CSS.201.1 Algorithms

Instructor: Umang Bhaskar

TIFR 2024, Aug-Nov

Scribe: Soham Chatterjee

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

Contents

Chapter 1 Greedy Algorithm — Page 3

1.1 Maximal Matching 3
1.2 Huffman Encoding 5
1.2.1 Optimal Binary Encoding Tree Properties 5
1.2.2 Algorithm 7
1.3 Matroids 8
1.3.1 Examples of Matroid 9
1.3.2 Finding Max Weight Base 11
1.3.3 Job Selection with Penalties 12

Chapter 2 Kruskal Algorithm with Data Structure — Page 14

2.1 Kruskal Algorithm 14
2.2 Data Structure 1: Linear Array 16
2.3 Data Structure 2: Left Child Right Siblings Tree 16
2.3.1 Construction 16
2.3.2 LCRS-Union Function 17
2.3.3 Amortized analysis of LCRS-Union 18
2.4 Data Structure 3: Union Find 18
2.4.1 Analyzing the Union-Find Data-Structure 18

Chapter 1

Greedy Algorithm

1.1 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 1.1.1: 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 1.1.2: 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 1.1.3: 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 1: Maximal-Matching

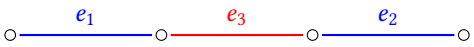
Input: Graph $G = (V, E)$
Output: Maximal Matching $M \subseteq E$ of G

```
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 1.1

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 1.2

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

This brings us to the following result:

Theorem 1.1.1

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: 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 since $M^\star$ is maximal matching, $e \in M^\star$. Then $e \notin M$. That means $e, e_1, e_2 \notin M$ which is not possible since M is also a maximal matching. 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 $|M| \geq \frac{1}{2}|M^\star|$.

■

1.2 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 $\text{COST}(b) = \sum_{i=1}^{n} f_i \cdot \text{LEN}(b(a_i))$ 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 $p(a_i) := \frac{f_i}{\sum_{i=1}^{n} f_i}$. So the we can see the updated cost function

$$\text{COST}(b) = \sum_{i=1}^{n} p(a_i) \cdot \text{LEN}(b(a_i))$$

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

1.2.1 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 1.2.1
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

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

Since $p(x) < p(y)$ and $h_1 < h_2$ we have $\text{COST}(T) - \text{COST}(T') > 0$. 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 1.2.2

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 1.2.1 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 1.2.3

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 $h_x, hy \geq h$. 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''.

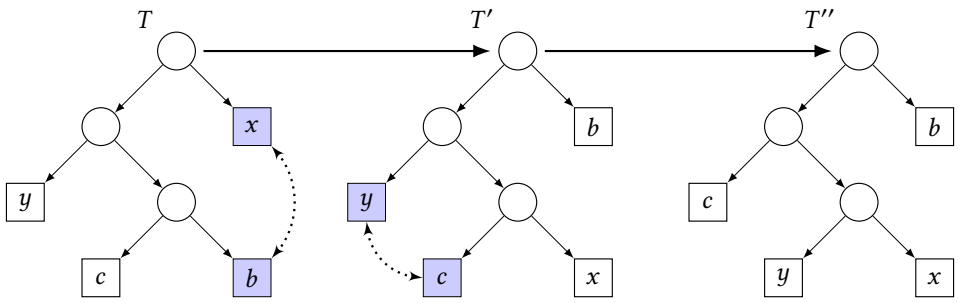


Figure 1.1: Showing that the lowest probability nodes are siblings at the tree's lowest level.

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 1.2.2 and Lemma 1.2.3 we have that the least frequent element and the second least frequent element are siblings and they have the maximum height.

Observation. *The cost of the trees T_n and T_{n-1} differ only by the fixed term $p(z) = p(x) + p(y)$ which does not depend on the tree's structure. Therefore minimizing the cost for T_n is equivalent to minimizing the cost of T_{n-1}.*

Theorem 1.2.4

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

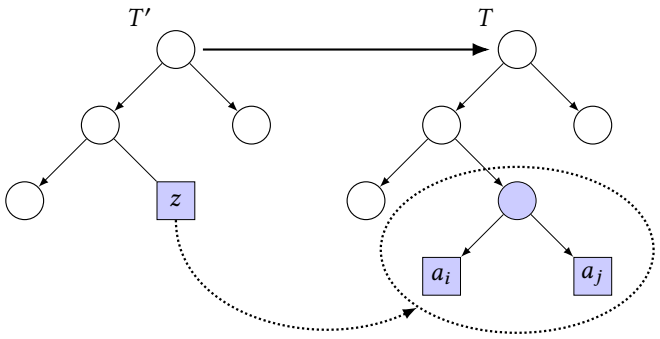
$$\begin{array}{cccccccc} a_1, & a_2, & \cdots, & a_i, & \cdots, & a_j, & \cdots, & a_n \\ p(a_1), & p(a_2), & \cdots, & p(a_i), & \cdots, & p(a_j), & \cdots, & p(a_n) \end{array} \quad \text{with probabilities}$$

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 1.2.3. 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. ■

1.2.2 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 2: 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 $\text{Cost}(b) = \sum_{i=1}^{n} p(a_i) \cdot \text{Len}(b(a_i))$.

```
begin
    n ⟵ |A|
    Q ⟵ Min-Priority Queue
    for x ∈ A do
        Insert(Q, x)
    for i = 1, ... n − 1 do
        z ⟵ New internal tree node
        x ⟵ Extract-Min(Q), y ⟵ Extract-Min(Q)
        left[z] ⟵ x, right[z] ⟵ y
        p(z) ⟵ p(x) + p(y)
        Insert(Q, z)
    return Last element left in Q as root
```

Theorem 1.2.5 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 1.2.3 and Lemma 1.2.1 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 1.2.4 we have T_n is also optimal. ■

1.3 Matroids

Definition 1.3.1: 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 1.3.1

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 $S \in I$: $\{e \in E : rank(S) = rank(S + e)\}$

1.3.1 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 1.3.2**
> $M = (E, \mathcal{I})$ 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 1.3.3**
> $M = (E, \mathcal{I})$ 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, $\mathscr{L} = \{L_1, \ldots, L_l\}$ such that $\forall\, i, j \in [l]$, 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 1.3.4**
> $M = (E, \mathcal{I})$ 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 1.3.5**
> $M = (E, \mathcal{I})$ 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 1.3.6**
> $M = (E, \mathcal{I})$ 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.

1.3.2 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 3: 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 1.3.7
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 1.3.1
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 $B' = B_{i-1} + i - j$.

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

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. ■

Claim 1.3.2
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 t_{i-1}. 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 assumes $wt(\hat{T}_i) > wt(T_i)$. 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. ■

1.3.3 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 lets define what is a schedule and what is a feasible schedule:

Definition 1.3.2: 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 $t \in \mathbb{Z}_+$, define $N_t(S) = \{j \in S \colon d_j \leq t\}$. Then we have the following lemma:

Lemma 1.3.8

The following are equivalent:

① S is feasible

② $\forall\, t \in \mathbb{Z}_t$, $|N_t(S)| \leq t$

③ The schedule that orders jobs 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 finished. Hence contradiction.

2 $\implies$ 3: The schedule orders the jobs by deadline. We 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 they all can be completed within time t in any order. Therefore if we complete the jobs with deadline $< t$ first then also 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 1.3.9

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

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

Then take $\mathcal{I}' = \{S \subseteq J : |N_t(S)| \leq t \ \forall t \in [D]\}$. By Lemma 1.3.4 $M = (J, \mathcal{I}')$ is a laminar matroid. And by Lemma 1.3.8 $\mathcal{I}'$ 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 S 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.

■

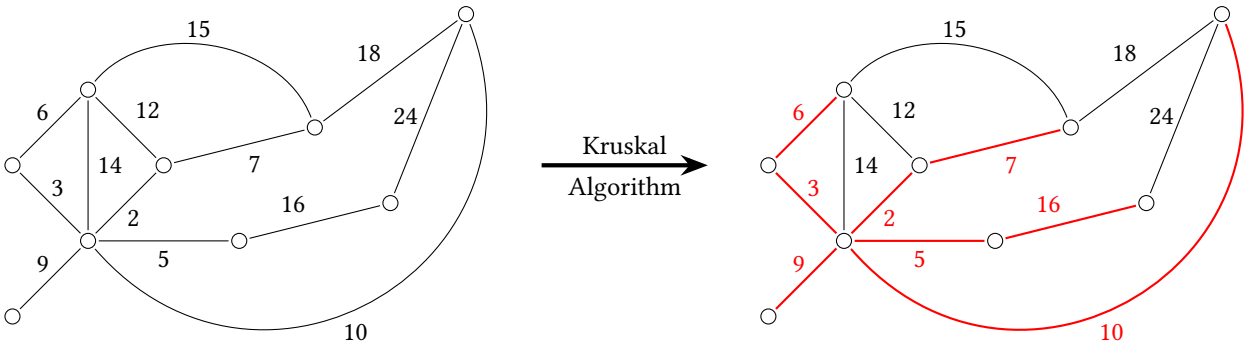
Chapter 2

Kruskal Algorithm with Data Structure

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 algorithm which gives a greedy solution to the problem. Then we will discuss the data structure that we can use to implement the Kruskal algorithm efficiently.

2.1 Kruskal Algorithm



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

Definition 2.1.1: 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 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 add 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 1.3.5 that the set of collection of acyclic sets in

Algorithm 4: KRUSKAL 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      if G is not connected then
3          return None                                   // Use DFS or BFS
4      T ⟵ ∅
5      Sort the edges in E in non-decreasing order of their weights so that w(e_1) ≤ w(e_2) ≤ ··· ≤ w(e_m)
6      for i = 1, ..., m do
7          Let e_i = (u, v)
8          if T ∪ {e_i} is acyclic then
9              T ⟵ T ∪ {e_i}
10         if |T| = |V| − 1 then
11             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 1.3.2. 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 2.1

hat 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 algorithm using proper data structure: The Kruskal Algo-

Algorithm 5: KRUSKAL 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      if G is not connected then
3          return None                                   // Use DFS or BFS
4      T ⟵ ∅
5      Sort the edges in E in non-decreasing order of their weights so that w(e_1) ≤ w(e_2) ≤ ··· ≤ w(e_m)
6      for i = 1, ..., m do
7          Let e_i = (u, v)
8          if FIND(u) ≠ FIND(v) then
9              T ⟵ T ∪ {e_i}
10             UNION(u, v)
11         if |T| = |V| − 1 then
12             return T
```

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

2.2 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 6: 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 algorithm using this data structure is $m \cdot O(1) + n \cdot O(n) = O(m + n^2) = O(n^2)$.

2.3 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).

2.3.1 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 immagine 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.

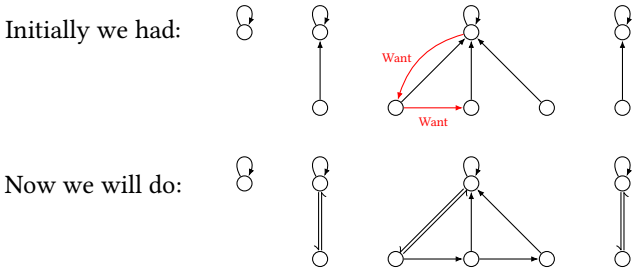


Figure 2.1: Left Child Right Sibling

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.

2.3.2 LCRS-UNION Function

For the LCRS-UNION function we do the following

Algorithm 7: 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
15         vp.left ⟵ up.left
16         vp.left ⟵ up
17         vp.size ⟵ vp.size + up.size
18     else
19         vp.right ⟵ v
20         x ⟵ vp
21         while x.right == None do
22             x.parent ⟵ up, x ⟵ x.right
23         x.right ⟵ up.left
24         up.left ⟵ vp.left
25         up.left ⟵ vp
26         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

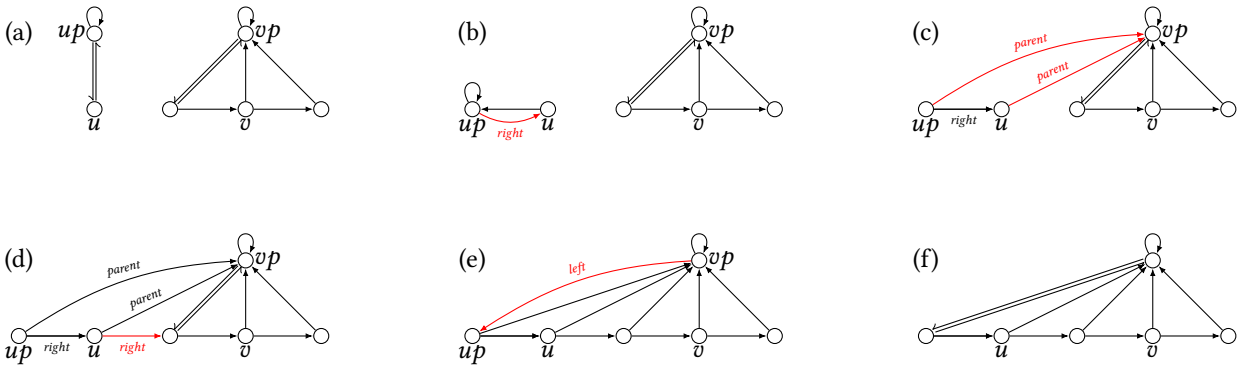


Figure 2.2: A run of LCRS-UNION(u, v)

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

2.3.3 Amortized analysis of LCRS-Union

2.4 Data Structure 3: Union Find

2.4.1 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 (reversed) tree.

Lemma 2.4.1

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 2.4.2

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 2.4.3

Ranks are monotonically increasing in the reversed 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 $\text{rank}(u) < \text{rank}(v)$. ■

Lemma 2.4.4

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

Corollary 2.4.5

For all vertices v, $v.rank \leq \lfloor \log n \rfloor$

Corollary 2.4.6

Height of any tree $\leq \lfloor \log_2 n \rfloor$

Lemma 2.4.7

The number of nodes that get assigned rank k throughout the execution of the Union-Find data-structure is at most $\frac{n}{2^k}$.

Define $N(r)$ = #vertices with rank at least k. Then by the above lemma we have $N(r) \leq \frac{n}{2^k}$.

Lemma 2.4.8

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 $O((n+m)\log^* n)$

Lemma 2.4.9

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

Lemma 2.4.10

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. $\text{INDEX}_B(x) = i$) before $x.parent$ is no longer in the i^{th} block. That is $\text{INDEX}_B(x.parent) > i$.

Lemma 2.4.11

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

Lemma 2.4.12

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

Theorem 2.4.13

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

Theorem 2.4.14

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