Computer Science Fundamentals:

Intro to Algorithms, Systems, & Data Structures

Christian J. Rudder

October 2024

Contents

This page is left intentionally blank.

Preface

We start at high-level understandings (admittedly somewhat dry with definitions), gaining familiarity with data-structures and algorithms (where the fun begins), number systems (e.g,. binary), which will allow us to dive deeper and create our own theoretical models of computation, i.e., a computer (plenty visuals, less text). After such, we revisit algorithms and data-structures, learning the classic methods and strategies that motivated our modern system today (The must knows for any interview or real-world application).

Big thanks to **Christine Papadakis-Kanaris**
for teaching Intro. to Computer Science II,
Dora Erdos and **Adam Smith**
for teaching BU CS330: Introduction to Analysis of Algorithms,
with contributions from:
S. Raskhodnikova, E. Demaine, C. Leiserson, A. Smith, and K. Wayne,
at Boston University

Please note: These are my personal notes, and while I strive for accuracy, there may be errors. I encourage you to refer to the original slides for precise information.
Comments and suggestions for improvement are always welcome.

Prerequisites

You can skip this: These prerequisites help, but they are not strictly necessary. Don't spend much if any time here. It may serve as a reference later on. Most if not all definitions are self contained and don't skip any logical steps.

However it would be helpful (not strictly) to be familiar with at least one programming language, specifically **Java** as we reference the language a few times. Not that we'll need to code, or will use them, but easily be able to read pseudocode (make-believe code that often strongly resembles real programming languages).

Use this entire text as a supplementary resource—If it doesn't make sense, cross reference other sources,and hopefully comeback and improve this one, as it's open-source.

Theorem 0.1: Common Derivatives

Power Rule: For $n \neq 0$	$\frac{d}{dx}(x^n) = n \cdot x^{n-1}$. E.g., $\frac{d}{dx}(x^2) = 2x$
Derivative of a Constant:	$\frac{d}{dx}(c) = 0$. E.g., $\frac{d}{dx}(5) = 0$
Derivative of ln:	$\frac{d}{dx}(\ln x) = \frac{1}{x}$
Derivative of $\log_a$:	$\frac{d}{dx}(\log_a x) = \frac{1}{x \ln a}$
Derivative of $\sqrt{x}$:	$\frac{d}{dx}(\sqrt{x}) = \frac{1}{2\sqrt{x}}$
Derivative of function $f(x)$:	$\frac{d}{dx}(x) = 1$. E.g., $\frac{d}{dx}(5x) = 5$
Derivative of the Exponential Function:	$\frac{d}{dx}(e^x) = e^x$

Theorem 0.2: L'Hopital's Rule

Let $f(x)$ and $g(x)$ be two functions. If $\lim_{x\to a} f(x) = 0$ and $\lim_{x\to a} g(x) = 0$, or $\lim_{x\to a} f(x) = \pm\infty$ and $\lim_{x\to a} g(x) = \pm\infty$, then:

$$\lim_{x\to a} \frac{f(x)}{g(x)} = \lim_{x\to a} \frac{f'(x)}{g'(x)}$$

Where $f'(x)$ and $g'(x)$ are the derivatives of $f(x)$ and $g(x)$ respectively.

Theorem 0.3: Exponents Rules

For $a, b, x \in \mathbb{R}$, we have:

$$x^a \cdot x^b = x^{a+b} \text{ and } (x^a)^b = x^{ab}$$

$$x^a \cdot y^a = (xy)^a \text{ and } \frac{x^a}{y^a} = \left(\frac{x}{y}\right)^a$$

Note: The := symbol is short for "is defined as." For example, $x := y$ means x is defined as y.

Definition 0.1: Logarithm

Let $a, x \in \mathbb{R}$, $a > 0$, $a \neq 1$. Logarithm x base a is denoted as $\log_a(x)$, and is defined as:

$$\log_a(x) = y \iff a^y = x$$

Meaning *log* is inverse of the exponential function, i.e., $\log_a(x) := (a^y)^{-1}$.

Tip: To remember the order $log_a(x) = a^y$, think, "base a," as a is the base of our *log* and y.

Theorem 0.4: Logarithm Rules

For $a, b, x \in \mathbb{R}$, we have:

$$\log_a(x) + \log_a(y) = \log_a(xy) \text{ and } \log_a(x) - \log_a(y) = \log_a\left(\frac{x}{y}\right)$$

$$\log_a(x^b) = b\log_a(x) \text{ and } \log_a(x) = \frac{\log_b(x)}{\log_b(a)}$$

Definition 0.2: Permutations

Let $n \in \mathbb{Z}^+$. Then the number of distinct ways to arrange n objects in order is $n! := n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$. When we choose r objects from n objects, it's Denoted:

$$^{n}P_{r} := \frac{n!}{(n-r)!}$$

Where $P(n, r)$ is read as "n permute r."

Definition 0.3: Combinations

Let n and k be positive integers. Where order doesn't matter, the number of distinct ways to choose k objects from n objects is it's *combination.* Denoted:

$$\binom{n}{k} := \frac{n!}{k!(n-k)!}$$

Where $\binom{n}{k}$ is read as "n choose k.", and $\binom{\cdot}{\cdot}$, the *binomial coefficient.*

Theorem 0.5: Binomial Theorem

Let a and b be real numbers, and n a non-negative integer. The binomial expansion of $(a+b)^n$ is given by:

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$$

which expands explicitly as:

$$(a+b)^n = \binom{n}{0} a^n + \binom{n}{1} a^{n-1} b + \binom{n}{2} a^{n-2} b^2 + \cdots + \binom{n}{n-1} a b^{n-1} + \binom{n}{n} b^n$$

where $\binom{n}{k}$ represents the binomial coefficient, defined as:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

for $0 \leq k \leq n$.

Theorem 0.6: Binomial Expansion of 2^n

For any non-negative integer n, the following identity holds:

$$2^n = \sum_{i=0}^{n} \binom{n}{i} = (1+1)^n.$$

Definition 0.4: Well-Ordering Principle

Every non-empty set of positive integers has a least element.

Definition 0.5: “Without Loss of Generality”

A phrase that indicates that the proceeding logic also applies to the other cases. i.e., For a proposition not to lose the assumption that it works other ways as well.

Theorem 0.7: Pigeon Hole Principle

Let $n, m \in \mathbb{Z}^+$ with $n < m$. Then if we distribute m pigeons into n pigeonholes, there must be at least one pigeonhole with more than one pigeon.

Theorem 0.8: Growth Rate Comparisons

Let n be a positive integer. The following inequalities show the growth rate of some common functions in increasing order:

$$1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n!$$

These inequalities indicate that as n grows larger, each function on the right-hand side grows faster than the ones to its left.

— 1 —

Greedy Algorithms

1.1 Shortest Path

Theorem 1.1: Dijkstra's Algorithm

Proposition: Suppose that there is a shortest path from nodes $u \to v$. Then any sub-path between these nodes, say $x \to y$, is also the shortest path.

Algorithm: Given a weighted graph G and a source node s,

(i.) Keep track of best distances, start at a queue with s.

(ii.) Pop off the queue, flag item as visited.

(iii.) View all children weights, update if it's the new shortest path to that node.

(iv.) Queue children in ascending order of weight (smallest→largest)

Preform steps (ii) to (iv) until the queue is empty, having visited all possible nodes.

Proof 1.1: Proof of Correctness for Dijkstra's Algorithm

Invariant: For each node $u \in S$, $d(u)$ is length of shortest path $s \rightsquigarrow u$. By induction on $|S|$:
Base case: $|S| = 1$ is true since $S = \{s\}$ and $d(s) = 0$.
Inductive hypothesis: Assume true for $|S| = k \geq 1$.

- Let v be the next node added to S, and let (u, v) be the final edge.
- A shortest $s \rightsquigarrow u$ path plus (u, v) is an $s \rightsquigarrow v$ path of length $\pi(v)$.
- Consider any $s \rightsquigarrow v$ path P. We show that it is no shorter than $\pi(v)$.
- Let (x, y) be the first edge in P that leaves S, and let P' be the subpath to x.
- P is already too long as soon as it reaches y.

$$\ell(P) \geq \ell(P') + \ell(x, y) \geq d(x) + \ell(x, y) \geq \pi(y) \geq \pi(v)$$

■

To visualize our proof consider paths the following diagram:

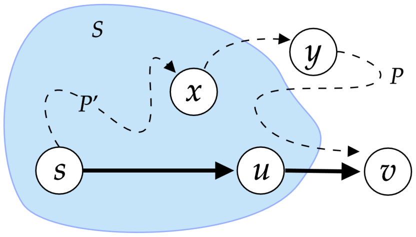


Figure 1.1: A system of subset paths (blue), and an exact point of exit $u \to v$ and $x \to y$

Since $u \to y$ and $x \to y$ are at the same exact point of exit, i.e., say $s \to v \cong s \to y$, then to go from $y \to v$ must take some additional step. Therefore, $s \to y \to v$ is longer. **E.g:** Let $s \to y := 3$ and $s \to v := 3$, and all steps take 1. Then $s \to y \to v = 4$, while $s \to v = 3$. Therefore $s \to v$ is the shortest path.

Dijkstra Example (i): To emphasize the BFS nature of Dijkstra's algorithm:

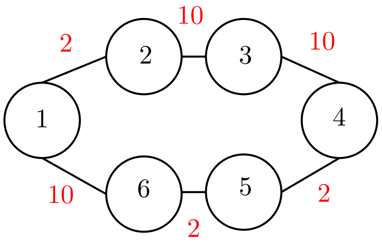


Figure 1.2: A weighted graph.

Where iterations and the queue look like:

Iteration Init	from 1	from 2	from 6	from 3	from 5
$1:0$	$1:0$	$1:0$	$1:0$	$1:0$	$1:0$
$2:\infty$	$2:2$	$2:2$	$2:2$	$2:2$	$2:2$
$3:\infty$	$3:\infty$	$3:12$	$3:12$	$3:12$	$3:12$
$4:\infty$	$4:\infty$	$4:\infty$	$4:\infty$	$4:22$	$4:14$
$5:\infty$	$5:\infty$	$5:\infty$	$5:12$	$5:12$	$5:12$
$6:\infty$	$6:10$	$6:10$	$6:10$	$6:10$	$6:10$

Queue Init	from 1	from 2	from 6	from 3	from 5
$[1]$	$[2,6]$	$[6,3]$	$[3,5]$	$[5,4]$	$[4]$

Finally visiting 4 to see 3 and 5 have already been visited, ending the algorithm as the queue's empty.

Dijkstra Example (ii):

Q	A	B	C	D	E
$\pi(v)$	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

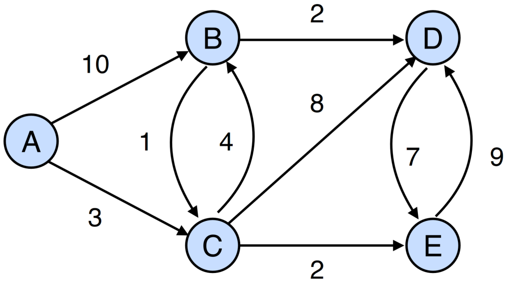


Figure 1.3: A weighted graph and its shortest paths.

In figure (1.3), the first row is 0 as $s \to s$, other nodes are assumed ∞,i.e., undefined. Each subsequent row finds the next shortest path, while updating the table about information it gathers. However we have one problem with Dijkstra's algorithm, it does not work with negative weights.

Theorem 1.2: Dijkstra's Algorithm and Negative Weights

Dijkstra's algorithm does not work with negative weights. This is because it assumes that the shortest path is the sum of the shortest paths. Therefore, if the algorithm believes it has found the shortest path, it assumes any further traversals will only increase the path length.

To illustrate the deterioration of Dijkstra's algorithm:

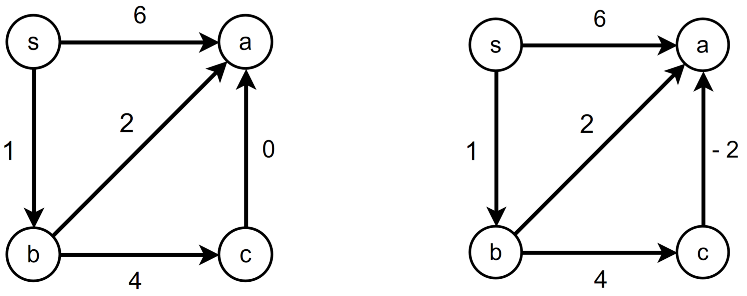


Figure 1.4: Shows two weighted graphs, one positive, and the other negative.

In Figure (1.4), our negative graph will never figure out the shortest path ($s \to b \to c \to a$=1). It will always assume ($s \to b \to a = 3$) is the shortest path. As when it looks at $c = 4$, it will think that it's impossible to yield any shorter of a path 3 as anything beyond 4 must be larger.

Function 1.1: Dijkstra Algorithm - Dijkstra(G, s)

Finds the shortest path in a weighted directed graph.

Input: A graph $G = (V, E)$ with adjacency list $G[u][v] = l(u, v)$ and source node s.
Output: Shortest distances $d[u]$ and parent nodes for paths.

```
Function Dijkstra($G, s$):
    $\pi \leftarrow \{\}$// hash table, current best list for $v$
    $d \leftarrow \{\}$// hash table, distance of $v$
    $parents \leftarrow \{\}$// parents in shortest path tree
    $Q \leftarrow \mathrm{PQ}()$// priority queue to track minimum $\pi$
    $\pi[s] \leftarrow 0$;
    $Q$.INSERT($\langle 0, s \rangle$);
    for $v \neq s$ in $G$ do
        $\pi[v] \leftarrow \infty$;
        $Q$.INSERT($\langle \pi[v], v \rangle$);
    while $Q$ is not empty do
        $\langle \pi[u], u \rangle \leftarrow$ EXTRACT-MIN($Q$);
        $d[u] \leftarrow \pi[u]$;
        for $v \in G[u]$ do
            if $\pi[v] > d[u] + l(u, v)$ then
                DECREASE-KEY($\langle \pi[v], v \rangle, \langle d[u] + l(u, v), v \rangle$);
                $\pi[v] \leftarrow d[u] + l(u, v)$;
                $parents[v] \leftarrow u$;
```

Time Complexity: $O(m \log n)$ where m is the number of edges and n is the number of nodes, assuming G is connected $(n - 1 \leq m)$; Otherwise, $O((n + m) \log n)$.
Space Complexity: $O(n + m)$ storing the hash-table of the graph and priority queue.

Tip: Video Demo of Dijkstra's Algorithm:
https://youtu.be/gaXTSp2BBdY?si=Egc7bAv4SRPgtffa

1.2 Spanning trees

Definition 2.1: Spanning Tree

A **spanning tree** of a graph G is a subgraph containing edges to each $n \in G$ without cycles.

Definition 2.2: Minimum Spanning Tree (MST)

A **minimum spanning tree** of a graph G is a spanning tree with the smallest sum of edge weights.

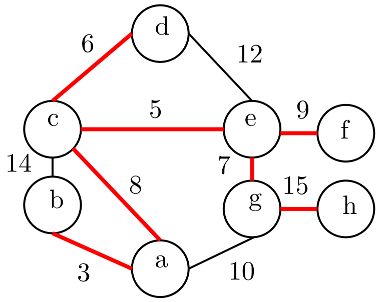


Figure 1.5: Example of an graph with MST highlighted in red.

This tree visits each node once taking the shortest path which connects all of them.
Possible Algorithms:

- **Prim's:** Start with some root node s. Grow a tree T from s outward. At each step, add to T the cheapest edge e with exactly one endpoint in T.
- **Kruskal's:** Start with $T = \emptyset$. Consider edges in ascending order of weights. Insert edge e in T unless doing so would create a cycle.
- **Reverse-Delete:** Start with $T = E$. Consider edges in descending order of weights. Delete edge e from T unless doing so would disconnect T.
- **Borůvka's:** Start with $T = \emptyset$. At each round, add the cheapest edge leaving each connected component of T. Terminates after at most $\log(n)$ rounds.

Next we revisit cycles and introduce **cuts**, which will have important implications when approaching this problem.

Definition 2.3: Endpoint

An **endpoint** is either end of an edge. So for edge $e = u \leftrightarrow v$, u and v are endpoints. If $u \rightarrow v$, then v is an endpoint of u.

Definition 2.4: Cut

Given a graph G, partitioning of the nodes into a set is called a **cut**, say G'. Nodes, with exactly one endpoint in G', are the **cut-set**.

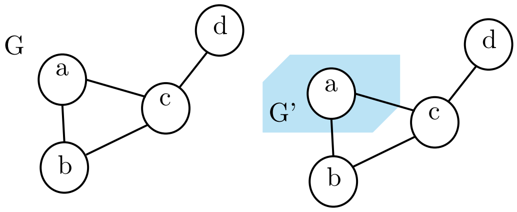


Figure 1.6: Illustration of a graph G and a cut G' of the graph.

We see in Figure (1.6) that $G' = \{a\}$ and our cut-set contains edge-pairs (a, b) and (a, c). Where the edge (d, c) is not included as the cut G' does not intersect it.

Theorem 2.1: Cycles & Cut-sets

If a cut-set crosses a cycle, then the cut-set intersects an even number of edges in the cycle. As what comes in, must come out.

Given Figure (1.6), the cut-set G' intersects the cycle (a, b, c), yielding an even cut-set.

Theorem 2.2: Cycle Property

In a graph with a cycle, the edge with the largest weight in that cycle is not in the MST. As taking an edge from a cycle does not disconnect the graph, the largest edge is not necessary.

Theorem 2.3: Cut Property

Given a graph G and a cut-set C, where e is the lightest-edge in C; e must be in the MST. As if $e \notin G$ and G is an MST, adding e creates a cycle. By the cycle property, e must replace the heaviest edge in that cycle.

Example: Given the below Figure (1.7), point e must be in the MST to connect all nodes (cut property). Say dash-edge f is the largest in its cycle, then f is not in the MST (cycle property).

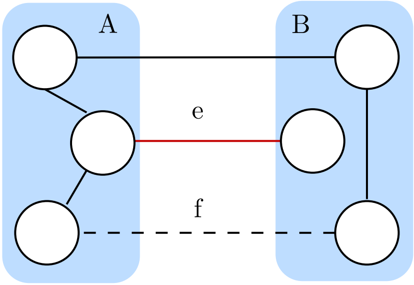


Figure 1.7: A graph cut into two disjoint sets, with a highlighted edge e and a dashed edge f.

Theorem 2.4: Prim's Algorithm

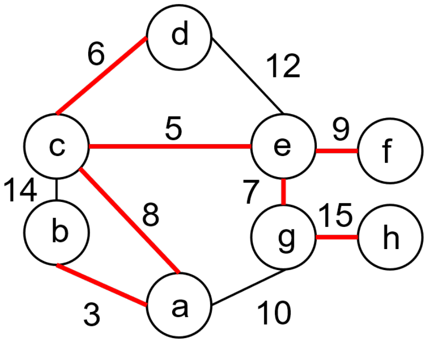
Given a connected graph G with n nodes and m edges, we produce the MST via:

(i.) Initialize an MST table T, and a priority queue Q with each n of weight ∞.

(ii.) Start a round with an arbitrary node s, evaluating children nodes v.

(iii.) Update each $T[v] = s$, if $w(s, v)$ is lighter than $T[v]$.

(iv.) End this round, take the top node in Q as the new s, repeat (ii.)-(iv.) until all $n \in T$.



NODE	PARENT
a :	NULL
b :	a
c :	a
d :	c
e :	c
g :	e
f :	e
h :	g

Figure 1.8: Prim's Alg. in the order $a \to b \to c \to e \to d \to g \to f \to h$, and a parent table.

The above example shows how our parent table will result with the following table. Next we will discuss in detail how one could approach implementation.

Tip: Live Demo of Prim's Algorithm https://www.youtube.com/watch?v=cplfcGZmX7I.

Function 2.1: Prim's Algorithm - PALG()

Input: A connected, undirected graph G of V nodes. With weights $w(u, v)$, and $u, v \in V$.
Output: Minimum Spanning Tree (MST) formed by edges $T < (v, \texttt{parent}[v]) >$

```
1  $Q < weight, node >$; // Min-heap of <key, data>
2  $V.forEach((v) => \{Q[v] \leftarrow \infty\})$; // for each $v \in V$ set it's weight to $\infty$
3  $Q[V_0] \leftarrow 0$; // Picking arbitrary node $V_0$, pushing it to the top of $Q$
4  $T$; // Hashtable where $T[u]$ is the parent $v$ of $u$
5  while $Q \neq \emptyset$ do
6      $u \leftarrow Q.Extract()$;
7      foreach $v \in G[u]$ do
8          if $(v \in Q)$ and $(w(u, v) < Q[v])$ then
               // Edit the node's weight in $Q$ then re-balance.
9              $Q[v] \leftarrow w(u, v)$;
10             $Q.DecreaseKey(v)$;
11             $T[v] \leftarrow u$;
12 return $T$
```

Correctness: We run a form of BFS on the graph, which touches every node. BFS creates levels each iteration, resulting in a cut-set with an end-point $G[u]$. By the cut property, any new lightest edge $w(u, v)$ is added or replaces a heavier edge in T. Thus forming an MST as all nodes are considered.
Time Complexity: $O((n + m) \log n)$. Line 7 at worse checks every adjacency, $O(n + m)$, for m edges of n nodes. Say lines 8-9 takes $O(1)$ time to find $v \in Q$ via hash-table. Line 10 takes $O(\log n)$ time to re-balance the heap. Thus, $O((n + m) \log n)$ or $O(m \log n)$.
Space Complexity: $O(n+m)$, as we at most store the data items a hash-table representation of our graph.

Note: Lines 8 to 10 may require additional implementation. Since basic min-heaps only store weights, one might need a direct reference to each member in the heap. Say a reference hash-table R, where $R[v]$ points to v node in Q. Once we update $R[v]$, we tell the Q to sort the new v weight. We say "*DecreaseKey*" as our new weight should be lighter, bubbling up the heap.

To check if a node has been visited before, doesn't matter in our case, as we update our solution T with a better solution once it is found. We additionally discard the lightest node each round to avoid infinite loops. If one wanted to, they could use T's entries as a visited list. If entries are undefined upon access, then they have not been visited.

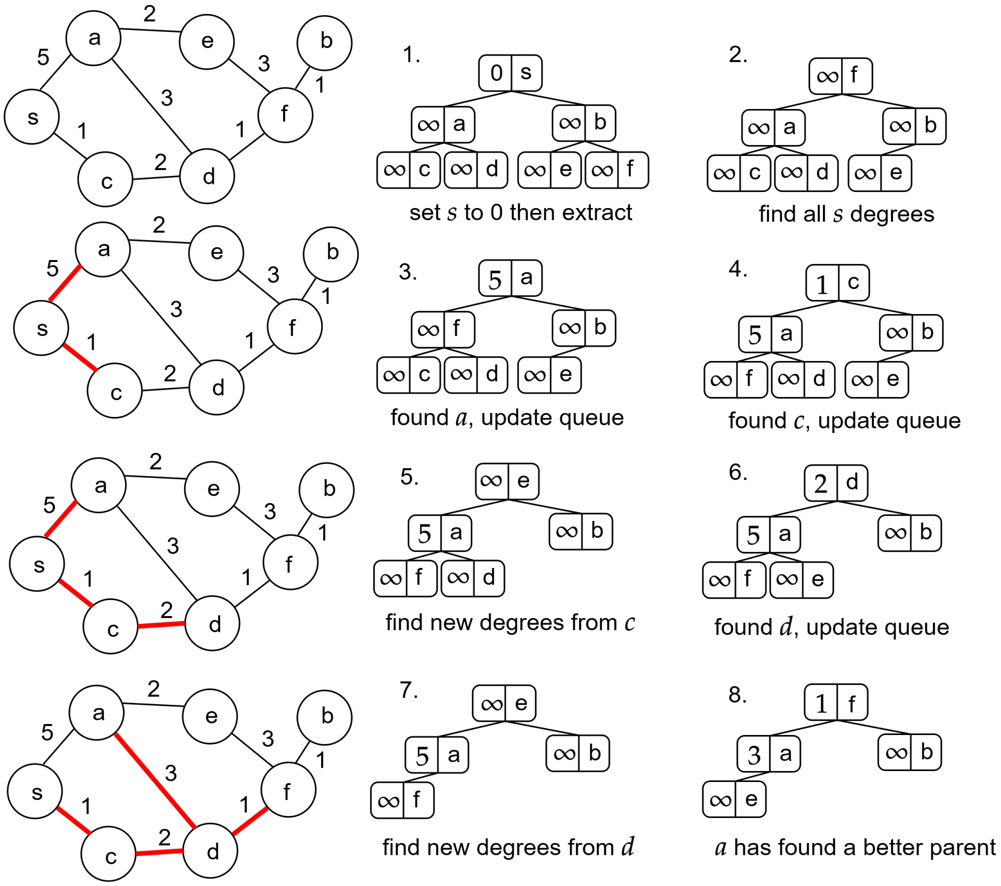


Figure 1.9: A diagram illustrating the pattern of Prim's Algorithm through each iteration.

The above diagram shows Prim's algorithm each round, pick the lightest edge at the top of the heap, check its degrees which have not been, update weights, re-balance the heap, and repeat. The algorithm will terminate when all nodes have been picked from the heap.

Proof 2.1: Prim's Hash-tables vs. Min-heaps Implementation

Say we had a hash-table of priorities where $R[v]$ returns weight $w(u,v)$:

- **Hash-table - Extract:** $O(n)$, search all indices, **Decrease-Key:** $O(1)$, update $R[v]$.
- **Min-heap - Extract:** $O(1)$, top element, **Decrease-Key:** $O(\log n)$, re-balance heap.

```
1  $Q < weight, node >$; // Min-heap of <key, data>
2  $V.forEach((v) => \{Q[v] \leftarrow \infty\})$; // for each $v \in V$ set it's weight to $\infty$
3  $Q[V_0] \leftarrow 0$; // Picking arbitrary node $V_0$, pushing it to the top of $Q$
4  $T$; // Hashtable where $T[u]$ is the parent $v$ of $u$
5  while $Q \neq \emptyset$ do
6      $u \leftarrow Q.Extract()$;                 $O(n + m)$
7      foreach $v \in G[u]$ do
8          if $(v \in Q)$ and $(w(u,v) < Q[v])$ then
               // Edit the node's weight in $Q$ then re-balance.
9              $Q[v] \leftarrow w(u,v)$;
10             $Q.DecreaseKey(v)$;
11             $T[v] \leftarrow u$;
12 return $T$
```

line 5 stores all n nodes to be visited $O(n)$, line 7 iterates all neighbors for each n. Let all edges be m, then $\sum_{v \in n} \text{degree}(v) = m$. Since line 6 is $O(1)$, we have $O(n+m)$ for iterating all edges. Now for every edge, we might evaluate Decrease-Key, which is $O(\log n)$. This gives us $O((n+m)\log n)$, or $O(m \log n)$.

When using a hash-table, line 6 becomes $O(n)$, and Decrease-Key is $O(1)$. Thus, $O((n(n)+m)\cdot 1) = O(n^2+m)$ or $O(n^2)$. Hence,

Hash-table: $O(n^2)$. **Min-heap:** $O(m \log n)$.

For any connected graph $n - 1 \leq m \leq \frac{n(n-1)}{2}$, where m is the number of edges. Most often m is much less than n, making the min-heap the better choice. E.g., say $m = \frac{n^2}{2}$, then $O(n^2)$ vs. $O(n^2 \log n)$, hash-table wins. However, say $m = \frac{n}{2}$, then $O(n^2)$ vs. $O(n \log n)$, min-heap wins. ■

Theorem 2.5: Prim's Hash-tables vs. Min-heaps

For Prim's algorithm,
when $m \leq n$, min-heap is better $(O(n \log n) < O(n^2))$. When m is significantly larger than n, hash-tables are better $(O(n^2) < O(n^2 \log n))$.

1.2.1 Union-Find Data Structures

Definition 2.5: Union-Find Data Structure

A **Union-Find** data structure is a data structure that keeps track of a set of elements partitioned into multiple disjoint subsets. It supports two useful operations: **Union:** Merge two subsets into a single subset. **Find:** Determine which subset a particular element is in.

We *could* simply use a hash-table to keep track of the parent of each node, which gives us **Find O(1); However, Union is O(n)**, as we would have to update every node to its new parent. Given a large set of n nodes, with m edges in a hash-table, to find and union all n nodes results in $\mathbf{O(n^2+m)}$, as for every n we find, we make n updates, where our finds accumulate to the number of total edge connections.

Scenario - *Follow The Leader:* Say you have n people playing rock-paper-scissors. If n_i beats n_j, then n_j follows n_i. This creates large sets of people following a leader. When leader n_k beats n_i, n_i follows n_k with n_i's followers tagging along.

Say we are trying to figure out which component n_x is in. We ask n_x, "who is your leader," they say n_i, then n_i says n_k, and n_k replies, "I am the leader." Therefore n_x is in group n_k.

Definition 2.6: Forest

A **forest** is a collection of disjoint trees, where each tree has a **representative** r node. We may union-join trees A and B by making A be B's representative. Where $b \in B$ still point to B and B points to A.

Trees S with smaller heights should be added to bigger trees B. As height indicates the number of nodes who report to a leader. By adding the bigger tree to the smaller, we increase the time complexity of finding leaf nodes.

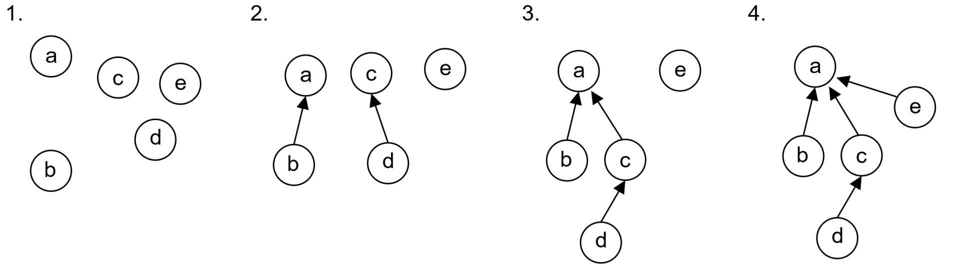


Figure 1.10: Showing disjoint nodes Union-Join with each other.

In the above figure nodes we see in step two we have two disjoint trees, with leaders a and c. In step three we join a and c by making c point to a. Finally e points to a to join the group.

We see a lot of redundancy, with nodes n_x reporting to leaders n_i, until a final leader n_k is reached. We may improve this overtime by setting n_x's leader to n_k directly.

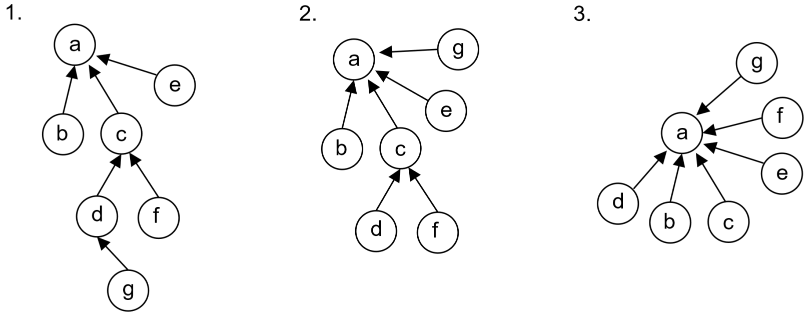


Figure 1.11: Showing a forest compress over multiple finds.

Given the figure above, we first ask g who their leader is, they report to c who reports to a. We now set g's leader to a. We do the same for f and d. Now once we ask g, f, or d who their leader is, they report to a directly without the need to traverse the tree. This is called **Path Compression**.

Definition 2.7: Path Compression

Path Compression is a technique used in Union-Find data structures to flatten the structure of the tree. After finding the leader of a node in a whole component, we set the node's parent directly to that leader. This reduces the time complexity of find operations for subsequent queries.

We compare all of our techniques in the following table:

Operation\Implementation	Simple Hash-table	Forest	Forest with Path Compression
Find (worst-case)	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$
Union of sets A, B (worst-case)	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Total for n unions and n finds, starting from singletons	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta\big(m \cdot \alpha(m,n)\big)$

Table 1.1: Time-complexity comparison of different implementations. Where $\alpha(m, n)$ is the inverse Ackermann function, which grows very slowly.

Theorem 2.6: Kruskal's Algorithm

Given a connected graph G of V nodes and E edges, we produce the MST via:

(i.) Sort all $e \in E$ by weight in ascending order into an array W.

(ii.) Initialize a forest T with all V nodes as singletons.

(iii.) For each $e \in W$ Union-find its endpoints u and v.

- If u and v are in different sets, Union-join u and v.

Return the resulting forest T as the MST.

Function 2.2: Kruskal's Algorithm - KALG()

Input: a connected graph G of V nodes and E edges.
Output: Minimum Spanning Tree (MST) formed by forest Union-find data structure.

```
1 $W[\,] \leftarrow Sort(E)$; // Sort all edges by weight
2 $T \leftarrow new\ UnionFind(G)$; // new forest with all $G$'s nodes as singletons
3 for $i = 1$ to $W.size()$ do
4     $(u, v) \leftarrow e$; // Get the endpoints of $e$
5     if $T.Find(u) \neq T.Find(v)$ then
6         $T.Union(u, v)$; // Union-join $u$ and $v$
7 return $T$
```

Correctness: Sorting edges in ascending order, ensures lightest possible edge is picked first before redundancy checks with Union-find, which avoids cycles (Line 5). Any new unique edge e is added to the MST. This yields a connected graph as all edges are considered no matter their weight.
Time Complexity: $O(E \log E)$ or $O(E \log V)$. Line 1 sorts all edges, which takes $O(E \log E)$ time, where E is at most V^2 (all nodes connect to each other). Thus, $\Theta(E \log_2 V)$ is $O(E \log_2 E)$ as the exponent to reach V and E are the same. Then iterating through all E edges; actions are one-to-one for each *find-merge* operation E_i. This results in one operation per edge, rather than a compounded combination of operations, like a nested for-loops.
Space Complexity: $O(V+E)$, as we at most store the data items a hash-table representation of our graph.

Note: Lines 1 and 4, might require additional implementation such as a reference hash-table R to keep track of edges and their end-points after sorting.

Exercise 2.1: Let there be a graph $G(V, E)$ with an MST T and a shortest paths tree S. Now suppose a weight of 1 is added to each edge in G.

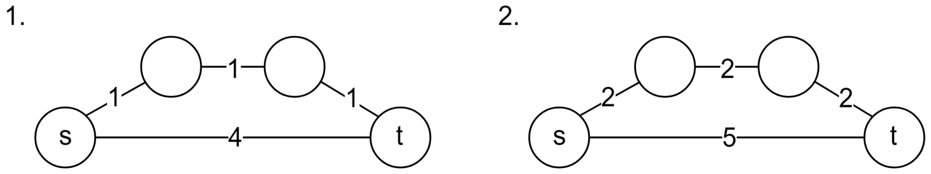
1. Will the MST T change?
2. Will the shortest paths tree S change?

Exercise 2.2: Let there be a graph $G(V, E)$ with an MST T and a shortest paths tree S. Suppose a new edge e is added to G.

1. The weight of e is the heaviest in G.
   a) Will the MST T change?
   b) Will the shortest paths tree S change?
2. The weight of e is the lightest in G.
   a) Will the MST T change?
   b) Will the shortest paths tree S change?

Answer 2.1:

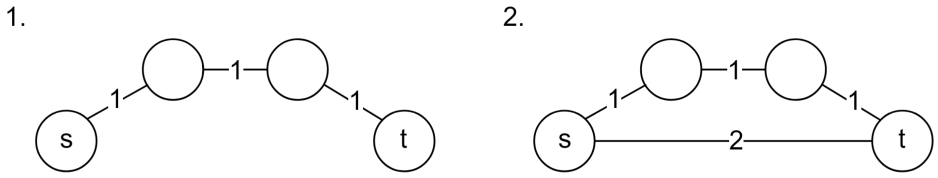
1. **No.** All edges in T are still the lightest edges in G.
2. **Yes.** Shortest paths is not MST. Here is a counter example:



The shortest path from s to t changed when all edges were increased by 1.

Answer 2.2:

1. a) **No.** By the cycle property, the heaviest edge is not necessary in the MST.
   b) **Yes.** The new edge may present a shorter path. Here is a counter example:



2. a) **Yes.** By the cycle property, an existing edge will be replaced by a lighter edge.
   b) **Yes.** Consider a long expensive path, a new edge may present a shorter path.

— 2 —

Dynamic Programming

2.1 Formulating Recursive Cases

Definition 1.1: Dynamic Programming

In recursive algorithms, there are many cases where we repeat the same computations multiple times. To reduce this redundancy, we store the results of these computations in a table. This is known as **memoization**. This paradigm of programming is called **dynamic programming**.

Scenario - *Fibonacci Sequence*: The Fibonacci sequence is defined as $F_n = F_{n-1} + F_{n-2}$, with $F_0 = 0$ and $F_1 = 1$. Our first 8 terms are $\{0, 1, 1, 2, 3, 5, 8, 13\}$. To compute F_5, we do $0 + 1 = 1$, $1 + 1 = 2$, $1 + 2 = 3$, and $2 + 3 = 5$.

A recursive approach would be:

Function 1.1: Slow Fibonacci Sequence - *Fib()*

Input: n the index of the Fibonacci sequence we wish to compute.
Output: F_n the n_{th} Fibonacci number.

```
1 Function Fib(n):
2     if $n \leq 1$ then
3         return $n$;
4     else
5         return $Fib(n-1) + Fib(n-2)$;
```

Time Complexity: $O(2^n)$. Since line 6 depends on both calls, we reflect such in our recurrence relation, $T(n) = T(n-1) + T(n-2) + O(1)$, Theorem (**??**). Since we make calls of size $n - 1$ and $n - 2$, which are both $O(n)$, we have an exponential time complexity $O(2^n)$.

When we unravel the recursion tree, we see plenty of redundancies:

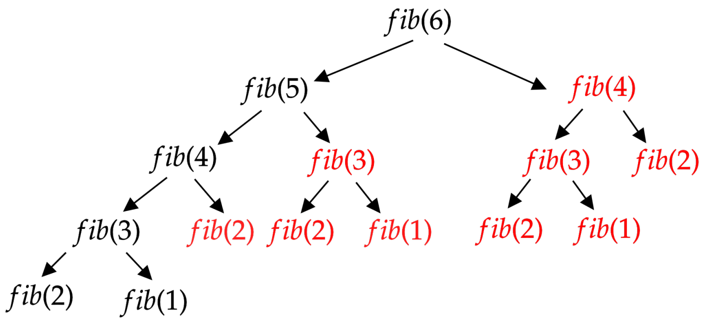


Figure 2.1: Recursion Tree for Fibonacci Sequence

We see that we've already computed F_2 and F_3 multiple times. We can store these values in a table, and use them when needed:

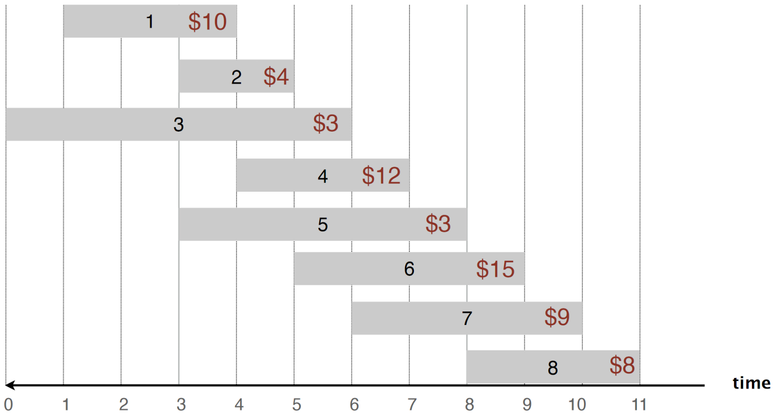
Function 1.2: Memo Fibonacci Sequence - *Fib()*

Input: n the index of the Fibonacci sequence we wish to compute.
Output: F_n the n_{th} Fibonacci number.

```
$F[\,]$; // Table to store Fibonacci numbers
Function $Fib(n)$:
    if $n \leq 1$ then
        return $n$;
    else
        if $F[n]$ is not defined then
            $F[n] = Fib(n-1) + Fib(n-2)$;
        return $F[n]$;
```

Time Complexity: $O(n)$. Since we only need to compute F_n once, we at most recurse $n-1$ times. As we unravel we have all our necessary values stored in our table.
Space Complexity: $O(n)$. We store n and recurse at most $n-1$ times.

Scenario - ***Weighted Interval Scheduling***: Say we have n paying jobs which overlap each other. We want to find the best set of jobs that allows us to maximize our profit. Recall Section (**??**)



Let us define **OPT(j)** as the maximum profit from jobs $\{1 \ldots j\}$, and $\mathbf{v_j}$ as j_{th}'s value. Then $OTP(8)$, considers jobs $1 \ldots 8$. Let $p(j) :=$ The largest index $i < j$, s.t., job i is compatible with j. (if none, then $p(j) = 0$). We have two cases:

$$OPT(8) = \begin{cases} OPT(7) & \text{if job 8 is not selected} \\ v_8 + OPT(p(8)) & \text{if job 8 is selected} \end{cases}$$

Then $p(8) = 5$, $p(5) = 0$, yielding \$11, which isn't the optimal solution, see job 6. If we don't choose j, then the optimal solution resides in $\{1 \ldots j-1\}$. So we want to know, if our current $OPT()$ solution larger than the next solution. We derive the following cases, and algorithm:

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ max\{\underbrace{v_j + OPT(p(j))}_{\text{build solution}}, \underbrace{OPT(j-1)}_{\text{next solution}}\} & \text{else} \end{cases}$$

Function 1.3: Weighted Interval Scheduling - *RecOPT()*

Compute all $OPT(j)$ recursivley, unraveling seeing which $OPT(j)$ is larger; $\mathbf{O(2^n)}$ **Time**.

```
Function RecOPT(j):
    if $j = 0$ then
        return 0;
    else
        $OPT(j) \leftarrow max\{v_j + RecOPT(p(j)), RecOPT(j-1)\}$;
        return $OPT(j)$;
```

Now we employ memoization to store our results in a table, and use them when needed:

Function 1.4: Memo Weighted Interval Scheduling - *OPT()*

```
1  Sort jobs by finish time; // $O(n \log n)$
2  Compute all $p(1), \dots, p(n)$; // $O(n)$
3  $OPT[\,]$; // Table to store $OPT(j)$
4  Function $OPT(j)$:
5      if $j = 0$ then
6          return 0;
7      else
8          if $OPT[j]$ is not defined then
9              $OPT[j] \leftarrow max\{v_j + OPT(p(j)), OPT(j-1)\}$; // $O(n)$
10         return $OPT[j]$;
```

Time Complexity: $O(n \log n)$, as we are bottle-necked by our sorting algorithm. Line 10 is $O(n)$, following the same memoization pattern as the Fibonacci sequence.

2.2 Bottom-Up Dynamic Programming

In the fibonacci, sequence we don't have to compute it recursively. As shown before we can compute it linearly. Computing F_5, we do $0 + 1 = 1$, $1 + 1 = 2$, $1 + 2 = 3$, and $2 + 3 = 5$.

Function 2.1: Bottom-Up Fibonacci Sequence - *Fib()*

```
$F[0] \leftarrow 0$; $F[1] \leftarrow 1$; // Base cases (array of size $n + 1$)
for $i \leftarrow 2$ to $n$ do
    $F[i] \leftarrow F[i-1] + F[i-2]$;
return $F[n]$;
```

Time Complexity: $O(n)$. We compute F_n linearly, only needing to compute F_i once.

To offer intuition, recall figure (2.1), we see that that we only really take one branch of the tree. All other branches are redundant. I.e., it's almost as if we have a linear path from the root to the leaf. Hence, there's no need for recursion.

Likewise, we can compute the weighted interval scheduling problem linearly:

Function 2.2: Bottom-Up Weighted Interval Scheduling - *OPT()*

```
Sort jobs by finish time; // $O(n \log n)$
Compute all $p(1), \dots, p(n)$; // $O(n)$
$OPT[0] \leftarrow 0$; // Base case (array of size $n+1$)
for $j \leftarrow 1$ to $n$ do
    $OPT[j] \leftarrow max\{v_j + OPT[p(j)], OPT[j-1]\}$;
return $OPT[n]$;
```

Time Complexity: $O(n \log n)$. We sort our jobs, and compute $p(1), \dots, p(n)$ in $O(n)$ time. We then compute $OPT(j)$ linearly, only needing to compute $OPT(j)$ once.

j	0	1	2	3	4	5	6	7	8
OPT(j)	\$0	\$4	\$4	\$10	\$10	\$12	\$19	\$19	\$20

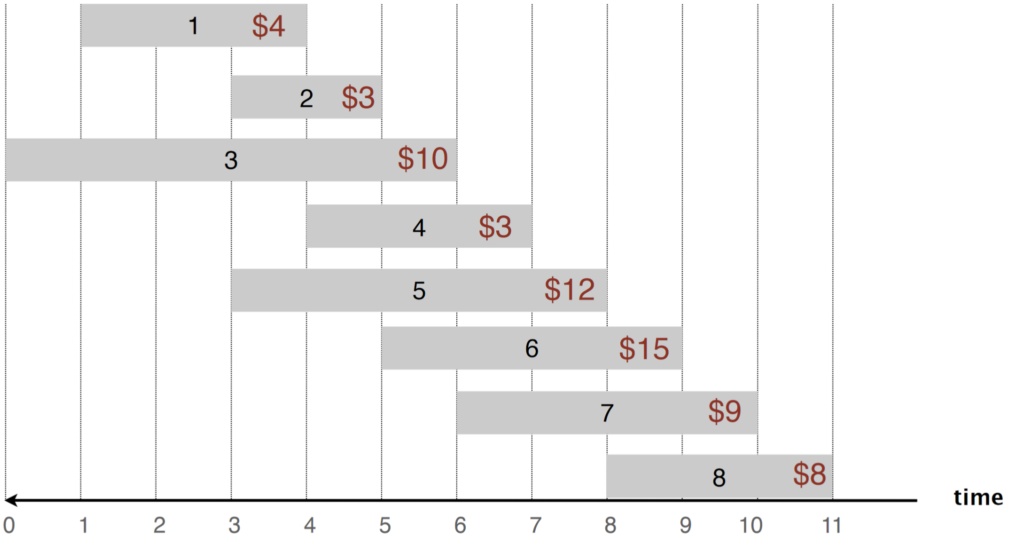


Figure 2.2: Optimal Values for Each Index j

Where,
$OTP[1] \leftarrow max\{v_1 + OPT(p(1)), OPT(0)\} = max\{4+0, 0\} = 4;$
$OPT[2] \leftarrow max\{v_2 + OPT(p(2)), OPT(1)\} = max\{4+0, 4\} = 4;$
$OPT[3] \leftarrow max\{v_3 + OPT(p(3)), OPT(2)\} = max\{10+0, 4\} = 10;$
$OPT[4] \leftarrow max\{v_4 + OPT(p(4)), OPT(3)\} = max\{3+4, 10\} = 10;$
$OPT[5] \leftarrow max\{v_5 + OPT(p(5)), OPT(4)\} = max\{12+0, 10\} = 12;$
and so on.

2.3 Backtracking

Definition 3.1: Backtracking

During recursion we may build solutions based on some number of constraints. During recursion, if we at all, hit some constraint or *dead-end*, we **backtrack** to a previous state to try another path.

Additionally, after a solution is found, we may want to trace which calls lead to our solution. This also called **backtracking**.

Given our weighted interval scheduling (WIS) problem in Figure (2.2), we want to reverse-engineer the jobs that gave us our final solution. Since we have already computed all $OPT(j)$ stored in $OPT[\ ]$, and all $p(j)$, we can backtrack to find the jobs that gave us our optimal solution.

Function 3.1: Backtracking Weighted Interval Scheduling - *Backtrack()*

```
  // $OPT[\ ]$ (optimal solutions of $j$ job) and $p()$ (next comptible job) are
     already computed for $1, \dots, j$
1 $j \leftarrow OPT.length - 1$; $S \leftarrow \{\}$; // $S$ is our set of jobs
2 $Backtrack(OPT, j)$;
3 Function $Backtrack(OPT,\ j)$:
4     if $v_j + OPT[p(j)] > OPT[j-1]$ then
5         return $S \cup Backtrack(OPT, p(j))$;
6     else
7         return $Backtrack(OTP, j-1)$;
```

Correctness: In our example above, the WIS's last index was the optimal solution. However, let $8 = \$1$, then jobs $(6, 1)$ would have been the optimal solution. This leaves $OPT[8] = \$19$, rather than \$20. Line 4 finds the first occurance where we found the optimal solution. As if we first found the optimal solution at index 6, then $6, \dots, j$ would contain $OPT(6)$. This is why we exclude the choice $(7, 3) = \$19$.

We then repeat such pattern on the next compatible job. We know the set $N := \{1 \dots p(j)\}$ must contain an element of the optimal solution. Similar to our Dijkstra's proof (1.1), that within the optimal path, a subpath's shortest path is also optimal. We check if $p(j)$ is the first occurance of the optimal solution in N, if not we continue to backtrack.

Time Complexity: $O(n)$. At most, iterate through all n jobs, and add them to our set S.

Notice that our backtracking closely mimics our orginal recursive formula.

2.4 Subset Sum

2.4.1 Weighted Ceiling

Definition 4.1: Problem - Subset Sum (Weighted Ceiling)

Given a set of integers, say $S = \{3, 6, 1, 7, 2\}$, and a target sum $T = 9$, find the max subset P of S, such that $P \leq T$.

We know that when building our solution, we may pick S_i and try all other combinations with $S_{i+1}, \ldots, S_n$ where $n = |S|$. This may cause us to repeat computations as we build our solution. We now know to use dynamic programming to store our results. We start by finding subproblems:

$$S = \{3, 6, 1, 7, 2\}, T = 9; \text{ (choose 3)} \Rightarrow S' = \{6, 1, 7, 2\}, T' = 6$$

Where S' and T' are S and T after removing 3. If we kept $T = 9$, then we'd be asking, "find a max subset P of S', s.t., $P \leq 9$," which isn't our goal. If we decide $3 \in P$, then S' may also contain an optimal contribution (similar to our above Proof (3.1)). While building our solution, if $S_i > T$ we know not to consider it.

We derive the following cases, where T is a changing target sum, and S_i the value of index i (Alternitevely we could use subtraction, instead popping tail elements rather than head elements):

$$OPT(i, T) = \begin{cases} 0 & \text{if } T = 0 \\ OPT(i+1, T) & \text{if } S_i > T \\ max\{\underbrace{OPT(i+1, T)}_{\text{next solution}}, \underbrace{S_i + OPT(i+1, T - S_i)}_{\text{build solution}}\} & \text{else} \end{cases}$$

Moreover, if S_i is compatible with T, we check other solutions. In WIS (8), we only kept track of one changing variable. However, in this case, we change 2 states during each recurrence; Hence our array is two-dimensional:

	Target Sum t									
Index i	0	1	2	3	4	5	6	7	8	9
{}	0	0	0	0	0	0	0	0	0	0
4 ({2})	0	0	2	2	2	2	2	2	2	2
3 ({7,2})	0	0	2	2	2	2	2	7	7	9
2 ({1,7,2})	0	1	2	3	3	3	3	7	8	9
1 ({6,1,7,2})	0	1	2	3	3	3	6	7	8	9
0 ({3,6,1,7,2})	0	1	2	3	4	5	6	7	8	9

Table 2.1: Subset Sum Dynamic Programming Table (DP Table), where $OTP[i][t]$ is the max combination P of $S_i, \ldots, S_n$ s.t., $P \leq t$.

An additional explanation of the above table on the next page.

The above table (2.1), when we reach $i = 4$, we only have $\{2\}$ to consider. This is fine as we've already considered all 2's possible combinations. Observe a nested for-loop approach to find all pairs in $S = \{3, 6, 1, 7, 2\}$:

- Start with 3, then: $\{(3, 6), (3, 1), (3, 7), (3, 2)\}$
- Then with 6, then: $\{(6, 1), (6, 7), (6, 2)\}$
- Then with 1, then: $\{(1, 7), (1, 2)\}$
- Then with 7, then: $\{(7, 2)\}$
- Then with 2, then: $\{2\}$

Notice how we already found all 2's combinations, $(3, 2), (6, 2), (1, 2), (7, 2)$. So even though we only have 2 to consider at $i = 4$, we've already accounted for all possible combinations.

Hence the algorithm below. We change the next and build step to subtraction to allow us to mimic recursion in a bottom up approach. Here we start at the top left progressing forward, rather than the bottom right:

Function 4.1: Subset Sum - *OPT()*

```
$S; T; OPT[\,][\,]$; // Set $S$, Weight ceiling $T$, DP table $OPT(i,T)$
$OPT[0][*] \leftarrow 0$; // Base case (array of size 0)
$OPT[*][0] \leftarrow 0$; // Base case (array of size $T = 0$)
for $i \leftarrow 1$ to $S.length$ do
    for $t \leftarrow 1$ to $T$ do
        if $S[i] > t$ then
            $OPT[i][t] \leftarrow OPT[i-1][t]$;
        else
            $OPT[i][t] \leftarrow max\{\underbrace{OPT[i-1][t]}_{\text{next solution}}, \underbrace{S[i] + OPT[i-1][t-S[i]]}_{\text{build solution}}\}$;
return $OPT$;
```

Time Complexity: $O(nT)$. We iterate through all n jobs, and for each job, we iterate through all T target sums.

The above table (2.1) shows our best possible combination at $OPT[0][9]$; However, we don't know which elements contributed to our solution. We backtrack to find such elements on the next page.

In our table below, ignore the fact that the numbers come out nicely. Where $OPT[0][9] = 9$, could have been $OPT[0][9] = 8$, if we excluded 2 and 3 from our orginal set.

We want to know at each stage, what S_i we picked to obtain T. Just like, in our WIS problem (3.1), we want to know the first occurance of the optimal solution existing. I.e., which S_i was first to contribute to our solution.

Below we give the algorithm to compute such, and an explanation below the table:

Function 4.2: Backtracking Subset Sum - *Backtrack()*

```
$i \leftarrow 0$; $t \leftarrow T$; $S \leftarrow \{\}$; // $S$ is our set of jobs
$Backtrack(OPT, i, t)$;
Function Backtrack(OPT, i, t):
    // Where $OTP.length$ is the number of rows
    while $i < OPT.length$ do
        if $OPT[i][t] > OPT[i+1][t]$ then
            $S \leftarrow S \cup \{S[i]\}$;
            $t \leftarrow t - S[i]$;
        $i \leftarrow i + 1$;
    return $S$;
```

Time Complexity: $O(n)$. At most, iterate through all n jobs, and add them to our set S.

	Target Sum t									
Index i	0	1	2	3	4	5	6	7	8	9
{}	0	0	0	0	0	0	0	0	0	0
4 ({2})	0	0	2	2	2	2	2	2	2	2
3 ({7,2})	0	0	2	2	2	2	2	7	7	9
2 ({1,7,2})	0	1	2	3	3	3	3	7	8	9
1 ({6,1,7,2})	0	1	2	3	3	3	6	7	8	9
0 ({3,6,1,7,2})	0	1	2	3	4	5	6	7	8	9

Here we know 3 combinations did not start our solution, neither did 6 or 1. However, 7 combinations did. We know that $7 \in P$. We reduce T to 2, and our first element to contribute to the optimal solution was 2. We reduce again hitting 0, hence, $P = \{7, 2\}$.

2.4.2 Knapsack

Definition 4.2: Problem - Subset Sum (Knapsack)

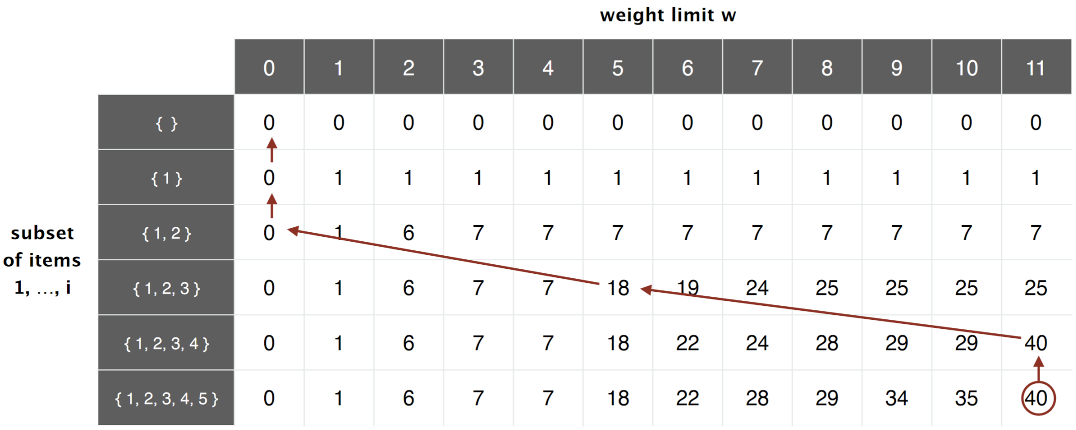
Given a set S of n items, each with a weight w_i and value v_i, and a knapsack of capacity W, find a subset P of S, s.t., $\sum_{i \in P} w_i \leq W$ and $\sum_{i \in P} v_i$ is the highest value achievable.

To the right is an example of a knapsack instance. Say we want to solve this by repeatedly taking some combination of weights and their totals. Each time we take an item, we can no longer choose it, and our knapsack weight and value increase. Our base case must be when we run out of items. We also know to step back when we exceed our weight limit.
We want our build step to track the value in some way, decrement our choices, and decrease our weight limit. Then if we don't pick the item, we move to the next item with no change in weight or value. This becomes strikingly similar to the previous subset sum problem with minor adjustments:

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

knapsack instance
(weight limit $W = 11$)

$$OPT(i, w) = \begin{cases} 0 & \text{if } i == 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ max\{\underbrace{OPT(i-1, w)}_{\text{next solution}}, \underbrace{v_i + OPT(i-1, w-w_i)}_{\text{build solution}}\} & \text{else} \end{cases}$$



subset of items 1, …, i \ weight limit w	0	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

OPT(i, w) = max-profit subset of items 1, …, i with weight limit w.

We achieve the a similar table as we saw before in the subset sum problem.

We skip the implementation as the function is the same as Function (4.1), with the only difference being that we add v_i instead of S_i at build step. Again the runtime is the dimension of our DP table, $O(nW)$.

Function 4.3: Backtracking Knapsack- *Backtrack()*

```
$i \leftarrow OPT.length$; $w \leftarrow W$; $S \leftarrow \{\}$; // $S$ is our set of jobs
while $i > 0$ AND $w > 0$ do
    if $OPT[i][w] > OPT[i-1][w]$ then
        $S \leftarrow S \cup \{i\}$;
        $w \leftarrow w - w_i$;
    $i \leftarrow i - 1$; // In both cases we $i - 1$
return $S$;
```

Time Complexity: $O(n)$.

2.4.3 Unbounded Knapsack

Definition 4.3: Problem - Unbounded Knapsack

Given a set S of n items, each with a weight w_i and value v_i, and a knapsack of capacity W, find a subset P of S, s.t., $\sum_{i \in P} w_i \leq W$ and $\sum_{i \in P} v_i$ is the highest value achievable. There are infinite copies of each item.

The same logic applies from the above knapsack problem. However, we can now take an item multiple times, meaning we may have to iterate over all possible choices between each item. We proceed as follows:

$$OPT(i, w) = \begin{cases} 0 & \text{if } i == 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\limits_{j=0,\ldots,\infty} \{\underbrace{OPT(i-1, w)}_{\text{next solution}}, \underbrace{j \cdot v_i + OPT(i-1, w - j \cdot w_i)}_{\text{build solution}}\} & \text{else} \end{cases}$$

At each build step we iterate $j \to \infty$, until $w_i \cdot j > w$. This third step may seem 3-dimensional, as in our $n \times W$ table, each entry has another W entries. However, there's no need to store all W entries, as we only need to store j.

Moreover, we don't want to directly store j in our table OPT as it would overwrite the v_i value. Instead, we keep two tables both $n \times W$, so that we can read both values v_i and its j.

Let us call our choices j table C and our value table M. We illustrate this relationship by imagining the two tables sandwiched on top of each other. Then for every i and w we can reference both values at the same time:

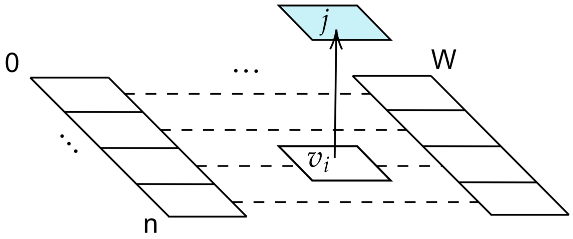


Figure 2.3: Illustrating the relationship between C and M, where v_i relates to its respective j

We implement the following algorithm:

Function 4.4: Unbounded Knapsack - *UnboundedKnapsack()*

```
$M \leftarrow (n+1) \times (W+1)$ table // DP table
$C \leftarrow (n+1) \times (W+1)$ table // Number of copies
$M[0][*] \leftarrow 0$ and $M[*][0] \leftarrow 0$;
for $i \leftarrow 1$ to $n$ do
    for $w \leftarrow 1$ to $W$ do
        $M[i][w] \leftarrow M[i-1][w]$; // Start with previous solution
        $m \leftarrow 1$;
        while $m \cdot w_i \leq w$ do
            $val \leftarrow m \cdot v_i + M[i-1][w - m \cdot w_i]$;
            if $val > M[i][w]$ then
                $M[i][w] \leftarrow val$;
                $C[i][w] \leftarrow m$;
            $m \leftarrow m + 1$;
return $M, C$;
```

Time Complexity: $O(n \times W^2)$ Though we said our tables won't need to be W deep, we still iterate at most W times for each w.
Space Complexity: $O(n \times W)$, as M and C are both $n \times W$, hence $nW + nW = O(nW)$.

Our backtracking algorithm is almost identical to the previous knapsack problem (4.3), however, we now have to consider the number of copies $C[i][w]$.

Function 4.5: Backtracking Unbounded Knapsack - *BKBacktrack()*

```
sol ← ∅;
i ← n and w ← W;
while i > 0 and w > 0 do
    sol ← sol ∪ {i · C[i][w]}; // Add item i with its count C[i][w]
    w ← w − C[i][w] · w_i;
    i ← i − 1;
return sol;
```

Time Complexity: $O(n)$, just like before, as we traverse vertically up our $n \times W$ table, finding at what point $M[i][w]$ change. This indicates i was used in the solution.

Unbounded knapsack (Bottom-Up Approach)

Alternatively we can use a bottom-up approach with a slight modification to our recursive formula. Instead of iterating over j, we iterate over w, continuously taking v_j until we exceed w or move onto the next item. This is shown by:

$$\text{OPT}(i, w) = \begin{cases} 0 & \text{if } i = 0, \\ \text{OPT}(i-1, w) & \text{if } w_i > w, \\ \max\left(\text{OPT}(i-1, w), v_i + \text{OPT}(i, w - w_i)\right) & \text{otherwise.} \end{cases}$$

Same scenario, though now we build our solution $n \times W$ starting from the top left, rather than the bottom right. Meaning we start with one item, then two, then three, and so on.

Each iteration we find the largest weight we can take as we grow our constraint w. While growing w if the solution at at iteration i is greater than the last, we take the item. This will consider all possible combinations.

For an exercise, consider the previous table (2.4.2), and follow the table left to right, top to bottom.

Function 4.6: Unbounded Knapsack - *UnboundedKnapsack2()*

```
$M \leftarrow (n+1) \times (W+1)$ table$^*$ // DP table
$M[0][*] \leftarrow 0$ and $M[*][0] \leftarrow 0$ // Set first row and column to 0
for $i \leftarrow 1$ to $n$ do
    for $w \leftarrow 1$ to $W$ do
        if $w_i > w$ then
            $M[i][w] \leftarrow M[i-1][w]$;
        else
            $val \leftarrow v_i + M[i][w - w_i]$ // Take one more copy of $i$
            if $val > M[i-1][w]$ then
                $M[i][w] \leftarrow val$;
            else
                $M[i][w] \leftarrow M[i-1][w]$;
return $M$;
```

Time Complexity: $O(n \times W)$, as we iterate over each item and weight once.

Function 4.7: Traceback Solution for Unbounded Knapsack - *Traceback()*

```
$i \leftarrow n$ // Start from the last item
$w \leftarrow W$ // Start from the maximum weight
$sol \leftarrow \emptyset$ // Initialize the solution set
while $i > 0$ and $w > 0$ do
    if $w \geq w_i$ and $(v_i + M[i][w - w_i] > M[i-1][w])$ then
        $sol.add(i)$ // Add item $i$ to the solution
        $w \leftarrow w - w_i$ // Reduce the weight
    else
        $i \leftarrow i - 1$ // Move to the previous item
return $sol$;
```

Time Complexity: $O(n + W)$, as we traverse the DP table in reverse order.

2.5 Shortest Paths - Bellman-Ford Algorithm

Revisiting the shortest path problem (1.1), Dijkstra's failed to account for negative edge weights. This was a result of fixing nodes too early in the algorithm, as it assumes paths can only get larger beyond each point. We look to correct this by considering all possible paths from the source node.

Theorem 5.1: Bellman-Ford Algorithm

Given a connected graph G with n nodes and a source node s, begin with setting every node's distance as ∞ and s as 0. Keep a parent-child list to build our solution. Let $d(v) :=$ "current distance of $s \to v$," and $w(v, u) :=$ "edge-weight $v \to u$." Then, for $n - 1$ iterations:

(i.) Iterate over all nodes $v \in G$ starting with s.

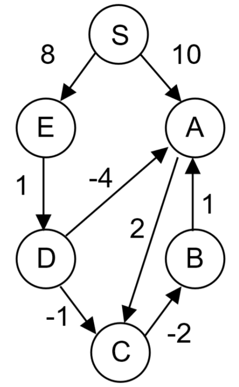
(ii.) For each v, iterate out-degrees u, and evaluate $w(v, u)$.

(iii.) If $d(v) + w(v, u) < d(u)$, update $d(u) = d(v) + w(v, u)$.

(iv.) Update parent-child list with u as child of v.

Say we start with s, and examine all out-degrees u. We update their distances $d(s) + w(s, u)$. Now all u nodes have a distance to contribute to their out-degrees. As hash-tables are unordered it is possible we reach a node $d(v) = \infty$, before an in-degree of v updates it. If such happens we skip the node, as it is not reachable from s.

Thus using the idea that the sub-paths of the shortest path are also shortest paths, we gather that each iteration a new shortest path is found. This allows us to find newer shortest paths.



	0	∞	∞	∞	∞	∞
i	S	A	B	C	D	E
0:	0	10	10	12	9	8
1:	0	5	10	8	9	8
2:	0	5	5	7	9	8
3:	0	5	5	7	9	8

	∅	∅	∅	∅	∅	∅
$P[v]$	S	A	B	C	D	E
	∅	S	C	A	E	S
	∅	D	C	D	E	S
	∅	D	C	A	E	S
	∅	D	C	A	E	S

Figure 2.4: Bellman-Ford Algorithm, where i depicts iterations and $P[v]$ parents of v.

Tip: Bellman-Ford Alg. Live Demo: https://www.youtube.com/watch?v=obWXjtg0L64.

Our first iteration goes as follows, $d(S) = 0$. We check $\underline{min\{\text{evaluation weight}, \text{current weight}\}}$.

- **S:** $d(S) + w(S, A) = 0 + 10; d(A) \leftarrow min\{10, \infty\}$
  $d(S) + w(S, E) = 0 + 8; d(E) \leftarrow min\{8, \infty\}$
- **A:** $d(A) + w(A, C) = 10 + 2; d(C) \leftarrow min\{12, \infty\}$
- **B:** $d(B) = \infty$, currently unreachable from S; skip.
- **C:** $d(C) + w(C, B) = 12 + (-2); d(B) \leftarrow min\{10, \infty\}$
- **D:** $d(D) = \infty$, currently unreachable from S; skip.
- **E:** $d(E) + w(E, D) = 8 + 1; d(D) \leftarrow min\{9, \infty\}$.

Notice how in the second iteration in Figure (2.4), A and C are updated as a direct consequence of us now being able to evaluate paths leaving D. This insight gives us the following theorem:

Theorem 5.2: Bellman-Ford Alg. - Early Termination

We may end the algorithm early if no updates are made in an iteration, as there are no new shortest paths to evaluate.

Though just like Dijkstra's algorithm, Bellman-Ford also has an Achilles' heel. If a negative cycle exists, the algorithm will loop indefinitely, as there will always be a new shortest path. Moreover on the next page.

Definition 5.1: Negative Cycle

A negative cycle is a cycle whose total weight is negative.

Below find the shortest path from $a \rightarrow c$:

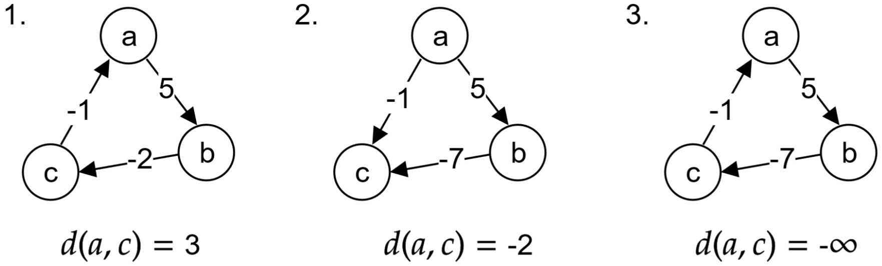


Figure 2.5: Three graphs: 1. A positive cycle, 2. A DAG, 3. A negative cycle.

Examining 3. in Figure (2.5), if we run Bellman-Ford on this graph, we will continuously find a new shortest path to c. We then update the shortest path for $a \rightarrow c$, subsequently updating a new shortest path $a \rightarrow b$, and so on.

Theorem 5.3: Bellman-Ford Alg. - Negative Cycle Detection

To detect a negative cycle, run Bellman-Ford for $n-1$ iterations. If a new shortest path is found in the last iteration, a negative cycle exists. As by then, we should have solidified a solution.

We identify the choices we make in Figure (2.4)'s routine into a recursive formula:

$$OPT(i,v) = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = s, \\ +\infty & \text{if } i = 0 \text{ and } v \neq s, \\ \min \left\{ \begin{array}{l} OPT(i-1,v), \\ \min\limits_{u \in G(v)} \left(OPT(i-1,u) + w(u,v)\right) \end{array} \right\} & \text{if } i > 0. \end{cases}$$

Say we are evaluating node v. We first take the last iteration's shortest path $d(v)$ and compare it to possible paths d'. To find such d' we iterate over all in-degrees $u \to v$ and see if $d(u)+w(u,v) < d(v)$. If so, we update $d(v) = d(u) + w(u,v)$.

In figure (2.4) we see on our second iteration, A and C are updated upon evaluating their new in-degree D. Below is pseudo-code for the Bellman-Ford algorithm, returning a DP table M and a parent list *parents*.

Function 5.1: Bellman-Ford - *BellmanFord()*

```
1  $G \leftarrow$ Graph // Graph $G$ with $n$ nodes
2  $parents \leftarrow$ length-$n$ table // Parents list for shortest paths tree
3  $M \leftarrow n \times n$ table // DP table $M[i][v] = OPT(i,v)$
4  $M[0][*] \leftarrow \infty$ // Set all base values
5  $M[0][s] \leftarrow 0$ // Base case for source
6  for $i \leftarrow 1$ to $n-1$ do
7      for $v \in G$ do
8          $M[i][v] \leftarrow M[i-1][v]$; // Copy previous iteration
9          for $u \in G[v]$ do
10             if $M[i-1][u] > M[i][v] + G[v][u]$ then
11                 $M[i][u] \leftarrow M[i][v] + G[v][u]$;
12                 $parents[u] \leftarrow v$;
13 return $M$, $parents$;
```

Time Complexity: $O(nm)$. We iterate $n-1$ times for $n+m$ edges. Then $O(n(n+m)) = (n^2+nm)$; however, even in tree structures $m = n-1$. So here m could be much larger, s.t., $m \geq n$. Therefore, nm dominates n^2 in the expression. Hence, $O(nm)$.

Let's again examine the above Figure at $i = 1$. Say we reach line 7, with $v = D$. We then copy it's previous iteration, 9, and iterate over it's out-degrees u. We reach $u = A$, where $M[i-1][A] = 10$ and $M[i][D] = 9$. We see $10 > 9 + (-4)$, and update $M[i][A] = 5$ and update $parents[A] = D$.

Bibliography