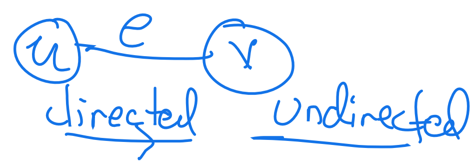
Greedy Graph Algorithms

Last Updated: October 4th, 2023

1 Review of Graph Terminology

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**.

Simple Path $P = v_0, v_1, \ldots, v_k$ is a called a **simple path** iff $v_0, v_1, \ldots, v_k$ are all distinct.

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

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

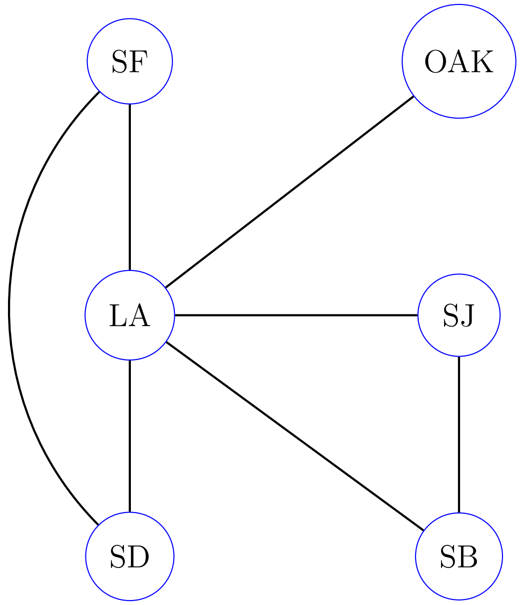


Figure 1: Graphical Representation of G

Example 1.2. Let $G = (V, E)$, where

$$V = \{SD, SB, SF, LA, SJ, OAK\}$$

are cities in California, and

$$E = \{(SD, LA), (SD, SF), (LA, SB), (LA, SF), (LA, SJ), (LA, OAK), (SB, SJ)\}$$

are edges, each of which represents the existence of one or more flights between two cities. Figure 1 shows a graphical representation of G. G has order 6 and size 7.

Figure 2 shows a simple path of length 4. Figure 3 shows a cycle of length 3. Let's verify the Handshaking theorem.

$$\deg(\text{SF}) + \deg(\text{LA}) + \deg(\text{SD}) + \deg(\text{OAK}) + \deg(\text{SJ}) + \deg(\text{SB}) =$$

$$2 + 5 + 2 + 1 + 2 + 2 = 14 = 2 \cdot 7 = 2|E|.$$

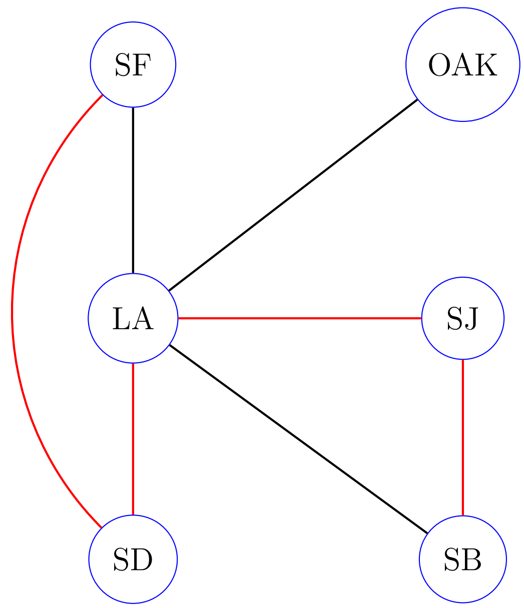


Figure 2: Simple path (in red) $P =$ SF,SD,LA,SJ,SB of length 4

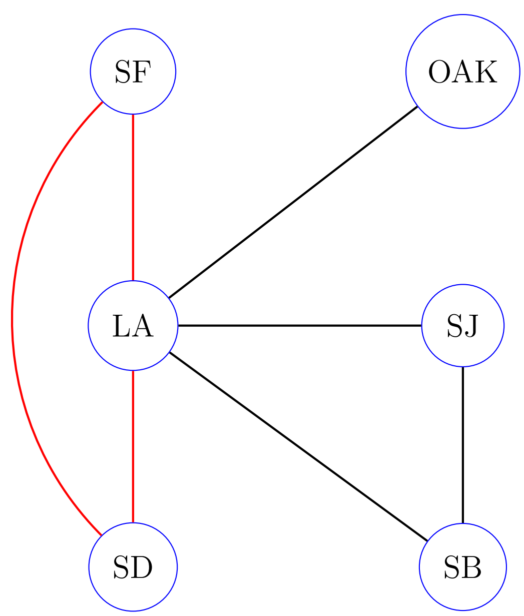


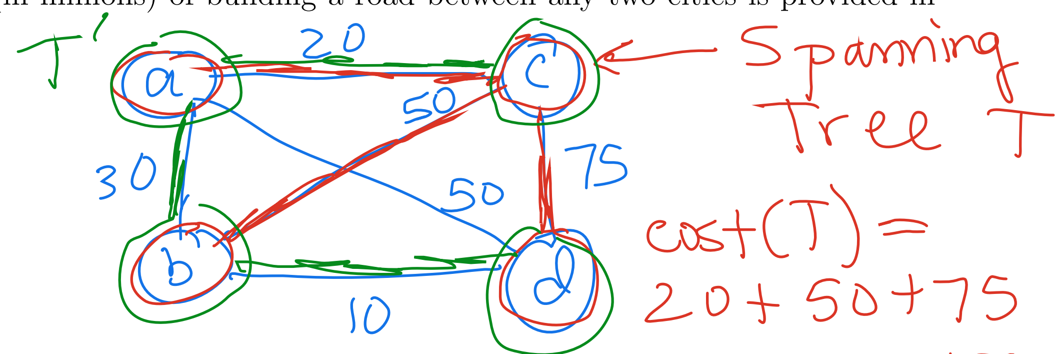
Figure 3: Cycle (in red) $C =$ SF,SD,LA,SF of length 3

2 Minimum Spanning Tree Algorithms

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 2.1. 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



cost(T) = 20 + 50 + 75 = 145

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

T is not a minimum spanning Tree

Cost(T′) = 20 + 30 + 10 = 60

T′ is a minimum spanning Tree (mst)

2.1 Kruskal's Algorithm

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

Kruskal's algorithm builds a minimum spanning tree in greedy stages. Assume that $V = \{v_1, \ldots, v_n\}$, for some $n \geq 1$. Define forest $\mathcal{F}$ that has n trees $T_1, \ldots, T_n$, 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,

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

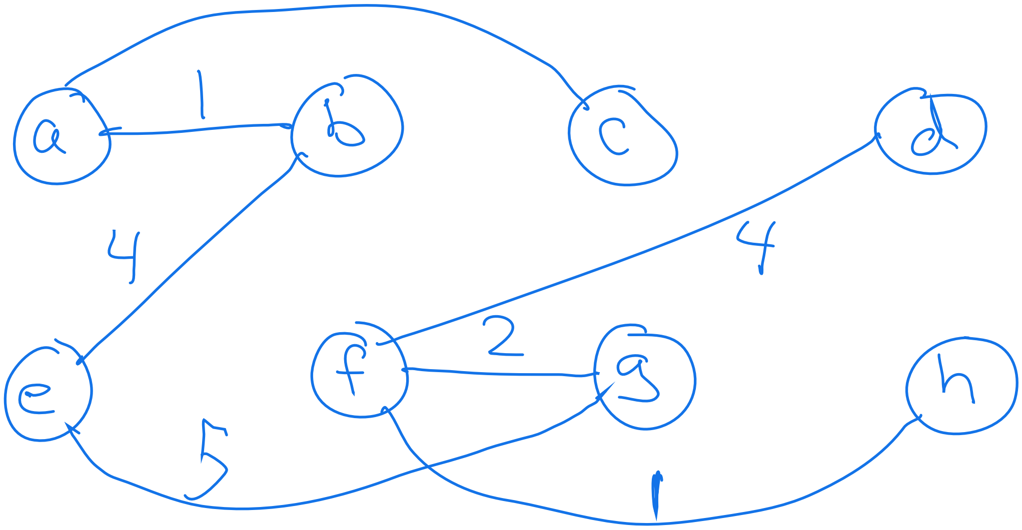
and

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

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

Example 2.2. Use Kruskal's algorithm to find an mst for the graph $G = (V, E)$, where the weighted edges are given by

$$E = \{(a, b, 1), (a, c, 3), (b, c, 3), (c, d, 6), (b, e, 4), (c, e, 5), (d, f, 4), (d, g, 4),$$
$$(e, g, 5), (f, g, 2), (f, h, 1), (g, h, 2)\}.$$



$\therefore$ Cost(T) = 1 + 3 + 4 + 5 + 1 + 2 + 4 = 20

2.2 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.

Example 2.3. An instance of the `Task Selection` problem is a finite set T of tasks, where each task t has a start time $s(t)$ and finish time $f(t)$ which indicate the interval for which the task should be completed by a single processor. 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. Show the set of tasks that result from the following greedy algorithm. Sort the tasks based on increasing finish time. Add the first task t in the sorted order to the solution set. And let f denote its finish time. Next, add to the solution set the first task t' (if it exists) in the order for which $s(t') \geq f$. Assign $f \leftarrow f(t')$ and repeat until all tasks in the sorted order have been considered.

(2, 11, 17) & (10, 9, 20) overlap

Apply the algorithm to the following set of tasks, where each triple in set T represents the id, start time, and finish time.

Id s f   Id

$$T = \{(1,9,12),(2,11,17),(3,10,12),(4,2,14),(5,2,7),(6,4,9),(7,18,19),(8,5,17),(9,6,17),$$
$$(10,9,20),(11,1,13),(12,9,12),(13,6,15),(14,3,5),(15,16,17)\}.$$

Solution = { (14, 3, 5), (1, 9, 12), (15, 16, 17), (7, 18, 19) }

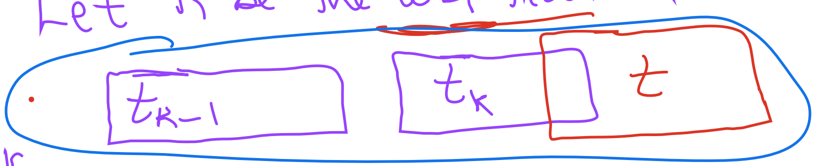
Last Finish Times: 0, 5, 12, 17, 19 → Stop: no task starts at or after 19

Example 2.4. Use the Replacement Method to prove the correctness of the algorithm described in the previous example.

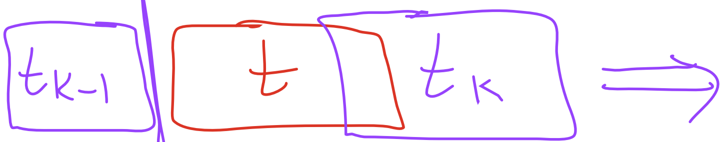
1. Let $S = t_1, t_2, \ldots, t_m$ be the solution returned by TSA

2. S_{opt} = an optimal solution

3. Let k be the least index for which $t_k \notin S_{opt}$. $t_1, \ldots, t_{k-1} \in S_{opt}$

4. 

$t \in S_{opt}$ is the earliest task in S_{opt} that follows t_{k-1}.



$\Longrightarrow$ TSA would have chosen t instead of t_k, a contradiction

6. 

$\Longrightarrow$ S_{opt} is not optimal

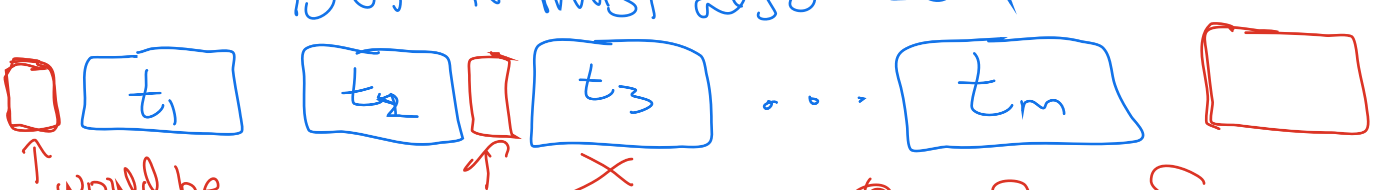
Let $\hat{S}_{opt} = S_{opt} - \{t\} \cup \{t_k\}$

which agrees with S up to index k.

5. Continuing, we can show that there is an S_{opt} for which $t_1, t_2, \ldots, t_m \in S_{opt}$. i.e.

$$S \subseteq S_{opt}$$

But it must also be true that $S_{opt} \subseteq S$



would be selected first by TSA

$S = S_{opt} \Longrightarrow$ TSA is optimal

Theorem 2.5. 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:

1. edges selected by Kruskal (i.e. $e_1, \ldots, e_{k-1}$), and
2. edges 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. $\square$

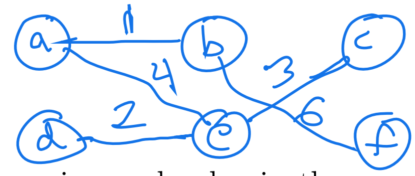
Now consider $\hat{T}_{\text{opt}} = T_{\text{opt}} - e + e_k$. This is a spanning tree since it is connected and the removal of e eliminates the cycle C. Finally, since $w(e) \geq w(e_k)$, $\text{cost}(\hat{T}_{\text{opt}}) \leq \text{cost}(T_{\text{opt}})$.

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. $\square$

Theorem 2.6. 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. This can be done with the use of the disjoint-set data structure. Moreover, since the algorithm begins with n trees in the forest $\mathcal{F}$ and there are at $O(m + n)$ disjoint-set operations, by Theorem 2.9 of the Greedy Algorithm Introduction lecture, checking tree membership of edge vertices can be done in $O(n + m \log^* n)$ steps. Therefore, the algorithm's running time is dominated by the sorting step to give $T(m, n) = \Theta(m \log m)$. □

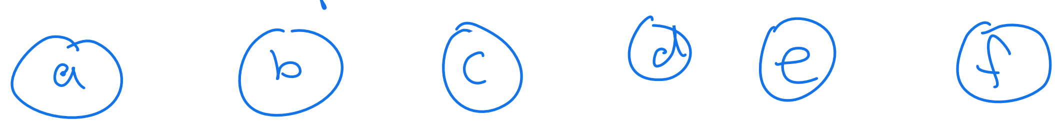
Example 2.7. For the weighted graph with edges



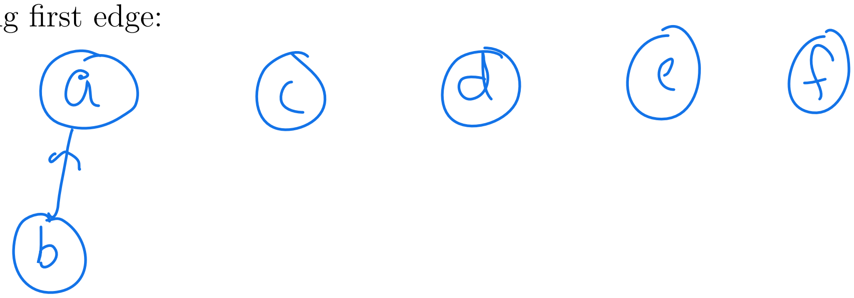
$$(b, d, 5), (a, e, 4), (a, b, 1), (e, c, 3), (b, f, 6), (e, d, 2),$$

Show how the forest of disjoint-set data structure trees changes 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.

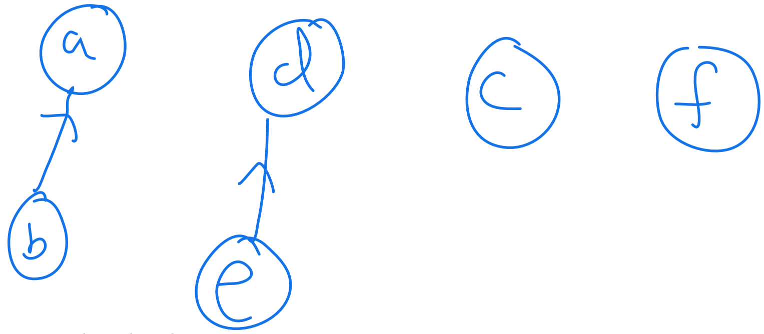
Solution.
Original Forest



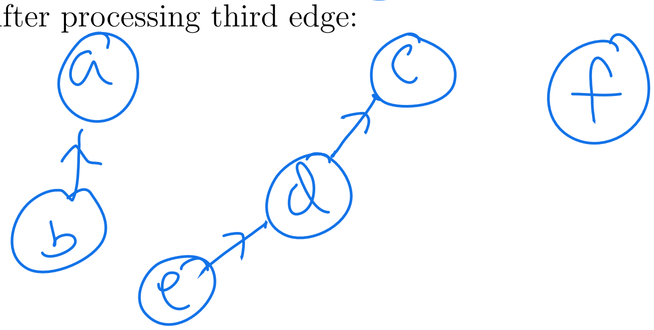
E1. After processing first edge:



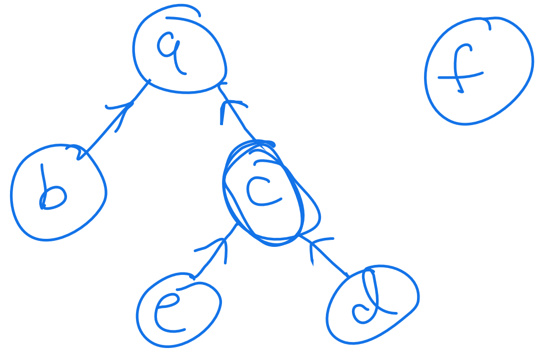
E2. After processing second edge:



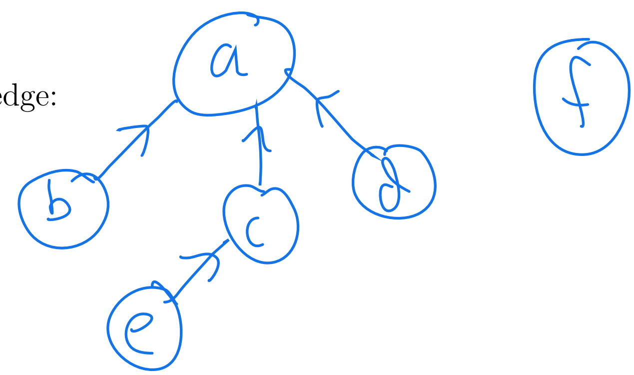
E3. After processing third edge:



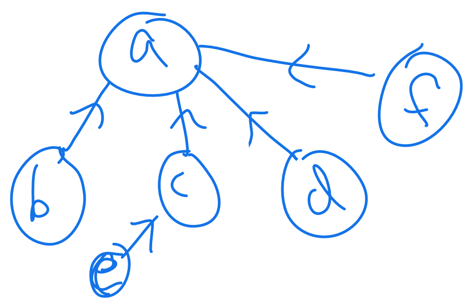
E4. After processing fourth edge:



E5. After processing fifth edge:



E6. After processing sixth edge:



2.3 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.

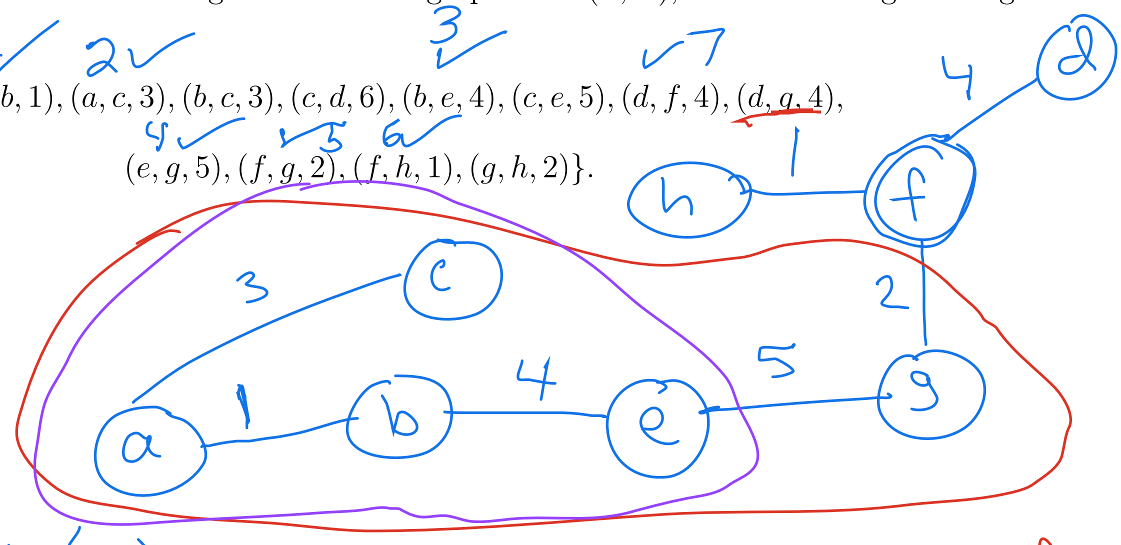
Now suppose tree T_i having i vertices has been constructed, for some $1 \leq i \leq n$. If $i = n$, then the algorithm terminates, and T_n is the desired spanning tree. Otherwise, let T_{i+1} be the result of adding to T_i a single edge/vertex $e = (u, w)$ that satisfies the following.

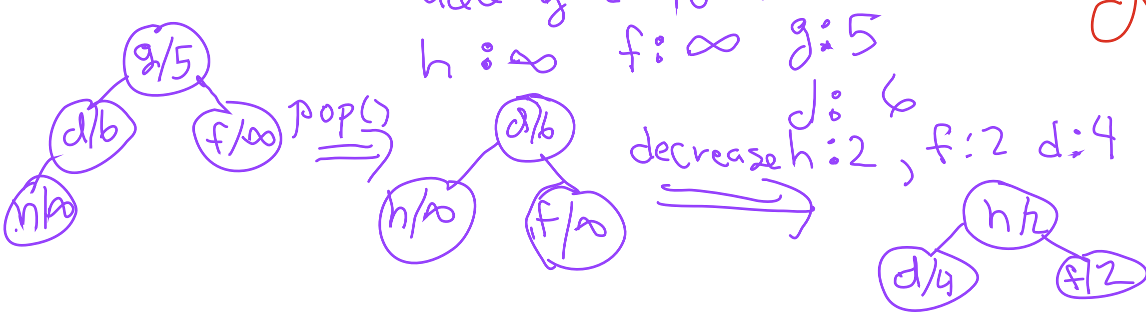
1. e is incident with one vertex in T_i and one vertex not in T_i.
2. Of all edges that satisfy 1., e has the least weight.

Example 2.8. 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 2.9. 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 is 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 2.10. For the heap H used in the implementation of Prim's algorithm, provide a plausible state for H once the size of Prim's tree reaches four in Example 2.8, and any `increase_priority` operations have been executed. Demonstrate the `pop` and `increase_priority` operations (if necessary) that occur as the result of adding the 5th vertex to Prim's tree.

- See Example 2.8 for Solution

3 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 $d(u, v) = \infty$.

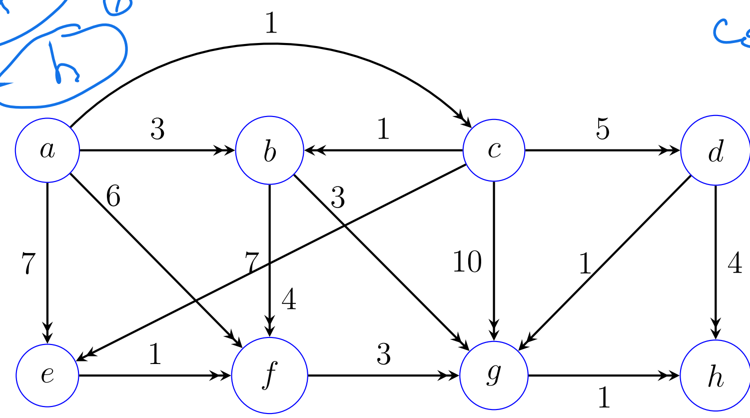
Dijkstra's algorithm is used to find the distances from a single source vertex $s \in V$ to every other vertex in V. The description of the algorithm is almost identical to that of Prim's algorithm. In what follows we assume that there is at least one path from s to each of the other $n - 1$ vertices 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 DDT_i denote the current DDT after round $i = 1, \ldots, n$. To begin, $\text{DDT}_0 = \emptyset$ denotes the empty tree and DDT_1 consists of the source vertex s.

Now suppose DDT_i has been defined. A vertex not in DDT_i is called **external**. An ***i*-neighboring path** from s to an external vertex v is any path from s to v that uses exactly one edge that is *not* in DDT_i. For each external vertex, let $d_i(s, v)$ denote the ***i*-neighboring distance** from s to v, i.e. the minimum cost of any i-neighboring path from s to v. We set $d_i(s, v) = \infty$ in case no such path exists (in this case we say that v is not an ***i*-neighbor** of s). Then DDT_{i+1} is obtained by adding the vertex v^* to DDT_i for which $d_i(s, v^*)$ is minimum among all possible external vertices. We also add to DDT_{i+1} the final edge e in the minimum-cost i-neighboring path from s to v^*. that achieves this minimum i-neighboring distance. Notice that e joins a vertex in DDT_i to v^*.

Then the final DDT is $\text{DDT} = \text{DDT}_n$.

Example 3.1. Demonstrate Dijkstra's algorithm on the directed weighted graph with the following edges.

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



Vertex	0nd / par	1nd / par	2nd / par	3nd / par	4nd / par	5nd / par	6nd / par	7nd / par
a	0/a	—	—	—	—	—	—	—
b	∞/∅	3/a	2/c	—	—	—	—	—
c	∞/∅	1/c	—	—	—	—	—	—
d	∞/∅	∞/∅	6/c	6/c	6/c	—	—	—
e	∞/∅	7/a	7/a	7/a	7/a	7/a	7/a	7/a
f	∞/∅	6/a	6/a	6/a	6/a	6/a	—	—
g	∞/∅	∞/∅	11/c	5/b	—	—	—	—
h	∞/∅	∞/∅	∞/∅	∞/∅	6/g	6/g	6/g	—

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.

Example 3.2. For the heap H used in the implementation of Dijkstra's algorithm in Example 3.1, provide a plausible state for H once the size of Dijkstra's tree reaches three, and any `increase_priority` operations have been executed. Demonstrate the `pop` and `increase_priority` operations (if necessary) that occur as the result of adding the 4th vertex to Dijkstra's tree.

See Previous Page

The following theorem establishes the correctness of Dijkstra's algorithm.

Theorem 3.3. . Let $d_i(s, v^*)$ be the minimum i-neighboring distance among all vertices that are external to DDT_i. Then

$$d_i(s, v^*) = d(s, v^*).$$

Proof of Theorem 3.3. Let P be the i-neighboring path from s to v^* for which $\mathrm{cost}(P) = d_i(s, v^*)$. Let R be any other path from s to v^*. Then

$$R = s, \dots, u, v, \dots, v^*,$$

where $s, \dots, u \in \mathrm{DDT}_i$, and $v \notin \mathrm{DDT}_i$. In other words, v is the first vertex reached by R that is not in DDT_i. Vertex v must exist since $v^* \notin \mathrm{DDT}_i$. Thus, $Q = s, \dots, u, v$ is an i-neighboring path and, since P has the minimum cost of all such paths and Q is a subpath of R, we have

$$\mathrm{cost}(R) \geq \mathrm{cost}(Q) \geq \mathrm{cost}(P).$$

Therefore, P is the minimum-cost path from s to v^*, i.e.,

$$d_i(s, v^*) = d(s, v^*).$$

□

Exercises

1. Prove that a tree T (i.e. undirected and acyclic graph) of size two or more must always have a degree-one vertex. Hint: consider the longest simple path in T. What can you say about its start and end vertices and why?

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

3. 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).

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

$$\{(a,b,2),(a,c,6),(a,e,5),(a,d,1),(b,c,9),(b,d,3),(b,e,7),(c,d,5),$$
$$(c,e,4),(d,e,8)\}.$$

   Informally perform Kruskal's algorithm to obtain a minimum spanning tree for G. Label each edge to indicate its order in the Kruskal sorted order. that it was added to the forest. Break ties be giving precedence to the edge that comes first in the above list of edges.

5. Repeat the steps of Example 2.7 but using the graph whose edge set is

$$E = \{(f,e,5),(a,e,4),(a,f,1),(b,d,3),(c,e,6),(d,e,2)\}.$$

   Show how the membership trees change when processing each edge in Kruskal's list of sorted edges. When unioning two trees, use the convention that the root of the resulting 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 unioned, then the resulting tree should have root a.

6. Repeat Exercise 4 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.

7. 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.

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

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

10. 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.

11. 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.

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

13. Prove that the Fuel Reloading greedy algorithm (See Exercise 1 of "Greedy Algorithms Overview" Lecture) always returns a minimum set of stations. Hint: use a replacement-type argument similar to that used in proving correctness of Kruskal's algorithm.

14. Prove that the Task Selection algorithm (See Exercise 2 of "Greedy Algorithms Overview" Lecture) is correct, meaning that it always returns a maximum set of non-overlapping tasks. Hint: this is essentially Example 2.4.

15. Prove that the FK algorithm (See Exercise 4 of "Greedy Algorithms Overview" Lecture) always returns a maximum container profit.

16. Prove that the Unit Task Scheduling greedy algorithm (See Exercise 7 of "Greedy Algorithms Overview" Lecture) always attains the maximum profit. Hint: use the Replacement Method.

Exercise Solutions

1. 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.

2. 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.

3. 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

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

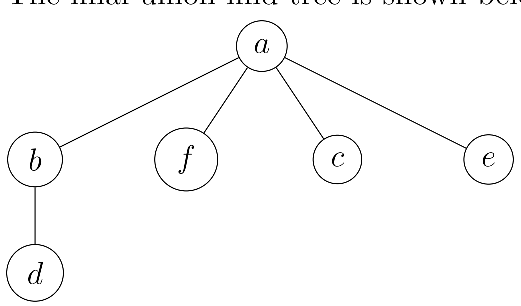
   This theorem implies G must have a degree-1 vertex u. Otherwise,

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

   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.

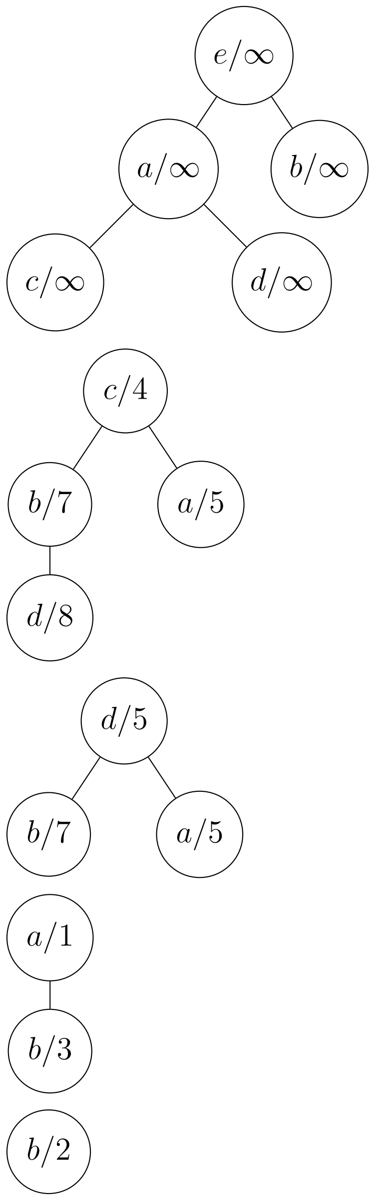
4. Edges added: $(a, d, 1), (a, b, 2), (c, e, 4), (a, e, 5)$ for a total cost of 12.

5. The final union-find tree is shown below.



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

7. 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.



8. 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.

9. 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.

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

11. ```
void print_optimal_path(int i, int parent[ ])
{
     if(i == 0)
          print 0

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

12. 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 $T_{\text{opt}}$ be an mst for $G$ that contains edges $e_1, \ldots, e_{k-1}$, but does not contain $e_k$, for some $1 \le k \le n-1$. We show how to transform$T_{\text{opt}}$ into an mst $T_{\text{opt2}}$ 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 $T_{\text{opt}}$ 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, $w_{e_k} \le w_e$.

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

13. 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 $S_{\text{opt}}$ be an optimal set of stations. Let $s_k$ be the first station of $S$ that is not in $S_{\text{opt}}$. In other words, $S_{\text{opt}}$ 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 $S_{\text{opt2}}$ contains the optimal number of stations. Moreover, notice that, when re-fueling at $s_k$ instead of $s$, the next station in $S_{\text{opt}}$ (and hence in $S_{\text{opt2}}$) can be reached from $s_k$, since $s_k$ is closer to this station than $s$. Thus, $S_{\text{opt2}}$ 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.

14. 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 $t_k \notin T_{\text{opt}}$. 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, TSA would have selected t instead of t_1 since $f(t) < f(t_1)$. In the latter case, TSA 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.

15. 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.

16. 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.