CHAPTER BASED LECTURE NOTES

CO 353: COMPUTATIONAL DISCRETE OPTIMIZATION

Prepared by: CALVIN KENT
www.student.math.uwaterloo.ca/~c2kent/

Instructor: RICARDO FUKASAWA
Term: WINTER 2020

Last revised: 7 March 2020

Page

Table of Contents

Preface and Notation

This PDF document includes lecture notes for CO 353 - Computational Discrete Optimization taught by Ricardo FUKASAWA in Winter 2020.

For any questions contact me at `c2kent(at)uwaterloo(dot)ca`.
Thanks to Taric Ali and Caleb Nicholas Chappell for providing me the notes for the classes I missed.

Notation

Course outline and relevant info: https://piazza.com/uwaterloo.ca/winter2020/co353

Throughout the course and the notes, unless otherwise is explicitly stated, we adopt the following conventions and notations.

- Algorithms use the same counter as definitions, theorems, examples etc.
- The university logo is used as a place holder.

Calvin KENT

Chapter 1 – Algorithm Runtime, Big-O Notn. and Graph Theory

1.1 Algorithm Running Time

We want to formally see which algorithms are more efficient. To compare algorithms, we measure runtime (number of steps) of an algorithm as a function of the input.

Definition 1.1.1: ***Size*** of an input is the number of bits needed to encode it. ◁

Example 1.1.2: Consider a list of positive integers $a_1, a_2, \ldots, a_n$ where $a_i \in \mathbb{Z}^+$ for $i = 1, \ldots, n$. For each integer a_i, we need $\lceil \log a_i \rceil$ bits. Hence, the number of bits needed to represent the input is $\sum_{i=1}^{n} \lceil \log a_i \rceil$. ◁

1.1.1 Finding an Estimate of Runtime (big-O Notation)

Definition 1.1.3: Let $f, g : \mathbb{Z}^+ \to \mathbb{R}^+$ be two functions. We say f is $O(g)$, read as ***big-O of*** g, if $\exists\, c \in \mathbb{R}^+$ and $\exists\, n' \in \mathbb{Z}^+$ such that $\forall\, n \geq n'$ we have $f(n) \leq cg(n)$. ◁

Example 1.1.4: Let $f(n) = 5 \log n$ and show $f(n)$ is $O(n)$. Let $g(n) = n$ and $c = 5$. Since for all $n \geq 1$ we have $\log n < n$. Then, for $n \geq 1$ we have $f(n) \leq 5n$. So, $f(n) \leq cg(n)$. Hence, $f(n)$ is $O(g) = O(n)$. ◁

Example 1.1.5: Let $f(n) = 2n^2 + 3n \log n$ and show $f(n)$ is $O(n^2)$.We have

$$f(n) = 2n^2 + 3n \log n \leq 5n^2,$$

so it follows that $f(n)$ is $O(n^2)$. ◁

Remark 1.1.6: We make the following remarks for polynomials, logarithms and exponentials.

1. $\sum_{k=0}^{d} \alpha_k n^k$ where $\alpha_k \in \mathbb{R}$ and $\alpha_d > 0$ is $O(n^d)$. i.e. polynomials are dominated by their leading term.
2. $\log_b n$ where $b > 1$ and $c > 0$ is $O(n^c)$. i.e. logarithms are dominated by polynomials.
   a. We recall logarithm rules. We have

   $$\log_b n = \frac{\log_2 n}{\log_2 b} = \left(\frac{1}{\log_2 b} \right) \log_2 n.$$

   So big-O does not get affected by log base. In this course we will use log base 2.
3. n^c where $b > 1$ and $c > 0$ is $O(b^n)$. i.e. polynomials are dominated by exponentials. ◁

Theorem 1.1.7 (Properties of big-O)**:** Let $f, g, h, f_i, g_i : \mathbb{Z}^+ \to \mathbb{R}^+$ be functions for $i = 1, \ldots, m$.

1. (Transitivity) If f is $O(g)$ and g is $O(h)$ then f is $O(h)$.

(2) If f, g are $O(h)$ then $f + g$ is $O(h)$.

(3) If f_i is $O(g_i)$ for all $i = 1, \ldots, m$, then $f_1 \cdots f_m$ is $O(g_1 \cdots g_m)$.

Proof: Exercise. $\triangleleft$

Definition 1.1.8: Let $f, g : \mathbb{Z}^+ \to \mathbb{R}^+$ be two functions. We say f is $\Omega(g)$, read as ***omega of*** g (or big-omega of g), if $\exists\, c \in \mathbb{R}^+$ and $\exists\, n' \in \mathbb{Z}^+$ such that $\forall\, n \geq n'$, we have $f(n) \geq cg(n)$.

We say f is $\Theta(g)$, read as ***theta of*** g (or big-theta of g), if f is $O(g)$ and $\Omega(g)$. $\triangleleft$

Definition 1.1.9: Operations involving a combination of basic arithmetic $(+, -, \times, \div)$ operations, comparisons, if-then-else statements and assignments are called ***basic operations*** (sometimes referred as *elementary operations*).

We say an algorithm has ***runtime*** $p(n)$ if the algorithm executes $p(n)$ basic operations on inputs of size n. $\triangleleft$

1.1.2 Arithmetic Model

Definition 1.1.10: Consider an algorithm with runtime of $p(n)$. If $p(n)$ is $O(g)$ for some polynomial function $g(n)$, then the algorithm said to be in ***polynomial time***. In short, we refer these algorithms as ***polytime algorithms*** and they are also called ***efficient algorithms***. $\triangleleft$

Remark 1.1.11: In practice, big-O does not always shows which algorithms are more efficient.

(1) Big-O analysis provides an upper bound (i.e. worst case) for an algorithm. For example, in linear programming simplex algorithm is widely used and considered to be efficient but it has big-O of exponential.

(2) Big-O hides constants. Depending on the input, an exponential algorithm can be more efficient that an polytime algorithm. For example, for small numbers for n, the exponential algorithm with runtime 1.0001^n is more efficient than the polytime algorithm with runtime $10^{20}n^{100}$. $\triangleleft$

We recall Example 1.1.2. Given integers $a_1, \ldots, a_n \in \mathbb{Z}^+$, we want to find an efficient algorithm that sorts these integers. We have the size of input as $\sum_{i=1}^{n} \lceil \log_2 a_i \rceil$. It becomes tricky to express runtime as a function of input size. So we pick some parameters that are at most a polynomial of actual input size.

If the input size is k, then we want to pick parameters n so that we have n is $\text{poly}(k)$.

Example 1.1.12: We refer back to our example. We have

- n as the number of integers,
- $a_{\max}$ as the largest integer (does not have to be unique unless specified).

So, if we find algorithms in time $\text{poly}(n, \log a_{\max})$, then time is polynomial in input size. Now, consider the following example:

Given distinct integers $a_1, \ldots, a_n \in \mathbb{Z}^+$ find the largest integer. We know the algorithm is polytime in terms of $n, \log a_{\max}$. Consider the following algorithm.

Algorithm 1.1.13: Finding largest integer

Input : $a_1, \ldots, a_n$ (distinct)
Output: $a_{\max}$ (such that $a_{\max} \geq a_i$ for $i = 1, \ldots, n$)

1 largest $\leftarrow a_1$
2 **for** $i = 1, \ldots, n$ **do**
3   **if** $a_i >$largest **then** $\}O(1)$ $\Big\}O(n)$
4   largest $\leftarrow a_i$ $\}O(1)$
5 **return** largest

Note that the operations in line 3 and 4 both take $O(1)$ time each since they are basic operations. The for loop in line 2 goes for n times. Hence, the algorithm is in $O(n)$. $\triangleleft$

Example 1.1.14: Let $S_1, \ldots, S_k$ be a partition of the set $S = \{1, \ldots, n\}$ where $\bigcup_{i=1}^{k} S_i = S$ and $S_i \cap S_j = \varnothing$ for all $i \neq j$ and $j, \ell \in \{1, \ldots, n\}$. We want to merge sets containing j and ℓ. Assume we have Label[t] that says which set t is on for all $t \in \{1, \ldots, n\}$. So we can consider Label and Merge as functions where Label: $\{1, \ldots, n\} \to \{1, \ldots, k\}$, where Merge is defined as follows.

Algorithm 1.1.15: Algo for partition problem

Input : $a_1, \ldots, a_n$ (distinct)
Output: $a_{\max}$ (such that $a_{\max} \geq a_i$ for $i = 1, \ldots, n$)

1 **if** `Label` $[j] \neq$`Label` $[\ell]$ **then** $\}O(1)$
2     $temp \leftarrow LABEL[\ell]$ $\}O(1)$
3   **for** $i = 1, \ldots, n$ **do**
4   **if** `Label` $=$**temp then** $\}O(1)$ $\Big\}O(n)$
5   `Label` $[i] \leftarrow$`Label` $[j]$ $\}O(1)$

Similarly to algorithm 1.1.13, lines 2,3,5 and 6 have comparison and assignment operations so they are in $O(1)$. The for loop in line 3 goes n times so it is in $O(n)$. Hence the algorithm is in $O(n)$. $\triangleleft$

1.2 Graph Theory

Definition 1.2.1: A ***graph*** $G = (V, E)$ is a tuple of vertices $v \in V$, and edges $e \in E$. $\triangleleft$

Example 1.2.2: An example of a graph $G = (V, E)$ where $V = \{a, b, c, d\}$ and $E = \{\{a, b\}, \{b, d\}, \{a, c\}\}$.

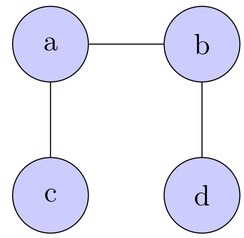


Figure 1.2.1: Simple graph.

◁

Remark 1.2.3: For convenience, we omit brackets when writing edges. i.e. $ab = \{a, b\}$.

In this course we will assume graphs are simple. i.e. no loops or parallel edges.

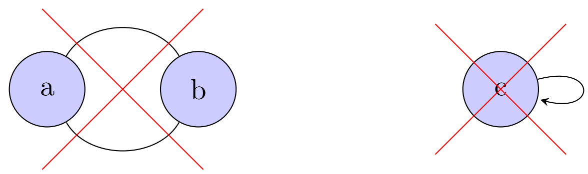


Figure 1.2.2: No parallel edges or loops.

◁

Definition 1.2.4: We provide the following definitions:

1. If $uv \in E$ we say u, v are ***endpoints*** of edge uv and u is ***adjacent*** to v.
2. A ***walk*** in $G = (V, E)$ is a sequence $v_1, \dots, v_k$ where $v_i \in V$, $v_i v_{i+1} \in E$ for all $i = 1, \dots, k-1$. Recall Example 1.2.2. We have a, b, d and b, a, b, d are walks but d, c, a is not a walk.
3. A *walk* $v_1, \dots, v_k$ is a ***path*** if for all distinct $i, j = 1, \dots, k-1$ we have $v_i \neq v_j$. i.e. if every edge and vertex in a path is traversed exactly once. For example a, b, d is a path but b, a, b, d is not a path.
4. A *walk* is ***closed*** if $v_1 = v_k$.
5. A *closed walk* is a ***cycle*** if $k \geq 4$. A graph with that contains at least one cycle is called ***cyclic***, graphs with no cycles are called ***acyclic***.

   Example: In the figure below, $1, 2, 3, 1$ is a cycle but $1, 2, 1$ is not.

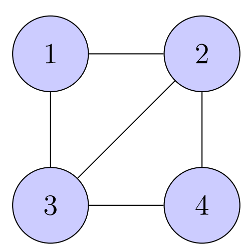


Figure 1.2.3: Simple graph with cycle.

◀

6 A graph $G = (V, E)$ is ***connected*** if for all $u, v \in V$, there exists a u-v path.

7 Let $G = (V, E)$ and $H = (U, F)$ be graphs. We say H is a ***subgraph*** of G if $U \subseteq V$ and $F \subseteq E$.

Example: Here H_1, H_2 and H_3 are subgraphs of G.

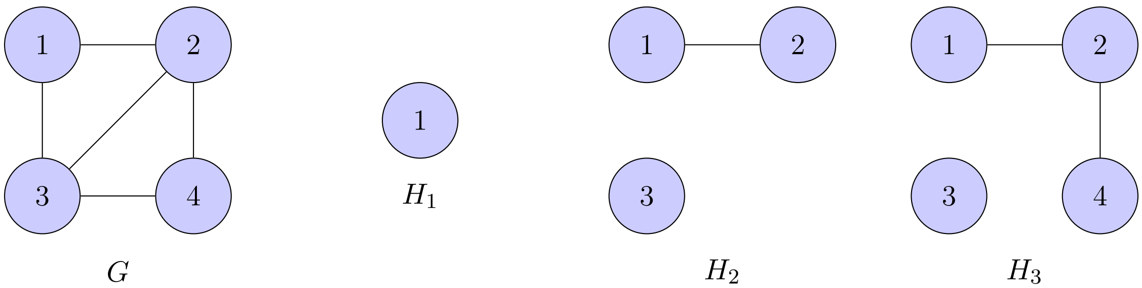


Figure 1.2.4: Some subgraphs of G.

◀

8 Let $G = (V, E)$ be a graph and let $S \subseteq V$. The graph $G[S] = (S, E(S))$ is called the ***subgraph induced by*** S where $E(S) = \{e \in E \mid \text{both endpoints in } S\}$.

Example: Let $G = (V, E)$ where $V = \{1, 2, 3, 4\}$ as below and let $S = \{2, 3, 4\}$. We have $G[S]$ as shown below.

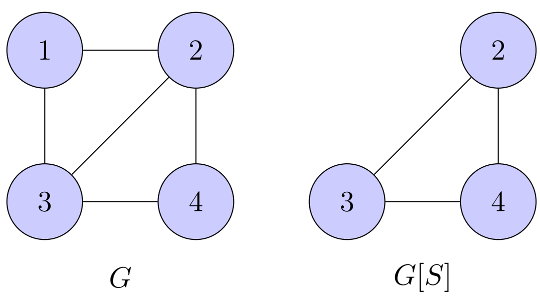


Figure 1.2.5: G and $G[S]$.

◀

9 Let $G = (V, E)$ be a graph. A ***connected component*** of G is a *maximal* induced subgraph of G, where ***maximal*** means adding another element violates its property. For example, if $G[S]$ is a connected component of G, then $G[S]$ is a connected subgraph induced by S but $G[S \cup \{v\}]$ is not connected for $v \in V \setminus S$. ◁

Algorithms Revisited

Example 1.2.5: Given a graph $G = (V, E)$, $u, v \in V$ where $|V| = n$ and $|E| = m$, determine if u, v are in same connected component.

This can be done in $O(n+m)$ with DFS. Assume G is given as $V = \{1, \dots, n\}$ and $E = \{e_1, \dots, e_m\}$. Similarly to partition problem we discussed in Example 1.1.14 we have the following algorithm.

Algorithm 1.2.6: Determining if vertices are in same connected component

```
Input : n, e_1, ..., e_m, u, v
1 for i = 1, ..., n do
2     Label[i] ← i
3 for i = 1, ..., m do                    // merging connected components as we go
4     let x, y be endpoints of e_i
5     Merge (Label, x, y)
6 if Label[u] = Label[v] then
7     return YES
8 else
9     return NO
```

The first for loop in line 1 is in $O(n)$. The merge function in line 5 is in $O(n)$, so the for loop in line 3 is in $O(mn)$. The if-then-else statement in line 6 is in $O(1)$. Hence, the algorithm is in $O(mn)$.

Note that this can be done better but at this point in the course we want easy to analyze examples to get a feel for big-O. Our focus right now is practicing big-O, not designing algorithms. ◁

1.2.1 Minimum Spanning Tree (MST)

Definition 1.2.7: Given $G = (V, E)$, a subgraph T of G is a ***tree*** if it is connected and acyclic. A tree T of G is ***spanning tree*** of G is $V(T) = V$. ◁

Example 1.2.8: Here both T_1 and T_2 are trees of G and T_2 is a spanning tree of G.

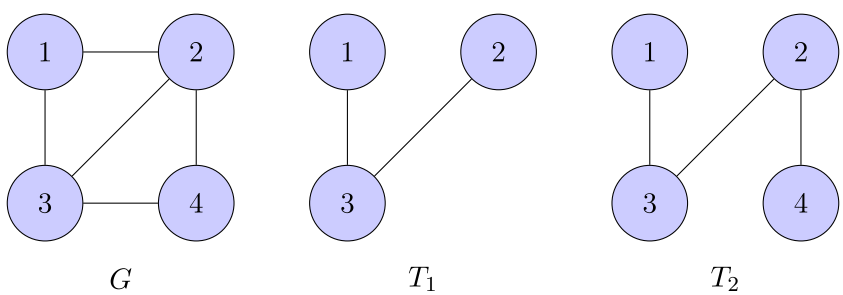


Figure 1.2.6: T_1 is a tree of G and T_2 is a spanning tree of G.

◁

1.2.1.1 MST Problem

Given a connected graph $G = (V, E)$ with edge costs $c_e \in \mathbb{Z}$ for all $e \in E$, we want to find a minimum spanning tree T of G that minimizes the cost function $c(T)$, defined by

$$c(T) \underset{\text{def}}{=} \sum_{e \in E(T)} c_e.$$

Theorem 1.2.9 (Properties of Spanning Trees)**:** Let T be a spanning tree of $G = (V, E)$. Then the following are true.

(1) For all $u, v \in V$, there exists a unique u-v path in T (call T_{uv}).

(2) T is minimally connected.

(3) T is maximally acyclic.

(4) T is spanning tree if and only if T is connected and has $n-1$ edges.

Proof:

(1) (Sketch) Suppose there exists distinct P_1, P_2 u-v paths where $P_1 = v_1, \ldots, v_k$ and $P_2 = w_1, \ldots, w_l$. So $v_1 = w_1 = u, v_k = w_l = v$. Then, there exists some t such that $v_t \neq w_k$ and $1 < t < k$. Since $v_k \in P_2$, there exists $t' > t$ for which $v_{t'} \in P_2$. Choose the smallest such t'. Let w_s be the corresponding vertex on P_2. Then, $v_{t-1}, v_t, \ldots, v_{t'} = w_s, \ldots, w_{k-1}$ is a closed walk that leads to a cycle. To illustrate this consider the following.

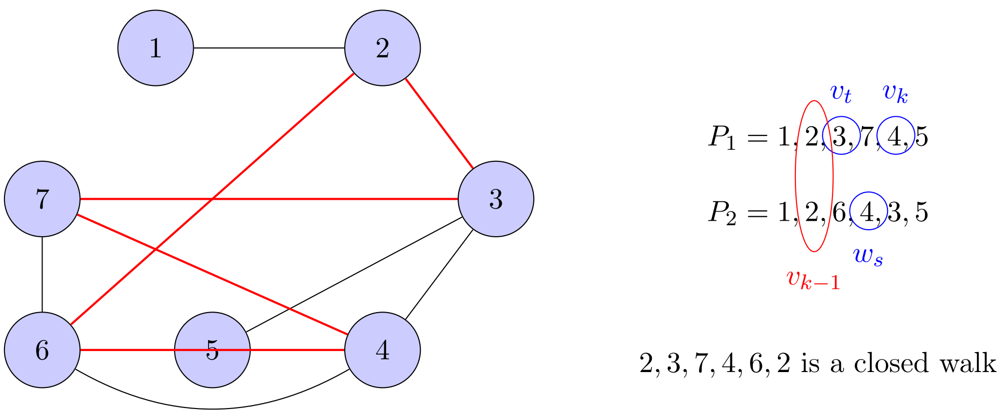


Figure 1.2.7: Closed walk is shown in red.

Note that if the closed walk is a, b, c, d, e, c, a then a, b, c, a is a cycle.

(2) Suppose $uv \in E(T)$ and $T - uv$ is connected. Note that $T - uv$ is the tree where the edge uv is removed, which is $T - uv = V(T) \cup E(T) \setminus \{uv\}$. Let $T' = T - uv$. T' has a u-v path T'_{uv} without the edge uv. Hence, if we attach the edge uv back to T'_{uv}, we get a cycle.

(3) Analogous to (b). Exercise.

(4) We recall and look at algorithm 1.2.6 to find if u, v are in same connected component.

Claim: Let $G = (V, E)$ where $|V| = n$. If $|E| < n - 1$ edges, then G is disconnected.

Proof: We see that algorithm 1.2.6 starts with n labels and graph is connected if at the end of the algorithm there is only one label remaining but for every edge the number of labels decreases by at most 1. Hence, if the graph has less than $n-1$ edges, then at the end there exists at least two distinct labels. Hence, graph is disconnected. ■

Claim: Let $G = (V, E)$ where $|V| = n$. If $|E| \geq n$ then G has a cycle.

Proof: Since $|E| \geq n$, then $\exists\ e = uv \in E$ for which the algorithm does not decrease number of labels. Hence, there exists a u-v path P that does not use uv. Hence, $P + uv$ is a cycle. ■

Hence, it follows that spanning trees are connected with $n-1$ edges. Reverse argument is also analogous (exercise)

□

Definition 1.2.10: Let $G = (V, E)$ and let $A \subseteq V$. We define $\delta_G(A)$ as the set of edges in G that only have one end point in A. That is, $\delta_G(A) \underset{\text{def}}{=} \{e \in E \mid |e \cap A| = 1\}$. This set is called ***the cut induced by (the vertices of) A in G.*** ◁

Example 1.2.11: Let $G = \{V, E\}$ where $V = \{1, 2, 3, 4, 5\}$ and let $A = \{1, 2, 3\} \subseteq V$. We have $\delta_G(A) = \{14, 24, 35\}$.

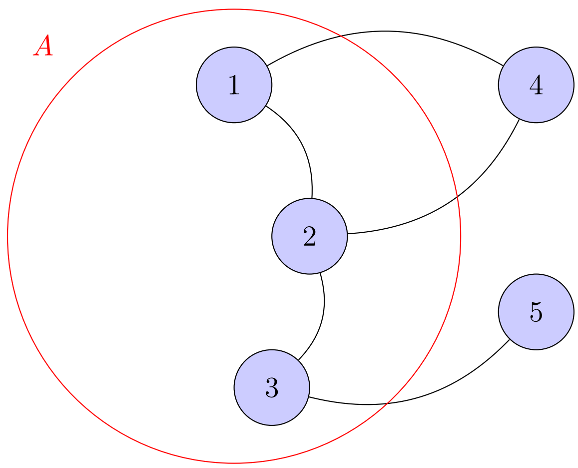


Figure 1.2.8: $A = \{1, 2, 3\} \subseteq V$

◁

Theorem 1.2.12: A graph $G = (V, E)$ is connected if and only if for all non-empty proper subsets of V we have $\delta_G(A) \neq \varnothing$. i.e. $\forall\ A \subsetneq V$ we have $A \neq \varnothing$ and $\delta_G(A) \neq \varnothing$.

Proof: Exercise. ◁

Theorem 1.2.13: Let $G = (V, E)$ and $c : E \to \mathbb{R}$ and T be a spanning tree of G. TFAE.

1. T is a minimum spanning tree of G.
2. For all $uv \in E \setminus E(T)$, $c_e \leq c_{uv}$ for all $e \in T_{uv}$.
3. Let $e \in E(T)$. If T_1, T_2 are two connected components of $T - e$, then e is a minimum cost edge in $\delta_G(T_1) = \delta_G(T_2)$.

Remark 1.2.14: Before starting the proof, we illustrate what we mean by part 2 and 3 in above theorem. Here we have $G = (V, E)$ where $E = \{e, e_i, E_j \mid i = 1, \ldots, 7 \text{ and } j = 1, 2\}$. $T = (V, E(T))$ is a spanning tree of G where $E(T) = E \setminus \{E_1, E_2\}$ (shown in zigzags) and T_1, T_2 are two connected components of $T - e$.

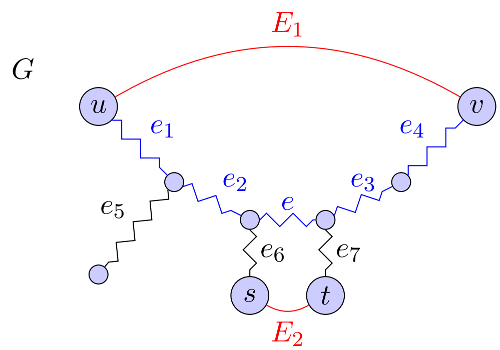


Figure 1.2.9: T_{uv} is shown in blue.

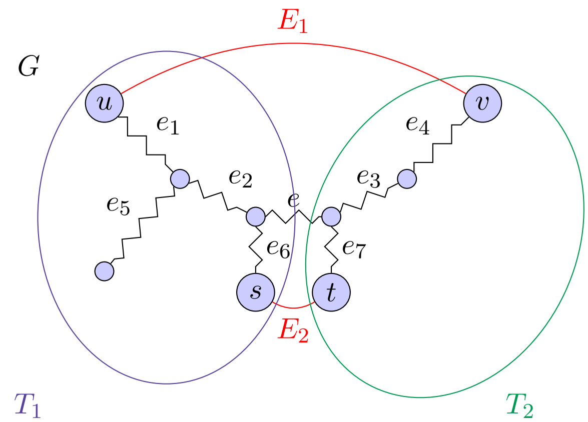


Figure 1.2.10: T_1 and T_2 are two connected components of $T - e$.

Part 2 of the theorem states that for $j = 1, 2$ we have $c_e \leq c_{E_j}$ and $c_{e_i} \leq c_{E_j}$ for $i = 1, \ldots, 5$. Part 3 of the theorem states that $c_e \leq c_{E_j}$ where $j = 1, 2$. $\triangleleft$

We continue the proof of Theorem 1.2.13.

Proof: We first show 1 $\implies$ 2. Suppose, for contradiction, $\exists\, e \in T_{uv}$ such that $c_e > c_{uv}$ for some $uv \in E \setminus E(T)$. Let $T' = T - e + uv$.

Claim: T' is a spanning tree of G.

Proof: Clearly $T' \subseteq G$ and $V(T') = V(T) = V$. Also, T' and T have same number of edges $(n-1)$. Hence, we only need to show T' is connected and acyclic. Since T is acyclic then after removing an edge e from T_{uv} we need to add at least two more edges to to create a cycle. Hence, T' is acyclic. Suppose, for contradiction, there exists a non-empty proper subset $A \subsetneq V$ such that $\delta_{T'}(A) = \varnothing$. Since $\delta_T(A)$ is connected, then $\delta_T(A) \neq \varnothing$. $T = T - uv + e$, then we must have $\delta_T(A) = e$. Then, $|\{u, v\} \cap A| = 1$ but this means $\delta_{T'}(A) \neq \varnothing$ which is a contradiction. Hence, for all non-empty proper subsets $A \subsetneq V$, we have $\delta_{T'} = \varnothing$. Hence, T' is connected. Thus, T' is a spanning tree of G. ■

Since both T' and T are spanning trees of G, then

$$c(T') = \sum_{e \in E(T)'} c_e = c(T) - c_e + c_{uv} < c(T).$$

But this means T cannot be MST which is a contradiction.

We now show 2 $\implies$ 3. Suppose, for contradiction, there exists $e \in E(T)$ such that for two connected components T_1 and T_2 of $T - e$, we have $c_e < c_{uv}$ for some $uv \in \delta_G(T_1)$. Note that since u and v are in different connected components of $T - e$ then $uv \notin E(T)$. But then $e \in T_{uv}$ which contradicts the hypothesis of 2.

Lastly, we show 3 $\implies$ 1. Suppose T satisfies 3. Let T^* be a MST maximizing $k = |E(T) \cap E(T^*)|$. If $k = n - 1$ we are done. Otherwise, there exists $uv \in E(T) \setminus E(T^*)$. Let T_1 and T_2 be two components of $T - uv$. Then, there exists $e \in \delta_{T^*}(T_1)$ such that $e \in T^*_{uv}$. Since, $uv \notin E(T^*)$, then $e \neq uv$.

From the hypothesis of 3, we have $c_{uv} \leq c_e$. Let $T' = T^* - e + uv$ then $|E(T')| = n - 1$. From the proof of 1 $\implies$ 2, we have that T' is also connected and T' is a spanning tree of G. Hence,

$$c(T') = c(T^*) - c_e + c_{uv} \leq c(T^*).$$

Then $c(T')$ is a MST but this gives us $|E(T) \cap E(T')| > |E(T) \cap E(T^*)|$ which contradicts the choice of T^*. $\square$

Chapter 2 – Greedy Algorithms and Matroids

2.1 Kruskal's Algorithm

Kruskal's algorithm takes a connected graph $G = (V, E)$ and edge costs as inputs and gives a MST of G as output. It operates as follows.

Algorithm 2.1.1: Kruskal's algorithm (idea)

Input : $G = (V, E)$ (connected), $c : E \to \mathbb{R}$
Output: MST T.
Init : $T = (V, \varnothing)$.

while T *is not a spanning tree* **do**
    Let e be the cheapest edge whose end points are different connected components of T.
    Add e to T.
return T

Example 2.1.2: For $G = (V, E)$ where $V = \{1, 2, 3, 4\}$ with edge costs below, Kruskal's algorithm adds edges to $V(T) = V(G)$ in the order shown.

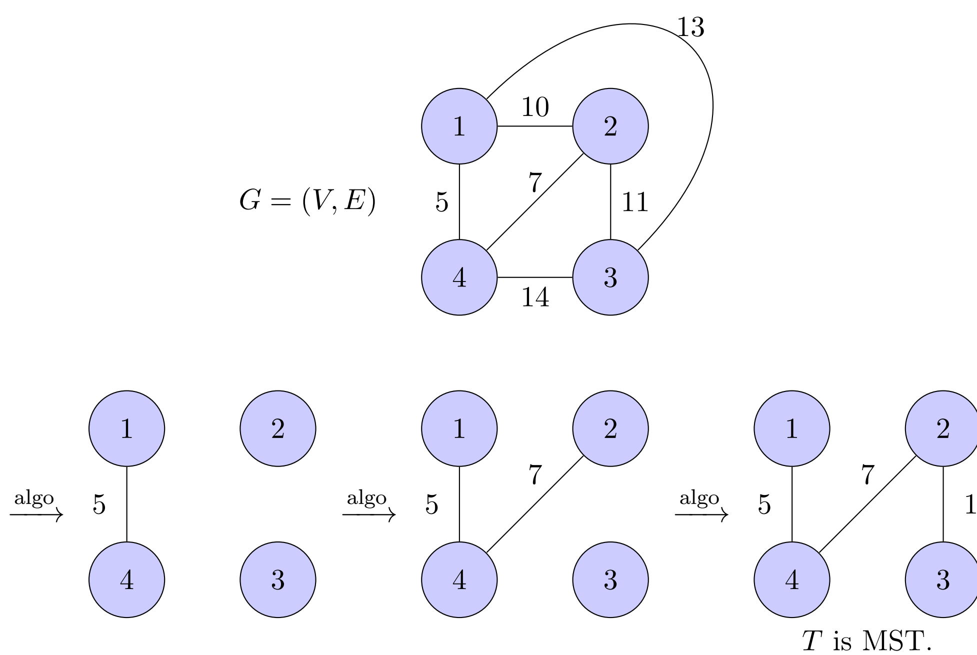


Figure 2.1.1: Kruskal's algorithm on $G = (V, E)$

◁

Remark 2.1.3: We make the following remarks about Kruskal's algorithm.

① Algorithm returns a spanning tree T.

② Algorithm doesn't get stuck since there always exists an edge e with minimum cost during the while loop. If such e didn't exist, then we can pick a connected component T_1 of T and have $\delta_G(V(T_1)) = \varnothing$. By Theorem 1.2.12 this means G is connected.

③ T has no cycles since every new added edge e connects T to a different connected component.

④ At every while loop iteration, the number of connected components of T goes down by one, so the algorithm terminates. ◁

2.1.1 Implementation of Kruskal's Algorithm

For $G = (V, E)$ where $|V| = n$ and $E = \{e_1, \dots, e_m\}$ with each edge e_i having cost c_i, we implement Kruskal's algorithm in the following way. Note that it is possible implement it more efficiently.

Algorithm 2.1.4: Kruskal's algorithm

Input : $n, e_1, \dots, e_m, c_1, \dots, c_m$
Output: MST T.
Init : $T = (V, \varnothing)$.

1 Reorder edges so that $c_1 \leq \cdots \leq c_m$ $\quad \} O(m \log m)$
2 **for** $j = 1, \dots, n$ **do** $\quad \} O(n)$
3 $\quad$ `Label`$[j] \leftarrow j$ $\quad \} O(1)$
4 **for** $i = 1, \dots, m$ **do** $\quad \} O(mn)$
5 $\quad$ Let uv be endpoints of e_i $\quad \} O(1)$
6 $\quad$ **if** `Label`$[u] \neq$ `Label`$[v]$ **then** $\quad \} O(1)$
7 $\quad\quad$ Add e_i to T $\quad \} O(1)$
8 $\quad\quad$ `Merge(Label,`u`,`v`)` $\quad \} O(n)$
9 **return** T

Since ordering m elements takes $m \log m$, line 1 is in $O(m \log m)$. The overall algorithm is in $O(mn)$ due to the for loop in line 4 with the Merge function in line 8. We can improve the algorithm by bringing the complexity of line 6 higher and lowering the complexity of line 8 (with clever use of data structures) so that both line 6 and line 8 have around same complexity $O(\log(n))$. We use labels for keeping track of connected components.

Remark 2.1.5: Note that algorithm 2.1.4 always returns MST T. To see this, suppose, for contradiction, T is not MST. Then by Theorem 1.2.13 there exists $uv \in E \setminus E(T)$ and $e \in T_{uv}$ such that $c_{uv} < c_e$. At the point where e was added to T, the vertices u and v were in different connected components which is a contradiction since there exists a u-v path which is connected. ◁

Remark 2.1.6: It is also easy to show that Kruskal's algorithm works with linear programming. Let $G = (V, E)$ where $|V| = n$ and $|E| = m$. For all $e \in E$, define variables x_e where

$$x_e = \begin{cases} 1 \text{ if } e \text{ is in MST,} \\ 0 \text{ if } e \text{ is not in MST.} \end{cases}$$

We have

$$
\begin{aligned}
(\mathrm{P_{st}}):\ \min \quad & \sum_{e\in E} c_e x_e, \\
\text{subject to} \quad & \sum_{e\in E} x_e = n-1, \\
& \sum_{e\in F} x_e \le n-\kappa(F),\ \forall\, F\subseteq E, \\
\text{with} \quad & 0 \le x_e \le 1.
\end{aligned}
$$

Here $\kappa(F)$, kappa of F, is the number of connected components of (V, F). $\triangleleft$

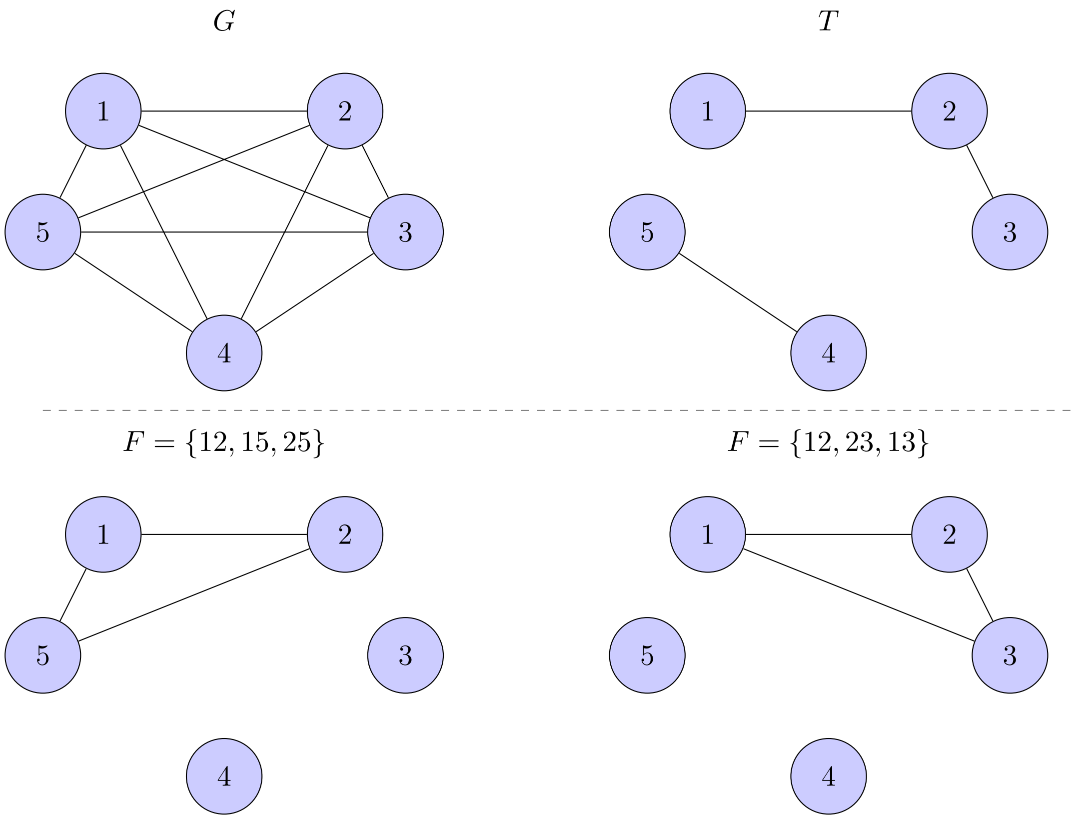
2.1.2 Validating Kruskal's Algorithm with Linear Programming

Recall MST problem we introduced in subsubsection 1.2.1.1. Given $G = (V, E)$ and $c : E \to \mathbb{R}$, we want to find a spanning tree of G of minimum cost.

Definition 2.1.7: A graph is called a ***forest*** if it is acyclic (contains no cycles). $\triangleleft$

Claim 2.1.8: If T is a forest of $G = (V, E)$ with $V(T) = V$, then for all $F \subseteq E$, T has at most $n - \kappa(F)$ edges of F.

Example 2.1.9:



(V, F)
$\kappa(F) = 3$
$n - \kappa(F) = 2$
$|E(T) \cap F| = 1$

(V, F)
$\kappa(F) = 3$
$n - \kappa(F) = 2$
$|E(T) \cap F| = 2$

Figure 2.1.2: Illustration of of Claim 2.1.8.

◁

Proof: Let $V_1, \ldots, V_{\kappa(F)}$ be the vectors of each connected component of (V, F). Let $n_i = |V_i|$ for all $i = 1, \ldots, \kappa(F)$. Consider $G_i = (V_i, E(T) \cap E(V_i) \cap F)$. If G_i is a tree, then it has $n_i - 1$ edges. If it's not a tree then it has at most $n_i - 1$ edges since it's acyclic. So, the number of edges in G is at most $n_i - 1$. We have

$$\begin{aligned} |E(T) \cap F| &= \sum_{i=1}^{\kappa(F)} |E(T) \cap E(V_i) \cap F| \\ &\leq \sum_{i=1}^{\kappa(F)} (n_i - 1) \\ &= \left(\sum_{i=1}^{\kappa(F)} n_i \right) - \kappa(F) \\ &= n - \kappa(F), \end{aligned}$$

as required. □

Remark 2.1.10: Given a spanning tree T, let $x^{\top}$ be its ***characteristic vector***. i.e.

$$x_e^{\top} = \begin{cases} 1 & \text{if } e \in E(T), \\ 0 & \text{if } e \notin E(T). \end{cases}$$

$x^{\top}$ is feasible for $(\mathrm{P_{st}})$. Thus, the optimal solution of $(\mathrm{P_{st}})$ is at most equal to the cost of MST.

Note that by convention, $\mathcal{X}$ is used to denote characteristic vector. In this course we'll use x. $\triangleleft$

Theorem 2.1.11: Let T be the tree returned by Kruskal's algorithm. Then $x^{\top}$ is optimal for $(\mathrm{P_{st}})$.

Proof: We have the linear program $(\mathrm{P_{st}})$ as follows.

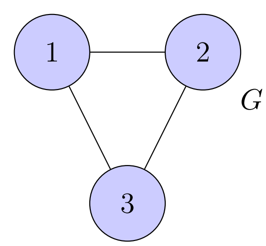
$$\begin{aligned} (\mathrm{P_{st}}):\ \min \quad & c^{\top}x, \\ \text{subject to} \quad & \sum_{e \in F} x_e \le n - \kappa(F), \quad \forall\, F \subsetneq E, \\ & \sum_{e \in F} x_e = n - 1, \\ \text{with} \quad & x_e \ge 0. \end{aligned}$$

We have the dual of $(\mathrm{P_{st}})$ as $(\mathrm{D_{st}})$ where

$$\begin{aligned} (\mathrm{D_{st}}):\ \max \quad & \sum_{\forall\, F \subseteq E} (n - \kappa(F)) y_F, \\ \text{subject to} \quad & \sum_{F : e \in F} y_F \le c_e, \quad \forall\, e \in E, \\ \text{with} \quad & y_F \le 0, \quad F \subsetneq E, \\ & y_E \text{ free.} \end{aligned}$$

We illustrate what we mean by above in the following example.

Example: Let $G = (V, E)$ where $V = \{1, 2, 3\}$ and $E = \{12, 13, 23\}$ as shown.



So we have $(\mathrm{P_{st}})$ as

$$\begin{array}{llllll} (\mathrm{P_{st}}):\min & c_{12}x_{12} + c_{13}x_{13} + c_{23}x_{23}, & & & & F \\ \text{subject to} & x_{12} & & & \le 3 - 2, & \{12\} \\ & & x_{13} & & \le 3 - 2, & \{13\} \\ & & & x_{23} & \le 3 - 2, & \{23\} \\ & x_{12} & + x_{13} & & \le 3 - 1, & \{12, 13\} \\ & x_{12} & + & x_{23} & \le 3 - 1, & \{12, 23\} \\ & & x_{13} & + x_{23} & \le 3 - 1, & \{13, 23\} \\ & x_{12} & + x_{13} & + x_{23} & \le 3 - 1, & \{12, 13, 23\} \\ \text{with} & x_{12} & + x_{13} & + x_{23} & = 2. & \end{array}$$

We have ($\mathrm{D_{st}}$) as

$$
\begin{array}{llllllll}
(\mathrm{D_{st}}) : \max & y_{12} + y_{13} + y_{23} + 2y_{\{12,13\}} & + 2y_{\{12,23\}} & + 2y_{\{13,23\}} & + 2y_{\{12,13,23\}} & \\
\text{subject to} & y_{12} + \quad\quad y_{\{12,13\}} & + y_{\{12,23\}} & + & y_{\{12,13,23\}} & \le c_{12} \\
 & \quad y_{13} + \quad y_{\{12,13\}} & + & y_{\{13,23\}} & + 2y_{\{12,13,23\}} & \le c_{13} \\
 & \quad\quad y_{23} + & y_{\{12,23\}} & + y_{\{13,23\}} & + 2y_{\{12,13,23\}} & \le c_{23},
\end{array}
$$

with all y's ≤ 0 except for $y_{12,13,23}$. ◁

We now let

$$
\begin{aligned}
E &= \{e_1, \ldots, e_m\} \text{ with } c_{e_1} \le \cdots \le c_{e_m}, \\
E_i &:= \{e_1, \ldots, e_i\}, \\
\overline{y}_{E_i} &= c_{e_i} - c_{e_{i+1}} \le 0, \quad \forall\, i = 1, \ldots, m-1, \\
\overline{y}_E &= c_{e_m}, \\
\overline{y}_F &= 0, \quad \text{for all other } F \subseteq E.
\end{aligned}
$$

Claim 2.1.12*:* $\overline{y}$ is feasible for ($\mathrm{D_{st}}$).

Proof: We immediately see that the sign restrictions are satisfied. Consider edge e_k. We have

$$\sum_{F : e_k \in F} \overline{y}_F = \sum_{i=k}^{m} \overline{y}_{E_i} = \left(\sum_{i=k}^{m} c_{e_i} - c_{e_{i+1}} \right) + c_{e_m} = c_{e_k}.$$

We see that the dual constraints are tight. ■

Complementary-Slackness conditions state the following.

1. If primal variable is non-zero, then corresponding dual constraint is tight.
2. If dual variable is non-zero, then corresponding primal constraint is tight.

Clearly 1 holds since by the proof of above claim, every dual constraint is tight. To show 2 is true, we make the following claims.

Claim: For all $F \subseteq E$, if T is a maximal forest of (V, F) (that is, if any more edges are added to T it's no longer a forest), then $|E(T) \cap F| = n - \kappa(F)$.

Proof: Exercise. ■

Claim: At every step of Kruskal's algorithm we have a maximal forest of $E_i = \{e_1, \ldots, e_i\}$.

Proof: Suppose T is a forest constructed after edges in E_i and suppose, for contradiction, T is not a maximal forest of (V, E_i). Then, there exists $e_k \in E_i \setminus E(T)$ such that $T + e_k$ is a forest. Then, when Kruskal's algorithm is at step $k \le i$, we had constructed $(V, E(T) \cap E_k)$ and e_k was not added. But that means adding e_k would have created a cycle and this a contradiction since T is a tree and $T + e_k$ is acyclic. ■

Hence, by above claims we have

$$\sum_{e \in F} x_e^\top n - \kappa(E_i), \quad \forall\, i = 1, \ldots, m.$$

Hence, $x^\top$ and $\overline{y}$ satisfy the Complementary-Slackness (C-S) conditions. □

Remark 2.1.13: The tight inequality we found in the proof of Claim 2.1.12 provides a certificate that verifies the MST obtained from Kruskal's algorithm is correct. ◁

2.2 Greedy Algorithms

Definition 2.2.1: In every step, Kruskal's algorithm picks the locally best option since it takes the cheapest edge that keeps the solution feasible. The algorithms that prioritize locally best options are called ***greedy algorithms***.

This greedy approach doesn't work on some problems. ◁

Definition 2.2.2: A cycle that goes through every edge exactly once is called a ***Hamiltonian cycle***. ◁

Example 2.2.3: Let $G = (V, E)$ as below. Then 123451 is a Hamiltonian cycle.

G

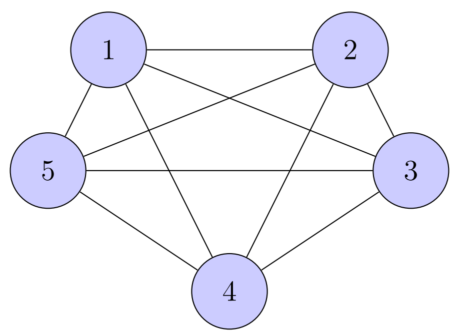


Figure 2.2.1: $G = (V, E)$ with Hamiltonian cycle 123451.

◁

Example 2.2.4: Consider the traveling salesman problem. The goal is to find a minimum cost Hamiltonian cycle in a given graph $G = (V, E)$ with edge costs $c : E \to \mathbb{R}$. A greedy algorithm for this problem can be of the following form.

Algorithm 2.2.5: Greedy algorithm for TSP.

Pick $v \in V$.
Let $v_1 = v$.
for $i = 1, \ldots, n-1$ **do**
  $v_{i+1} \leftarrow w$ where w is the vertex with minimum cost $c_{v_i w}$ and $w \notin \{v_1, \ldots, v_i\}$.
return $v_1, \ldots, v_n, v_1$.

Let $G = (V, E)$ as below and suppose $v_1 = 1$.

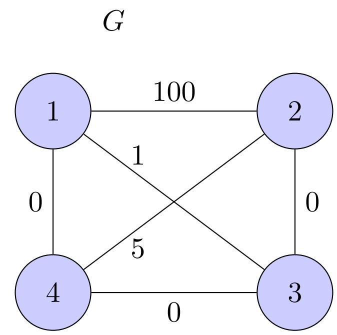


Figure 2.2.2: $G = (V, E)$ with Hamiltonian cycle 123451.

◁

If we use the greedy algorithm described in algorithm 2.2.5 start at $v_1 = 1$, then the greedy algorithm gives us the Hamiltonian cycle 14321 which has cost 100 but 13241 is also a Hamiltonian cycle but it has cost 2. Hence, greedy algorithm doesn't always work efficiently.

2.2.1 Maximum Cost Forest Problem

Given $G = (V, E)$, $c : E \to \mathbb{R}$ where G is connected and $F \subseteq E$ such that (V, F) is a forest, maximum cost forest problem tries to maximize $\sum_{e \in F} c_e$. Consider the following algorithm.

Algorithm 2.2.6: Pseudocode for max. cost forest problem

Define $E^- = \{e \in E \mid c_e \leq 0\}$.
Let $c'_e = -c_e$ for all $e \notin E^-$ and $c'_e = 0$ for all $e \in E^-$.
Run any algorithm that gives MST on $G = (V, E)$ with costs c'_e.
Let T be MST that is returned my the MST algorithm.
Delete all edges in T that belong to E^-.
return T.

Exercise 2.2.7: Show algorithm 2.2.6 works as required. ◁

2.2.1.1 Using Kruskal's Algorithm for Max. Cost Forest

We can use Kruskal's algorithm as follows for MCFP.

Algorithm 2.2.8: Kruskal's algorithm for MCFP.

Init : $H = (V, \varnothing)$, $\overline{E} \leftarrow E$

while *H is not a spanning tree and* $\overline{E} \neq \varnothing$ **do**
  Let $e \in \overline{E}$ be one with the largest cost c_e with endpoints in different connected components of H
  if $c_e > 0$ **then**
    Add e to H.
  $\overline{E} \leftarrow \overline{E} \setminus \{e\}$
return H.

The rough idea behind this algorithm is as follows.

while *There exists an edge e such that $c_e > 0$ with endpoints of e in different connected components,* **do**
  Choose c_e that is largest.
  Add e to H.
return H.

Exercise 2.2.9: Prove algorithm 2.2.8 works. ◁

2.2.1.2 Properties of Forests

We will refer forests by their edge sets. Forests have the following properties.

1. The empty set is a forest.
2. If F is a forest and if $F' \subseteq F$, then F' is a forest.
3. If $A \subseteq E$, then every inclusion-wise maximal forest $F \subseteq A$, has the same cardinality.

Note that these properties coincide with the definition of *matroids* which will be explained later.

Example 2.2.10: Let $G = (V, E)$ where $V = \{1, 2, 3, 4, 5\}$ and let $A \subseteq G$ be shown in blue zigzag below. An illustration of property 3 is as follows.

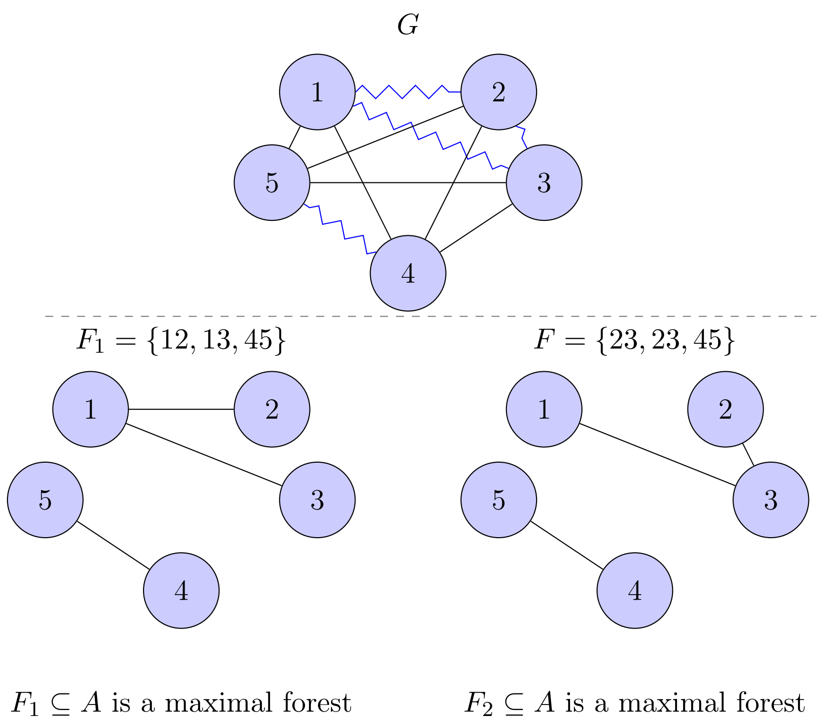


Figure 2.2.3: $G = (V, E)$ with $F_1, F_2 \subseteq A$ where ∿∿:A.

◁

Note that $F_1, F_2 \subseteq A$ are maximal forests since if any edge where added to F_1 or F_2, they no longer are subset forests of A.

Remark 2.2.11: We proved property ③ in MST problem with $|F| = n - \kappa(A)$. ◁

2.3 Matroids

We introduce the abstract notion of matroids. We will focus on how greedy algorithms work on matroids.

2.3.1 Independence Systems and Independent Sets

Definition 2.3.1: Let S be a finite set and let $\mathcal{I} \subseteq \mathcal{P}(S) = 2^S$. Here $\mathcal{P}(S)$ is the *power set* of S, which is the collection of all subsets of S. So, $\mathcal{I}$ is a collection of subsets of S. If $\mathcal{I}$ satisfies

M1 $\mathcal{I} \ni \varnothing$, and

M2 if $I_1 \in \mathcal{I}$ and $I_2 \subseteq I_1$, then $I_2 \in \mathcal{I}$,

then the pair $(S, \mathcal{I})$ is called an ***independence system*** and the elements $I \in \mathcal{I}$ are called ***independent sets***. This property is known as the *hereditary property* and it is equivalent to saying every subset of an independent set is independent. If $(S, \mathcal{I})$ is an independence system and if it also satisfies

M3 for all $A \subseteq S$, every inclusion-wise maximal element of $\mathcal{I}$ contained in A has same cardinality,

then the pair $\mathcal{M} = (S, \mathcal{I})$ is called a ***matroid***, where S is a finite set (which is called the ***ground set***) and $\mathcal{I}$ is a collection of subsets of the ground set. This property is known as the *augmentation property* or *(independent set) exchange property*. ◁

Example 2.3.2: Let $G = (V, E)$ with $V = \{1, 2, 3, 4, 5\}$ and let $E = \{12, 13, 14, 15, 23, \dots\}$ where E is the ground set.

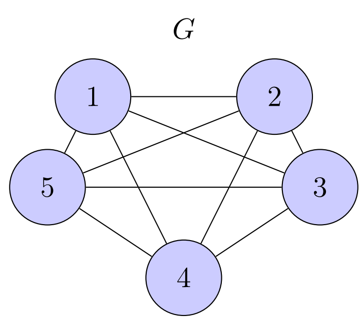


Figure 2.3.1: $G = (V, E)$ where $\mathcal{I}$, the set of all forests of G, is an independence system.

◁

$\mathcal{I}$ = the set of all forests of G is an independence system, $I_1 = \{12, 14, 34\}$ is an independent set but $A = \{12, 13, 23\}$ is not an independent set since it's not a forest. i.e. $A \notin \mathcal{I}$.

Example 2.3.3: Let $S = \{1, \ldots, m\}$ and $k \in \mathbb{Z}^+$. Let $\mathcal{I} = \{U \subseteq S \mid |U| \leq k\}$. Since $|\varnothing| = 0$, then $\varnothing \in \mathcal{I}$ and it is clear that for all $I_1 \in \mathcal{I}$, if $I_2 \subseteq I_1$, then $I_2 \in \mathcal{I}$. So, $\mathcal{I}$ is an independence system. $\mathcal{I}$ also satisfies property **M3**. To see this, let $A \subseteq S$ with $|A| \leq k$. Then, in this case the only maximal element of I in A is A. If $|A| > k$ and $I \in \mathcal{I}$ with $I \subseteq A$ and $|I| < k$, then there exists $e \in S$ such that $I \cup \{e\} \in \mathcal{I}$. i.e. I is not maximal. Hence, every maximal element of I has cardinality of 5. Hence, the pair $(S, \mathcal{I})$ is a matroid. ◁

Example 2.3.4: Let $S = \{1, 2, 3, 4\}$ and $\mathcal{I} = \{\varnothing, \{1\}, \{2\}, \{3\}, \{4\}, \{1, 2\}\}$. It is easy to see that **M1** and **M2** hold. Let $A = \{1, 2, 3\} \subseteq S$. We have

$$\begin{aligned} \{1,2\} &\subseteq A, &\text{and}\quad \{1,2\} &\subseteq \mathcal{I}, \\ \{3\} &\subseteq A, &\text{and}\quad \{3\} &\subseteq \mathcal{I}. \end{aligned}$$

Clearly $\{1, 2\}$ and $\{3\}$ are maximal but $|\{1, 2\}| \neq |\{3\}|$. So, **M3** doesn't hold. Hence, the pair $(S, \mathcal{I})$ is not a matroid but $\mathcal{I}$ is an independence system. ◁

Remark 2.3.5: We will show that greedy algorithms for independence systems $\mathcal{I} \subseteq 2^S$ give optimal solution if and only if the pair $(S, \mathcal{I}) = \mathcal{M}$ is a matroid. ◁

Definition 2.3.6: Let $(S, \mathcal{I})$ be an independence system. Given $A \subseteq S$, a ***basis of*** A is a maximal independent set contained in A. If $A = S$ where $\mathcal{M} = (S, \mathcal{I})$, then a basis of A is a basis of $\mathcal{M}$. ◁

Example 2.3.7: Consider the matrix B below.

$$B = \begin{array}{c} \begin{array}{ccccc} \textcircled{1} & \textcircled{2} & \textcircled{3} & \textcircled{4} & \textcircled{5} \end{array} \\ \begin{bmatrix} 1 & -1 & 0 & 0 & 1 \\ 2 & -2 & 0 & 1 & 4 \\ 4 & 1 & 1 & 0 & 2 \end{bmatrix} \end{array}.$$

Let $S = \{1, 2, 3, 4, 5\}$ (column indices) and let $\mathcal{I}$ be defined as follows.

$$\mathcal{I} = \{A \subseteq S \mid \text{corresponding columns are linearly independent}\}.$$

Since **M1**, **M2** and **M3** are satisfied, $(S, \mathcal{I}) = \mathcal{M}$ is a matroid. Any matroid of the form $\mathcal{M} = (S, \mathcal{I})$ where the ground set S is column (row) indices and $\mathcal{I}$ is the set of linearly independent columns (rows) is called a ***linear matroid***. Basis of $\mathcal{M}$ are the bases of the vector space generated by the corresponding columns. ◁

Remark 2.3.8: We can characterize the necessary matroid condition as follows.

M3 : $\forall\ A \subseteq S$, every inclusion-wise maximal element of $\mathcal{I}$ contained in A has cardinality $\iff$ $\forall\ A \subseteq S$, all bases of A have the same cardinality.

◁

Definition 2.3.9: Let $(S, \mathcal{I}) = \mathcal{M}$ be an independence system and let $A \subseteq S$. The ***rank of*** A, $\mathrm{r}(A)$, is the largest basis of A. That is,

$$\mathrm{r}(A) \underset{\text{def}}{=} \max\{|J| \mid J \subseteq A \text{ and } J \in \mathcal{I}\}.$$ ◁

If $A = S$, then $\mathrm{r}(A) = \mathrm{r}(S) = \mathrm{r}(\mathcal{M})$.

Remark 2.3.10: It is easy to see that $\mathrm{r}(A) = |A|$ if and only if $A \in \mathcal{I}$. Consider the graph $G = (V, E)$. Let $S = E$ and $\mathcal{I} = \{A \subseteq S \mid (V, A) \text{ is a forest}\}$. The pair $(S, \mathcal{I})$ of this form is called a ***graphic (forest) matroid***. We have

$$\mathrm{r}(A) = n - \kappa(A),$$

where $\kappa(A)$ is the number of connected connected components of A. ◁

2.3.2 Solving Maximum Weighted Independent Set Problem with Greedy Algorithm

Given $\mathcal{M} = (S, \mathcal{I})$ independence system and costs c_e for all $e \in S$, consider the problem of finding $A \in \mathcal{I}$ maximizing $\sum_{e \in A} c_e$. This problem is called *maximum weighted independent set problem.* Consider the greedy algorithm below.

Algorithm 2.3.11: Greedy Algorithm for Max Weighted Independent Set Problem

$J \leftarrow \varnothing$
while $\exists\, e \in S \setminus J$ *such that* $c_e > 0$ *and* $J \cup \{e\} \in \mathcal{I}$ **do**
  Let e be such element of largest c_e,
  $J \leftarrow J \cup \{e\}$
return J

Let $S' = \{e \in S \mid c_e > 0\}$. Define $\mathcal{I}' = \{A \subseteq S' \mid A \in \mathcal{I}\}$. So, $\mathcal{M}' = (S', \mathcal{I}')$ is an independence system. In fact, if $\mathcal{M}$ is a matroid (that is, if $\mathcal{M}$ satisfies **M3**) then so is $\mathcal{M}'$. Hence, solving maximum weighted independent set over $\mathcal{M}'$ solves the problem over $\mathcal{M}$. Note that since our goal is to maximize the sum of costs, we may assume that $c_{e'} > 0$ for all $e' \in S'$.

Theorem 2.3.12 (Rado '57, Edmonds '71)**:** Let $\mathcal{M} = (S, \mathcal{I})$ be a matroid and let $c : S \to \mathbb{R}^+$. Then, greedy algorithm in algorithm 2.3.11 finds maximum weighted independent set.

Proof: Exercise. ◁

Theorem 2.3.13: Let $\mathcal{M} = (S, \mathcal{I})$ be an independence system. Greedy algorithm finds a maximum weighted independent set for all $c \in \mathbb{R}^S$ if and only if $\mathcal{M}$ is a matroid.

Proof: For forward direction we will use contrapositive. Suppose $\mathcal{M}$ is not a matroid. Then, $\mathcal{M}$ does not satisfy **3**. Let $A \subseteq S$ such that A_1, A_2 are two bases of A with $|A_1| < |A_2|$. Note that such two bases exists by Remark 2.3.8. Let

$$c_e = \begin{cases} 1 + \epsilon & \text{if } e \in A_1, \\ 1 & \text{if } e \in A_2, \\ 0 & \text{otherwise.} \end{cases}$$

Hence, in this case greedy algorithm in algorithm 2.3.11 outputs A where

$$\sum_{e \in A_1} c_e = (1 + \epsilon)|A_1|.$$

But we also have $\sum_{e \in A_2} c_e = |A_2|$. So if we choose ϵ small enough where

$$\epsilon < \frac{|A_2|}{|A_1|} - 1,$$

then A_1 is not a maximum weighted independent set which proves the contrapositive. The converse immediately follows from Theorem 2.3.12. □

Remark 2.3.14 (Runtime of Greedy Algorithm)**:** Consider the greedy algorithm in algorithm 2.3.11. We see that the main loop is executed $O(|S|)$ times. Hence if the process of checking $J \in \mathcal{I}$ can be done in poly($|S|$), then greedy algorithm can run in polytime in $|S|$. ◁

Definition 2.3.15: Let $\mathcal{M} = (S, \mathcal{I})$ be an independence system and let $A \subseteq S$.

$$\rho(A) \underset{\text{def}}{=} \min\{|B| \mid B \text{ is a basis of } A.\}$$

Note that $\mathcal{M} = (S, \mathcal{I})$ is a matroid if and only if $\rho(A) = \operatorname{rank}(A)$ for all $A \subseteq S$. ◁

Definition 2.3.16: Let $\mathcal{M} = (S, \mathcal{I})$ be an independence system. The ***rank quotient of*** $\mathcal{M}$, $q(S, \mathcal{I})$, is defined as

$$q(S, \mathcal{I}) \underset{\text{def}}{=} \min_{A \subseteq S} \frac{\rho(A)}{\operatorname{Rank} A}.$$

Note that we always have $q(S, \mathcal{I}) \leq 1$ and it follows that $\mathcal{M}$ is a matroid if and only if $q(S, \mathcal{I}) = 1$. ◁

Theorem 2.3.17 (Jenkyns '76)**:** Let $M = (S, \mathcal{I})$ be an independence system. Let $\mathrm{GR}_{S,\mathcal{I}}$ be the total weight of solution found by the greedy algorithm in algorithm 2.3.11. Let $\mathrm{OPT}_{S,\mathcal{I}}$ be weight of optimal solution. Then

$$\mathrm{GR}_{S,\mathcal{I}} \geq q(S, \mathcal{I}) \mathrm{OPT}_{S,\mathcal{I}}.$$

Note that this implies if $\mathcal{M}$ is a matroid then greedy algorithm in algorithm 2.3.11 finds an optimal solution.

Proof: We prove Theorem 2.3.17 as follows. Let $S = \{e_1, \ldots, e_m\}$ with $c_{e_1} \geq \cdots \geq c_{e_m}$ and let $S_j = \{e_1, \ldots, e_j\}$ for all $j = 1, \ldots, m$. Let G be the solution obtained by the greedy algorithm and let σ be the optimal solution. So, $G, \sigma \subseteq S$. Let $G_j = G \cap S_j$ and $\sigma_j = \sigma \cap S_j$. Let $G_0 = \varnothing = \sigma_0$. We have

$$c(G) = \sum_{e_j \in G} c_{e_j} = \sum_{j=1}^{m} \big(\underbrace{|G_j| - |G_{j-1}|}_{(\star)}\big) c_{e_j} = \sum_{j=1}^{m-1} |G_j| \big(c_{e_j} - c_{e_{j+1}}\big) + c_{e_m} |G_m| = \sum_{j=1}^{m} |G_j| \big(\underbrace{c_{e_j} - c_{e_{j+1}}}_{\Delta_j \geq 0}\big),$$

where $c_{e_{m+1}} = 0$. Note that

$$(\star) = \begin{cases} = 1 \text{ if } e_i \in G, \\ = 0 \text{ otherwise.} \end{cases}$$

At step j, G_j is a basis of S_j. Hence, $|G_j| \geq \rho(S_j)$. Thus,

$$c(G) \geq \sum_{j=1}^{m} \rho(S_j) \Delta_j \geq \sum_{j=1}^{m} q(S, \mathcal{I}) \mathrm{r}(S_j) \Delta_j \geq q(S, \mathcal{I}) \underbrace{\sum_{j=1}^{m} |\sigma_j| \Delta_j}_{c(\sigma)}.$$ □

Corollary 2.3.18: If $\mathcal{M}$ is a matroid, then the greedy algorithm in algorithm 2.3.11 computes the optimal solution.

Theorem 2.3.19: If $\mathcal{M} = (S, \mathcal{I})$ is an independent system then

M3 : $\forall\, A \subseteq S$, every inclusion-wise maximal element of $\mathcal{I}$ contained in A has cardinality	$\iff$	**M3'** : $\forall\, X, Y \in \mathcal{I}$ such that $\lvert X\rvert < \lvert Y\rvert$, $\exists\, e \in Y \setminus X$ such that $X \cup \{e\} \in \mathcal{I}$.

Proof: Skipped, exercise. $\triangleleft$

Example 2.3.20: Let $S = \{1, 2, 3, 4\}$ and $\mathcal{I} = \{\varnothing, \{1\}, \{2\}, \{3\}, \{4\}, \{1, 2\}, \{3, 4\}\}$. Clearly **M1** and **M2** are satisfied but since $\{1, 2\}, \{3\} \in \mathcal{I}$ and $\{3\} \cup \{e\} \notin \mathcal{I}$ for any $\{e\} \in \{1, 2\} \setminus \{3\}$, then **M3'** is not satisfied. Hence, $(S, \mathcal{I})$ is an independence system but not a matroid. $\triangleleft$

Remark 2.3.21: We can fully specify an independence system or a matroid by listing its bases. In the above example, the set of bases of $(S, \mathcal{I})$ is $\mathcal{B} = \{\{1, 2\}, \{3, 4\}\}$. $\triangleleft$

Theorem 2.3.22: Let S be a finite set and let $\mathcal{B} \subseteq \mathcal{P}(S) = 2^S$. Then $\mathcal{B}$ is the set of bases of a matroid if and only if

1. $\mathcal{B} \neq \varnothing$,
2. For all $B_1, B_2 \in \mathcal{B}$ and $x \in B_1 \setminus B_2$, there exists $y \in B_2 \setminus B_1$ such that $(B_1 \setminus \{x\}) \cup \{y\} \in \mathcal{B}$.

Proof: Exercise. $\triangleleft$

Remark 2.3.23: Note that by this theorem, the set $\mathcal{B} = \{\{1, 2\}, \{3, 4\}\}$ in above example cannot be a set of basis of a matroid.

$M = (S, \mathcal{I}) = \{\varnothing\}$ is a matroid and set of bases $\mathcal{B} = \{\varnothing\} = \mathcal{I} \neq \varnothing$. $\triangleleft$

2.3.3 Matroid Constructions

Given a matroid $\mathcal{M}$, we can construct other matroids by using some operations.

Remark 2.3.24: Let $\mathcal{M} = (S, \mathcal{I})$ be a matroid. By using the following we can construct other matroids.

1. <u>Deletion:</u> If $J \subseteq S$ then $\mathcal{M} \backslash J = \{S', \mathcal{I}'\}$ is a matroid where $S' = S \backslash J$ and $\mathcal{I}' = \{A \subseteq S' \mid A \in \mathcal{I}\}$.

   We use **backslash, ($\backslash$)**, to denote deletion of J from matroid $\mathcal{M}$. Some sources use $M - J$ notation for deletion which is objectively better.

2. <u>Truncation:</u> Given $k \in \mathbb{Z}^+$ define $\mathcal{I}' = \{A \in \mathcal{I} \mid |A| \leq k\}$. Then, $\mathcal{M}' = (S, \mathcal{I}')$ is a matroid.

3. <u>Dual:</u> Let $\mathcal{I}^* = \{A \subseteq S \mid S \setminus A \text{ has a basis of } \mathcal{M}\}$. Equivalently, $\mathrm{r}(S \setminus A) = \mathrm{r}(S)$. We call $\mathcal{M}^* = (S, \mathcal{I}^*)$ the ***dual matroid of*** $\mathcal{M}$. Note that $(\mathcal{M}^*)^* = \mathcal{M}$

   We will prove that $\mathcal{M}^*$ is a matroid and $\mathrm{r}_{M^*}(A) = |A| + \mathrm{r}_M(S \setminus A) - \mathrm{r}_M(S)$.

4. <u>Contraction:</u> If $J \subseteq S$ and if $\mathcal{B}$ is a basis of J, then $\mathcal{M}/J = (S', \mathcal{I}')$ is a matroid where $S' = S \setminus J$ and $\mathcal{I}' = \{A \subseteq S' \mid A \cup \mathcal{B} \in \mathcal{I}\}$.

   We use **forward slash, ($/$)**, to denote deletion of J from matroid $\mathcal{M}$.

5. <u>Disjoint Union:</u>[1]Let $\mathcal{M}_i = (S_i, \mathcal{I}_i)$ be matroids. If S_i are distinct for all $i = 1, \ldots, k$ then the union of these matroids is a direct sum and $\bigoplus_{i=1}^k \mathcal{M}_i = \mathcal{M}_1 \oplus \cdots \oplus \mathcal{M}_k = \mathcal{M} = (S', \mathcal{I}')$ is a

matroid where

$$S' = \bigcup_{i=1}^{k} S_i, \quad \mathcal{I}' = \bigcup_{i=1}^{k} \mathcal{I}_i \quad \text{and} \quad A \in \mathcal{I}' \iff A = \bigcup_{i=1}^{k} A_i \text{ where } A_i \in \mathcal{I}_i \text{ for } i = 1, \ldots, k. \quad \triangleleft$$

Exercise 2.3.25: Show that duality operation on matroids is an involution. i.e. $M = (M^*)^*$. $\triangleleft$

Example 2.3.26: Let $G = K_4$ (complete graph with 4 vertices). We have G and G' as below where G' is obtained by contracting edge $e = 23$.

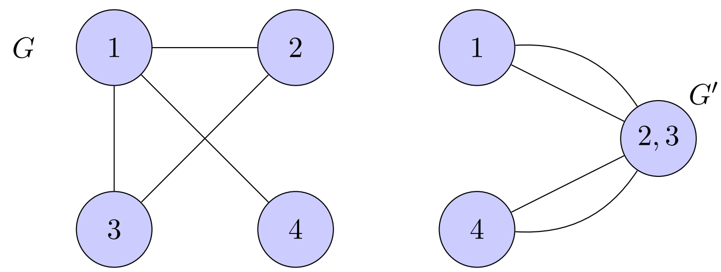


Figure 2.3.2: $G = K_2$ and G'.

$\triangleleft$

Aside: This is a digression and the material here is beyond the scope of this course. We make the following remarks about union and disjoint union of matroids:

- Let $\mathcal{M}$ and $\mathcal{N}$ be two matroids with ground sets E and F respectively. The *direct sum* of matroids M and N is the matroid whose ground set is the disjoint union of E and F, and whose independent sets are the disjoint unions of an independent set of $\mathcal{M}$ with an independent set of $\mathcal{N}$.
  The *union* of $\mathcal{M}$ and $\mathcal{N}$ is the matroid whose ground set is the union (not the disjoint union) of E and F, and whose independent sets are those subsets that are the union of an independent set in $\mathcal{M}$ and one in $\mathcal{N}$. Usually the term "union" is applied when $E = F$, but that assumption is not essential. If E and F are disjoint, the union is the direct sum.

- The ***disjoint union*** of two sets A and B is a binary operator that combines all distinct elements of a pair of given sets, while retaining the original set membership as a distinguishing characteristic of the union set. The disjoint union is denoted
  $$A \bigsqcup B = (A \times \{0\}) \bigcup (B \times \{1\}) = A^* \bigcup B^*$$
  where $A \times S$ is a Cartesian product. For example, the disjoint union of sets $A = \{1, 2, 3, 4, 5\}$ and $B = \{1, 2, 3, 4\}$ can be computed by finding
  $$A^* = \{(1,0), (2,0), (3,0), (4,0), (5,0)\},$$
  $$B^* = \{(1,1), (2,1), (3,1), (4,1)\}.$$
  So, $A \sqcup B = A^* \cup B^* = \{(1,0), (2,0), (3,0), (4,0), (5,0), (1,1), (2,1), (3,1), (4,1)\}$. In this case A_i^* is referred to as a copy of A_i. Disjoint unions are also sometimes written as $\biguplus_{i \in I} A_i$, or $\dot{\bigcup}_{i \in I} A_i$ or $\bigcup^*_{A \in C} A$. In category theory the disjoint union is defined as a coproduct and $\coprod$ is used.

[1]Disjoint union was covered on another lecture (L10, on Feb. 2020) but it was included in this list for the sake of completeness.

- Some authors use $\vee$ to denote matroid union. $\triangleleft$

Remark 2.3.27: We verify that using operations of deletion, truncation, taking the dual and contraction on a matroid gives a matroid. Let $(S,\mathcal{I}) = \mathcal{M}$ be a matroid.

① Deletion: Recall that we have if $J \subseteq S$ then $\mathcal{M} \setminus J = \{S',\mathcal{I}'\}$. We want to show $(S',\mathcal{I}')$ is a matroid where $S' = S \setminus J$ and $\mathcal{I}' = \{A \subseteq S' \mid A \in \mathcal{I}\}$.

For any $J \subseteq S$, we have $\varnothing \subseteq S \setminus J$. So, $\varnothing \in \mathcal{I}'$. Let $I \in \mathcal{I}'$ and $K \subseteq I$. Since $I \in \mathcal{I}$, then $K \in \mathcal{I}$ and since $I \subseteq S'$ then so is K. Hence, hereditary property holds. Let $X,Y \in \mathcal{I}'$ with $|X| < |Y|$. Then, $X,Y \in \mathcal{I}$ since $\mathcal{M}$ is a matroid. Then, there exists $x \in Y \setminus X$ such that $X \cup \{x\} \in \mathcal{I}$ but $X \cup \{x\} \subseteq S'$. Hence, $X \cup \{x\} \in \mathcal{I}'$. So, $(S,\mathcal{I}')$ is a matroid.

② Truncation: Recall that given $k \in \mathbb{Z}^+$ we define $\mathcal{I}' = \{A \in \mathcal{I} \mid |A| \leq k\}$. We will show, $\mathcal{M}' = (S,\mathcal{I}')$ is a matroid.

Since $\varnothing \in \mathcal{I}$ and since $|\varnothing| = 0 \leq k$ then $\varnothing \in \mathcal{I}'$. Let $A \subseteq \mathcal{I}'$ and $B \subseteq A$. Since $B \in \mathcal{I}$ and since $|B| \leq |A| \leq k$, then $B \in \mathcal{I}'$. So, hereditary property holds. Let $X,Y \in \mathcal{I}'$ with $|X| < |Y|$. Then, $X,Y \in \mathcal{I}$ and $X \cup \{x\} \in \mathcal{I}$ where $\{x\} \in Y \setminus X$. Since $|X \cup \{x\}| = |X| + 1 \leq |Y| \leq k$, then $X \cup \{x\} \in \mathcal{I}'$. Hence, $(S,\mathcal{I}')$ is a matroid.

③ Dual: Recall that we let $\mathcal{I}^* = \{A \subseteq S \mid S \setminus A \text{ has a basis of } \mathcal{M}\}$. Equivalently, $\mathrm{r}(S \setminus A) = \mathrm{r}(S)$. We will show $\mathcal{M}^* = (S,\mathcal{I}^*)$ is a matroid.

Since $\mathrm{r}(S \setminus \varnothing) = \mathrm{r}(S)$, then $\varnothing \in \mathcal{I}^*$. Let $A \in \mathcal{I}^*$ and $B \subseteq A$. Note that we have $\mathrm{r}(S \setminus A) = \mathrm{r}(S)$ if and only if deleting A from S still leaves us with an $\mathcal{M}$-basis of S. Hence, $S \setminus B$ still has an $\mathcal{M}$-basis of A. Hence, $B \in \mathcal{I}^*$, which means hereditary property holds. So $(S,\mathcal{I}^*)$ is an independence system. Now, consider any subset $A \subseteq S$. All $\mathcal{M}^*$ bases of A have the same cardinality. Let $J \subseteq A$ be an $\mathcal{M}^*$-basis of A. Let B be an $\mathcal{M}$-basis of $S \setminus A$. Extend it to B', an $\mathcal{M}$-basis of $S \setminus J$. So, $|B'| = \mathrm{r}(S \setminus J) = \mathrm{r}(S)$.

Claim 2.3.28*:* $A \setminus J \subseteq B'$.

Proof: Suppose, for contradiction, there exists $e \in A \setminus J$ such that $e \notin B'$. Since we have $B' \subseteq S \setminus (J \cup \{e\})$, then $J \cup \{e\} \in \mathcal{I}^*$ which is a contradiction since J is an $\mathcal{M}^*$-basis of A. ■

We know $|J| = |A| - |A \setminus J|$ and $B' = (A \setminus J) \cup B$ and that $|B'| = |A \setminus J| + |B|$. Hence,

$$\left|B'\right| = \mathrm{r}_M(S) = |A \setminus J| = \mathrm{r}_M(S \setminus A).$$

Then, $|J| = |A| - \mathrm{r}_M(S) + \mathrm{r}_M(S \setminus A)$. So sizes of all $\mathcal{M}^*$-bases of A are the same.

Remark 2.3.29: The dual matrix $(S,\mathcal{I}^*) = \mathcal{M}^*$ has the rank function

$$\mathrm{r}_{M^*}(A) = |A| - \mathrm{r}_M(S) + \mathrm{r}_M(S \setminus A). \qquad \triangleleft$$

Example 2.3.30: Consider the graphical matroid $\mathcal{M} = (E,\mathcal{I})$ presented by G below.

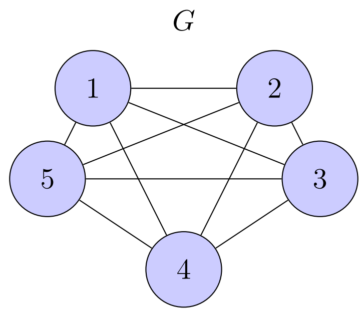


Figure 2.3.3: $\mathcal{M} = (E, \mathcal{I})$.

Here we have the following.

- $\mathcal{M}^* = (E, \mathcal{I}^*) = \{$set of edges which we can remove from $\mathcal{M}$ without making it disconnected$\}$.
- $A = \delta(2) = \{e \in E \mid e \text{ is incident to } 2\}$.
- $J = \{12, 25, 24\} \in \mathcal{I}^*$ is an $\mathcal{M}^*$-basis of A.
- $B = \{13, 14, 15\}$.
- $B' = \{13, 14, 15, 23\}$.
- $|J| = 3 = |A| - \mathrm{r}_M(E) + \mathrm{r}_M(E \setminus A) = 4 - 4 + 3 = 3$. $\triangleleft$

Remark 2.3.31: Suppose we can explore edges of a graph but to collect value (for example cost of an edge), we must destroy the edge. We want to proceed our exploration in a way that doesn't leave the graph disconnected. We see that greedy algorithm is applicable for such an exploration. $\triangleleft$

4 Contraction: Recall that if $J \subseteq S$ and if $\mathcal{B}$ is a basis of J, then we defined $S' = S \setminus J$ and $\mathcal{I}' = \{A \subseteq S' \mid A \cup \mathcal{B} \in \mathcal{I}\}$. We will show $\mathcal{M}/J = (S', \mathcal{I}')$ is a matroid

For any $J \subseteq S$, we have $\varnothing \subseteq S \setminus J$. Since for any base $\mathcal{B}$ of J we have $\varnothing \cup \mathcal{B} = \mathcal{B} \in \mathcal{I}$ then, $\varnothing \in \mathcal{I}'$. Let $K \in \mathcal{I}'$ and $L \subseteq K$. Then, $K \subseteq S'$ and $K \cup \mathcal{B} \in \mathcal{I}$. Then, $L \cup \mathcal{B} \subseteq K \cup \mathcal{B}$. Since $K \cup \mathcal{B} \in \mathcal{I}$, then any subset of it is also independent since $(S, \mathcal{I})$ is a matroid. Then, $L \cup \mathcal{B} \in \mathcal{I}$. Since $L \subseteq K \subseteq S'$, then $L \in \mathcal{I}'$ Hence, hereditary property holds. We now prove the following claim.

Claim: $\mathcal{M}/\mathcal{B}$ is a matroid and $\mathrm{r}_{\mathcal{M}/\mathcal{B}}(A) = \mathrm{r}_{\mathcal{M}}(A \cup \mathcal{B}) - \mathrm{r}_{\mathcal{M}}(B)$.

Proof: Let $A \subseteq S \setminus \mathcal{B}$ and let J' be an $\mathcal{M}/\mathcal{B}$ basis of A. Then, $J \cup J' \in \mathcal{I}$. We claim that $J \cup J'$ is an $\mathcal{M}$-basis of $A \cup \mathcal{B}$. Suppose there exists $e \in A \cup B$ such that $J \cup J' \cup \{e\} \in \mathcal{I}$. If $e \in \mathcal{B}$ then $J \cup \{e\} \in \mathcal{I}$ which contradicts the choice of J and if $e \notin \mathcal{B}$, then $J' \cup \{e\} \in \mathcal{I}$ which contradicts the choice of J'. Hence, $J \cup J'$ is an $\mathcal{M}$-basis of $A \cup \mathcal{B}$. Hence, $|J \cup J'| = \mathrm{r}_{\mathcal{M}}(A \cup B)$. Hence, $|J'| = \mathrm{r}_{\mathcal{M}/\mathcal{B}}(A) = |J \cup J'| - |J| = \mathrm{r}_{\mathcal{M}} A \cup B - \mathrm{r}_{\mathcal{M}}(\mathcal{B})$. ■

It follows that $\mathcal{M}/J$ is a matroid.

5 Disjoint Union:[2] Recall that if $\mathcal{M}_i = (S_i, \mathcal{I}_i)$ be matroids and if S_i are distinct for all $i = 1, \ldots, k$ then the union of these matroids is a direct sum and $\bigoplus_{i=1}^k \mathcal{M}_i = \mathcal{M}_1 \oplus \cdots \oplus \mathcal{M}_k =$

$\overline{\mathcal{M}} = (S, \mathcal{I})$. We will show that $\overline{\mathcal{M}}$ is a matroid where

$$S = \bigcup_{i=1}^{k} S_i, \quad \mathcal{I} = \bigcup_{i=1}^{k} \mathcal{I}_i \quad \text{and} \quad A \in \mathcal{I}' \iff A = \bigcup_{i=1}^{k} A_i \text{ where } A_i \in \mathcal{I}_i \text{ for } i = 1, \ldots, k.$$

Exercise 2.3.32: Show $\overline{\mathcal{M}} = (S, \mathcal{I})$ is an independence system. $\triangleleft$

Let $A \subseteq S$. Consider a basis $\mathcal{B}$ in $\overline{\mathcal{M}} = \mathcal{M}_1 \oplus \cdots \oplus \mathcal{M}_k$ of A. We have $\mathcal{B}_j = \mathcal{B} \cup S_j \in \mathcal{I}_j$. $\mathcal{B}_j$ is a basis of $A \cap S_j$ in $\mathcal{M}_j$. Since if $\mathcal{B}_j$ isn't maximal, then there exists $e \in (A \cap S_j) \setminus \mathcal{B}_j$ such that $\mathcal{B}_j \cup \{e\} \in \mathcal{I}_j$ which implies $\mathcal{B} \cup \{e\} \in \mathcal{I}$ but this contradicts the maximality of $\mathcal{B}$. We have

$$\mathcal{B} = \sum_{j=1}^{k} |\mathcal{B}_j| = \sum_{j=1}^{k} \mathrm{r}(A \cap S_j).$$

Hence, every basis of A in $\overline{\mathcal{M}}$ has same size. Hence, $\overline{\mathcal{M}}$ is a matroid. $\triangleleft$

[2] The part about disjoint union was covered in another lecture (L10, on Feb. 2020) but it was included in this list for the sake of completeness.

Chapter 3 – Dynamic Programming

3.1 Weighted Interval Scheduling

We will consider an example of *weighted interval scheduling*. Given n tasks where each task has a start time, s_i, and finish time, f_i and value v_i for all $i = 1, \ldots, n$. At most one task can be executed at each point in time and if start and finish times are same for some tasks, they can be executed at the same time. We want to find subset of tasks S to be executed maximizing $\sum_{j \in S} v_j$.

Example 3.1.1: Consider these 5 tasks and their visual representation below.

j	s_j	t_j	v_j
1	0	3	2
2	1	5	3
3	3	7	4
4	2	9	10
5	6	9	8

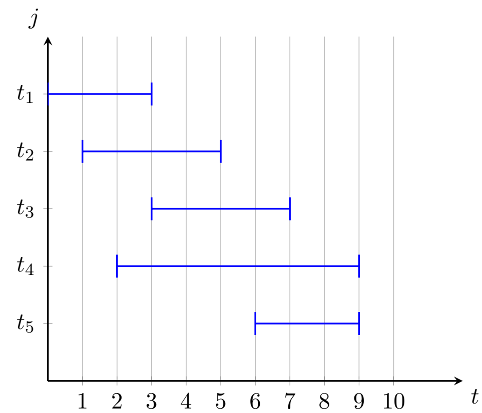


Figure 3.1.1: Graph of tasks.

Let $S = \{1, \ldots, n\}$ and $\mathcal{I} = \{A \subseteq S \mid A \text{ can be all scheduled tasks in a feasible way}\}$. We can show that $(S, \mathcal{I})$ is an independence set but not a matroid. There can exist tasks t_1, t_2 and t_3 as follows.

j	s_j	t_j	v_j
1	1	3	v_1
2	3	5	v_2
3	1	5	v_3

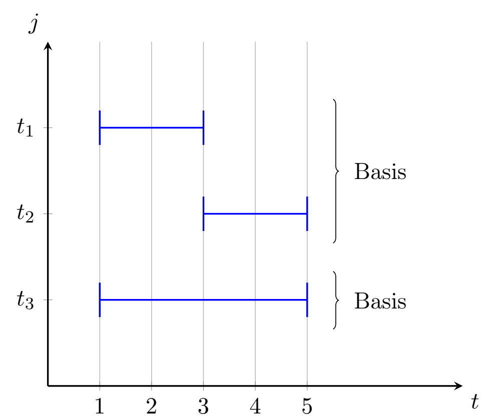


Figure 3.1.2: Graph of t_1, t_2 and t_3.

In this case both $\{t_1, t_2\}$ and $\{t_3\}$ are bases but they have different cardinalities. ◁

To solve this problem, we first assume the tasks are sorted with respecting to their finishing time in ascending order. If they were not ordered, we can order them in $n \log n$ time. We have n tasks ordered in a way so that

$$f_1 \leq \cdots \leq f_n.$$

Let

$$p(j) = \begin{cases} \max\{i < j \mid f_i \leq s_j\}, \\ 0 \text{ if none exists for all } j = 1, \ldots, n. \end{cases}$$

So, $p(j)$ is the last job that can be possibly scheduled with task j. In the above example we have

$$p(1) = 0, \quad p(2) = 0, \quad p(3) = 1, \quad p(4) = 0, \quad p(5) = 2.$$

We see that in an optimal solution, either we perform task n or we don't. This is a very obvious observation but it helps us construct algorithms to solve this problem.

Suppose we use task n. Then, we cannot use tasks $p(n) + 1, \ldots, n - 1$ and we can use tasks $1, \ldots, p(n)$ since for all $k = 1, \ldots, p(n)$, we have $f_k \leq f_{n-1} \leq s_n$. So in this example, if we use task 5, then we cannot use task 3 and task 4 but we can use tasks 1 and 2.

Let $\mathrm{OPT}(j)$ be the optimal value (not the optimal solution) for instance with tasks $1, \ldots, j$. We have

$$\mathrm{OPT}(n) = v_n + \mathrm{OPT}(p(n)).$$

If we don't use task n, then we have $\mathrm{OPT}(n-1) = \mathrm{OPT}(n)$. This approach allows us to break up the problem into smaller problems. In general, when we implement the algorithm we have

$$\begin{aligned} &\mathrm{OPT}(0) = 0, \\ &\mathrm{OPT}(j) = \max\{v_j + \mathrm{OPT}(p(j)), \mathrm{OPT}(j-1)\}. \end{aligned}$$

Using this approach we can compute a recursive OPT function with the following recursion tree.

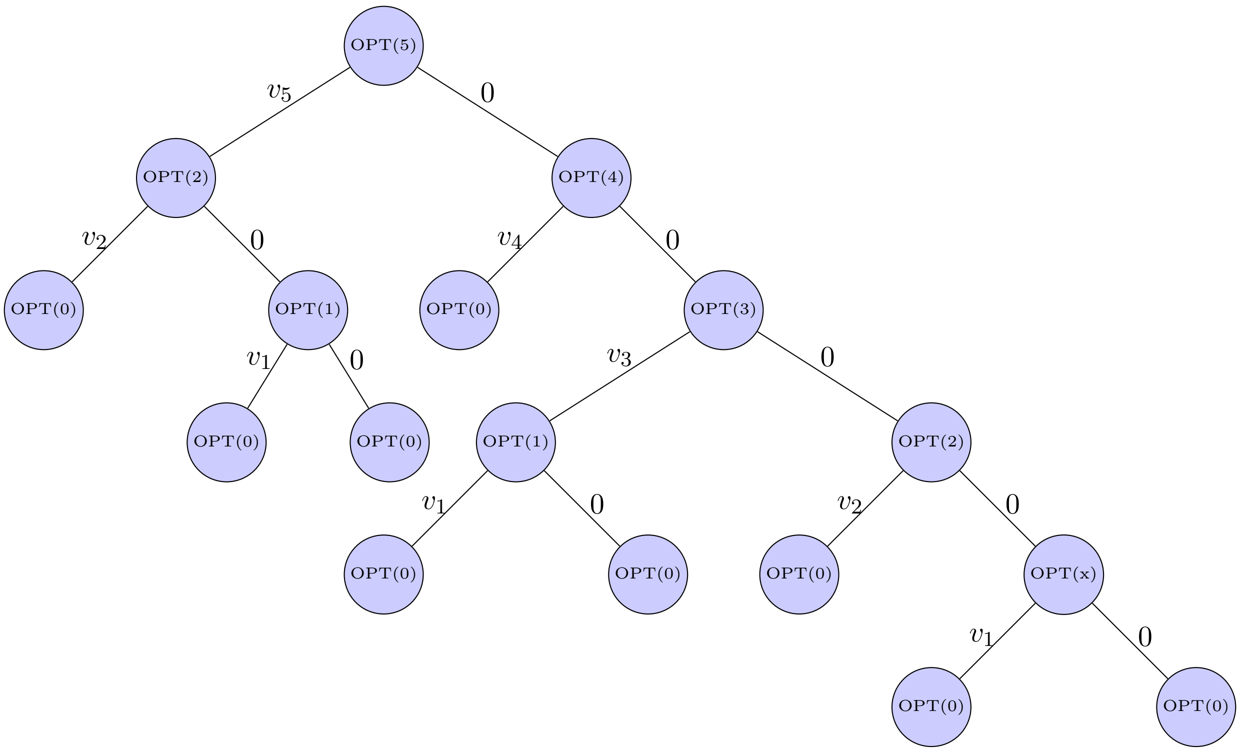


Figure 3.1.3: Recursion tree.

This procedure reuses many of the results it calculates. To be more efficient, we want to store the OPT(j) for all j in a table. We consider the following function for our algorithm.

Algorithm 3.1.2: Computing optimal value of j recursively.

Function `Compute_OPT`(j):
    if $j = 0$ **then**
        return 0
    else if $M[j]$ *has been computed* **then**
        return $M[j]$
    else
        $M[j] = \max\{v_j + \texttt{Compute\_OPT}(p(j)), \texttt{Compute\_OPT}(j\text{-}1)\}$
        return $M[j]$

Alternatively, we have the iterative version as follows.

Algorithm 3.1.3: Computing optimal value of j iteratively.

for $j = 0, \ldots, n$ **do**
    if $j = 0$ **then**
        $M[j] = 0$
    else
        $M[j] = \max\{v_j + M[p(j)], M[j-1]\}$
return $M[n]$

This clearly runs in $O(n)$. So we have a polytime algorithm to solve the problem. This process of remembering (caching) results is called ***memoization***. This algorithm gives optimal value. To get the optimal solution, we store the decision algorithm made in $S[j]$ as follows.

$$\begin{aligned} &0 \text{ if } v_j + M[p(j)] > M[j-1], \\ &0 \text{ otherwise.} \end{aligned}$$

This gives us the following function algorithm.

Algorithm 3.1.4: Finding optimal solution recursively.

```
Function Find_Soln(S,j):
    if j = 0 then
        return ∅
    else if S[j] = 1 then
        return Find_Soln(S,p(j)) ∪ {j}
    else
        return Find_Soln(S,j − 1)
```

Alternatively, we have iterative version of this algorithm as below.

Algorithm 3.1.5: Finding optimal solution iteratively.

```
k ← n
Sol ← ∅
while k > 0 do
    if S[k] = 1 then
        Sol ← Sol ∪ {k}
        k ← p(k)
    else
        k ← k − 1
return Sol
```

3.1.1 Dynamic Programming Overview

1. Write optimal solution to a subproblem as a function of a small number of subproblems (Bellman equation).
2. The total number of subproblems needed is "small".
3. Store (memoize) optimal solutions of previously computed subproblems.

3.1.2 Knapsack Problem

Given n items with weights $a_j \in \mathbb{Z}^+$, profits $c_j \in \mathbb{Z}^+$ and a Knapsack capacity b. We want to find a subset S of items that maximizes $\sum_{j \in S} c_j$ subject to $\sum_{j \in S} a_j \leq b$.

WLOG, we order items $1, \dots, n$ and let $\mathrm{OPT}(i, w)$ be the optimal solution using items $1, \dots, i$ and backpack (knapsack) capacity w. We want to find $\mathrm{OPT}(n, b)$.

Case 1: $\mathrm{OPT}(i, w)$ uses i. Then, $\mathrm{OPT}(i, w) = \mathrm{OPT}(i-1, w-a_i) + c_i$.
Case 2: $\mathrm{OPT}(i, w)$ does not use i. Then, $\mathrm{OPT}(i, w) = \mathrm{OPT}(i-1, w)$.

- if $i > 1$ and $a_i \le w$, then $\mathrm{OPT}(i, w) = \max\{\mathrm{OPT}(i-1, w-a_i) + c_i, \mathrm{OPT}(i-1, w)\}$,
- if $i > 1$ and $0 \le w \le a_i$, then $\mathrm{OPT}(i, w) = \mathrm{OPT}(i-1, w)$,
- if $i = 1$ and $a_1 \le w$ then $\mathrm{OPT}(i, w) = c_1$,
- otherwise, $\mathrm{OPT}(i, w) = 0$.

Note that case 2 leads us to the following recursive definition

$$\mathrm{OPT}(i, w) = \max \begin{cases} \max \begin{Bmatrix} \mathrm{OPT}(i-1, w-a_1) + c_i, \\ \mathrm{OPT}(i-1, w) \end{Bmatrix} & \text{if } i > 1 \quad \text{and} \quad a_i \le w, \\ \mathrm{OPT}(i-1, w) & \text{if } i > 1 \quad \text{and} \quad 0 \le w \le a_i, \\ c_i & \text{if } i = 1 \text{ and } a_1 \le w, \\ 0 & \text{otherwise.} \end{cases}$$

Remark 3.1.6: $\mathrm{OPT}(i, w)$ has $O(nb)$ entries and it takes $O(1)$ time to compute. Hence, the runtime is $O(nb)$.

We have $1 \le i \le n$ and $1 \le w \le b$. This is not polytime since input size is measured in $\log b$, not b. If $b \in O(n^k)$ for some fixed k, then the algorithm above is a *pseudo-polytime algorithm.* ◁

Definition 3.1.7: If a numeric algorithm runs in polytime in the numeric value of the input (the largest integer present in the input) but not necessarily in the length of the input (the number of bits to represent it) then it runs in ***pseudo-polynomial time*** (pseudo-polytime for short). ◁

One perspective on dynamic programming is that we have a memoization table to compute and each table entry defines a *state*. Each state is determined by optimal solutions to some previous states. Imparts a *partial order* on states.

Example 3.1.8: Consider the Knapsack problem with 3 items and capacity 5. Let

$$\begin{aligned} a_1 &= 2, & c_1 &= 3, \\ a_2 &= 1, & c_2 &= 2, \\ a_3 &= 5, & c_3 &= 4. \end{aligned}$$

We can construct a directed graph for this problem. Solution to our dynamic program can be found by computing the longest path from s to t (or shortest path if we multiply costs by -1). ◁

3.1.3 Shortest Paths

Definition 3.1.9: A ***directed graph*** or ***digraph*** is an ordered pair $D = (V, A)$ where V is a set of *vertices* and A is a set of ordered pairs of vertices, called *arcs*, *directed edges* or *arrows.*

An ***arc*** $a = (x, y)$ is considered to be directed from x to y and

- x is called the *tail* of the arc and x is said to be a *direct predecessor* of y,
- y is called the *head* of the arc and y is said to be a *direct successor* of y and y is *reachable* from x. $\triangleleft$

Given a directed graph $D = (V, A)$ with non-negative arc costs c_a for all $a \in A$ and vertices $s, t \in V$. We want to find an s-t path P which minimizes $\sum_{a \in A(P)} c_a$.

Definition 3.1.10: Let $D = (V, A)$ be a directed graph. Let $\varnothing \subseteq S \subseteq V$. We define

$$\begin{aligned}\delta^+(S) &= \{(u, v) \in A \mid u \in S, v \notin S\} \text{ (the set of arcs leaving } S\text{)},\\ \delta^-(S) &= \{(u, v) \in A \mid u \notin S, v \in S\} \text{ (the set of arcs entering } S\text{)}.\end{aligned}$$

The set $\delta(S)$ is called ***the cut induced by*** S. We have $\delta(S) = \delta^+(S) \cup \delta^-(S)$. $\triangleleft$

Example 3.1.11: Consider the directed graph $D = (V, A)$ below.

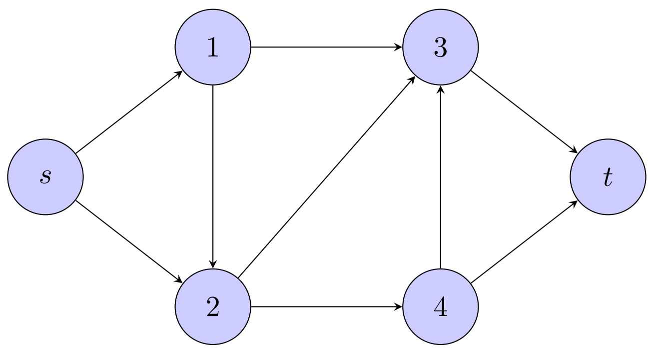


Figure 3.1.4: Directed graph $D = (V, A)$.

For $S = \{2, 3, 4\}$ we have

$$\begin{aligned}\delta^+ &= \{(3, t), (4, t)\},\\ \delta^- &= \{(s, 2), (1, 2), (1, 3)\}.\end{aligned}$$ $\triangleleft$

3.1.3.1 Dijkstra's Algorithm

For a directed graph $D = (V, A)$, keep a set $S \subseteq S$ of vertices for which we know the shortest s-v path for all $v \in S$.

Algorithm 3.1.12: Dijkstra's algorithm.

$S \leftarrow \{s\}$
$\text{OPT}(s) \leftarrow 0$, $\text{OPT}(v) \leftarrow \infty$ for all $v \neq s$
while $S \neq V$ **do**
    find $(u, v) \in \delta^+(S)$ with smallest $c_{uv} + \text{OPT}(u)$
    $\text{OPT}(v) \leftarrow c_{uv} + \text{OPT}(u)$
    $S \leftarrow S \cup \{v\}$
return $\text{OPT}(t)$ (where t is the last vertex added to S)

Example 3.1.13: Consider the directed graph $D = (V, A)$ with costs below.

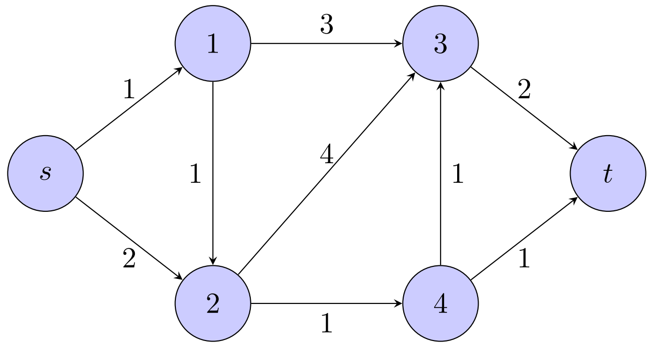


Figure 3.1.5: Directed graph $D = (V, A)$ with given costs.

We obtain the following (in the given order during while loop) by using Dijkstra's algorithm.

1. $S = \{s\}$,
2. $S = \{s, 1\}$ and $\mathrm{OPT}(1) = 1$,
3. $S = \{s, 1, 2\}$ and $\mathrm{OPT}(2) = 2$,
4. $S = \{s, 1, 2, 4\}$ and $\mathrm{OPT}(4) = 3$,
5. $S = \{s, 1, 2, 4, 3\}$ and $\mathrm{OPT}(3) = 4$,
6. $S = \{s, 1, 2, 4, 3, t\}$ and $\mathrm{OPT}(t) = 4$ (since $\mathrm{OPT}(4) = 3$ and $c_{4t} = 1$, so $\mathrm{OPT}(t) = \mathrm{OPT}(4) + c_{4t} = 3 + 1 = 4$).

$\triangleleft$

To show the correctness of Dijkstra's algorithm, we prove the following claim.

Claim 3.1.14: At any point of execution, for all $v \in S$, $\mathrm{OPT}(v)$ is the shortest s-v path length.

Proof: We use induction on $|S|$. For $|S| = 1$ the claim holds since the path s-s has length 0. Suppose claim holds for $|S| = k$. We want to show it also holds for $k + 1$. Consider the step when algorithm chooses v to add to S. Let (u, v) be the arc used at this stage. Note that (u, v) was chosen to minimize $c_{uv} + \mathrm{OPT}(u)$. Suppose, for contradiction, there exists a shorter s-v path. Let y be the first vertex on this path not in S such that x precedes y.

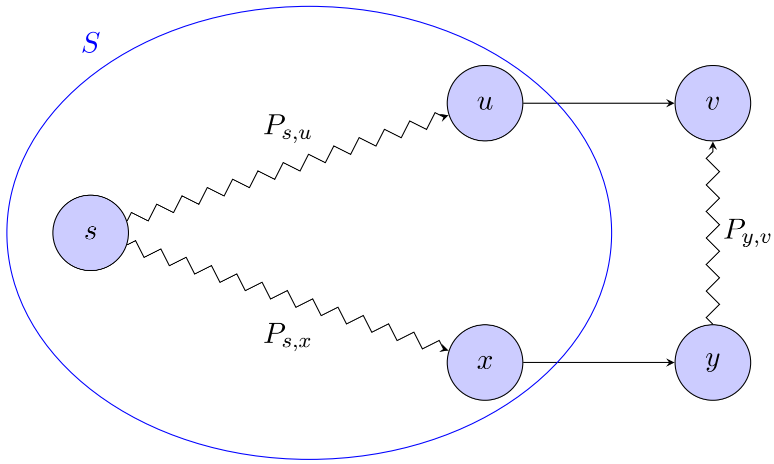


Figure 3.1.6: Example illustrating shorter s-v path where zigzags denote paths and arrows denote arcs.

By assumption we have

$$c(P_{s,u}) + c_{uv} > c(P_{s,x}) + c_{xy} + \underbrace{c(P_{y,v})}_{>0} \geq \text{OPT}(x) + c_{xy},$$

but this is a contradiction since the algorithm should have chosen (x, y) instead of (u, v). □

Recall Dijkstra's algorithm in algorithm 3.1.12.

Remark 3.1.15: Dijkstra's algorithm runs in polytime. The while loop runs in $O(n)$ and finding smallest $c_{uv} + \text{OPT}(u)$ runs in $O(m)$ so the algorithm runs in $O(mn)$ time where $|A| = m$ and $|V| = n$. ◁

Example 3.1.16: Dijkstra's algorithm can fail if there exists negative cost arcs. Consider the directed graph $D = (V, A)$ with gives costs below.

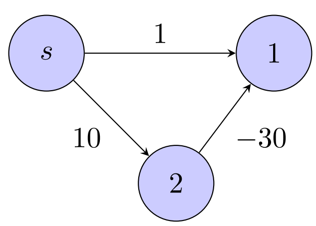


Figure 3.1.7: Directed graph $D = (V, A)$ with given costs.

The algorithm first finds $\text{OPT}(s) = 0$, then $\text{OPT}(1) = 1$ and then $\text{OPT}(2) = 10$ but this is wrong since the path s-2-1 has cost -20. ◁

3.1.3.2 Shortest Paths Without Negative Cycles

We assume that there does not exist a directed cycle $v_1, \ldots, v_k$ where v_k.

Definition 3.1.17: A ***directed cycle*** is a non-empty directed walk in which all arcs are distinct where first and last vertices are the same. A ***cycle's cost*** is the sum of all costs of its edges or arcs. ◁

Remark 3.1.18: For a directed graph $D = (V, A)$, if there are no negative cost directed cycles, then there exists a minimum cost shortest walk with no cycles. We want to find the shortest s-t directed path. ◁

Example 3.1.19: Suppose there are no negative cost cycles exist and let W_1 be the shortest walk from 1 to 7 where

$$W_1 : \text{1-2-4-3-2-6-7}.$$

Note that by assumption we also have

$$W_2 : \text{1-2-6-7}$$

where $\text{cost}(W_2) \leq \text{cost}(W_1)$ since $\text{cost}(W_1) = \text{cost}(W_2) + \underbrace{\text{cost(2-4-3-2)}}_{\geq 0} \geq \text{cost}(W_2)$. ◁

Remark 3.1.20: We observe that cost of shortest s-t path is at least as large as the cost of the shortest s-t walk. Hence, by the above remark we find that

$$\text{cost of shortest } s\text{-}t \text{ path} = \text{cost of shortest } s\text{-}t \text{ walk}.$$

Hence, we can solve finding shortest s-t path problem by finding shortest s-t walk with $n-1$ edges or arcs. ◁

Let $\text{OPT}(i, v)$ be the shortest s-v walk using at most i edges. Then,

- $\text{OPT}(0, v) = \infty$ for all $v \in V \setminus \{s\}$ (Bellman equation),
- $\text{OPT}(0, s) = 0$,
- $\text{OPT}(i, v) = \min \left\{ \begin{matrix} \text{OPT}(i-1, v), \\ \min\limits_{\substack{u \in V \\ \text{s.t.} (u,v) \in A}} \{\text{OPT}(i-1, u) + c_{uv}\} \end{matrix} \right\}.$

Remark 3.1.21: We need to compute $\text{OPT}(i, v)$ for all $v \in V$ and $i = 0, \ldots, n-1$. This takes $O(n^2)$ time. Computing each entry takes $O(n)$ time so the whole procedure is in $O(n^3)$, so it's in polytime. Note that this can be implemented in $O(mn)$ time. We can also show the correctness of this algorithm with inductive arguments. ◁

Exercise 3.1.22: Show the correctness of above algorithm. ◁

Example 3.1.23: Consider the directed graph $D = (V, E)$ below with given costs on the left. We have its table on the right as follows.

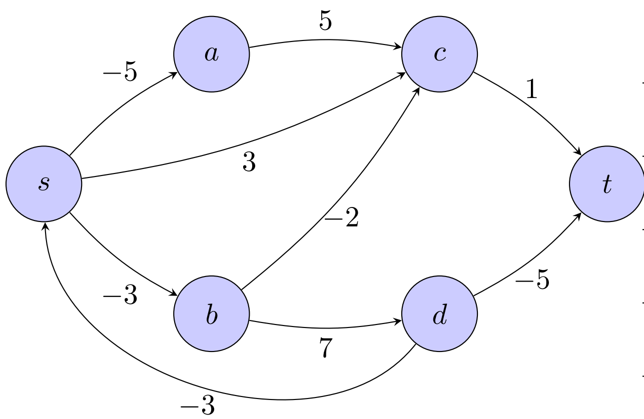


Figure 3.1.8: Directed graph $D = (V, A)$ with given costs.

$v \backslash i$	0	1	2	3	4	5
t	∞	∞	4	$-4^{\dagger}$	-4	-4
d	∞	∞	4	4	4	4
c	∞	3	$-5^{\dagger\dagger}$	-5	-5	-5
b	∞	-3	-3	-3	-3	-3
a	∞	-5	-5	-5	-5	-5
s	0	0	0	0	0	0

Consider the calculations in † and ††. Clearly, if we use at most use 2 arcs to get to t, then $\text{OPT}(t) = 3 + 1 = 4$. If we use 3 arcs, then we take

$$\begin{aligned}
\text{OPT}(3, t) &= \min\left\{\text{OPT}(2, t), \min_{u \in V \text{s.t.} (u,t) \in A} \{\text{OPT}(i-1, u) + c_{ut}\}\right\} \\
&= \min\{4, \min\{-5+5+1, -3-2+1, -3+7-5\}\}, \\
&= \min\{4, -4\}, \\
&= -4.
\end{aligned}$$

Similarly, to get to c with using at most 2 arcs, have

$$\begin{aligned}
\text{OPT}(2, c) &= \min\left\{\text{OPT}(1, c), \min_{u \in V \text{s.t.} (u,c) \in A} \{\text{OPT}(i-1, u) + c_{uc}\}\right\} \\
&= \min\{3, \min\{-5+5, -3-2\}\}, \\
&= \min\{3, -5\}, \\
&= -5.
\end{aligned}$$

Note that the constraint $u \in V$ s.t. $(u, v) \in A$ for minimizing $\text{OPT}(i, v)$ looks every arc entering v to give a minimum cost path. Hence, we find that the shortest s-t path in this example is s-b-c-t with cost -4. ◁

Remark 3.1.24: From our observations we see that we can consider dynamic programs as shortest path problems. ◁

Chapter 4 – Complexity Theory

Complexity theory tries to address the question of if there exists a polytime algorithm to solve a problem of interest.

4.1 Polytime Reductions

Definition 4.1.1: Given two problems X and Y, we say Y is ***polytime reducible to*** X, denoted by $Y \leq_p X$, if there exists an algorithm to solve instances of Y of input size n that does

1. poly(n) basic operations,
2. poly(n) many calls to an algorithm that solves problem X. ◁

Example 4.1.2: We have seen that

$$\begin{aligned}&\text{finding maximum cost forest} \leq_p \text{MST problem, and}\\&\text{MST problem} \leq_p \text{finding maximum cost forest.}\end{aligned}$$

◁

Remark 4.1.3: If there exists a polytime algorithm to solve X and $Y \leq_p X$, then there exists a polytime algorithm to solve Y. Conversely, if there does not exist a polytime algorithm to solve Y and if $Y \leq_p X$, then there does not exist a polytime algorithm to solve X.

This definition implies that input to solving problem X must be poly(k) in time. ◁

4.1.1 Examples of Polytime Reducible Problems

Definition 4.1.4: Let $G = (V, E)$ be a graph. An ***independent set*** $S \subseteq V$ ***in*** G is a set such that for all $u, v \in S$, we have $uv \notin E$. That is, there are no edges that connects any two vertices in S. ◁

Definition 4.1.5: Let $G = (V, E)$ be a graph. A ***clique*** $S \subseteq V$ ***in*** G is a set such that for all distinct $u, v \in S$, we have $uv \in S$. That is, every vertex in S is connected. ◁

Example 4.1.6: Consider the graph $G = (V, E)$ below.

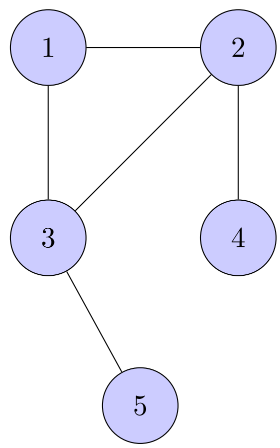


Figure 4.1.1: $G = (V, E)$.

Here we have that
- $\{1, 2, 3\}$ is a clique,
- $\{1, 4, 5\}$ is an independent set.

◁

Example 4.1.7: Independent set problem, IND-SET(G, k), reduces to clique problem, CLIQUE(G, k). So,

$$\text{IND-SET} \leq_p \text{CLIQUE}.$$

We have

Algorithm: IND-SET(G, k)

Input : $G = (V, E)$, $k \in \mathbb{Z}^+$
Output: YES if G has ind. set of size at least k, NO otherwise

Algorithm: CLIQUE(G, k)

Input : $G = (V, E)$, $k \in \mathbb{Z}^+$
Output: YES if there exists a clique in G of size at least k, NO otherwise

To see that IND-SET $\leq_p$ CLIQUE, we can use the following algorithm.

Algorithm 4.1.8: Calling CLIQUE to solve IND-SET(G, k)

Input : $G = (V, E)$, $k \in \mathbb{Z}^+$

1 Construct $\overline{G} = (V, E)$ so that $uv \in \overline{E} \iff uv \notin E$ (so $\overline{G}$ is complement of G)
2 **return** CLIQUE$(\overline{G}, k)$

Similarly we can verify CLIQUE $\leq_p$ IND-SET. So, if we find a solution to either of these problems, we can also solve the other one. ◁

Example 4.1.9: IND-SET(G, k), reduces to maximum independent set problem, MAX-IND-SET(G). So,

$$\text{IND-SET} \leq_p \text{MAX-IND-SET}.$$

We have

Algorithm: MAX-IND-SET(G)

Input : $G = (V, E)$
Output: Ind. set of largest size

To see that $\textsc{Ind-Set} \leq_p \textsc{Max-Ind-Set}$, we can use the following algorithm.

Algorithm 4.1.10: Calling $\textsc{Max-Ind-Set}$ to solve $\textsc{Ind-Set}(G, k)$

Input : $G = (V, E)$, $k \in \mathbb{Z}^+$

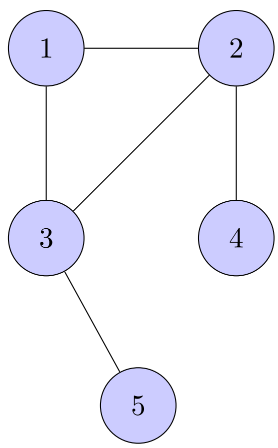
$S \leftarrow \textsc{Max-Ind-Set}(G)$

return $\textsc{Yes} \iff |S| \geq k$

$\triangleleft$

Definition 4.1.11: Let $G = (V, E)$ be a graph. A ***vertex cover*** $S \subseteq V$ ***of*** G is a set such that for all $e \in E$, we have $|e \cap S| \geq 1$. That is, every edge of G has an end point in S. $\triangleleft$

Example 4.1.12: Consider the graph $G = (V, E)$ below.



Here $\{2, 3\}$ is a vertex cover of G.

Figure 4.1.2: $G = (V, E)$.

$\triangleleft$

Lemma 4.1.13: Let $G = (V, E)$ be a graph. Then $S \subseteq V$ is an independent set if and only if $\overline{S} = V \setminus S$ is a vertex cover.

Proof: Suppose $S \subseteq V$ is an independent set and suppose, for contradiction, $\overline{S}$ is not a vertex cover. Then, there exists $uv \in E$ such that $u, v \notin \overline{S}$. Then $u, v \in S$ but u and v is connected in S which contradicts that S is an independent set. Conversely, suppose $\overline{S}$ is a vertex cover and consider $S \subseteq V$. Suppose, for contradiction, there exists $uv \in E$ such that $u, v \in S$. Then $\{u, v\} \cap \overline{S} = \varnothing$ but this contradicts that $\overline{S}$ is a vertex cover. $\square$

Example 4.1.14: $\textsc{Ind-Set}(G, k)$, reduces to vertex cover problem, $\textsc{Vtx-Cover}(G, k)$. So,

$$\textsc{Ind-Set} \leq_p \textsc{Vtx-Cover}.$$

We have

Algorithm: $\textsc{Vtx-Cover}(G, k)$

Input : $G = (V, E)$

Output: $\textsc{Yes}$ if G has a vertex cover of size at least k, $\textsc{No}$ otherwise

To see that IND-SET $\leq_p$ VTX-COVER, we can use the following algorithm.

Algorithm 4.1.15: Calling VTX-COVER to solve IND-SET(G, k)

Input : $G = (V, E)$, $k \in \mathbb{Z}^+$

Call VTX-COVER$(G, n-k)$ where $|V| = n$
return VTX-COVER$(G, n-k)$

$\triangleleft$

Definition 4.1.16: Let $U = \{1, \ldots, n\}$ be a finite set and let $\mathcal{C}$ be a collection of subsets of U. We say $\mathcal{C}$ is a ***set cover of*** U if $\bigcup_{S \in \mathcal{C}} S = U$. Given a collection subsets $S_1, \ldots, S_m \subseteq U = \{1, \ldots, n\}$, the set cover problem tries to find the smallest set cover I of U such that $\bigcup_{i \in I} S_i = U$. $\triangleleft$

Example 4.1.17: VTX-COVER(G, k), reduces to set cover problem, SET-COVER(x). So,

$$\text{VTX-COVER} \leq_p \text{SET-COVER}.$$

We have

Algorithm: SET-COVER$(U, S_1, \ldots, S_m, k)$

Input : $U, S_1, \ldots, S_m, k \in \mathbb{Z}^+$ where $U = \{1, \ldots, n\}$ and $S_i \subseteq U$ for $i = 1, \ldots, m$
Output: YES if $I \subseteq \{1, \ldots, m\}$ such that $\bigcup_{i \in I} S_i = U$ and $|I| \leq k$, NO otherwise

To see that VTX-COVER $\leq_p$ SET-COVER, we can use the following algorithm.

Algorithm 4.1.18: Calling SET-COVER to solve VTX-COVER(G, k)

Input : $G = (V, E)$, $k \in \mathbb{Z}^+$

$U \leftarrow E$
$S_v \leftarrow \{e \in E \mid e \in \delta(v)\}$, that is, S_v is the set of edges that are incident to v for all $v \in V$
Call SET-COVER$(U, \{S_v\}_{v \in V}, k)$
return SET-COVER$(U, \{S_v\}_{v \in V}, k)$

Since IND-SET $\leq_p$ VTX-COVER and VTX-COVER $\leq_p$ SET-COVER then IND-SET $\leq_p$ SET-COVER. $\triangleleft$

Definition 4.1.19: A ***clause*** c is a finite disjunction of terms t_i where each term t_i is either x_j or its complement, $\overline{x_j}$. i.e. each term is a ***literal***. We say the clause c is ***satisfied*** if given an assignment of values $t_1, \ldots, t_\ell$ at least one of t_i is true where

$$c = t_1 \vee \cdots \vee t_\ell.$$

A ***satisfying assignment*** in a problem with clauses $c_1, \ldots, c_m$ is an assignment that satisfies all c_i for $i = 1, \ldots, m$. $\triangleleft$

Example 4.1.20: Consider literals x_1, x_2, x_3, x_4 and clauses

$$\begin{aligned} c_1 &= x_1 \vee \overline{x}_2, \\ c_2 &= \overline{x}_1 \vee \overline{x}_3 \vee x_4, \\ c_3 &= x_3 \vee \overline{x}_4. \end{aligned}$$

The assignment $x = (1, 0, 0, 1)$ is not a satisfying assignment because it

- satisfies c_1 since $1 \vee 1 = 1$,
- satisfies c_2 since $0 \vee 1 \vee 1 = 1$,
- does not satisfy c_3 since $0 \vee 0 = 0$.

The assignment $x = (1, 0, 1, 1)$ is a satisfying assignment since it satisfies c_1, c_2 and c_3. $\triangleleft$

Example 4.1.21: 3-SAT problem reduces to IND-SET. So,

$$\text{3-SAT} \leq_p \text{IND-SET}.$$

We have

Algorithm: 3-SAT$(x_1, \ldots, x_n, c_1, \ldots, c_m)$ where

Input : $x_1, \ldots, x_n$ (literals) and $c_1, \ldots, c_m$ clauses of length 3.
Output: YES if there exists a satisfying assignment for all c_i for $i = 1, \ldots, m$, NO otherwise

Before verifying this, we show an example of converting a 3-SAT problem into an independent set problem. $\triangleleft$

Example 4.1.22: Let $x_1, \ldots, x_5$ be literals with clauses of length 3 as follows.

$$c_1 = x_1 \vee x_2 \vee \overline{x}_3,$$
$$c_2 = \overline{x}_2 \vee x_4 \vee \overline{x}_5,$$
$$c_3 = x_1 \vee \overline{x}_2 \vee x_5.$$

Here each clause has length 3, so each clause has 3 terms (literals). For each j-th literal in each clause c_i, we put a vertex v_{ij} and connect vertices that belong to same clause with an edge as follows.

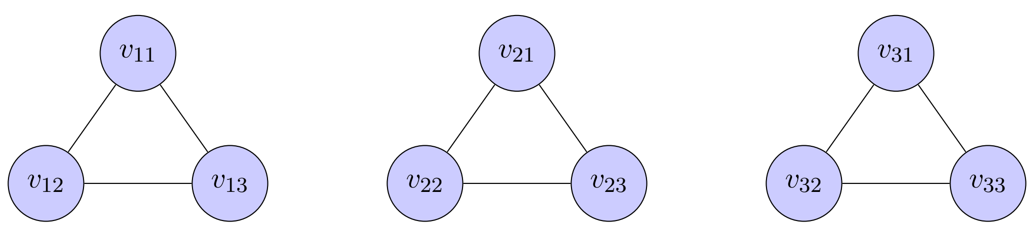


Figure 4.1.3: Constructing a graph from SAT problem.

We then connect vertices v_{ij} if j-th literal in c_i cannot be true in all clauses for all $i = 1, \ldots, m$. That is, we connect v_{ij_1} and $v_{\ell j_2}$ if there exists clauses c_i and c_ℓ such that v_{ij_1} and $v_{\ell j_2}$ correspond to same literal x_j where $x_j \in c_i$ and $\overline{x}_j \in c_\ell$. We obtain the following graph.

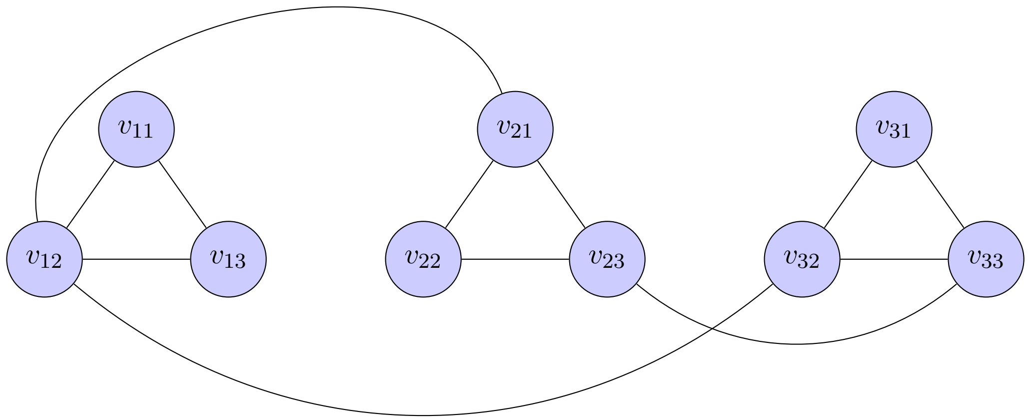


Figure 4.1.4: Constructing a graph from 3-SAT problem.

◁

Remark 4.1.23: We see that when we construct a graph $G = (V, E)$ as described above, any two vertex v_{ij} and v_{ik} connected. Hence, if there are m clauses and n literals, the maximum size independent set in G is of size m since any independent set cannot contain more than two vertices that belong to same clause. Moreover, any independent set of G with size m contains exactly one vertex from every clause. ◁

Example 4.1.24: We now show Example 4.1.21 and show 3-SAT $\leq_p$ IND-SET.

Algorithm 4.1.25: Calling IND-SET to solve 3-SAT$(x_1, \ldots, x_n, c_1, \ldots, c_m)$

Input : $x_1, \ldots, x_n$ (literals) and $c_1, \ldots, c_m$ clauses of length 3

1 Construct the graph $G = (V, E)$ described in Example 4.1.22.
2 **return** IND-SET(G, k)

It is not immediately clear that this algorithm gives the correct result. So, we need to verify its correctness for both YES and NO outputs.

1. If YES, there exists a satisfying assignment $\iff$ if there exists an independent set of size at least m, then there exists a satisfying assignment.

2. If NO, there does not exist a satisfying assignment $\iff$ if there there does not exist an independent set of size at least m, then there does not exist a satisfying assignment.

Hence, it is sufficient to prove the following claim.

Claim 4.1.26: Let $c_1, \ldots, c_m$ be clauses with finite length and let $G = (V, E)$ be the graph obtained from Example 4.1.22. Then, there exists a satisfying assignment for $c_1, \ldots, c_m$ if and only if there exists an independent set in G of size at least m.

Proof: Suppose $x'_1, \ldots, x'_n$ is a satisfying assignment and let $U = \varnothing$. Then, for each clause c_i, there exists at least 1 true term. Pick one such true term per clause arbitrarily and put corresponding vertex in U. We have $|U| = m$. If we were to construct a graph $G = (V, E)$ as described, then every vertex in U is in different triangle. Hence, there exists an edge in E that connects vertices in $v_{i_1 j_1}, v_{i_2 j_2} \in U$ if and only if the j_1-th term in c_{i_1} is complement of the j_2-th term in c_{i_2}. Hence, we cannot have such edge since if this is the case, then one of these literals is false so it cannot be in

U. Hence, U is an independent set in G of size m.

Conversely, let $U \subseteq V$ be an independent set in G of size m. Then, U contains exactly one vertex from each clause and there does not exist an edge that connects any two vertices in U. Hence, no pair of vertices in U can correspond to x_j and $\overline{x}_j$ for any literal x_j. Note that any independent set I in G of size less than m cannot contain one vertex from each clause since there are m clauses. Consider the assignment $x_1', \ldots, x_n'$ obtained by setting terms corresponding to vertices in U as true and other terms as false. This is a satisfying assignment since every vertex in U corresponds to different clause and since $|U| = m$, then each clause is satisfied. Moreover, we cannot have set both x_j and $\overline{x}_j$ to true at the same time since if there exists a vertex corresponding to x_j, say $u_1 \in U$, then there does not exists $u_2 \in U$ that correspond to $\overline{x}_j$ because there exists an edge in E that connects u_1 and u_2 in G and U is an independent set. □

Hence, by the claim above, 3-SAT $\leq_p$ IND-SET. ◁

4.1.2 Classes of P and NP

Definition 4.1.27: A problem X is called a ***decision problem*** if its outputs are YES or NO. The set (or class) of all decision problems for that are solvable in polytime is called P. ◁

Example 4.1.28: The problems 3-SAT, IND-SET, VTX-COVER, SET-COVER etc. are all decision problems. MST problem (that gives MST of a graph) is not a decision problem but the decision version of the MST problem (that answers if there exists a spanning tree of cost at most $k \in \mathbb{Z}$) is a decision problem.

DECISION-MST and DECISION-MAX-COST-FOREST are problems in P but it is not known if IND-SET is in P. ◁

Definition 4.1.29: A ***certifier*** $C(s,t)$ for a decision problem X is an algorithm that for every input s to X,

$$X(s) \text{ is YES} \iff \text{there exists } t \text{ such that } C(s,t) \text{ returns YES.}$$

In this case, t is called a ***Yes certificate***. ◁

Example 4.1.30: Consider the decision problem IND-SET(G,k). If the answer for this problem is yes for given a graph $G = (V,E)$ and $k \in \mathbb{Z}^+$, then one way to validate this answer is to provide $U \subseteq V$ such that $|U| \geq k$ and U is independent. In this case, U is a YES certificate. For 3-SAT$(x_1, \ldots, x_n, c_1, \ldots, c_m)$, a YES certificate is a satisfying assignment $x_1', \ldots, x_n'$. ◁

Definition 4.1.31: A certifier $C(s,t)$ for a decision problem X is called a ***polytime certifier*** if

1. size of t is polynomial in size of s and
2. $C(s,t)$ does polynomially many basic operations in size of s. ◁

Definition 4.1.32: Let X be a decision problem. We say X is in ***non-deterministic polytime*** if there exists a polytime certifier for X. The set (or class) of algorithms that are in non-deterministic polytime is called NP. Equivalently, $X \in \mathsf{NP}$ if there exists a polytime non-deterministic algorithm that solves X.

A *non-deterministic algorithm* is an algorithm that, even for the same input, can exhibit different behaviors on different runs, as opposed to a deterministic algorithm. An algorithm that solves a problem in non-deterministic polynomial time can run in polynomial time or exponential time depending on the choices it makes during execution. The non-deterministic algorithms are often used to find an approximation to a solution, when the exact solution would be too costly to obtain using a deterministic one. $\triangleleft$

Remark 4.1.33: Let X be a decision problem. If $X \in \mathsf{P}$, then X is solvable in polytime. Hence, its certifier is also in polytime (we can just take its certifier as itself). Hence $X \in \mathsf{NP}$. Hence, $\mathsf{P} \subseteq \mathsf{NP}$. It is not known if $\mathsf{NP} \subseteq \mathsf{P}$ but the consensus is $\mathsf{NP} \not\subseteq \mathsf{P}$. $\triangleleft$

4.1.3 NP-Completeness

Definition 4.1.34: A decision problem X is called NP-complete if

1. $X \in \mathsf{NP}$, and
2. for all $Y \in \mathsf{NP}$, $Y \leq_p X$ (so X is at least as hard as any problem in NP-class). $\triangleleft$

Theorem 4.1.35: Let X be NP-complete. There exists a polytime algorithm to solve X if and only if $\mathsf{P} = \mathsf{NP}$.

Proof: Suppose X is a polytime algorithm. Since for all $Y \in \mathsf{NP}$ we have $Y \leq_p X$, then we can use the polytime algorithm which we use to solve X to solve Y in polytime. Hence, $\mathsf{NP} \subseteq \mathsf{P}$. So $\mathsf{P} = \mathsf{NP}$. Conversely, suppose $\mathsf{P} = \mathsf{NP}$. Since $X \in \mathsf{NP}$ then $X \in \mathsf{P}$. $\square$

Theorem 4.1.36 (Cook-Levin '71)**:** Circuit-Sat problem is NP-complete.

Proof: Proof is beyond the scope of this course. We provide this theorem to show existence of NP-complete problems. $\triangleleft$

Theorem 4.1.37: Let X and Y be decision problems. If Y is NP-complete and if

1. $X \in \mathsf{NP}$ and
2. $Y \leq_p X$ (that is, we can solve Y using X as a subroutine),

then X is NP-complete.

Proof: Let $Z \in \mathsf{NP}$ be arbitrary. Since Y is NP-complete then $Z \leq_p Y$. Since $Y \leq_p X$, then we have $Z \leq_p Y \leq_p X$ for any $Z \in \mathsf{NP}$ and since $X \in \mathsf{NP}$, then X is NP-complete. $\square$

Remark 4.1.38: Without proof, we state Circuit-Sat $\leq_p$ 3-Sat. This shows 3-Sat is NP-complete since

1. Circuit-Sat is NP-complete by Cook-Levin '71 theorem,
2. 3-Sat $\in \mathsf{NP}$. Let $x'_1, \ldots, x'_n$ be a certificate for 3-Sat. For each clause c_i, we can check if c_i is satisfied by checking each term in c_i. Since each c_i has length 3 and since there are m clauses, then the certifier for 3-Sat is in polytime. $\triangleleft$

Remark 4.1.39: Note that when showing $Y \leq_p X$, we need to show reduction is correct in both Yes and No outputs. $\triangleleft$

Example 4.1.40: SUBSET-SUM is NP-complete. — This example is long and will be written later. ◁

4.1.4 NP-Hardness

Index