linear programming duality. Hence, we have that:

$$\max_p\{\min_q \text{Score}(p, q)\} = \min_q\{\max_p \text{Score}(p, q)\}$$

This is the famous **Minimax Theorem** by John von Neumann. The order of play does not change the optimal game value. There exists an optimal strategy of the row player irrespective of the column player, this is called the minimax strat.

Chapter 17

Lecture 17: P and NP

What does it mean when a solution to a problem is efficient?

Definition 17.0.1: Efficiently Solvable

A problem is **efficiently solvable** if it can be solved in polynomial time ($O(n^k)$)

Going from a brute force algorithm to a polynomial time algorithm means uncovering some non-trivial hidden substructure of the problem. Any exponential algorithm will eventually exceed any poltynomial algorithm.

Definition 17.0.2: P

P is the complexity class of all problems that are efficiently solvable.

Definition 17.0.3: NP

NP is the complexity class of all problems whose solutions can be verified efficiently.

NP stands for non-deterministic polynomial time.

17.1 The 3-coloring problem

- A graph $G = (V, E)$
- A three coloring $c : V \to \{\text{Red}, \text{Green}, \text{Blue}\}$ of G such that each edge receives 2 different colors

The trivial algorithm is to permute all the different colorings of the graph, for which there are $3^{|V|}$ possible colorings. For each coloring, verification can be done in $O(|E|)$ time. The best possible algorithm known for this is $O(1.33^{|V|})$. However, if we are given a coloring, we can verify it in polynomial time. Hence, 3-coloring is in NP. However, it is not known whether 3-coloring is in P.

17.2 Factorization Problem

- **Input:** n-bt number N
- **Output:** Two numbers $p, q > 1$ such that $N = pq$.

The trivial algorithm is to try all possible ps up to $\sqrt{N}$, which takes $O(2^{n/2})$ time. The best know algorithm is the **General number field sieve** with an improved runtime of $C^{n^{\frac{1}{3}}\log(a)^{\frac{2}{3}}}$. However, if we are given p and q, we can verify that they are factors of N in polynomial time by multiplying them together. Hence, factorization is in NP. However, it is not known whether factorization is in P.

17.3 Rudrata Cycle aka Hamiltonian Cycle

- **Input:** Graph $G = (V, E)$
- **Output:** A cycle that visits each vertex exactly once.

The trivial algorithm is to try all possible permutations of the vertices, which takes $O(|V|!)$ time. A better algorithm is to apply the dynammic programming algorithm for traveling salesman problem, which runs in $O(2^{|V|}|V|^2)$ time. However, if we are given a cycle, we can verify that it is a Rudrata cycle in polynomial time by checking that each vertex is visited exactly once. Hence, Rudrata cycle is in NP. However, it is not known whether Rudrata cycle is in P. The best possible known algorithm is $O(1.657^n)$ time.

17.4 Eulerian Cycle

- **Input:** Graph $G = (V, E)$
- **Output:** A cycle that visits each edge exactly once.

The trivial algorithm is to try all possible permutations of the edges, which takes $O(|E|!)$ time. However, there exists a polynomial time algorithm that solves this problem. We first check that all vertices have even degree, which can be done in $O(|V| + |E|)$ time. Then, we may construct the Eulerian cycle by starting at any vertex and following edges until we return to the starting vertex. If there are any unused edges, we may start at any vertex on the current cycle that has unused edges and repeat the process until all edges are used. This algorithm runs in $O(|E|)$ time. Hence, Eulerian cycle is in P.

17.5 Traveling Salesman Problem

- **Input:** Weighted graph $G = (V, E)$
- **Output:** A cycle that visits each vertex exactly once with minimum total weight.

We consider an optimized version of this problem **Min-TSP**, where we find a tour with minimum total weight. The best known algorithm has time $O(n^2 2^n)$. However, it is not know whether Min-TSP is in NP.

We also consider a search version of this problem **Search-TSP**, where we are given a budget and we just have to find a tour such that the cost is within the budget. This problem is in NP, as we can verify a tour in polynomial time. However, it is not known whether Search-TSP is in P.

There is one more version of this problem: **Decision-TSP**, where we are given a budget B and we have to decide whether there exists a tour with cost at most B.

17.6 Definitions revisited

Definition 17.6.1: Binary relaion

A **binary relation** is a subset of $\Sigma^* \times \Sigma^*$. For any problem Π, we may define a binary relation R_Π such that $(x, y) \in R_\Pi$ if and only if y is a valid solution for the problem instance x.
There are several problems that we may consider:

- `Verify(R)`: given (x, y), decide whether $(x, y) \in R$.
- `Search(R)`: given x, find y such that $(x, y) \in R$.
- `Decision(R)`: given (x, k), decide whether there exists y such that $(x, y) \in R$

Definition 17.6.2: NP and N

NP is the class of binary relations R such that `Verify(R)` is in P. P is the class of all $R \in NP$ such that `Search(R)` can be solved in poly(n).

Disclaimer: These are nonstandard definitions for P and NP. In real world, these are sometimes called FP and FNP. P and NP typically defined for decision problems.

There are problems that we are aware of that are believed not to be in NP, such as the halting problem.

Chapter 18

Reductions

Definition 18.0.1: Reduction, $A \preceq_p B$

Problem A reduces in polynomial time to problem B if any efficient algorithm for B can be used to solve A. Aka, $A \preceq B$

When $A \preceq B$, it implies that A's difficulty is lesser than or equal to B's difficulty. Consider **Two-player zero sum games** where:

- **Input:** A payoff matrix M
- **Output:** An optimal strategy for the row player.

Theorem 18.0.2: Zero sum $\preceq_p$ Linear programming

The two player zero sum game problem reduces to linear programming

Proof. We may convert the input of a zero sum game into the input of an LP problem. Because of this, we can then apply the LP algorithm onto the LP input to get a solution for the LP solution. We can then run a recovery algorithm to convert the LP solution into a zero sum solution. □

In any reduction, we have to take an input of A, and design a reduction algorithm to convert an input of A to an input of B. We then have to use B to solve for the solution of B, and then use a recovery algorithm to solve for the solution of A.

18.1 Rudrata Cycle reduces to Search TSP

For the **Rudrata/Hamiltonian cycle** problem, we have an input graph of:

- **Input:** Graph $G = (V, E)$
- **Output:** Tour of G that visits each vertex exactly once.

For the search-TSP algorithm, we have:

- **Input:** cities $1, \ldots, n$ with pairwise distances d_{ij} and outputs a budget B.

- **Output:** Tour whose total distance $\leq B$

Theorem 18.1.1: Rudrata Cycle $\preceq_p$ Search-TSP

The rudrata cycle problem reduces to search-TSP

Proof. We simply convert our original graph into a weighted graph with distances set to 1. And run search-TSP on budget equal to the number of vertices. □

There are some properties of reductions:

Claim: Reduction Properties

- Reduction and recovery algas must be efficient (poly-time)
- $A \preceq_p B$ and $B \preceq_p C \implies A \preceq_p C$.
- We can prove that $A \preceq_p B$, even if A and B are not known to be efficient.
- $A \preceq_p B$ and A reduces to B is confusing. Proving that $A \preceq_p B$ by taking an instance of B and creating a reduction algorithm to an instance of A is not valid. Problems can be weaker than other problems, which means that if A reduces to B, then B does not necessarily reduce to A.

Max flow reduces to linear programming, but linear programming does not reduce to max flow. Bipartite matching reduces to max flow, but max flow does not reduce to bipartite matching.

Interestingly, factoring reduces into 3-coloring, and it also reduces into the rudrata path problem

Definition 18.1.2: NP-Hard, NP-Complete

A problem A is **NP-Hard** if every problem $B \in NP$ reduces to A. A is $NP-complete$ if $A \in NP$ and A is $NP-$hard.

Fact 18.1.3

If A and B are NP complete, then $A \preceq_p B$ and $B \preceq_p A$

Fact 18.1.4

If there exists a polynomial-time algorithm for any NP-complete problem, then it implies $P = NP$

18.2 How to show NP-Hard and NP-Completeness?

The **Cook-Levin theorem** states that every problem in NP reduces to a problem called **Circuit Sat**. There is a reduction from **Circuit Sat** to 3-Sat. We can keep showing this

for many many problems to build up the set of problems that are NP-complete.

To show that problem A is NP-complete:

- $A \in NP$ show verification algorithm
- Pick some problem B that is $NP - complete$ (usually well-known), and show that $B \preceq_p A$.

18.3 Circuit sat reduces to 3-sat

We first must define what a boolean circuit is:

Definition 18.3.1: Boolean Circuit

A **boolean circuit** is a directed acyclic graph (DAG) with:

- input nodes $x_1, \ldots, x_n$
- one output node
- gates marked AND, OR, NOT

The circuit sat problem can be describe as:

- **Input:** A circuit C with n inputs and m gates
- **Output:** An assignment $x \in \{0,1\}^n$ such that $C(x) = 1$

Theorem 18.3.2: Cook-Levin Theorem

Circuit Sat is NP complete

The 3-Sat problem can be described as:

- **Input:** N Boolean variables $x_1, \ldots, x_n \in \{0,1\}$, $m \leq$ 3-variable clauses.
- **Output:** An assignment $x_1, \ldots, _N \to \{0,1\}$ satisfying all m clauses

We now prove that $3 - SAT$ is an NP-complete problem.

Theorem 18.3.3: $3 - SAT \in NP$-complete

$3 - SAT$ is an NP-complete problem.

Proof. We must show that $3 - SAT$ is in NP and circuit sat reduces to 3-SAT. To show that circuit-sat reduces to 3-sat, we can create constraints for each node in a circuit tree. □

Chapter 19

Lecture 19: NP-Completeness

19.1 Independent Set

- **Input:** Graph $G = (V, E)$ with n vertices and m edges
- **Output:** An independent set of size k. k vertices in the graph V with no edges between them.

We first sketch out a naive algorithm for this problem, where we try all sets of k vertices. There are $\binom{n}{k}$ sets of vertices that we can check, and this is roughly n^k sets. Hence, this naive algorithm runs in time $\Omega(n^k)$. Although this is exponential, note that if k happened to be constant, then the runtime decays to a polynomial time algorithm. However, if k is not constant, then the algorithm is exponential time, so it's not efficient.

Theorem 19.1.1: Independent Set is NP-hard

Independent set is NP-Hard

Proof. To show this, we pick any NP-complete problem A, and show that $A \preceq_p$ Independent Set. Recall that every problem in NP reduces to Circuit Sat. Additionally, $Circuit - SAT$ reduces to $3 - SAT$. Hence, we may pick $3 - SAT$ as our problem A. We now show that $3 - SAT \preceq_p$ Independent Set.

To formally prove problem a problem A is NP-complete, we must show that $A \in NP$ and some NP-complete B reduces to A.

For any solution to the Independent set problem, we can show that there is verification algorithm in polynomial time to verify a solution, We may just check that there does not exist an edge between any two vertices in the independent set. This takes $O(k^2)$ time, which is polynomial in the size of the input. Hence, Independent Set is in NP.

Now for the reduction. Given an instance of $3 - SAT$ caled ψ, recall that we have a series of clauses $C_1, \ldots, C_m$, each with 3 literals. We now design a polynomial-time reuction algorithm from ψ to an instance of Independent set. We construct a graph $G = (V, E)$ as follows:

- For each clause C_i, we create 3 vertices in V, one for each literal in the clause.

- For each pair of vertices corresponding to literals in the same clause, we add an edge between them.
- For each pair of vertices corresponding to literals that are negations of each other, we add an edge between them.

We now set $k = m$, and we claim that ψ is satisfiable if and only if G has an independent set of size k. Any independent set that is generated from the graph G can only have one vertex from each clause, as all vertices in a clause are connected to each other. Additionally, no two vertices in the independent set can be negations of each other, as they are also connected. Hence, we may set all literals corresponding to vertices in the independent set to true, and set all other literals to false. This satisfies all clauses, as each clause has at least one literal that is true. This forms the basis for our **reduction algorithm**.

Conversely, if we have a satisfying assignment for ψ, then we many construct an independent set of size k by picking one true literal from each clause. No two literals in the independent set can be negations of each other, as they are both true. Additionally, no two literals in the independent set can be from the same clause, as we only pick one literal from each clause. Hence, the vertices corresponding to the literals in the independent set have no edges between them, so they form an independent set of size k. This forms the basis for our **recovery algorithm**. □

There are four things that we need to prove for any reduction:

- *Show that the reduction algorithm runs in polynomial time.*
- *Show that the recovery algorithm runs in polynomial time.*
- *Show the correctness of the reduction algorithm.*

  Proof. If 3Sat instance x has SAT assign $A : \{x_1 \ldots, x_n\} \to \{0, 1\}$ then graph G has ind. set I of size $k - m$.

  - For each clause $x_i \vee \overline{x_j} \vee x_l$, A sets either x_i, $\overline{x_j}$, or x_l to true.
  - Pick one (say x_i) and add x_i to I.
  - So I is of size of $k = m$ by construction, as we picked one vertex from each clause.
  - No two vertices in I are connected by an edge, as A cannot set both x_i and $\overline{x_i}$ to true, and only one vertex is picked from each clause.
  - Hence, I is an independent set of size k.

  □

- *Show the correctness of the recovery algorithm.*

  Proof. Let I be an ind set in G of size k. Then the recovery algorithm outputs a satisfying assignment. The recovery algorithm sets x_i to true if vertex corresponding to x_i is in I, and sets x_i to false otherwise. Note that:

  - For each i only x_i or $\overline{x_i}$ is in I. So the recovery algorithm is well-defined.

- For each clause, at least one vertex is in I (as $|I| = k = m$ and only one vertex can be picked from each clause). So at least one literal in each clause is set to true.

□

19.2 Integer Programming (IP)

- **Input:** A linear program.
- **Output:** An integer solution to the linear program.

Theorem 19.2.1: Integer Programming is NP-Hard

Integer programming is NP-Hard, as independent set reduces down to integer Programming

Proof. We want to design a polynomial time reduction algorithm from independent set to integer programming. Given an instance of independent set, which is a graph $G = (V, E)$ and an integer k, we construct the following integer program:

$$\max 1$$

subject to:

$$x_u + x_v \leq 1 \quad \forall (u, v) \in E$$

$$x_v \in \{0, 1\} \quad \forall v \in V$$

$$\sum_{v \in V} x_v = k$$

Suppose there exists an independent set I of size k, we want to show there exists a solution to the IP instance. We just set $x_i = 1$ if $i \in I$, and $x_i = 0$ otherwise. This satisfies all constraints of the IP instance, so there exists a solution to the IP instance. Conversely, suppose there exists a solution to the IP instance. We want to show that there exists an independent set of size k. We construct the independent set I as follows:

$$I = \{v \in V : x_v = 1\}$$

We see that $|I| = k$ because of the last constraint. Additionally, no two vertices in I are connected by an edge, because of the first constraint. Hence, I is an independent set of size k. □

19.3 Clique

- **Input:** Graph $G = (V, E)$ with n vertices and m edges, an integer $1 \leq k \leq n$.
- **Output:** A clique of size k. k vertices in the graph V with all possible edges between them.

Theorem 19.3.1: Clique is NP-Hard

Clique is NP-Hard, as independent set reduces down to clique.

Proof. Take an independent set instance with graph $G = (V, E)$ with integer k and convert it into a clique instance with graph $G' = (V, E')$ where E' is the complement of E. The integer remains the same. We claim that G has an independent set of size k if and only if G' has a clique of size k. If G has an independent set of size k, then no two vertices in the independent set are connected by an edge in G. Hence, all pairs of vertices in the independent set are connected by an edge in G', so the independent set forms a clique of size k in G'. Conversely, if G' has a clique of size k, then all pairs of vertices in the clique are connected by an edge in G'. Hence, no two vertices in the clique are connected by an edge in G, so the clique forms an independent set of size k in G. □

19.4 Rudrata (s, t) path

- **Input:** Graph $G = (V, E)$ with n vertices and m edges, two vertices $s, t \in V$.
- **Output:** A Rudrata path from s to t. A path that visits each vertex exactly once.

Theorem 19.4.1: Rudrata (s, t) path $\preceq_p$ Rudrata Cycle

Rudrata (s, t) path reduces to Rudrata cycle.

Proof. Given an instance of Rudrata (s, t) path with graph $G = (V, E)$ and vertices $s, t \in V$, we construct an instance of Rudrata cycle as follows. We create a new graph $G' = (V', E')$ where $V' = V \cup \{u\}$, where u is a new vertex. We set $E' = E \cup \{(t, u), (u, s)\}$. We claim that there exists a Rudrata (s, t) path in G if and only if there exists a Rudrata cycle in G'. If there exists a Rudrata (s, t) path in G, then we can construct a Rudrata cycle in G' by following the Rudrata (s, t) path from s to t, then going from t to u, and then going from u back to s. Conversely, if there exists a Rudrata cycle in G', then we can construct a Rudrata (s, t) path in G by removing the edges (t, u) and (u, s) from the cycle. Th □

Chapter 20

Lecture 20: Approximation Algorithms

Suppose a problem Q was shown to be NP-hard. There likely does not exist an efficient solution for Q. There are three things that we may do:

- **Learn more about inputs:** Maybe we can find a special case of inputs that is easier to solve. The register allocation problem is NP-hard in general, but if the interference graph is an interval graph, then it can be solved in polynomial time.
- **Heuristic:** Maybe we can find an algorithm that works well in practice, even if there is no guarantee on its performance. Simplex method for LP is an example of this.
- **Approximation algorithms:** Maybe we can find an efficient algorithm that finds a solution that is close to optimal.

20.1 α-Approximation Algorithms

Definition 20.1.1: Approximation Algorithm

For a minimization problem Q, an algorithm A is an **α-approximation algorithm** if for all instances I of Q, we have:

$$A(I) \leq \alpha \cdot OPT(I)$$

where $\alpha \geq 1$. For a maximization problem Q, an algorithm A is an α-approximation algorithm if for all instances I of Q, we have:

$$A(I) \geq \alpha \cdot OPT(I)$$

where $0 \leq \alpha \leq 1$.

20.1.1 Vertex-Cover

- **Input:** A graph $G = (V, E)$.

- **Output:** A vertex cover $C \subseteq V$ of minimum size.

Definition 20.1.2: Vertex Cover

A **vertex cover** of a graph is a subset of vertices such that every edge in the graph is incident to at least one vertex in the subset.

Theorem 20.1.3: Vertex Cover is NP-Hard

Vertex cover is NP-Hard, as independent set reduces to vertex cover.

Since vertex cover is NP-Hard, we probably cannot find an efficient algorithm to solve it. However, we can find an approximation algorithm for it. To do this, we find a simpler problem that we can solve, that is a simplification of the original problem. We then solve the simpler problem, and use the solution to construct a solution for the original problem.

20.1.2 Algorithm 1 for Verted Cover

1. We first compute a maximal matching M in G.

   Definition 20.1.4: Maximal Matching

   A **matching** is a set of edges w/ no overlapping vertices. It is **maximal** if no more edges can be added to it.

   This may be computed greedily by iterating through edges and adding them to the matching if they do not overlap with any edges already in the matching.

2. We then output $C = \{\text{Both end points of all edges in } M\}$

Theorem 20.1.5: C is a good vertex cover

The set C is a verte cover and $|C| \leq 2 \cdot OPT$.

Proof. We first claimed that C is a vertex cover. Assume for the sake of contradiction that C was not a vertex cover. Then there exists some edge $(u, v) \in E$ s.t. $u, v \notin C$. Then we could add (u, v) to M, contradicting the maximality of M, so C is a vertex cover.

We then claimed that $|C| \leq 2 \cdot OPT$. Take any vertex cover, it must cover each edge $(u, v) \in M$. Hence, the vertex cover must include either the vertex u or v, or both. Since there are M edges in M, the vertex cover must include at least M vertices. Hence, $OPT \geq |M|$. We see that $|C| = 2|M| \leq 2 \cdot OPT$. □

Now an interesting question we may ask is whether or not we may take any problem that reduces to vertex-cover, and apply the approximation algorithm for vertex-cover to get an approximation algorithm for the original problem. The answer is no, as the reduction may not preserve the approximation ratio.

20.1.3 Algorithm 2 for Vertex Cover

We convert the vertex cover problem into a Linear program. For each vertex, we create a variable $x_v \in \{0, 1\}$, where $x_v = 1$ if v is in the vertex cover, and 0 otherwise. We can then set the following integer program:

Solution

$$\min \sum_{v \in V} x_v$$

subject to:

$$x_u + x_v \geq 1 \quad \forall (u, v) \in E$$

$$x_v \in \{0, 1\} \quad \forall v \in V$$

We claim that:

Claim: LP-OPT $\leq$ OPT

The optimal value of the LP relaxation is at most the optimal value of the integer program.

Proof. The feasible region of the linear program contains the feasible region of all vertex covers. The OPT vertex Cover is a feasible solution to LP. But there are also fractional solutions that are feasible to LP but not vertex covers. Hence, $LP - OPT \leq OPT$. □

We attempt to make an algorithm by taking any solution to the LP relaxation, and rounding it to get a vertex cover. To do this, we apply the following rounding rule:

- Let $\{x_i^* \mid i = 1, \ldots, n\}$ be the optimal solution.
- Rounding Rule:

$$\begin{cases} X_i^* \geq \frac{1}{2} & \rightarrow x_i = 1, v \in C \\ X_i^* < \frac{1}{2} & \rightarrow x_i = 0, v \notin C \end{cases}$$

Claim: C is a vertex cover

The set C is a vertex cover.

Proof. By our LP, we know that for all edges $(u, v) \in E$, we have $x_u^* + x_v^* \geq 1$. Hence, at least one of x_u^* or x_v^* is at least $\frac{1}{2}$. By our rounding rule, at least one of u or v is in C. Hence, C is a vertex cover. □

Claim: $|C| \leq 2 \cdot LP - OPT$

The size of the vertex cover C is at most twice the optimal value of the LP relaxation.

Proof. For all $i \in C$, we have $x_i^* \geq \frac{1}{2}$. Hence,

$$|C| = \sum_{i \in C} 1 \leq \sum_{i \in C} 2x_i^* \leq 2 \sum_{i=1}^{n} x_i^* = 2 \cdot \text{LP-OPT}$$

Since $LP - OPT \leq OPT$, we have $|C| \leq 2 \cdot OPT$. □

20.2 Traveling Salesperson Problem

- **Input:** n cities with pairwise distance $d_{i,j}$.
- **Output:** Minimum distance tour that visits each city exactly once.

Theorem 20.2.1: No approximation for TSP

There is no α-approximation algorithm for TSP for any $\alpha \geq 1$, unless $P = NP$.

Proof. We show this by reducing the Rudrata cycle problem to TSP. Given an instance of Rudrata cycle with graph $G = (V, E)$, we construct an instance of TSP as follows. We set the cities to be the vertices V. We set the pairwise distances as follows:

$$d_{i,j} = \begin{cases} 1 & (i,j) \in E \\ \alpha n + 1 & (i,j) \notin E \end{cases}$$

We claim that G has a Rudrata cycle if and only if the optimal TSP tour has length at most n. If G has a Rudrata cycle, then we can construct a TSP tour by following the Rudrata cycle. Since all edges in the Rudrata cycle are in E, the total length of the TSP tour is n. Conversely, if the optimal TSP tour has length at most n, then all edges in the TSP tour must be in E, as any edge not in E has length $\alpha n + 1 > n$. Hence, the TSP tour corresponds to a Rudrata cycle in G. □

.

20.2.1 Metric TSP

We consider a special case of TSP called **metric TSP**, where the pairwise distances satisfy the triangle inequality:

$$d_{ij} + d_{jk} \geq d_{ik} \quad \forall i, j, k$$

Intuitively, this means that the direct path is always the shortest path. We can design an approximation algorithm for metric TSP as follows:

1. Find the MST of the graph. (The MST has a lower total weight than the optimal TSP tour, since removing any edge from the optimal TSP tour gives a spanning tree.)
2. Perform a DFS traversal of the MST, starting from any vertex, and record the order in which the vertices are visited.
3. The cost of the DFS traversal is 2·MST, since each edge is traversed twice.

4. Create a tour by visiting the vertices in the order they were first visited in the DFS traversal, skipping any already visited vertices. This is the final TSP tour.

Theorem 20.2.2: Metric TSP Approximation

The above algorithm is a 2-approximation algorithm for metric TSP.

Proof. Let C be the cost of the TSP tour produced by the algorithm, and let OPT be the cost of the optimal TSP tour. We have:

$$C \leq 2 \cdot \text{MST} \leq 2 \cdot OPT$$

The first inequality holds because the DFS traversal visits each edge of the MST twice, and the second inequality holds because the MST is a lower bound on the optimal TSP tour. □

Chapter 21

Lecture 21: Randomized Algorithms

21.1 Randomized Algorithms

Algorithms that use **random bits** to solve a problem. Randomized algorithms tend to be faster and more elegant. We consider two types of randomized algorithms:

- **Las Vegas:** The output is always correct, but runtime depends on randomness. For example, the expected runtime for Quicksort is $O(n \log n)$.
- **Monte Carlo (Maybe Correct):** Always runs in a fixed amount of time, but the output is not always guaranteed to be correct.

What does it mean for an algorithm to be maybe correct? For decision (Yes/No) decision problems, we want to make sure that our algorithm is always beating an uneducated guess, such that our probability of correctess is at least greater than $\frac{1}{2}$. This is called a **Two-sided error**. We may also consider the one-sided error. Suppose we were given a yes instance, we want to make sure that our algorithm always outputs Yes, and given a no instance, we want it to output No with some accuracy $p > 0$.

21.1.1 Boosting

Given a **One-sided error** algorithm with accuracy $p > 0$, we can boost the accuracy of the algorithm to a high (near certain) value. To do this, we:

- Run the algorithm t times
- If any t runs output No, output No. Otherwise, output Yes.

Given that the input is a Yes instance, our algorithm will always output Yes. Hence, we are concerned with the case where the input is a No instance. The probability of our algorithm outputting Yes in this case is the probability that all t runs output Yes, which has probability:

$$(1-p)^t$$

Now note that $(1-p)^t \leq e^{pt}$. Hence, by choosing $t = \frac{\ln(1/\delta)}{p}$ ($e^{pt} \leq (1-p)$), we can make the probability of error at most δ.

21.2 Integer Factorization

Given a 500 dgit number n, we want to find the prime factorization $N = p_1 p_2 \ldots, p_k$. Consider the following algorithm:

- Check every integer $2 \leq x \leq \sqrt{n}$ to see if x divides n

Suppose that $N \approx 10^5 00$, then $\sqrt{N} = \sqrt{10^{500} = 10^{250}}$. This algorithm is exceedingly inefficient!

21.3 Primality Testing

Given an n digit number N, we want to determine if N is prime or composite. This is a decision problem. Our first idea is to just factor it, but as we've seen, this is inefficient. We can use another fact about prime numbers to solve this problem. We may employ **Fermat's Little Theorem**:

Theorem 21.3.1: Fermat's Little Theorem

If N is prime, then $a^{N-1} \equiv 1 \mod N$ for all integers $1 \leq a < N$.

We can use this theorem to design a randomized algorithm for primality testing. This procedure we design is called the **Fermat Primality Test**:

- Randomly pick an integer $a \in \{1, \ldots, N-1\}$ uniformly at random.
- If $a^{N-1} \equiv 1 \pmod N$, output **prime**. Otherwise, output **composite**.

We want to show that the Fermat test is a Monte Carlo algorithm with one-sided error.

Fact 21.3.2

If N is prime, this algorithm will always output **prime**.

On the otherhand, if N is composite, we want to show that the algorithm outputs **composite** with some probability $p > 0$. Our goal is to show that we can design an algorithm where p is at least $\frac{1}{2}$. Note that if we ever pick a non-coprime number a (i.e. $\gcd(a, N) \neq 1$), then we will always output **composite**. Note that for a large number $N = p^2$, we have that only $1/p$ of the numbers in $\{1, \ldots, N-1\}$ are coprime to N. Hence, if we pick a random number, we will output **composite** with probability at least $1 - 1/p$. Hence, we see that the fermat test is only good if $a^{N-1} \not\equiv 1 \pmod N$ for lots of coprime a. Unfortunately, we must consider the **Carmichael numbers**.

21.3.1 Carmichael numbers

Definition 21.3.3: Carmichael Number

A **Carmichael number** is a composite number N such that $a^{N-1} \equiv 1 \pmod{N}$ for all integers $1 \leq a < N$ with $\gcd(a, N) = 1$.

The Carmichael Number will pass the Fermat test for all coprime a. Hence, the only as that will fail the Fermat test are the non-coprime as. Lets pretend that the carmichael numbers do not exist.

Theorem 21.3.4: Fermat Test without Carmichael numbers

Suppose N is composite and not Carmichael. Then $\mathbb{P}[\text{Fermat Test}(N) = \textbf{composite}] \geq \frac{1}{2}$

Proof. Not being carmichael means that there exists some coprime b such that $b^{N-1} \neq 1 \pmod{N}$. We can show that the existance of one such coprime b implies a large number of such comprimes.

Claim

Suppose a passes Fermat Test ($a^{N-1} \equiv 1 \pmod{N}$), then $a \cdot b \pmod{N}$ fails the Fermat Test.

Proof. Note that:

$$(a \cdot b)^{N-1} \equiv a^{N-1} \cdot b^{N-1} \equiv 1 \cdot b^{N-1} \not\equiv 1 \pmod{N}$$

□

Hence, for every a that passes the test, we can find a unique $a \cdot b \pmod{N}$ that fails the test. This gives us a one-to-one mapping between the set of coprime as that pass the test and the set of coprime as that fail the test. Let:

$$A = \{\text{coprime } a : a^{N-1} \equiv 1 \pmod{N}\}$$
$$B = \{\text{coprime } a : a^{N-1} \not\equiv 1 \pmod{N}\}$$

We see that $A \subseteq B$, so $|A| \leq |B|$. Hence, we know that there exists more coprime numbers that fail the Fermat test than coprime numbers that pass it. Hence, when we pick a random a, the probability that it is coprime and fails the Fermat test is at least $\frac{1}{2}$. □

21.3.2 Runtime Analysis

To compute $a^{N-1} \pmod{N}$, we can use **exponentiation by squaring**. We can also include another check to detect Carmichael numbers. At the end, the test we've designed is called the **Miller-Rabin Primality Test**, and it has time complexity $O(k \cdot n^3)$, where k is the number of iterations of the test we run, and n is the number of digits in N.

21.4 BPP vs P

Definition 21.4.1: BPP

The class of decision problems that can be solved by a Monte Carlo algorithm with two-sided error in polynomial time is called **BPP** (Bounded-error Probabilistic Polynomial time).

It is known that $P \subseteq BPP$, since any deterministic polynomial time algorithm is also a Monte Carlo algorithm with no error. It is an open question whether or not $BPP \subseteq P$.

Chapter 22

Lecture 22: Randomiation Algorithms II and Online Algorithms

Randomized algorithms are neat because they either allow us to construct efficient algorithms, or elegant solutions.

22.1 Minimum Cut

As a recap, given an unweighted, undirected graph $G = (V, E)$, we want to return a non-trivial cut $(C, \overline{C})$ of minimum size. Ford-fulkerson only gives us the minimum $s - t$ cut for some source s and destination t. However, to find the minimum cut, we will need to consider all possible pairs of s and t. This is inefficient, as there are $O(n^2)$ such pairs. We try to use randomization to construct a Monte Carlo algorithm that runs in quadratic time $O(n^2)$ with probability of success 99%.

22.1.1 Karger's Algorithm

We pick a uniformly random edge e_i on the ith iteration of the algorithm, and contract the edge e_i, combining the endpoints of e_i into a single vertex. We repeat this until we only have two edges left, where we output the cut defined by the two remaining vertices. The contraction of an edge (u, v) is defined as follows:

Definition 22.1.1: Edge Contraction

- Remove the edge (u, v) from the graph.
- Merge vertices u and v into a single vertex w.
- For each edge (v, x) or (u, x), add an edge (w, x).

Intuitively, we want to avoid contracting edges that are a part of the minimum cut. The

chances of us actually selecting an edge across the minimum edge is low, as there are many more edges that are not part of the minimum cut. Generally speaking, the more edges that we have in a graph, the less likely we are to pick an edge in the minimum cut.

Theorem 22.1.2: Karger's algorithm success probability

Let $C = (S, \overline{S})$ be a min cut of size k. Then $\mathbb{P}(\text{Karger's alg outputs C}) \leq \frac{1}{\binom{n}{w}} = \frac{2}{n(n-1)}$.

Proof. Let $G_i =$ Graph at the beginning of the ith iteration. Let $H_i =$ The ith contracted edge e_i does not cross the cut. We claim that:

$$\begin{aligned}\mathbb{P}(\text{Alg outputs } (C,\overline{C})) &= \mathbb{P}(H_1 \cap H_2 \cap \cdots \cap H_{n-2}) \\ &= \mathbb{P}(H_1) \cdot \mathbb{P}(H_2|H_1) \cdots \mathbb{P}(H_{n-2}|H_1 \cap \cdots \cap H_{n-3})\end{aligned}$$

Now:

$$\begin{aligned}\mathbb{P}(H_i \mid H_1, \ldots, H_{i-1}) &= 1 - \mathbb{P}(\text{Contract edge in} C \mid H_1, \ldots, H_{k-1}) \\ &= 1 - \frac{\text{Number of edges across} C}{\text{Number of edges in} G_i}\end{aligned}$$

We claim several facts:

Fact 22.1.3

- Min-Cut$(G_i) \geq k$
- Number of vertices in $G_i = n - (i+1) = n - i + 1$
- Degree of each vertex in $G_i \geq k$
- Number of edges in $G_i \geq \frac{(n-i+1)k}{2}$ (By handshake Lemma)

Hence, we have:

$$\begin{aligned}\mathbb{P}(H_i \mid H_1, \ldots, H_{i-1}) &\geq 1 - \frac{k}{\frac{1}{2}k(n-i+1)} \\ &= 1 - \frac{1}{2(n-i+1)} \\ &= \frac{n-i-1}{n-i+1}\end{aligned}$$

Hence, this probability is close to 1 if n is large compared to i. We can now compute the

overall probability:

$$\begin{aligned}\mathbb{P}(\text{Alg outputs } (C,\overline{C})) &\geq \prod_{i=1}^{n-2} \frac{n-i-1}{n-i+1} \\ &= \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{2}{4} \cdot \frac{1}{3} \\ &= \frac{2}{n(n-1)} \\ &= \binom{n}{2}^{-1}\end{aligned}$$

□

If we want Karger's algorithm to succeed with constant probability, we can just repeat $100n^2$ times.

Fact 22.1.4

The number of min-cuts in a graph with n vertices is at most $\binom{n}{2}$.

Proof. The probability that our algorithm outputs an min-cut is at least the sum of the probabilities of outputting each min-cut. Since the total probability is at most 1, we have:

$$1 \geq \mathbb{P}(Outputmincut) \geq \sum_{\text{min cut}} \frac{1}{\binom{n}{2}} = \frac{\text{Number of min cuts}}{\binom{n}{2}}$$

□

22.2 Online Algorithms

We introduce the **Expert's Problem**. We have n experts $E_1, \ldots, E_n$ and T days. On each day $t = 1, \ldots, T$, each expert makes a prediction. The algorithm must make a prediction based on the experts' predictions. After making the prediction, the correct answer is revealed, and we can see which experts were correct. The goal is to minimize the number of mistakes made by the algorithm.

Our model has several features:

- The model does not assume past predictions predict future predictions.
- Outcomes of each day can be picked adversarially, based on past predictions, even current prediction.
- The input is provided piece-by-piece.

Example.

A simple example is that one of the n experts is always correct. We want to make the fewest mistakes possible.

One algorithm is actually pretty simple. We can always follow the first expert until they make a mistake. When they make a mistake, we switch to the next expert. We repeat this process until we find the expert that is always correct. This algorithm makes at most $n-1$ mistakes, as we can only switch experts $n-1$ times.

However, we can do better with randomization. We use the halving algorithm. For day $t = 1, \ldots, T$. We set keep track of a set C as the set of experts that haven't made a mistake yet. We predict YES if at least half of correct experts do so, otherwise, we predict no.

Theorem 22.2.1: Halving Algorithm Mistake Bound

The halving algorithm makes at most $\log_2(n)$ mistakes.

Proof. When halving makes a mistake, $|C_{t+1}| \leq \frac{1}{2}|C_t|$. Can only do so $\log_2(n)$ times. □

A slightly harder case occurs when one of the n experts makes at most m mistakes.

Example.

Our new algorithm is as follows. For each date $t = 1, \ldots, T$, we now keep track of C as the set of experts who have made less than m mistakes. We predict YES if at least half of the experts in C predict YES, otherwise we predict NO.

Theorem 22.2.2: Mistake Bound with m mistakes

The algorithm makes at least $(m+1)\log_2(n)$ mistakes.

Proof. Each mistake halves the size of C. Initially, $|C_1| = n$. At the end, $|C_T| \geq 1$ (the expert with at most m mistakes). Hence:

$$1 \leq |C_T| \leq \frac{n}{2^{\text{number of mistakes}}} \implies \text{number of mistakes} \leq (m+1)\log_2(n)$$

□

Chapter 23

Lecture 23: Online Algorithms II

A quick reminder of the **Experts Problem**, there are N experts $E_1, \ldots, E_n$ and T days. One each day $t = 1, \ldots T$:

- Each expert makes a prediction (YES/NO)
- We make a prediction
- The true outcome is revealed

Our goal is that at time $t+1$, the number of mistakes we made is less than or equal to the number of mistakes by the best expert with a small margin of error. We revisit the special case where one of the experts makes at most m mistakes. The naive algorithm we had was:

- Let $\texttt{Plausible}_t \subseteq [n]$ be the set of experts that have made at most m mistakes up to day t.
- Predict YES if at least half of experts in $\texttt{Plausible}_t$ predict YES, otherwise predict NO.

Theorem 23.0.1: Naive Algorithm is not good

The naive algorithm makes at least $(m+1)\log_2(n)$ mistakes.

Proof. Intuitively, we can eliminate halves at a time, but it takes each half $m+1$ mistakes to eliminate each half. Hence, we can make as many as $(m+1)\log_2(n)$ mistakes. □

23.1 Weighted Majority Algorithm

Given a parameter $0 < \epsilon < 1$, we assign each expert a weight $w_i^{(1)} = 1$ at the start of the algorithm. For each day $t = 1, \ldots, T$:

- Output the **weighted majority** of the experts according to $w^{(t)}$ weights.

- For each expert i that made a mistake, update their weight:

$$w_i^{(t+1)} = w_i^{(t)} \cdot (1 - \epsilon)$$

- For each expert i that was correct, keep their weight the same:

$$w_i^{(t+1)} = w_i^{(t)}$$

Suppose we run weighted majority algorithm with $\epsilon = \frac{1}{2}$.

Fact 23.1.1

Let $W^{(t)}$ = total weight of experts at time T. Suppose WN made a mistake at time t. Then $W^{(t+1)} \leq \frac{3}{4} W^{(t)}$.

Fact 23.1.2

Suppose expert i makes m_i mistakes. Then $w_i^{(T+1)} = (1/2)^{m_i}$.

Theorem 23.1.3: Weighted Majority Bound

Let M = number of mistkaes made by WN with $\epsilon = \frac{1}{2}$. Then we claim $M \leq 2.4(\texttt{OPT} + \log_2(n))$

Proof. Let i^* = best expert. The by the second fact, we have that $W_{i^*}^{(T+1)} = \frac{1}{2}^{\texttt{OPT}}$. So $W^{(T+1)} \geq \left(\frac{1}{2}\right)^{\texttt{OPT}}$. Suppose the algorithm makes M mistakes, then the weight $W^{(T+1)} \leq \left(\frac{3}{4}\right)^M \cdot W^{(1)} = \left(\frac{3}{4}\right)^M \cdot n$. Hence:

$$\begin{aligned}
\left(\frac{1}{2}\right)^{\texttt{OPT}} &\leq \left(\frac{3}{4}\right)^M \cdot n \\
\left(\frac{4}{3}\right)^M &\leq n \cdot 2^{\texttt{OPT}} \\
M \cdot \log_2\left(\frac{4}{3}\right) &\leq \log_2(n) + \texttt{OPT} \\
M &\leq \frac{1}{\log_2(4/3)}(\log_2(n) + \texttt{OPT}) \\
&\leq 2.4(\log_2(n) + \texttt{OPT})
\end{aligned}$$

□

Interestingly, and sadly, it is impossible to do better than a bound of $2 \cdot \texttt{OPT}$. The fix for this is to use randomization. This is called **Randomized Weighted Majority Algorithm**.

23.2 Randomized Weighted Majority Algorithm

Given a parameter $0 < \epsilon < 1$, we assign each expert a weight $w_i^{(1)} = 1$ at the start of the algorithm. For each day $t = 1, \ldots, T$:

- Pick an expert i with probability proportional to their weight:

$$\mathbb{P}[\text{Pick expert } i] = \frac{w_i^{(t)}}{W^{(t)}}$$

- Output the prediction of the picked expert.
- For each expert i that made a mistake, update their weight:

$$w_i^{(t+1)} = w_i^{(t)} \cdot (1 - \epsilon)$$

- For each expert i that was correct, keep their weight the same:

$$w_i^{(t+1)} = w_i^{(t)}$$

Theorem 23.2.1: Randomized Weighted Majority Bound

Let M = expected number of mistakes made by RWMA with parameter ϵ. Then we claim:

$$\mathbb{E}[M] \leq (1 + \epsilon)\texttt{OPT} + \frac{\log(n)}{\epsilon}$$

If we set $\epsilon = \sqrt{\frac{\log(n)}{\texttt{OPT}}}$, then we have:

$$\mathbb{E}[M] \leq \texttt{OPT} + 2\sqrt{\texttt{OPT} \cdot \log(n)}$$

23.3 General Setting

We consider the general setting where the experts can make arbitrary predictions, not just YES/NO. Suppose tere are n actions at each step, on day t, we play $i_t \in [n]$ where action i has cost $c_i^{(t)} \in [0, 1]$. The goal is to minimize total cost:

$$\sum_{t=1}^{T} c_{i_t}^{(t)} \leq \min_{i^*} \sum_{t=1}^{T} c_{i^*}^{(t)} + \text{small margin of error}$$

The main algorithm to solve this is called the **Multiplicative Weights Algorithm**.

23.3.1 Multiplicative Weights Algorithm

- Initialize weights $w_i^{(1)} = 1$ for all $i \in [n]$.
- For each day $t = 1, \ldots, T$:

- Pick action i_t with probability proportional to its weight:

$$\mathbb{P}[\text{Pick action } i] = \frac{w_i^{(t)}}{W^{(t)}}$$

- Observe costs $c_1^{(t)}, \ldots, c_n^{(t)}$
- Update weights for all $i \in [n]$:

$$w_i^{(t+1)} = w_i^{(t)} \cdot (1-\epsilon)^{c_i^{(t)}}$$

Theorem 23.3.1: No regret

Set $\epsilon = \sqrt{\log_2(n)/T}$. Then $\mathbb{E}[\text{Regret}] \leq O(\sqrt{T \log_2(n)})$.

Chapter 24

Lecture 24: Online Algorithms III

We revisit the multiplicative weights algorithm to solve the general experts problem. We have n experts, and T days. On each day $t = 1 \dots T$:

- We specify a distribution $w^{(t)} = (w_1^{(t)}, \dots, w_n^{(t)})$ over experts.
- We observe costs $c^{(t)} = (c_1^{(t)}, \dots, c_n^{(t)})$ in $w_1^{(t)}$ for each expert.
  - The expected cost incurred on day t is:

$$\mathbb{E}[\text{cost}] = \sum_{i=1}^{n} w_i^{(t)} c_i^{(t)} = w^{(t)} \cdot c^{(t)}$$

- Update weights for all $i \in [n]$:

$$w_i^{(t+1)} = w_i^{(t)} \cdot (1 - \epsilon)^{c_i^{(t)}}$$

Our multiplicative weights algorithm guarantees that our average cost is bounded by the aberage cost of the best expert with a small margin:

$$\frac{1}{T} \sum_{t=1}^{T} w^{(t)} \cdot c^{(t)} \leq \min_{i^*} \frac{1}{T} \sum_{t=1}^{T} c_{i^*}^{(t)} + O(\sqrt{\frac{\log(n)}{T}})$$

The small margin, $O(\sqrt{\frac{\log(n)}{T}})$, is called the **regret** of the algorithm. Which is how much worse we do compared to the best expert in hindsight by not trusting them. We see that as the number of days T increases, our regret decreases. Eventually, we see that with enough days, we do as well as the best expert in hindsight. This means that the multiplicative weights algorithm is a **no-regret** algorithm. The regret actually scales with a range of costs. If the costs are in $[a, b]$, then the regret scales as $O((b-a)\sqrt{\frac{\log(n)}{T}})$.

There are many applications of the multiplicative weights algorithm. We will consider a couple applications.

24.1 Application 1: Zero-Sum Games

Recall that in a zero sm game, we have some payoff matrix M and mixed strategies p and q for the row and column player respectively. The score of the game is defined as:

$$\text{Score}(p, q) = \sum_{i,j} p_i q_j M_{ij} = p^T M q$$

The minimax theorem states that there exists an equilibrium value v^* such that:

$$\min_q \max_p \text{Score}(p, q) = \max_p \min_q \text{Score}(p, q) = v^*$$

This means that as long as both the row and column players play optimally, it doesn't matter who plays first, the score will always be v^*. There were two directions to prove this theorem. The easier direction is to show that:

$$\min_q \max_p \text{Score}(p, q) \geq \max_p \min_q \text{Score}(p, q)$$

As allowing the row player to pick p after seeing q can only help them. The harder direction is to show that:

$$\min_q \max_p \text{Score}(p, q) \leq \max_p \min_q \text{Score}(p, q)$$

Where we previously used linear programming duality. However, now we introduce a new proof using the multiplicative weights algorithm and self play. Intuitively, we may think of the different strategies as experts, and the weights as the probability of picking each strategy. As we play the game multiple times, it's equivalent to running the multiplicative weights algorithm on the experts. After sufficiently enough days, we will converge to the best expert in hindsight, which is the optimal mixed strategy.

Proof. For $t = 1, \ldots, T$ rounds:

- Column player picks a mixed strat $q^{(t)} = (q_1^{(t)}, \ldots, q_n^{(t)})$.
- Row player responds with a mixed strat $p^{(t)} = \text{argmax}_{p^{(t)}}\{(p^{(t)})^T M q^{(t)}\}$.
- Define the cost vector $c^{(t)} = (c_1^{(t)}, \ldots, c_n^{(t)}) = p^{(t)} \cdot M$ for columnn player is revealed.
- Column player updates weights using multiplicative weights algorithm with costs $c^{(t)}$.

$$c^{(t)} \cdot q^{(t)} = (p^{(t)})^T M q^{(t)}$$

At the end, we guess the minimax strategies as:

$$\bar{p} = \frac{1}{T} \sum_{t=1}^{T} p^{(t)}, \quad \bar{q} = \frac{1}{T} \sum_{t=1}^{T} q^{(t)}$$

We claim that:

Claim:

$\bar{p}$ and $\bar{q}$ are "approximate" minimax strategies:

$$\underbrace{\max_p pM\bar{q}}_{\text{row player best responds}} \leq \underbrace{\min_q \bar{p}Mq}_{\text{col player best responds}} + O\left(\sqrt{\frac{\log(n)}{T}}\right)$$

Note that:

$$\max_p pM\hat{q} = \max_p \frac{1}{T}\sum_{t=1}^{T} pMq^{(t)} \leq \frac{1}{T}\sum_{t=1}^{T} \max_p pMq^{(t)}$$

This upperbound works because the sum of maxes is always greater than the max of sums, which is equivalent to saying that the row player can best respond to each $q^{(t)}$ individually. Now note that:

$$\begin{aligned}
\frac{1}{T}\sum_{t=1}^{T} \max_p pMq^{(t)} &= \frac{1}{T}\sum_{t=1}^{T} p^{(t)}Mq^{(t)} \\
&\leq \min_{i\in[n]} \frac{1}{T}\sum_{t=1}^{T} p^{(t)}Mq^{(t)} + O\left(\sqrt{\frac{\log(n)}{T}}\right) \\
&= \min_{i\in[n]} (\bar{p}M)_i + O\left(\sqrt{\frac{\log(n)}{T}}\right) \\
&= \min_q \bar{p}Mq + O\left(\sqrt{\frac{\log(n)}{T}}\right)
\end{aligned}$$

Now see that:

$$\begin{aligned}
\min_q \max_p pMq &\leq \max_p pM\bar{q} \\
&\leq \min_q \bar{p}Mq + O\left(\sqrt{\frac{\log(n)}{T}}\right) \\
&\leq \max_p \min_q pMq + O\left(\sqrt{\frac{\log(n)}{T}}\right)
\end{aligned}$$

By letting $T \to \infty$, we have:

$$\min_q \max_p pMq \leq \max_p \min_q pMq$$

□

Chapter 25

Lecture 25: Quantum Computing

Quantum computing is a new model of computation that uses principles from quantum mechanics to perform computations. The basic unit of quantum information is the **qubit**, which can exist in a superposition of states. This allows quantum computers to perform certain computations much faster than classical computers.

Now suppose that we have n electrons. Each electron can have a particular spin of either up or down. Hence, there is a total of 2^n possible configuration of spins. We introduce the **Quantum Simulation Problem**, where we are given a starting configuration of n electrons, and want to determine the configuration after T time steps. Classically, the best algorithm for this problem runs in time $O(2^n)$.

In this chapter, we will explore how quantum computers can solve certain problems more efficiently than classical computers.

25.1 Deutsh-Jozsa Problem

Given an input function of $f : \{0,1\}^n \to \{0,1\}$, either:

- $f(x) = 0$ for all x (constant function)
- $f(x) = 1$ for all x (constant function)
- $f(x) = 0$ for exactly half of the inputs, and $f(x) = 1$ for the other half (balanced function)

For Deterministic algorithms, we need to query f at least $2^{n-1} + 1$ times to determine if f is constant or balanced. A quantum computer though, can solve this problem in $O(n)$ time, but this is unsatisfactory, as an approximation algorithm can solve this problem in $O(1)$ time with high probability.

25.2 Bernstein-Vazirani Problem

Given an input function of $f : \{0,1\}^n \to \{0,1\}$, such that there exists a hidden string $s \in \{0,1\}^n$ where:

$$f(x) = s \cdot x = \sum_{i=1}^{n} s_i x_i \mod 2$$

The goal is to determine the hidden string s. Classically, we need to query f at least $n+1$ times to determine s. However, a quantum computer can solve this problem in $O(1)$ time.

25.3 Shor's algorithm

Shor's algorithm is a quantum algorithm for integer factorization. Given a composite integer N, the goal is to find its prime factors. Classically, the best known algorithm for integer factorization runs in sub-exponential time. However, Shor's algorithm can factor integers in polynomial time $O(n^2)$ on a quantum computer. This has significant implications for cryptography, as many encryption schemes rely on the difficulty of factoring large integers.

25.4 Grover's algorithm

Circuit-SAT is an NP-complete problem where the best classical algorithm runs in $O(2^n)$ time. Grover's algorithm can solve circuit-SAT in $O(2^{n/2})$ time on a quantum computer. Actually, Grover's algorithm can be used to search an unsorted database of N items in $O(\sqrt{N})$ time, which is a quadratic speedup over classical search algorithms.

25.5 Quantum Speedups

There are three main types of quantum speedups:

- **Shor-type speedups:** Exponentially faster than classical but only for certain problems (e.g. integer factorization).
- **Grover-type speedups:** Works for many problems, but only polynomially faster than classical
- **Physics Simulation-type speedups:** Exponentially faster for simulating physical systems.

Interestingly, we can talk about complexity classes for quantum computing. The class of decision problems that can be solved by a quantum computer in polynomial time is called **BQP** (Bounded-error Quantum Polynomial time). It is known that $P \subseteq BQP$, since any deterministic polynomial time algorithm is also a quantum algorithm with no error. For example:

- Factoring is in BQP (Shor's algorithm) and NP
- Circuit-SAT is in NP but not in BQP (Grover's algorithm only gives a quadratic speedup)
- Physics Simulation is in BQP but not known to be in NP

25.6 Building a Quantum Computer

It's difficult to build a quantum computer because of noise and outside interference. Interfering particles could potentially flip the state of a qubit, causing errors in computation. To mitigate this, quantum error correction techniques are employed, where multiple physical qubits are used to represent a single logical qubit.

25.7 Quantum Mechanics

A classical computer has classical bits that are either 0 or 1. A quantum computer has quantum bits, or qubits, that can be $|0\rangle$, $|1\rangle$, called ket 0 and ket 1. Besides being in either state, it can be in a superposition of $|0\rangle$ and $|1\rangle$:

$$|\psi\rangle = x \cdot |0\rangle + y \cdot |1\rangle$$

where $|psi\rangle$ denotes a superposition and $x, y \in \mathbb{C}$ are complex numbers called amplitudes. The amplitudes must satisfy the normalization condition:

$$|x|^2 + |y|^2 = 1$$

For the sake of this class, we will pretend that $x, y \in \mathbb{R}$. There are two special states called **plus state** and **minus state**:

$$|+\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle, \quad |-\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$$

Now note that when we measure a particular qubit, we get either $|0\rangle$ or $|1\rangle$ with probabilities $|x|^2$ and $|y|^2$ respectively. We can plot this in 2D space on a unit circle:

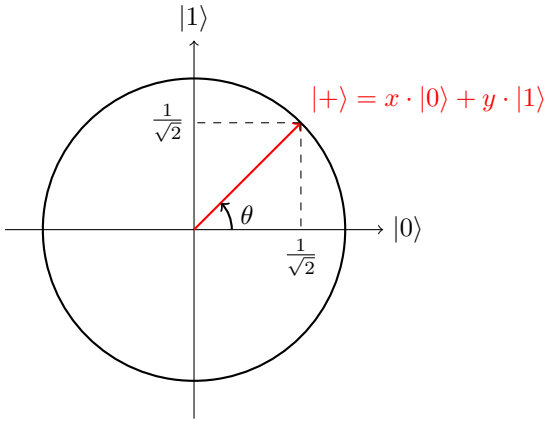


Figure 25.1: Qubit representation on 2D unit circle

In this representation, we can characterize the state of a qubit by an angle θ:

$$|+\rangle = \cos(\theta)|0\rangle + \sin(\theta)|1\rangle$$

25.7.1 Quantum Measurement

To interact with a qubit with state $|\phi\rangle = x \cdot |0\rangle + y \cdot |1\rangle$, we can measure it. With probability $|x|^2$, we get outcome $|0\rangle$ and the qubit collapses to state $|0\rangle$. With probability $|y|^2$, we get outcome $|1\rangle$ and the qubit collapses to state $|1\rangle$. This is called the **collapse of the wavefunction**. By interacting with a qubit, we can only get classical information out of it. In the plus state, we have equal probability of getting $|0\rangle$ or $|1\rangle$. Simiarly, for the minus state, we also have equal probability of getting $|0\rangle$ or $|1\rangle$.

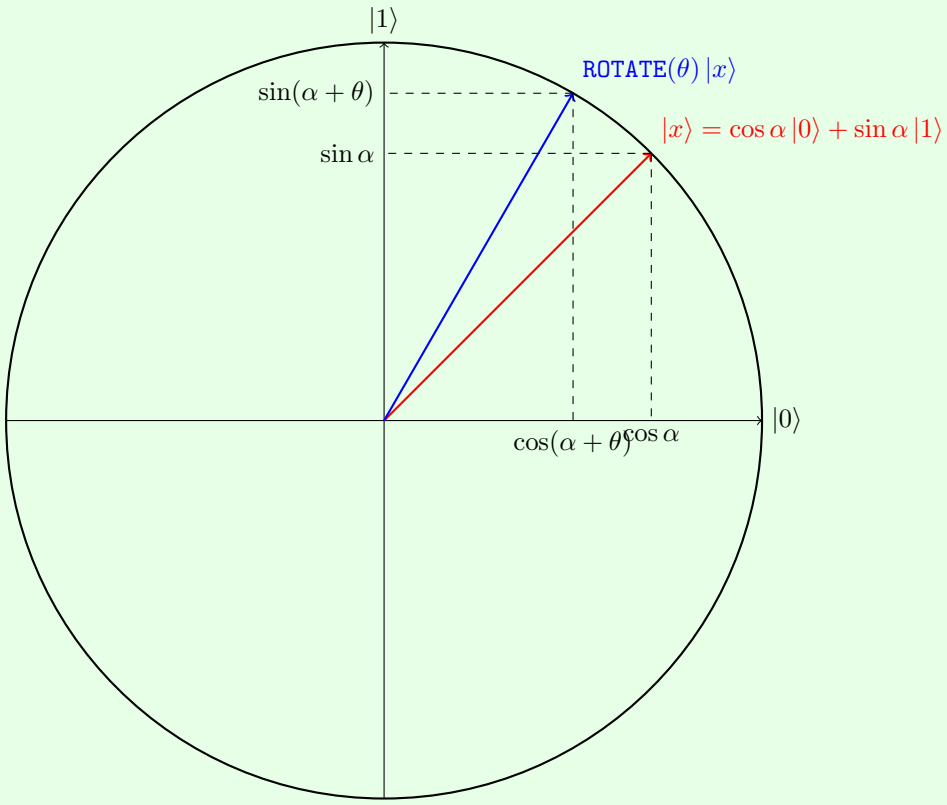
25.8 Quantum Operations and Algorithms

To create quantum algorithms, we need to be able to manipulate qubits. We can do this using quantum operations.

Definition 25.8.1: Rotation Operation $\texttt{ROTATE}(\theta)$

If $|x\rangle = \cos\alpha\,|0\rangle + \sin\alpha\,|1\rangle$, then:

$$\texttt{ROTATE}(\theta)\,|x\rangle = \cos(\alpha+\theta)\,|0\rangle + \sin(\alpha+\theta)\,|1\rangle$$



25.8.1 Elitzur-Vaidman Bomb Algorithm

Suppose we have a bomb that will explode if it interacts with a photon. We want to determine if the bomb is functional without exploding it. Classically, we can only test the bomb by sending a photon towards it, which will cause it to explode if it is functional. However, using quantum mechanics, we can determine if the bomb is functional without exploding it. To test it:

- If the bomb is not functinoal, and we send $|x\rangle = x\,|0\rangle + y\,|1\rangle$ towards it, the state remains the same.
- If the bomb is functional, and we send $|x\rangle = x\,|0\rangle + y\,|1\rangle$ towards it. A sensor measures ψx the state collapses to $|0\rangle$ with probability $|x|^2$ and explodes with probability $|y|^2$.

The classical approach is to send a photon towards the bomb. If it explodes, we know it is functional. If it doesn't explode, we know it is not functional. This approach has a 50% chance of exploding the bomb if it is functional. However, we can use quantum mechanics to improve this.

25.8.2 Quantum Solution to Elitzur-Vaidman Bomb Problem

- Initialize the state as $|\psi_0\rangle = |0\rangle$.
- Rotate by $\frac{\pi}{4}$, $\texttt{ROTATE}(\frac{\pi}{4})\,|\psi_0\rangle = |\psi_1\rangle = \frac{1}{\sqrt{2}}\,|0\rangle + \frac{1}{\sqrt{2}}\,|1\rangle$.
- Send $|\psi_1\rangle$ towards the bomb.
- Measure $|\psi_1\rangle$:
  - If 0 is observed, output "Bomb is functional".
  - If 1 occurs, output "Not sure if bomb is functional".

If there is no bomb, the state is is rotated to the $|1\rangle$ state with probability 1, so we always output not sure.

$$\texttt{ROTATE}(\frac{\pi}{4})\,|0\rangle = |+\rangle = \frac{1}{\sqrt{2}}\,|0\rangle + \frac{1}{\sqrt{2}}\,|1\rangle$$
$$\texttt{ROTATE}(\frac{\pi}{4})\,|+\rangle = |1\rangle = 0\cdot|0\rangle + 1\cdot|1\rangle$$

If the bomb is functional, then the bomb will measure $|+\rangle$, and with half probability, the bomb will observe 0 and we output "Bomb is functional" without exploding it. With half probability we observe 1 and the bomb explodes. Hence, we have a 25% chance of detecting a functional bomb without exploding it. A 50% chance of exploding it, and a 25% chance of being unsure.

25.8.3 Improved Quantum Solution to Elitzur-Vaidman Bomb Problem

We can boost the probability of detecting a functional bomb without exploding it by repeating the process multiple times. We do this by rotating by a smaller angle $\theta = \frac{\pi}{2N}$ for some large integer N. We repeat the following process N times:

- Rotate by θ: $|\psi_i\rangle = \texttt{ROTATE}(\theta)\,|\psi_{i-1}\rangle$.
- Send $|\psi_i\rangle$ towards the bomb.
- Measure $|\psi_i\rangle$:
  - If 0 is observed, output "Bomb is functional".
  - If 1 occurs, output "No Bomb.

If there is no bomb, after N iterations, the state is rotated to $|1\rangle$ with probability 1, so we always output not sure. If the bomb is functional, then at each iteration, the bomb will measure $|\psi_i\rangle$, and with probability $\cos^2(\theta)$, the bomb will observe 1 and continue to the next iteration. With probability $\sin^2(\theta)$, the bomb will observe 0 and we output "Bomb is functional". Hence, the probability of detecting a functional bomb without exploding it is:

$$P(\text{Detect functional bomb}) = 1 - P(\text{explosion})$$

We have:

$$\begin{aligned} P(\text{explosion}) &= \sum_{i=1}^{N} P(\text{explosion at time } i) \\ &\leq P(\cos\theta\,|0\rangle + \sin\theta\,|1\rangle \text{ survives } N \text{ iterations}) \\ &= N \cdot \sin^2(\theta) \leq N \cdot \theta^2 \\ &= N \cdot \frac{\pi^2}{4N^2} = \frac{\pi^2}{4N} \\ &\leq \frac{2.5}{N} \end{aligned}$$