CSS.201.1 Algorithms

Instructor: Umang Bhaskar

TIFR 2024, Aug-Nov

Scribe: Soham Chatterjee

soham.chatterjee@tifr.res.in
Website: sohamch08.github.io

Abstract

These notes comprise a scribed record of the lectures for the *Algorithms* course conducted at the TIFR during the August–November 2024 semester, instructed by Prof. Umang Bhaskar. The official course webpage is available at this link.

The primary reference for the course was *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein (CLRS), complemented by supplementary materials and the lecturer's own notes, which are accessible through the course webpage. For the chapter on approximation algorithms, chapters 14, 15, and 17 of *Approximation Algorithms* by Vijay Vazirani were also used.

These notes were prepared both as a personal academic resource and to facilitate revision for other students. They have been typeset using a custom theme developed by me. The source code of the notes is available at this repository, and the source code of the theme can be found in my GitHub repository. Any corrections, comments, or suggestions are welcome and reach out to me at soham.chatterjee@tifr.res.in.

Contents

Chapter 1

Finding Closest Pair of Points

Find Closest
Input: Set $S = \{(x_i, y_i) \mid x_i, y_i \in \mathbb{R},\ \forall\ i \in [n]\}$. We denote $P_i = (x_i, y_i)$.
Question: Given a set of points find the closest pair of points in $\mathbb{R}^2$ find P_i, P_j that are at minimum l_2 distance i.e. minimize $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.

1.1 Naive Algorithm

Now the naive algorithm for this will be checking all pairs of points and take their distance and output the minimum one. There are total $\binom{n}{2}$ possible choices of pairs of points. And calculating the distance of each pair takes $O(1)$ time. So it will take $O(n^2)$ times to find the closest pair of points.

Idea: $\forall\ P_i, P_j \in S$ find distance $d(P_i, P_j)$ and return the minimum. Time taken is $O(n^2)$.

1.2 Divide and Conquer Algorithm

Below we will show a Divide and Conquer algorithm which gives a much faster algorithm.

Definition 1.2.1: Divide and Conquer

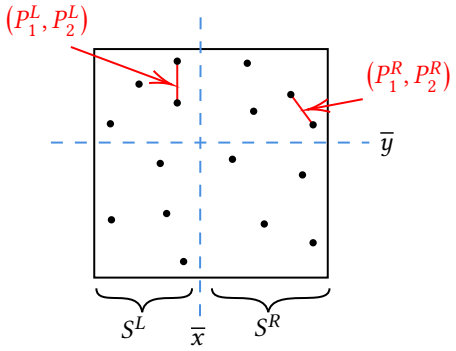
- Divide: Divide the problem into two parts (roughly equal)
- Conquer: Solve each part individually recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
- Combine: Combine the solutions to the subproblems into the solution.

1.2.1 Divide

So to divide the problem into two roughly equal parts we need to divide the points into two equal sets. That we can do by sorting the points by their $x-$coordinate. Suppose S^x denote we get the new sorted array or points. And similarly we obtain S^y which denotes the array of points after sorting S by their $y-$coordinate.

Algorithm 1: Step 1 (Divide)

Function *Divide*:
- Sort S by $x-$coordinate and $y-$coordinate
- $S^x \longleftarrow S$ sorted by $x-$coordinate
- $S^y \longleftarrow S$ sorted by $y-$coordinate
- $\bar{x} \longleftarrow \lfloor \frac{n}{2} \rfloor$ highest $x-$coordinate
- $\bar{y} \longleftarrow \lfloor \frac{n}{2} \rfloor$ highest $y-$coordinate
- $S^L \longleftarrow \{P_i \mid x_i < \bar{x},\ \forall\ i \in [n]\}$
- $S^R \longleftarrow \{P_i \mid x_i \geq \bar{x},\ \forall\ i \in [n]\}$



1.2.2 Conquer

Now we will recursively get the pair of closest points in S_L and S_R. Suppose the (P_1^L, P_2^L) are the closest pair of points in S^L and (P_1^R, P_2^R) are the closest pair of points in S^R.

Algorithm 2: Step 1 (Solve Subproblems)

Function *Conquer*:
- Solve for S_L, S^R.
- (P_1^L, P_2^L) are the closest pair of points in S_L.
- (P_1^R, P_2^R) are the closest pair of points in S_R.
- $\delta^L = d(P_1^L, P_2^L)$, $\delta^R = d(P_1^R, P_2^R)$
- $\delta_{min} \longleftarrow \min\{\delta^L, \delta^R\}$

1.2.3 Combine

Now we want to combine these two solutions.

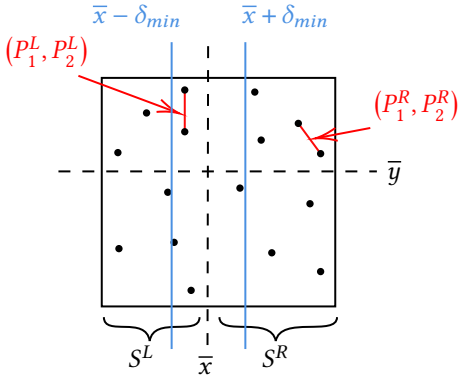
Question 1.1: We are not done

Is there a pair of points $P_i, P_j \in S$ such that $d(P_i, P_j) < \delta_{min}$

If Yes:

- One of them must be in S_L and the other is in S_R.
- $x-$coordinate $\in [\bar{x} - \delta_{min}, \bar{x} + \delta_{min}]$.
- $|y_i - y_j| \leq \delta_{min}$

So we take the strip of radius δ_{min} around $\bar{x}$. Define $T = \{P_i \in S \mid |x_i - \bar{x}| \leq \delta_{min}\}$

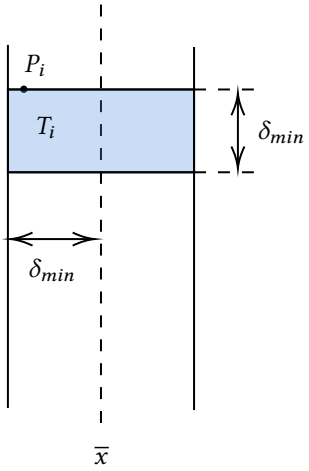


We now sort all the points in the T by their decreasing $y-$coordinate. Let T_y be the array of points. For each $P_i \in T_y$ define the region

$$T_i = \{P_j \in T_y \mid 0 \leq y_j - y_i \leq \delta_{min}, j > i\}$$

Lemma 1.2.1

Number of points (other than P_i) that lie inside the box is at most 8



Proof: Suppose there are more than 8 points that lie inside the box apart from P_i. The box has a left square part and a right square part. So one of the squares contains at least 5 points. WLOG suppose the left square has at least 5 points. Divide each square into 4 parts by a middle vertical and a middle horizontal line. Now since there are 5 points there is one part which contains 2 points, but that is not possible as those two points are in S_L and their distance will be less than δ_{min} which is not possible. Hence, contradiction. Therefore, there are at most 8 points inside the box. ■

Hence by the above lemma for each $P_i \in T_y$ there are at most 8 points in T_i. So for each $P_j \in T_i$ we find the $d(P_i, P_j)$ and if it is less than δ_{min} we update the points and the distance

1.2.4 Pseudocode and Time Complexity

Assumption. *We will assume for now that for all $P_i.P_j \in S$ we have $x_i \neq x_j$ and $y_i \neq y_j$. Later we will modify the pseudocode to remove this assumption*

Algorithm 3: Find-Closest(S)

Input: Set of n points, $S = \{(x_i, y_i) \mid x_i, y_i \in \mathbb{R},\ \forall\ i \in [n]\}$. We denote $P_i = (x_i, y_i)$.
Output: Closest pair of ponts, (P_i, P_j, δ) where $\delta = d(P_i, P_j)$

```
1  begin
2      if |S| ≤ 10 then
3          Solve by Brute Force (Consider every pair of points)
4      S^x ⟵ S sorted by x−coordinate,        S^y ⟵ S sorted by y−coordinate
5      x̄ ⟵ ⌊n/2⌋ highest x−coordinate,        ȳ ⟵ ⌊n/2⌋ highest y−coordinate
6      S^L ⟵ {P_i | x_i < x̄, ∀ i ∈ [n]},      S^R ⟵ {P_i | x_i ≥ x̄, ∀ i ∈ [n]}
7      (P_1^L, P_2^L, δ^L) ⟵ Find-Closest(S^L),    (P_1^R, P_2^R, δ^R) ⟵ Find-Closest(S^R)
8      δ_min ⟵ min{δ^L, δ^R}
9      if δ_min < δ^L then
10         P_1 ⟵ P_1^R, P_2 ⟵ P_2^R
11     else
12         P_1 ⟵ P_1^L, P_2 ⟵ P_2^L
13     T ⟵ {P_i | |x_i − x̄| ≤ δ_min}
14     T_y ⟵ T sorted by decreasing y−coordinate
15     for P ∈ T_y do
16         U ⟵ Next 8 points
17         for P̂ ∈ U do
18             if d(P, P̂) < δ_min then
19                 δ_min ⟵ d(P, P̂)
20                 (P_1, P_2) ⟵ (P, P̂)
21     return (P_1, P_2, δ_min)
```

Notice we used the assumption in the line 5 for finding the medians. So the line 4 takes $O(n \log n)$ times. Lines 5,6 takes $O(n)$ time. Since $\overline{x}$ is the median, we have $|S^L| = \lfloor \frac{n}{2} \rfloor$ and $|S^R| = \lceil \frac{n}{2} \rceil$. Hence Find-Closest($S^L$) and Find-Closest($S^R$) takes $T\left(\frac{n}{2}\right)$ time. Now lines $8-12$ takes constant time. Line 13 takes $O(n)$ time. And line 14 takes $O(n \log n)$ time. Since U has 8 points i.e. constant number of points the lines $16-20$ takes constant time for each $P \in T_y$. Hence the for loop at

line 15 takes $O(n)$ time. Hence total time taken

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

1.3 Improved Algorithm for $O(n \log n)$ Runtime

Notice once we sort the points by $x-$coordinate and $y-$coordinate we don't need to sort the points anymore. We can just pass the sorted array of points into the arguments for solving the smaller problems. Their is another time where we need to sort which is in line 14 of the above algorithm. This we can get actually from S^y without sorting just checking one by one backwards direction if the $x-$coordinate of the points satisfy $|x_i - \overline{x}| \leq \delta_{min}$. So

$$T_y = \text{REVERSE}(\{P_i \in S^y \mid |x_i - \overline{x}| \leq \delta_{min}\})$$

So we form a new algorithm which takes the input S^x and S^y and then finds the closest pair of points. Then we will use that subroutine to find closest pair of points in any given set of points.

Algorithm 4: FIND-CLOSEST-SORTED(S^x, S^y)

Input: Set of n points, $S = \{(x_i, y_i) \mid x_i, y_i \in \mathbb{R},\ \forall\ i \in [n]\}$. S^x and S^y are the sorted array of points with respect to $x-$coordinate and $y-$coordinate respectively

Output: Closest pair of ponts, (P_i, P_j, δ) where $\delta = d(P_i, P_j)$

```
1  begin
2      if |S| ≤ 10 then
3          Solve by Brute Force
4      x̄ ⟵ ⌊n/2⌋ highest x−coordinate
5      ȳ ⟵ ⌊n/2⌋ highest y−coordinate
6      S^L ⟵ {P_i ∈ S^x | x_i < x̄, ∀ i ∈ [n]}
7      S^L_y ⟵ {P_i ∈ S^y | x_i < x̄}
8      S^R ⟵ {P_i ∈ S^x | x_i ≥ x̄, ∀ i ∈ [n]}
9      S^R_y ⟵ {P_i ∈ S^y | x_i ≥ x̄}
10     (P^L_1, P^L_2, δ^L) ⟵ FIND-CLOSEST-SORTED(S^L, S^L_y)
11     (P^R_1, P^R_2, δ^R) ⟵ FIND-CLOSEST-SORTED(S^R, S^R_y)
12     δ_min ⟵ min{δ^L, δ^R}
13     if δ_min < δ^L then
14         P_1 ⟵ P^R_1, P_2 ⟵ P^R_2
15     else
16         P_1 ⟵ P^L_1, P_2 ⟵ P^L_2
17     T ⟵ {P_i | |x_i − x̄| ≤ δ_min}
18     T_y ⟵ REVERSE({P_i ∈ S^y | |x_i − x̄| ≤ δ_min})
19     for P ∈ T_y do
20         U ⟵ Next 8 points
21         for P̂ ∈ U do
22             if d(P, P̂) < δ_min then
23                 δ_min ⟵ d(P, P̂)
24                 (P_1, P_2) ⟵ (P, P̂)
25     return (P_1, P_2, δ_min)
```

Algorithm 5: FIND-CLOSEST(S)

Input: Set of n points, $S = \{(x_i, y_i) \mid x_i, y_i \in \mathbb{R},\ \forall\ i \in [n]\}$. We denote $P_i = (x_i, y_i)$.

Output: Closest pair of ponts, (P_i, P_j, δ) where $\delta = d(P_i, P_j)$

```
1  begin
2      if |S| ≤ 10 then
3          Solve by Brute Force
4      S^x ⟵ S sorted by x−coordinate
5      S^y ⟵ S sorted by y−coordinate
6      return FIND-CLOSEST-SORTED(S^x, S^y)
```

This algorithm only sorts one time. So time complexity for FIND-CLOSEST-SORTED(S^x, S^y) is

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

and therefore times complexity for FIND-CLOSEST(S) is $O(n \log n)$.

1.4 Removing the Assumption

For this there nothing much to do. For finding the median $\overline{x}$ if we have more than one points with same $x-$coordinate which appears as the $\lfloor \frac{n}{2} \rfloor$ highest $x-$coordinate we sort only those points with respect to their $y-$coordinate update the S^x like that and then take $\lfloor \frac{n}{2} \rfloor$ highest point in S^x. We do the same for S^y and update accordingly. All this we do so that S^L and S^R has the size $\frac{n}{2}$.

Chapter 2

Median Finding in Linear Time

MEDIAN FIND
Input: Set S of n distinct integers
Question: Find the $\lfloor \frac{n}{2} \rfloor^{th}$ smallest integer in S

2.1 Naive Algorithm

The naive algorithm for this will be to sort the array in $O(n \log n)$ time then return the $\lfloor \frac{n}{2} \rfloor^{th}$ element. This will take $O(n \log n)$ time. But in the next section we will show a linear time algorithm.

2.2 Linear Time Algorithm

In this section we will show an algorithm to find the median of a given set of distinct integers in $O(n)$ time complexity. Consider the following two problems:

RANK-FIND (S, k)
Input: Set S of n distinct integers and an integer $k \leq n$
Question: Find the k^{th} smallest integer in S

APPROXIMATE-SPLIT(S)
Input: Set S of n distinct integers
Question: Given S, return an integer $z \in S$ such that z where $rank(z) \in \left[\frac{n}{4}, \frac{3n}{4}\right]$

2.2.1 Solve RANK-FIND using APPROXIMATE-SPLIT

Algorithm 6: RANK-FIND(S,k)

Input: Set S of n distinct integer and $k \in [n]$
Output: k^{th} smallest integer in S

```
1 begin
2     if |S| ≤ 100 then
3         Sort S, return k^th smallest element in S
4     z ⟵ APPROXIMATE-SPLIT(S)        (z is the r^th smallest element for some r ∈ [n/4, 3n/4])
5     S_L ⟵ {x ∈ S | x ≤ z}, S_R ⟵ {x ∈ S | x > z}
6     if k ≤ |S_L| then
7         return RANK-FIND(S_L, k)
8     return RANK-FIND(S_R, k − |S_L|)
```

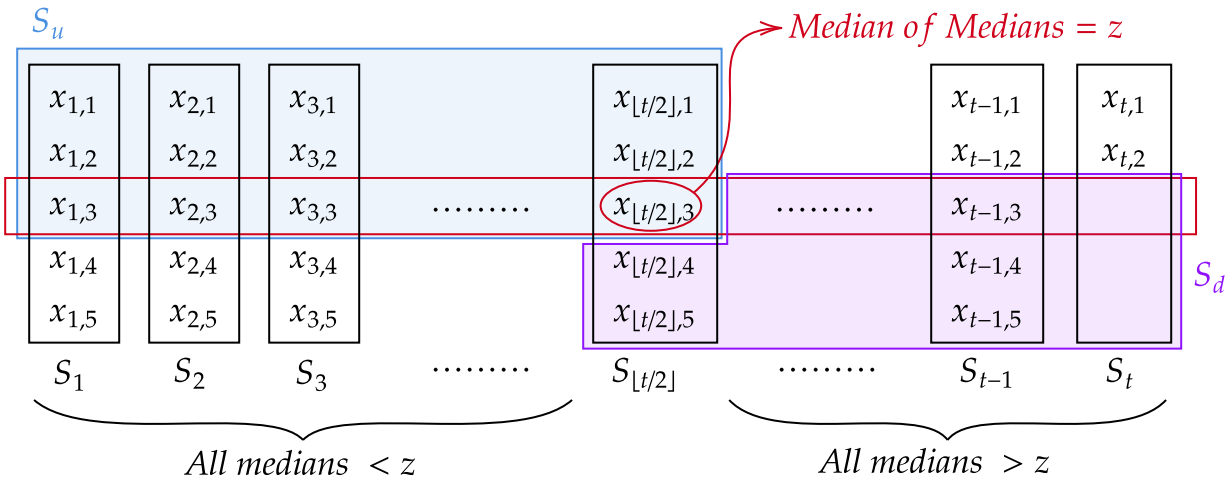
Certainly if we can solve Rank-Find(S, k) for all $k \in [n]$ we can also solve Median-Find. We will try to use both the problems and recurse to solve Rank-Find in linear time.

In the above algorithm $rank(z) \in \left[\frac{n}{4}, \frac{3n}{4}\right]$. So $\frac{n}{4} \leq |S_L|, |S_R| \leq \frac{3n}{4}$. For now suppose Rank-Find(S, k) takes $T_{RF}(n)$ time and Approximate-Split(S) takes $T_{AS}(n)$ time. Then the time taken by the algorithm is

$$T_{RF}(n) \leq O(n) + T_{AS}(n) + T_{RF}\left(\frac{3n}{4}\right)$$

2.2.2 Solve Approximate-Split using Rank-Find

We first divide S into groups of 5 elements. So take $t = \lceil \frac{n}{5} \rceil$. Now we sort each group. Since each group have constant size this can be done in $O(n)$ time. So now consider the scenario:



After sorting each of the groups we takes the medians of each group. Let z be the median of the medians. We claim that $rank(z) \in \left[\frac{n}{4}, \frac{3n}{4}\right]$.

Algorithm 7: Approximate-Split(S)

Input: Set S of n distinct integers
Output: An integer $z \in S$ such that z where $rank(z) \in \left[\frac{n}{4}, \frac{3n}{4}\right]$

```
1  begin
2      if |S| ≤ 100 then
3          Sort, return Exact median
4      t ⟵ ⌈n/5⌉
5      S_i ⟵ i^th block of 5 elements in S for i ∈ [t − 1]
6      S_t ⟵ Whatever is left in S
7      for i ∈ [t] do
8          Sort S_i, Let h_i be the median of S_i
9      T ⟵ {h_i | i ∈ [t]}
10     return Rank-Find(T, ⌊t/2⌋)
```

So in the picture among elements in upper left the highest element is z and among the elements in lower right the lowest element is z. We will show that the number of elements smaller than z is between $\frac{n}{4}$ and $\frac{3n}{4}$. Lets call the set of elements in upper left box is S_u and the set of elements in lower right box is S_d.

Lemma 2.2.1
$|S_u|, |S_d| \geq \frac{n}{4}$

Proof: $|S_u| \geq 3 \times \left\lfloor \frac{t}{2} \right\rfloor$. For $n \geq 100$, $3\left\lfloor \frac{t}{2} \right\rfloor > \frac{n}{4}$. Hence $|S_u| \geq \frac{n}{4}$. Now similarly $|S_d| \geq 3\left\lfloor \frac{t}{2} - 1 \right\rfloor \geq \frac{n}{4}$. ■

Lemma 2.2.2
Number of elements in S smaller than z lies between $\frac{n}{4}$ and $\frac{3n}{4}$.

Proof: Now number of elements in S smaller than $z \geq |S_u| \geq \frac{n}{4}$. The number of elements greater than $z \geq |S_d| \geq \frac{n}{4}$. So number of elements in S smaller than $z \leq n-$ number of elements greater than $z \leq n - \frac{n}{4} = \frac{3n}{4}$. ■

Hence the APPROXIMATE-SPLIT(S) takes time

$$T_{AS}(n) = O(n) + T_{RF}\left(\frac{n}{5}\right)$$

2.2.3 Pseudocode and Time Complexity

Hence using APPROXIMATE-SPLIT the final algorithm for RANK-FIND is the following:

Algorithm 8: RANK-FIND(S,k)

Input: Set S of n distinct integer and $k \in [n]$
Output: k^{th} smallest integer in S

```
1  begin
2      if |S| ≤ 100 then
3          Sort S, return k^th smallest element in S
4      t ⟵ ⌈n/5⌉
5      S_i ⟵ i^th block of 5 elements in S for i ∈ [t − 1]
6      S_t ⟵ Whatever is left in S
7      for i ∈ [t] do
8          Sort S_i, Let h_i be the median of S_i
9      T ⟵ {h_i | i ∈ [t]}
10     z ⟵ RANK-FIND(T, ⌊t/2⌋)
11     S_L ⟵ {x ∈ S | x ≤ z}, S_R ⟵ {x ∈ S | x > z}
12     if k ≤ |S_L| then
13         return RANK-FIND(S_L, k)
14     return RANK-FIND(S_R, k − |S_L|)
```

Replacing $T_{AS}(n)$ in the time complexity equation of $T_{RF}(n)$ we get the equation:

$$T_{RF}(n) \leq O(n) + T_{RF}\left(\frac{n}{5}\right) + T_{RF}\left(\frac{3n}{4}\right)$$

Let $T_{RF}(n) \leq kn + +T_{RF}\left(\frac{n}{5}\right) + T_{RF}\left(\frac{3n}{4}\right)$. We claim that $T_{RF}(n) \leq cn$ for some $c \in \mathbb{N}$ for all $n \geq n_0$ where $n_0 \in \mathbb{N}$. By induction we have

$$T_{RF}(n) \leq kn + \frac{cn}{5} + \frac{3cn}{4} = \left(k + \frac{19c}{20}\right)n$$

To have $k + \frac{19c}{20} \leq c$ we have to have $k + \frac{19c}{20} \leq c \iff c \geq 20k$. So take $c \geq 20k$ and our claim follows. Hence $T_{RF}(n) = O(n)$. Since we can find any k^{th} smallest number in a given set of distinct integers in linear time we can also find the median in linear time.

Chapter 3

Polynomial Multiplication

Polynomial Multiplication

Input: Given 2 univariate polynomials of degree $n-1$ by 2 arrays of their coefficients $(a_0, \ldots, a_{n-1})$ and $(b_0, \ldots, b_{n-1})$ such that $A(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$ and $B(x) = b_0 + b_1 x + \cdots + b_{n-1}x^{n-1}$ respectively

Question: Given 2 polynomials of degree $n-1$ find their product polynomial $C(x) = A(x)B(x)$ of degree $2n-2$ by returning the array of their coefficients.

3.1 Naive Algorithm

We can do this naively by calculating each coefficient of C in $O(n)$ time since for any $i \in \{0, \ldots, 2n-2\}$

$$c_i = \sum_{j=0}^{i} a_j b_{i-j}$$

Since there are $2n-1 = O(n)$ total coefficients of C it takes total $O(n^2)$ time. In the following section we will do this in $O(n \log n)$ time.

3.2 Strassen-Schönhage Algorithm

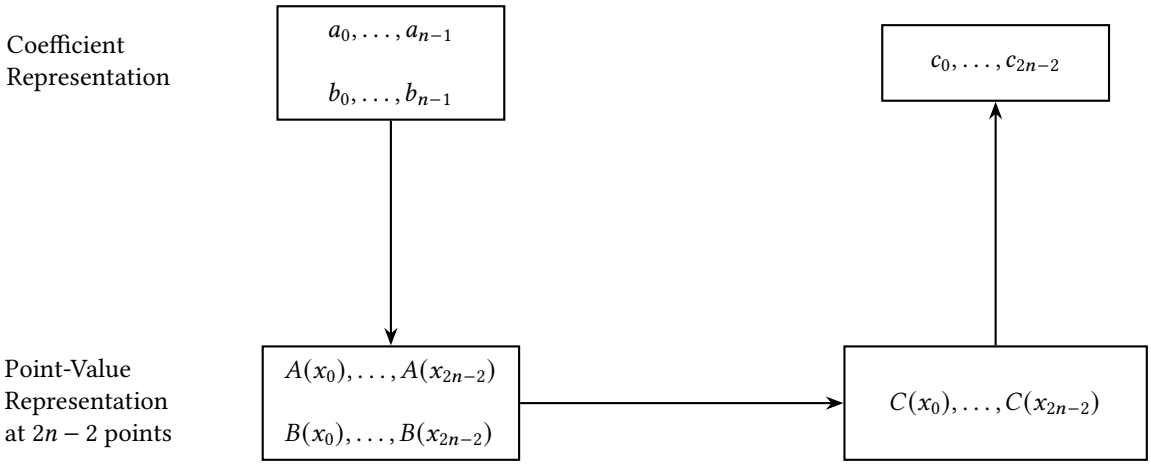
Before diving into the algorithm first let's consider how many ways we can represent a polynomial. Often changing the representation helps to solve the problem in less time.

- **Coefficients**: We can represent a polynomial by giving the array of all its coefficient.
- **Point-Value Pairs**: We can evaluate the polynomial in distinct n points and give all the point-value pairs. This also uniquely represents a polynomial since there is exactly one polynomial of degree $n-1$ which passes through all these points.

Theorem 3.2.1

Given n distinct points $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ in $\mathbb{R}^2$ with $x_i \neq x_j$ for all $i \neq j$ there is a unique $(n-1)$−degree polynomial $P(x)$ such that $P(x_i) = y_i$ for all $i \in [\![n-1]\!]$

Since we want to find the polynomial $C(x) = A(x)B(x)$ and $C(x)$ has degree $2n-2$, we will evaluate the polynomials $A(x)$ and $B(x)$ in $2n-1$ distinct points. So we will have the algorithm like this:



3.2.1 Finding Evaluations of Multiplied Polynomial

Suppose we were given $A(x)$ and $B(x)$ evaluated at $2n-1$ distinct points $x_0, \ldots, x_{2n-2}$. Then we can get $C(x)$ evaluated at $x_0, \ldots, x_{2n-2}$ by

$$C(x_i) = A(x_i)B(x_i) \; \forall \; i \in [\![2n-2]\!]$$

Since there are $O(n)$ many points and for each point it takes constant time to multiply we can find evaluations of C at $x_0, \ldots, x_{2n-2}$ in $O(n)$ time.

3.2.2 Evaluation of a Polynomial at Points

Question 3.1

Suppose there is only one point, x_0. Can we evaluate an $n-1$ degree polynomial $A(x) = \sum_{i=0}^{n-1} a_i x^i$ at x_0 efficiently?

We can rewrite $A(x)$ as

$$A(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots (a_{n-1} + x(a_n)) \cdots)))$$

In this represent it is clear that we have to do n additions and n multiplications to find $A(x_0)$. Hence, we can evaluate an $n-1$ degree polynomial at a point in $O(n)$ time

But we have $O(n)$ points. And if each point takes $O(n)$ time to find the evaluation of the polynomial then again it will take total $O(n^2)$ time. We are back to square one. So instead we will evaluate the polynomial in some special points, and we will evaluate in all of them in $O(n \log n)$ time. So now the problem we will discuss now is to find some special n points where we can evaluate an $n-1$-degree polynomial in $O(n \log n)$ time.

Idea: Evaluate at roots of unity and use Fast Fourier Transform

Assume n is a power of 2. We have the polynomial $A(x) = \sum_{i=0}^{n-1} a_i x^i$. So now consider the following two polynomials

$$A_0(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{\frac{n}{2}-1} \qquad A_1(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{\frac{n}{2}-1}$$

Certainly we have

$$A(x) = A_0(x^2) + xA_1(x^2)$$

Hence we can get $A(1)$ and $A(-1)$ by

$$A(1) = A_0(1) + A_1(1) \qquad A(-1) = A_0(1) - A_1(1)$$

Hence like this by evaluating two $\frac{n}{2} - 1$ degree polynomials at one point we get evaluation of A at two points. More generally for any $y \geq 0$ we have

$$A(\sqrt{y}) = A_0(y) + \sqrt{y}A_1(y) \qquad A(-\sqrt{y}) = A_0(y) - \sqrt{y}A_1(y)$$

So by recursing like this evaluating at $1, -1$ we can get evaluations of A at n^{th} roots of unity.

Let

$$\omega_n^k = n^{th} \text{ root of unity for } k \in [\![n-1]\!] = e^{i\frac{k}{n}2\pi} = \cos\left(\frac{k}{n}2\pi\right) + i\sin s\left(\frac{k}{n}2\pi\right)$$

Hence we have

$$A\left(\omega_n^k\right) = A_0\left(\omega_n^{2k}\right) + \omega_n^k A_1\left(\omega_n^{2k}\right) = A_0\left(\omega_{\frac{n}{2}}^k\right) + \omega_n^k A_1\left(\omega_{\frac{n}{2}}^k\right)$$
$$A\left(-\omega_n^k\right) = A\left(\omega_n^{\frac{n}{2}+k}\right) = A_0\left(\omega_n^{2k}\right) - \omega_n^k A_1\left(\omega_n^{2k}\right) = A_0\left(\omega_{\frac{n}{2}}^k\right) - \omega_n^k A_1\left(\omega_{\frac{n}{2}}^k\right)$$

Hence now we will solve the following problem:

Recursive-DFT

Input: $(a_0, \ldots, a_{n-1})$ representing $(n-1)-$degree polynomial $A(x) = \sum_{i=0}^{n-1} a_i x^i$

Question: Find the evaluations of the polynomial $A(x)$ in all n^{th} roots of unity

Since A_0 and A_1 have degree $\frac{n}{2} - 1$ we can use recursion. Hence, the algorithm is

Algorithm 9: Recursive-DFT(A)

Input: $A = (a_0, \ldots, a_{n-1})$ such that $A(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$
Output: $A(x)$ evaluated at n^{th} roots of unity ω_n^k for all $k \in [\![n-1]\!]$

```
1  begin
2      if n == 1 then
3          return A[0]
4      A_0 ⟵ (A[0], A[2], ..., A[n-2])
5      A_1 ⟵ (A[1], A[3], ..., A[n-1])
6      Y^0 ⟵ Recursive-DFT(A_0)
7      Y^1 ⟵ Recursive-DFT(A_1)
8      for k = 0 to n/2 - 1 do
9          Y[k] ⟵ Y^0[k] + ω_n^k Y^1[k]                    // A(ω_n^k) = A_0(ω_{n/2}^k) + ω_n^k A_1(ω_{n/2}^k)
10         Y[k + n/2] ⟵ Y^0[k] - ω_n^{n/2+k} Y^1[k]        // A(-ω_n^k) = A_0(ω_{n/2}^k) - ω_n^k A^1(ω_{n/2}^k)
11     return Y
```

Time Complexity: $T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n\log n)$.

Therefore, we can evaluate an $n-1$ degree polynomial in all the n^{th} roots of unity in $O(n\log n)$ time. Hence, with this algorithm we will get evaluations of the polynomial $C(x) = A(x)B(x)$ in all the $2n^{th}$ roots of unity. Now we need to interpolate the polynomial $C(x)$ from its evaluations. We will describe the process in the next subsection.

3.2.3 Interpolation from Evaluations at Roots of Unity

In this section we will show how to interpolate an $n-1$ degree polynomial from evaluations at all n^{th} roots of unity. Previously we had

$$\underbrace{\begin{bmatrix} C\left(\omega_n^0\right) \\ C\left(\omega_n^1\right) \\ C\left(\omega_n^2\right) \\ \vdots \\ C\left(\omega_n^{n-1}\right) \end{bmatrix}}_{Y} = \underbrace{\begin{bmatrix} 1 & \omega_n^0 & \omega^{0\cdot 2} & \cdots & \omega^{0\cdot(n-1)} \\ 1 & \omega_n^1 & \omega^{1\cdot 2} & \cdots & \omega^{1\cdot(n-1)} \\ 1 & \omega_n^2 & \omega^{2\cdot 2} & \cdots & \omega^{2\cdot(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega^{(n-1)\cdot 2} & \cdots & \omega^{(n-1)\cdot(n-1)} \end{bmatrix}}_{V = \text{Vandermonde Matrix}} \underbrace{\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{bmatrix}}_{C}$$

Now Vandermonde matrix is invertible since all the n^{th} roots are distinct. Therefore, $C = V^{-1}Y$. But we can not do a matrix inversion to interpolate the polynomial because that will take $O(n^2)$ time. Instead, we have this beautiful result:

Lemma 3.2.2
$\left(V^{-1}\right)_{j,k} = \frac{1}{n}\omega_n^{-jk}$ for all $0 \le j,k \le n-1$

Proof: Consider the matrix $n \times n$ matrix T such that $(T)_{j,k} = \frac{1}{n}\omega_n^{-jk}$. Now we will show $VT = I$ This will confirm that $V^{-1} = T$. Now

$$\sum_{k=0}^{n-1}(V)_{i,j}\,(T)_{j,k} = \sum_{k=0}^{n-1}\omega_n^{ij} \times \frac{1}{n}\omega_n^{-jk} = \frac{1}{n}\sum_{k=0}^{n-1}\left(\omega_n^{i-k}\right)^j = \begin{cases} \frac{1}{n}\sum_{k=0}^{n-1} 1 = 1 & \text{when } i = k \\ \frac{1}{n}\frac{1-\omega_n^n}{1-\omega} = 0 & \text{when } i \neq k \end{cases}$$

Hence in VT there are $1'$s on the diagonal and rest of the locations are 0. Hence, $VT = I$. So $V^{-1} = T$. ■

Hence, we can see the inverse of the Vandermonde matrix is also a Vandermonde matrix with a scaling factor. We will denote $y_i = C\left(\omega_n^i\right)$ for $i \in [\![n-1]\!]$ since these values are given to us somehow, and we have to find the corresponding polynomial. Therefore, we have

$$\underbrace{\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{bmatrix}}_{C} = \frac{1}{n}\underbrace{\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^{-1} & \omega^{-1\cdot 2} & \cdots & \omega^{-1\cdot(n-1)} \\ 1 & \omega_n^{-2} & \omega^{-2\cdot 2} & \cdots & \omega^{-2\cdot(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega^{-(n-1)\cdot 2} & \cdots & \omega^{-(n-1)\cdot(n-1)} \end{bmatrix}}_{V^{-1}}\underbrace{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix}}_{Y}$$

Observation. $nc_j = y_0 + y_1\omega_n^{-j} + y_2\omega_n^{-2j} + \cdots + y_{n-1}\omega_n^{-(n-1)j}$ *for all* $j \in [\![n-1]\!]$.

We can also see this situation as we have the polynomial $Y(x) = y_0 + y_1x + y_2x^2 + \cdots + y_{n-1}x^{n-1}$ and c_j is just $Y(x)$ evaluated as $\omega_n^{-j} = \omega_n^{n-j}$ multiplied by n. Hence, we just re-index the n^{th} roots of unity and evaluate Y n^{th} roots of unity in $O(n\log n)$ time using the algorithm described in subsection 3.2.2

Chapter 4

Dynamic Programming

Definition 4.1: Dynamic Programming

Dynamic Programming has 3 components:

1. Optimal Substructure: Reduce problem to smaller independent problems
2. Recursion: Use recursion to solve the problems by solving smaller independent problems
3. Table Filling: Use a table to store the result to solved smaller independent problems.

4.1 Longest Increasing Subsequence

Longest Increasing Subsequence
Input: Sequence of distinct integers $A = (a_1, \ldots, a_n)$
Question: Given an array of distinct integers find the longest increasing subsequence i.e. return maximum size set $S \subseteq [n]$ such that $\forall\, i, j \in S, i < j \implies a_i < a_j$

4.1.1 $O(n^2)$ Time Algorithm

Given $A = (a_1, \ldots, a_n)$ first we will create a n-length array where i^{th} entry stores the length and longest increasing subsequence ending at a_i. Certainly we have the following recursion relation

$$\text{LIS}(k) = 1 + \max_{j<k,\ a_j<a_k} \{\text{LIS}(j)\}$$

since if a subsequence $S \subseteq [n]$ is the longest increasing subsequence ending at a_k then certainly $S - \{k\}$ is the longest increasing subsequence which ends at $a_j < a_k$ for some $j < k$. Hence, in the table we start with 1st position and using the recursion relation we fill the table from left. And after the table is filled we look for which entry of the table has maximum length. So the algorithm will be following:

Algorithm 10: LIS(A)

Input: Sequence of distinct integers $A = (a_1, \ldots, a_n)$
Output: Maximum size set $S \subseteq [n]$ such that $\forall\, i, j \in S, i < j \implies a_i < a_j$.

```
1 begin
2     Create an array T of length n
3     for i ∈ [n] do
4         T[i][1] ⟵ 1 + max{T[j][1] : j < k, a_j < a_k}        // Finds LIS[i]
5         T[i][2] ⟵ T[T[i][1] − 1][2]
6     Index ⟵ max{T[j][1] : j ∈ [n]}
7     return T[Index]
```

Time Complexity: For each iteration of the loop it takes $O(n)$ time to find LIS$[i]$. Hence, the time complexity of this algorithm is $O(n^2)$.

4.1.2 $O(n \log n)$ Time Algorithm

In the following algorithm we update the longest increasing sequence every time we see a new element of the given sequence. At any time we keep the best available sequence.

Idea. *We can make an increasing subsequence longer by picking the smallest number for position k so that there is an increasing subsequence of length k. Doing this we can maximize the length of the subsequence.*

Theorem 4.1.1

If $S \subseteq A$ is the longest increasing subsequence of length t then for any $k \in [t]$ the number $S(k)$ is the smallest number in the subarray of A starting at first and ending at $S(k)$ such that there is an increasing subsequence of length k ending at $S(k)$.

Proof: Assume the contrary. Suppose $\exists\ k \in [t]$ such that k is the smallest number in $[t]$ such that $S(k)$ is not the smallest number to satisfy the condition. Now denote the subarray of A starting at first and ending at $S(k)$ by A_k. Now let $x \in A_k$ be the smallest number in A_k such that there is an increasing subsequence of length k ending at x. Certainly $x < S(k)$ by our assumption. Now since k is the smallest index which does not satisfy the given condition, $\forall\ j \in [k-1]$, $S(j)$ is the smallest number in A_j such that there is an increasing subsequence of length j ending at $S(j)$. Then consider the subsequence $\{S(1), \ldots, S(k-1), x, S(k), S(k+1), \ldots, S(t)\}$. This is an increasing subsequence of A and has length $t+1$. But this contradicts the minimality of S. Hence, contradiction ↯ Every element of S follows the given condition. ■

So we will construct an increasing subsequence by gradually where each step this property is followed, i.e. at each step we will ensure that the sequence built at some time have the above property. So now we describe the algorithm.

Algorithm 11: QUICKLIS(A)

Input: Sequence of distinct integers $A = (a_1, \ldots, a_n)$
Output: Maximum size set $S \subseteq [n]$ such that $\forall\ i, j \in S, i < j \implies a_i < a_j$.

```
1  begin
2      Create an array T of length n with all entries 0
3      Create an array M of length n
4      for i = 1, ..., n do
5          M[i] ⟵ ∞
6      for i = 1, ..., n do
7          k ⟵Find the smallest index such that M[k] ≥ a_i using BINARY-SEARCH
8          M[k] ⟵ a_i
9          T[i] ⟵ M[k − 1]        // Pointer to the previous element of the sequence
10     l ⟵ Largest l such that M[l] is finite
11     Create an array S of length l
12     for i = l, ..., 1 do
13         if i = l then
14             S[l] ⟵ M[l]
15             Continue
16         S[i] ⟵ T[S[i + 1]]     // T[S[i+1]] is pointer to previous value of sequence
17     return (l, S)
```

Time Complexity: To create the arrays and the first for loop takes $O(n)$ time. In each iteration of the for loop at line 6 it takes $O(\log n)$ time to find k and rest of the operations in the loop takes constant time. So the for loop takes $O(n \log n)$ time. Then To find l and creating S it takes $O(n)$ time. Then in the for loop at line 12 in each iteration it takes constant time. So the for loop at line 12 takes in total $O(n)$ time. Therefore, the algorithm takes $O(n \log n)$ time.

We will do the proof of correctness of the algorithm now.

Lemma 4.1.2

For any index $M[k]$ is non-increasing

Proof: Every time we change a value of $M[k]$ we replace by something smaller. So $M[k]$ is non-increasing. ■

We denote the state of array M at i^{th} iteration by M^i. Then we have the following lemma:

Lemma 4.1.3

At any time i, $M^i[1] < M^i[2] < \cdots < M^i[n]$

Proof: We will prove this by induction on i. The base case follows naturally. Now for i^{th} iteration suppose $M^{i-1}[k]$ is replaced by x_i. Then we know $\forall\ j < k$ we have $M^i[j] < x_i$. By inductive hypothesis at time $t-1$ we have M as an increasing sequence. Now before replacing $M^{i-1}[k] < M^{i-1}[k+1] < \cdots M^{i-1}[n]$. Now by Lemma 4.1.2 $M^{i-1}[k]$ is nonincreasing. So we still have $M^{i-1}[1] < \cdots M^{i-1}[k-1] < x_i < M^{i-1}[k+1] < \cdots < M^{i-1}[n]$. Therefore, M^i is an increasing subsequence. Hence, but mathematical induction it holds. ■

Now suppose at i^{th} iteration k_i is largest such that $M^i[k_i] < \infty$. Then S^i denote the set constructed like the way we constructed at line 12–16 in the algorithm i.e.

$$S^i[k_i] = M^i[k_i] \qquad \text{and} \qquad S^i[j] = T[S^i[j+1]] \quad \forall\ j \in [k_i - 1]$$

Lemma 4.1.4

After any i^{th} iteration, for $k \in [n]$ if $M^i[k] < \infty$ then $S^i[k]$ stores the smallest value in $x_1, \ldots, x_i$ such that there is an increasing subsequence of size k that ends in $S^i[k]$.

Proof: We will use induction on i. Base case: This is true after first iteration since only $M^1[1] < \infty$. So this naturally follows.

Suppose this is true after i iterations. Now at $(i+1)^{th}$ iteration suppose t be the smallest index such that $M^i[t] > x_{i+1}$. Then we have

$$M^i[1] < \cdots < M^i[t-1] < x_{i+1} \leq M^i[t] < \cdots < M^i[n] \implies S^i[1] < \cdots < S^i[t-1] < x_{i+1} \leq S^i[t], \ldots, S^i[k_i]$$

Now for $k \leq t-1$ it is true by the inductive hypothesis. For $k > t$ and if $M^{i+1}[k] < \infty$ then $S^{i+1}[k]$ is the smallest value in $x_1, \ldots, x_{i+1}$ such that there is an increasing subsequence of size k that ends in $S^{i+1}[k]$ since this was true for i^{th} iteration.

Now only the case when $k = t$ is remaining. If $S^{i+1}[k]$ does not store the smallest value in $x_1, \ldots, x_{i+1}$ to have an increasing subsequence of size k ending at $S^{i+1}[k]$ then let x_j was the smallest value to satisfy this condition where $j < i+1$. Then naturally $x_j < x_{i+1}$. Then $M^i[t] \leq x_j < x_{i+1}$. But we t was the smallest number such that $M^i[t] \geq x_{i+1}$. Hence, contradiction. Therefore, $S^i[k]$ is the smallest value in $x_1, \ldots, x_{i+1}$ to have an increasing subsequence of size k ending at $S^{i+1}[k]$. Therefore, by mathematical induction this is true for all iterations. ■

Theorem 4.1.5

S is the longest increasing subsequence of A.

Proof: After the n^{th} iteration $S^n = S$ and $k_n = l$. Hence by Lemma 4.1.4 we can say for all $k \in [l]$, $S[k]$ is the smallest number such that there is an increasing sequence of length k ending at $S[k]$. Now we want to show that this increasing sequence is the longest increasing subsequence of A. Suppose S is not the longest increasing subsequence. Let T be the longest increasing subsequence of length $t > l$. Then suppose $j \leq l$ be the smallest index such that $S[j] \neq T[j]$. Now $S[j]$ is the smallest number in $x_1, \ldots, x_n$ such that there is an increasing subsequence of length j ending at $S[j]$. Hence, we have $S[j] < T[j]$. Now for all $i < j$ we have $S[i] = T[i]$. Then we form this new subsequence $\hat{T} = \{T[1], T[2], \ldots, T[j-1], S[j], T[j], \ldots, T[t]\}$. Certainly $\hat{T}$ has length $t+1$ and it is also an increasing subsequence. But this contradicts the maximal condition of T. Hence, S is indeed the longest increasing subsequence. ■

4.2 Optimal Binary Search Tree

OPTIMAL BST

Input: A sorted array $A = (a_1, \ldots, a_n)$ of search keys and an array of their probability distributions $P = (p(a_1), \ldots, p(a_n))$

Question: Given array of keys A and their probabilities the probability of accessing a_i is $p(a_i)$ then return a binary tree with the minimum cost where for any binary tree T, $\text{COST}(T) = \sum_{i=1}^{n} p(a_i) \cdot height_T(a_i)$.

So let T be the optimal binary search tree with a_k as its root for some $k \in [n]$. Let T_l and T_r denote the tree rooted at the left child and right child of a_k in T respectively. Then:

$$\text{COST}(T) = p_k + \sum_{i<k} p_i \left(1 + height_{T_l}(a_i)\right) + \sum_{i>k} p_i \left(1 + height_{T_r}(a_i)\right) = \sum_{i=1}^{n} p_i + \underbrace{\sum_{i<k} p_i \cdot height_{T_l}(a_i)}_{\text{COST}(T_l)} + \underbrace{\sum_{i>k} p_i \cdot height_{T_l}(a_i)}_{\text{COST}(T_r)}$$

In general we will use the notation $\text{OPTCOST}(i,k) = \text{COST}(T_i^k)$ where T_i^k is the optimal binary tree of the subarray $A[i \ldots k]$ for any $i \le k \le n$. Therefore, we arrive at the following recurrence relation

$$\text{OPTCOST}(i,k) = \begin{cases} 0 & \text{when } i > k \\ \sum_{j=i}^{k} p(a_j) + \min_{i \le r \le k} \{\text{OPTCOST}(i, r-1) + \text{OPTCOST}(r+1, k)\} & \text{otherwise} \end{cases}$$

So the algorithm for constructing the optimal binary search tree is following:

Algorithm 12: OPTIMALBST(A, P)

Input: A sorted array $A = (a_1, \ldots, a_n)$ of search keys and an array of their probability distributions $P = (p(a_1), \ldots, p(a_n))$

Output: Binary Tree T with the minimum search cost, $\text{COST}(T) = \sum_{i=1}^{n} p(a_i) \cdot height_T(a_i)$

```
1  begin
2      for i = 1, ..., n do
3          OPTCOST[i, i] ⟵ (p(a_i), a_i), OPTCOST[0, i] ⟵ (0, None)
4      for d = 2, ..., n do
5          for i ∈ [n + 1 − d] do
6              minval ⟵ 0
7              for k = i + 1, ..., i + d − 2 do
8                  newval ⟵ OPTCOST[i, k − 1][1] + OPTCOST[k + 1, i + d − 1][1]
9                  if minval > newval then
10                     minval ⟵ newval
11                     Index ⟵ k
12             OPTCOST[i, i + d − 1] ⟵ (minval + Σ_{k=1}^{i+d−1} p(a_k), k)
13             a_k.left ⟵ OPTCOST[i, k − 1][2]
14             a_k.right ⟵ OPTCOST[k + 1, i + d − 1][2]
15     return OPTCOST[1, n]
```

Time Complexity: To two for loops at line 4 and line 5 takes $O(n^2)$ many iterations. Now the innermost for loop at line 7 runs $O(n)$ iterations where in each iteration it takes constant runtime. So the total running time of the algorithm is $O(n^3)$.

Chapter 5

Greedy Algorithm

5.1 Maximal Matching

Maximal Matching
Input: Graph $G = (V, E)$
Question: Find a maximal matching $M \subseteq E$ of G

Before diving into the algorithm to find a matching or maximal matching we first define what is a matching.

Definition 5.1.1: Matching

Given a graph $G = (V, E)$, $M \subseteq E$ is said to be a matching if M is an independent set of edges i.e. no two edges of M are incident on same vertex.

Definition 5.1.2: Maximal Matching

For a graph $G = (V, E)$ a matching $M \subseteq E$ is maximal if it cannot be extended and still by adding an edge.

There is also a maximum matching which can be easily understood from the name:

Definition 5.1.3: Maximum Matching

For a graph $G = (V, E)$ a matching $M \subseteq E$ is maximum if it is maximal and has the maximum size among all the maximal matchings.

Idea. *The idea is to create a maximal matching we will just go over each edge one by one and check if after adding them to the set M the matching property still holds.*

Algorithm 13: Maximal-Matching

Input: Graph $G = (V, E)$
Output: Maximal Matching $M \subseteq E$ of G

```
1 begin
2     M ⟵ ∅
3     Order the edges E = {e_1, ..., e_k} arbitrarily
4     for e ∈ E do
5         if M ∪ {u} is matching then
6             M ⟵ M ∪ {e}
7     return M
```

Question 5.1

Do we always get the largest possible matching?

Solution: Clearly algorithm output is not optimal always. We get a maximal matching sure. But we don't get a maximum matching always. For example the following graph

If we start from e_1 we get the matching $\{e_1.e_2\}$ which is maximum matching but if we start from e_3 then we get only the maximal matching $\{e_3\}$ which is not maximum. ■

Since the algorithm output may not be optimal always we can ask the following question

Question 5.2

How large is the matching obtained compared to the maximum matching?

This brings us to the following result:

Theorem 5.1.1

For any graph G let the greedy algorithm obtains the matching M and the maximum matching is $M^\star$. Then

$$|M| \geq \frac{1}{2}|M^\star|$$

Proof: Consider an edge $e \in M^\star$ but $e \notin M$. Since e wasn't picked in M, $\exists\, e' \in M \setminus M^\star$ such that e and e' are incident on same vertex. Thus define the function $f : M^\star \to M$ where

$$f(e) = \begin{cases} e & \text{when } e \in M \\ e' & \text{when } e \in M^\star \setminus M \text{ where } e' \in M \setminus M^\star \text{ such that } e' \cap e \neq \emptyset \end{cases}$$

Now note that there are at most two edges in $M^\star$ that are adjacent to an edge $e' \in M$ which will be mapped to e'. Hence,

$$|M \setminus M^\star| \geq \frac{1}{2}|M^\star \setminus M|$$

Therefore $|f^{-1}(e')| \leq 2 \;\forall\; e' \in M$. Hence,

$$|M^\star| = |M \cap M^\star| + |M^\star \setminus M| \leq |M \cap M^\star| + 2|M \setminus M^\star| \leq 2|M|$$

Therefore we have the result $|M| \geq \frac{1}{2}|M^\star|$. ■

Alternate Proof : Let M_1 and M_2 are two matchings. Consider the symmetric difference $M_1 \triangle M_2$. This consists of edges that are in exactly one of M_1 and M_2. Now in $M \triangle M^\star$ we have the following properties:

(a) Every vertex in $M \triangle M^\star$ has degree $\leq 2 \implies$ Each component is a path or an even cycle.

(b) The edges of M and $M^\star$ alternate.

Now we will prove the following property about the connected components of $M \triangle M^\star$.

Claim 5.1.1

No connected component is a single edge.

Proof: This is because let e be a connected component. So the two edges e_1, e_2 which are adjacent to e, they are either in both M and $M^\star$ or not in M and $M^\star$. The former case is not possible because then e_1, e_2, e are all in either M or $M^\star$ which is not possible as they do not satisfy the condition of matching. For the later case let $e \in M^\star$. Then $e \notin M$. That means $e, e_1, e_2 \notin M$ which is not possible since M is also a maximum matching. By similar reasoning if $e \in M$ and $e \notin M^\star$ then also an impossible event occurs. Therefore, no connected component is a single edge. ■

Therefore, every path has length ≥ 2. Therefore, ratio of # edges of M to # edges of $M^\star$ in a path is ≤ 2. And for cycles we have # edges of M = # edges of $M^\star$. So in every connected component C of $M \triangle M^\star$ the ratio $\frac{|M^\star \cap C|}{|M \cap C|} \leq 2$. Therefore, we have

$$\frac{|M^\star|}{|M|} = \frac{|M \cap M^\star| + \sum_C |M^\star \cap C|}{|M \cap M^\star| + \sum_C |M \cap C|} \leq 2$$

Hence we have $|M| \geq \frac{1}{2}|M^\star|$.

■

5.2 Huffman Encoding

Huffman Coding
Input: n symbols $A = (a-1, \ldots, a_n)$ and their frequencies $P = (f_1, \ldots, f_n)$ of using symbols
Question: Create a binary encoding such that:

- Prefix Free: The code for one word can not be prefix for another code
- Minimality: Minimize $\textsc{Cost}(b) = \sum_{i=1}^{n} f_i \cdot \textsc{Len}(b(a_i))$ where $b : A \to \{0,1\}^*$ is the binary encoding

Assignment of binary strings can also be scene as placing the symbols in a binary tree where at any node 0 means left child and 1 means right child. Then the first condition implies that there can not be two codes which lies in the same path from the root to a leaf. I.e. it means that all the codes have to be in the leaves. Then the length of the binary coding for a symbol is the height of the symbol in the binary tree.

We can think the frequencies as the probability of appearing for a letter. We denote the probability of appearing of the letter a_i by $p(a_i) \coloneqq \frac{f_i}{\sum_{i=1}^{n} f_i}$. So the we can see the updated cost function

$$\textsc{Cost}(b) = \sum_{i=1}^{n} p(a_i) \cdot \textsc{Len}(b(a_i))$$

And from now on we will see the frequencies as probabilities and cost function like this

5.2.1 Optimal Binary Encoding Tree Properties

Then our goal is to finding a binary tree with minimum cost where all the symbols are at the leaves. We have the following which establish the optimality of Huffman encoding over all prefix encodings where each symbol is assigned a unique string of bits.

Lemma 5.2.1
In the optimal encoding tree least frequent element has maximum height.

Proof: Suppose that is not the case. Let T be the optimal encoding tree and let the least frequent element x is at height h_1 and the element with the maximum height is y with height h_2 and we have $h_1 < h_2$. Then we construct a new encoding tree T' where we swap the positions of x and y. So in T' height of y is h_1 and height of x is h_2. Then

$$\textsc{Cost}(T) - \textsc{Cost}(T') = (p(x)h_1 + p(y)h_2) - (p(x)h_2 + p(y)h_1) = (p(x) - p(y))(h_1 - h_2)$$

Since $p(x) < p(y)$ and $h_1 < h_2$ we have $\textsc{Cost}(T) - \textsc{Cost}(T') > 0$. But that is not possible since T is the optimal encoding tree. So T should have the minimum cost. Hence contradiction. x has the maximum height. ■

Lemma 5.2.2

The optimal encoding binary tree must be complete binary tree. (i.e. every non-leaf node has exactly 2 children)

Proof: Suppose T be the optimal binary tree and there is a non-leaf node r which has only one child at height h. By Lemma 5.2.1 the least frequent element x has the maximum height, h_m.

Then consider the new tree $\hat{T}$ where we place the least frequent element at height h and make it the second child of the node r. Then

$$\text{Cost}(T) - \text{Cost}(\hat{T}) = p(x)h_m - p(x)h = p(x)(h_m - h) > 0$$

But this is not possible as T is the optimal binary tree and it has the minimal cost. Hence contradiction. Therefore the optimal encoding binary tree must be a complete binary tree. ■

Lemma 5.2.3

There is an optimal binary encoding tree such that the least frequent element and the second least frequent element are siblings at the maximum height.

Proof: Let T be optimal binary encoding tree. Suppose x, y are the least frequent element and the second least frequent element. And suppose b, c be two siblings at the maximum height of the tree (There may be many such siblings, and if so pick any such pair.). If $\{x, y\} = \{b, c\}$ we are done. So suppose not. Let the frequencies of x, y, b, c are respectively $p(x), p(y), p(b), p(c)$ and heights of x, y, b are h_x, h_y and h respectively. WLOG assume $p(x) \leq p(y)$ and $p(b) \leq p(c)$.

Now since we know x, y have the smallest frequencies we have $p(x) \leq p(b)$ and $p(y) \leq p(c)$. And since b, c have the maximum height we have $h_x, h_y \leq h$. So we switch the position of x with b to form the new tree T'. And from T' we swap the positions fo y and c to form a new tree T''.

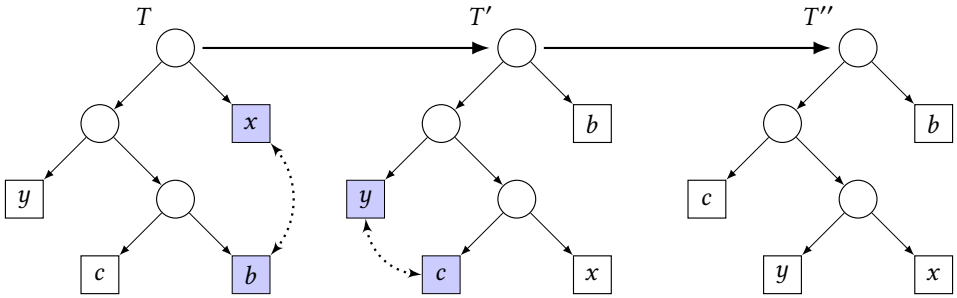


Figure 5.1: Showing that the lowest probability nodes are siblings at the tree's lowest level.

Now we will calculate how the cost changes as we go from T to T' and T' to T''. First check for $T \to T'$. Almost all the nodes contribute the same except x, b. So we have

$$\text{Cost}(T) - \text{Cost}(T') = (h_x \cdot p(x) + h \cdot p(b)) - (h_x \cdot p(b) + h \cdot p(x)) = (p(b) - p(x))(h - h_x) \geq 0$$

Therefore swapping x and b does not increase the cost and since T is the optimal binary encoding tree the cost doesn't decrease either. Therefore the costs are equal. Hence T' is also an optimal tree.

Similarly we calculate cost for going from T' to T'' we have

$$\text{Cost}(T') - \text{Cost}(T'') = (h_y \cdot p(y) + h \cdot p(c)) - (h_y \cdot p(c) + h \cdot p(y)) = (p(c) - p(y))(h - h_y) \geq 0$$

Therefore swapping y and c also does not increase the cost and since T' is the optimal binary encoding tree the cost doesn't decrease either. Therefore the costs are equal. Hence T'' is also an optimal tree. Hence T'' is the optimal tree where the least frequent element and second last frequent element are siblings. ■

By the Lemma 5.2.2 and Lemma 5.2.3 we have that the least frequent element and the second least frequent element are siblings, and they have the maximum height.

Theorem 5.2.4

Given an instance with symbols $\mathcal{I}$:

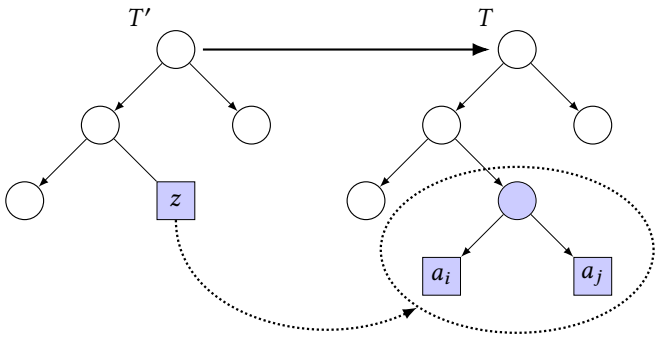
$$\begin{array}{cccccccc} a_1, & a_2, & \cdots, & a_i, & \cdots, & a_j, & \cdots, & a_n \\ p(a_1), & p(a_2), & \cdots, & p(a_i), & \cdots, & p(a_j), & \cdots, & p(a_n) \end{array} \quad \text{with probabilities}$$

such that a_i, a_j are the least frequent and second least frequent elements respectively. Consider the instance with $n-1$ symbols $\mathcal{I}'$:

$$\begin{array}{ccccccccccc} a_1, & a_2, & \cdots, & a_{i-1}, & a_{i+1}, & \cdots, & a_{j-1}, & a_{j+1}, & \cdots, & a_n, & z \\ p(a_1), & p(a_2), & \cdots, & p(a_{i-1}), & p(a_{i+1}) & \cdots, & p(a_{j-1}), & p(a_{j+1}), & \cdots, & p(a_n), & p(a_i)+p(a_j) \end{array}$$

Let T' be the optimal tree for this instance $\mathcal{I}'$. Then there is an optimal tree for the original instance $\mathcal{I}$ obtained from T' by replacing the leaf of b by an internal node with children a_i and a_j.

Proof: We will prove this by contradiction. Suppose $\hat{T}$ is optimal for $\mathcal{I}$. Then $\text{Cost}(\hat{T}) < \text{Cost}(T)$. In $\hat{T}$ we know a_i and a_j are siblings by Lemma 5.2.3. Now consider $\hat{T}'$ for instance $\mathcal{I}'$ where we merge a_i, a_j leaves and their parent into a leaf for symbol z.



Then

$$\text{Cost}(\hat{T}') = \text{Cost}(\hat{T}) - p(a_i) - p(a_j) < \text{Cost}(T) - p(a_i) - p(a_j) = \text{Cost}(T')$$

This contradicts the fact that T' is optimal binary encoding tree for $\mathcal{I}'$. Hence T is optimal. ■

5.2.2 Algorithm

Idea: We are going to build the tree up from the leaf level. We will take two characters x, y, and "merge" them into a single character, z, which then replaces x and y in the alphabet. The character z will have probability equal to the sum of x and y's probabilities. Then we continue recursively building the code on the new alphabet, which has one fewer character.

Since we always need the least frequent element and the second least frequent element we have to use the data structure called MIN-PRIORITY QUEUE. So the following algorithm uses a MIN-PRIORITY QUEUE Q keyed on the probabilities to identify the two least frequent objects.

Time Complexity: To create the priority queue it takes $O(n)$ time in line 4-5. Then for each iteration of the for loop in line 6 the EXTRACT-MIN operation takes $O(\log n)$ time and then to insert an element it also takes $O(\log n)$ time. Hence each iteration takes $O(\log n)$ time. Since the for loop has $n-1 = O(n)$ many iterations the running time for the algorithm is $O(n \log n)$.

Remark: We can reduce the running time to $O(n \log\log n)$ by replacing the binary min-heap with a van Emde Boas tree.

Algorithm 14: HUFFMAN-ENCODING(A, P)

Input: Set of n symbols $A = \{a_1, \ldots, a_n\}$ and their probabilities $P = \{p_1, \ldots, p_n\}$

Output: Optimal Binary Encoding $b : A \to \{0,1\}^*$ for A with minimum $\text{COST}(b) = \sum_{i=1}^{n} p(a_i) \cdot \text{LEN}(b(a_i))$.

```
1  begin
2      n ⟵ |A|
3      Q ⟵ MIN-PRIORITY QUEUE
4      for x ∈ A do
5          INSERT(Q, x)
6      for i = 1, ... n − 1 do
7          z ⟵ New internal tree node
8          x ⟵ EXTRACT-MIN(Q), y ⟵ EXTRACT-MIN(Q)
9          left[z] ⟵ x, right[z] ⟵ y
10         p(z) ⟵ p(x) + p(y)
11         INSERT(Q, z)
12     return Last element left in Q as root
```

Theorem 5.2.5 Correctness of Huffman's Algorithm

The above Huffman's algorithm produces an optimal prefix code tree

Proof: We will prove this by induction on n, the number of symbols. For base case $n = 1$. There is only one tree possible. For $n = k$ we know that by Lemma 5.2.3 and Lemma 5.2.1 that the two symbols x and y of lowest probabilities are siblings and they have the maximum height. Huffman's algorithm replaces these nodes by a character z whose probability is the sum of their probabilities. Now we have 1 less symbols. So by inductive hypothesis Huffman's algorithm computes the optimal binary encoding tree for the $k-1$ symbols. Call it T_{n-1}. Then the algorithm replaces z with a parent node with children x and y which results in a tree T_n whose cost is higher by a fixed amount $p(z) = p(x) + p(y)$. Now since T_{n-1} is optimal by Theorem 5.2.4 we have T_n is also optimal. ■

5.3 Matroids

Definition 5.3.1: Matroid

A matroid $M = (E, \mathcal{I})$ has a ground set E and a collection I of subsets of E called the *Independent Sets* st

1. Downward Closure: If $Y \in \mathcal{I}$ then $\forall\, X \subseteq Y, X \in \mathcal{I}$.
2. Exchange Property: If $X, Y \in \mathcal{I}$, $|X| < |Y|$ then $\exists\, e \in Y - X$ such that $X \cup \{e\}$ also written as $X + e \in \mathcal{I}$

An element $x \in E$ extends $A \in \mathcal{I}$ if $A \cup \{x\} \in \mathcal{I}$. And A is maximal if no element can extend A.

Lemma 5.3.1

If A, B are maximal independent set, then $|A| = |B|$ i.e. all maximal independent sets are also maximum

Proof: Suppose $|A| \neq |B|$. WLOG assume $|A| > |B|$. Then by the exchange property $\exists\, e \in A - B$ such that $B \cup \{e\} \in \mathcal{I}$. But we assumed that B is maximal independent set. Hence contradiction. We have $|A| = |B|$. ■

Base: Maximal Independent sets are called bases.
Rank of $S \in I$: $\max\{|X| : X \subseteq S, X \in I\}$
Rank of a Matroid: Size of the base.
Span of $S \in I$: $\{e \in E : rank(S) = rank(S + e)\}$

5.3.1 Examples of Matroid

- **Uniform Matroid:** Given $E = \{e_1, \ldots, e_n\}$, and $k \in \mathbb{Z}_0$ take $\mathcal{I} = \{S \subseteq E : |S| \leq k\}$

> **Lemma 5.3.2**
> $M = (E, \mathcal{I})$ defined as above is a matroid

Proof:

① Downward Closure: $A \in \mathcal{I}, B \subseteq A \implies |B| \leq k \implies B \in \mathcal{I}$

② Exchange Property: $A, B \in \mathcal{I},\ |B| < |A| \leq k \implies |B| < k \implies \forall\, e \in A - B,\ |B \cup \{e\}| \leq k \implies B \cup \{e\} \in \mathcal{I}$

Therefore M is a matroid ■

- **Partition Matroid:** Given E, $\{P_1, \ldots, P_l\}$ such that $E = \bigsqcup_{i=1}^{l} P_i$ and $k_1, \ldots, k_l \in \mathbb{Z}_0$ then take

$$\mathcal{I} = \{S \subseteq E : \forall\, k \in [l],\ |S \cap P_j| \leq k_j\}$$

> **Lemma 5.3.3**
> $M = (E, \mathcal{I})$ defined as above is a matroid

Proof:

① Downward Closure: $A \in \mathcal{I}, B \subseteq A \implies \forall\, j \in [l]\ |B \cap P_j| \leq |A \cap P_j| \leq k_j \implies B \in \mathcal{I}$

② Exchange Property: $A, B \in \mathcal{I},\ |B| < |A| \implies \exists\, j \in [l],\ |B \cap P_j| < |A \cap P_j| \leq k_j \implies e \in (A \cap P_j) - (B \cap P_j),\ |(B \cup \{e\}) \cap P_j| = |B \cap P_j| + 1 \leq k \implies B \cup \{e\} \in \mathcal{I}$

Therefore M is a matroid ■

- **Laminar Matroid:** Given E, $\mathscr{L} = \{L_1, \ldots, L_l\}$ is a laminar family i.e. $\forall\, i, j \in [l]$, either $L_i \subseteq L_j$ or $L_i \supseteq L_j$ or $L_i \cap L_j = \emptyset$ and also given $k_1, \ldots, k_l \in \mathbb{Z}_0$. Then take

$$\mathcal{I} = \{S \subseteq E : \forall\, j \in [l],\ |S \cap L_j| \leq k_j\}$$

For any $L \in \mathscr{L}$ we denote $k(L)$ be the given number corresponding to L.

> **Lemma 5.3.4**
> $M = (E, \mathcal{I})$ defined as above is a matroid

Proof:

① Downward Closure: $A \in \mathcal{I}, B \subseteq A \implies \forall\, j \in [l]\ |B \cap L_j| \leq |A \cap L_j| \leq k_j \implies B \in \mathcal{I}$

② Exchange Property: Let $A, B \in \mathcal{I}$ with $|B| < |A|$. If there exists $e \in A \backslash B$ such that $e \notin L$ for any $L \in \mathscr{L}$, then $|(B + e) \cap L| = |B \cap L| \leq k(L)$ for any $L \in \mathscr{L}$.

Hence assume that for each $e \in A \backslash B$ there exists $L \in \mathscr{L}$ with $e \in L$. For each $e \in A \backslash B$, let $\mathscr{L}_e$ be the collection of $L \in \mathscr{L}$ with $e \in L$. For each $e \in A \backslash B$ and any $L \in \mathscr{L} \backslash \mathscr{L}_e$, we have $|(B + e) \cap L| = |B \cap L| \leq k(L)$.

Hence it remains to show that there exists $e \in A \backslash B$ such that $|(B + e) \cap L| \leq k(L)$ for any $L \in \mathscr{L}_e$. Note that $\mathscr{L}_e$ is a chain, as $\mathscr{L}$ is a laminar. Let $\mathscr{L}' = \{L_{e_1}, \ldots, L_{e_l}\}$ be the collection of inclusion-wise maximal sets in $\mathscr{L}$ such that $|B \cap L_{e_i}| \leq k(L_{e_i})$ with $e_i \in A \backslash B$. Then $L_{e_i} \cap L_{e_j} = \emptyset$. Moreover, $|A| > |B|$ and $|A \cap L_{e_i}| \leq k(L_{e_i})$ imply that $|A \backslash (\cup L_{e_i})| > |B \backslash (\cup L_{e_i})|$. Hence there $\exists\, e_i$ such that $|A \cap L_{e_i}| > |B \cap L_{e_i}|$.

Now we take a look at the chain $\mathscr{L}_{e_i}$. For brevity we will use e instead of e_i. So in the chain $\mathscr{L}_e = \{L_1, \ldots, L_n\}$ such that we have

$$L_n \supseteq L_{n-1} \supseteq \cdots \supseteq L_2 \supseteq L_1$$

Then take $i \in [n]$ to be the largest index such that $|A \cap L_i| \leq |B \cap L_i|$. There will be such index because otherwise we will have $|A| \leq |B|$ which is not possible. Then take $e^* \in (A \cap L_{i+1}) - (L_i \cup B)$. Such an e^* will exist because $|A \cap L_{i+1}| > |A \cap L_{i+1}| \implies A \cap (L_{i+1} - L_i \neq \emptyset$ and also $A \cap (L_{i+1} - L_i \not\subseteq B \cap (L_{i+1} - L_i)$ because otherwise we will have

$$|A \cap L_{i+1}| = |A \cap (L_{i+1} - L_i| + |A \cap L_{i+1}| \leq |B \cap (L_{i+1} - L_i)| + |B \cap L_i| = |B \cap L_{i+1}|$$

which is not possible. Hence there exists e^* such that $e^* \in (A \cap L_{i+1}) - (L_i \cup B)$. Therefore take $B^* = B \cup \{e^*\}$. Then for all $j < i$ we have $B^* \cap L_j = B \cap L_j$ so we don't have a problem there. Now for all $j \geq i$ we have $|A \cap L_j| > |B \cap L_j|$. Hence now $|B^* \cap L_j| \leq |B \cap L_j| + 1 \leq |A \cap L_j| \leq k(L_j)$. Therefore we have $B^* \in \mathcal{I}$. Hence the exchange property follows.

Therefore M is a matroid. ■

- **Graphic Matroid:** Given a graph $G = (V, E)$ E is the ground set and take

$$\mathcal{I} = \{E' \subseteq E : E' \text{ is acyclic}\}$$

> **Lemma 5.3.5**
> $M = (E, \mathcal{I})$ defined as above is a matroid

Proof:

① Downward Closure: If a set of edges S is acyclic then naturally any subset of edges of S is also acyclic. Hence downward closure property follows.

② Exchange Property: $A, B \in \mathcal{I}$, and $|B| < |A|$. Let $G_1, \ldots, G_k$ are the connected components due to B. For each component G_i, we have $|G_i \cap A| \leq |G_i \cap B|$ since each component is a tree and B has maximum number of edges for that component. Then A contains an edge e connecting 2 components G_i and G_j. Then $B \cup \{e\} \in \mathcal{I}$.

Therefore M is a matroid ■

- **Linear Matroid:** Given a $m \times n$ matrix $M \in \mathbb{Z}^{m \times n}$, $E = [n]$ and take

$$\mathcal{I} = \{S \subseteq E : \text{Columns of } M \text{ corresponding to } S \text{ are linearly independent}\}$$

> **Lemma 5.3.6**
> $M = (E, \mathcal{I})$ defined as above is a matroid

Proof:

① Downward Closure: $A \in \mathcal{I}$, $B \subseteq A$. Subset of linearly independent set is also linearly independent. Hence $B \in \mathcal{I}$.

② Exchange Property: $A, B \in \mathcal{I}$, $|B| < |A|$. Then take span $\langle A \rangle$ over $\mathbb{Q}$. Now we know a set of integral vectors are linearly independent over integers if and only if they are linearly independent over rationals. Hence $|A| = \dim_{\mathbb{Q}} \langle A \rangle > \dim_{\mathbb{Q}} \langle B \rangle = |B|$. Hence we can extend B by an element $e \in A - B$ such that $\langle B \cup \{e\} \rangle = |B| + 1$. Hence $B \cup \{e\} \in \mathcal{I}$.

Therefore M is a matroid ■

This matroid is also called Metric Matroid.

5.3.2 Finding Max Weight Base

MAX WEIGHT BASE
Input: A matroid $M = (E, I)$ is given as an input as an oracle and a weight function $W : E \to \mathbb{R}$.
Question: Find the maximum weight base of the matroid.

We will solve this using greedy algorithm.

Algorithm 15: MAX-WEIGHT-BASE(E, W)

Input: A matroid $M = (E, I)$ is given as an input as an oracle and a weight function $W : E \to \mathbb{R}$.
Output: Find the maximum weight base of the matroid

```
1 begin
2     Assume w(1) ≥ ··· ≥ w(n)
3     S ← ∅
4     I ← {S}
5     for i = 1 to n do
6         if S + i ∈ I then
7             S ← S + i
8     return S
```

Theorem 5.3.7
The above algorithm outputs a maximum weight base

Proof: Let M be a matroid. We will prove that this greedy algorithm works by inducting on i. At any iteration i we need to prove the following claim:

Claim 5.3.1
At any iteration i there is a max weight base B_i such that $S_i \subseteq B_i$ and $B_i \setminus S_i \subseteq \{i+1, \ldots, n\}$.

Proof: Base case: $S = \emptyset$. So for base case the statement is true trivially. Assume that the statement is true up to $(i-1)$ iterations.

Now $S_{i-1} \subseteq B_{i-1}$ where B_{i-1} is a maximum weight base and $B_{i-1} - S_{i-1} \subseteq \{i, \ldots, n\}$. Now three cases arise:

Case 1: If $i \in B_{i-1}$ then $S_{i-1} + i \subseteq B_{i-1}$. Therefore, $S_{i-1} + i$ is independent. So now $B_i = B_{i-1}$ and $S_i = S_{i-1} + i$ and $B_i - S_i \subseteq \{i+1, \ldots, n\}$.

Case 2: If $i \notin B_{i-1}$ and $S_{i-1} + i \notin \mathcal{I}$. Then $S_i = S_{i-1}$ and $B_i = B_{i-1}$. And $B_i - S_i \subseteq \{i+1, \ldots, n\}$.

Case 3: If $i \notin B_{i-1}$ but $S_{i-1} + i \in \mathcal{I}$. Then $S_i = S_{i-1} + i$. Now S_i can be extended to a B' by adding all but one element of B_{i-1}. So $|B'| = |B_{i-1}|$. Let the element which is not added is $j \in B_{i-1}$. So $B' = B_{i-1} + i - j$.

$$wt(B') = wt(B_{i-1}) - wt(j) + wt(i)$$

But we have $wt(i) \geq wt(j)$. So $wt(B') \geq wt(B_{i-1})$. Now since B_{i-1} has maximum weight we have $wt(B') = wt(B_{i-1})$. Then our $B_i = B'$. So $B_i - S_i \subseteq \{i+1, \ldots, n\}$.

Hence, the claim is true for the ith stage as well. Therefore, the claim is true. ∎

Let the final set after n iterations be the set $T = \{t_1, \ldots, t_l\}$. Now we will prove that T is a maximum weight independent set.

Claim 5.3.2
At any iteration, $T_i = \{t_1, \ldots, t_k\}$, then T_i is a maximum weight independent set with at most i elements

Proof: We will prove by induction. Base Case: $i = 0$. Then $T_i = \emptyset$. So the statement follows naturally.

Assume T_{i-1} is maximum weight independent set with at most $i-1$ elements. Now for a contradiction, say $\hat{T}_i \in \mathcal{I}$ of size at most i with strictly larger weight than T_i. Then $\exists\, x \in \hat{T}_i - T_{i-1}$ such that $T_{i-1} \cup \{x\} \in \mathcal{I}$. Then we have

$$wt(\hat{T}_i - x) \leq wt(T_{i-1})$$

by inductive hypothesis. The only element that extend T_{i-1} are those which comes after t_i. Therefore, $wt(x) \leq wt(t_i)$. Hence, we have

$$wt(\hat{T}_i - x) + wt(x) \leq wt(T_{i-1}) + wt(t_i) \implies wt(\hat{T}_i) \leq wt(T_i)$$

But we assumed that $wt(\hat{T}_i) > wt(T_i)$. Hence, contradiction ↯ ■

Therefore using the claims, after the algorithm finished we have no elements left to check, so the current set has the maximum weight which is also an independent set. So the algorithm successfully returns a maximum weight base. ■

5.3.3 Job Selection with Penalties

FIND FEASIBLE SCHEDULE

Input: Set J of n jobs with deadlines $d_1, \ldots, d_n$ and rewards $w_1, \ldots, w_n$

Question: Each jobs unit time and we have a single machine to process their jobs. Give a feasible schedule of jobs with maximum reward

First let's define what is a schedule and what is even a feasible schedule:

Definition 5.3.2: Feasible Schedule

For a subset S of jobs:

1. A schedule is an ordering of S
2. A feasible schedule is one where one job in S gets finished by deadline.
3. A set $S \subseteq J$ is feasible if S has a feasible schedule.

Now for any $S \subseteq J$, and $t \in \mathbb{Z}_0$, define $N_t(S) = \{j \in S : d_j \leq t\}$. Then we have the following lemma:

Lemma 5.3.8

Let $S \subseteq J$. The following are equivalent:

1. S is feasible
2. $\forall\, t \in \mathbb{Z}_0$, $|N_t(S)| \leq t$
3. A schedule that orders jobs in S by deadline is feasible

Proof:

3 $\implies$ 1: This follows naturally

1 $\implies$ 2: Suppose not. Then $\exists\, t$ such that $|N_t(S)| > t$. Then by time t, greater than t many jobs have to be completed. But S is feasible, so every job is finished by deadlines and each job takes unit take. Hence by time t, more than t jobs can not be finished. Hence, contradiction.

2 $\implies$ 3: The schedule orders the jobs by deadline. We will use induction on t. For $t = 1$ we have $|N_1(S)| \leq 1$. Hence, by $t = 1$ at most one job is completed. At $t = 1$ the jobs are completed within deadline. Suppose till time $t-1$ the jobs are completed within deadlines. At time t we have $|N_t(S)| \leq t$. Therefore, all the jobs with deadlines $\leq t$ in S. So all the jobs in $N_t(S)$ can be completed within time t in any order. Therefore, if we complete the jobs with deadline $< t$ first, and then we can complete all the jobs with deadline t within time t. Hence, at time t all the jobs are completed within their deadlines. Hence, by mathematical induction at time $t = n$ all the jobs are completed within deadline. Therefore, the schedule orders jobs by deadline then it is a feasible schedule.

■

Lemma 5.3.9

Consider $M = (J, \mathcal{I})$ where S is feasible $\implies S \in \mathcal{I}$. Then M is a matroid. (Assume that no two jobs have same deadline)

Proof: Suppose $D \coloneqq$ the maximum of all deadlines. Consider the set

$$\mathscr{L} = \{N_t(J) : t \in [D]\}$$

Hence $\mathscr{L}$ is a laminar family. Then take $\mathcal{I}' = \{S \subseteq J : |N_t(S)| \leq t \ \forall t \in [D]\}$. By Lemma 5.3.4 $M = (J, \mathcal{I}')$ is a laminar matroid. And by Lemma 5.3.8 $\mathcal{I}'$ is the set of feasible schedules. Therefore $\mathcal{I}' = \mathcal{I}$. Hence M is a matroid. ■

Alternate Proof :

① Downward Closure: If $S \in \mathcal{I}$ then S is feasible. Then for any subset T of S all the jobs are completed within deadlines since S is feasible. So $T \in \mathcal{I}$.

② Exchanges Property: Given $S, T \in \mathcal{I}$ and $|T| < |S|$. Now order S and T by deadlines. Let j be the job with largest deadline that is not in T i.e. $j = \max_{i \in S \setminus T} d_i$. Then we claim that $T \cup \{j\} \in \mathcal{I}$.

Now define

$$T^{<} = \{i \in T : d_i < d_j\} \qquad T^{>} = \{i \in T : d_i > d_j\}$$

And also similarly define

$$S^{<} = \{i \in S : d_i < d_j\} \qquad S^{>} = \{i \in S : d_i > d_j\}$$

As we defined j we have $T^{>} = S^{>}$. Since we have $|S| > |T|$ we have $|S^{<}| \geq |T^{<}|$.

Now if $T \cup \{j\}$ is not feasible then $\exists\ t$ such that $|N_t(T \cup \{j\})| > t$. Since T is feasible we have $|N_t(T)| \leq t$. Hence $t \geq d_j$ otherwise $N_t(T \cup \{j\}) = N_t(T)$. But then

$$|N_t(T \cup \{j\})| = |T^{<}| + 1 + |\{i \in T \cup \{j\} : d_j < d_i \leq t\}| \leq |S^{<}| + 1 + |\{i \in S \cup \{j\} : d_j < d_i \leq t\}| = |N_t(S)| \leq t$$

Therefore we obtain $|N_t(T \cup \{j\})| \leq t$. Hence contradiction. Therefore $T \cup \{j\}$ is feasible.

■

Chapter 6

Dijkstra Algorithm with Data Structures

Minimum Weight Path
Input: Directed Graph $G = (V, E)$, $s \in V$ is source and $W = \{w_e \in \mathbb{Z}_0 : e \in E\}$
Question: $\forall\, v \in V - \{s\}$ find minimum weight path $s \rightsquigarrow v$.

This is the problem we will discuss in this chapter. In this chapter we will often use the term 'shortest distance' to denote the minimum weight path distance. One of the most famous algorithm for finding out minimum weight paths to all vertices from a given source vertex is Dijkstra's Algorithm

6.1 Dijkstra Algorithm

We will assume that the graph is given as adjacency list. Dijkstra Algorithm is basically dynamic programming. Suppose $\delta(v)$ is the shortest path distance from $s \rightsquigarrow v$. Then we have the following relation:

$$\delta(v) = \min_{u:(u,v)\in E} \{\delta(u) + e(u, v)\}$$

And suppose for any vertex $v \in V - \{s\}$, $dist(v)$ be the distance from s estimated by the algorithm at any point. This is why Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with minimum shortest-path estimate and estimates the distances of neighbors of u. So here is the algorithm:

Algorithm 16: Dijkstra(G, s, W)

Input: Adjacency Matrix of digraph $G = (V, E)$, source vertex $s \in V$ and weight function $W = \{w_e \in \mathbb{Z}_0 : e \in E\}$
Output: $\forall\, v \in V - \{s\}$ minimum weight path from $s \rightsquigarrow v$

```
1 begin
2     S ⟵ ∅, U ⟵ V
3     dist(s) ⟵ 0, ∀ v ∈ V − {s}, dist(v) ⟵ ∞
4     while U ≠ ∅ do
5         u ⟵ min_{u∈U} dist(u) and remove u from U
6         S ⟵ S ∪ {u}
7         for e = (u, v) ∈ E do
8             dist(v) ⟵ min{dist(v), dist(u) + w(u, v)}
```

Here below we give an example of how the Dijkstra algorithm works:

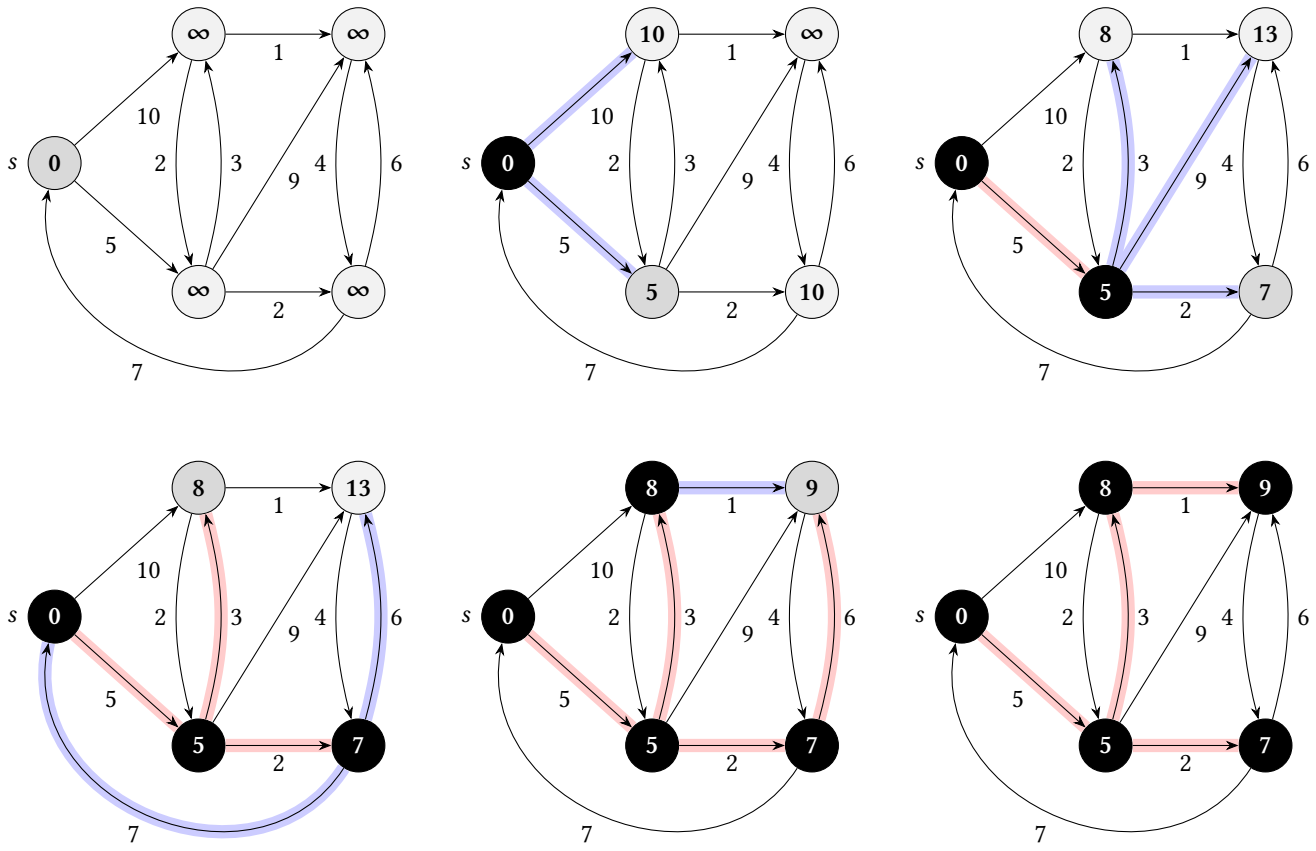


Figure 6.1: The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S and at any iteration of while loop the shaded vertex has the minimum value. At any iteration the red edges are the edges considered in minimum weight path from s using only vertices in S.

Suppose at any iteration t, let $dist_t(v)$ denotes the distance v from s calculated by algorithm for any $v \in V$ and $S^{(t)}$ denote the content of S at t^{th} iteration. In order to show that the algorithm correctly computes the distances we prove the following lemma:

Theorem 6.1.1

For each $v \in S^{(t)}$, $\delta(v) = dist_t(v)$ for any iteration t.

Proof: We will prove this induction. Base case is $|S^{(1)}| = 1$. S grows in size. Then only time $|S^{(1)}| = 1$ is when $S^{(1)} = \{s\}$ and $d(s) = 0 = \delta(s)$. Hence, for base case this is correct.

Suppose this is also true for $t-1$. Let at t^{th} iteration the vertex $u \in V - S^{(t-1)}$ is picked. By induction hypothesis for all $v \in S^{(t)} - \{u\}$, $dist_t(v) = dist_{t-1}(v) = \delta(v)$. So we have to show that $dist_t(u) = \delta(u)$.

Suppose for contradiction the shortest path from $s \rightsquigarrow u$ is P and has total weight $= \delta(u) = w(P) < dist_t(u)$. Now P starts with vertices from $S^{(t)}$ by eventually leaves S. Let (x, y) be the first edge in P which leaves S i.e. $x \in S$ but $y \notin S$. By inductive hypothesis $dist_t(x) = \delta(x)$. Let P_y denote the path $s \rightsquigarrow y$ following P. Since y appears before u we have

$$w(P_y) = \delta(y) \leq \delta(u) = w(P)$$

Now

$$dist_t(y) \leq dist_t(x) + w(x, y)$$

since y is adjacent to x. Therefore

$$dist_t(y) \leq dist_t(x) + w(x, y) = \delta(y) \leq dist_t(y) \implies dist_t(y) = \delta(y)$$

Now since both $u, y \notin S^{(t)}$ and the algorithm picked up u we have $\delta(u) < dist_t(u) \leq dist_t(y) = \delta(y)$. But we can not have both $\delta(y) \leq \delta(u)$ and $\delta(u) < \delta(y)$. Hence contradiction ↯ Therefore $\delta(u) = dist_t(u)$. Hence by mathematical induction for any iteration t, for all $v \in S^{(t)}$, $\delta(v) = dist_t(v)$. ■

Therefore, by the theorem after all iterations S has all the vertices with their shortest distances from s and henceforth the algorithm runs correctly.

Now in the algorithm there are two things which we needed to keep track of. At every iteration of the while loop we needed to find the vertex u which had the minimum distance from the source vertex, and we needed to update the distance of a vertex by decreasing the value as needed. So after the decrease we need to update the minimum distance vertex. So in any data structure used to do these two operations we need the following things:

- Need to store: $\forall\, v \in V$, $dist(v)$
- Operations:
  - EXTRACT-MIN: Gives the vertex with minimum distance in v and remove from the data structure.
  - DECREASE-KEY: For vertex v reduce $dist(v)$ to k.

In the algorithm we called EXTRACT-MIN n times and DECREASE-KEY m times.

6.2 Data Structure 1: Linear Array

So naively we can use a linear array of size n where each element of the array corresponds to a vertex. Each element of the array is a tuple of flag of being in U and the value $dist(v)$. To EXTRACT-MIN it takes $O(n)$ time since we need to compare all the elements and for DECRESE-KEY it takes $O(1)$ time. Therefore Dijkstra takes $n \cdot O(n) + O(m) = O(n^2)$ time.

6.3 Data Structure 2: Min Heap

A binary heap data structure is an array object that we can view as "Almost complete binary" tree (ACB tree). Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest which is filled from the left up to a point.

Let the ACB tree has height h. Then heap is completely field until height $h-1$ i.e. every vertex up to level $h-2$ has exactly two children and a node at height $h-1$ if missing a child then:

- Either both children are missing or only right child are missing.
- Every vertex to the right of the node is missing both children.

An ACB tree height h is represented as an array of size $2^h - 1$. For vertex v stored at $A[i]$, the left child of A is at $A[2i]$ and the right child is at $A[2i+1]$ and the parent of A is at $A\left[\left\lfloor \frac{i}{2} \right\rfloor\right]$.

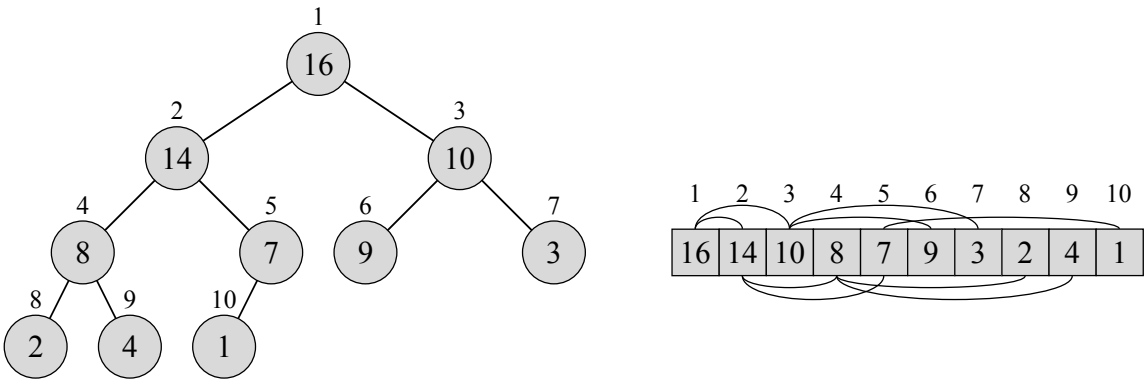


Figure 6.2: A max-heap viewed as an ACB tree (left) and as an array (right)

Here we will study min-heap where the value of the children is more than the value of its parent.

6.3.1 Extracting the Minimum

For minimum, we already know the root of the heap or the first element of A is the minimum. But after extracting the minimum we need to balance the heap so that it gets the properties of min-heap back. For that we replace the root with the right most element in the array A. Then balance the heap by moving it down if one of the child has *key* smaller than *node.key* and keep doing it until both child is larger.

Algorithm 17: Extract-Min(A)

```
1  t ⟵ A.size, Minval ⟵ A[1].key
2  A[1] ⟵ A[t]
3  t ⟵ t − 1, i ⟵ 1
4  while True do
5      if 2i ≤ t then
6          left − val ⟵ A[2i].key
7      else
8          return minval                          // No left child i.e. already at leaf
9      if 2i + 1 ≤ t then
10         right − val ⟵ A[2i + 1].key
11     else
12         right − val ⟵ ∞                                          // No right child
13     if left − val ≤ right − val and A[i].key < left − val then
14         curr_elm ⟵ A[i]
15         A[i] ⟵ A[2i]
16         U[2i] ⟵ curr_elm
17         i ⟵ 2i
18     else if right − val < left − val then
19         curr_elm ⟵ A[i]
20         A[i] ⟵ A[2i + 1]
21         U[2i + 1] ⟵ curr_elm
22         i ⟵ 2i + 1
23     else
24         Break
25     return minval
```

In this algorithm for extracting min each time the height of the new root node increases by one at each iteration of the while loop. Hence, this takes at most $O(\log n)$ time.

6.3.2 Decreasing Key of a Node

After decreasing the *key* of a node it may have smaller key than it's parent node. So move it upward i.e. replace with its parent node, and we keep doing it until the parent node has smaller value than it. Here again at each iteration of the

Algorithm 18: Decrease-Key(A, i, k)

```
1  t ⟵ A.size
2  A[i] ⟵ k
3  while i > 1 and A[i].key < A[⌊i/2⌋].key do
4      curr_elm ⟵ A[i]
5      A[i].key ⟵ A[⌊i/2⌋]
6      A[⌊i/2⌋] ⟵ curr_elm
7      i ⟵ ⌊i/2⌋
```

while loop the height decreases by 1. Hence, this takes at most $O(\log n)$ time.

6.3.3 Time Complexity Analysis of Dijkstra

Both EXTRACT-MIN and DECREASE-KEY takes $O(\log n)$ time for a min-heap. Now in a Dijkstra algorithm there are n calls for EXTRACT-MIN and m calls for DECREASE-KEY. Therefore, the total time taken by Dijkstra is $O(n \log n) + O(m \log n) = O(m \log n)$. But this is better when $m = o\left(\frac{n^2}{\log n}\right)$. Now we will show an improvement of min-heap where the amortized cost of EXTRACT-MIN is $O(\log n)$ and amortized cost of DECREASE-KEY is constant. But first we will take a detour of explaining amortized analysis.

6.4 Amortized Analysis

In amortized analysis, we average the time required to perform a sequence of data-structure operations over all the operations performed. With amortized analysis, we can show that the average cost of an operation is small, if we average over a sequence of operations, even though a single operation within the sequence might be expensive.

Note:-

Amortized analysis in not average-case analysis as amortized analysis guarantees the average performance of each operation in the worst case

Consider the following algorithm: Now the number of bit flips in this process is $1, 2, 1, 3, \ldots, n, \ldots$. At any point

Algorithm 19: Amortized Analysis

```
Input: n
begin
    t ⟵ 0, X(0) ⟵ 0^n
    while True do
        t ⟵ t + 1
        X(t) ⟵ X(t − 1) + 1
```

the number of bit flips can be at most n. In the worst case an operation has cost n. We want to compute the average cost for an operation. We will show that starting from $X(0) = 0^n$ the average cost is at most 2. Furthermore, we will show 3 different proofs of this.

Lemma 6.4.1

The total cost of bit flips for t operations is at most $2t$.

Proof 1 (Counting): In t operations:

$$\begin{aligned} n^{th} \text{ bit gets flipped:} &\quad t \text{ times} \\ (n-1)^{th} \text{ bit gets flipped:} &\quad \lfloor \tfrac{t}{2} \rfloor \text{ times (when } n^{th} \text{ bit is 1)} \\ (n-2)^{th} \text{ bit gets flipped:} &\quad \lfloor \tfrac{t}{4} \rfloor \text{ times (when } (n-1)^{th} \text{ bit is 1)} \\ \vdots &\quad \vdots \end{aligned}$$

Therefore the total number of bit flips we get is $\leq t\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right) \leq 2t$. ■

Now we will give a proof using the accounting method. In the accounting method of amortized analysis, we assign each operation an amortized cost that may differ from its actual cost. If the amortized cost is higher, the excess is stored as credit on data structure objects; if lower, credit is used to cover the gap. This way, expensive operations can be balanced by cheaper ones, and different operations may have different amortized costs.

Proof 2 (Charging): Suppose every operation costs 2 Rs.

- Every change from $0 \rightarrow 1$ charges 1 Rs and store 1 Rs.

- Every change from $1 \rightarrow 0$ charges 2 Rs.

Now as you can see to change from $1 \rightarrow 0$ that bit was previously changed from $0 \rightarrow 1$. So to change from $1 \rightarrow 0$ we can use the stored 1 Rs. Hence, in average every operation costs exactly 2 Rs. Since there are t operations total number of bit flips is at most $2t$. ■

In the next proof we will analyze by computing a necessary potential function. After each operation we can calculate the potential difference.

Proof 3 (Potential): Consider the potential function $\Phi(i) = \#1\text{'s in } X(i)$. Let at i^{th} operation t_i bits were flipped from $1 \rightarrow 0$. Now any operation flips at most 1 bit from $0 \rightarrow 1$. Therefore, number of bit flips in i^{th} operation is at most $t_i + 1$. Therefore, we have

$$\Phi(i) \leq \Phi(i-1) - t_i + 1$$

since the number of 1's in $\Phi(i)$ is decreased by t_i many $1 \rightarrow 0$ flips and then increased because of 1 flip from $0 \rightarrow 1$. Therefore, the cost at i^{th} operation is $t_i + 1 \leq \Phi(i-1) - \Phi(i) + 2$. Hence, the total number of bit flips in t operations is $\Phi(0) - \Phi(t) + 2t \leq 2t$. ■

Hence, after t operations the total number of bit flips is at most $2t$. Therefore, on average the cost per operation is at most 2. Hence, the amortized cost of the operation is 2. So we will use such amortized analysis on the next data structure to optimize the run time of Dijkstra Algorithm.

6.5 Data Structure 3: Fibonacci Heap

Instead of keeping just one Heap we will now keep an array of Heaps. We will also discard the idea of binary trees. We will now use a data structure which will take the benefit of the faster time of both the data structure i.e.

	EXTRACT-MIN	DECREASE-KEY
Linear Array	$O(n)$	$\boxed{O(1)}$
Min-Heap	$\boxed{O(\log n)}$	$O(\log n)$
Fibonacci Heap	$O(\log n)^*$	$O(1)^*$

Remark: The $*$ is because of the amortized time.

Since Fibonacci heap is an array of heaps there is a *rootlist* which is the list of all the roots of all the heaps in the Fibonacci heap. There is a *min-pointer* which points to the root with the minimum key. For each node in the Fibonacci heap we have a pointer to its parent and we keep 3 variables. The 3 variables are *degree*, *size* and *lost* where *lost* is a Boolean Variable.

- For any node x in the Fibonacci heap the $x.degree$ is the number of children x has.
- $x.size$ is the number of nodes in the tree rooted at x.
- $x.lost$ is 1 if and only if x has lost a child before.

Why any node will lose a child that explanation we will give later. With this set up let's dive into the data structure.

6.5.1 Inserting Node

To insert a node we call the FIB-INSERT function and in the function the algorithm initiates the node with setting up all the pointers and variables then add the node to the *rootlist*.

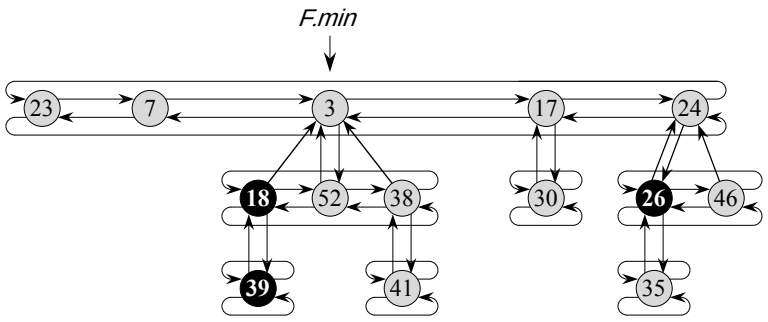


Figure 6.3: A Fibonacci Heap with 5 heaps in the rootlist

Algorithm 20: FIB-CREATE-NODE(v)

```
1 x.degree ⟵ 0
2 x.parent ⟵ None
3 x.child ⟵ None
4 x.lost ⟵ 0
5 x.key ⟵ v
6 return x
```

Algorithm 21: FIB-INSERT(F, v)

```
1 x ⟵ FIB-CREATE-NODE(v)
2 if F.min == None then
3 |   F.rootlist ⟵ [x]
4 |   F.min ⟵ x
5 else
6 |   F.rootlist.append(x)
7 |   if x.key < F.min.key then
8 |   |   F.min ⟵ x
```

All of this can be done in $O(1)$ time. Therefore, to insert a node in the Fibonacci heap it takes $O(1)$ time.

6.5.2 Union of Fibonacci Heaps

To unite two Fibonacci heaps F_1 and F_2 we simply concatenate the root lists of F_1 and F_2 and then determine the new minimum node. All the operations here can be done in constant time. Hence, FIB-UNION takes $O(1)$ time.

Algorithm 22: FIB-UNION(F_1, F_2)

```
1 F ⟵ MAKE-FIB-HEAP
2 F.min ⟵ F1.min
3 F.rootlist ⟵ F1.rootlist++F2.rootlist
4 if F2.min < F1.min then
5 |   F.min ⟵ F2.min
6 return F
```

6.5.3 Extracting the Minimum Node

The FIB-EXTRACT-MIN function extracts the minimum node from the Fibonacci heap F and then rearranges the heap array. It works by first making a root node out of each of the minimum node's children and removing the minimum node from the rootlist. It then consolidates the root list by linking roots of equal degree until at most one root remains of each degree.

Algorithm 23: Fib-Extract-Min(F)

```
1  z ⟵ F.min
2  if z ≠ None then
3      for x ∈ z.child do
4          F.rootlist.append(x)
5          x.parent ⟵ None
6      Remove z from F.rootlist
7      if z == z.right then
8          F.min ⟵ None
9      else
10         F.min ⟵ z.right consolidate(F)
11 return z
```

Algorithm 24: Fib-Heap-Link(H, y, x)

```
1 Remove y from F.rootlist
2 y.parent ⟵ x
3 y.lost ⟵ 0
```

Algorithm 25: Consolidate(F)

```
1  Initialize array A[0, ..., D(n)] with None elements.
2  for x ∈ F.rootlist do
3      d ⟵ x.degree
4      if A[d] == None then
5          A[d] ⟵ x
6      while A[d] ≠ None do
7          y ⟵ A[d]
8          if y.key < x.key then
9              Exchange x with y
10         Fib-Heap-Link(F, y, x)
11         A[d] ⟵ None, d ⟵ d + 1
12     A[d] ⟵ x
13 F.min ⟵ None
14 for i = 0 to D do
15     if A[i] ≠ None then
16         if F.min == None then
17             F.rootlist ⟵ [A[i]], F.min ⟵ A[i]
18         else
19             F.rootlist.append(A[i])
20             if A[i].key < F.min.key then
21                 F.min ⟵ A[i]
```

Here $D(n)$ denotes the maximum degree a node can have after Consolidate. The procedure Consolidate uses an auxiliary array of size A of size $D(n)$ which we will choose later. For each $i \leq D(n)$ it keeps a heap of degree i. And if it finds two heaps of same degree then it makes the one with higher key to be the child of the other one. The function

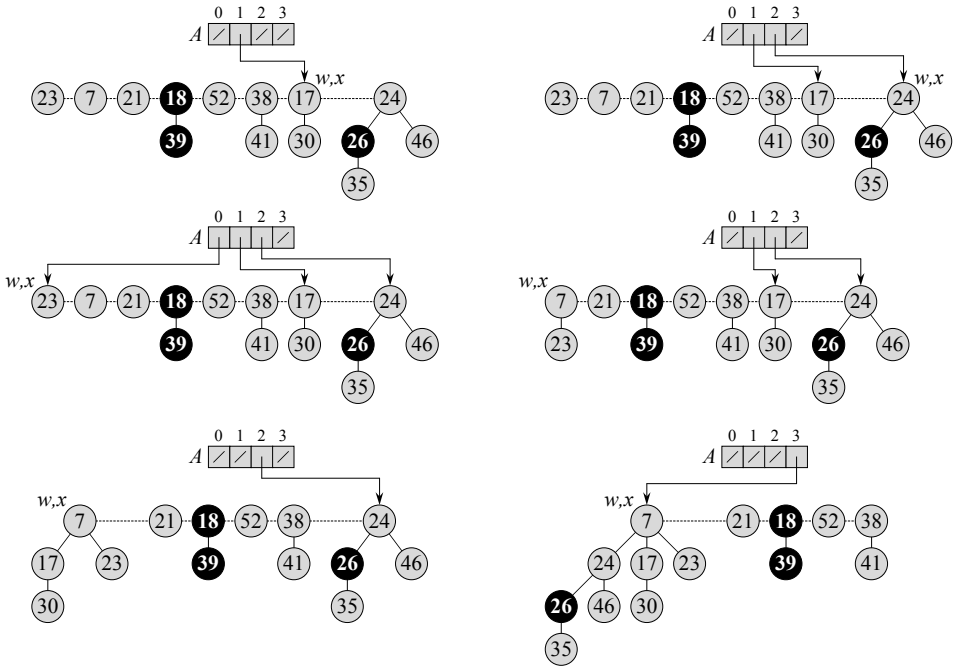


Figure 6.4: A run of Consolidate

Fib-Heap-Link does this process of linking two heaps of same degree.

Of course in order to allocate array we have to know how to calculate the upper bound for $D(n)$ on the maximum degree. We will show an upper bound of $O(\log n)$ in subsection 6.5.5.

Now in FIB-EXTRACT-MIN in each iteration of the outer for loop or inner while loop it operates on one heap in $F.rootlist$. Hence it takes $O(D(n) + \#\text{heaps in } F.rootlist)$ time.

6.5.4 Decreasing Key of a Node

In this section we will show how to decrease a key of a node in a Fibonacci heap in $O(1)$ amortized time. The FIB-DECREASE-KEY function decreases the key value of the target node then if the min-heap order the node is in is violated then we use the CASCADING-CUT function to restore the min-heap property again. These two functions operates like the following:

Algorithm 26: FIB-DECREASE-KEY(F, x, k)

```
1 if k > x.key then
2     return Error
3 x.key ⟵ k
4 y ⟵ x.parent
5 if y ≠ None and x.key < y.key then
6     CUT(F, x, y)
7     CASCADING-CUT(F, y)
8 if k < F.min.key then
9     F.min ⟵ x
```

Algorithm 27: CASCADING-CUT(F, y)

```
1 if y.parent ≠ None then
2     if y.lost == 0 then
3         y.lost ⟵ 1
4     else
5         CUT(F, y, y.parent)
6         CASCADING-CUT(F, y.parent)
```

Algorithm 28: CUT(F, x, y)

```
1 Remove x from y.child
2 y.degree ⟵ y.degree − 1
3 F.rootlist.append(x)
4 x.parent ⟵ None, x.lost ⟵ 0
```

After decreasing the key of the target node if the min-heap order has been violated then we start by cutting the link between x and its parent and adding it to the rootlist. Let x is a node in F. At some time x was a root. Then x was linked

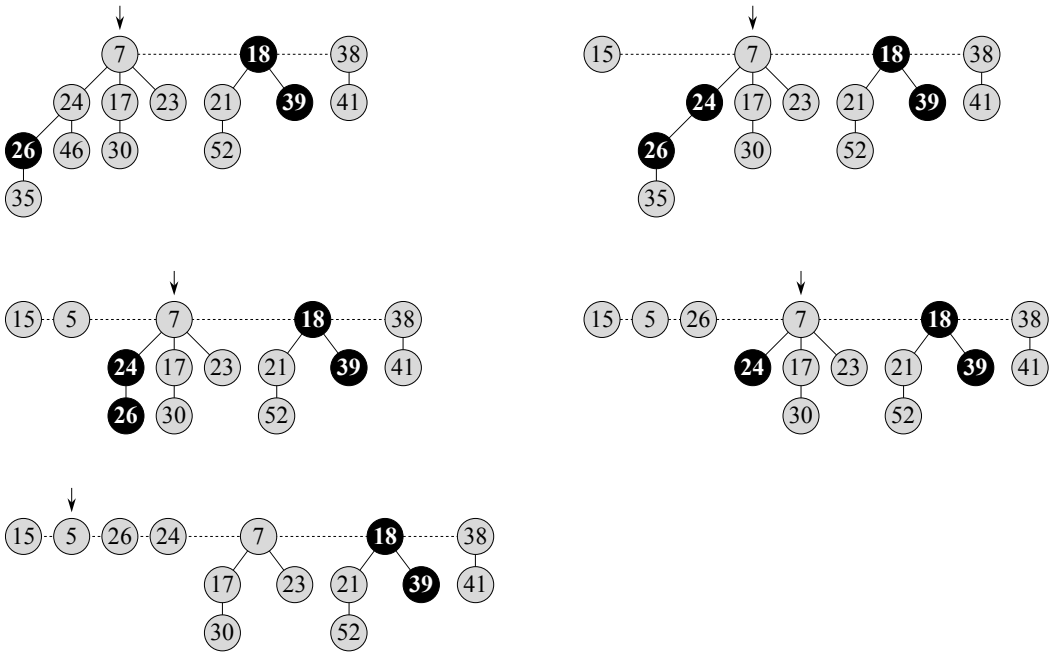


Figure 6.5: A run of CASCADING-CUT. First FIB-DECREASE-KEY$(F, 46, 15)$ and then FIB-DECREASE-KEY$(F, 35, 5)$ are called.

to another node. Suppose at some time two children of x were removed by cuts. As soon as second child has been lost we cut x from its parent and make it a new root. But we are not done yet. Since x might be the second child cut from its

parent. So we have to check for its parent. Therefore, we recursively run Cascading-Cut on its parent till we reach the root or cut the first child from a node.

Notice at each run of Cascading-Cut the *lost* bit of a node is getting reset. Therefore, the total time taken by Fib-Decrease-Key is $O(1 + \#\text{lost bits reset})$.

6.5.5 Bounding the Maximum Degree

To prove that the amortized time of Fib-Extract-Min and Fib-Decrease-Key are $O(\log n)$ and $O(1)$ we must show that upper bound of the maximum degree of any node after Consolidate function is $O(\log n)$. In particular, we will show its $\left\lfloor \log_\phi n \right\rfloor$ where ϕ is the golden ratio.

Lemma 6.5.1

Let x be any node in a Fibonacci heap, and suppose that $x.degree = k$. Let $y_1, \ldots, y_k$ denote the children of x in the order in which they were linked to x from the earliest to the latest. Then $y_1.degree \geq 0$ and $y_i.degree \geq i - 2$ for $i = 2, \ldots, k$.

Proof: Obviously $y_1.degree \geq 0$. The only function that adds a child to a node is the function Consolidate. Now for $i \geq 2$, y_i was linked to x when all of $y_1, \ldots, y_{i-1}$ were children of x, and therefore we must have had $x.degree \geq i - 1$. Because node y_i is linked to x only if $x degree = y_i.degree$ we must also have $y_i.degree \geq i - 1$. Since then node y_i has lost at most one child, since it would have been cut from x by Cascading-Cut if it had lost two children. We conclude that $y_i.degree \geq i - 2$. ■

Lemma 6.5.2

Let x be a node in a Fibonacci heap and let $k = x.degree$. Then

$$size(x) \geq F_{k+2} \geq \phi^k$$

Proof: We will prove this using induction. For $k = 0$, $F_2 = 1$ so this is obviously true. For $k = 1$ there is one child of x. Hence, $size(x) = 2 = F_3$. Suppose this is true for $1, \ldots, k - 1$. Let $y_1, \ldots, y_k$ are the children of x in the order in which they were linked to x. By the above lemma we have $y_1.degree \geq 0$ and $y_i.degree \geq i - 2$ for all $i = 2, \ldots, k$. Hence, by Induction hypothesis we have $size(y_i) \geq F_{i-2}$ for all $i = 2, \ldots, k$. Therefore,

$$size(x) \geq 1 + \sum_{i=1}^{k} size(y_k) \geq 2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=1}^{k} F_i = F_{k+2} \geq \phi^k$$

Hence, we have the lemma. ■

Corollary 6.1

The maximum degree of any node in Consolidate, $D(n) = O(\log n)$.

6.5.6 Time Complexity Analysis of Dijkstra

Now we will calculate the amortized time of Dijkstra algorithm. Before that we will calculate the amortized cost of the data structure. Let in an algorithm Fib-Extract-Min was called t times. Therefore, total cost of all t many Fib-Extract-Min calls is $O(t \log n + \text{total \#heaps created})$. Now heaps are created because of Fib-Extract-Min functions and Fib-Decrease-Key function. We know Fib-Extract-Min were called t times and each time it created $O(\log n)$ heaps. Hence, in total Fib-Extract-Min created $O(t \log n)$ heaps. Therefore, time taken by the t many Fib-Extract-Min calls is $O(t \log n + \#\text{Fib-Decrease-Key calls})$.

Now suppose in that algorithm the function Fib-Decrease-Key were called k times. Hence, this takes $O(k + \#\text{total number of lost bits reset}) = O(k + \#\text{total number of lost bits set})$ time. Now the *lost* bits are set only by the

Fib-Decrease-Key. Therefore, #total number of lost bits rset = #Fib-Decrese-Key was called. Therefore, the total time taken by all the Fib-Decrese-Key calls is $O(k)$.

Hence, in an algorithm if t times Fib-Extract-Min was called and k times Fib-Decrese-Key was called then total time taken by Fib-Extract-Min is $O(t \log n + k)$ and total time taken by Fib-Decrese-Key is $O(k)$. Therefore, amortized time taken by Fib-Extract-Min is $O(\frac{t}{k} \log n)$ and by Fib-Decrese-Key is $O(1)$.

Now in the Dijkstra algorithm Fib-Extract-Min is called n times and Fib-Decrese-Key is called $O(m)$ times where n is the number of vertices in the graph and m is the number of edges in the graph. Hence, the amortized cost of Fib-Extract-Min is $O(\log n)$ and Fib-Decrease-Key is $O(1)$. Therefore, using Fibonacci heap Dijkstra takes $(n \log n + m)$ time.

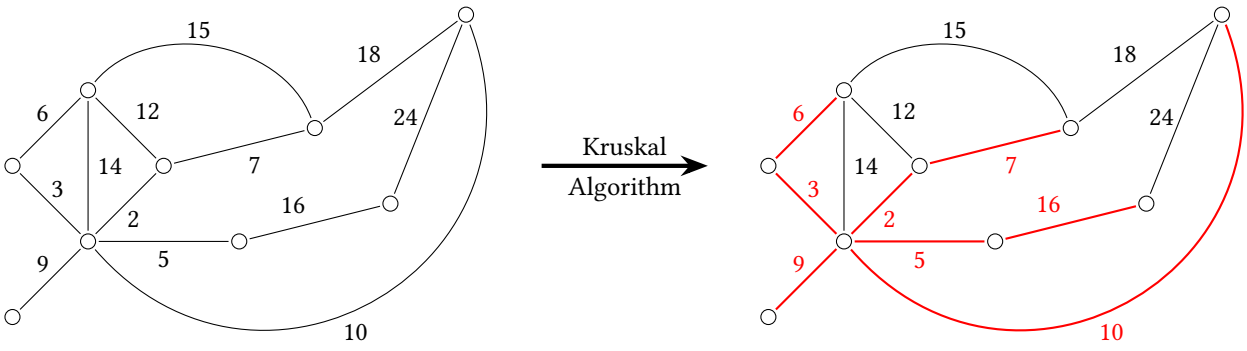
Chapter 7

Kruskal's Algorithm with Data Structures

Minimum Spanning Tree
Input: Weighted undirected graph $G = (V, E)$ and weights of edges $W = \{w_e \in \mathbb{Z}_0 : e \in E\}$.
Question: Find a spanning tree $T \subseteq E$ such that $\sum_{e \in T} w_e$ is minimum.

In this chapter we will discuss this problem. We will first discuss the Kruskal's algorithm which gives a greedy solution to the problem. Then we will discuss the data structure that we can use to implement the Kruskal's algorithm efficiently. We assume the graph is connected otherwise the algorithm can use a DFS to check connectivity.

7.1 Kruskal's Algorithm



The Kruskal's algorithm uses a concept of component to find the minimum spanning tree.

Definition 7.1.1: Component

In a graph $G = (V, E)$, a *component* is a maximal subgraph $G' = (V', E')$ of G such that

(1) (V', E') is connected.

(2) $\forall\, v \notin V'$, there is no edge $e \in E$ such that e connects v to any vertex in V'.

In Kruskal's algorithm we maintain a set of components each of them is a tree so basically we maintain a forest. And we find a safe edge which is always the least weight edge in the graph that connects two distinct components and adds that edge to the collection of edges in the forest and update the components.

So the algorithm first sorts the edges in non-decreasing order of their weights. Then it initializes a forest F with all the vertices in the graph and no edges. Then it iterates through the sorted edges and checks if the edge connects two distinct components. If it does, then it adds the edge to the forest and merges the two components. The algorithm stops when we have $n-1$ edges in the forest. We have shown in Lemma 5.3.5 that the set of collection of acyclic sets in

Algorithm 29: KRUSKAL'S ALGORITHM

Input: $G = (V, E)$, and weights of edges $W = \{w_e \in \mathbb{Z}_0 : e \in E\}$
Output: A minimum spanning tree $T \subseteq E$ of G

```
1 begin
2     T ⟵ ∅
3     Sort the edges in E in non-decreasing order of their weights so that w(e_1) ≤ w(e_2) ≤ ··· ≤ w(e_m)
4     for i = 1, ..., m do
5         Let e_i = (u, v)
6         if T ∪ {e_i} is acyclic then
7             T ⟵ T ∪ {e_i}
8         if |T| = |V| − 1 then
9             return T
```

any graph is a matroid. Hence, here we are basically finding a base of the graphic matroid with minimum weight. The algorithm is exactly similar to the greedy algorithm for finding max-weight base of a matroid in subsection 5.3.2. So you can use the similar arguments to show that the algorithm is correct and returns the minimum spanning tree of the graph.

Now in the algorithm the checking of $T \cup \{e_i\}$ is acyclic can be done by checking if both the end points are in same component or not. And if they are not then we need to combine those to components. But there comes a question:

Question 7.1

What it means to give a component?

We will use some vertex to represent the component. We keep a pointer $v.parent$ for each vertex which points to representative of component v is in. Hence, we need a data structure that can do the following two operations efficiently:

- FIND(u): Returns the component u is in.
- UNION(u, v): Merges the components of u and v into a single component.

So we can use the updated algorithm to implement the Kruskal's algorithm using proper data structure: The Kruskal's

Algorithm 30: KRUSKAL'S ALGORITHM

Input: $G = (V, E)$, and weights of edges $W = \{w_e \in \mathbb{Z}_0 : e \in E\}$
Output: A minimum spanning tree $T \subseteq E$ of G

```
1  begin
2      T ⟵ ∅
3      Sort the edges in E in non-decreasing order of their weights so that w(e_1) ≤ w(e_2) ≤ ··· ≤ w(e_m)
4      for i = 1, ..., m do
5          Let e_i = (u, v)
6          if FIND(u) ≠ FIND(v) then
7              T ⟵ T ∪ {e_i}
8              UNION(u, v)
9          if |T| = |V| − 1 then
10             return T
```

Algorithm calls m times the Find operation and n times the Union operation.

7.2 Data Structure 1: Linear Array

We create an n length array A which hold the parent pointer of each vertex. Initially for all vertices $A[v] = v$. So Array-Find(u) will just return $A[u]$. Hence, Find takes $O(1)$ time. For Union(u, v) we use the following: Therefore,

Algorithm 31: Array-Union(u, v)

```
1 if A[u] ≠ A[v] then
2     for i = 1, ..., n do
3         if A[i] == A[v] then
4             A[i] ⟵ A[u]
```

Array-Union(u, v) takes $O(n)$ time. Hence, the time complexity of the Kruskal's algorithm using this data structure is $m \cdot O(1) + n \cdot O(n) = O(m + n^2) = O(n^2)$.

7.3 Data Structure 2: Left Child Right Siblings Tree

Using an array is not efficient enough. One place we can optimize is if given the components is there a faster way to get the vertices in the component? We can use the following tricks to optimize:

1. For every representative of a component, store pointers to all vertices in that component.
2. Change representative for the smaller component while doing Union(u, v).

7.3.1 Construction

So now every representative of a component we point to one vertex which is also in the component. And from that vertex we can iterate through all the vertices in that component. So basically we can imagine a 2 level tree where all the children point towards the root which is the representative of the component. The root points to one of the children take the left most child. And then all the other children points to the immediate right child of the root.

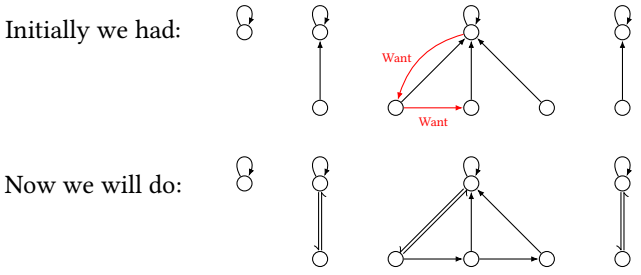


Figure 7.1: Left Child Right Sibling

We can also store a variable to store the number of vertices in the component so that we can use it to compare the size of two components and then update for the smaller one. Therefore, the data structure now stores:

- $v.parent$ for each v which points to the vertex representing the component v is in.
- $v.size$ for size of the component for each component representative v.
- $v.left$ for the left most child for each component representative v.
- $v.right$ for the immediate right sibling of v for all vertices in a component which are not representatives of the components.

This data structure is called Left Child Right Sibling. So in this data structure the LCRS-Find(u) just returns the value of $u.parent$. Hence, LCRS-Find takes $O(1)$ time.

7.3.2 LCRS-UNION Function

For the LCRS-UNION function we do the following

Algorithm 32: LCRS-UNION(u, v)

```
1  up ⟵ u.parent
2  vp ⟵ v.parent
3  if up ≠ vp then
4      if up == u then
5          u.parent ⟵ vp
6          u.right ⟵ vp.left
7          vp.left ⟵ u
8          vp.size ⟵ vp.size + 1
9      else if up.size ≤ vp.size then
10         up.right ⟵ u
11         x ⟵ up
12         while x.right == None do
13             x.parent ⟵ vp, x ⟵ x.right
14         x.right ⟵ vp.left, vp.left ⟵ up.left
15         vp.left ⟵ up
16         vp.size ⟵ vp.size + up.size
17     else
18         vp.right ⟵ v
19         x ⟵ vp
20         while x.right == None do
21             x.parent ⟵ up, x ⟵ x.right
22         x.right ⟵ up.left, up.left ⟵ vp.left
23         up.left ⟵ vp
24         up.size ⟵ up.size + vp.size
```

Below we have shown how the LCRS-UNION function works. This way we can unite two components and update the

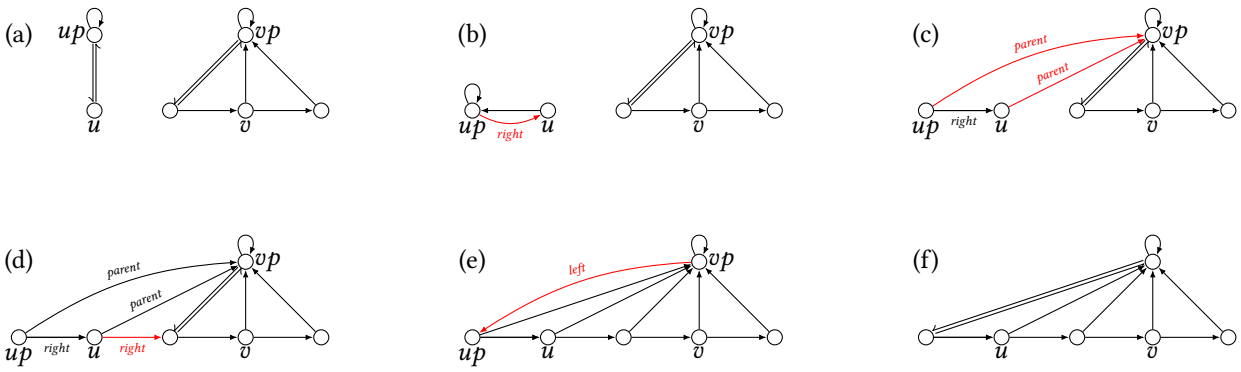


Figure 7.2: A run of LCRS-UNION(u, v)

corresponding component representative in the vertices of the smaller component. In the next section we will analyze the amortized time complexity of the LCRS-UNION function

7.3.3 Amortized analysis of LCRS-UNION

Lemma 7.3.1

For any vertex $v \in V$, $v.parent$ can change at most $O(\log n)$ times.

Proof: Initially size of v's component is 1. Each time $v.parent$ is changed the size of the component v is in becomes at least double. Therefore, at most $O(\log n)$ times $v.parent$ can change. ■

Now since there are n vertices at most $O(n \log n)$ times change of *parent* for any vertex happens. Now change of *parent* for any vertex happens only in LCRS-UNION function. Total time taken by all the LCRS-UNION operations is $O(n \log n)$ time. Since LCRS-UNION was called n times the amortized cost of LCRS-UNION is $O(\log n)$.

7.3.4 Time Complexity Analysis of Kruskal

We have shown above that LCRS-FIND takes $O(1)$ time and amortized cost of LCRS-UNION is $O(\log n)$. Since LCRS-FIND is called m times and LCRS-UNION is called n times the total run time of Kruskal's Algorithm using the Left Child Right Sibling data structure is $O(m + n \log n)$.

7.4 Data Structure 3: Union Find

We will now give up on the idea of height 1 trees to optimize more. Representative of a component is still vertex at root, but we will do the following: changes:

- UNION just changes parent pointers of root nodes, but it takes $O(1)$.
- Instead of size, we will maintain a variable *rank* of a component roughly which can be thought of as height of the component.
- For UNION root of smaller rank will be changed to point to root of the component of larger rank.
- The FIND operation does something called path compression which we will explain later.

To implement this data structure we will use a tree for each component and every node has a *parent* pointer. And there we will use the FIND and UNION operations.

Our goal is to run Kruskal's algorithm in almost $O(n + m)$ time. More precisely $O((n + m) \log^* n)$ time where $\log^* n$ is the number of times we need to compose log on n to get 1.

7.4.1 FIND Operation

The FIND operation does something called path compression i.e. for any vertex $v \in V$, if FIND(v) is called then it starts from v changes it's parent to the root of the component v is in, and then it moves to its parent and changes his parent to be the root and move to his parent and keep on doing like that till it reaches the root. In other words in path from the root to v, for every vertex in that path the FIND operation changes the *parent* pointer to the root.

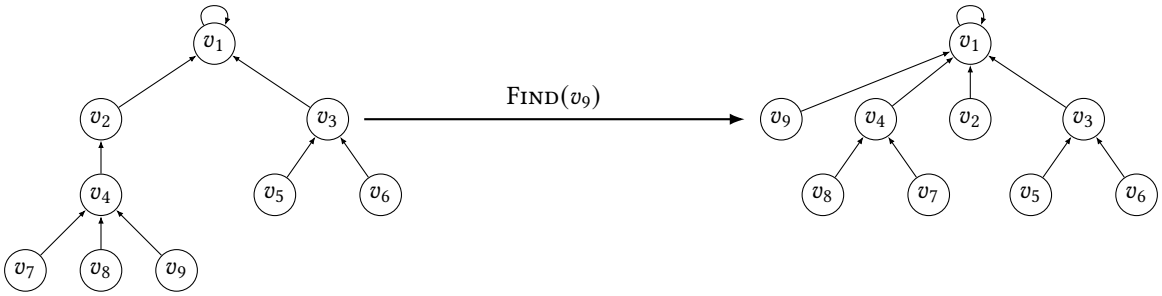


Figure 7.3: Path compression during FIND(v_9) operation

Here is the pseudocode of the FIND operation below: We will not discuss the amortized cost of this operation. Instead, we will do it in subsection 7.4.3.

Algorithm 33: FIND(v)

```
1 if v.parent ≠ v then
2     v.parent ⟵ FIND(v.parent)
3 return v.parent
```

7.4.2 UNION Operation

For the UNION operation we change the root of smaller rank to point to the root of the component of larger rank. So for the UNION(u, v) operation we assume that u, v are the roots of their respective components. Since in UNION(u, v) we assume

Algorithm 34: UNION(u, v)

```
1 if u.rank > v.rank then
2     v.parent ⟵ v
3 else if v.rank > u.rank then
4     u.parent ⟵ v
5 else
6     v.parent ⟵ u
7     u.rank ⟵ vu.rank + 1
```

u, v are component representatives before using UNION(u, v) we use FIND on u and v to get the roots of their components, and then we apply UNION on the roots. Hence, UNION(u, v) takes $O(1)$ time.

7.4.3 Analyzing the Union-Find Data-Structure

We call a node in the union-find data-structure a *leader* if it is the root of the tree.

Lemma 7.4.1

Once a node stop being a leader (i.e. the node in top of a tree). It can never become a leader again.

Proof: A node x stops being a leader only because of the UNION operation which made x child of a node y which is a leader of a tree. From this point on, the only operation that might change the parent pointer of x is the FIND operation which traverses through x. Since path-compression only change the parent pointer of x to point to some other node y. Therefore, the parent pointer of x will never become equal to itself i.e. x can never be a leader again. Hence, once x stops being a leader it can never be a leader again. ■

Lemma 7.4.2

Once a node stop being a leader then its rank is fixed.

Proof: The rank of a node changes only by a UNION operation. But the UNION operation only changes the rank of nodes that are leader after the operation is done. Therefore, once a node stops being a leader it's rank will not being changed by a UNION operation. Hence, once a node stop being a leader then its rank is fixed. ■

Lemma 7.4.3

Ranks are monotonically increasing in the trees, as we travel from a node to the root of the tree.

Proof: To show that the ranks are monotonically increasing it suffices to prove that for all edge $u \to v$ in the data structure we have $\text{rank}(u) < \text{rank}(v)$. And this is true because the UNION operation only changes the parent pointer

of a node u to point to a node v if u and v were leaders and either $\text{rank}(u) < \text{rank}(v)$ before making the change or $\text{rank}(u) = \text{rank}(v)$ before making the change and then the algorithm increases the rank of v by 1. Hence, the ranks are monotonically increasing in the trees, as we travel from a node to the root of the tree. ■

Lemma 7.4.4

When a node gets rank k than there are at least $\geq 2^k$ elements in its subtree.

Proof: We'll prove it using induction. For $k = 0$ it is obvious since a single element in the set. Now a node gets rank k only if two roots of rank $k-1$ were merged. By inductive hypothesis they each have at least $\geq 2^{k-1}$ nodes in their subtrees. Hence, the merged tree has $\geq 2^{k-1} + 2^{k-1} = 2^k$ nodes. ■

With this lemma we get the following corollaries:

Corollary 7.4.5

The following are true:

1. The number of nodes that get assigned rank k throughout the execution of the Union-Find data-structure is at most $\frac{n}{2^k}$.
2. For all vertices v, $v.rank \leq \lfloor \log n \rfloor$
3. Height of any tree $\leq \lfloor \log_2 n \rfloor$

Lemma 7.4.6

The time to perform a single find operation when we perform union by rank and path compression is $O(\log n)$ time.

We will show that we can do much better. In fact, we will show that for m operations over n elements the overall running time is $O((n+m)\log^* n)$

Definition 7.4.1: $\log^* n$

It is the number of times we need to take log to get less than or equal to 1. So

$$\log^* n = \begin{cases} 0 & \text{if } n \leq 1 \\ 1 + \log^*(\log n) & \text{otherwise} \end{cases}$$

Thus $\log^* 2 = 1$, $\log^* 2^2 = 2$. Similarly, $\log^* 2^{2^2} = 1 + \log^* 2^2 = 2 + \log^* 2 = 3$. It will also be useful to look at the inverse function of $\log^*$, *Tower*.

Definition 7.4.2: *Tower*(k)

$Tower(k) = 2^{Tower(k-1)}$ and $Tower(0) = 1$.

Observation 7.1. $\log^*(\text{Tower}(k)) = k$

Let $Block(i)$ denotes a subset of vertices, and it corresponds to the interval $[Tower(i-1)+1, Tower(i)]$. We correspond $Block(0)$ with the interval $[0, 1]$. Then we say a node $v \in Block(i)$ if $v.rank \in [Tower(i-1)+1, Tower(i)]$.

Observation 7.2. *The largest k such that for all $t > k$,* $\text{Block}(t) = \emptyset$ *will be* $k = \log^* n$.

This is because

$$Block(k) = \left\{ v \mid v.rank \in \left[Tower(\log^* n - 1) + 1, Tower(\log^* n) \right] \right\} = \left\{ v \mid v.rank \in \left[\log n + 1, n \right] \right\}$$

Hence the number of blocks available is $O(\log^* n)$.

Now we will analyze the FIND(v) operation. Let P be the path visited. Let $P = v = v_0 \to v_1 \to \cdots \to v_k = \text{root}$. Now consider the ranks of the vertices in P. We will have

$$v_0.rank < v_1.rank < \cdots < v_k.rank$$

Imagine partitioning P into which blocks each element rank belongs to. Let INDEX$_B(v)$ is the index of the block that contains v. Since ranks of the vertices are strictly increasing INDEX$_B(v)$ is also increasing.

During FIND operation let going from one block to another is called jumping between blocks and going from one vertex to another inside a block is called internal jump. We call a vertex small if the vertex and its parent are in same block. Otherwise, we call the vertex large.

Lemma 7.4.7

During a single FIND(x) operation, the number of jumps between blocks along the search path is $O(\log^* n)$.

Proof: During FIND operation since the ranks are strictly increasing once we pass through from a node in i^{th} block to a node in $(i+1)^{th}$ block we can never go back to the i^{th} block. So in a FIND operation we can do jumps between blocks at most $O(\log^* n)$ times. ■

Therefore, there can be at most $O(\log^* n)$ many large vertices encountered in any FIND operation.

Lemma 7.4.8

At most $|Block(i)| \leq Tower(i)$ many FIND operations can pass through an element x which is in the i^{th} block (i.e. INDEX$_B(x) = i$) before $x.parent$ is no longer in the i^{th} block. That is INDEX$_B(x.parent) > i$.

Proof: Now consider the case that v and $v.parent$ are both in the same block. Let $v.parent$ is not root. Now we perform a FIND operation that passes through x. Let r_{bef} is the $v.parent.rank$ before the FIND operation and r_{aft} is the $v.parent.rank$ after the FIND operation. By path compression we have $r_{aft} > r_{bef}$.

By the above discussion we have that the parent of a vertex v increases its rank every-time an internal jump goes through v. Since there are at most $|Block(i)|$ different values in i^{th} block and by definition we have $|Block(i)| \leq Tower(i)$ we have the lemma. ■

Hence any vertex v appears small at most $Tower(i)$ times over the course of FIND operations that passes through v.

Lemma 7.4.9

There are at most $\frac{n}{Tower(i)}$ nodes that have ranks in the i^{th} block throughout the algorithm execution.

Proof: By Corollary 7.4.5 we have that the number of elements with rank in the i^{th} block is at most

$$\sum_{k=Tower(i-1)+1}^{Tower(i)} \frac{n}{2^k} = n \sum_{k=Tower(i-1)+1}^{Tower(i)} \frac{1}{2^k} \leq \frac{n}{2^{Tower(i-1)}} = \frac{n}{Tower(i)}$$

■

Lemma 7.4.10

The number of internal jumps performed, inside the i^{th} block, during the lifetime of UNION-FIND data structure is $O(n)$.

Proof: A vertex v in the i^{th} block can have at most $|Block(i)|$ internal jumps, before all jumps through v are jumps between block by Lemma 7.4.5. There are at most $\frac{n}{Tower(i)}$ vertices with ranks in i^{th} block through out the algorithm execution by Lemma 7.4.9. Thus, the total number of internal jumps is $|Block(i)| \cdot \frac{n}{Tower(i)} \leq Tower(i) \cdot \frac{n}{Tower(i)} = n$. Hence there are at most $O(n)$ many internal jumps. ■

Theorem 7.4.11

The number of internal jumps performed by the Union-Find data structure overall $O(n \log^* n)$.

Proof: Since in each block the number of internal jumps is $O(n)$ and there are $O(\log^* n)$ many blocks we have total number of internal jumps by the Union-Find data structure is $O(n \log^* n)$. ■

Hence over the course of all Find operations the total number of small vertices can be encountered is $O(n \log^* n)$

Theorem 7.4.12

The overall time spent on m Find operations, throughout the lifetime of a Union-Find data structure defined over n elements is $O((n+m) \log^* n)$.

Proof: With m many Find operations the total number of small vertices encountered is $O(n \log^* n)$ as we proved earlier. And the total number of large vertices encountered is $O(m \log^* n)$ since each Find can encounter $O(\log^* n)$ many large vertices by Lemma 7.4.7. Hence total taken is $O((m+n) \log^* n)$. ■

Chapter 8

Red Black Tree Data Structure

A red-black tree is a special type of binary search tree with one extra bit of storage per node, its color which can be either red or black. Also, we keep the tree approximately balanced by enforcing some properties on the tree.

Definition 8.1: Perfect Binary Tree

It is a Binary Tree in which every internal node has exactly two children and all leaves are at the same level.

Lemma 8.1

Every perfect binary tree with k leaves has $2k - 1$ nodes (i.e. $k - 1$ internal nodes).

Definition 8.2: Red Black Tree

A red-black tree is a binary tree with the following properties:

- Every internal node is key/NIL node. Every leaf is a "NIL" node.
- Each node (NIL and key) is colored either red or black.
- Root and NIL nodes are always black.
- Any child of a red node is black.
- The path from root to any leaf has the same number of black nodes.

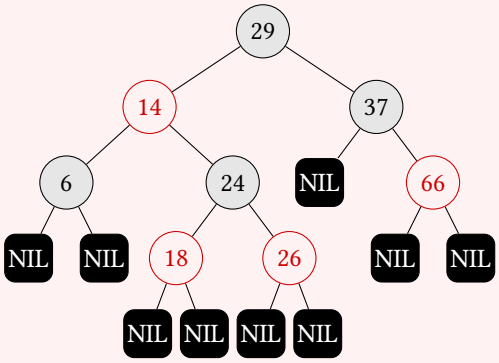


Figure 8.1: A Red Black Tree

We call the number of black nodes on any simple path from but not including a node x down to a leaf the *black-height* of the node, denoted by $bh(x)$. We generally confine our interest to the internal nodes of a red-black tree, since they hold the key values.

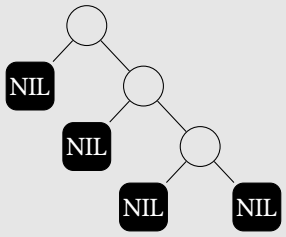
Lemma 8.2

A Red-Black Tree with n internal nodes or key nodes has height at most $O(\log n)$.

Proof: We will first show that for any subtree rooted at node x contains at least $2^{bh(x)} - 1$ internal nodes. We will show this using induction on the height of the tree. For the base case let height of x is 1. Then x must be a leaf. Therefore, the subtree rooted at x has at least $bh(x) = 1$. Hence, $2^{bh(x)} - 1 = 2^1 - 1 = 1$ nodes which is true. For inductive step let x has height greater than 1, and it is an internal node of the R-B Tree. Now x has two children. Hence, each child has black-height either $bh(x)$ or $bh(x) - 1$. By inductive hypothesis, the subtrees rooted at the children of x have at least $2^{bh(x)-1} - 1$ internal nodes. Thus, subtree rooted at x has at least $2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 1 = 2^{bh(x)} - 1$ internal nodes.

Now if the R-B tree has height h. Then any path from the root to a leaf at least half the nodes including the root must be black. So $bh(root) \geq \frac{h}{2}$. Thus, $n \geq 2^{\frac{h}{2}} - 1 \implies h \leq 2\log(n+1)$. Hence, we have the lemma. ■

Note:-



Not all trees can be colored in a way that satisfies the properties of a red-black tree. Consider the following tree:

In this example the root has to be black. The other two internal nodes can not be black since otherwise the path from the leaf of the root to root has only 2 black nodes but in the path from bottom most leaf to root will have 3. Then those two internal nodes has to be red. But that violates the property that a red node can not have a red child. Hence, this tree can not be colored in a way that satisfies the properties of a red-black tree.

Since by the lemma the R-B tree has height at most $O(\log n)$ and it is a binary search tree we can perform search of a node using FIND in $O(\log n)$ time. So now we will focus on the insertion and deletion operations in a red-black tree. To insert or delete a node in a red-black tree we will do rotations to balance the tree again. So first we will visit rotations.

8.1 Rotation

A rotation is a local operation that changes the structure of a binary tree without violating the binary search tree property. There are two types of rotations: left rotation and right rotation.

When we do a left rotation on a node we assume that its right child is not NIL. The left rotation "pivots" around the link from the node to its right child and makes the right child the new root of the subtree with the node as its left child. Similarly, we can explain the right rotation. The rotations behave like the following:

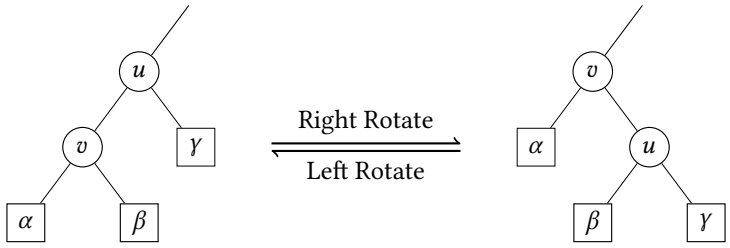


Figure 8.2: Left and Right rotate about $u - v$

Algorithm 35: LEFT-ROTATE(T, x)

```
1  y ⟵ x.right
2  x.right ⟵ y.left
3  if y.left ≠ NIL then
4      y.left.parent ⟵ x
5  y.parent ⟵ x.parent
6  if x.parent == NIL then
7      T.root ⟵ y
8  else if x == x.parent.left then
9      x.parent.left ⟵ y
10 else
11     x.parent.right ⟵ y
12 y.left ⟵ x
13 x.parent ⟵ y
```

Algorithm 36: RIGHT-ROTATE(T, x)

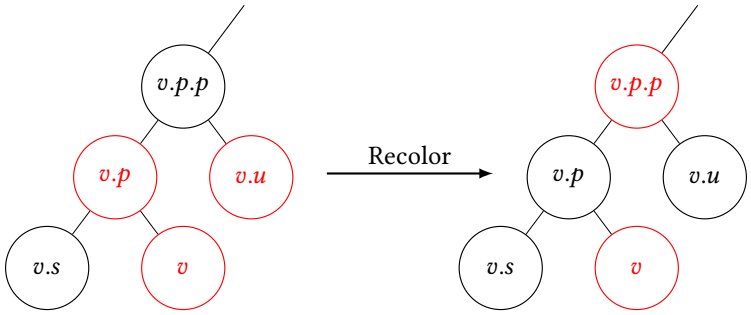
```
1  y ⟵ x.left
2  x.left ⟵ y.right
3  if y.right ≠ NIL then
4      y.right.parent ⟵ x
5  y.parent ⟵ x.parent
6  if x.parent == NIL then
7      T.root ⟵ y
8  else if x == x.parent.left then
9      x.parent.left ⟵ y
10 else
11     x.parent.right ⟵ y
12 y.right ⟵ x
13 x.parent ⟵ y
```

Both LEFT-ROTATE and RIGHT-ROTATE take $O(1)$ time. Only some constantly many pointers are changed by rotation all other attributes in a node remain the same.

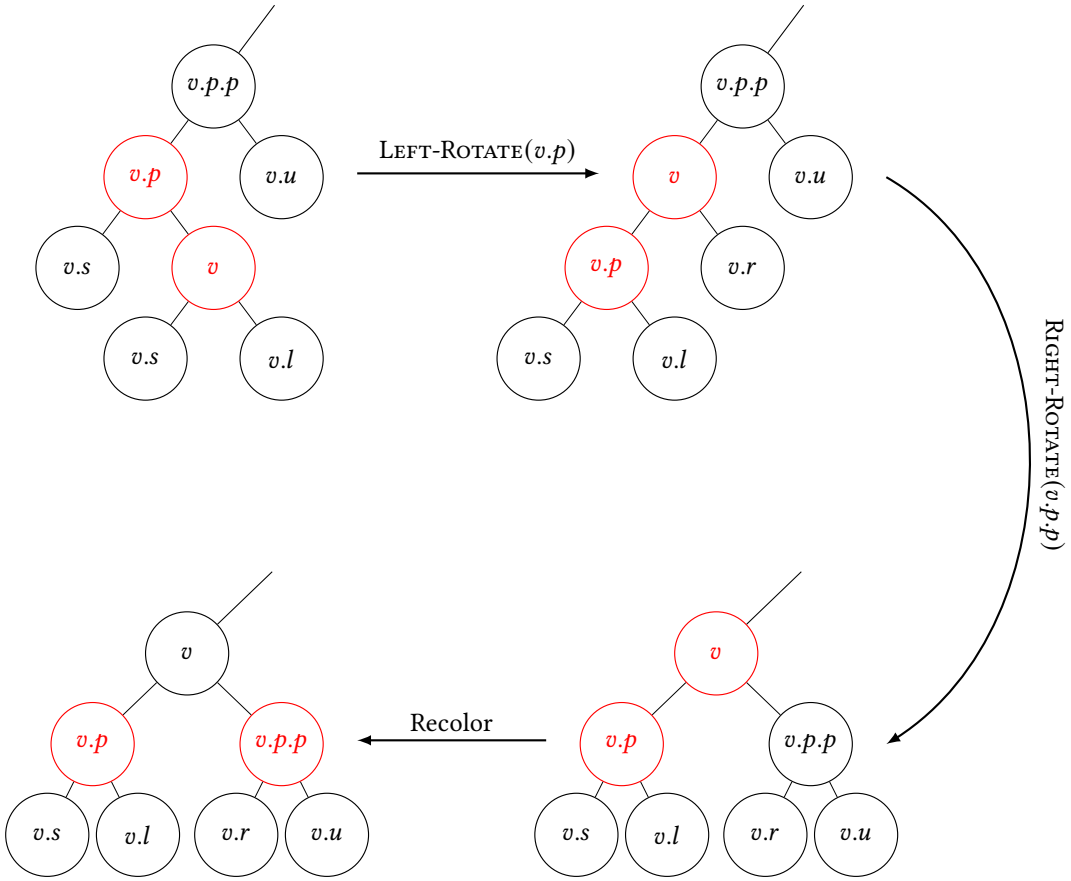
8.2 Insertion

We will now describe how to insert a node in a red-black tree in $O(\log n)$ time. We will insert the node in the tree in place of a leaf replacing a NIL node. After that we will color the node red. Let the node added is v. We define the attribute *uncle* which is basically sibling of the parent. Now two cases can happen:

Case I: $v.uncle.color$ = Red: Then $v.parent.parent$ is black. In this case we can recolor $v.parent.parent$ to red and both $v.parent$ and $v.uncle$ to be black. This will preserve the number of black nodes in any simple path from root to any leaf. Now the color of $v.parent.parent$ is red, and therefore we iterate the same process on $v.parent.parent$.



Case II: $v.uncle.color$ = Black: In this case we need two rotations. First we do a left rotation on $v.parent$. After that we do a right rotation on $v.parent.parent$. After the rotations, we recolor the nodes. The color of $v.parent.parent$



and the color of $v.parent$ will be red. The color of v will be recolored black. This will preserve the number of black nodes in any simple path from root to any leaf. And this case now stabilizes the tree, and we can stop the process.

So analyzing the insertion process we can insert a node in a red-black tree and using the two cases we can recolor the nodes the balance the tree.

Algorithm 37: RB-INSERT(T, v)

```
1  y ⟵ NIL, x ⟵ T.root
2  while x ≠ NIL do
3      y ⟵ x
4      if v.key < x.key then
5          x ⟵ x.left
6      else
7          x ⟵ x.right
8  v.parent ⟵ y
9  if y == NIL then
10     T.root ⟵ v
11 else if v.key < y.key then
12     y.left ⟵ v
13 else
14     y.right ⟵ v
15 v.left ⟵ NIL, v.right ⟵ NIL, v.color ⟵ RED
16 RB-INSERT-FIXUP(T, v)
```

Algorithm 38: RB-INSERT-FIXUP(T, v)

```
1  while v.parent.color == RED do
2      if v.parent.parent == NIL then
3          v.parent.color ⟵ BLACK
4          Break
5      vu ⟵ v.parent.parent.right          // Uncle
6      vpp ⟵ v.parent.parent
7      if vu.color == RED then
8          v.parent.color ⟵ BLACK          // Case I
9          vu.color ⟵ BLACK
10         vpp.color ⟵ RED
11         v ⟵ vpp
12     else
13         LEFT-ROTATE(T, v.parent)        // Case II
14         RIGHT-ROTATE(T, vpp)
15         v.color ⟵ BLACK
16         vpp.color ⟵ RED
17         Break
```

Since the Case I can happen at most $O(\log n)$ times as each use of Case I increase the height by 2, the while loop can run at most $O(\log n)$ times. Therefore, insertion of a node in a red-black tree takes $O(\log n)$ time.

8.3 Deletion

Like insertion, deletion of a node involves recoloring and rotations to maintain the properties of a red-black tree. Here we will use a notion of double-black color. In the deletion we will use something called in-order traversal of the binary tree and use successor and predecessor of a node in the traversal.

Definition 8.3.1: In-Order Traversal

In-Order Traversal of a binary tree is a traversal where:

- Recursively traverse the current node left subtree.
- Visit the current node.
- Recursively traverse the current node right subtree.

The in-order *successor* (*predecessor*) of a node is the next (previous) node in the in-order traversal of the tree.

Observation 8.1. *For any node x the in-order successor of x is the leftmost node in the right subtree of x. Similarly, the in-order predecessor of x is the rightmost node in the left subtree of x.*

To delete a node x we will replace its *key* by the key of its in-order successor or predecessor (say y) and then delete y i.e. after replacing the key of x by the key of y, it will still have the color of $x.color$. We replace with in-order successor unless x has no right child. In that case we replace with in-order predecessor. If x has no children then we have $y = x$.

Note:-

y is either a non-NIL leaf or has exactly one child.

1. y has a child then child must be colored red since otherwise the NIL child of y and any NIL node in the subtree rooted at child of y will have different black-height. Therefore, y must be colored black. Hence, we replace y by its child and color it black.

2. y is a non-NIL leaf and its colored red. Then we can simply remove y from the tree.

So the only case remained to analyze is when y is a non-NIL leaf and colored black. Now the situation is complicated since removing y would create black-height imbalance in the tree.

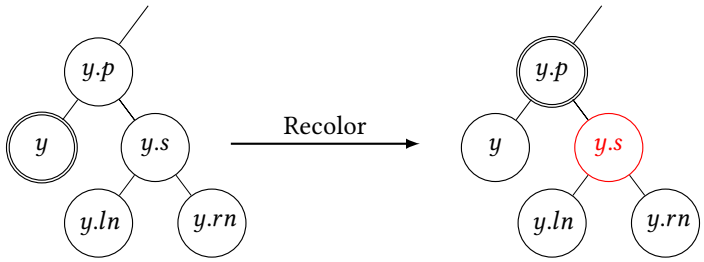
Observation 8.2. *If* y.color *is black then* y *must have a sibling since otherwise sibling of* y *is NIL. Then that NIL node and any NIL node in the subtree rooted at* y *will have different black-height.*

So we replace y with a NIL node and color it *double-black* which will be counted has 2 black nodes to maintain the black-height. Now we will resolve the double-black color by rotation, recoloring or pushing up the double-black color. We will use the following pointers

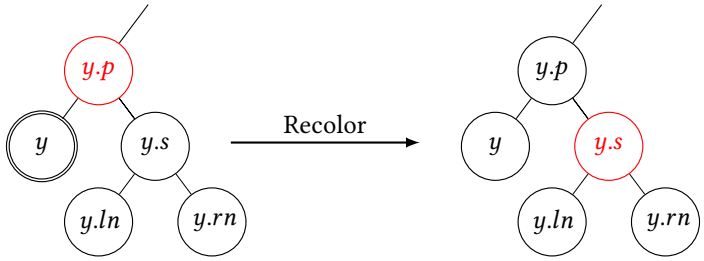
- *y.sibling* to denote the sibling of y.
- *y.left-nephew* and *y.right-nephew* to denote the left and right child of *y.sibling*.

We will use the following cases to resolve the double-black color:

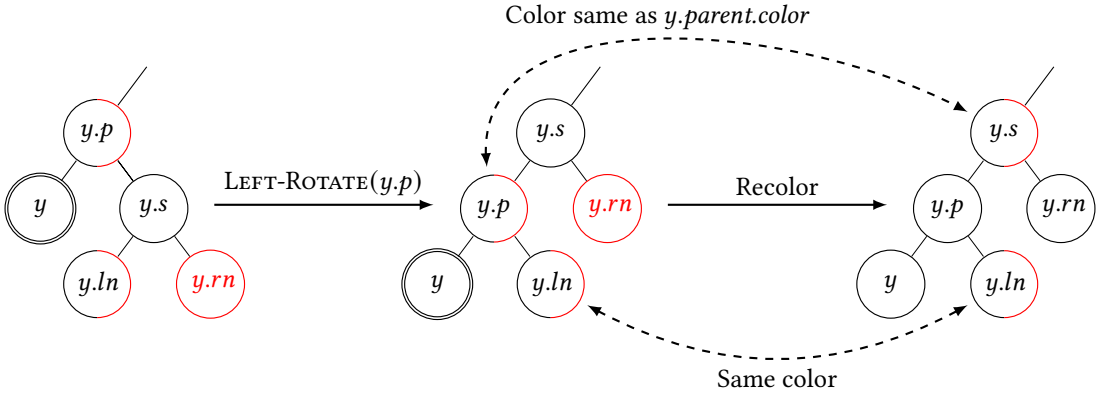
Case I: *y.sibling*, *y.parent*, *y.left-nephew*, *y.right-nephew* are all Black. In this case we can make *y.parent* the double black instead of y and recolor the y has black node and sibling of y red color.



Case II: *y.sibling*, *y.left-nephew*, *y.right-nephew* are Black & *y.parent* is Red. Here we recolor *y.parent* to black and *y.sibling* to red. This will preserve the number of black nodes in any path from root to any leaf. So we stop.

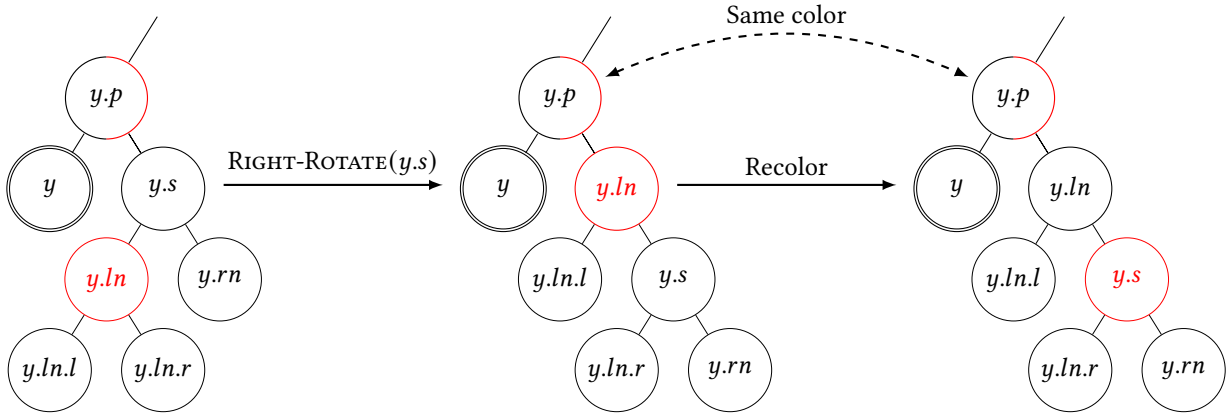


Case III: *y.sibling* is Black and *y.right-nephew* is Red. Then we do a LEFT-ROTATE on *y.parent*. Then we recolor *y.sibling*



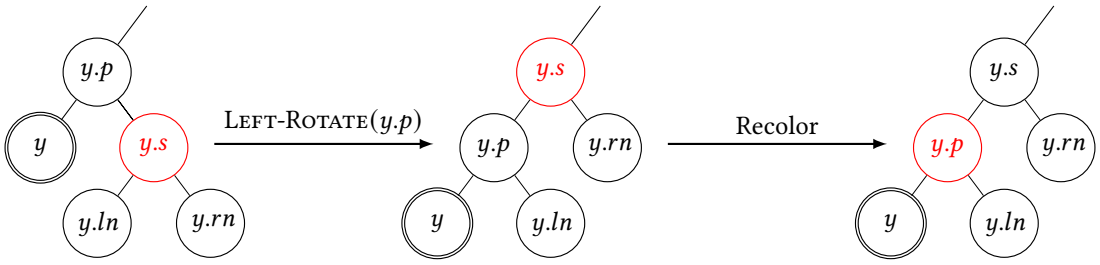
to the same color as *y.parent*. And we recolor *y.parent* to black, *y.right-nephew* to red and y to black. And now we stop.

Case IV: *y.sibling*, *y.right-nephew* are Black & *y.left-nephew* is Red. Therefore, both the children of *y.left-nephew* have color black. Here we first do a RIGHT-ROTATE on *y.sibling*. Then we recolor *y.left-nephew* to black and *y.sibling*



to red. Now we have exactly the same situation as in Case III with respect to node y after recoloring. So we follow the steps of Case III.

Case V: *y.sibling* = Red. In this case *y.left-nephew* and *y.right-nephew* must be black. Since *y.sibling* is Red, *y.parent* is Black. Then we do a LEFT-ROTATE on *y.parent*. Then we switch the colors of *y.parent* and *y.sibling* i.e. we color



y.parent to red and *y.sibling* to black. Now we have the sibling of y has color black. So we are now in one of the Case I-IV. So we can follow the suitable case to resolve.

This completes the description of the deletion process in a red-black tree. Now notice every time we are pushing the double-black color up the tree, or we are stopping. Hence, it only takes $O(\log n)$ time to resolve the double-black color. So the deletion process in a red-black tree takes $O(\log n)$ time.

Chapter 9

Maximum Flow

9.1 Flow

Suppose we are given a directed graph $G = (V, E)$ with a source vertex s and a target vertex t. And additionally for every edge $e \in E$ we are given a number $c_e \in \mathbb{Z}_0$ which is called the capacity of the edge.

Definition 9.1.1: Flow

An $s - t$ flow is a function $f : E \to \mathbb{R}_0$ which satisfies the following:

① $\forall\, e \in E, f(e) \leq c_e$

② $\forall\, v \in V \setminus \{s, t\}, \sum_{e \in in(v)} f(e) = \sum_{e \in out(v)} f(e)$

Also the value of a flow f is denoted by $|f| := \sum_{e \in out(s)} f(e)$.

Before proceeding into the setup and the problem first we will assume some things

Assumption.

- $in(s) = \emptyset$ *i.e. there is no edge into* s.
- $out(t) = \emptyset$ *i.e. there is no edge out of* t.
- *There are no parallel edges*

Lemma 9.1.1

For any flow f, $|f| = \sum_{e \in in(t)} f(e)$

Proof: We have for every edge $e \in E$, $\exists\, v \in V$ such that $e \in in(v)$ and $\exists\, u \in V$ such that $e \in out(u)$. Hence we get

$$\sum_{e \in E} f(e) = \sum_{v \in V} \sum_{e \in in(v)} f(e) = \sum_{v \in V} \sum_{e \in out(v)} f(e) \implies \sum_{v \in V} \left[\sum_{e \in in(v)} f(e) - \sum_{e \in out(v)} f(e) \right] = 0$$

Now we know $\forall\, v \in V \setminus \{s, t\}$. $\sum_{e \in in(v)} f(e) = \sum_{e \in out(v)} f(e)$. Therefore we get

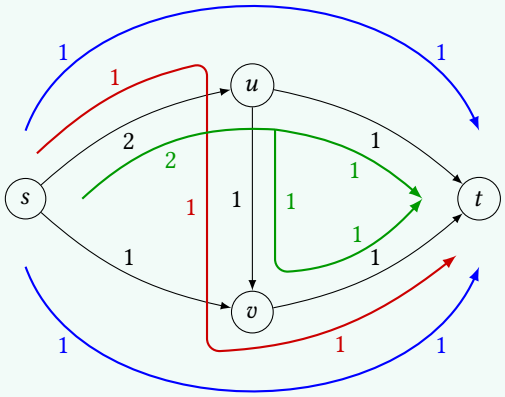
$$\sum_{v \in V} \left[\sum_{e \in in(v)} f(e) - \sum_{e \in out(v)} f(e) \right] = 0 \implies \sum_{v \in \{s,t\}} \left[\sum_{e \in in(v)} f(e) - \sum_{e \in out(v)} f(e) \right] = 0 \implies \sum_{e \in out(s)} f(e) - \sum_{e \in in(t)} f(e)$$

Hence we have $|f| = \sum_{e \in in(t)} f(e)$. ∎

Max Flow
Input: A directed graph $G = (V, E)$ with source vertex s and target vertex t and for all edge $e \in E$ capacity of the edge $c_e \in \mathbb{Z}_0$
Question: Given such a graph and its capacities find an $s - t$ flow which has the maximum value

Example 9.1.1

Consider the following directed graph with capacities: $V = \{s, t, u, v\}$, $c_{s,u} = 2, c_{s,v} = c_{u,t} = c_{v,t} = c_{u,v} = 1$. Firstly the following function: $f' : f'(s,u) = 2 = f(u,t)$. It is not a flow since $f(u,t) = 2 > 1 = c_{u,t}$. Now we define three different flow functions:



- f: $f(s,u) = f(u,v) = f(v,t) = 1$ and otherwise 0. Therefore $|f| = 1$
- g: $g(s,u) = g(u,t) = 1$, $g(s,v) = g(v,t) = 1$ and otherwise 0. Therefore $|g| = 2$
- h: $h(s,u) = 2$, $h(u,t) = h(u,v) = h(v,t) = 1$ and otherwise 0. Therefore $|h| = 2$

Notice here g and h has the maximum flow value.

9.2 Ford-Fulkerson Algorithm

Definition 9.2.1: Residual Graph

Given a directed graph $G = (V, E)$ and capacities C_e for all $e \in E$ and an $s - t$ flow f the residual graph $G_f = (V, E_f)$ has the edges with the following properties:

① If $(u,v) \in E$ and $f(u,v) > 0$ then $(v,u) \in E_f$ and $c^f_{v,u} = f(u,v)$. Such an edge is called a *backward* edge.

② If $(u,v) \in E$ and $f(u,v) < c_{u,v}$ then $(u,v) \in E_f$ and $c^f_{u,v} = c_{u,v} - f(u,v)$. It is called *forward* edge.

Algorithm 39: Ford-Fulkerson

Input: Directed graph $G = (V, E)$, source s, target t and edge capacities C_e for all $e \in E$
Output: Flow f with maximum value

```
1 begin
2     for e ∈ E do
3         f(e) = 0
4     while ∃ s ⇝ t path P in G_f do
5         δ ⟵ min_{e∈P}{c_e^f} for e = (u,v) ∈ P do
6             if e is Forward Edge then
7                 f(u,v) ⟵ f(u,v) + δ
8             else
9                 f(u,v) ⟵ f(v,u) − δ
```

We call one iteration of the While loop at line 4 *Flow Augmentation.*

Lemma 9.2.1

At any iteration the f' obtained after the flow augmentation of the flow f is a valid flow

Proof: At any iteration let P be the path from $s \rightsquigarrow t$ and $\delta = \min_{e \in P} c_f(e)$. Let f' be the new function such that for each $(u,v) \in P$ if (u,v) is forward edge in G_f then $f'(u,v) = f(u,v) + \delta$ and if (u,v) is backward edge in G_f then $f'(v,u) = f(v,u) - \delta$ and for other edges $e \in E \setminus P$, $f'(e) = f(e)$.

Now since $\delta = \min_{e \in P} c_f(e)$, $c_f(e) \geq \delta$ for all $e \in P$. Hence if (u,v) is backward edge then $(v,u) \in E$ and $c_f(u,v) = f(u,v)$. Hence $f'(v,u) = f(v,u) - \delta \geq 0$. Therefore for all $e \in E$, $f'(e) \geq 0$.

Now first we will show $f'(e) \leq c_e$ for all $e \in E$. If $(u,v) \in P$ is a forward edge then $(u,v) \in E$ and $c_f(u,v) = c_{u,v} f(u,v)$. Therefore $f'(u,v) = f(u,v) + \delta \leq f(u,v) + c_{u,v} - f(u,v) = c_{u,v}$. Now if $(u,v) \in P$ is a backward edge then $(v,u) \in E$ and $c_f(u,v) = f(u,v)$. Therefore $f'(u,v) = f(u,v) - \delta \leq f(u,v) \leq c_{u,v}$. For other edges $e \in E \setminus P$, $f'(e) = f(u) \leq c_e$. Therefore $f'(e) \leq c_e$ for all $e \in E$

Now we will prove for all $v \in V \setminus \{s,t\}$, $\sum_{e \in in(v)} f'(e) = \sum_{e \in out(v)} f'(e)$. If v is not in the path P in G_f then, $f'(e) = f(e)$ for all edges $e \in in(v) \cup out(v)$. Hence the condition is satisfied for such vertices. Suppose v is in the path P. Then there are two edges e_1 and e_2 in P which are incident on e. If both are forward edges or both are backward edges then one of them is in $in(v)$ and other one is in $out(v)$. WLOG suppose $e_1 \in in(v)$ and $e_2 \in out(v)$ we have

$$\sum_{e \in in(v)} f'(e) = \sum_{e \in in(v) \setminus \{e_1\}} f(e) + f(e_1) \pm \delta = \sum_{e \in out(v) \setminus \{e_2\}} f(e) + f(e_2) \pm \delta = \sum_{e \in out(v)} f'(e)$$

If one of e_1, e_2 forward edge and other one is backward edge then either $e_1, e_2 \in in(v)$ (when e_1 is forward and e_2 is backward) or $e_1, e_2 \in out(v)$ (when e_1 is backward and e_2 is forward). Now if $e_1, e_2 \in in(v)$, $f'(e_1) + f'(e_2) = f(e_1) + \delta + f(e_2) - \delta = f(e_1) + f(e_2)$ and if $e_1, e_2 \in out(v)$ then $f'(e_1) + f'(e_2) = f(e_1) - \delta + f'(e_2) + \delta = f(e_1) + f(e_2)$. Hence

$$\sum_{e \in in(v)} f'(e) = \sum_{e \in in(v)} f(e) = \sum_{e \in out(v)} f(e) = \sum_{e \in out(v)} f'(e)$$

Hence f' is a valid flow. ■

Lemma 9.2.2

At any iteration Given G_f if the flow, f' obtained after flow augmentation of f by δ then

$$|f'| = |f| + \delta$$

Proof: Since we augment flow along an $s \rightsquigarrow t$ path, the first edge of the path is always in $out(s)$. Let the first edge is $e = (s,u)$. Now e has to be a forward edge because otherwise $(u,s) \in E$ and then there is an incoming edge in G which is not possible. Hence

$$|f'| = \sum_{e \in out(s)} f'(e) = \sum_{e \in out(s) \setminus \{e\}} f(e) + f'(e) = \sum_{e \in out(s) \setminus \{e\}} f(e) + f(e) + \delta = \sum_{e \in out(s)} f(e) + \delta = |f| + \delta$$

Hence we have the lemma. ■

Lemma 9.2.3

At every iteration of the Ford-Fulkerson Algorithm the flow values and the residual capacities of the residual graph are non-negative integers.

Proof: Initial flow and the residual capacities are non-negative integers. Let till i^{th} iteration the flow values and the residual capacities were non-negative integers. Let the flow after i^{th} iteration was f. Hence $\forall\, e \in E, f(e) \in \mathbb{Z}_0$. Therefore in the G_f for all $e \in E_f$, $c_f(e) \in \mathbb{Z}_0$. Hence $\delta \in \mathbb{Z}_0$. Therefore $\forall\, e \in E, f'(e) \in \mathbb{Z}_0$. And therefore for all $e \in E_{f'}$ where $G_{f'}$ is the residual graph of the flow f', $c_{f'}(e) \in \mathbb{Z}_0$. Hence by mathematical induction the lemma follows. ■

At any iteration let P be the path from $s \rightsquigarrow t$. Then for all $e \in P$, $c_f(e) > 0$. Therefore $\delta = \min_{e \in P} c_f(e) \geq 1$. Therefore the algorithm must stop in at most $\sum_{e \in out(s)} c_e$ since we can have the value of a flow to be at max the value of the sum of capacities of edges in $out(s)$ and therefore we can increase the flow at max that many times.

Lemma 9.2.4

If f is a max flow then there is no $s \rightsquigarrow t$ path in G_f.

Proof: Suppose there is an $s \rightsquigarrow t$ path P in G_f. We will show that then f is not a max flow following the algorithm. Then $\forall\ e \in P$, $c_f(e) > 0$. Hence $\delta = \min_{e \in P} c_f(e) \geq 1$. Now after the flow augmentation process of f by δ we get a new valid flow f' by Lemma 9.2.1 and by Lemma 9.2.2 we have $|f'| = |f| + \delta > ||f|$. Hence f is not a maximum flow. Hence contradiction. Therefore there is no $s \rightsquigarrow t$ path in G_f. ■

9.2.1 Max Flow Min Cut

Definition 9.2.2: Cut Set

For a graph $G = (V, E)$ and a subset $A \subseteq V$, the cut $(A, V \setminus A)$ is a bipartition of V where the edges E_A of the graph $G_A = (A, V \setminus A, E_A)$ is the set $E_A = E \cap (A \times (V \setminus A))$.

Now if s, t are two vertices of G then an $s - t$ *Cut* $(A, V \setminus A)$ is a cut such that $s \in A$ and $t \in V \setminus A$.

Now we define for a cut $(A, V \setminus A)$ the *Capacity of the Cut* $(A, V \setminus A) = \sum_{e \in E_A} c_e$. For an $s - t$ cut $(A, V \setminus A)$ we denote the capacity of the cut by $cap(A)$ A *Min* $s - t$ *Cut* is a $s - t$ cut of minimum capacity. Then we have the following relation between cut and flow.

Lemma 9.2.5

Given a graph $G = (V, E)$, s, t, $c_e \in \mathbb{Z}_0$ for all $e \in E$ for any flow f and a $s - t$ cut $(A, V \setminus A)$

$$|f| \leq cap(A)$$

Proof: Given f and the $s - t$ cut $(A, V \setminus A)$ we have

$$\begin{aligned}
|f| &= \sum_{e \in out(s)} f(e) \\
&= \sum_{v \in A} \left[\sum_{e \in out(v)} f(e) - \sum_{e \in in(v)} f(e) \right] \\
&= \sum_{\substack{e=(u,v),\\ u \in A, v \notin A}} f(e) - \sum_{\substack{e=(u,v),\\ u \notin A, v \in A}} f(e) && \text{[Edges for both endpoints in } A \text{ are canceled out]} \\
&= \sum_{e \in out(A)} f(e) - \sum_{e \in in(A)} f(e) \\
&\leq \sum_{e \in out(A)} f(e) \leq \sum_{e \in out(A)} c_e = cap(A)
\end{aligned}$$

Hence we have the lemma. ■

Having this lemma we have have for any flow f and $s - t$ cut $(A, V \setminus A)$ we have

$$|f| \leq cap(A) \implies \max_f |f| \leq \min_{s-t \text{ cut } (A, V \setminus A)} cap(A)$$

So we have the following theorem that the value of maximum flow is equal to the capacity of minimum cut.

Theorem 9.2.6 Max Flow Min Cut

Given a graph $G = (V, E)$, $s, t, c_e \in \mathbb{Z}_0$ for all $e \in E$. Then the following are equivalent:

(1) f is a maximum flow.

(2) There is no $s \rightsquigarrow t$ path in G_f

(3) There exists an $s - t$ cut of capacity $|f|$

Proof:

(1) $\implies$ (2): This is by Lemma 9.2.4.

(2) $\implies$ (3): We are given a flow f such that there is no $s \rightsquigarrow t$ path in G_f. We will construct a $s - t$ cut which has the capacity $|f|$. Now take A to be all the vertices reachable from s in G_f. This is a valid $s - t$ cut since $s \in A$ and as there is no $s \rightsquigarrow t$ path in G_f, $t \notin A$. Now

$$|f| = \sum_{e \in out(A)} f(e) - \sum_{e \in in(A)} f(e)$$

Now $\forall\ e = (u, v) \in E$ where $u \in A$ and $v \notin A$ we have $c_{u,v} = f(u, v) \implies c_{u,v} - f(u, v) = 0$ since otherwise $c_{u,v} - f(u, v) \neq 0 \implies c_{u,v} > f(u, v) \implies (u, v) \in E_f$ and therefore v is reachable from s but $v \notin A$, contradiction. Therefore (u, v) is a backward edge and hence . Now $\forall\ e = (u, v) \in E$ where $u \notin A$ and $v \in A$ we have $f(u, v) = 0$ since otherwise $f(u, v) > 0 \implies (v, u) \in E_f$ and therefore u is reachable from s but $u \notin A$, contradiction. Hence we have

$$|f| = \sum_{e \in out(A)} f(e) - \sum_{e \in in(A)} f(e) = \sum_{e \in out(A)} c_e = cap(A)$$

(3) $\implies$ (1): Now by Lemma 9.2.5 we have for any flow f and $s - t$ cut

$$|f| \leq cap(A) \implies \max_f |f| \leq \min_{s-t\ cut\ (A, V \setminus A)} cap(A)$$

Now given f there exists an $s - t$ cut of capacity $|f|$. Hence f is a max flow.

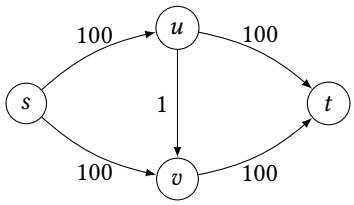
■

We will get another proof of the Max Flow Min Cut Theorem in subsection 14.4.4.

Hence at the end of the Ford-Fulkerson Algorithm let the flow returned by the algorithm is f. The algorithm terminates when there is no $s \rightsquigarrow t$ path in G_f. Hence by Max Flow Min Cut Theorem we have f is a maximum flow. This completes the analysis of the Ford-Fulkerson Algorithm.

Since the capacities of the edges can be very large we want an algorithm return the maximum flow with running time $poly(n, m, \log c_e)$ where n is the number of vertices and m is number of edges and $\log c_e$ basically means number of bits at most needed to represent the capacities.

But Ford-Fulkerson algorithm takes does not run in $poly(n, m, \log c_e)$ instead $poly(n, m, c_e)$ as the while loop in the algorithm takes $poly(c_e)$ many iterations. For example in the following graph: it takes around 100 steps



and in general Ford-Fulkerson takes $O(|f_{\max}|)$ time. For this reason we will now discuss a modification of the Ford-Fulkerson Algorithm which takes $poly(n, m, \log c_e)$ time, Edmonds-Karp Algorithm.

9.2.2 Edmonds-Karp Algorithm

To get a $poly(n, m, \log c_e)$ time algorithm we will always pick the shortest $s \rightsquigarrow t$ path in the residual graph. This algorithm is known as the Edmonds-Karp Algorithm

Suppose f_i be the total flow after i^{th} iteration. And G_{f_i} be the residual graph with respect f_i. Then $f_0(e) = 0$ for all $e \in E$ and $G_{f_0} = G$. Also suppose $dist_i(v)$ = Shortest $s \rightsquigarrow v$ path distance in the residual graph G_{f_i}. Hence $dist_i(s) = 0$ for all i and $dist_i(t) = \infty$ at the end of the algorithm.

Note:-

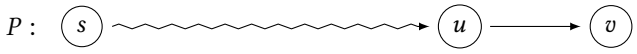
In i^{th} iteration of the Ford-Fulkerson Algorithm or Edmonds-Karp Algorithm if P is the $s \rightsquigarrow t$ in the residual graph G_{f_i} where $e = (u, v) \in P$ and $c_{f_i}(u, v) = \delta = \min_{e \in P} c_{f_i}(e)$ then the edge (u, v) is not present in the next residual graph $G_{f_{i+1}}$. Thus at least one edge disappears in each iteration of Ford-Fulkerson or Edmonds-Karp Algorithm.

Now we will prove following two lemmas which will help us to prove that the Edmond-Karp algorithm takes $O(mn)$ iterations.

Lemma 9.2.7

At any iteration i, $\forall\, v \in V$, $dist_i(v) \leq dist_{i+1}(v)$

Proof: Suppose this is not true. Then let i be the first iteration in which there exists a vertex $v \in V$ such that $dist_i(v) > dist_{i+1}(v)$. We pick such v which minimizes $dist_{i+1}(v)$. Consider the shortest path P from $s \rightsquigarrow v$ in $G_{f_{i+1}}$. Hence length of P, $|P| = dist_{i+1}(v)$. Let (u, v) be the last edge of P.



Then

$$dist_{i+1}(v) = dist_{i+1}(u) + 1 \geq dist_i(u) + 1$$

Here the last inequality follows because v is the vertex which has the minimum $dist_{i+1}(v)$ among all the vertices $w \in V$ which follows $dist_i(w) > dist_{i+1}(w)$. Now we will analyze case wise.

- **Case 1:** $(u, v) \in E_{f_i}$. Then
$$dist_i(v) \leq dist_i(u) + 1 \leq dist_{i+1}(v)$$
But this is not possible since $dist_i(v) > dist_{i+1}(v)$.
- **Case 2:** $(u, v) \notin E_{f_i}$. Then $(v, u) \in E_{f_i}$. Since $(u, v) \in E_{f_{i+1}}$ then we must have sent flow along (v, u). Since we take the shortest $s \rightsquigarrow t$ path in G_{f_i} in the algorithm we have $dist_i(u) = dist_i(v) + 1$. But then
$$dist_i(u) \leq dist_{i+1}(v) - 1 \implies dist_{i+1}(v) \geq dist_i(v) + 2$$
But this is not possible.

Hence contradiction ↯ Therefore for all iterations i, for all vertices $v \in V$, $dist_i(v) \leq dist_{i+1}(v)$. ■

Lemma 9.2.8

For any edge $e = (u, v) \in E$ the number of iterations where either (u, v) appears or (v, u) appears is at most $O(n)$ i.e.

$$\left|\left\{i \colon (u, v) \notin G_{f_i}, (u, v) \in G_{f_{i+1}}\right\}\right| + \left|\left\{i \colon (v, u) \notin G_{f_i}, (v, u) \in G_{f_{i+1}}\right\}\right| = O(n)$$

Proof: Following the proof of Lemma 9.2.7 in the second case we showed if $(u, v) \notin G_{f_i}$ but $(u, v) \in G_{f_{i+1}}$ then $dist_{i+1}(v) \geq dist_i(v) + 2$. Hence the distance increases by at least 2. Now this can happen at most $O(n)$ many times since $\forall\, i$, $dist_i(v) \leq n - 1$. Hence the number of iterations where either (u, v) appears or (v, u) appears is at most $O(n)$. ■

With this this lemma we will prove that the Edmonds-Karp Algorithm takes $O(mn)$ iterations.

Theorem 9.2.9

Edmonds-Karp Algorithm terminates in $O(mn)$ many iterations.

Proof: For k iterations at least k edges must disappear. Since each edge can reappear $O(n)$ times by Lemma 9.2.8, it can disappear at most $O(n)$ many times. In each iteration at least one edge disappears. Now after $O(mn)$ iterations number of disappearances is at most $O(mn)$. But after $O(mn)$ many disappearances there are no edge remaining and therefore there is no $s \rightsquigarrow t$ path. Hence the algorithm terminates. Therefore the Algorithm terminates in $O(mn)$ iterations. ■

Hence Edmond-Karp Algorithm takes $O(m^2n)poly(\log c_e) = O\left(m^2 n \log^{O(1)}(c_e)\right)$ time since it takes $O(mn)$ iterations and in each iteration it finds the shortest $s \rightsquigarrow t$ path in G_{f_i} in $O(m)$ time and in each iteration it does addition and subtraction and finds minimum of the capacities which takes polynomial of the bits needed to represent them time.

9.3 Preflow-Push/Push-Relabel Algorithm

In this algorithm we will maintain something called "Preflow" which is not a valid flow. Unlike Ford-Fulkerson, Edmonds-Karp it does not maintain a $s \rightsquigarrow t$ path in the residual graph and the algorithm stops when the preflow is actually a valid flow.

Definition 9.3.1: Preflow

Given a graph $G = (V, E)$ and the edge capacities c_e, a function $f : E \to \mathbb{R}_0$ is a preflow if is satisfies:

① $\forall\, e \in E, f(e) \leq c_e$.

② $\forall\, v \in V \setminus \{s\}, \sum\limits_{ein(v)} f(e) \geq \sum\limits_{e \in out(v)} f(e)$

Notice here unlike the definition of Flow here in the second criteria we need $\sum\limits_{ein(v)} f(e) \geq \sum\limits_{e \in out(v)} f(e)$ instead of $\sum\limits_{ein(v)} f(e) = \sum\limits_{e \in out(v)} f(e)$.

Now define for all $v \in V$ and for all preflow f, $excess_f(v) = \sum\limits_{e \in in(v)} f(e) - \sum\limits_{e \in out(v)} f(e)$. If f is a preflow then $excess_f(s) \leq 0$ and $\forall\, v \in V \setminus \{s\}$, $excess_f(v) \geq 0$

Lemma 9.3.1

For all preflow f

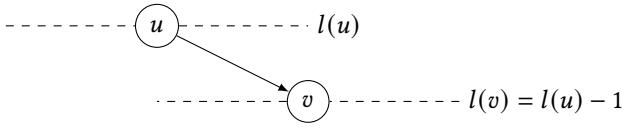
$$\sum_{v \in V} excess_f(v) = 0$$

Proof:

$$\begin{aligned}\sum_{v \in V} excess_f(v) &= \sum_{v \in V} \left[\sum_{e \in in(v)} f(e) - \sum_{e \in out(v)} f(e) \right] \\ &= \sum_{v \in V} \sum_{e \in in(v)} f(e) - \sum_{v \in V} \sum_{e \in out(v)} f(e) \\ &= \sum_{e \in E} f(e) - \sum_{e \in E} f(e) = 0\end{aligned}$$

■

Now for each $v \in V$ we assign a label $l(v) \in \mathbb{Z}_0$. The algorithm then sends flow from $u \to v$ if $l(v) = l(u) - 1$.



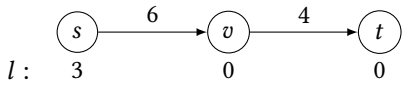
Algorithm 40: PREFLOW-PUSH

Input: Directed graph $G = (V, E)$, source s, target t and edge capacities C_e for all $e \in E$
Output: Flow f with maximum value

```
1  begin
2      Initially ∀ e = (s,u) ∈ E, f(e) = c_e and f(e) for all other edges.
3      l(s) ⟵ n
4      for v ∈ V \ {s} do
5          l(v) ⟵ 0
6      while ∃ v ≠ t, excess_f(v) > 0 do
7          if ∃ u, such that (v,u) ∈ E_f and l(u) = l(v) − 1 then
8              δ ⟵ min{excess_f(v), c_f(v,u)}
9              if (v,u) is Forward Edge then
10                 f(v,u) ⟵ f(v,u) + δ
11             else
12                 f(u,v) ⟵ f(u,v) − δ
13         else
14             l(v) ⟵ l(v) + 1                                  //Relabeling
```

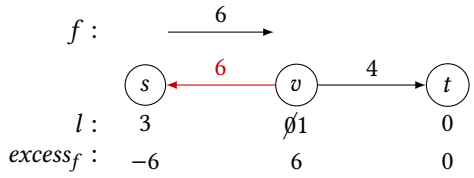
In the algorithm in line 8 if $\delta = c_f(v, u)$ then we call it *saturating push* and if $\delta = excess_f(v)$ then we call it *non-saturating push*.

Now we will show an example of how the algorithm on a graph. We will start the algorithm with the following graph:



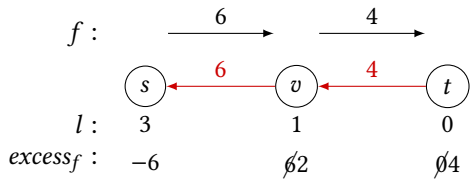
Below we will show change of the residual graph and preflow in each iteration of the WHILE loop:

- Step 1:



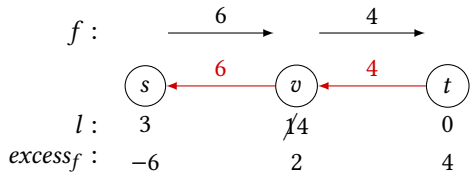
Since $excess_f(v) = 6 > 0$. So in first iteration v is taken. Since there is no edge (v, u) with $l(u) = l(v) - 1$, label of v got increased

- Step 2:



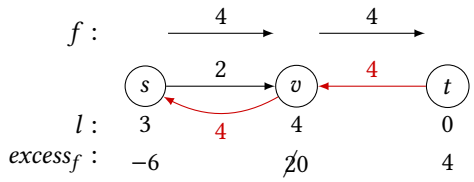
Since $excess_f f(v) = 2 > 0$, in second iteration again v is selected. There is an edge (v, t) with $l(t) = 0 = l(v) - 1 = 1 - 1$. Now $\delta = c_f(v, t) = 4$. Hence saturating push. The preflow gets updated, $f(s, v) = 6$, $f(v, t) = 4$.

- Step 5:



Since $excess_f f(v) = 2 > 0$, in next 3 iterations again v is selected. Since there is no edge (v, u) with $l(u) = l(v) - 1$, label of v gets increased every time. Which becomes 4 after 3 iterations.

- Step 6:



Since $excess_f f(v) = 2 > 0$, in this iteration again v is selected. There is an edge (v, s) with $l(t) = 3 = l(v) - 1 = 4 - 1$. Now $\delta = excess_f(v, s) = 2$. Hence it's non-saturating push. So the preflow gets updated $f(s, v) = 6 - 2 = 4$, $f(v, t) = 4$. Now it's a valid flow. Now there is no vertex with postive *excess*. Hence the algorithm stops.

Observation 9.1. *Labels are monotone non-decreasing.*

Observation 9.2. *For every iteration f is always a preflow. The proof is similar to Lemma 9.2.1 but use inequalities.*

Observation 9.3. $\sum_{v \in V} excess_f(v) = 0$ *and* $\forall\, v \in V \setminus \{s\}$, $excess_f(v) \geq 0$. *Hence* $excess_f(s) \leq 0 \implies l(s)$ *is unchanged.*

Now suppose f^i denote the preflow after the i^{th} iteration of the algorithm. Then

$$f^0(e) = \begin{cases} c_e & \text{when } e = (s, u) \\ 0 & \text{otherwise} \end{cases}$$

Now we will show the correctness of the algorithm.

Lemma 9.3.2

$\forall\, v \in V, \forall\, i, excess_{f^i}(v) > 0 \implies \exists\, v \rightsquigarrow s$ in G_{f^i}

Proof: First we fix v and i such that $excess_{f^i} > 0$. Let X be the set of vertices reachable from v in G_{f^i}. Now

$$\sum_{u \in X} excess_{f^i}(u) = \sum_{u \in X} \left[\sum_{e \in in(v)} f^i(e) - \sum_{e \in out(v)} f^i(e) \right] = \sum_{e \in in(X)} f^i(e) - \sum_{e \in out(X)} f^i(e)$$

Now if $\sum_{e \in in(X)} f^i(e) > 0$ then $\exists\ e = (u', u) \in E$ such that $u' \notin X$ and $u \in X$ and $f^i(e) > 0$. Then the backward edge $(u, u') \in E_{f^i}$. Then u' is reachable from v in G_{f^i}. But $u' \notin X$. Contradiction ↯ Therefore $\sum_{e \in in(X)} f^i(e) = 0$. Hence

$$\sum_{u \in X} excess_{f^i}(u) = \cancel{\sum_{e \in in(X)} f^i(e)} - \sum_{e \in out(X)} f^i(e) \leq 0$$

But from Observation 9.3 we have $\forall\ w \in V \setminus \{s\}$, $excess_{f^i}(w) \geq 0$. But at the same time $\sum_{u \in X} excess_{f^i}(u) \leq 0$ and $excess_{f^i}(v) > 0$. Hence $\exists$ a vertex $u \in X$ such that $excess_{f^i}(u) < 0$. But we know only vertex with negative *excess* is s. Therefore $s \in X$. Hence s is reachable from v. ∎

Lemma 9.3.3

$\forall\, i$, if $(u, v) \in G_{f^i}$ then $l(v) \geq l(u) - 1$.

Proof: We will prove this using induction on i. Initially $l(s) = n$ and $l(v) = 0$ for all $v \in V \setminus \{s\}$. Hence for all edges (u, v) where $u, v \neq s$ this is satisfied. All the other edges incident on s are in $in(s)$ in the residual graph. And $l(s) = n \geq l(u) = 0$. Therefore the base case is followed.

Now suppose the condition is true for f^{i-1}. Now in the i^{th} iteration suppose the selected vertex is $v \in V \setminus \{t\}$ with $excess_{f^{i-1}} > 0$. Now there are two possible cases.

- **Case 1:** If the step is relabeling then $f^{i-1} = f^i$, $G_{f^{i-1}} = G_{f^i}$ but v is relabeled by $l(v) + 1$. Now for any edge $e = (u, v) \in in(v)$ by Inductive Hypothesis $l(v) \geq l(u) - 1 \implies l(v) + 1 \geq l(u) - 1$. Now consider any edge $e = (v, w) \in out(v)$. By Inductive Hypothesis we have $l(w) \geq l(v) - 1$. Now if $l(w) = l(v) - 1$ then we would have pushed flow along the edge (v, w). Since that is not the case we have $l(w) > l(v) - 1$. Therefore $l(w) \geq (l(v) + 1) - 1$. Hence the condition is satisfied.

- **Case 2:** If the step is pushing flow then suppose we push flow along the edge $(v, w) \in E_{f^{i-1}}$ and $l(w) = l(v) - 1$. Now if we push flow along the edge (v, w) we might introduce the reverse edge (w, v) in G_{f^i}. In that case $l(v) = l(w) + 1 \geq l(w) - 1$. Hence the condition is satisfied.

Therefore by mathematical induction $\forall\ i, \forall\ (u, v) \in E_{f^i}, l(v) \geq l(u) - 1$. ■

Corollary 9.3.4

There is no $s \rightsquigarrow t$ path in G_{f^i} in any iteration i. Thus when the algorithm terminates f is a max flow.

Proof: Now $l(s) = n$ and $l(t) = 0$. We fix v and i. If there is a $s \rightsquigarrow v$ path in G_{f^i} then length of the path is at most $n - 1$. For each edge in the path the label decreases by at most 1 by Lemma 9.3.3. Hence $l(v) \geq 1$. Therefore for every vertex $v \in V$, reachable from s we have $l(v) \geq 1$. But $l(t) = 0$. Hence t is not reachable from s. Hence if the algorithm terminates, band if f is a valid flow then y Max Flow Min Cut Theorem it is a max flow. ■

Corollary 9.3.5

$\forall\ v \in V, \forall\ i, l(v) \leq 2n$.

Proof: Suppose $\exists\ v, i$ such that $l(v) = 2n$ and $excess_{f^i}(v) > 0$. By Lemma 9.3.2 there exists an $v \rightsquigarrow s$ path in G_{f^i}. Now by Lemma 9.3.3 for each edge in the path the label decreases by at most 1 and the length of the path is at most $n - 1$. Since $l(v) = 2n$, $l(s) \geq n + 1$. But we know $l(s)$ for all i by Observation 9.3. Hence contradiction ↯ Therefore for all $v \in V$ and $\forall\ i, l(v) \leq 2n$. ■

Corollary 9.3.6

Total number relabeling operations is $\leq 2n^2$

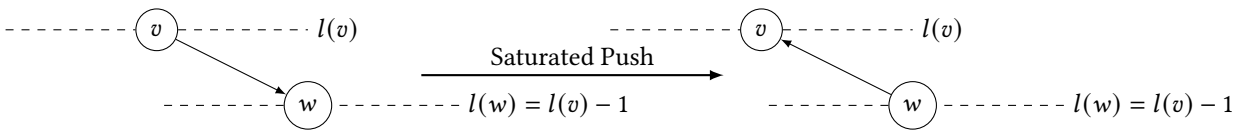
Proof: By Corollary 9.3.5 each vertex label can be at most $2n$. So total number of relabeling operations done in the algorithm is at most $2n^2$ ■

Now we need a bound on the number of push operations. We will count separately the number of Saturating Pushes and number of Non-Saturating Pushes.

Lemma 9.3.7

Total number of saturating pushes is $\leq 2mn$

Proof: We first fix an edge (v, w). Now we will count the number of saturating pushes along (v, w). Then $\delta = c_f(v, w)$. Now consider the scenario of two consecutive saturating pushes along (v, w). When the first saturating push along (v, w) occurred we have $l(w) = l(v) - 1$. Now if (v, w) is forward edge then $\delta = c_f(v, w) = c_{v,w} - f(v, w)$. Then new flow along (v, w) is $f(v, w) + c_f(v, w) = c_{v,w}$. Hence the edge (v, w) vanishes and the flow along (w, v) is $c_{v,w}$. If (v, w) is a backward edge then $\delta = c_f(w, v) = f(w, v)$. Hence then new flow along (w, v) is $f(w, v) - \delta = 0$. Hence again the (w, v) edge vanishes and the flow along (w, v) is $f(w, v)$.



Therefore after a saturated push along (v, w) the edge vanishes and the (w, v) edge is there. Hence in order for another push along (v, w) the algorithm must push flow along (w, v). And this happens when we have the new labels of

v, w follow the condition $l'(w) = l'(v) + 1$. Since by Observation 9.1 the labels never decreases in order for $l(w) = l(v) + 1$ the label of v must increase by at least 2.

Now starting from $l(v) = 0$ we have by Lemma 9.3.5 $l(v) \leq 2n$ and for each saturating push along (v, w) the $l(v)$ increase by 2. Hence at most n many saturating pushes occurred along (v, w). Now in the original graph since there are m edges the total number of saturating pushes is $\leq 2mn$. ■

Now we will count the number of non-saturating pushes. For such pushes along any edge (v, u) the $excess_f(v)$ goes to 0. We define the potential function for a preflow f,

$$\Phi(f) = \sum_{v:\ excess_f(v)>0} l(v)$$

Now $\Phi(f) \geq 0$ for all preflow f and initially at the start of the algorithm $\Phi(f^0) = 0$.

Lemma 9.3.8

For each non-saturating push $\Phi(f)$ decreases by at least 1.

Proof: Suppose at any iteration i a non-saturating push occur along an edge (v, w). Therefore $l(w) = l(v) - 1$. We will show that $\Phi(f^i) \leq \Phi(f^{i-1}) - 1$. We have $\delta = excess_{f^{i-1}}(v)$. Now if (v, w) is a forward edge then new flow along (v, w) is $f^i(v, w) = f^{i-1}(v, w) + excess_{f^{i-1}}(v)$. Since $(v, w) \in out(v)$

$$excess_{f^i}(v) = \sum_{e\in in(v)} f^i(e) - \sum_{e\in out(v)} f^i(e) = \sum_{e\in in(v)} f^{i-1}(e) - \sum_{e\in out(v)\setminus\{(v,w)\}} f^{i-1}(e) - f^i(v, w) = excess_{f^{i-1}}(v) - \delta = 0$$

Otherwise if (v, w) is a backward edge. Then ew flow along (w, v) is $f^i(w, v) = f^{i-1}(w, v) - excess_{f^{i-1}}(v)$. Since $(w, v) \in in(v)$

$$excess_{f^i}(v) = \sum_{e\in in(v)} f^i(e) - \sum_{e\in out(v)} f^i(e) = f^i(w, v) + \sum_{e\in in(v)\setminus\{(w,v)\}} f^{i-1}(e) - \sum_{e\in out(v)} f^{i-1}(e) = -\delta + excess_{f^{i-1}}(v) = 0$$

In both cases $excess_{f^i}(v) = 0$. Therefore v goes out of the summation. Now there are two cases depending on the value of $excess_{f^{i-1}}(w)$

- **Case 1:** If $excess_{f^{i-1}}(w) > 0$ i.e. w had excess flow before push operation then $\Phi(f^{i-1})$ decreases by $l(v)$ i.e. $\Phi(f^i) = \Phi(f^{i-1}) - l(v)$. Since $l(w) = l(v) - 1$ and by Observation 9.1 $l(v) \geq 1$. Therefore $\Phi(f^i) = \Phi(f^{i-1}) - l(v) \leq \Phi(f^{i-1}) - 1$.
- **Case 2:** If $excess_{f^{i-1}}(w) = 0$, then $excess_{f^i}(w) = excess_{f^{i-1}}(w) + \delta > 0$ since $\delta = excess_{f^{i-1}}(v) > 0$ and therefore $\Phi(f^i) = \Phi(f^{i-1}) - l(v) + l(w) = \Phi(f^{i-1}) - 1$

Hence for both the cases $\Phi(f^i) \leq \Phi(f^{i-1}) - 1$. Therefore $\Phi(f^{i-1})$ decreases by at least 1. ■

Observation 9.4. *For relabeling operation $\Phi(f)$ increases by* 1.

Since there are at most $2n^2$ relabeling operations by Corollary 9.3.6, $\Phi(f)$ increases by at most $2n^2$ with relabeling operations.

Observation 9.5. *For each saturating push $excess_f(v, w)$ might not go to 0 and therefore Φ might increase.*

Now by Lemma 9.3.7 total number of saturated pushes is at most $2mn$. And by Corollary 9.3.5 each vertex has label at most $2n$. Hence in total $\Phi(f)$ can increase at most $2mn \times 2n = 4mn^2$ by saturated pushes. Hence $\Phi(f)$ increases at most $2n^2 + 2mn \times 2m = O(mn^2)$.

Now

$$\#\text{Non-saturating Pushes} \leq \text{Total decrease in } \Phi \leq \text{Total increase in } \Phi \leq 2n^2 + 4mn^2 = O(mn^2)$$

Therefore total number of iterations of the WHILE loop is #Relabeling+#Saturated Push+#Non-saturated Push= $2n^2 + 2mn + O(mn^2) = O(mn^2)$. There fore the algorithm takes $O(mn^2)$ iteration. In each iteration it takes $O(m + n)$ time. Therefore the runtime of the algorithm is $O(mn^2)O(n + m) = O(m^2n^2)$.

Chapter 10

Randomized Algorithm

Here we will study randomized algorithm for tow basic problems. Later we will discuss other randomized algorithms too in the next chapters. We will also try to derandomize an algorithm in the next chapter.

10.1 Estimated Binary Search Tree Height

In this section we will calculate the expected height of a tree obtained by constructing a binary tree by picking elements uniformly at random from a given array. For this we have the following simple Intersection Algorithm

Algorithm 41: Simple Intersection Algorithm

Input: Array A of n elements of $[n]$ in any order.
Output: Construct a binary tree from A

```
1 begin
2     S ⟵ A
3     T ⟵ ∅
4     while S ≠ ∅ do
5         u ⟵ Extract(S)
6         Insert each element at the appropriate leaf of T
7     return T
```

Question 10.1

What is the expected height of the tree obtained by this Simple Intersection Algorithm assuming sequence of keys is uniformly random permutation of $[n]$.

Suppose X_n be the random variable for the height of the tree obtained by the algorithm running on any permutation of $[n]$. Let R_n be the random variable for the root of the tree obtained by the algorithm. Now consider the random variable Y_n defined as $Y_n = 2^{X_n}$. Then if we know $R_n = i$ we have

$$X_n = 1 + \max\{\text{Height of left subtree, Height of right subtree}\} = 1 + \max\{X_{i-1} + X_{n-i}\} \implies Y_n = 2\max\{Y_{n-1}, Y_{n-i}\}$$

Now for the case of $n = 1$ $Y_1 = 1$ since there is only one element and for the convenience we define $Y_0 = 0$. Now consider the following indicator random variable $Z_{n,i}$ where

$$Z_{n,i} = \begin{cases} 1 & \text{if } i \text{ is first element} \\ 0 & \text{otherwise} \end{cases}$$

So basically $Z_{n,i} = \mathbb{1}\{R_n = i\}$. Now if i is the first element then i the root of the tree obtained by the algorithm. Therefore

we have

$$\begin{aligned} Y_n &= \sum_{i=1}^{n} Z_{n,i}\,(1+\max\{Y_{i-1}, Y_{n-i}\}) \\ &\leq 2\sum_{i=1}^{n} Z_{n,i}(Y_{i-1}+Y_{n-i}) && \text{[Using Lemma 10.1.1]} \end{aligned}$$

Lemma 10.1.1 Soft Max

For any $a, b \in \mathbb{R}$,

$$\max\{a,b\} \leq \log(2^a + 2^b)$$

Therefore we have

$$\begin{aligned} \mathbb{E}[Y_n] &\leq 2\sum_{i=1}^{n} \mathbb{E}\Big[Z_{n,i}(Y_{i-1}+Y_{n-i})\Big] \\ &= 2\sum_{i=1}^{n} \mathbb{E}[Z_{n,i}]\mathbb{E}[Y_{i-1}+Y_{n-i}] \\ &= \frac{2}{n}\sum_{i=1}^{n}(\mathbb{E}[Y_{i-1}]+\mathbb{E}[Y_{n-i}]) = \frac{4}{n}\sum_{i=0}^{n-1}\mathbb{E}[Y_i] \end{aligned}$$

Now to compute $\mathbb{E}[Y_n]$ we use the following lemma

Lemma 10.1.2

$\mathbb{E}[Y_n] \leq \frac{1}{4}\binom{n+3}{3}$

Proof: We will prove this using induction on n. The base case is true for $n = 0$. Suppose this is true for $0, \ldots, n-1$.

$$\mathbb{E}[Y_n] \leq \frac{4}{n}\sum_{i=0}^{n-1}\mathbb{E}[Y_i] \leq \frac{1}{n}\sum_{i=0}^{n-1}\binom{i+3}{3} = \frac{1}{n}\binom{n+3}{4} = \frac{1}{n}\frac{(n+3)!}{4!(n-1)!} = \frac{1}{4}\binom{n+3}{3}$$

Hence by mathematical induction this is true for all n. ■

Hence by the lemma we have $\mathbb{E}[Y_n] \leq \frac{1}{4}\binom{n+3}{3} = O(n^3)$. Now by Jensen Inequality we have

$$\mathbb{E}[Y_n] = \mathbb{E}[2^{X_n}] \geq 2^{\mathbb{E}[X_n]}$$

Therefore $\mathbb{E}[X_n] \leq O(\log n)$. Therefore the expected height of a binary search tree is $O(\log n)$.

10.2 Solving 2-SAT

In this section we will discuss a randomized algorithm for deciding if a n-variate 2-SAT boolean formula is satisfiable or not.

2-SAT
Input: 2-SAT formula φ consisting of n variables.
Question: Given n-variate 2-SAT boolean formula determine if φ is satisfiable.

Here we give a simple randomized algorithm for solving the 2-SAT problem:

Algorithm 42: 2-SAT Randomized Algorithm

Input: n variate 2-SAT formula φ
Output: Decide if φ is satisfiable or not

```
begin
    ∀ i ∈ [n], Set x_i = 0
    while ∃ clause C that is not satisfied do
        Let x_i and x_j be variables in C
        Pick from {x_i, x_j} with equal probability and flip the assignment for that variable.
    return x
```

Now if the algorithm terminates it terminates with a satisfying assignment. For now assume that φ is satisfiable. We will deal with the case that φ is not satisfiable later.

Now since there are n variables there can be at most $O(n^2)$ many clauses can be in the formula. Therefore for each step of the while loop to occur it can at most take $O(n^2)$ time to find a clause which is not satisfied.

Let S represents the set of satisfying assignments for φ. Let at j^{th} iteration let A_j denote the current assignment of the variables. Let X_j be the random variable which denotes maximum number of variables of A_j that matches with some satisfying assignment of S i.e.

$$X_j = \max\{n - |x - A_j| : x \in S\}$$

At any step if $X_j = n$ then the algorithm terminates since the algorithm has found a satisfying assignment. Now starting with $X_j < n$ we consider how X_j evolves over time and how long it takes before X_j reaches n.

Now at each step we pick a clause which is unsatisfied. So we know A_j and all assignments of S disagree on the value of at least one variable of this clause. If all the assignments in S disagree with A_j on both variables changing either one will increase X_j. If there are assignments in S which disagree on the value of one of the two variables then with probability $\frac{1}{2}$ we choose that variable and increase X_j by 1 and with probability $\frac{1}{2}$ we choose the other variable and decrease X_j by 1.

Therefore X_j behaves like a random walk on a line starting from 0 which denotes the worst possible case and ends once it reaches at n where at any nonzero point it goes up or down by 1 with probability $\frac{1}{2}$. This is a Markov Chain. We want to calculate how many steps does it take on average for X_j to stumble all the way up to n. Before that we first properly define our Markov Chain.

The Markov Chain consists states from 0 to n. Where from 0 it goes to 1 with probability 1 and from n it always stays at n. And for any other state i it goes to $i+1$ with probability $\frac{1}{2}$ and goes to $i-1$ with probability $\frac{1}{2}$. Now let

$$T(k) = \text{Expected time to walk from } k \text{ to } n$$

Then we have

$$T(n) = 0, \qquad T(0) = T(1) + 1, \qquad \forall\, i \in [n-1],\ T(i) = \frac{T(i-1)}{2} + \frac{T(i+1)}{2} + 1$$

Then we have n unknowns and n equations in the above system. Therefore on average at most $O(n^2)$ steps needed to find a solution.

Now at first we said we are assuming we are dealing with the case of there exists a solution.

Question 10.2

How to deal with the issue of no solution?

In this case we will run for more number of iterations before we give up since when we give up we me might just not have found the solution. So we will run the algorithm for $100n^2$ steps. And if no solution was found then we will give up.

We first of all divide the execution of the algorithm into segments of $2n^2$ steps each. We will calculate the failure case of each segment. If the 2-SAT formula has no solution then the algorithm gives correct output. Suppose it has a solution. Then by Markov's Inequality the probability of number of steps needed to find the solution is greater than the expected number of steps needed to find a solution is at most $\frac{1}{2}$. Now after total $100n^2$ steps the probability none of the segments found a solution is 2^{-50}.

Chapter 11

Derandomization

In this section we will see a derandomization technique called Conditional Expectation. With this technique we will show derandomization of some randomized algorithms in the following sections.

11.1 Conditional Expectation

Let $\mathscr{A}$ be a randomized algorithm which is successful with probability at least $\frac{2}{3}$. Suppose $\mathscr{A}$ uses m random bits and suppose the random bits are $R_1, \ldots, R_m$. Then we have

$$\mathop{\mathbb{P}}_{R_1,\ldots,R_m}[\mathscr{A}(x, R_1, \ldots, R_m) = \text{Correct}] \geq \frac{2}{3}$$

We want to derandomize $\mathscr{A}$.

Now think of $\mathscr{A}$ as a binary tree which, given x, branches on the sampled value of each random bit R_i where it goes to left child if the random bit takes value 0 and goes to right child if the random bit takes value 1. Every path in this tree from root to leaf corresponds to different possible random strings and the leaf nodes corresponds to the output of the algorithm with the corresponding random string. Since $\mathscr{A}$ succeeds with probability at least $\frac{2}{3}$ means that at least $\frac{2}{3}$ of the leaves are good outputs for the input x.

Idea. *To derandomize $\mathscr{A}$ we need to find a deterministic algorithm that traverses from the root to a leaf which at any branch at level i chooses a direction which leads to a good output.*

Now suppose $r_1, \ldots, r_m \in \{0, 1\}$ denote the values taken by the random variables $R_1, \ldots, R_m$. Now let $P(r_1, \ldots, r_i)$ denote the fraction of the leaves of the subtree below the node obtained by following the path $r_1, \ldots, r_i$. Formally,

$$P(r_1, \ldots, r_i) = \mathbb{P}[\mathscr{A}(x, R_1, \ldots, R_m) \mid R_1 = r_1, \ldots, R_i = r_i] = \frac{1}{2}P(r_1, \ldots, r_i, 0) + \frac{1}{2}P(r_1, \ldots, r_i, 1)$$

From the last equality it is clear that there is a choice r_{i+1} such that $P(r_1, \ldots, r_{i+1}) \geq P(r_1, \ldots, r_i)$. Therefore to find a good path in the tree it suffices at each branch to pick such an $r \in \{0, 1\}$. Then we would have

$$P(r_1, \ldots, r_m) \geq P(r_1, \ldots, r_{m-1}) \geq \cdots \geq P(r_1) \geq \mathbb{P}[\mathscr{A}(x, R_1, \ldots, R_m) = \text{Correct}] \geq \frac{2}{3}$$

Since $P(r_1, \ldots, r_m)$ is either 0 or 1 it must be 1.

11.2 Max-SAT

Max-SAT

Input: SAT formula φ with n variables and m clauses and non negative weights w_c on clauses.

Question: Given a SAT formula φ with n variables and m clauses and non negative weights w_c on clauses find an assignment that maximizes weight of satisfied clauses.

We will first show a randomized algorithm for this problem. Then we will use conditional expectation to derandomize the algorithm.

11.2.1 Randomized Algorithm

First lets see what is the expected weight of satisfied clauses. Let Y_c be the indicator random variable if clause C is satisfied. Suppose there are k variables in C. Then we have $\mathbb{E}[Y_c] = 1 - \frac{1}{2^k} \geq \frac{1}{2}$. Therefore expected weight of satisfied clauses is

$$\mathbb{E}\left[\sum_C w_c Y_c\right] = \sum_C w_c \mathbb{E}[Y_c] \geq \frac{1}{2}\sum_C w_c$$

Let OPT be the optimal Max-SAT solution for the given formula. Then we have $\sum_C w_c \geq \text{OPT}$. Therefore

$$\mathbb{E}\left[\sum_C w_c Y_c\right] \geq \frac{1}{2}\text{OPT}$$

Hence we have the following randomized algorithm:

Algorithm 43: 2-Approximate Max-SAT

Input: SAT formula φ with n variables and m clauses and non negative weights w_c on clauses.
Output: Find an assignment that maximizes weight of satisfied clauses.

```
1 begin
2     for i ∈ [n] do
3         x_i ⟵ Pick a value from {0,1} uniformly at random
4     return x
```

By the above discussion we have an assignment with an expected weight of satisfied clauses at least half the maximum.

11.2.2 Derandomization

Now we want to derandomize the algorithm using conditional expectation. Let $X_1, \ldots, X_n$ denote the random variable for each variables and $x_1, \ldots, x_n \in \{0,1\}$ denote the value the random variables took. A key step will be evaluate the conditional probabilities:

$$\mathbb{E}\left[\sum_C w_c Y_c \mid X_1 = x_1, \ldots, X_i = x_i\right] = \sum_C w_c \mathbb{P}[Y_c = 1 \mid X_1 = x_1, \ldots, X_i = x_i] \quad \forall\, i \in [n]$$

Hence we have to find the value of $\mathbb{P}[Y_c = 1 \mid X_1 = x_1, \ldots, X_i = x_i], \forall\, i \in [n]$. Now if the clause C is already satisfied by the setting $x_1, \ldots, x_i$ then $Y_C = 1$. Else if C has r variables from $x_{i+1}, \ldots, x_n$ then

$$\mathbb{P}[Y_c = 1 \mid X_1 = x_1, \ldots, X_i = x_i] = 1 - \frac{1}{2^r}$$

. Now if at height i, we find $\mathbb{E}\left[\sum_C w_c Y_c \mid X_1 = x_1, \ldots, X_i = 0\right]$ and $\mathbb{E}\left[\sum_C w_c Y_c \mid X_1 = x_1, \ldots, X_i = 1\right]$ and which ever gives the higher value we will set the assignment for X_i to be that one. Thus we can derandomize the algorithm.

11.3 Set Balancing

Set-Balance
Input: $A \in \{0,1\}^{n \times n}$ matrix with A_i is the i^{th} row of A and $A_{i,j}$ is the $(i,j)^{th}$ entry
Question: Given $n \times n$, 0-1 matrix A find $b \in \{1,-1\}^n$ to minimize $\|Ab\|_\infty = \max_{i \in [n]} |A_i b|$.

In the following sections we will not optimize on $\|Ab\|_\infty$. Instead we will give bound on how large min $\|Ab\|_\infty$ can be for any A.

Algorithm 44: SET-BALANCING

Input: $A \in \{0,1\}^{n \times n}$ matrix
Output: Find an $b \in \{1,-1\}^n$ to minimize $\|Ab\|_\infty$

```
1 begin
2     for i ∈ [n] do
3         x_i ⟵ Pick a value from {1, −1} uniformly at random
4     return x
```

11.3.1 Randomized Algorithm

Clearly for each row $i \in [n]$ we have

$$\mathbb{E}[A_i b] = \mathbb{E}\left[\sum_j A_{i,j} b_j\right] = \sum_j \mathbb{E}[A_{i,j} b_j] = 0$$

But that does not mean $\mathbb{E}[|A_i b|] = 0$. To get a bound on $\mathbb{E}[|A_i b|]$ we will use Hoeffding's Inequality

Theorem 11.3.1 Hoeffding's Inequality

Let $Y_1, \ldots, Y_n$ be independent random variables with bounded supposer $[l_i, u_i]$ for Y_i and let $Y = \sum_{i=1}^{n} Y_i$. Then for any $\delta > 0$

$$\mathbb{P}[|Y - \mathbb{E}[Y]| > \delta] \leq 2e^{-\frac{2\delta^2}{\sum_i (u_i - l_i)^2}}$$

In our case we have $Y_{i,j} = A_{i,j} b_j$ and $Y_i = \sum_j A_{i,j} b_j$. Then each $Y_{i,j} \in \{-1, 0, 1\}$, $\mathbb{E}[Y_{i,j}] = 0$ and $\mathbb{E}[Y_i] = 0$. Therefore

$$\mathbb{P}[|Y_i| > \delta] \leq 2e^{-\frac{2\delta^2}{4n}}$$

Now we choose $\delta = 2\sqrt{n \ln n}$

$$\mathbb{P}[|A_i b| \geq 2\sqrt{n \ln n}] \leq \frac{2}{n^2}$$

Therefore $\mathbb{P}[\|Ab\|_\infty \geq 2\sqrt{n \ln n}] \leq \frac{2}{n}$ by union bound. Hence choosing each entry b uniformly at random from ± 1 we can obtain $\|Ab\|_\infty \leq 2\sqrt{n \ln n}$ with high probability.

11.3.2 Derandomization

Again we will use conditional expectation to derandomize the algorithm. Let a node at height j corresponds to a setting of $b_1, \ldots, b_j$ and we will calculate $\mathbb{P}[\|Ab\|_\infty > 2\sqrt{n \ln n} \mid b_1, \ldots, b_j]$. Now consider a leaf corresponding to some choice of $b_1, \ldots, b_n$ such that the value of the leaf is < 1. But there is no randomness at the leaf. Then $\mathbb{P}[\|Ab\|_\infty > 2\sqrt{n \ln n} \mid b_1, \ldots, b_n] = 0$. Hence for this choice of $b_1, \ldots, b_n$ it must have $\|Ab\|_\infty \leq 2\sqrt{n \ln n}$. Now

$$\mathbb{P}[\|Ab\|_\infty > 2\sqrt{n \ln n} \mid b_1, \ldots, b_j] = \mathbb{P}[\|Ab\|_\infty > 2\sqrt{n \ln n} \mid b_1, \ldots, b_j, 0] + \mathbb{P}[\|Ab\|_\infty > 2\sqrt{n \ln n} \mid b_1, \ldots, b_j, 1]$$

One of them have

$$\mathbb{P}[\|Ab\|_\infty > 2\sqrt{n \ln n} \mid b_1, \ldots, b_j, b_{j+1}] \leq \mathbb{P}[\|Ab\|_\infty > 2\sqrt{n \ln n} \mid b_1, \ldots, b_j]$$

So we choose that one. Also note that at the root $\mathbb{P}[\|Ab\|_\infty > 2\sqrt{n \ln n}] < \frac{2}{n}$. Then for choosing such a path for the corresponding choice of b we will have $\|Ab\|_\infty \leq M = 2\sqrt{n \ln n}$. But this depends on being able to calculate $\mathbb{P}[\|Ab\|_\infty > M \mid b_1, \ldots, b_j]$ which we don't know how to do in polynomial time. Instead we will use pessimistic estimator which.

11.3.3 Using Pessimistic Estimator to Derandomize

Instead of $\mathbb{P}[\|Ab\|_\infty > M \mid b_1, \ldots, b_j]$ we will use $\sum_{i\in[n]} \mathbb{P}[|A_i b| > M \mid b_1, \ldots, b_j]$. Naturally we have

$$\sum_{i\in[n]} \mathbb{P}[|A_i b| > M \mid b_1, \ldots, b_j] \geq \mathbb{P}[\|Ab\|_\infty > M \mid b_1, \ldots, b_j]$$

Now we know how to calculate $\mathbb{P}[|A_i b| > M \mid b_1, \ldots, b_j]$. For any $i \in [n]$ we have

$$\mathbb{P}[|A_i b| > M \mid b_1, \ldots, b_j] = \sum_{k=M+1}^{n} \mathbb{P}[A_i b = k \mid b_1, \ldots, b_j] + \mathbb{P}[A_i b = -k \mid b_1, \ldots, b_j]$$

Let $S_i = \{j' > j : A_{i,j'} = 1\}$ and $l = \sum_{j'\leq j} A_{i,j'}$. Then

$$\mathbb{P}[A_i b = k \mid b_1, \ldots, b_j] = \mathbb{P}\left[\sum_{j'\in S_i} b_{j'} = k - l\right]$$

Let in S_i n_i coordinates of b are 1 and rest of the coordinates of b in S_i are -1. Then

$$\sum_{j'\in S_i} b_{j'} = 2n_i - |S_i| = k - l \implies n_i = \frac{1}{2}(k - l + |S_i|)$$

Therefore we have

$$\mathbb{P}[A_i b = k \mid b_1, \ldots, b_j] = \frac{1}{2^{|S_i|}} \binom{|S_i|}{\frac{1}{2}(k - l + |S_i|)}$$

Thus we can calculate $\mathbb{P}[A_i b = k \mid b_1, \ldots, b_j]$ for all $n \geq |k| > M$. Therefore we can calculate $\mathbb{P}[|A_i b| > M \mid b_1, \ldots, b_j]$ and henceforth $\sum_{i\in[n]} \mathbb{P}[|A_i b| > M \mid b_1, \ldots, b_j]$. With this pessimistic estimator we calculate at height j both $\sum_{i\in[n]} \mathbb{P}[|A_i b| > M \mid b_1, \ldots, b_j, b_{j+1} = 0]$ and $\sum_{i\in[n]} \mathbb{P}[|A_i b| > M \mid b_1, \ldots, b_j, b_{j+1} = 1]$ and the one which have value less than 1 we will follow that path and eventually we will get an assignment of b for which $\|Ab\|_\infty \leq 2\sqrt{n \ln n}$.

Chapter 12

Global Min Cut

Global Min Cut
Input: Undirected graph $G = (V, E)$
Question: Find cut $(S, V \setminus S)$ that minimizes $|\delta(S)|$ where $\delta(S) = \{e = (u, v) \mid u \in S, v \notin S\}$.

12.1 Naive Algorithm

In previous chapter we have seen the algorithm to find $s - t$ min cut given any $s, t \in V$ in $O(n^2\sqrt{m})$ time. So naively we can run over all possible vertex pairs (s, t) and output the global min cut in $O(n^4\sqrt{m})$ time.

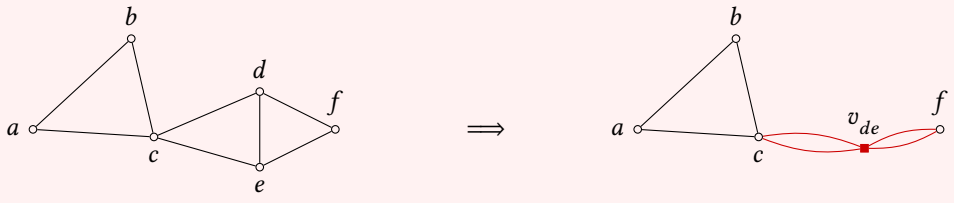
Or we can fix a vertex $s \in V$ and then for all $t \in V$ we can find the $s - t$ min cut and output the minimum. This takes $O(n^3\sqrt{m})$ time.

12.2 Karger's GMC Algorithm

Instead of naively solving the problem like above we will use randomization and will construct an algorithm which will output a global min-cut with high probability using edge contraction.

Definition 12.2.1: Edge Constraction

Given a graph $G = (V, E)$, $e = (u, v)$ edge contraction gives a multigraph (graph with multiple edges between two vertices but no self-loops) $G \setminus e = (V', E')$ where $V' = V \setminus \{u, v\} \cup \{v_e\}$ and for all $e' \in E$ if $e \cap e' = \emptyset$ then $e' \in E$ and otherwise $e' = (w, u)$ then $(w, v_e) \in E'$. The vertex v_e is called the supernode.



Observation. *For any edge $e \in G$:*

- *Any cut in $G \setminus e$ is also a cut in G of same size.*
- *Size of min cut in $G \setminus e$ is at least the size of min cut in G.*
- *Any cut in G that does not separate vertices of e is also cut in $G \setminus e$.*

Then we have the following lemma:

Lemma 12.2.1

Say k is the size of global min cut in $G' = (V', E')$ [G possible a multigraph] i.e. $\exists\ S \subseteq V'$ such that $|\delta(S)| = k$. Then $\min\{\deg(v) \mid v \in V'\} \geq k$ and $|E'| \geq \frac{k}{2}|V'|$.

Proof: If any vertex $v \in V'$ has degree less than k then we can take the cut $(\{v\}, V' \setminus \{v\})$ then $|\delta(v)| < k$, but that contradicts the fact that size of global min cut is k. Hence, contradiction ↯ Therefore $\forall\ v \in V'$, $\deg(v) \geq k$. Therefore, $|E'| = \frac{1}{2}\sum_{v \in V'} \deg(v) \geq \frac{k}{2} \cdot |V'|$. ■

So we at each round we will pick an edge from the graph uniformly at random and then contract that edge and in the next round we will pick an edge from the contracted graph. We will do $n-2$ such iterations since after that we are left with 2 supernodes $(X, V \setminus X)$.

Algorithm 45: Karger's GMC Algorithm

Input: Undirected graph $G = (V, E)$
Output: Find a cut $(S, V \setminus S)$ such that $|\delta(S)|$ is minimum

```
1 begin
2     H ⟵ G;
3     for i = 1, ..., n − 2 do
4         e ⟵Picked uniformly at random from E;
5         H ⟵ H \ e;
6     return E(H)
```

Question 12.1

What is the probability that the above algorithm returns a global min cut?

Let $(S, V \setminus S)$ is the global min cut with $|\delta(S)| = k$. Now probability that the algorithm returns $(S, V \setminus S)$ is equal to the probability that none of the edges in $\delta(S)$ is picked. So let $e_1, \ldots, e_{n-2}$ are the edges that are picked in the $n-2$ iterations of the algorithm. We need to calculate $\mathbb{P}[e_i \notin \delta(S),\ \forall\ i \in [n-2]]$

Lemma 12.2.2

$\mathbb{P}[e_1 \notin \delta(S)] \geq 1 - \frac{2}{n}$

Proof: We have $|\delta(S)| = k$. Hence, we have $|E| \geq \frac{n \cdot k}{2}$. Since e_1 is picked uniformly at random we have

$$\mathbb{P}[e_1 \notin \delta(S)] \geq 1 - \frac{k}{\frac{nk}{2}} = 1 - \frac{2}{n}$$

Hence we have the lemma. ■

Lemma 12.2.3

$\mathbb{P}[e_i \notin \delta(S) \mid e_1, \ldots, e_{i-1} \notin \delta(S)] \geq 1 - \frac{2}{n-i+1}$

Proof: Let $e_1, \ldots, e_{i-1} \notin \delta(S)$. Hence S is still a min cut in $G \setminus \{e_1, \ldots, e_{i-1}\}$. Then number of edges after contracting $e_1, \ldots, e_{i-1}$ is at least $\frac{k(n-i+1)}{2}$. Therefore

$$\mathbb{P}[e_i \notin \delta(S) \mid e_1, \ldots, e_{i-1} \notin \delta(S)]1 - \frac{k}{\frac{k(n-i+1)}{2}} = 1 - \frac{2}{n-i+1}$$

Therefore we have the lemma. ■

Hence we have

$$\begin{aligned}\mathbb{P}[\text{Success}] &\geq \mathbb{P}[e_i \notin \delta(S),\ \forall\ i \in [n-2]] \\ &= \prod_{i=1}^{n-2}\left(1 - \frac{2}{n-i+1}\right) \\ &= \frac{2}{n(n-1)} = \frac{1}{\binom{n}{2}} = O\left(\frac{1}{n^2}\right)\end{aligned}$$

So we run the above algorithm $2n^2 \log n$ times then take the cut which gives minimum size. Then we have

$$\begin{aligned}\mathbb{P}[\text{Succeeds}] &= 1 - \mathbb{P}[\text{All } 4n^2 \log n \text{ runs fails}] \\ &\geq 1 - \left(1 - \frac{2}{n^2}\right)^{4n^2 \log n} \\ &\geq 1 - \exp\left[-\frac{2}{n^2} 2n^2 \log n\right] \\ &= 1 - \frac{1}{n^4}\end{aligned}$$

Hence, this gives a much higher probability of success. So our final algorithm is

Algorithm 46: Multiple run of Karger's GMC Algorithm

Input: Undirected graph $G = (V, E)$
Output: Find a cut $(S, V \setminus S)$ such that $|\delta(S)|$ is minimum

```
1  begin
2      S ⟵ ∅;
3      cutEdgeSize ⟵ |E|;
4      for i ∈ [2n² log n] do
5          H ⟵ G;
6          for j = 1, ..., n − 2 do
7              e ⟵Picked uniformly at random from E;
8              H ⟵ H \ e;
9          if |E(H)| < cutEdgeSize then
10             Let H = (X, V \ X);
11             S ⟵ X;
12             cutEdgeSize ⟵ |E(H)|;
13     return S
```

12.3 Karger-Stein Algorithm

In Karger's algorithm the probability of getting a min cut is low because in later stages the probability of picking an edge from a min-cut is high because

$$\mathbb{P}[e_i \in \delta(S) \mid e_1, \ldots, e_{i-1} \notin \delta(S)] \leq \frac{2}{n-i+1} \implies \mathbb{P}[e_1, \ldots, e_i \notin \delta(S)] \geq \frac{\binom{n-i}{2}}{\binom{n}{2}}$$

If the above probability is at least $\frac{1}{2}$ then $2(n-i)^2 \geq n^2 \implies n - i \geq \frac{n}{\sqrt{2}}$. Hence, i can't be too high.

So instead of running the entire algorithm $\tilde{O}(n^2)$ times we can just run the later stages multiple times. So after $i \leq n - \frac{n}{\sqrt{2}} - 1$ iterations of Karger's GMC algorithm we have

$$\mathbb{P}[e_1, \ldots, e_i \notin \delta(S)] \geq \frac{(n-i)(n-i-1)}{n(n-1)} \geq \frac{n^2}{2n(n-1)} \geq \frac{1}{2}$$

from Lemma 12.2.3. We also have the following lemma:

Lemma 12.3.1

For any $1 \leq i < j \leq n-2$ we have

$$\mathbb{P}[e_i, e_{i+1}, \ldots, e_j \notin \delta(S) \mid e_1, \ldots, e_{i-1} \notin \delta(S)] \geq \frac{(n-j)(n-j-1)}{(n-i+1)(n-i)}$$

Now fix an $i \leq n-2$. Let $l = n-i+1$. Then For $j \leq n - \frac{l}{\sqrt{2}} - 1$ we have

$$\mathbb{P}[e_i, \ldots, e_{i+j-1} \notin \delta(S) \mid e_1, \ldots, e_{i-1} \notin \delta(S)] \geq \frac{l^2}{2l(l-1)} \geq \frac{1}{2}$$

So we have the following algorithm:

Algorithm 47: KS-Algorithm

Input: Undirected graph $G = (V, E)$
Output: Find a cut $(S, V \setminus S)$ such that $|\delta(S)|$ is minimum

```
1 begin
2     if |V| = 2 then
3         return Any vertex of V
4     Run Karger's GMC Algorithm on H for n − n/√2 − 1 iterations.;
5     Let H be the resulting multigraph.;
6     S_1 ⟵ KS-Algorithm(H);
7     S_2 ⟵ KS-Algorithm(H);
8     return arg min{|S_i| : i ∈ [2]}
```

Let $p(n)$ the probability of success for KS-Algorithm for a graph with n vertices. Then probability of not picking an edge until $\frac{n}{\sqrt{2}} + 1$ nodes remain is $\geq \frac{1}{2}$ as we have calculated above. Now the resulting graph has $\frac{n}{\sqrt{2}} + 1$ nodes. Hence, probability that KS-ALGORITHM(H) returns the min-cut is at least $\frac{1}{2}p\left(\frac{n}{\sqrt{2}} + 1\right)$. Therefore,

$$\mathbb{P}[\text{At least one of the run KS-ALGORITHM(H) returns the min cut}] \geq 1 - \left(1 - \frac{1}{2}p\left(\frac{n}{\sqrt{2}} + 1\right)\right)^2$$

Therefore we have

$$p(n) \geq 1 - \left(1 - \frac{1}{2}p\left(\frac{n}{\sqrt{2}} + 1\right)\right)^2$$

Solving this recursion relation we have $p(n) \geq \frac{1}{\log n}$. Hence, to succeed with high probability we need to run $2\log^2 n$ times.

Now For each run of the KS-Algorithm we have the recursion relation

$$T(n) \geq 2T\left(\frac{n}{\sqrt{2}} + 1\right) + O(n^2)$$

Solving the recursion relation we have $T(n) = O(n^2 \log n)$. Therefore, the time complexity of the total running time is $O(n^2 \log^3 n)$.

Chapter 13

Matching

In section 5.1 we saw how to find a maximal matching in a graph using matroids. Here we will try to find maximum matching.

Maximum Matching
Input: Graph $G = (V, E)$
Question: Find a maximum matching $M \subseteq E$ of G

First we will solve finding maximum matching in bipartite graphs first. Then we will extend the algorithm to general graphs.

13.1 Bipartite Matching

So in this section we will study the following problem:

Bipartite Maximum Matching
Input: Graph $G = (L \cup R, E)$
Question: Find a maximum matching $M \subseteq E$ of G

13.1.1 Using Max Flow

One approach to find a maximum matching is by using max-flow algorithm. For this we introduce 2 new vertices s and t where there is an edge from s to every vertex in L and there is an edge from every vertex in R to t and all edges have capacity 1. Then the max-flow for this directed graph is the maximum matching of the bipartite graph. So we have the algorithm:

Lemma 13.1.1

There exists a max-flow of value k in the modified graph $G' = (V, E')$ if and only there is a matching of size k

Proof: Suppose G' has a matching M of size k. Let $M = \{(u_i, v_i) : i \in [k]\}$ where $u_i \in L$ and $v_i \in R$ for all $i \in [k]$. Then we have the flow f, $f(s, u_i) = f(u_i, v_i) = f(v_i, t) = 1$ for all $i \in [k]$. This flow has value k.

Now suppose there is a flow f of value k. Since each edge has capacity 1 then either an edge has flow 1 or it has 0 flow. Since value of flow is k there are exactly k edges outgoing from s with positive flow. Let the edges are (s, u_i) for $i \in [k]$. Now from each u_i there is exactly one edge going out which has positive flow. Now if $\exists\ i \neq j \in [k]$ such that $\exists\ v \in R$, $f(u_i, v) = (u_j, v) = 1$ then $f(v, t) = 2$ but $c_{v,t} = 1$. So this is not possible. Therefore, the edges going out from each u_i goes to distinct vertices. These edges now form a matching of size k. ∎

Therefore, the algorithm successfully returns a maximum matching of the bipartite graph. But we don't know any algorithm for finding maximum matching in general graphs using max-flow. In the next algorithm we will use something called Augmenting paths to find a maximum matching which we will extend to general graphs.

Algorithm 48: BP-Max-Matching-Flow

Input: $G = (L \cup R, E)$ bipartite graph
Output: Find a maximum matching

begin
  $V \longleftarrow A \cup B \cup \{s, t\}$
  $E' \longleftarrow E$
  for $v \in L$ **do**
    $E' \longleftarrow E' \cup \{(s, v)\}$
  for $v \in R$ **do**
    $E' \longleftarrow E' \cup \{(v, t)\}$
  for $e \in E'$ **do**
    $c_e \longleftarrow 1$
  $f \longleftarrow$ Edmonds-Karp$(G' = (V, E'), \{c_e : e \in E'\})$
  return $\{e : f(e) > 0, e \in E\}$

13.1.2 Using Augmenting Paths

Definition 13.1.1: M-Alternating Path and Augmenting Path

In a graph $G = (V, E)$ and M be a matching in G. Then an M-alternating path is where the edges from M and $E \setminus M$ appear alternatively.

An M-alternating path between two unmatched (also called exposed) vertices is called an augmenting path.

Given a matching M and if there exists an augmenting path p then we can obtain a larger matching M' just by taking the edges in p not in M. Now suppose we are given a bipartite graph $G = (L \cup R, E)$. Let M is a matching in G. Suppose M is a maximum matching. If there exists an augmenting path p then we can obtain a larger matching just by taking the edges in p not in M. This contradicts with M is maximum matching. Hence, there are no augmenting paths.

Now we will show that given G and M which is not maximum then we can find an augmenting path with an algorithm. Since M is not maximum there is a vertex v which is not matched

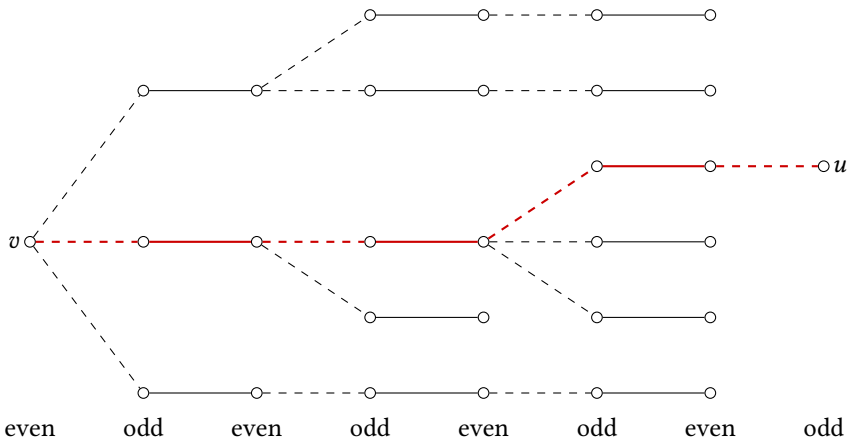
Algorithm 49: Find-Augmenting-Path(G, v)

Input: $G = (L \cup R, E)$ bipartite graph, matching M (not maximum) and an exposed vertex v
Output: Find an augmenting path starting from v

begin
  $v.mark \longleftarrow even$
  for $u \in L \cup R \setminus \{v\}$ **do**
    $u.mark \longleftarrow$ Null
  Queue Q // For BFS
  Enqueue(Q, v)
  while Q *not empty* **do**
    AUTree(Q)
  return *Fail*

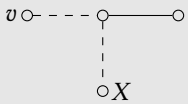
Algorithm 50: AUTree(Q)

$u \longleftarrow$ Dequeue(Q)
if u.mark == even **then**
  for $(u, w) \in E \setminus M$ **do**
    if w.mark == *Null* **then**
      Enqueue(Q, w)
      $w.mark \longleftarrow odd$
      $w.p \longleftarrow u$
if u.mark == odd **then**
  if $\exists\ (u, w) \in M$ *and* w.mark == *Null* **then**
    $w.mark \longleftarrow even$
    $w.p \longleftarrow u$
    Enqueue(Q, w)
  else
    Print "$v \rightsquigarrow u$ augmenting path found"



The above algorithm in each iteration checks if the new vertex has mark NULL before adding to the queue. Because of this we are not adding same vertex more than one into the queue and if we follow the parent and child pointers, this forms a tree. We call this tree to be an M-alternating tree. Denote the tree by T.

Note:-



The algorithm may not visit all the vertices in $L \cup R$ in the tree. For example in case of the graph at left the algorithm will not find the vertex

Since the algorithm runs a BFS if there was an edge between two vertices at levels separated by 2 we would have explored that vertex earlier. So our first observation is:

Observation 13.1. *In the tree T there are no edges between vertices at levels separated by 2.*

Observation 13.2. *All even vertices except v are matched in T.*

Observation 13.3. *There are no edges between two odd levels or even levels.*

Lemma 13.1.2

If leaf u is odd there is a $v \rightsquigarrow u$ augmenting path.

Proof: If the odd vertex u is unmatched then clearly there is a $v \rightsquigarrow u$ augmenting path. So let's assume u is matched. Say $(u, w) \in M$. If w is not in T then u can not be a leaf as the algorithm will take the edge $(u, w) \in M$ for next iteration.

So suppose w is in T. Then $w.mark = even$ since otherwise we would have taken then (w, u) edge in T before. But by Observation 13.2 all the even vertices except v are matched in the tree already. So u can not be matched with w ■

Now from the tree T we partition the vertices of T into the even marked vertices and odd marked vertices. So let $L_T = L \cap T$ and $R_T = R \cap T$. Therefore, L_T is the set of even marked vertices and R_T is the set of odd marked vertices.

Lemma 13.1.3

$N(L_T) = R_T$

Proof: Vertices in L_T are even vertices from which we explore all the edges not in M. Also, all the even vertices except v are matched. So except v for all the vertices in L_T their parent is the matched vertex. Hence, for all even vertices except v all the neighbors are in R_T. Since v is exposed v has no matched neighbor. So all the neighbors of v are also in R_T. Therefore, $N(L_T) = R_T$. ■

Lemma 13.1.4

Suppose we start the algorithm from an exposed vertex v. Suppose there is no augmenting path from v and let the tree formed by the algorithm is T. Then $|L_T| = |R_T| + 1$.

Proof: Since there is no augmenting path the graph all the leaves of T are even vertices. Otherwise, the leaves are odd vertices and then all of them have to be matched. If not then there will exists an augmenting path. Therefore, all the leaves of T are even vertices. Now since the vertices in L_T are even vertices and all even vertices except v are matched to unique odd vertex in R_T we have $|L_T| = |R_T| + 1$. ■

Now suppose M is a matching. Let $L' = \{v_1, \ldots, v_k\} \subseteq L$ are unmatched vertices. Therefore, $|M| = |L| - k$. Then consider the following algorithm:

- Let T_1 be M-alternating tree from v_1 by Find-Augmenting-Path(G, v_1). L_{T_1}, R_{T_1} are vertices of T_1.
- Let T_2 be M-alternating tree from v_2 by Find-Augmenting-Path$(G \setminus T_1, v_2)$. L_{T_2}, R_{T_2} are vertices of T_2.
- Let T_3 be M-alternating tree from v_3 by Find-Augmenting-Path$(G \setminus (T_1 \cup T_2), v_3)$. L_{T_3}, R_{T_3} are vertices of T_3. $\cdots$
- Let T_k be M-alternating tree from v_k by Find-Augmenting-Path$\left(G \setminus \left(\bigcup_{i=1}^{k-1} T_i\right), v_k\right)$. L_{T_k}, R_{T_k} are vertices of T_k.

Observation 13.4. *v_i is not in T_j for any $j < i$ because otherwise we would have found an augmenting path in T_j.*

Now L_{T_i} for all $i \in [k]$ are disjoint and R_{T_i} for all $i \in [k]$ are disjoint. If G had no augmenting path from v_i for all $i \in [k]$ then there are no augmenting paths in $G \setminus \left(\bigcup_{i=1}^{j} T_i\right)$ for all $j \in [k-1]$ from v_{j+1}. Therefore, by Lemma 13.1.4 we have $|L_{T_i}| = |R_{T_i}| + 1 \; \forall \; i \in [k]$. Hence, we have

$$\sum_{i=1}^{k} |L_{T_i}| = \sum_{i=1}^{K} (|R_{T_i}| + 1) \implies \left|\bigcup_{i=1}^{k} L_{T_i}\right| = \left|\bigcup_{i=1}^{k} R_{T_i}\right| + k$$

Now by Lemma 13.1.3, $N(L_{T_{j+1}}) = R_{T_{j+1}}$ for all $j \in [k-1]$ in $G \setminus \left(\bigcup_{i=1}^{j} T_i\right)$. Hence,

$$N(L_{T_j}) \subseteq \bigcup_{i=1}^{j} R_{T_i} \implies N\left(\bigcup_{i=1}^{k} L_{T_i}\right) = \bigcup_{i=1}^{k} R_{T_i}$$

But $\left|\bigcup_{i=1}^{k} L_{T_i}\right| = \left|\bigcup_{i=1}^{k} R_{T_i}\right| + k$. Therefore, any matching of $\bigcup_{i=1}^{k} L_{T_i}$ must leave at least k vertices unmatched. Now all the vertices in $L \setminus \left(\bigcup_{i=1}^{k} L_{T_i}\right)$ with $R \setminus \left(\bigcup_{i=1}^{k} R_{T_i}\right)$ and vice versa. Therefore, any matching of L must leave at least k vertices unmatched. Since M is a matching such that exactly k vertices are unmatched. M is a maximum matching. Therefore, if there is no augmenting path in G then M is a maximum matching.

We also showed before that if M is a maximum matching then there is no augmenting path in G. Therefore, we have the following theorem:

Theorem 13.1.5 Berge's Theorem

A matching M is maximum if and only if there are no augmenting paths in G.

Therefore, if we start with any matching and each time we find an augmenting path we update the matching by taking the odd edges in the augmenting path and obtain a larger matching. After continuously doing this once when there is no augmenting path we can conclude that we obtained a maximum matching.

Since every time the size of the maximal matching is increased by at least 1. The total number of iterations the algorithm takes to output the maximal matching is $O(n)$ where n is the number of vertices in G. In each iteration it calls the Find-Augmenting-Path algorithm which takes the time same as time taken in BFS. Hence, Find-Augmenting-Path takes $O(m+n)$ time. Therefore, the BP-Maximum-Matching algorithm takes $O(n(n+m))$ time.

Algorithm 51: BP-MAXIMUM-MATCHING(G)

Input: $G = (L \cup R, E)$ bipartite graph
Output: Find a maximum matching

```
1  begin
2      M ⟵ ∅
3      while True do
4          v ⟵ unmatched vertex
5          p ⟵ FIND-AUGMENTING-PATH
6          if p == FAIL then
7              return M
8          for e ∈ p do
9              if e ∈ M then
10                 M ⟵ M \ {e}
11             else
12                 M ⟵ M ∪ {e}
```

13.1.3 Using Matrix Scaling

Here we will show a new algorithm for deciding if a bipartite graph has a perfect matching using matrix scaling. The paper which we will follow is [LSW98]

BIPARTITE PERFECT MATCHING
Input: Graph $G = (L \cup R, E)$
Question: Decide if G has a perfect matching or not.

Suppose $G = (L \cup R, E)$ a bipartite graph. If bipartite adjacency matrix of the graph G is A then the permanent of the matrix A,

$$per(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} x_{i,\sigma(i)}$$

counts the number of perfect matchings in G. So we want to check if for a given bipartite graph $(L \cup R, E)$, $per(A) > 0$ or not where A is the bipartite adjacency matrix. Now there is a necessary and sufficient condition for existence of perfect matching in a bipartite graph which is called Hall's condition.

Theorem 13.1.6 Hall's Condition

A bipartite graph $G = (L \cup R, E)$ has an L-perfect matching if and only if $\forall\, S \subseteq L$, $|S| \leq |N(S)|$ where $N(S) = \{v \in R \colon \exists\, u \in L,\ (u, v) \in E\}$

Proof: Now if G has an L-perfect matching then for every $S \subseteq L$, S is matched with some $T \subseteq R$ such that $|S| = |T|$. Therefore, $T \subseteq N(S) \implies |S| = |T| \leq |N(S)|$.

Now we will prove the opposite direction. Suppose for all $S \subseteq L$ we have $|S| \leq |N(S)|$. Assume there is no L-perfect matching in G. Let M be a maximum L-matching in G. Let $u \in L$ is unmatched. Now consider the following sets:

$$X = \{x \in L \colon \exists\, M\text{-alternating path from } u \text{ to } x\}, \qquad Y = \{y \in R \colon \exists\, M\text{-alternating path from } u \text{ to } y\}$$

Now notice that $N(X) \subseteq Y$. Since in a M-alternating path from u whenever the odd edges are not matching edges and the even edges are matching edges. So in the odd edges we can pick any neighbor except the one it is matched with and the immediate even edge before that connects that vertex with the vertex in R it is matched with. Hence, we have $N(X) \subseteq Y$.

Now it suffices to prove that $|X| > |Y|$. Now let $y \in Y$. Suppose $u \rightsquigarrow x' \to y$ be the M-alternating path. If y is not matched then we could increase the matching by taking the odd edges of the path and thus obtain a matching with larger size than M. But M is maximum matching. Hence, y is matched. Therefore, we can extend the path by taking the matching edge incident on y and go the vertex $x'' \in L$ i.e. the new M-alternating path becomes $u \rightsquigarrow x' \to y \to x''$ to have an M-alternating path $u \rightsquigarrow x''$. So $|X| > |Y|$.

Therefore, we obtained a set of vertices $X \subseteq Y$ such that $|X| > |Y| \geq N(X)|$. This contradicts the assumption. Hence, contradiction. Therefore, G has an L-perfect matching. ∎

We will use hall's condition on the adjacency matrix to check if $per(A)$ is positive or not. Now multiplying a row or a column of a matrix by some constant c also multiplies the permanent of the matrix by c as well. In fact if $d_1, d_2 \in \mathbb{R}^n_+$ and $D_1 = diag(d_1)$ and $D_2 = diag(D_2)$ then $per(D_1AD_2) = \left(\prod_{i=1}^{n} d_{1_i}\right)\left(\prod_{i=1}^{n} d_{2_i}\right) per(A)$. So we can scale our original matrix A to obtain a different matrix B and from B we can approximate $per(A)$ by approximating $per(B)$. A natural strategy is to seek an efficient algorithm for scaling A to a doubly stochastic B.

Definition 13.1.2: Doubly Stochastic Matrix

A matrix $M \in \mathbb{R}^{m\times m}$ is doubly stochastic if entries are non-negative and each row and column sum to 1.

First we will show that Hall's Condition holds for doubly stochastic matrix. First let's see what it means for a matrix to satisfy hall's condition. A matrix with all entries non-negative holds Hall's Condition if for all $S \subseteq [n]$ if $T = \{i \in [n] : \exists\ j \in S,\ A(i,j) \neq 0\}$ then $|T| \geq |S|$. This also corresponds to the bipartite adjacency matrix satisfying the hall's condition since for any set of rows S the number of columns for which in the S rows at least one entry is non-zero should be greater than or equal to $|S|$.

Lemma 13.1.7

Hall's Condition holds for doubly stochastic matrix.

Proof: Let M be the doubly stochastic matrix. Let $S \subseteq [n]$. So consider the $|S| \times n$ matrix which only consists of the rows in S. Call this matrix M^r_S. Now suppose T be the set of columns in M^r_S which has nonzero entries. Now consider the $n \times |T|$ matrix which only consists of the columns in T. Call this matrix M^c_T. Now since M is doubly stochastic we know sum of entries of M^r_S is $|S|$ and sum of entries of M^c_T is $|T|$. Our goal is to show $|S| \leq |T|$. Now since T is the only set of columns which have nonzero columns in M^r_S the elements which contributes to the sum of entries in M^r_S are in the T columns in M^r_S. Since these elements are also present in M^c_T we have $|T| \geq |S|$. ∎

Hence, for doubly stochastic matrices the permanent is positive. Now not all matrices are doubly stochastic. And in fact matrices with permanent zero will not be doubly stochastic, so no amount of scaling will make it doubly stochastic. So we will settle for approximately doubly stochastic matrix. In order to make a matrix doubly stochastic first for each row we will divide the row with their row some. Now it becomes row stochastic. Then if it's not approximately doubly stochastic for each column we will divide the column entries with their column sum. But first what ϵ-approximate doubly stochastic matrix means.

Definition 13.1.3: ϵ−Approximate Doubly Stochastic Matrix

A matrix is ϵ−approximate doubly stochastic if for each column, the column sum is in $(1-\epsilon, 1+\epsilon)$ and for each row, the row sum is in $(1-\epsilon, 1+\epsilon)$

Now we will show that even for ϵ-approximate doubly stochastic matrix the hall's condition holds.

Lemma 13.1.8

Halls's Condition holds for ϵ-approximate doubly stochastic matrix for $\epsilon < \frac{1}{10n}$

Proof: Let M is ϵ−approximate doubly stochastic matrix. Let $S \subseteq [n]$. So consider the $|S| \times n$ matrix which only consists of the rows in S. Call this matrix M^r_S. Now suppose T be the set of columns in M^r_S which has nonzero entries. Now consider the $n \times |T|$ matrix which only consists of the columns in T. Call this matrix M^c_T. Now the sum of entries in M^r_S is $\geq |S|(1-\epsilon)$ and sum of entries in M^c_T is $\leq |T|(1-\epsilon)$. Now since T is the only set of columns which have nonzero columns in M^r_S the elements which contributes to the sum of entries in M^r_S are in the T columns in M^r_S. Since these elements are also present in M^c_T we have $||T|(1+\epsilon) \geq |S|(1-\epsilon)$. Therefore we have

$$|T| \geq |S|\frac{1-\epsilon}{1+\epsilon} = |S|\left(1 - \frac{2\epsilon}{1+\epsilon}\right) \geq |S|(1-2\epsilon) > |S|\left(1 - \frac{1}{5n}\right) \geq |S|\left(1 - \frac{1}{|S|}\right) > |S| - 1$$

Since T is an integer we have $|T| \geq |S|$. Hence the Hall's condition holds. ∎

Therefore, permanent of ϵ-approximate doubly stochastic matrix is also positive. Hence, our algorithm for bipartite perfect matching is:

Algorithm 52: BP-MATRIX-SCALING

Input: Bipartite adjacency matrix A of $G = (L \cup R, E)$
Output: Decide if G has a perfect matching.

```
begin
    while True do
        A ⟵ Scale every row of A to make it row stochastic.
        if All column-sums are in (1 − ϵ, 1 + ϵ) then
            return Yes
        A ⟵ Scale every column of A to make it column stochastic.
        if All row-sums are in (1 − ϵ, 1 + ϵ) then
            return Yes
```

In both if conditions we are checking if the matrix is $\epsilon-$approximate doubly stochastic matrix. The moment it becomes a $\epsilon-$approximate doubly stochastic matrix we are done.

Now if G doesn't have a perfect matching then we will never reach a ϵ-approximate doubly stochastic matrix since otherwise Hall's condition will hold, and then we will have that the permanent is positive. So if G doesn't have a perfect matching the algorithm will run in an infinite loop. We only need to check if G has a perfect matching the algorithm returns Yes.

We will now define a potential function $\Phi\colon \mathbb{Z}_0 \to \mathbb{R}_+$. Let $\sigma \in S_n$ such that $a_{i,\sigma(i)} \neq 0$ for all $i \in [n]$. Now if an entry of the matrix is nonzero then it is always nonzero since all the entries are non-negative. Now since the scalings are symmetric we will define the potential function for i^{th} scaling (row/column) is $\Phi(i) = \prod_{i=1}^{n} a_{i,\sigma(i)}$. So we have $\Phi(0) = 1$ since at first all the entries of the matrix are from $\{0, 1\}$. Also, we know $\Phi(t) \leq 1$ for all t since every time we are scaling the matrix. Now $\Phi(1) \geq \frac{1}{n^n}$ since every row-sum can be at most n so it will be divided by n and therefore $a_{i,\sigma(i)} \geq \frac{1}{n}$ for all $i \in [n]$. Now to show the while loop stops if G has a perfect matching it suffices to show that $\Phi(t)$ increases by a multiplicative factor. So we have the following lemma.

Lemma 13.1.9

For all t, $\Phi(t+1) \geq \Phi(t)(1+\alpha)$ for some $\alpha \in (0, 1)$.

Proof: Let A' denote the matrix at the t^{th} scaling where the $(t-1)^{th}$ scaling was column-scaling. Let A'' denote the matrix after row-scaling. Now since we went to the next iteration not all column-sums are in $(1-\epsilon, 1+\epsilon)$ after scaling the rows. Now the row sums of A'' are 1. Therefore we have

$$\frac{\Phi(t)}{\Phi(t+1)} = \prod_{i=1}^{n} \textit{Col-sum}_i(A'') \leq \left(\frac{\sum_{i=1}^{n} \textit{Col-sum}_i(A'')}{n}\right)^n = \left(\frac{\sum_{i=1}^{n} \textit{Row-sum}_i(A'')}{n}\right)^n = 1 \implies \Phi(t) \leq \Phi(t+1)$$

Similarly we can say the same if $(t-1)^{th}$ scaling was row-scaling. Since not all column-sums are in $(1-\epsilon, 1+\epsilon)$ we have $\sum_{i=1}^{n} (\textit{Col-sum}_i(A'') - 1)^2 \geq \epsilon^2$. Therefore using Lemma 13.1.10 we have

$$\frac{\Phi(t)}{\Phi(t+1)} \leq 1 - \frac{\epsilon^2}{2} \implies \Phi(t+1) \geq \left(1 + \frac{\epsilon^2}{2}\right)\Phi(t)$$

Therefore we have the lemma. ∎

We have $\epsilon < \frac{1}{10n}$. Therefore, if $t \geq 200n^4$ then we have

$$1 \geq \Phi(t) \geq \frac{1}{n^n}\left(1 + \frac{1}{200n^2}\right)^t \geq \frac{1}{n^n} e^{n^2} > 1$$

Hence the while loop will iterate for at most $200n^4$ iterations. Hence, this algorithm takes $O(n^4)$ time. Hence, if G has a perfect matching the algorithm runs for at most $O(n^4)$ iterations. And if G doesn't have a perfect matching then the loop never stops. So we have the new modified algorithm to prevent infinite looping:

Algorithm 53: BP-Matrix-Scaling

Input: Bipartite adjacency matrix A of $G = (L \cup R, E)$
Output: Decide if G has a perfect matching.

```
begin
    ϵ ⟵ 1/(20n)
    for i ∈ [200n^4] do
        A ⟵ Scale every row of A to make it row stochastic.
        if All column-sums are in (1 − ϵ, 1 + ϵ) then
            return Yes
        A ⟵ Scale every column of A to make it column stochastic.
        if All row-sums are in (1 − ϵ, 1 + ϵ) then
            return Yes
```

We will prove the helping lemma needed to prove Lemma 13.1.9.

Lemma 13.1.10

Suppose $x_1, \ldots, x_n \geq 0$ and $\sum_{i=1}^{n} x_i = n$ and $\sum_{i=1}^{n} (1 - x_i)^2 \geq \delta$. Then $\prod_{i=1}^{n} x_i \leq 1 - \frac{\delta}{2} + o(\delta)$.

Proof: Denote $\rho_i = x_i - 1$. So $\sum_{i=1}^{n} \rho_i = 0$ and $\sum_{i=1}^{n} \rho^2 \geq \delta$. Now

$$\log(1+\rho_i) = \sum_{j=1}^{\infty} (-1)^{j-1} \frac{\rho_i^j}{j} \implies \log(1+\rho_i) \leq \rho_i - \frac{\rho_i^2}{3} + \frac{\rho_i^3}{3} \implies 1 + \rho_i \leq e^{\rho_i - \frac{\rho_i^2}{3} + \frac{\rho_i^3}{3}}$$

Therefore we have

$$\prod_{i=1}^{n} x_i \leq \exp\left[\sum_{i=1}^{n} \rho_i - \sum_{i=1}^{n} \frac{\rho_i^2}{3} + \sum_{i=1}^{n} \frac{\rho_i^3}{3}\right] \leq \exp\left[0 - \frac{\delta}{2} + \frac{\left(\sum_{i=1}^{n} \rho_i^2\right)^{\frac{3}{2}}}{3}\right] = \exp\left[-\frac{\delta}{2} + \frac{\delta^{\frac{3}{2}}}{3}\right] \leq 1 - \frac{\delta}{2} + o(\delta)$$

Therefore we have the lemma. ∎

There is also a survey, [Ide16] on use of matrix scaling in different results.

13.2 Matching in General Graphs

Here we give a similar algorithm[1] for finding maximum matching in general graph as in the case of bipartite graphs in subsection 13.1.2. We will give a similar characterization for the maximum matching in general graphs. First we will show an extension of Berge's lemma to general graphs.

Theorem 13.2.1 Berge's Lemma

For any graph $G = (V, E)$, $M \subseteq E$ is a maximal matching if and only if there is no augmenting paths in G.

[1] I learned this algorithm in both Algorithm course by Umang and Combinatorial Optimization course by Kavitha. So I am mixing their notes here.

Proof: Suppose M is a maximal matching. Then if G has an augmenting path p. Then we can just take the odd edges in p and then replace the edges in $M \cap p$ with those edges i.e. $M \triangle p$ and this is a larger matching than M. But this contradicts the maximum property of M. Hence, G has no augmenting paths.

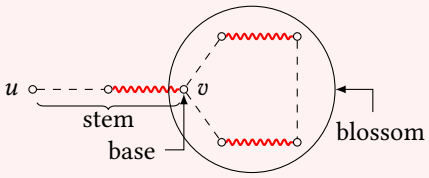
Now we will show that if M is not a maximum matching then G has an augmenting path. So suppose M is not a maximum matching. Let M' is a maximum matching. Then consider the graph $G' = (V, E')$ where $E' = M \triangle M'$. Now every vertex in V has degree $\in \{0, 1, 2\}$ in G'. Hence, the connected components of G' are isolated vertices, paths and cycles. In a path or cycle the edges of M and M' not in both appear alternatively. Therefore, the cycles are even cycles. Since $|M'| > |M|$ there exists a path p such that number $|p \cap M'| > |p \cap M|$. Therefore, the starting and ending edge of p are in M'. Hence, p is an augmenting path in G. ■

13.2.1 Flowers and Blossoms

By the above theorem like in the case of bipartite graphs we will search for augmenting paths in G for matching M and if we can find an augmenting path p we will update the matching by taking $M' = M \triangle p$ and obtain a larger matching. But unlike bipartite graphs we can not run the same algorithm for finding augmenting paths as there can be edges between two odd layers or two even layers. So in the M-alternating tree there can be odd cycles, but these odd cycles have all vertices except one vertex are matched using edges of the cycle. So we look for these special structures in the M-alternating tree called *blossom* and *flower*.

Definition 13.2.1: Flower and Blossom

For a matching M a *flower* consists of an even M-alternating path P from an exposed vertex u to vertex v, called the *stem* and an odd cycle containing v in which the edges alternate between in and out of the matching except for the two edges incident to v. This odd cycle is called the *blossom*.



Observation 13.5. *For a flower since the stem is an even augmenting path the base of the blossom is even as well ass all the other vertices of the blossom are even.*

Since blossoms are in the way of getting augmenting paths we want to remove the blossoms from the graph.

13.2.2 Shrinking Blossoms

In order to remove the blossoms from the graph we will shrink the blossoms into a single vertex every time we encounter a blossom while constructing the augmenting tree.

Question 13.1

How to shrink a blossom into a single vertex?

Let B be a blossom in G. Then the new graph is $G/B = (V', E')$ where

$$V' = (V \setminus B) \cup \{v_b\}, \qquad E' = \Big(E \setminus \{(u,v) : u \in B \text{ or } v \in B\}\Big) \cup \{(u, v_b) : u \notin B, v \in B, (u,v) \in E\}$$

So if M is a matching in G then we can also a get a matching M/B in G/B from M after shrinking B into a single vertex where $M/B = M \setminus \{\text{Matching edges in } B\}$.

Theorem 13.2.2

Let B be a blossom wrt M. M is a maximum matching in G if and only if M/B is a maximum matching in G/B.

Proof: ($\Longrightarrow$) : Suppose M/B is not maximum matching in G/B. Let N is a matching in G/B larger than M/B. Now if N has no edge incident to v_b then N is also a matching in B. Let N' is the matching in G. If there is an edge (u, v_b) incident on v_b in N then we can expand the blossom B and get the matching where the base of B is matched with u and other

vertices of B are matched inside B. So we have the matching $N' = (N \setminus \{e\}) \cup \{(u, w)\}$ where w in B connected to u in G. Since $|N'| = |N| > |M/B|$ and B has $\frac{|B|-1}{2}$ matching edges we have $|N| + \frac{|B|-1}{2} > |M/B| + \frac{|B|-1}{2} = |M|$. But M is maximum matching in G. Hence, contradiction. Therefore, M/B is a maximum matching in G/B.

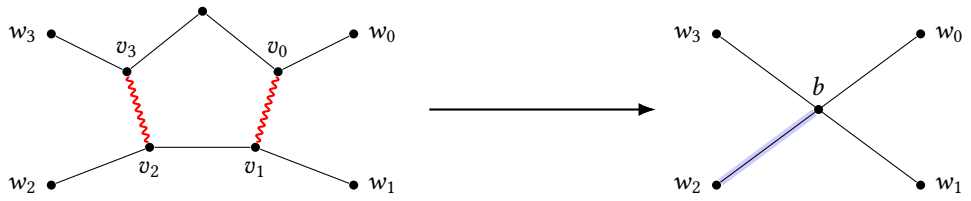
($\Longleftarrow$) : Suppose M is not a maximum matching in G. Now WLOG we can assume the blossom has an empty stem. Otherwise, if Q is the stem of B we can consider the matching $M' = M \oplus Q$ and $|M'| = |M|$. We now we will work with a matching which gives B empty stem, and we will call the matching M. This will make the base of the blossom B an exposed vertex. Now since M is not a maximum matching in G there exists an augmenting path $P : u \rightsquigarrow v$. Now if P has no vertex of B then P/B is also an augmenting path in G/B, but we assumed that M/B is a maximum matching in G/B. Hence, P must have a vertex of B. Let w be the first vertex of P in B. Then vertex v_b in G/B is unmatched. We remove the part $w \rightsquigarrow v$ from P. Let $P' = u \rightsquigarrow w$. Now if w is the base of B then P' is an augmenting path, and it is also an augmenting path of G/B which is not possible. So w is not the base of the B.

The last edge of P' is matched then it is also an edge of B. Then w is not the first vertex of P since the other end of the last of P' is before w. If the last edge of P' is not matched then P' is already an odd length alternating path from an exposed vertex. Inside B we can find an even length alternating path from w to the base of B where the edge incident on w is matched edge. Let that path is $\hat{P}$. Now consider the path $\tilde{P} = P' + \hat{P}$. It is an augmenting path from u to the base of B. Now since v_b is unmatched in G/B, $\tilde{P}/B$ is also an augmenting path in G/B. But this contradicts the assumption that M/B is a maximum matching in G/B. Then P can not exist. Hence, M is a maximum matching in G. ■

Question 13.2

If we find a maximum matching M^* in G/B and then let $N = M^* \cup$ (Matching edges in B), is N a maximum matching in G?

The answer is no. Consider the following example



$M = \{(u_0, u_1),\ (u_2, u_3)\}$

$N = \{(w_2, b)\}$

$\Downarrow$

$N' = \{(w_2, u_2),\ (u_3, u_4),\ (u_0, u_1)\}$

N' is not maximum matching. Since $\{(v_i, w_i) \mid i \in \{0, 1, 2, 3\}\}$ is maximum matching.

Note:-

The above is not contradicting the theorem since the blossom with respect to N' is not the blossom with respect to M.

13.2.3 Algorithm for Maximum Matching

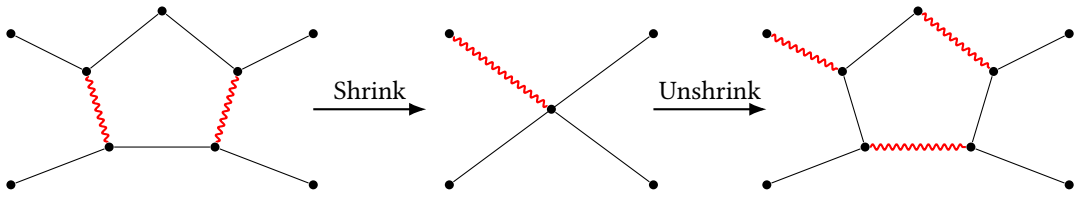
Suppose we start with a matching M in G. We mark all the exposed vertices to be even and keep all the other vertices unmarked at this point. Hence, initially all the vertices are marked even. Now we use the same algorithm for finding augmenting paths as in the case of bipartite graphs but with slight modifications. In the case of bipartite graphs at each iteration we created the M-alternating tree we went one level at a time. But in the case of general graphs we'll go two levels at a time. So at each iteration we start with a vertex which is labeled even. Let u be a vertex labeled. Now for each neighbor v of u:

Case 1: v is unmarked. This implies v is matched. Then mark v as odd and $M(v)$ i.e. the vertex matched with v as even and continue the algorithm.

Case 2: v is marked, and it is in the same tree as u. Then we have a blossom B with respect to M. We shrink the blossom B and therefore, we have the matching M/B in the graph G/B. Mark the shrunk vertex v_b even and continue the algorithm in the graph G/B with the matching M/B.

If we get an even cycle then we just ignore i.e. ignore any neighbor marked odd.

Case 3: v is marked even, and it is in a different tree from u. Let r_u and r_v are the root exposed vertices in the tree of u or v respectively. Then consider the path $P : r_u \rightsquigarrow u \rightarrow v \rightsquigarrow r_v$. This is an augmenting path from r_u to r_v. Hence, the algorithm found an augmenting path with respect to M. Now unshrink the blossoms in P to get the alternating path in G. Let that path is $\hat{P}$. So the algorithm updates the matching M to $M \oplus \hat{P}$, and then we will start the algorithm again with the new matching $M \oplus \hat{P}$.



Time Complexity: The algorithm performs at most n augmentations. Between two augmentations, it will shrink a blossom at most $\frac{n}{2}$ times as each shrinking reduces the number of vertices by at least 2. The time to construct the alternating tree is at most $O(n+m)$. Hence, the total time taken by the algorithm is $O(n^2(n+m)) = O(n^2m)$.

13.2.4 Tutte-Berge Theorem

Theorem 13.2.3

In any graph $G = (V, E)$ for any matching M in G and any $S \subseteq V$,

$$|M| \leq \frac{|V| + |S| - Odd(G - S)}{2}$$

where $Odd(G - S)$ is the number of odd components in $G - S$.

Proof: If $|S| > Odd(G - S)$ then we already have this since $|M| \leq \frac{|V|}{2}$. So let $|S| \leq Odd(G - S)$. Now each odd size component has at least one vertex unmatched in that component. So if that vertex is matched it is matched with a vertex in S. So M leaves at least $Odd(G - S) - |S|$ vertices unmatched. Hence, at most all the rest $|V| - (Odd(G - S) - |S|)$ vertices are matched in G. Therefore, $|M| \leq \frac{|V|+|S|-Odd(G-S)}{2}$. ■

Now the algorithm stops if none of the cases 1, 2, or 3 happens. Let $G' = (V', E')$ is the final graph after shrinking all the blossoms algorithm encountered in its runtime. And let M' is the matching in G' after the algorithm stops. Now we will show that M' is a maximum matching in G'.

Lemma 13.2.4

When none of the 3 cases holds the matching M' is maximum in G'.

Proof: We will show that M' attains equality in Theorem 13.2.3 for some subset S of vertices. Since the algorithm stops the M'-alternating tree in G' has no blossoms. So the M'-alternating tree is a forest. The algorithm marked the vertices of G' as even or odd. Take S be the set of all odd marked vertices. Hence, all the components of $G' - S$ are odd components where each component contains single vertex which is labeled even. So $\text{ODD}(G - S) = |V'| - |S|$. Therefore, $\frac{|V'|+|S|-Odd(G-S)}{2} = \frac{|V'|+S|-(|V|-|S|)}{2} = |S|$. Since all the odd vertices are matched with even vertices in G' we have $|S| = |M'|$. Hence, $|M'| = \frac{|V|+|S|-Odd(G-S)}{2}$. Therefore, M' is a maximum matching in G'. ■

Let the algorithm performs k blossom shrinking. Let $B_1, \ldots, B_k$ are the blossoms. And let M_i be the corresponding matching. $i = 0$ corresponds to the original graph. Let $G_i = (V_i, E_i)$ be the graph after i^{th} blossom shrinking. So $G_0 = G$ and G_k is the final graph after the algorithm stops. The above lemma shows that M_k is a maximum matching in G_k. We

will show that if we unshrink the blossoms one at a time in the reverse order of shrinking then we will get a maximum matching.

Lemma 13.2.5

If M_k is a maximum matching in G_k. Then M_{k-1} is a maximum matching in G_{k-1}.

Proof: G_k is obtained from G_{k-1} by shrinking the blossom B_k. So $G_k = G_{k-1}/B_k$, $M_k = M_{k-1}/B_k$. So $|V_{k-1}| = |V_k| + |B_k| - 1$ and $|M_{k-1}| = |M_k| + \frac{1}{2}(|B_k| - 1)$. Let S be the set of odd vertices in G_k. Now while unshrinking the blossom B_k we add an even number of vertices ($|B_k| - 1$) to one of the connected components to one of the connected components of $G_k - S$ and all these vertices are marked even. So the set of odd vertices of G_{k-1} are the same as set of odd vertices in G_k. Hence, $Odd(G_k - S) = Odd(G_{k-1} - S)$. Therefore,

$$\frac{|V_{k-1}| + |S| - Odd(G_{k-1} - S)}{2} = \frac{|V_k| + |B_k| - 1 + |S| - Odd(G_k - S)}{2} = |M_k| + \frac{|B_k| - 1}{2} = |M_{k-1}|$$

Therefore M_{k-1} is a maximum matching in G_{k-1}. ■

Using the same S we can show that if M_{i+1} is a maximum matching in G_{i+1} then M_i is a maximum matching in G_i. Hence, we can conclude that if M_k is a maximum matching in G_k then M_0 is a maximum matching in G. Therefore, after unshrinking all the blossoms in the reverse order of shrinking we get a maximum matching in G. Therefore, the above algorithm returns a maximum matching of G.

Also, we have shown that the maximum matching attains equality in Theorem 13.2.3 for the set of odd vertices S. Hence, we have the following theorem.

Theorem 13.2.6 Tutte-Berge Theorem

For any graph $G = (V, E)$,

$$\max_{M \text{ matching in } G} |M| = \min_{S \subseteq V} \frac{|V| + |S| - Odd(G - S)}{2}$$

where $Odd(G - S)$ is the number of odd components in $G - S$.

Now from the Tutte-Berge Theorem we conclude that a graph has a perfect matching if and only if for every $S \subseteq V$, the number of odd components in $G - S$ is at most $|S|$. Hence, we have the following corollary.

Corollary 13.2.7 Tutte's Matching Theorem

For any graph $G = (V, E)$, G has a perfect matching if and only if for every $S \subseteq V$, $Odd(G - S) \leq |S|$.

Chapter 14

Linear Programming

14.1 Introduction

Definition 14.1.1: Linear Program

A linear programming problem asks for a vector $x \in \mathbb{R}^d$ that maximizes or minimizes a given linear function, among all vectors x that satisfy given set of linear inequalities.

The general form of a maximization linear programming problem is the following: given $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, $a_i \in \mathbb{R}^n$ for each $i \in [m]$ then

$$\begin{aligned} \text{maximize} \quad & c^T x \\ \text{subject to} \quad & a_i^T x \le b_i \quad \forall\, i \in [p], \\ & a_i^T x = b_i \quad \forall\, i \in \{p+1, \dots, p+q\}, \\ & a_i^T x \ge b_i \quad \forall\, i \in \{p+q+1, \dots, m\}, \\ & x_j \ge 0 \quad \forall j \in [k], \\ & x_j \le 0 \quad \forall j \in [\{k+1, \dots, k+l\} \quad \text{(Some } x_j\text{'s are free)} \end{aligned}$$

The similar goes for minimization linear programming problem. For maximization problem we can always write the LP in the form

$$\begin{aligned} \text{maximize} \quad & c^T \hat{x} \\ \text{subject to} \quad & \hat{a}_i^T x \le b_i' \quad \forall\, i \in [m], \\ & x_j' \ge 0 \quad \forall j \in [n] \end{aligned}$$

And then the LP is said to be in the ***canonical form***. What we can do is the following:

- For $i \in \{p+q+1, \dots, m\}$, we can replace $a_i^T x \le b_i$ with $-a_i^T x \ge -b_i$
- For $i \in \{p+1, \dots, p+q\}$, we can replace with two constraints $a_i^T x \ge b_i$ and $a_i^T x \le b_i$
- For $j \in \{k+1 \dots, k+l\}$, we can replace $x_j \le 0$ with $-x_j \ge 0$
- For $j \in \{k+l+1 \dots, n\}$, we can replace the free x_j's with $x_j^+ - x_j^-$ all the equations where $x_j^+, x_j^- \ge 0$

This way we can always get a LP of that form. Now we can replace the $\hat{a}_i$ for $i \in [m]$ with a matrix $A \in \mathbb{R}^{m \times n}$ and replace the constraint $\hat{a}_i^T x \le b_i'$, $\forall\, i \in [m]$ with $Ax \le b$

$$\begin{aligned} \text{maximize} \quad & c^T x \\ \text{subject to} \quad & Ax \le b, \\ & x \ge 0 \end{aligned} \qquad \begin{aligned} \text{minimize} \quad & c^T x \\ \text{subject to} \quad & Ax \ge b, \\ & x \ge 0 \end{aligned}$$

14.2 Geometry of LP

Definition 14.2.1: Feasible Point and Region

A point $x \in \mathbb{R}^n$ is *feasible* with respect to some LP if it satisfies all the linear constraints. The set of all feasible points is called the *feasible region* for that LP.

Feasible region of a LP has a particularly nice geometric structure. Before that we will first introduce some geometric terminologies used in the linear programming context:

Definition 14.2.2: Hyperplane, Polyhedron, Polytope

- **Line**: The set $\{x + \lambda d, \lambda \in \mathbb{R}\}$ is line for any $x, d \in \mathbb{R}^n$.
- **Hyperplane**: The set $\{x \in \mathbb{R}^n : a^x = b\}$ is a hyperplane for any $a \in \mathbb{R}^n$ and $b \in \mathbb{R}$.
- **Hyperspace**: The set $\{x \in \mathbb{R}^n : a^x \leq b\}$ is a hyperspace or half-space for any $a \in \mathbb{R}^n$ and $b \in \mathbb{R}$.
- **Polyhedron**: A polyhedron is the intersection of a finite set of half-spaces i.e. the set $\{x \in \mathbb{R}^n : Ax \leq b\}$ for any $A \in \mathbb{R}^{n \times m}$, $b \in \mathbb{R}^m$.
- **Polytope**: A bounded polyhedron is called a polytope.

Now it is not hard to verify that any polyhedron is a convex set i.e. if a polyhedron contains two points then it contains the entire line segment joining those two points.

Lemma 14.2.1

Polyhedron is a convex set

Hence the feasible region of a LP creates a polyhedron in $\mathbb{R}^n$. And $c^T x$ is the hyperplane normal to the vector c and the objective of the LP is by moving the plane normal to the vector c for which point in the polyhedron the hyperplane $c^T x$ has the highest value. Since polyhedron can be unbounded there may not exists any point x where $c^T x$ is maximum.

Suppose we have a LP

$$\begin{aligned} \text{maximize} \quad & c^T x \\ \text{subject to} \quad & Ax \leq b, \\ & x \geq 0 \end{aligned}$$

Let P be the polyhedron $P = \{x \in \mathbb{R}^n : Ax \leq b\}$. Then given $x^* \in P$ if any constraint $a_i^T x^* = b_i$ then this constrain is said to be *tight* or *binding* or *active* at x^*. Now two constraints $a_i^T x \leq b_i$ and $a_j^T x \leq b_j$ are said to be linearly independent if a_i and a_j are linearly independent.

Definition 14.2.3: Basic Solution and Basic Feasible Solution

$x^* \in \mathbb{R}^n$ is a basic solution if n linearly independent constraints are active at x^* (Doesn't need to be feasible).

$x^* \in \mathbb{R}^n$ is a basic feasible solution if x^* is a basic solution and $x^* \in P$. The basic feasible solutions are also called *corners* of a polyhedron.

Theorem 14.2.2

Given a LP

$$\begin{aligned} \text{minimize} \quad & c^T x \\ \text{subject to} \quad & Ax \geq b, \\ & x \geq 0 \end{aligned}$$

Let P is the polyhedron $\{x \in \mathbb{R}^n : Ax \leq b, x \geq 0\}$. Suppose P is non-empty and has at least one basic feasible

solution then either the optimal value is $-\infty$ or there is an optimal basic feasible solution.

Theorem 14.2.3

If polyhedron P does not contain a line it contains at least one basic feasible solution (Hence if P is bounded it contains at least one basic feasible solution).

With this geometry in hand, we can easily picture two pathological cases where a given linear programming problem has no solution. The first possibility is that there are no feasible points; in this case the problem is called *infeasible*. The second possibility is that there are feasible points at which the objective function is arbitrarily large; in this case, we call the problem *unbounded*. The same polyhedron could be unbounded for some objective functions but not others, or it could be unbounded for every objective function.

Example 14.2.1

- **Maximum Matchings:** Given undirected graph $G = (V, E)$. Say variable x_e for each $e \in E$, $x_e = 1 \implies e$ in matching and $x_e = 0$ otherwise.

$$\begin{aligned} \text{maximize} \quad & \sum_{e \in E} x_e \\ \text{subject to} \quad & \sum_{e \text{ incident on } v} x_e \leq 1 && \forall\, v \in V, \\ & x_e \geq 0 && \forall\, e \in E, \\ & x_e \in \{0, 1\} && \forall\, e \in E \end{aligned}$$

  Observation. *M is a matching iff $\{x : x_e = 1$ if $e \in M, = 0$ otherwise$\}$ is a feasible solution*

- **Maximum $s - t$ Flow:** Given directed graph $G = (V, E)$ with vertices s, t and capacity c_e on edges. Say variable x_e for each edge and equal to flow on that edge. Then the LP of this problem:

$$\begin{aligned} \text{maximize} \quad & \sum_{e \in out(s)} x_e \\ \text{subject to} \quad & \sum_{e \in in(v)} x_e - \sum_{c \in out(v)} x_e = 0 && \forall\, v \in V, v \neq s, t, \\ & c_e \geq x_e \geq 0 && \forall\, e \in E \end{aligned}$$

We will now introduce a theorem without proof that for any LP with a polytope we can find a solution in polynomial time.

Theorem 14.2.4

Let $P = \{x \in \mathbb{R}^n : Ax \geq b\}$ be a polytope. Then we can find an optimal basic feasible solution for the LP $\min c^T x$ where $x \in P$ in polynomial time.

14.3 LP Integrality

For the LP for matchings in bipartite graphs $G = (L \cup R, E)$ we have:

$$\begin{aligned} \text{maximize} \quad & \sum_{e \in E} x_e \\ \text{subject to} \quad & \sum_{e \text{ incident on } v} x_e \leq 1 && \forall\, v \in V, \\ & x_e \geq 0 && \forall\, e \in E \end{aligned}$$

We want $x_e \in \{0,1\}$ i.e. we want to have integral solution for this LP

Question 14.1

LPs can give fractional solutions. When is solution integral?

Sufficient Condition: Every basic feasible solution of the feasible polytope is integral i.e. x^* is basic feasible solution $\implies x^* \in \mathbb{Z}^n$. If all basic feasible solution are integral then for all $I \subseteq [m]$ with $|I| = n$, $A_I^{-1} b_I$ is integral. Let $x = A_I^{-1} b_I$ Then j^{th} component $x_j = \frac{|A_I^j|}{|A|}$ (Cramer's Rule).

14.3.1 Totally Unimodular Matrix

Definition 14.3.1: Totally Unimodular Matrix (TUM)

A matrix $A \in \{0,1,-1\}^{m \times n}$ is totally unimodular (TU) if every square submatrix of A has determinant $-1, 0, 1$.

Hence in the above LP is A is TU and b is integral then all basic feasible solutions are integral.

Lemma 14.3.1

Let A be TUM and $b \in \mathbb{Z}^n$ then $P = \{x : Ax \geq b\}$ is integral i.e. every basic feasible solution is integral.

Hence using Theorem 14.2.4 if the polytope is integral we can find optimal integral solution in polynomial time. We will now discuss properties of Totally Unimodular Matrix.

Lemma 14.3.2

$A \in \{0,1,-1\}^{m \times n}$ is TU iff the following are TU:

(i) $-A$

(ii) A^T

(iii) $\begin{bmatrix} A & e_i \end{bmatrix}, \begin{bmatrix} A & -e_i \end{bmatrix}$

(iv) $\begin{bmatrix} A & I \end{bmatrix}, \begin{bmatrix} A & -I \end{bmatrix}$

(v) $\begin{bmatrix} A & A_i \end{bmatrix}, \begin{bmatrix} A & -A_i \end{bmatrix}$ where A_i is the i^{th} column of A.

Corollary 14.3.3

If A is TUM and $a, b, c, d \in \mathbb{Z}^n$ are integer vectors then the polytope $Q = \{x \in \mathbb{R}^n : a \leq Ax \leq b, c \leq x \leq d\}$ is integral.

Proof: We can combine the four inequalities in one inequality. Consider the matrix $\begin{bmatrix} A & -A & I & -I \end{bmatrix}^T$. Then the given polytope is

$$Q = \left\{ x \in \mathbb{Z}^n : \begin{bmatrix} A \\ -A \\ I \\ -I \end{bmatrix} x \leq \begin{bmatrix} b \\ -a \\ d \\ -c \end{bmatrix} \right\}$$

By Lemma 14.3.2, $\begin{bmatrix} A & -A & I & -I \end{bmatrix}^T$ is a TUM since A is TUM. Therefore the polytope Q is integral. ■

The following theorem lets us to give a necessary and sufficient condition to check if a given matrix is TUM. Again we will accept the following theorem without the proof since the proof is a little nontrivial.

Theorem 14.3.4

Let $A \in \{-1, 0, 1\}^{m \times n}$. Then A is TU iff every set $S \subseteq [n]$ can be partitioned into S_1, S_2 such that

$$\sum_{i \in S_1} A_i - \sum_{i \in S_2} A_i \in \{-1, 0, 1\}^m$$

where A_i is the i^{th} column of A. C

14.3.2 Integrality of Some Well-Known Polytopes

Now using this theorem we will show that the polytope for bipartite maximum matching is integral. The LP for bipartite maximum matching is given by:

$$\begin{aligned} \text{maximize} \quad & \sum_{e \in E} x_e \\ \text{subject to} \quad & \sum_{e \text{ incident on } v} x_e \leq 1 \quad \forall\, v \in V, \\ & x_e \geq 0 \qquad\qquad\quad \forall\, e \in E \end{aligned}$$

Lemma 14.3.5

The polytope for bipartite maximum matching is integral.

Proof: Let A be the matrix for the polytope. Now clearly from the construction of the polytope we have $A \in \{0, 1\}^{n \times m}$ where $n = |V|$ and $m = |E|$. Now we will show that A^T is TUM. Let L and R are the two sets of vertices in the bipartite graph. Now suppose $S \subseteq L \cup R$. Then take $S_1 = S \cap L$ and $S_2 = S \cap R$. Then for any row $e \in E$, we have

$$\sum_{i \in S_1} A_i - \sum_{i \in S_2} A_i \in \{-1, 0, 1\}$$

Hence A^T is TUM and therefore by Lemma 14.3.2 A is TUM. Hence the polytope for bipartite maximum matching is integral. ■

Note:-

For general graphs this polytope is not integral. Consider the triangle graph K_3. Then the point $\left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}\right)$ is a feasible solution but not in the convex hull of the integral solutions $(1, 0, 0)$, $(0, 1, 0)$ and $(0, 0, 1)$.

Lemma 14.3.6

The LP for $s - t$ max flow is

$$\begin{aligned} \text{maximize} \quad & \sum_{e \in out(s)} x_e \\ \text{subject to} \quad & \sum_{e \in in(v)} x_e - \sum_{e \in out(v)} x_e = 0 \quad \forall\, v \in V, v \neq s, t, \\ & c_e \geq x_e \geq 0 \qquad\qquad\qquad\quad \forall\, e \in E \end{aligned}$$

Then the max flow polytope is integral.

Proof: Let A be the matrix for the polytope. We will show that A is TUM. Given $S \subseteq V \setminus \{s, t\}$ take $S_1 = S$ and $S_2 = \emptyset$. By the first condition of the polytope for all vertices we already have satisfied the condition

$$\sum_{i \in S_1} A_i - \sum_{i \in S_2} A_i = 0 \in \{-1, 0, 1\}^m$$

Therefore the polytope is TUM and hence integral. ■

14.4 Duality

Suppose we have the following LP:

$$\begin{aligned} \text{minimize} \quad & x_1 + 2x_2 \\ \text{subject to} \quad & x_1 - x_2 \geq 3, \\ & 2x_1 + x_2 \geq 1, \\ & x_1, x_2 \geq 0 \end{aligned}$$

Suppose we want to have a lower bount on the optimal solution of the LP. Then we will try to find a linear combination of the constriants such that in the LHS we obtain some thing which is at most the objective function and on the RHS we get the lower bound. So let we multiply the first constraint with y_1, second with y_2. For now y_1, y_2 are unknowns. Then we have the following:

$$\begin{aligned} x_1 + 2x_2 &\geq (y_1 + 2y_2)x_1 + (-y_1 + y_2)x_2 \\ &= y_1(x_1 - x_2) + y_2(2x_1 + x_2) \geq 3y_1 + y_2 \end{aligned}$$

But we also have the conditions that the coefficients of x_1 and x_2 can not be more than the coefficients of x_1 and x_2 in the objective function respectively. So we have the following conditions:

$$\begin{aligned} y_1 + 2y_2 &\leq 1 \\ -y_1 + y_2 &\leq 2 \end{aligned}$$

So now we have found a maximization LP which gives us a lower bound on the optimal solution of the original LP:

$$\begin{aligned} \text{maximize} \quad & 3y_1 + y_2 \\ \text{subject to} \quad & y_1 + 2y_2 \leq 1, \\ & -y_1 + y_2 \leq 2, \\ & y_1, y_2 \geq 0 \end{aligned}$$

This is called the *dual* of the original LP. The original LP is called the *primal* of the dual. The primal and dual are related in a very nice way. The following theorem gives us the relation between primal and dual.

For every minimization LP there is a dual LP that provides a lower bound on the optimal value of the primal LP.

Note:-

If the Primal LP is unbounded then the dual LP is infeasible.

Lemma 14.4.1

Dual of Dual is the primal LP

14.4.1 Dualization of LP

If the primal LP is in canonical form then we have the following:

$$\begin{aligned} \text{maximize} \quad & c^T x \\ \text{subject to} \quad & Ax \leq b, \\ & x \geq 0 \end{aligned} \qquad \Longleftrightarrow \qquad \begin{aligned} \text{minimize} \quad & b^T y \\ \text{subject to} \quad & A^T y \leq c, \\ & x \geq 0 \end{aligned}$$

Primal Dual

Proof of Lemma 14.4.1: Suppose for $A \in \mathbb{R}^{m\times n}$, $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$ we have the following LP:

$$\begin{aligned} &\text{maximize} && c^T x \\ &\text{subject to} && Ax \leq b \quad x \in \mathbb{R}^n \end{aligned}$$

Then the dual the LP is

$$\begin{aligned} &\text{minimize} && b^T y \\ &\text{subject to} && A^T y = c \quad y \in \mathbb{R}^m, \\ & && y \geq 0 \end{aligned}$$

Now consider the LP

$$\begin{aligned} &\text{maximize} && -b^T y \\ &\text{subject to} && A^T y \leq c \quad y \in \mathbb{R}^m, \\ & && -A^T y \leq -c, \\ & && -y \leq 0 \end{aligned}$$

These two LP's are equivalent. Now we obtained another LP which is equivalent to the dual LP. Now we will work with this one. Let

$$\tilde{A} = \begin{bmatrix} A^T \\ -A^T \\ -I_m \end{bmatrix}, \tilde{A} \in \mathbb{R}^{(2n+m)\times m} \qquad \tilde{c} = \begin{bmatrix} c \\ -c \\ 0 \end{bmatrix}, \tilde{c} \in \mathbb{R}^{2n+m} \qquad \tilde{b} = -b$$

Then the above LP is basically the following

$$\begin{aligned} &\text{maximize} && \tilde{b}^T y \\ &\text{subject to} && \tilde{A} y \leq \tilde{c} \quad y \in \mathbb{R}^m \end{aligned}$$

Hence the dual of this LP is

$$\begin{aligned} &\text{minimize} && \tilde{c}^T z \\ &\text{subject to} && \tilde{A}^T z = \tilde{b} \quad z \in \mathbb{R}^{2n+m}, \\ & && z \geq 0 \end{aligned}$$

Let $z = \begin{bmatrix} p & q & r \end{bmatrix}^T$ where $p, q \in bbR^n$ and $r \in \mathbb{R}^m$ and let r_i denote the i^{th} coordinate of r for $i \in [m]$. . For any feasible solution z of the LP $Ap - Aq - r = -b \iff A(q-p) + r = b$. Take $w = q - p$ then

$$Ap - Aq - r = -b \iff A(q-p) + r = b \iff Aw + r = b$$

Since $r \geq 0$ we have $Aw \leq b$. And we have the minimize

$$\tilde{c}^z = c^T p - \tilde{c}^T q = c^T (p - q)$$

Hence it is equivalent to maximize $c^T w$. Since final cost vector doesn't depend on the vector r we can disregard r from the constraints and replace with $Aw \leq r$. Therefore the above LP is equivalent to

$$\begin{aligned} &\text{maximize} && c^T w \\ &\text{subject to} && Aw \leq b \quad w \in \mathbb{R}^n \end{aligned}$$

This is LP is exactly the primal LP. Hence the dual of dual LP is the primal LP. ∎

But if the primal LP is not in the canonical form then we have two options: either we can convert the primal to the canonical form and the dualize it or we can directly dualize the primal LP. The following method gives us a way to dualize the primal LP without converting it tot the canonical form.

$$
\begin{aligned}
&\text{maximize} && c^T x && \\
&\text{subject to} && A_j x \geq b_j && \forall\, j \in [d], \\
& && A_j x = b_j && \forall\, j \in \{d+1, \ldots, m\}, \\
& && x_i \geq 0 && \forall\, i \in [k], \\
& && x_i \text{ is free} && \forall\, i \in \{k+1, \ldots, n\}
\end{aligned}
\qquad \Longleftrightarrow \qquad
\begin{aligned}
&\text{minimize} && b^T y && \\
&\text{subject to} && \sum_{j=1}^{m} A_{ji} y_j \leq c_i && \forall\, i \in [k], \\
& && \sum_{j=1}^{m} A_{ji} y_j = c_j && \forall\, i \in \{k+1, \ldots, n\}, \\
& && y_j \geq 0 && \forall\, j \in [d], \\
& && y_j \text{ is free} && \forall\, i \in \{d+1, \ldots, m\}
\end{aligned}
$$

Primal Dual

So we have the following observations:

Observation. *In dualization of a LP which is not in canonical form*

Primal		*Dual*
Non-negative variables	$\Longleftrightarrow$	*Inequality constraints*
Free variables	$\Longleftrightarrow$	*Equality constraints*

14.4.2 Weak and Strong Duality

Now as the motivation for constructing the dual LP. We have the following theorem which proves the any feasible solution of the dual LP indeed gives a lower bound on the optimal solution of the primal LP.

Theorem 14.4.2 Weak Duality Theorem

If x, y are feasible solutions for the primal and dual LPs respectively and then $c^T x \geq b^T y$.

Proof: We have

$$b^T \leq \sum_{j=1}^{d} y_j (A_j x) + \sum_{j=d+1}^{m} y_j (A_j x) = \sum_{j=1}^{d} y_j A_j x = \sum_{j=1}^{m} \sum_{i=1}^{n} y_j A_{ji} x_i = \sum_{i=1}^{n} x_i \sum_{j=1}^{m} A_{ji} y_j \leq \sum_{i=1}^{m} x_i c_i = c^x$$

Hence we have the theorem. ■

We also have a much stronger theorem which tells us that the optimal solutions of the primal and dual LPs are equal.

Theorem 14.4.3 Strong Duality Theorem

Let the primal and dual LP are feasible and x^*, y^* are the optimal solutions of the primal and dual LPs respectively. Then $c^T x^* = b^T y^*$.

Notice that if for any feasible solution y of the dual LP is $c^T x^* = b^T y$ then y must be the optimal solution of the dual LP.

14.4.3 Complementary Slackness

Question 14.2

Suppose we have optimal solutions x^*, y^* of the primal and dual LPs respectively. What can be said about which constraints are tight in the primal and dual?

Theorem 14.4.4 Complementary Slackness

Let x^*, y^* be the optimal solutions of the primal and dual LPs respectively iff:

(i) If $A_j x^* > b_j$ then $y_j^* = 0$.

(ii) If $A^{i^T} y^* < c_i$ then $x_i^* = 0$.

Proof: Suppose x^*, y^* are the optimal solutions of the primal and dual LPs respectively. Then by Strong Duality Theorem we have

$$\sum_{i=1}^{k} x_i \sum_{j=1}^{m} A_{ji} y_j + \sum_{i=k+1}^{n} x_i \sum_{j=1}^{m} A_{ji} y_j = \sum_{i=1}^{k} x_i c_i + \sum_{i=k+1}^{n} x_i c_i$$

So we have

$$\sum_{i=1}^{k} x_i \sum_{j=1}^{m} A_{ji} y_j = \sum_{i=1}^{k} x_i c_i$$

Hence either $x_i = 0$ or $\sum_{j=1}^{m} A_{ji} y_j = c_i$ for all $i \in [k]$. So $A^{i^T} y^* < c_i$ implies $x_i^* = 0$. Similarly we have $A_j x^* > b_j$ then $y_j^* = 0$. ■

There is also a relaxed version of the complementary slackness theorem, Theorem 15.1.4 which is useful in practice. It is explained in the next chapter.

14.4.4 Max-Flow Min-Cut Theorem

So here using LP-duality we give another proof of Max-Flow Min-Cut Theorem. The LP for maximum flow is given by:

$$\begin{aligned}
\text{maximize} \quad & \sum_{e \in out(s)} x_e \\
\text{subject to} \quad & \sum_{e \in in(v)} x_e - \sum_{c \in out(v)} x_e = 0 \quad \forall\, v \in V, v \neq s, t, \\
& c_e \geq x_e \quad \forall\, e \in E, \\
& x_e \geq 0 \quad \forall\, e \in E
\end{aligned}$$

We can convert this LP by adding edges of $in(s)$ and giving them capacity 0. So we have the modified LP:

$$\begin{aligned}
\text{maximize} \quad & \sum_{e \in out(s)} x_e - \sum_{e \in in(s)} x_e \\
\text{subject to} \quad & \sum_{e \in in(v)} x_e - \sum_{c \in out(v)} x_e = 0 \quad \forall\, v \in V, v \neq s, t, \\
& c_e \geq x_e \quad \forall\, e \in E, \\
& x_e \geq 0 \quad \forall\, e \in E
\end{aligned}$$

For the first constraint we have the variables α_v and for the second constrain we have the variables β_e. So the dual of this LP is given by:

$$\begin{aligned}
\text{minimize} \quad & \sum_{e \in E} c_e \beta_e \\
\text{subject to} \quad & -\alpha_u + \alpha_v + \beta_e \geq 0 \quad \forall\, e = (u, v) \in E, u, v \notin \{s, t\}, \\
& \alpha_v \geq 0 \quad \forall\, v \in V, v \neq s, t, \\
& \beta_e \geq 0 \quad \forall\, e \in E
\end{aligned}$$

Now we can add $\alpha_s = 1$ and $\alpha_t = 0$ to the dual LP and obtain the modified dual LP:

$$\begin{aligned}
\text{minimize} \quad & \sum_{e \in E} c_e \beta_e \\
\text{subject to} \quad & \beta_e \geq \alpha_u - \alpha_v + \quad \forall\, e = (u,v) \in E, u, v \notin \{s,t\}, \\
& \alpha_v \geq 0 \qquad\qquad \forall\, v \in V, v \neq s,t, \\
& \beta_e \geq 0 \qquad\qquad \forall\, e \in E, \\
& \alpha_s = 1, \\
& \alpha_t = 0
\end{aligned}$$

Now for the max-flow LP we already proved in Lemma 14.3.6 that the polytope is integral. By Lemma 14.3.2 the polytope for the dual is also integral. Let $x^*, (\alpha^*, \beta^*)$ be the optimal solution of the primal and dual LPs respectively. Now by Complementary Slackness we have the following:

$$x_e^* > 0 \implies \beta_e^* = \alpha_u^* - \alpha_v^* \qquad \text{and} \qquad \beta_e^* > 0 \implies x_e^* = c_e$$

Now $\alpha_s^* = 1$. Let $X = \{v \colon \alpha_v^* \geq 1\}$. Then $s \in X$ and $t \notin X$. Hence X is a $s-t$ cut. Now consider an edge (u,v) out of X. Then

$$\alpha_u^* \geq 1 \text{ and } \alpha_v^* < 1 \implies \beta_e^* > 0 \implies x_e^* = c_e$$

And for an edge $e = (u,v)$ in to X

$$x_e^* > 0, \alpha_u^* < 1, \alpha_v^* \geq 1 \implies \beta_e^* < 0$$

Hence for an edge e into X, $x_e^* = 0$. Hence maximum flow is equal to the $\sum_{e \in out(X)} c_e$ and this is the minimum cut.

14.4.5 Maximum Bipartite Matching minimum Vertex Cover

The maximum bipartite matching problem is given by the following LP:

$$\begin{aligned}
\text{maximize} \quad & \sum_{e \in E} x_e \\
\text{subject to} \quad & \sum_{e \text{ incident on } v} x_e \leq 1 \quad \forall\, v \in V, \\
& x_e \geq 0 \qquad\qquad \forall\, e \in E
\end{aligned}$$

The dual of the LP si given by

$$\begin{aligned}
\text{minimize} \quad & \sum_{v \in V} y_v \\
\text{subject to} \quad & y_u + y_v \geq 1 \quad \forall\, (u,v) \in E, \\
& y_v \geq 0 \quad \forall\, v \in V
\end{aligned}$$

Since in Lemma 14.3.5 we have proved the polytope for bipartite maximum matching is integral the polytope for the dual is also integral.

Definition 14.4.1: Vertex Cover

Given $G = (V, E)$ a vertex cover is a subset $C \subseteq V$ such that $\forall\, e \in E$ at least one of the endpoints of e is in C.

Then we have the following lemma:

Lemma 14.4.5

Let C be a vertex cover. Then there exists a dual feasible solution y such that $\sum_v y_v = |C|$.

Proof: Consider the vector $y \in \{0,1\}^{|V|}$ such that $y_v = 1$ if $v \in C$ and $y_v = 0$ otherwise. Then we have the lemma. ■

Lemma 14.4.6

Let y be an integral dual solution. Then $C = \{v : y_v \geq 1\}$ is a vertex cover.

Proof: For every edge $e = (u, v)$ we have $y_u + y_v \geq 1$. So either $y_u \geq 1$ or $y_v \geq 1$ as y is integral. Hence either $u \in C$ or $v \in C$. Hence every edge is covered by C and hence C is a vertex cover. ■

Note:-

In general graphs computing a minimum sized vertex cover in NP-hard. But since for bipartite graph the polytope is integral we can compute minimum weight vertex cover in polynomial time.

Chapter 15

Approximation Algorithms using LP

In this chapter we will study some approximation algorithms using linear programming to get better approximation ratios of the optimal solution.

15.1 Set Cover

Set Cover
Input: $\mathcal{U}$: Universe of all elements $u_1, \ldots, u_n$
$\mathcal{S} = \{S_1, \ldots, S_m\}$, $S_i \subseteq \mathcal{U}$ forall $i \in [m]$
Function $c : \mathcal{S} \to \mathbb{Z}_0$
Question: Given $\mathcal{U}, \mathcal{S}$ and the function c find $T \subseteq [m]$ such that $\bigcup_{i \in T} S_i = \mathcal{U}$ to minimize the total cost $c(T) = \sum_{i \in T} c(S_i)$

Since the special case of Set Cover is basically the Vertex Cover problem we discussed earlier, we know that Set Cover is NP-hard. We will discuss NP-hardness in the next chapter.

Theorem 15.1.1
Set Cover is NP-hard.

Since we are going to find approximate solutions using LP let's first write the linear program for Set Cover:

$$\begin{aligned} \text{minimize} \quad & \sum_{S \in \mathcal{S}} c(S) x_S \\ \text{subject to} \quad & \sum_{S : u \in S} x_S \geq 1 \quad \forall\, u \in \mathcal{U}, \\ & x_S \geq 0 \quad \forall\, S \in \mathcal{S} \end{aligned}$$

15.1.1 Frequency f-Approximation Algorithm

Let for any element $u \in \mathcal{U}$, f_u is the frequency of the element u in $\mathcal{S}$ i.e. $f_u = |\{S \in \mathcal{S} : u \in S\}|$. Then let $f = \max\{f_u : u \in \mathcal{U}\}$. Then we want to find a f-approximation algorithm for set cover.

Question 15.1
For vertex cover what is f?

For all $e \in E$ we have $f_e = 2$ since the elements of universe corresponds to the edges and the set corresponds to vertices and each edge is contained in exactly 2 sets. So $f = 2$.

Algorithm 54: f-Approximate Algorithm

Input: $\mathcal{U}, \mathcal{S}, c$
Output: $T \subseteq [m]$ such that $\bigcup_{i \in T} S_i = \mathcal{U}$ and $\sum_{i \in T} c(S_i)$ is minimized

```
1 begin
2     T ⟵ ∅
3     x̂ ⟵ 0^{|S|}
4     Let x* is the optimal solution of the LP for Set Cover problem
5     for S_i ∈ S do
6         if x*_{S_i} ≥ 1/f then
7             T ⟵ T ∪ {i}
8             x̂_{S_i} ⟵ 1
9     return T
```

Lemma 15.1.2

$\hat{x}$ is a feasible solution.

Proof: For all $e \in \mathcal{U}$ there are at most f sets containing e. Thus, at most f terms in the summation in *LHS* of the first constraint for each $e \in \mathcal{U}$ Thus in x^* at least one such term is $\geq \frac{1}{f}$. ■

Lemma 15.1.3

$$\sum_{S \in \mathcal{S}} c(S)\hat{x}_S \leq f \cdot \sum_{S \in \mathcal{S}} c(S)x^*_S$$

Proof: In $\hat{x}$ if $\hat{x}_S = 1$ that means $x^*_S \geq \frac{1}{f}$. Therefore, we have the lemma. ■

Hence, with this algorithm we can get a f-approximation for Set Cover problem. In the next subsection we will see a new way of getting the same approximation ratio.

15.1.2 Frequency f-Approximation Algorithm through Dual Fitting

First let's write the dual of the LP for Set Cover problem:

$$\begin{array}{lll} \text{minimize} & \sum_{S \in \mathcal{S}} c(S)x_S & \\ \text{subject to} & \sum_{S: u \in S} x_S \geq 1 & \forall\, u \in \mathcal{U}, \\ & x_S \geq 0 & \forall\, S \in \mathcal{S} \end{array} \iff \begin{array}{lll} \text{maximize} & \sum_{u \in \mathcal{U}} y_u & \\ \text{subject to} & \sum_{u \in S} y_u \leq c(S) & \forall\, S \in \mathcal{S}, \\ & y_u \geq 0 & \forall\, u \in \mathcal{U} \end{array}$$

Primal — "Covering Problem" | Dual — "Packing Problem"

Both the primal and dual are feasible. Let x, y are feasible solutions of the primal and dual respectively. Then by Weak Duality we have

$$\sum_{S \in \mathcal{S}} c(S)x_S \geq \sum_{u \in \mathcal{U}} y_u$$

Let x^*, y^* are the optimal solutions of primal and dual respectively. Then by Complementary Slackness

$$x^*_S > 0 \implies \sum_{u \in S} y^*_u = c(S), \qquad y^*_u > 0 \implies \sum_{S: u \in S} x^*_S = 1$$

Theorem 15.1.4 Relaxed Complementary Slackness

Suppose x, y are feasible solutions of the primal and dual respectively and they satisfy the following conditions:

1. If $x_j > 0$ then $\frac{1}{\alpha} \cdot c_j \leq {A^j}^T y \leq c_j$ where $\alpha \geq 1$.
2. If $y_i > 0$ then $b_i \leq A_i^T x \leq \beta \cdot b_i$ where $\beta \geq 1$.

Then

$$c^T x \leq \alpha\beta \cdot b^T y \leq \alpha\beta \cdot c^T x^* = \alpha\beta \cdot \text{OPT}$$

Proof: x, y are the feasible solutions of the primal and dual respectively. Then we have

$$c^T x = \sum_{j=1}^{m} c_j x_j \leq \sum_{j=1}^{m} \left(\alpha {A^j}^T y\right) x_j = \alpha \sum_{j=1}^{m}\sum_{i=1}^{n} A_{ij} y_i x_j = \alpha \sum_{i=1}^{n} \left(\sum_{i=1}^{m} A_{ij} x_j\right) y_i \leq \alpha \sum_{i=1}^{m} \beta \cdot b_i y_i = \alpha\beta \cdot b^T y$$

Hence we have $c^T x \leq \alpha\beta \cdot b^T y \leq \alpha\beta \cdot c^T x^* = \alpha\beta \cdot \text{OPT}$. ■

To show a f-approximation algorithm for set cover we will first find feasible solutions of primal, dual, x, y which satisfies:

1. x is integral.
2. x satisfies the first condition of Relaxed Complementary Slackness with $\alpha = f$.

Algorithm 55: Dual Fitting Algorithm for Set Cover

Input: $\mathcal{U}, \mathcal{S}, c$
Output: $T \subseteq [m]$ such that $\bigcup_{i \in T} S_i = \mathcal{U}$ and $\sum_{i \in T} c(S_i)$ is minimized

```
begin
    Initialize U' ⟵ U, C ⟵ ∅
    while ∃ u ∈ U' do
        Increase y_u until for some S ∈ S with u ∈ S we have Σ_{u'∈S} y_{u'} = c(S)
        C ⟵ C ∪ { S ∈ S : Σ_{u∈S} y_u = c(S) }
        for S ∈ C do
            U' ⟵ U' \ S
    return C
```

From C we con construct x by $x_S = 1$ if $S \in C$ and otherwise $x_S = 0$ for all $S \notin C$. Now we have the observations:

Observation. *After the algorithm terminates we have:*

1. *At the beginning of the loop if $u \in \mathcal{U}$, $y_u = 0$.*
2. *If $x_S = 1$ and $u \in S$ then y_u is not increased.*
3. *$x \in \{0,1\}^{|S|}$ is integral.*

Lemma 15.1.5

1. x is feasible at the end of the algorithm.
2. y is feasible at every iteration of the while loop

Proof: The algorithm terminates when $\mathcal{U}' = \emptyset$. That means all the elements of the universe are covered. Hence, the set C output after the algorithm terminates is indeed a set cover. Hence, x is a feasible solution.

At the start of the algorithm $y = 0^{|\mathcal{U}|}$. Hence, y is feasible. Now suppose at any iteration y is feasible. If the algorithm goes through another iteration then there exists an element in $\mathcal{U}'$ which is not covered. Let $u \in \mathcal{U}'$ which is not covered. Hence, $y_u = 0$. Since in the previous iteration y was feasible we have $\sum_{S:u\in S} y_u \leq c(S)$. Now we increase y_u to the point we achieve the equality $\sum_{u'\in S} y_{u'} = c(S)$ for all $S \in \mathcal{S}$ with $u \in S$. Therefore, even after updating y_u all the constraints of dual are satisfied. Hence, y is a feasible solution after another iteration of the while loop. Therefore, y is feasible at every iteration of the while loop. ■

Lemma 15.1.6

x, y satisfy the Relaxed Complementary Slackness conditions.

Proof: If for any $S \in \mathcal{S}$, $x_S > 0$ then we have $\sum_{u\in S} y_u = c(S)$ by the construction of C in the algorithm. Therefore,

$$x_S > 0 \implies \sum_{u\in S} y_u = c(S)$$

Hence $\alpha = 1$.

Now let for some $u \in \mathcal{U}$, $y_u > 0$. Since f is the maximum frequency of any element of the universe we have $f \geq \sum_{S:u\in S} x_S \geq 1$. Therefore,

$$y_u > 0 \implies f \geq \sum_{S:u\in S} x_S \geq 1$$

Hence $\beta = f$. ■

Therefore, by Relaxed Complementary Slackness C is an f-approximate solution for the set cover problem. But f-approximation is not good enough since one element can be in too many sets, and then it doesn't give a good approximation. In the next subsection we will show how to get a better approximation ratio.

15.1.3 $O(n \log n)$-Approximation Algorithm through Randomized Rounding

Here we will show a randomized algorithm to get better approximation ratio. The idea is to use the LP we constructed earlier and then randomly select the sets with probability proportional to the value of the corresponding variable in the LP. This is known as randomized rounding.

Algorithm 56: $O(n \log n)$-Approximate Algorithm

Input: $\mathcal{U}, \mathcal{S}, c$
Output: $T \subseteq [m]$ such that $\bigcup_{i\in T} S_i = \mathcal{U}$ and $\sum_{i\in T} c(S_i)$ is minimized

```
begin
    x̂ ⟵ 0^{|S|}
    Let x* is the optimal solution of the LP for Set Cover problem
    for S ∈ S do
        Set x̂_S ⟵ 1 with probability x*_S.
    return x̂
```

From the construction of $\hat{x}$ we have $\mathbb{E}\left[\sum_{S\in\mathcal{S}} c(S)\hat{x}_S\right] = \sum_{S\in\mathcal{S}} c(S)x^*_S$. Now suppose we fixed an element $u \in \mathcal{U}$. Then

$$\mathbb{P}[u \text{ is not covered}] = \prod_{S:u\in S} \mathbb{P}[S \text{ is not selected}] = \prod_{S:u\in S} (1 - x^*_S) \leq \prod_{S:u\in S} e^{-x^*_S} = \exp\left[-\sum_{S:u\in S} x^*_S\right] \leq e^{-1}$$

Hence to reduce the probability of not covering an element of $\mathcal{U}$ we repeat the algorithm multiple times. Hence, we have the updated algorithm:

Algorithm 57: $O(n \log n)$-Approximate Algorithm

Input: $\mathcal{U}, \mathcal{S}, c$
Output: $T \subseteq [m]$ such that $\bigcup_{i \in T} S_i = \mathcal{U}$ and $\sum_{i \in T} c(S_i)$ is minimized

```
1 begin
2     Let x* is the optimal solution of the LP for Set Cover problem
3     for i ∈ [2 log n] do
4         C_i ⟵ ∅
5         for S ∈ 𝒮 do
6             Put S in C_i with probability x*_S.
7     C ⟵ ⋃_{i=1}^{2 log n} C_i
8     return C
```

Again now we fix an element $u \in \mathcal{U}$. Now we will calculate the probability that u is not covered in the union of all C_i's.

$$\mathbb{P}[u \text{ is not covered by } C] = \mathbb{P}[u \text{ is not covered by } C_i \text{ for all } i \in [2 \log n]] \leq e^{-2 \log n} = \frac{1}{n^2}$$

Hence, the probability that e is covered is at least $1 - \frac{1}{n^2}$. Therefore,

$$\mathbb{P}[\exists\, e \in \mathcal{U} \text{ is not covered by } C] \leq \sum_{u \in \mathcal{U}} \frac{1}{n^2} = \frac{1}{n}$$

Hence, $\mathbb{P}[C \text{ is a set cover}] \geq 1 - \frac{1}{n}$. Now we have to bound the cost of C. By Markov's inequality we have

$$\mathbb{P}\left[c(C) \geq 6 \log n \sum_{S \in \mathcal{S}} c(S) x_S^*\right] \leq \frac{1}{3}$$

$$\mathbb{P}\left[C \text{ is not a set cover OR cost of C} \geq 6 \log n \sum_{S \in \mathcal{S}} c(S) x_S^*\right] \leq \frac{1}{n} + \frac{1}{3} \leq \frac{1}{2}$$

Therefore

$$\mathbb{P}\left[C \text{ is set cover AND } c(S) \leq 6 \log n \sum_{S \in \mathcal{S}} c(S) x_S^*\right] \geq \frac{1}{2}$$

Hence with probability at least $\frac{1}{2}$ we have a set cover C such that $c(C) \leq 6 \log n \sum_{S \in \mathcal{S}} c(S) x_S^*$ which gives us an $O(\log n)$-approximation algorithm for Set Cover problem.

Note:-

$O(\log n)$-approximatiobn is also the best we can do for set cover. Doing better than that is NP-hard.

15.2 Makespan Minimization

MAKESPAN

Input: $\mathcal{M}$: Set of m machines
$\mathcal{J}$: Set of n jobs
$P \in \mathbb{N}^{m \times n}$: Matrix where P_{ij} is the time taken by machine i to complete job j.

Question: Given a set of machines M, set of jobs $\mathcal{J}$ and the matrix of time taken by i^{th} machine to complete j^{th} job find an assignment $\sigma : \mathcal{J} \to \mathcal{M}$ of jobs to machines to minimize the makespan $S_\sigma = \max\{l_i : i \in \mathcal{M}\}$ where $l_i = \sum_{j:\sigma(j)=i} P_{ij}$ i.e. time taken by machine i to complete all jobs assigned by σ

Theorem 15.2.1

Makespan problem is weakly NP-hard by reduction from subset-sum.

Note:-

Weakly NP-hard means there exists a pseudo polynomial time algorithm i.e. if all parameters are polynomially large the algorithm can solve the problem in polynomial time.

Theorem 15.2.2

It is NP-hard to approximate within a factor of 1.5

Here we will show a 2-approximate solution of makespan optimization. First let's construct the LP for makespan optimization.

15.2.1 LP Construction

We'll use the variable x_{ij} as an indicator for j^{th} job assigned to i^{th} machine. Then here is the LP:

$$\begin{aligned} \text{minimize} \quad & T \\ \text{subject to} \quad & \sum_{i \in \mathcal{M}} x_{ij} \geq 1 \quad \forall\, j \in \mathcal{J}, \\ & \sum_{j \in \mathcal{J}} P_{ij} x_j \leq T \quad \forall\, i \in \mathcal{M}, \\ & x_{ij} \geq 0 \quad \forall\, i \in \mathcal{M}, j \in \mathcal{J} \end{aligned}$$

So here the first constrain basically says that every job assigned to some machine. The second constraint says that for every machine the total time taken by the machine to complete the jobs should be at most the makespan where T denotes the makespan. But this LP is not good enough. Consider the following example where there is only one job and $P_{i1} = m$ then $\text{OPT}_{LP} = 1$ by setting $x_{i1} = \frac{1}{m}$ where as actually the optimal makespan is m. Hence this LP will not work. We have to strengthen the LP.

So now assume we already know the optimal makespan T. Then if any $P_{ij} > T$ then we know that we can't assign the j^{th} job to i^{th} machine. So now we have the new updated LP:

$$\begin{aligned} \text{minimize} \quad & 0 \\ \text{subject to} \quad & \sum_{i \in \mathcal{M}} x_{ij} \geq 1 \quad \forall\, j \in \mathcal{J}, \\ & \sum_{j \in \mathcal{J}} P_{ij} x_j \leq T \quad \forall\, i \in \mathcal{M}, \\ & x_{ij} \geq 0 \quad \forall\, i \in \mathcal{M}, j \in \mathcal{J}, \\ & x_{ij} = 0 \quad \text{If } P_{ij} > T,\ \forall\, i \in \mathcal{M}\ j \in \mathcal{J} \end{aligned}$$

This basically checks the feasibility for a specific T. Hence, now we can do a binary search over T's to find the smallest feasible T.

Theorem 15.2.3

By binary search $O(\log n)$ round we can find the smallest T such that $LP(T)$ is feasible.

Now suppose we have the smallest feasible time. Let's call this $\hat{T}$. Then $\hat{T} \leq \text{OPT}_I$. Let $\tilde{x}$ is the basic feasible solution for $\hat{T}$. We will now show a polynomial time algorithm to obtain an integral assignment with makespan = $2\hat{T}$.

15.2.2 Rounding to Get 2-Approximate Solution

Now we have the smallest feasible time $\hat{T}$ and the basic feasible solution for that $\tilde{x}$ which is also an extreme point. Now we can think $\tilde{x}$ as a weighted bipartite graph between $\mathcal{J}$ and $\mathcal{M}$ with fractional weights i.e. one job assigned to multiple machines fractionally. Let the graph is $G = (L \sqcup R, E)$ where $e = (i, j) \in E$, if $\tilde{x}_{ij} > 0$ with $w(i, j) = \tilde{x}_{ij}$. Hence, we also have for all $(i, j) \in E$, $\tilde{x}_{ij} \leq \hat{T}$.

Lemma 15.2.4

In $\tilde{x}$ at least $n - m$ jobs are assigned integrally.

Proof: There are total $n + m + nm$ constraints in the LP. But the LP is nm dimensional. Therefore at $\tilde{x}$, nm constraints are tight. So at most $m + n$ constraints of the type $x_{ij} \geq 0$ are not tight i.e. at most $m + n$ many $\tilde{x}_{ij}$ are not zero. Suppose α jobs are set integrally and β fractionally. So for each of the β jobs it is assigned to at least 2 machines. Now each of the $\tilde{x}_{ij}$ corresponds to an edge of the graph. Therefore we have the following two equations:

$$\alpha + \beta = n, \quad \alpha + 2\beta \leq m + n \implies \beta \leq m \implies \alpha \geq n - m$$

Therefore there at least $n - m$ jobs which are assigned integrally. ∎

Lemma 15.2.5

In every connected component of G, #edges $\leq$ #vertices.

Proof: In the graph G, as we showed earlier at most $m + n$ constraints of the type $x_{ij} \geq 0$ are not tight i.e. at most $m + n$ many $\tilde{x}_{ij}$ are not zero. Hence

$$\#\text{edges} = |\{\tilde{x}_{ij} \mid \tilde{x}_{ij} > 0\}| \leq m + n = \#\text{vertices}$$

Suppose C is a connected component. Let $\mathcal{J}_C$, $\mathcal{M}_C$ be the jobs and machines of C and $\tilde{x}|_C$ is $\tilde{x}$ restricted to C. Then $\tilde{x}|_C$ is a basic feasible solution for the instance restricted to $\mathcal{M}_C$, $\mathcal{J}_C$ with $\hat{T}$ being a feasible time. If $\tilde{x}|_C$ was not feasible for $\mathcal{M}_C$ and $\mathcal{J}_C$ then there exists y_C and z_C with $y_C \neq z_C$ such that $x|_C = \lambda y_C + (1 - \lambda) z_C$ where $\lambda \in (0, 1)$. Then

$$\tilde{x} = \lambda \left(y_C, \tilde{x}|_{\overline{C}}\right) + (1 - \lambda) \left(z_C, \tilde{x}|_{\overline{C}}\right)$$

Then $\tilde{x}$ can not be an extreme point. And therefore by the same logic as above we have in the connected component #edges $\leq$ #vertices. Since C is arbitrary connected component this is true for every connected component. ∎

Now we create a feasible solution $\hat{x}$ for $2\hat{T}$. We first initiate $\hat{x}$ setting all 0's. We fix a connected component C in G. Furthermore, we call a vertex in $\mathcal{J}_C \cup \mathcal{M}_C$ leaf if it has degree 1. If for any job $j \in \mathcal{J}_C$ it is assigned integrally in $\tilde{x}|_C$ then j is a leaf. So we remove the node j and assign the job to the machine $i \in \mathcal{M}_C$, j is connected to. This also removes the edge incident on j.

After doing this we still have #edges $\leq$ #vertices because we basically removed same number of jobs and edges from the graph. But now every job is connected to at least two machines.

If a machine $i \in \mathcal{M}_C$ is a leaf, let the edge incident on i is (i, j) then we remove both i, j from the graph and assign the job j to machine i i.e. basically we set $\hat{x}_{ij} = 1$. So the load added to i^{th} machine is at most $\hat{T}$. We do this for every leaf machine.

Now the graph has no leaves remaining. Since the graph is bipartite it is an even cycle. So find a matching of jobs to machines in the cycle and assign the jobs accordingly i.e. if M is a matching and $e = (i, j) \in M$ then set $\hat{x}_{ij} = 1$.

So we have the following final algorithm:

Algorithm 58: Makespan 2-Approximate Algorithm

Input: $\mathcal{M}, \mathcal{J}, P$ where $|\mathcal{M}| = m$, $|\mathcal{J}| = n$ and $P \in \mathbb{Z}_0^{m \times n}$

Output: $\sigma : \mathcal{J} \to \mathcal{M}$ assignment of jobs to machines to minimize $\max\{l_i : i \in \mathcal{M}\}$ where $l_i = \sum_{j:\sigma(j)=i} P_{ij}$ i.e. time taken by machine i to complete all jobs assigned by σ

begin
- Do binary search to find the minimum feasible T for the LP.
- Let $\hat{T}$ is the minimum feasible time and $\tilde{x}$ is the basic feasible solution.
- Construct the weighted graph $G = (\mathcal{M} \sqcup \mathcal{J}, E)$ where $(i, j) \in E$ if $\tilde{x}_{ij} > 0$ and $w(i, j) = \tilde{x}_{i,j}$.
- $C \longleftarrow$ Connected Components of G.
- **for** $C \in C$ **do**
  - **while** $\exists\ j \in \mathcal{J}_C$ *such that* $\deg(j) = 1$ **do**
    - Let $(i, j) \in E$
    - $\sigma(j) \longleftarrow i$
    - $\mathcal{J} \longleftarrow \mathcal{J} \setminus \{j\}$
  - **while** $\exists\ i \in \mathcal{M}_C$ *such that* $\deg(i) = 1$ **do**
    - Let $(i, j) \in E$
    - $\sigma(j) \longleftarrow i$
    - $\mathcal{M} \longleftarrow \mathcal{M} \setminus \{i\}$
    - $\mathcal{J} \longleftarrow \mathcal{J} \setminus \{j\}$
  - $M \longleftarrow$ BP-MAXIMUM-MATCHING. M will be a perfect matching.
  - **for** $e = (i, j) \in M$ **do**
    - $\sigma(j) \longleftarrow i$
- **return** σ

This algorithm works in polynomial time since solving the LP, constructing the weighted graph and finding the connected components can be done in polynomial time and then for every component the while loops and finding matching can also be done in polynomial time. So the algorithm is polynomial time.

This algorithm gives a 2-approximate solution because each machine i is assigned the jobs it is set integrally and another job j if $\tilde{x}_{ij} > 0$.

Chapter 16

P, NP and Reductions

Almost all the algorithms we have studied thus far have been **polynomial time algorithms** i.e. on inputs of size n, their worst-case running time is $O(n^k)$ for some constant k. A natural question to ask is whether all problems can be solved in polynomial time. The answer is no.

There are problems that can be solved but not in polynomial time and there are problems which can not be solved via an algorithm. To discuss problems in general think of computational tasks as language recognition problem. A language is a subset of $\{0,1\}^*$. For example:

$$L_{\text{CONN}} = \{x \in \{0,1\}^* \mid x \text{ represents a connected graph}\}$$

So main problem we want to think about is to decide whether a given string is in the language or not. These problems are also called decision problems.

Definition 16.1: Decision Problems

Given a language $L \subseteq \{0,1\}^*$ and a string $x \in \{0,1\}^*$ decide whether $x \in L$ or not.

An algorithm $\mathcal{A}$ solves this problem if $x \in L \iff \mathcal{A}(x) = 1$. Time complexity of $\mathcal{A}$: $T_{\mathcal{A}}(n)$ is the maximum running time of $\mathcal{A}$ on any string x of length n. Since we can work over any set of alphabets and alphabets can be encoded into binary we will say languages are subset of Σ^* where Σ is the finite set of alphabets.

16.1 Introduction to Complexity Classes

Depending on time, space and some other resources based on how much they are used we divide the computational problems into several sets. We call these sets as complexity classes.

Definition 16.1.1: Polynomial Running Time

A language $L \subseteq \Sigma^*$ has a polynomial-time algorithm if there exists $\mathcal{A}$ that solves L and $T_{\mathcal{A}}(n) = O(n^k)$ for some constant k.

Now we introduce our first complexity class now. This class is called P.

$$\text{P} := \{L \subseteq \Sigma^* \mid \text{ there exists a polynomial time algorithm that decides } L\}$$

Till now all the algorithms we have studied are in P.

Question 16.1

What about $L_{\text{SAT}}, L_{\text{3COL}}, L_{\text{2SAT}}, L_{\text{CONN}}$?

We know $L_{\text{CONN}} \in \text{P}$ since we can run a DFS to check if all the vertex is reachable from a vertex. Also we know $L_{\text{2SAT}} \in \text{P}$. For other languages we don't know if they are in P. But these problems have another noticable nature. Given

a potential solution for the problem one can check if that is indeed a solution of the problem or not in polynomial-time. Let's abstract this notion:

Definition 16.1.2: Short Certificate of Membership

A language have a short certificate of membership if there exists an algorithm $\mathcal{A}$ that runs in polynomial time and

$$\forall\, x \in L,\ \exists\, y \in \text{poly}(|x|), \quad \mathcal{A}(x, y) = 1$$
$$\forall\, x \notin L,\ \forall\, y \in \text{poly}(|x|), \quad \mathcal{A}(x, y) = 0$$

What are the certificates or above-mentioned problems.

- L_{SAT}: Assignment of the variables. Then we can verify if every clause is satisfied
- L_{3COL}: Coloring of the edges. We can verify if all the edges follows the coloring constraint.
- L_{CONN}: A spanning tree. We can verify if every vertex is present there.

Now we introduce another complexity class called NP.

$$\text{NP} := \{L \subseteq \Sigma^* \mid L \text{ has a short certificate of membership}\}$$

For NP we call the algorithm to check for the certificate *verifier*. Another way to think about the class NP is to extend the computer to make "guesses" or exists in multiple states simultaneously. This is known as non-determinism. Then NP is the class of languages decided by a polynomial time non-deterministic Turing machine. For example a non-deterministic algorithm for 3SAT is

- Make a guess for the assignment for each variable.
- If ϕ is satisfied return yes else return no.

Naturally any problem which is in P has a short certificate.

Theorem 16.1.1

P $\subseteq$ NP.

Another complexity class which come associated with NP is coNP.

$$\text{coNP} := \overline{\text{NP}}$$

i.e. the complement set of NP.

Observation 16.1. $P = \overline{P}$

Theorem 16.1.2

P $\subseteq$ coNP

Apart from these two we don't know any relation between NP and coNP whether they are equal or not.

16.2 Reductions

Question 16.2

What does it mean for a problem to be at least as hard as another?

To relate hardness of one problem to another we introduce the notion of reductions. There are many reductions. We will only focus on polynomial-time many-one karp reduction.

Definition 16.2.1: Many-One Karp Reduction

$L_1, L_2 \subseteq \Sigma^*$ are two languages. L_1 is reducible to L_2 under polynomial time many-one karp reduction if and only if there exists a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ such that $\forall\, x \in \Sigma^*$

$$x \in L_1 \iff f(x) \in L_2$$

and we denote it by $L_1 \leq_m^{\text{poly}} L_2$.

We call a language L to be NP-hard if for every language $L' \in \text{NP}$, $L' \leq_m^{\text{poly}} L$. And L is called NP-complete if $L \in \text{NP}$ and L is NP-hard.

Theorem 16.2.1 Cook's Theorem

3SAT is NP-complete.

Corollary 16.2.2

$\overline{\text{SAT}}$ is coNP-complete.

16.3 Some other NP-complete Languages

We will now show 3 other problems which are also NP-complete. We will show the following three problems to be NP-complete

- IndSet $:= \{(G, k) \mid \text{Graph } G \text{ has an independent set of size at least } k\}$
- VC $:= \{(G, k) \mid \text{Graph } G \text{ has a vertex cover of size at least } k\}$
- SubsetSum $:= \left\{(s_1, \ldots, s_t, T) \mid \exists\, X \subseteq [t], \sum_{i \in X} s_i = T\right\}$

Theorem 16.3.1

IndSet is NP-complete.

Proof: It is natural to see that IndSet $\in$ NP. Furthermore, we will show a reduction from 3SAT to IndSet. On the input of ϕ of 3SAT we want to find a (G, k) instance such that

$$\phi \text{ is satisfiable} \iff G \text{ has an independent set of size} \geq k$$

Let ϕ has m clauses on n variables. We build a graph G with $3m$ vertices with a triangle for each clause. Each vertex in a triangle corresponds to a literal. Add edge between x_i and $\overline{x}_i$ for all variables x_i.

Now with this construction we have ensured that for any variable x_i if the literal x_i is in the independent set then $\overline{x}_i$ is not in the independent set and vice versa. For each clause one vertex from each triangle is in the independent set. So the target independent set size is of size n.

Now if there is a satisfying assignment for ϕ then we can pick the corresponding vertices representing the literals which are set true and this will constitute an independent set. Similarly, if there is an independent set of size n in G then for each variable we have picked only one literal and from each triangle we have picked only one, so this corresponds to a satisfying assignment. ■

Theorem 16.3.2

VC is NP-complete.

Proof: It is natural to see that VC $\in$ NP. We will show a reduction from IndSet to VC for NP-hardness of VC. Notice that for any $S \subseteq V$, S is a vertex cover in G if and only if $V \setminus S$ is an independent set in G. Therefore, from the input (G, K) we create the $(G, n-k)$ and this way we found a bijection between independent sets and vertex cover. Hence, VC is NP-complete. ■

Theorem 16.3.3

SubsetSum is NP-complete.

Proof: Again it is very easy to see that SubsetSum $\in$ NP. Like IndSet for this problem we will show a reduction from 3SAT. Let we are given a boolean formula ϕ with n variables and m clauses.

Now each s_i, T are given by $n+m$ long integer. First n positions are indexed by variables and last m positions are indexed by the clauses. Each variable x_i corresponds to 2 integers, s_{x_i} and $s_{\overline{x}_i}$, one for each literal. For each literal x_i, s_{x_i} defined as the number which has 1 at the position of corresponding variable and 1's at the position of clauses in which that literal is present. Now each clause c_i corresponds to 2 integers, s_{c_i}, $s_{c_i'}$. Both s_{c_i}, $s_{c_i'}$ has a 1 in the corresponding clause position. Now T is defined to be the integer where it has 1's in first n positions and 3's in the last m positions.

Now notice if there is a satisfying assignment then we pick those numbers which corresponds to the literals which are set to be two. Their sum matches with the first n positions of T. Now for the last m bits we pick the necessary number of clause numbers to adjust. Similarly, if there is a subset sum then we set the corresponding literals to be true. Since the first n positions of T are 1 all the variables are assigned to some value. Hence, we get SubsetSum is NP-complete. ■

Chapter 17

Bibliography

[Ide16] Martin Idel. A review of matrix scaling and Sinkhorn's normal form for matrices and positive maps. *arXiv preprint*, 2016.

[LSW98] Nathan Linial, Alex Samorodnitsky, and Avi Wigderson. A deterministic strongly polynomial algorithm for matrix scaling and approximate permanents. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing - STOC '98*, STOC '98, pages 644–652. ACM Press, 1998.