Greedy Algorithms

Last Updated: January 28th, 2022

1 Introduction

In this lecture we begin the actual "analysis of algorithms" by examining greedy algorithms, which are considered among the easiest algorithms to describe and implement. A **greedy algorithm** is characterized by the following two properties:

1. the algorithm works in stages, and during each stage a choice is made that is locally optimal
2. the sum totality of all the locally optimal choices produces a globally optimal solution

If a greedy algorithm does not always lead to a globally optimal solution, then we refer to it as a **heuristic**, or a **greedy heuristic**. Heuristics often provide a "short cut" (not necessarily optimal) solution. Henceforth, we use the term `algorithm` for a method that always yields a correct/optimal solution, and `heuristic` to describe a procedure that may not always produce the correct or optimal solution.

The following are some problems that that can be solved using a greedy algorithm.

Minimum Spanning Tree finding a spanning tree for a graph whose weight edges sum to a minimum value

Fractional Knapsack selecting a subset of items to load in a container in order to maximize profit

Task Selection finding a maximum set of non-overlapping tasks (each with a fixed start and finish time) that can be completed by a single processor

Huffman Coding finding a code for a set of items that minimizes the expected code-length

Unit Task Scheduling with Deadlines finding a task-completion schedule for a single processor in order to maximize the total earned profit

Single source distances in a graph finding the distance from a source vertex in a weighted graph to every other vertex in the graph

Like all families of algorithms, greedy algorithms tend to follow a similar analysis pattern.

Greedy Correctness Correctness is usually proved through some form of induction. For example, assume their is an optimal solution that agrees with the first k choices of the algorithm. Then show that there is an optimal solution that agrees with the first $k + 1$ choices.

Greedy Complexity The running time of a greedy algorithm is determined by the ease in maintaining an ordering of the candidate choices in each round. This is usually accomplished via a static or dynamic sorting of the candidate choices.

Greedy Implementation Greedy algorithms are usually implemented with the help of a static sorting algorithm, such as Quicksort, or with a dynamic sorting structure, such as a heap. Additional data structures may be needed to efficiently update the candidate choices during each round.

2 Huffman Coding

Huffman coding represents an important tool for compressing information. For example, consider a 1MB text file that consists of a sequence of ASCII characters from the set $\{\text{`A'}, \text{`G'}, \text{`T'}\}$. Moreover, suppose half the characters are A's, one quarter are G's, and one quarter are T's. Then instead of having each byte encode a letter, we instead let each byte encode a sequence of zeros and ones. We do this by assigning the codeword 0 to 'A', 10 to 'G', and 11 to 'T'. Then rather than the sequence $AGATAA$ requiring 6 bytes of memory, we instead store it as the single byte 01001100. Moreover, the average length of a codeword is

$$(1)(0.5) + (2)(0.25) + (2)(0.25) = 1.5 \text{ bits}$$

which yields a compression percentage of 81%, meaning that the file size has been reduced to 0.19 MB. For example, bit length 1 is weighted by 0.5 since half the characters are A's, while 2 is weighted with 0.25 since one quarter of the characters are G's.

With a moment's thought one can see that 1.5 is the least attainable average by any binary **prefix code** that encodes the three characters with respect to the given frequencies, where a prefix code means that no codeword can be a prefix of any other codeword. For example, $\mathcal{C} = \{0, 1, 11\}$ is not a prefix code since 1 is a prefix of 11. The advantage of a prefix code is that the encoding of any sequence of characters is **uniquely decodable**, meaning that the encoding of two different character sequences will produce two different bit sequences.

The **Huffman Coding Algorithm** is a recursive greedy algorithm for assigning an optimal prefix code to a set of characters/members $\mathcal{X} = \{x_1, \ldots, x_n\}$, where element i has weight p_i, with $\sum_{i=1}^{n} p_i = 1$. In the following we let $h(x)$ denote the codeword that Huffman's algorithm assigns to element x.

Base Case If $\mathcal{X} = \{x_1\}$ consists of a single element, then $h(x_1) = \lambda$, the empty word.

Recursive Case Assume $\mathcal{X} = \{x_1, \ldots, x_n\}$, with $n \geq 2$. Without loss of generality, assume $p_1 \geq p_2 \geq \cdots \geq p_n$, and so x_{n-1} and x_n are the two least probable members. Merge these two members into a new member y whose probability equals $p_{n-1} + p_n$. Then apply the algorithm to the input $\mathcal{X}' = \{x_1, \ldots, x_{n-2}, y\}$ to obtain the prefix code $\mathcal{C}'$. Finally, define $\mathcal{C}$ by $h(x_i) = h'(x_i)$, for all $i = 1, \ldots, n-2$, and $h(x_{n-1}) = h'(y) \cdot 0$, $h(x_n) = h'(y) \cdot 1$. In words, we use the returned code $\mathcal{C}'$, and assign x_{n-1} and x_n the codeword asssigned to y followed by a 0 or 1 so that they may be distinguished.

Example 2.1. Apply Huffman's algorithm to $\mathcal{X} = \{1, 2, 3, 4, 5\}$ whose members have respective probabilities $p_1 = 0.3$, $p_2 = 0.25$, $p_3 = 0.2$, $p_4 = 0.15$, $p_5 = 0.10$.

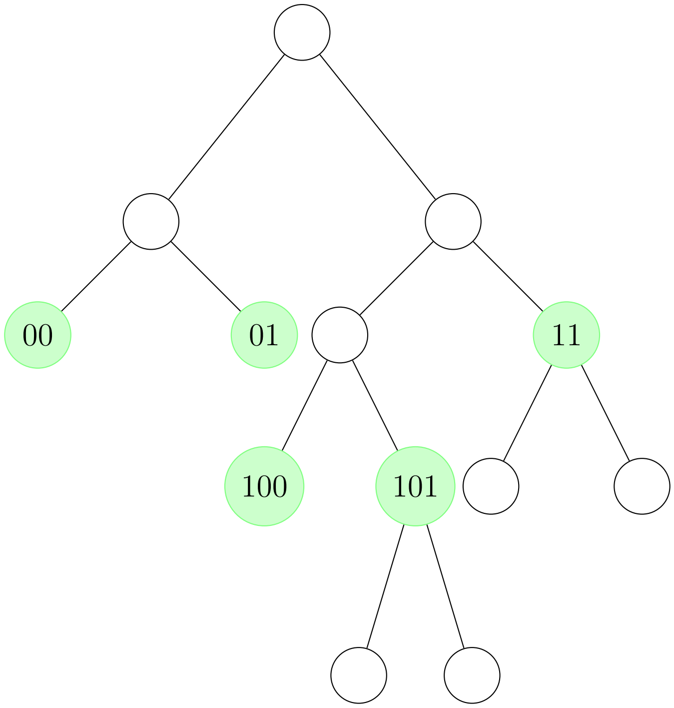


Figure 1: Code tree for prefix code $\mathcal{C} = \{00, 01, 100, 101, 11\}$

Theorem 2.2. Huffman's algorithm is correct in that it always returns an **optimal prefix code**, i.e. one of minimum average bit length.

Proof. It helps to think of the codewords of a binary prefix code as nodes on a binary tree. For example, the codeword 1 represents the right child of the tree root, while 01 represents the right child of the left child of the tree root (or the root's left-right grandchild). Moreover, being a prefix code means that no codeword can be an ancestor of any other codeword. Figure 1 shows a binary code tree for the prefix code $\mathcal{C} = \{00, 01, 100, 101, 11\}$.

Claim: there is an optimal prefix code for $\mathcal{X} = \{x_1, \ldots, x_n\}$ for which the two least probable codewords are (tree) siblings.

Proof of Claim: Without loss of generality (WLOG), assume the respective codeword probabilities are $p_1 > \cdots > p_{n-1} > p_n$. Suppose w_{n-1} and w_n are the two least probable codewords of an optimal prefix code $\mathcal{C}$. First notice that w_{n-1} and w_n must be the two longest codewords. For suppose codeword w_i has a length that exceeds $\max(|w_{n-1}|, |w_n|)$. If this were true then, since $p_i > p_{n-1} > p_n$, we may assign item x_i codeword w_n, and x_n codeword w_i, resulting in a lowering of the average codeword length (show this!) and contradicting the fact that the code is optimal.

The next observation is that we must have $|w_{n-1}| = |w_n|$. For suppose $|w_n| > |w_{n-1}|$, i.e., w_n exists at a lower level of the tree than that of w_{n-1}. Then w_n is the only codeword at this level (why?), and hence its parent is not the ancestor of any other codeword. Thus, w_n may be replaced by its parent to obtain a code of smaller average length, a contradiction.

Finally, given that w_{n-1} and w_n reside at the same (bottom) tree level, if w_n has no sibling codeword, then we may replace w_{n-1} with w_n's sibling, to obtain another (optimal) code having the same average length. On the other hand, if, say, codeword w_{n-2} is the sibling of w_n, then we may swap the codewords of x_{n-1} and x_{n-2} to obtain another (optimal) code in which the two least probable codewords are siblings. $\square$

Continuing with the proof, let $\mathcal{X} = \{x_1, \ldots, x_n\}$ be the item set, and assume $p_1 \geq p_2 \geq \cdots \geq p_n$.

For the basis step, if $n = 1$, then $h(x_1) = \lambda$ is optimal, since the average bit length equals $0(1) = 0$.

For the inductive step, assume Huffman's algorithm always returns an optimal prefix code for sets with $n-1$ or fewer members, for some $n \geq 2$. Consider $\mathcal{X} = \{x_1, \ldots, x_n\}$. Merge x_{n-1} and x_n into the single member y, whose probability is $p_{n-1} + p_n$, and consider $\mathcal{X}' = \{x_1, \ldots, x_{n-2}, y\}$. Then, by the inductive assumption, the recursive call to Huffman's algorithm returns an optimal code $\mathcal{C}_1$. Moreover, we take this code, and replace $h'(y)$ with $h(x_{n-1}) = h'(y) \cdot 0$, and $h(x_n) = h'(y) \cdot 1$, yielding the code $\mathcal{C}_2$ that is returned by Huffman for the original input $\mathcal{X}$ of size n. Now, letting $L(\mathcal{C})$ denote the average length of a prefix code $\mathcal{C}$, we thus have equation

$$L(\mathcal{C}_2) = L(\mathcal{C}_1) + p_{n-1} + p_n,$$

In other words, replacing $h'(y)$ with $h'(y) \cdot 0$ amd $h'(y) \cdot 1$ adds $p_{n-1} + p_n$ more in average length for code $\mathcal{C}_2$.

Now consider an optimal prefix code $\mathcal{C}_3$ for $\mathcal{X}$ in which the two least-probable codewords are siblings, letting $y = \{x_{n-1}, x_n\}$, we may use this code to create the code $\mathcal{C}_4$ for $\mathcal{X}' = \{x_1, \ldots, x_{n-2}, y\}$, with the only change being replacement of codewords $h(x_{n-1}) = w_{n-1}$ and $h(x_n) = w_n$ with their common parent y. This yields

$$L(\mathcal{C}_3) = L(\mathcal{C}_4) + p_{n-1} + p_n.$$

Thus, we have established the two following facts.

1. An optimal code $\mathcal{C}_1$ for $n-1$ members yields a code $\mathcal{C}_2$ for n members whose average length is $p_{n-1} + p_n$ more than that of $\mathcal{C}_1$.
2. An optimal prefix code $\mathcal{C}_3$ for n members yields a prefix code $\mathcal{C}_4$ for $n-1$ members whose average bit length is $p_{n-1} + p_n$ less than that of $\mathcal{C}_3$.

The above two facts imply that

$$L(\mathcal{C}_2) - L(\mathcal{C}_1) = L(\mathcal{C}_3) - L(\mathcal{C}_4),$$

which in turn implies that $L(\mathcal{C}_2) = L(\mathcal{C}_3)$, meaning that $\mathcal{C}_2$ is optimal (since $\mathcal{C}_3$ is optimal). With an eye towards a contradiction, suppose instead we have $L(\mathcal{C}_2) > L(\mathcal{C}_3)$. Then the above equation would force $L(\mathcal{C}_1) > L(\mathcal{C}_4)$, which is a contradiction, since $\mathcal{C}_1$ is optimal by the inductive assumption. Therefore, $\mathcal{C}_2$ is optimal and, since this is the code returned by Huffman for an input of size n, we see that Huffman's algorithm is correct. ☐

Example 2.3. The following table shows each subproblem that is solved by Huffman's algorithm for the problem instance provide in Example 2.1, its optimal code, the code's average length, and how the difference in average length between a parent and child code is equal to the sum of the two least probabilities of the parent code.

i	**Char Set**	**Prob**	**Code**	$L(C_i)$	$L(C_i) - L(C_{i-1})$
1	$\{\{\{1,2\},\{3,\{4,5\}\}\}\}$	$\{1.0\}$	$\{\lambda\}$	0	
2	$\{\{1,2\},\{3,\{4,5\}\}\}$	$\{0.55, 0.45\}$	$\{0, 1\}$	1	$1 - 0 = 0.55 + 0.45$
3	$\{\{3,\{4,5\}\}, 1, 2\}$	$\{0.45, 0.3, 0.25\}$	$\{1, 00, 01\}$	1.55	$1.55 - 1 = 0.3 + 0.25$
4	$\{1, 2, \{4, 5\}, 3\}$	$\{0.3, 0.25, 0.25, 0.2\}$	$\{00, 01, 11, 10\}$	2	$2 - 1.55 = 0.25 + 0.2$
5	$\{1, 2, 3, 4, 5\}$	$\{0.3, 0.25, 0.2, 0.15, 0.1\}$	$\{00, 01, 10, 110, 111\}$	2.25	$2.25 - 2 = 0.15 + 0.1$

3 Minimum Spanning Tree Algorithms

A **graph** $G = (V, E)$ is a pair of sets V and E, where V is the **vertex set** and E is the edge set for which each member $e \in E$ is a pair (u, v), where $u, v \in V$ are vertices. Unless otherwise noted, we assume that G is **simple**, meaning that i) each pair (u, v) appears at most once in E, and ii) G has no loops (i.e. no pairs of the form (u, u) for some $u \in V$), and iii) each edge is undirected, meaning that (u, v) and (v, u) are identified as the same edge.

The following graph terminology will be used repeatedly throughout the course.

Adjacent $u, v \in G$ are said to be **adjacent** iff $(u, v) \in E$.

Incident $e = (u, v) \in E$ is said to be **incident** with both u and v.

Directed and Undirected Graphs G is said to be **undirected** iff, for all $u, v \in V$, the edges (u, v) and (v, u) are identified as the same edge. On the other hand, in a **directed** graph (u, v) means that the edge starts at u and ends at v, and one must follow this order when traversing the edge. In other words, in a directed graph (u, v) is a "one-way street". In this case u is referred to as the **parent** vertex, while b is the **child** vertex.

Vertex Degree The **degree** of vertex v in a simple graph, denoted $\deg(v)$, is equal to the number of edges that are incident with v. Handshaking property: the degrees of the vertices of a graph sum to twice the number of edges of the graph.

Weighted Graph G is said to be **weighted** iff each edge of G has a third component called its **weight** or **cost**.

Path A **path** P in G of **length** k from v_0 to v_k is a sequence of vertices $P = v_0, v_1, \ldots, v_k$, such that $(v_i, v_{i+1}) \in E$, for all $i = 0, \ldots, k-1$. In other words, starting at vertex v_0 and traversing the k edges $(v_0, v_1), \ldots, (v_{k-1}, v_k)$, one can reach vertex v_k. Here v_0 is called the **start vertex** of P, while v_k is called the **end vertex**.

Connected Graph G is called **connected** iff, for every pair of vertices $u, v \in V$ there is a path from u to v in G.

Cycle A path P having length at least three is called a **cycle** iff its start and end vertices are identical. Note: in the case of directed graphs, we allow for cycles of length 2.

Acyclic Graph G is called **acyclic** iff it admits no cycles.

Tree Simple graph G is called a tree iff it is connected and has no cycles.

Forest A **forest** is a collection of trees.

Subgraph $H = (V', E')$ is a subgraph of G iff i) $V' \subseteq V$, ii) $E' \subseteq E$, and iii) $(u, v) \in E'$ implies $u, v \in V'$.

The proof of the following Theorem is left as an exercise.

Theorem 3.1. If $T = (V, E)$ is a tree, then

1. T has at least one degree-1 vertex, and
2. $|E| = n - 1$.

Let $G = (V, E)$ be a simple connected graph. Then a **spanning tree** $T = (V, E')$ of G is a subgraph of G which is also a tree. Notice that T must include all the vertices of G. Thus, a spanning tree of G represents a minimal set of edges that are needed by G in order to maintain connectivity. Moreover, if G is weighted, then a **minimum spanning tree (mst)** of G is a spanning tree whose edge weights sum to a minimum value.

Example 3.2. Consider a problem in which roads are to be built that connect all four cities a, b, c, and d to one another. In other words, after the roads are built, it will be possible to drive from any one city to another. The cost (in millions) of building a road between any two cities is provided in the following table.

cities	a	b	c	d
a	0	30	20	50
b	30	0	50	10
c	20	50	0	75
d	50	10	75	0

Using this table, find a set of roads of minimum cost that will connect the cities.

In this section we present Kruskal's greedy algorithm for finding an MST in a simple weighted connected graph $G = (V, E)$.

3.1 Kruskal's Algorithm

Kruskal's algorithm builds a minimum spanning tree in greedy stages. Assume that $V = \{v_1, \dots, v_n\}$, for some $n \geq 1$. Define forest $\mathcal{F}$ that has n trees $T_1, \dots, T_n$, where T_i consists of the single vertex v_i. Sort the edges of G in order of increasing weight. Now, following this sorted order, for each edge $e = (u, v)$, if u and v are in the same tree T, then continue to the next edge, since adding e will create a cycle in T. Otherwise, letting T_u and T_v be the respective trees to which u and v belong, replace T_u and T_v in $\mathcal{F}$ with the single tree T_{u+v} that consists of the merging of trees T_u and T_v via the addition of edge e. In other words,

$$T_{u+v} = (V_{u+v}, E_{u+v}) = (V_u \cup V_v, E_u \cup E_v \cup \{e\}),$$

and

$$\mathcal{F} \leftarrow \mathcal{F} - T_u - T_v + T_{u+v}.$$

The algorithm terminates when $\mathcal{F}$ consists of a single (minimum spanning) tree.

Example 3.3. Use Kruskal's algorithm to find an mst for the graph $G = (V, E)$, where the weighted edges are given by

$$E = \{(a, b, 1), (a, c, 3), (b, c, 3), (c, d, 6), (b, e, 4), (c, e, 5), (d, f, 4), (d, g, 4),$$

$$(e, g, 5), (f, g, 2), (f, h, 1), (g, h, 2)\}.$$

3.2 Replacement method

The **replacement method** is a method for proving correctness of a greedy algorithm and works as follows.

Greedy Solution Let $S = c_1, \ldots, c_n$ represent the solution produced by a greedy algorithm that we want to show is correct. Note: c_i denotes the i th greedy choice, $i = 1, \ldots, n$.

Optimal Solution Let S_{opt} denote the optimal solution.

First Disagreement Let $k \geq 1$ be the least index for which $c_k \notin S_{\text{opt}}$, i.e. $c_1, \ldots, c_{k-1} \in S_{\text{opt}}$, but not c_k.

Replace Transform S_{opt} into a new optimal solution $\hat{S}_{\text{opt}}$ for which $c_1, \ldots, c_k \in \hat{S}_{\text{opt}}$. Note: this usually requires replacing something in S_{opt} with c_k.

Continue Continuing in this manner, we eventually arrive at an optimal solution that has all the choices made by the greedy algorithm. Argue that this solution must equal the greedy solution, and hence the greedy solution is optimal.

Theorem 3.4. When Kruskal's algorithm terminates, then $\mathcal{F}$ consists of a single minimum spanning tree.

Proof Using Replacement Method.

Greedy Solution Let $T = e_1, e_2, \ldots, e_{n-1}$ be the edges of the spanning tree returned by Kruskal, and written in the order selected by Kruskal. We'll let these edges represent Kruskal's spanning tree T. Note: here n represents the order of problem instance G.

Optimal Solution Let T_{opt} be an mst of G.

First Disagreement Let $k \geq 1$ be the least index for which $e_k \notin T_{\text{opt}}$, i.e. $e_1, \ldots, e_{k-1} \in T_{\text{opt}}$, but not e_k.

Replace Consider the result of adding e_k to T_{opt} to yield the graph $T_{\text{opt}} + e_k$. Then, since $T_{\text{opt}} + e_k$ is connected and has n edges, it must have a cycle C containing e_k.

Claim. There must be some edge e in C that comes after e_k in Kruskal's list of sorted edges. Hence, $w(e) \geq w(e_k)$.

Proof of Claim. Suppose no such edge e exists. Then all edges of C must come before e_k in Kruskal's list of sorted edges. Moreover, these edges fall into two categories: i) those selected by Kruskal (i.e. $e_1, \ldots, e_{k-1}$), and ii) those rejected by Kruskal. However, notice that none of the rejected edges can be in C. This is true since $e_1, \ldots, e_{k-1} \in T_{\text{opt}}$, and so having a rejected edge in T_{opt} would create a cycle. Therefore, this means that $C \subseteq \{e_1, \ldots, e_{k-1}, e_k\}$ which is a contradiction, since $\{e_1, \ldots, e_{k-1}, e_k\} \subseteq T$, and T has no cycles. Therefore, such an edge $e \in C$ does exist. ☐

Now consider $\hat{T}_{\text{opt}} = T_{\text{opt}} - e + e_k$. This is a spanning tree since it is connected and the removal of e eliminates the cycle C. Finally, since $w(e) \geq w(e_k)$, $\text{cost}(\hat{T}_{\text{opt}}) \leq \text{cost}(T_{\text{opt}})$.

Continue Continuing in this manner, we eventually arrive at an mst that has all of Kruskal's edges. But this tree must equal Kruskal's tree, since any two mst's have the same number of edges. ☐

Theorem 3.5. Kruskal's algorithm can be implemented to yield a running time of $T(m,n) = \Theta(m \log m)$, where $m = |E|$.

Proof. Given connected simple graph $G = (V, E)$, sort the edges of E by increasing order of weight using Mergesort. This requires $\Theta(m \log m)$ steps. The only remaining issue involves checking to see if the vertices of an edge e belong in the same tree. If yes, then e is omitted. Otherwise, it is added and merges two of the trees in the Kruskal forest. Thus, checking and merging must both be done efficiently, and we may accomplish both by associating with each graph vertex v a unique **membership node**, or **M-node**, $M(v)$ that has a `parent` attribute, where $M(v)$`.parent` either equals `null`, in which case it is called a **root node**, or references an M-node of some other vertex v' that belongs in the same M-tree as v. In general, we say that M-node n_1 is an **ancestor** of M-node n_2 iff either i) n_1 is referenced by the parent of n_2, or ii) n_1 is the ancestor of the M-node referenced by the parent of n_2. Finally, there is a unique M-tree associated with each tree in the Kruskal forest, and every M-node n belongs to a unique M-tree whose root is an ancestor of n.

Now consider an edge $e = (u, v)$. To determine if e should be added to the solution, we simply trace up and locate the M-tree root nodes associated with $M(u)$ and $M(v)$, and add e to the solution iff the two root nodes are different (i.e. $M(u)$ and $M(v)$ belong to different M-trees). In addition, as a side effect, the `parent` attribute of any node involved in the upward tracing is set to its tree's respective root node, so that a future root-node lookup involving such a node will require O(1) steps. This is referred to as **path compression**. Finally, if $M(u)$ and $M(v)$ belong to different M-trees, then e is added to the solution, and the `parent` of $M(u)$'s root node is now assigned the root node of $M(v)$. This has the effect of merging the two tress, in that M-nodes associated with both trees now possess the same root-node ancestor.

The collection of all M-trees is referred to as the **disjoint-set** data structure, and can be used in any situation where one needs to keep track of a collection of disjoint sets, and perform subsequent union (i.e. merging) and membership-query operations. Moreover, it can be shown that a sequence of m merge and query operations requires a running time $T(m) = \mathrm{O}(\alpha(m)m)$, where $\alpha(m) = o(\log m)$ is an extremely slow growing function. Therefore, Kruskal's algorithm has a running time of $T(m,n) = \Theta(m \log m)$, ☐

We summarize the two M-node operations that are needed for Kruskal's algorithm.

root(n) Returns the M-node that is the root r of the tree for which M-node n belongs. Has the side effect of compressing the path from n to r.

merge(n_1, n_2) Has the effect of assigning the root(n_1) as the parent for root(n_2). This results in the merging of the tree containing n_1 with the tree containing n_2.

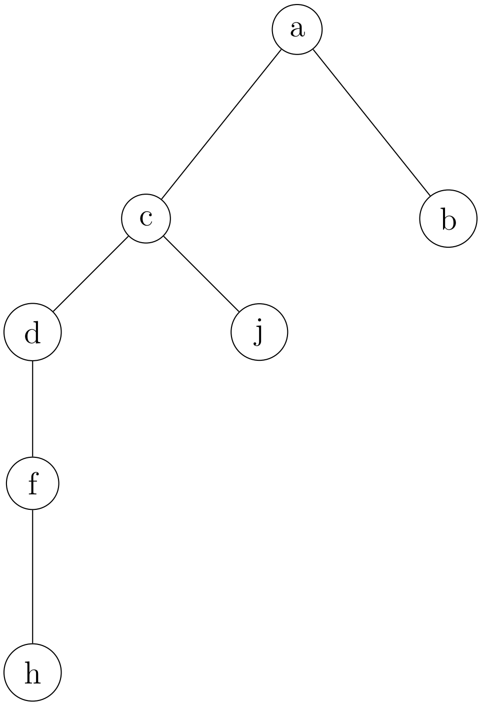


Figure 2: An M-tree

Example 3.6. Figure 2 shows an example of an M-tree. Figure 3 shows that result of calling function root(f)

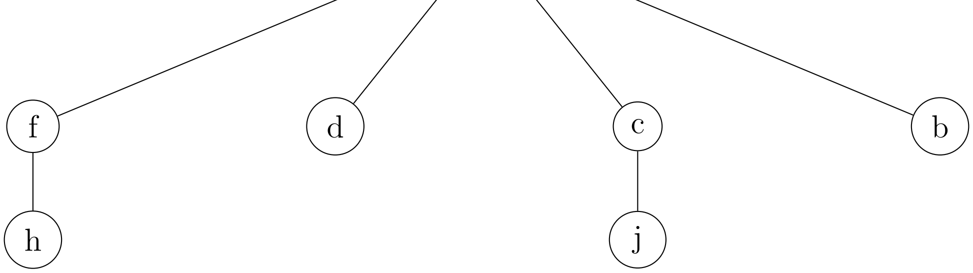


Figure 3: M-tree from Figure 2 after calling root(f).

Example 3.7. For the weighted graph with edges

$$(b, d, 5), (a, e, 4), (a, b, 1), (e, c, 3), (b, f, 6), (e, d, 2),$$

Show how the membership trees change when processing each edge in the Kruskal's sorted list of edges. When merging two trees, use the convention that the root of the merged tree should be the one having *lower* alphabetical order. For example, if two trees, one with root a, the other with root b, are to be merged, then the merged tree should have root a.

Solution. E1. After processing first edge:

E2. After processing second edge:

E3. After processing third edge:

E4. After processing fourth edge:

E5. After processing fifth edge:

E6. After processing sixth edge:

3.3 Prim's Algorithm

Prim's algorithm builds a single tree in stages, where a single edge/vertex is added to the current tree at each stage. Given connected and weighted simple graph $G = (V, E)$, the algorithm starts by initializing a tree $T_1 = (\{v\}, \emptyset)$, where $v \in V$ is a vertex in V that is used to start the tree.

Now suppose tree T_i having i vertices has been constructed, for some $1 \leq i \leq n$. If $i = n$, then the algorithm terminates, and T_n is the desired spanning tree. Otherwise, let T_{i+1} be the result of adding to T_i a single edge/vertex $e = (u, w)$ that satisfies the following.

1. e is incident with one vertex in T_i and one vertex not in T_i.
2. Of all edges that satisfy 1., e has the least weight.

Example 3.8. Demonstrate Prim's algorithm on the graph $G = (V, E)$, where the weighted edges are given by

$$E = \{(a, b, 1), (a, c, 3), (b, c, 3), (c, d, 6), (b, e, 4), (c, e, 5), (d, f, 4), (d, g, 4),$$

$$(e, g, 5), (f, g, 2), (f, h, 1), (g, h, 2)\}.$$

Solution.

Theorem 3.9. Prim's algorithm returns a minimum spanning tree for input $G = (V, E)$.

The proof of correctness of Prim's algorithm is very similar to that of Kruskal's algorithm, and his left as an exercise. Like all exercises in these lectures, the reader should make an honest attempt to construct a proof before viewing the one provided in the solutions.

Prim's algorithm can be efficiently implemented with the help of a binary min-heap. The first step is to build a binary min-heap whose elements are the n vertices. A vertex is in the heap iff it has yet to be added to the tree under construction. Moreover, the priority of a vertex v in the heap is defined as the least weight of any edge $e = (u, v)$, where u is a vertex in the tree. In this case, u is called the **parent** of v, and is denoted as $p(v)$. The current parent of each vertex can be stored in an array. Since the tree is initially empty, the priority of each vertex initialized to ∞ and the parent of each vertex is undefined.

Now repeat the following until the heap is empty. Pop the heap to obtain the vertex u that has a minimum priority. Add u to the tree. Moreover, if $p(u)$ is defined, then add edge $(p(u), u)$ to the tree. Finally, for each vertex v still in the heap for which $e = (u, v)$ is an edge of G, if w_e is less than the current priority of v, then set the priority of v to w_e and set $p(v)$ to u.

The running time of the above implementation is determined by the following facts about binary heaps.

1. Building the heap can be performed in $\Theta(n)$ steps.
2. Popping a vertex from the heap requires $O(\log n)$ steps.
3. When the priority of a vertex is reduced, the heap can be adjusted in $O(\log n)$ steps.
4. The number of vertex-priority reductions is bounded by the number $m = |E|$, since each reduction is caused by an edge, and each edge $e = (u, v)$ can contribute to at most one reduction (namely, that of v's priority) when u is popped from the heap.

Putting the above facts together, we see that Prim's algorithm has a running time of $O(n + n \log n + m \log n) = O(m \log n)$.

Example 3.10. Demonstrate the heap implementation of Prim's algorithm with the graph from Example 3.2.

4 Dijkstra's Algorithm

Let $G = (V, E)$ be a weighted graph whose edge weights are all nonnegative. Then the **cost** of a path P in G, denoted $\text{cost}(P)$, is defined as the sum of the weights of all edges in P. Moreover, given $u, v \in V$, the **distance** from u to v in G, denoted $d(u, v)$, is defined as the minimum cost of a path from u to v. In case there is no path from u to v in G, then $d(u, v) = \infty$.

Dijkstra's algorithm is used to find the distances from a single source vertex $s \in V$ to every other vertex in V. The description of the algorithm is almost identical to that of Prim's algorithm. In what follows we assume that there is at least one path from s to each of the other $n - 1$ vertices in V. Like Prim's algorithm, the algorithm builds a single **Dijkstra distance tree (DDT)** in rounds $1, 2, \ldots, n$, where a single edge/vertex is added to the current tree at each round. We let T_i denote the current DDT after round $i = 1, \ldots, n$. To begin, T_1 consists of the source vertex s.

Now suppose T_i has been defined. A vertex not in T_i is called **external**. For each external vertex, let $d_i(s, v)$ denote the **neighboring distance** from s to v, i.e. the minimum cost of any path from s to v that includes, aside from v, only vertices in T_i. We set $d_i(s, v) = \infty$ in case no such path exists (in this case at least one other external vertex must be visited before v can be reached from s). Then T_{i+1} is obtained by adding the vertex v^* to T_i for which $d_i(s, v^*)$ is minimum among all possible external vertices v. We also add to T_{i+1} the final edge e in the path that achieves this minimum neighboring distance. Notice that e joins a vertex in T_i to v^*.

Then the final DDT is $T = T_n$.

Example 4.1. Demonstrate Dijkstra's algorithm on the directed weighted graph with the following edges.

$$(a, b, 3), (a, c, 1), (a, e, 7), (a, f, 6), (b, f, 4), (b, g, 3), (c, b, 1), (c, e, 7), (c, d, 5), (c, g, 10), (d, g, 1),$$

$$(d, h, 4), (e, f, 1), (f, g, 3), (g, h, 1).$$

The heap implementation of Prim's algorithm can also be used for Dijkstra's algorithm, except now the priority of a vertex v is the minimum of $d(s, u) + w_e$, where $e = (u, v)$ is an edge that is incident with a vertex u in the tree. Also, the priority of s is initialized to zero.

Although we are able to copy the implementation of Prim's algorithm, and appy it to Dijkstra's algorithm, we cannot do the same with Prim's correctness proof, since finding an mst is inherently different from that of finding distances in a graph.

Theorem 4.2. After round i of Dijkstra's algorithm, and for each $v \in T_i$, the cost of the unique path from root s to v in T_i is equal to the distance from s to v in graph G.

Proof by Induction on the round number $i \geq 1$.

Basis Step: Round 1. After round 1, we have $T_1 = \{s\}$ and the distance from s to s equals zero, both in T_1 and in G.

Induction Step. Now assume the theorem's statement is true up to round $i - 1$. In other words, the distances computed from s to other vertices in T_{i-1} are equal to the distances from s to those vertices in G. Now consider the vertex v^* added to T_{i-1} to form T_i. Notice that $d_i(s, v^*)$ equals the distance from s to v^* in T_i, since the unique path from s to v^* in T_i is the same path used to compute $d_i(s, v^*)$. Moreover, if there was a path P from s to v^* in G for which $\text{cost}(P) < d_i(s, v^*)$, then there must exist a first external (relative to T_{i-1}) vertex u in P. Furthermore, this vertex cannot equal v^* since, by definition, any path whose only external vertex is v^* must have a cost that is at least $d_i(s, v^*)$. Finally, by definition, the cost along P from s to u must equal $d_i(s, u) \geq d_i(s, v^*)$, which implies $\text{cost}(P) \geq d_i(s, v^*)$, a contradiction. $\square$

Exercises

1. Use Huffman's algorithm to provide an optimal average-bit-length code for the 7 elements $\{a, b, \ldots, g\}$ whose respective probabilities are

   $$0.2, 0.2, 0.15, 0.15, 0.15, 0.1, 0.05.$$

   Compute the average bit-length of a codeword.

2. Prove that a tree (i.e. undirected and acyclic graph) of size two or more must always have a degree-one vertex.

3. Prove that a tree of size n has exactly $n - 1$ edges.

4. Prove that if a graph of order n is connected and has $n-1$ edges, then it must be acyclic (and hence is a tree).

5. Draw the weighted graph whose vertices are a-e, and whose edges-weights are given by

   $$\{(a, b, 2), (a, c, 6), (a, e, 5), (a, d, 1), (b, c, 9), (b, d, 3), (b, e, 7), (c, d, 5),$$

   $$(c, e, 4), (d, e, 8)\}.$$

   Perform Kruskal's algorithm to obtain a minimum spanning tree for G. Label each edge to indicate the order that it was added to the forest. When sorting, break ties based on the order that the edge appears in the above set.

6. For the weighted graph with edges

   $$(f, e, 5), (a, e, 4), (a, f, 1), (b, d, 3), (c, e, 6), (d, e, 2),$$

   Show how the membership trees change when processing each edge in Kruskal's list of sorted edges. When merging two trees, use the convention that the root of the merged tree should be the one having *lower* alphabetical order. For example, if two trees, one with root a, the other with root b, are to be merged, then the merged tree should have root a.

7. Repeat Exercise 5 using Prim's algorithm. Assume that vertex e is the first vertex added to the mst. Annotate each edge with the order in which it is added to the mst.

8. For the previous exercise. Show the state of the binary heap just before the next vertex is popped. Label each node with the vertex it represents and its priority. Let the initial heap have e as its root.

9. Prove the correctness of Prim's algorithm. Hint: use the proof of correctness for Kruskal's algorithm as a guide.

10. Does Prim's and Kruskal's algorithm work if negative weights are allowed? Explain.

11. Explain how Prim's and/or Kruskal's algorithm can be modified to find a *maximum* spanning tree.

12. Draw the weighted directed graph whose vertices are a-g, and whose edges-weights are given by

$$\{(a,b,2),(b,g,1),(g,e,1),(b,e,3),(b,c,2),(a,c,5),(c,e,2),(c,d,7),(e,d,3),$$
$$(e,f,8),(d,f,1)\}.$$

Perform Dijkstra's algorithm to determine the Dijkstra spanning tree that is rooted at source vertex a. Draw a table that indicates the distance estimates of each vertex in each of the rounds. Circle the vertex that is selected in each round.

13. Let G be a graph with vertices $0, 1, \ldots, n-1$, and let parent be an array, where parent$[i]$ denotes the parent of i for some shortest path from vertex 0 to vertex i. Assume parent$[0] = -1$; meaning that 0 has no parent. Provide a recursive implementation of the function

```
void print_optimal_path(int i, int parent[ ])
```

that prints from left to right the optimal path from vertex 0 to vertex i. You may assume access to a print() function that is able to print strings, integers, characters, etc.. For example,

```
print i
print "Hello"
print ','
```

are all legal uses of print.

14. The **Fuel Reloading Problem** is the problem of traveling in a vehicle from one point to another, with the goal of minimizing the number of times needed to re-fuel. It is assumed that travel starts at point 0 (the origin) of a number line, and proceeds right to some final point $F > 0$. The input includes F, a list of stations $0 < s_1 < s_2 < \cdots < s_n < F$, and a distance d that the vehicle can travel on a full tank of fuel before having to re-fuel. Consider the greedy algorithm which first checks if F is within d units of the current location (either the start or the current station where the vehicle has just re-fueled). If F is within d units of this location, then no more stations are needed. Otherwise it chooses the next station on the trip as the furthest one that is within d units of the current location. Apply this algorithm to the problem instance $F = 25$, $d = 6$, and

$$s_1 = 4, s_2 = 7, s_3 = 11, s_4 = 13, s_5 = 18, s_6 = 20, s_7 = 23.$$

15. Prove that the Fuel Reloading greedy algorithm always returns a minimum set of stations. Hint: use a replacement-type argument similar to that used in proving correctness of Kruskal's algorithm.

16. Given a finite set T of tasks, where each task t is endowed with a start time $s(t)$ and finish time $f(t)$, the goal is to find a subset T_{opt} of T of maximum size whose tasks are pairwise non-overlapping, meaning that no two tasks in T_{opt} share a common time in which both are being executed. This way a single processor can complete each task in T_{opt} without any conflicts.

    Consider the following greedy algorithm, called the **Task Selection Algorithm (TSA)**, for finding T_{opt}. Assume all tasks start at or after time 0. Initialize T_{opt} to the empty set, and initialize variable `last_finish` to 0. Repeat the following step. If no task in T has a start time equal to or exceeding `last_finish`, then terminate the algorithm and return T_{opt}. Otherwise add to T_{opt} the task $t \in T$ for which $s(t) \geq$ `last_finish` and whose finish time $f(t)$ is a minimum amongst all such tasks. Set `last_finish` to $f(t)$.

Implement TSA on the following set of tasks.

Task ID	**Start time**	**Finish Time**
1	2	4
2	1	4
3	2	7
4	4	8
5	4	9
6	6	8
7	5	10
8	7	9
9	7	10
10	8	11

17. Prove that the Task Selection algorithm is correct, meaning that it always returns a maximum set of non-overlapping tasks. Hint: use a replacement-type argument similar to that used in proving correctness of Kruskal's algorithm.

18. Describe an efficient implementationn of the Task Selection algorithm, and provide the algorithm running time under this implementation.

19. Consider the following alternative greedy procedure for finding a maximum set of non-overlapping tasks for the Task Selection problem. Sort the tasks in order of increasing duration. Initialize $S = \emptyset$ to be the set of selected non-overlapping tasks. At each round, consider the task t of least duration that has yet to be considered in a previous round. If t does not overlap with any activity in S, then add t to S. Otherwise, continue to the next task. Prove or disprove that this procedure will always return a set (namely S) that consists of a maxiumum set of non-overlapping tasks.

20. In one or more paragraphs, describe how to efficiently implement the procedure described in the previous exercise. Provide the worst-case running time for your implementation.

21. The **Fractional Knapsack** takes as input a set of goods G that are to be loaded into a container (i.e. knapsack). When good g is loaded into the knapsak, it contributes a weight of $w(g)$ and induces a profit of $p(g)$. However, it is possible to place only a fraction α of a good into the knapsack. In doing so, the good contributes a weight of $\alpha w(g)$, and induces a profit of $\alpha p(g)$. Assuming the knapsack has a weight capacity $M \geq 0$, determine the fraction $f(g)$ of each good that should be loaded onto the knapsack in order to maximize the total container profit.

    The Fractional Knapsack greedy algorithm (FKA) solves this problem by computing the **profit density** $d(g) = p(g)/w(g)$ for each good $g \in G$. Thus, $d(g)$ represents the profit per unit weight of g. FKA then sorts the goods in decreasing order of profit density, and initializes variable `RC` to M, and variable `TP` to 0. Here, `RC` stands for "remaining capacity", while `TP` stands for "total profit". Then for each good g in the ordering, if $w(g) \leq$ `RC`, then the entirety of g is placed into the knapsack, `RC` is decremented by $w(g)$, and `TP` is incremented by $p(g)$. Otherwise, let

$\alpha = \texttt{RC}/w(g)$. Then $\alpha w(g) = \texttt{RC}$ weight units of g is addded to the knapsack, `TP` is incremented by $\alpha p(g)$, and the algorithm terminates.

For the following instance of the FK problem, determine the amount of each good that is placed in the knapsack by FKA, and provide the total container profit. Assume $M = 10$.

good	**weight**	**profit**
1	3	4
2	5	6
3	5	5
4	1	3
5	4	5

22. Prove that the FK algorithm always returns a maximum container profit.

23. Describe an efficient implementationn of the FK algorithm, and provide the algorithm running time under this implementation.

24. The **0-1 Knapsack** problem is similar to Fractional Knapsack, except now, for each good $g \in G$, either all of g or none of g is placed in the knapsack. Consider the following modification of the Fractional Knapsack greedy algorithm. If the weight of the current good g exceeds the remaining capacity `RC`, then g is skipped and the algorithm continues to the next good in the ordering. Otherwise, it adds all of g to the knapsack and decrements `RC` by $w(g)$, while incrementing `TP` by $p(g)$. Verify that this modified algorithm does *not* produce an optimal knapsack for the problem instance of Exercise 21.

25. **Scheduling with Deadlines.** The input for this problem is a set of n tasks $a_1, \ldots, a_n$. The tasks are to be executed by a single processor starting at time $t = 0$. Each task a_i requires one unit of processing time, and has an integer deadline d_i. Moreover, if the processor finishes executing a_i at time t, where $d_i \leq t$, then a profit p_i is earned. For example, if task a_1 has a deadline of 3 and a profit of 10, then it must be either the first, second, or third task executed in order to earn the profit of 10. Consider the following greedy algorithm for maximizing the total profit earned. Sort the tasks in decreasing order of profit. Then for each task a_i in the ordering, schedule a_i at time $t \leq d_i$, where t is the latest time that does not exceed d_i, and for which no other task has yet to be scheduled at time t. If no such t exists, then skip a_i and proceed to the next task in the ordering. Apply this algorithm to the following problem instance. If two tasks have the same profit, then ties are broken by alphabetical order. For example, Task b preceeds Task e in the ordering.

Task	a	b	c	d	e	f	g	h	i	j	k
Deadline	4	3	1	4	3	1	4	6	8	2	7
Profit	40	50	20	30	50	30	40	10	60	20	50

26. Prove that the Task-Scheduling greedy algorithm from the previous exercise always attains the maximum profit. Hint: use a replacement-type argument similar to that used in proving correctness of Kruskal's algorithm.

27. Explain how the Task-Scheduling greedy algorithm can be implemented in such a way to yield a $\Theta(n \log n)$ running time. Hint: use the disjoint-set data structure from Kruskal's algorithm.

28. Given the set of keys $1, \ldots, n$, where key i has weight w_i, $i = 1, \ldots, n$. The weight of the key reflects how often the key is accessed, and thus heavy keys should be higher in the tree. The

Optimal Binary Search Tree problem is to construct a binary-search tree for these keys, in such a way that

$$\mathrm{wac}(T) = \sum_{i=1}^{n} w_i d_i$$

is minimized, where d_i is the depth of key i in the tree (note: here we assume the root has a depth equal to one). This sum is called the **weighted access cost**. Consider the greedy heuristic for Optimal Binary Search Tree: for keys $1, \ldots, n$, choose as root the node having the maximum weight. Then repeat this for both the resulting left and right subtrees. Apply this heuristic to keys 1-5 with respective weights 50,40,20,30,40. Show that the resulting tree does not yield the minimum weighted access cost.

29. Given a simple graph $G = (V, E)$, a **vertex cover** for G is a subset $C \subseteq V$ of vertices for which every edge $e \in E$ is incident with at least one vertex of C. Consider the greedy heuristic for finding a vertex cover of minimum size. The heuristic chooses the next vertex to add to C as the one that has the highest degree. It then removes this vertex (and all edges incident with it) from G to form a new graph $G^{'}$. The process repeats until the resulting graph has no more edges. Give an example that shows that this heuristic does not always find a minimum cover.

Exercise Solutions

1. One such code is $h(a) = 00$, $h(b) = 01$, $h(c) = 100$, $h(d) = 101$, $h(e) = 110$, $h(f) = 1110$, $h(g) = 1111$.

   $$\text{Average bit length} = 2(0.2) + 2(0.2) + 3(0.15) + 3(0.15) + 3(0.15) + 4(0.1) + 4(0.05) = 2.75.$$

2. Consider the longest simple path $P = v_0, v_1, \ldots, v_k$ in the tree. Then both v_0 and v_k are degree-1 vertices. For example, suppose there was another vertex u adjacent to v_0, other than v_1. Then if $u \notin P$, then $P' = u, P$ is a longer simple path than P which contradicts the fact that P is the longest simple path. On the other hand, if $u \in P$, say $u = v_i$ for some $i > 1$, then $P' = u, v_0, v_1, \ldots, v_i = u$ is a path of length at least three that begins and ends at u. In other words, P' is a cycle, which contradicts the fact that the underlying graph is a tree, and hence acyclic.

3. Use the previous problem and mathematical induction. For the inductive step, assume trees of size n have $n-1$ edges. Let $\mathcal{T}$ be a tree of size $n+1$. Show that $\mathcal{T}$ has n edges. By the previous problem, one of its vertices has degree 1. Remove this vertex and the edge incident with it to obtain a tree of size n. By the inductive assumption, the modified tree has $n-1$ edges. Hence $\mathcal{T}$ must have n edges.

4. Use induction.

   Basis step If G has order $n = 1$ and $1 - 1 = 0$ edges, then G is clearly acyclic.

   Inductive step Assume that all connected graphs of order $n-1$ and size $n-2$ are acyclic. Let $G = (V, E)$ be a connected graph of order n, and size $n-1$. Using summation notation, the Handshaking property states that

   $$\sum_{v \in V} \deg(v) = 2|E|.$$

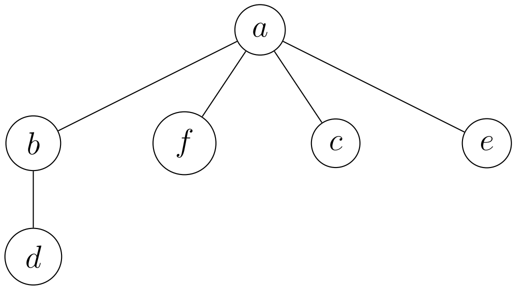
   This theorem implies G must have a degree-1 vertex u. Otherwise,

   $$\sum_{v \in V} \deg(v) \geq 2n > 2|E| = 2(n-1).$$

   Thus, removing u from V and removing the edge incident with u from E yields a connected graph G' of order $n-1$ and size $n-2$. By the inductive assumption, G' is acyclic. Therefore, since no cycle can include vertex u, G is also acyclic.

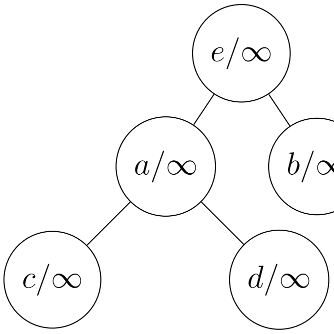
5. Edges added: $(a, d, 1), (a, b, 2), (c, e, 4), (a, e, 5)$ for a total cost of 12.

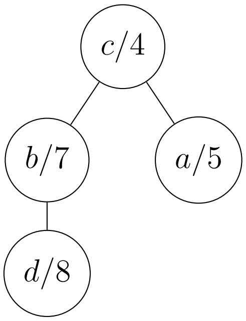
6. The final M-tree is shown below.

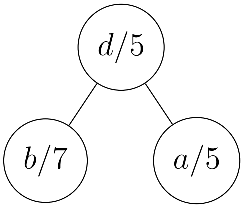


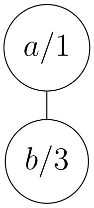
7. Edges added: $(c, e, 4), (c, d, 5), (a, d, 1), (a, b, 2)$ for a total cost of 12.

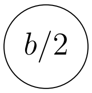
8. The heap states are shown below. Note: the next heap is obtained from the previous heap by i) popping the top vertex u from the heap, followed by ii) performing a succession of priority reductions for each vertex v in the heap for which the edge (u, v, c) has a cost c that less than the current priority of v. In the case that two or more vertices have their priorities reduced, assume the reductions (followed by a percolate-up operation) are performed in alphabetical order.











9. Let T be the tree returned by Prim's Algorithm on input $G = (V, E)$, and assume that $e_1, e_2, \ldots, e_{n-1}$ are the edges of T in the order in which they were added. T is a spanning tree (why?), and we must prove it is an mst. Let T_{opt} be an mst for G that contains edges $e_1, \ldots, e_{k-1}$, but does not contain e_k, for some $1 \leq k \leq n-1$. We show how to transformT_{opt} into an mst T_{opt2} that contains $e_1, \ldots, e_k$.

   Let T_{k-1} denote the tree that consists of edges $e_1, \ldots, e_{k-1}$; in other words, the tree that has been constructed after stage $k-1$ of Prim's algorithm. Consider the result of adding e_k to

T_{opt} to yield the new graph $T_{\text{opt}} + e_k$. Then, since $T_{\text{opt}} + e_k$ is connected and has n edges, $T_{\text{opt}} + e_k$ is not a tree, and thus must have a cycle C containing e_k. Now since e_k is selected at stage k of the algorithm, e_k must be incident with exactly one vertex of T_{k-1}. Hence, cycle C must enter T_{k-1} via e_k, and exit T_{k-1} via some other edge e that is not in T_{k-1}, but is incident with exactly one vertex of T_{k-1}. Thus, e was a candidate to be chosen at stage k, but was passed over in favor of e_k. Hence, $w_{e_k} \leq w_e$.

Now define T_{opt2} to be the tree $T_{\text{opt}}+e_k-e$. Then T_{opt2} has $n-1$ edges and remains connected, since any path in T_{opt} that traverses e can alternately traverse through the remaining edges of C, which are still in T_{opt2}. Thus, T_{opt2} is a tree and it is an mst since e was replaced with e_k which does not exceed e in weight. Notice that T_{opt2} agrees with T in the first k edges selected for T in Prim's Algorithm, where as T_{opt} only agrees with T up to the first $k-1$ selected edges. Therefore, by repeating the above transformation a finite number of times, we will eventually construct an mst that is identical with T, proving that T is indeed an mst.

10. Add a sufficiently large integer J to each edge weight so that the weights will be all nonnegative. Then perform the algorithm, and subtract J from each mst edge weight.

11. For Kruskal's algorithm, sort the edges by *decreasing* edge weight. For Prim's algorithm, use a max-heap instead of a min-heap. Verify that these changes can be successfully adopted in each of the correctness proofs.

12. Edges added in the following order: $(a, b, 2)$, $(b, g, 1)$, $(b, c, 2)$, $(g, e, 1)$, $(e, d, 3)$, $(d, f, 1)$. $d(a, a) = 0$, $d(a, b) = 2$, $d(a, g) = 3$, $d(a, c) = 4$, $d(a, e) = 4$, $d(a, d) = 7$, $d(a, f) = 8$.

13.
```
void print_optimal_path(int i, int parent[ ])
{
     if(i == 0)
          print 0

     print_optimal_path(parent[i], parent);
     print ‘‘ ’’;
     print i;
}
```

14. Minimal set of stations: s_1, s_2, s_4, s_5, s_7.

15. Let $S = s_1, \ldots, s_m$ be the set of stations returned by the algorithm (in the order in which they are visited), and S_{opt} be an optimal set of stations. Let s_k be the first station of S that is not in S_{opt}. In other words, S_{opt} contains stations $s_1, \ldots, s_{k-1}$, but not s_k. Since F is more than d units from s_{k-1} (why ?), there must exits some $s \in S_{\text{opt}}$ for which $s > s_{k-1}$. Let s be such a station, and for which $|s - s_{k-1}|$ is a minimum. Then we must have $s_{k-1} < s < s_k$, since the algorithm chooses s_k because it is the furthest away from s_{k-1} and within d units of s_{k-1}. Now let $S_{\text{opt2}} = S_{\text{opt}} + s_k - s$. Notice that S_{opt2} contains the optimal number of stations. Moreover, notice that, when re-fueling at s_k instead of s, the next station in S_{opt} (and hence in S_{opt2}) can be reached from s_k, since s_k is closer to this station than s. Thus, S_{opt2} is a valid set of stations, meaning that it is possible to re-fuel at these stations without running out of fuel. By repeating the above argument we are eventually led to an optimal set of stations that contain all the stations of S. Therefore, S is an optimal set of stations, and the algorithm is correct.

16. TSA returns $T_{\text{opt}} = \{1, 4, 10\}$.

17. Assume each task t has a positive duration; i.e., $f(t) - s(t) > 0$. Let $t_1, \ldots, t_n$ be the tasks selected by TSA, where the tasks are in the order in which they were selected (i.e. increasing start times). Let T_{opt} be a maximum set of non-overlapping tasks. Let k be the least integer for which $t_k \not\in T_{\text{opt}}$. Thus $t_1, \ldots, t_{k-1} \in T_{\text{opt}}$.

    Claim: $t_1, \ldots, t_{k-1}$ are the only tasks in T_{opt} that start at or before t_{k-1}. Suppose, by way of contradiction, that there is a task t in T_{opt} that starts at or before t_{k-1}, and $t \neq t_i$, $i = 1, \ldots, k-1$. Since t does not overlap with any of these t_i, either t is executed before t_1 starts, in between two tasks t_i and t_{i+1}, where $1 \leq i < k-1$. In the former case, ASA would have selected t instead of t_1 since $f(t) < f(t_1)$. In the latter case, ASA would have selected t instead of t_{i+1}, since both start after t_i finishes, but $f(t) < f(t_{i+1})$. This proves the claim.

    Hence, the first $k-1$ tasks (in order of start times) in T_{opt} are identical to the first $k-1$ tasks selected by TSA. Now let t be the k th task in T_{opt}. Since TSA selected t_k instead of t as the k th task to add to the output set, it follows that $f(t_k) \leq f(t)$. Moreover, since both tasks begin after t_{k-1} finishes, the set $T_{\text{opt2}} - t + t_k$ is a non-overlapping set of tasks (since t_k finishes before t, and starts after t_{k-1} finishes) with the same size as T_{opt}. Hence, T_{opt2} is also optimal, and agrees with the TSA output in the first k tasks.

    By repeating the above argument we are eventually led to an optimal set of tasks whose first n tasks coincide with those returned by TSA. Moreover, this optimal set could not contain any other tasks. For example, if it contained an additional task t, then t must start after t_n finishes. But then the algorithm would have added t (or an alternate task that started after the finish of t_n) to the output, and would have produced an output of size at least $n+1$. Therefore, there is an optimal set of tasks that is equal to the output set of TSA, meaning that TSA is a correct algorithm.

18. It is sufficient to represent the problem size by the number n of input tasks. Sort the tasks in order of increasing start times. Now the algorithm can be completed in the following loop.

    ```
    earliest_finish <- INFINITY
    output <- EMPTY_SET

    for each task t
         if f(t) < earliest_finish
              earliest_finish <- f(t)
              next_selected <- t

         else if s(t) >= earliest_finish
              earliest_finish <- f(t)
              output += next_selected
              next_selected <- t
    ```

    The above code appears to be a correct implementation of TSA. The only possible concern is for a task t that neither satisfies the `if` nor the `else-if` condition. Such tasks never get added to the final set of non-overlapping tasks. To see that this is justified, suppose in the `if` statement t is comparing its finish time $f(t)$ with that of t'. Then we have

    $$s(t') \leq s(t) < f(t'),$$

where the first inequality is from the fact that the tasks are sorted by start times, and the second inequality is from the fact that t does not satisfy the `else-if` condition. Hence, it follows that t and t' overlap, so, if t' is added to the optimal set, then t should not be added. Moreover, the only way in which t' is not added is if there exists a task t'' that follows t in terms of start time, but has a finish time that is less than that of t''s. In this case we have $s(t) \leq s(t'')$ and $f(t) \geq f(t') \geq f(t'')$ and so t overlaps with t''. And once again t should not be added to the final set.

Based on the above code and analysis, it follows that TSA can be implemented with an initial sorting of the tasks, followed by a linear scan of the sorted tasks. Therefore, $T(n) = \Theta(n \log n)$.

19. Hint: consider the case where there are three tasks t_1, t_2, and t_3, where there is overlap between t_1 and t_2, and t_2 and t_3.

20. The most efficient implementation has running time $\Theta(n \log n)$. Hint: your implementation should make use of a balanced (e.g. AVL) binary search tree.

21. The table below shows the order of each good in terms of profit density, how much of each good was placed in the knapsack, and the profit earned from the placement. The total profit earned is 14.4.

good	**weight**	**profit**	**density**	**placed**	**profit earned**
4	1	3	3	1	3
1	3	4	1.3	3	4
5	4	5	1.25	4	5
2	5	6	1.2	2	2.4
3	5	5	1	0	0

22. Let $(g_1, w_1), \ldots, (g_n, w_n)$ represent the ordering of the goods by FKA, where each w_i represents the amount of g_i that was added to the knapsack by FKA. Let C_{opt} be an optimal container, and let (g_k, w_k) be the first pair in the ordering for which w_k is not the amount of g_k that appears in C_{opt}. Thus, we know that C_{opt} has exactly w_i units of g_i, for all $i = 1, \ldots, k-1$. As for g_k, we must have $w_k > 0$. Otherwise, FKA filled the knapsack to capacity with $(g_1, w_1), \ldots, (g_{k-1}, w_{k-1})$, which means that C_{opt} could only assign 0 units of capacity for g_k, which implies C_{opt} agrees with FKA up to k, a contradiction. Moreover, it must be the case that C_{opt} allocates weight w for g_k, where $w < w_k$. This is true since FKA either included all of g_k in the knapsack, or enough of g_k to fill the knapsack. Thus, C_{opt} can allocate no more of g_k than that which was allocated by FKA. Now consider the difference $w_k - w$. This capacity must be filled in C_{opt} by other goods, since C_{opt} is an optimal container. Without loss of generality, assume that there is a single good g_l, $l > k$, for which C_{opt} allocates at least $w_k - w$ units for g_l. Then the total profit being earned by these weight units is $d(g_l)(w_k - w)$. But, since $l > k$, $d(g_l) \leq d(g_k)$, which implies

$$d(g_l)(w_k - w) \leq d(g_k)(w_k - w).$$

Now let C_{opt2} be the container that is identical with C_{opt}, but with $w_k - w$ units of g_l replaced with $w_k - w$ units of g_k. Then the above inequality implies that C_{opt2} must also be optimal, and agrees with the FKA container on the amount of each of the first k placed goods.

By repeating the above argument, we are eventually led to an optimal container that agrees with the FKA container on the amount to be placed for each of the n goods. In other words, FKA produces an optimal container.

23. The parameters n, and $\log M$ can be used to represent the problem size, where n is the number of goods. Notice how $\log M$ is used instead of M, since $\log M$ bits are needed to represent capacity M. Furthermore, assume each good weight does not exceed M, and the good profits use a constant number of bits. Then the sorting of the goods requires $\Theta(n \log n)$ steps, while the profit density calculations and updates of variables `RC` and `TP` require $O(n \log M + n)$ total steps. Therefore, the running time of FKA is $T(n) = O(n \log n + n \log M)$.

24. The table below shows the order of each good in terms of profit density, how much of each good was placed in the knapsack by modified FKA, and the profit earned from the placement. The total profit earned is 12. However, placing goods 2, 4, and 5 into the knapsack earns a profit of $14 > 12$. An alternative algorithm for 0-1 Knapsack will be presented in the Dynamic Programming lecture.

good	**weight**	**profit**	**density**	**placed**	**profit earned**
4	1	3	3	1	3
1	3	4	1.3	3	4
5	4	5	1.25	4	5
2	5	6	1.2	2	0
3	5	5	1	0	0

25. The optimal schedule earns a total profit of 300, and is shown below.

Time	1	2	3	4	5	6	7	8
Task	7	5	2	1		8	11	9
Profit	40	50	50	40		10	50	60

26. Let $(a_1, t_1), \ldots, (a_m, t_m)$ represent the tasks that were selected by the algorithm for scheduling, where a_i is the task, and t_i is the time that it is scheduled to be completed, $i = 1, \ldots, m$. Moreover, assume that these tasks are ordered in the same order for which they appear in the sorted order. Let S_{opt} be an optimal schedule which also consists of task-schedule-time pairs. Let k be the first integer for which $(a_1, t_1), \ldots, (a_{k-1}, t_{k-1})$ are in S_{opt}, but $(a_k, t_k) \notin S_{\text{opt}}$. There are two cases to consider: either a_k does not appear in S_{opt}, or it does appear, but with a different schedule time.

    First assume a_k does not appear in S_{opt}. Let a be a task that is scheduled in S_{opt} that is different from a_i, $i = 1, \ldots, k-1$, and is scheduled at time d_k. We now a must exist, since otherwise (a_k, d_k) could be added to S_{opt} to obtain a more profitable schedule. Now if $p(a) > p(a_k)$, then a comes before a_k in the sorted order. But since $a \neq a_i$, for all $i = 1, \ldots, k-1$, it follows that it is impossible to schedule a together with each of $a_1, \ldots, a_{k-1}$ (otherwise the algorithm would have done so), which is a contradiction, since S_{opt} schedules all of these tasks, and schedules $a_1, \ldots, a_{k-1}$ at the same times that the algorithm does. Hence, we must have $p(a) \leq p(a_k)$. Now define $S_{\text{opt2}} = S_{\text{opt}} - (a, d_k) + (a_k, d_k)$. Then S_{opt2} is an optimal schedule that agrees with the algorithm schedule up to the first k tasks.

    Now assume a_k appears in S_{opt}, but is scheduled at a different time $t \neq t_k$. First notice that t cannot exceed t_k, since the algorithm chooses the first unoccupied time that is closest to a task's deadline. Thus, every time between $t_k + 1$ and d_k (inclusive) must already be occupied by a task from $a_1, \ldots, a_{k-1}$, and hence these times are not available for a_k in S_{opt}. Thus, $t < t_k$. Now if t_k is unused by S_{opt}, then let $S_{\text{opt2}} = S_{\text{opt}} - (a_k, t) + (a_k, t_k)$. On the other hand, if t_k is used by some task a, then let

$$S_{\text{opt2}} = S_{\text{opt}} - (a_k, t) - (a, t_k) + (a_k, t_k) + (a, t).$$

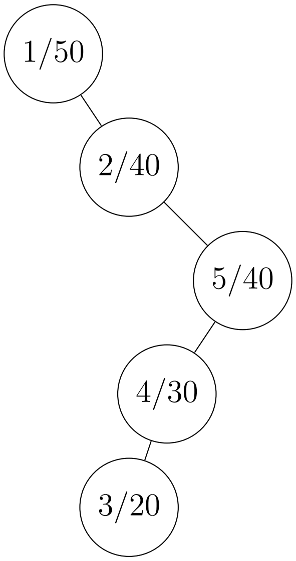
In both cases S_{opt2} is an optimal schedule that agrees with the algorithm schedule up to the first k tasks.

By repeating the above argument, we are eventually led to an optimal schedule that entirely agrees with the algorithm schedule. In other words, the algorithm produces an optimal schedule.

27. If one uses a naive approach that starts at a task's deadline and linearly scans left until an open time slot is found, then the worst case occurs when each of the n tasks has a deadline of n and all have the same profit. In this case task 1 is scheduled at n, task 2 at $n-1$, etc.. Notice that, when scheduling task i, the array that holds the scheduled tasks must be queried $i-1$ times before finding the available time $n-i+1$. This yields a total of $0+1+\cdots+n-1=\Theta(n^2)$ queries. Thus, the algorithm has a running time of $T(n)=\mathrm{O}(n^2)$.

    To improve the running time, we may associate an M-node (and hence an M-tree) with each time slot. Then if M-nodee n is associated with time slot t, and lies in M=tree T, then any task with a deadline of t is scheduled at time s, where the M-node of s is the root of T. Thus, scheduling a task requires a single M-tree membership query, followed by a single M-tree merging in which the M-tree associdated with s is merged with the M-tree associated with time $s-1$. This is necessary since time s is no longer available, and so any task that is directed to s must now be re-directed to a time for which any $s-1$-deadline task would get directed. Thus, a total of $2n$ membership-query and merge operations are required, yielding a running time of $T(n)=\alpha(n)n$, where $\alpha(n)=o(\log n)$. Therefore, the worst-case time is the $\Theta(n\log n)$ time required to sort the tasks.

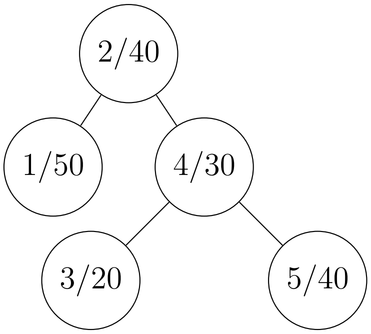
28. The heuristic produces the tree below.



Its weighted access cost equals

$$50(1)+40(2)+40(3)+30(4)+20(5)=470.$$

However, a binary-search tree with less weighted-access cost (380) is shown below.



29. In the graph below, the heuristic will first choose vertex a, followed by four additional vertices (either b, d, f, h, or c, e, g, i), to yield a cover of size five. However, the optimal cover $\{c, e, g, i\}$ has a size of four.