Recitation 4 Guide: Greedy Algorithms

Fall 2022

The main points:

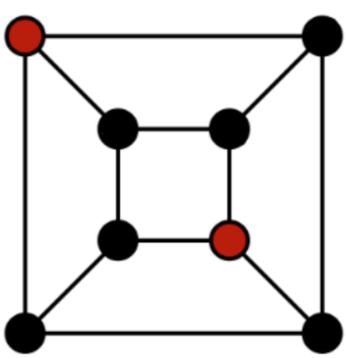
1. All maximUM solutions are maximAL, but there are maximAL solutions which are not maximUM.
2. Greedy algorithms will always find solutions which are maximAL. They may not be maximUM, though. We've seen examples where greedy algorithms fail to find maximUM solutions.
3. To prove that a greedy algorithm is correct, you are proving that the maximAL solution that the algorithm finds is necessarily a maximUM solution. Exchange arguments are arguments which prove that maximAL = maximUM, for that particular algorithm.

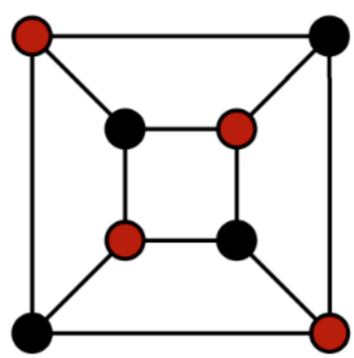
0 Review of Last Week: When Greedy Algorithms Don't Work

Last week, we saw problems where greedy algorithms don't always work:

1. Maximum Independent Set
2. Minimum Graph Coloring

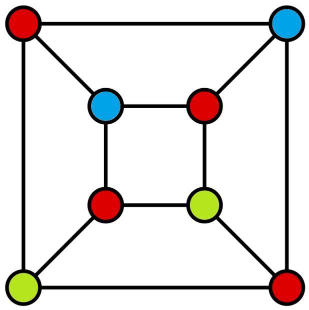
Greedy algorithms don't work for Independent Set and Graph Coloring because these problems have a major obstacle: they can have valid solutions which are optimAL (i.e. maximal or minimal), but not optimUM (maximum or minimum). For example:

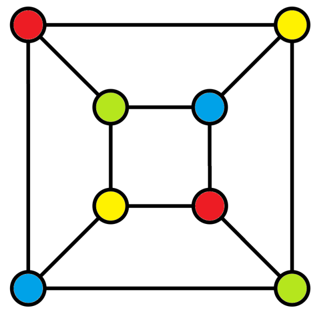


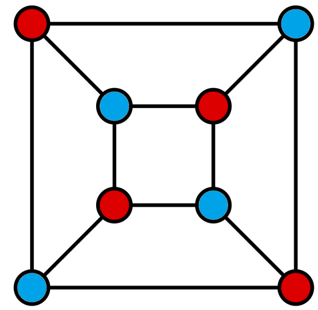


On the left cube, an independent set of size 2 is maximAL. We cannot add any more vertices to make it a larger independent set, even though the cube has a maximUM independent set of size 4 (on the right).

It is possible that an algorithm which greedily builds up an independent set could find the one on the left, and get stuck. The graph has larger independent sets, but they cannot be built out of the one on the left.







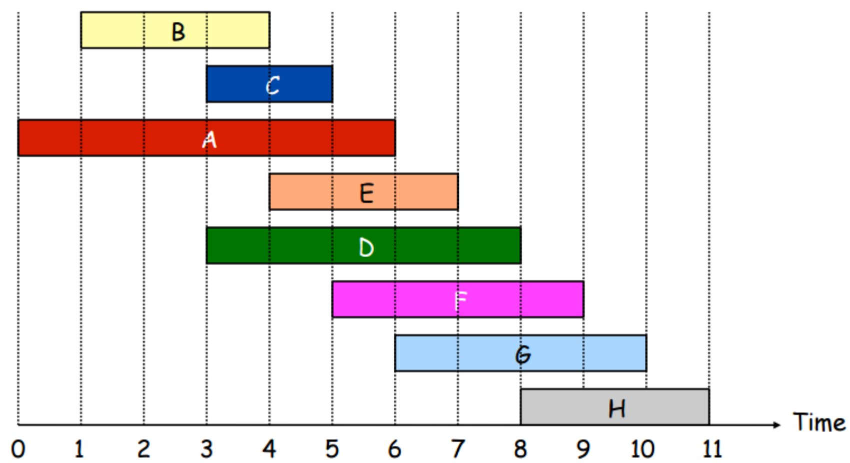
On the left cube, a 3-coloring which is not minimal. This kind of coloring would not be found by a greedy algorithm. Green can be eliminated and recolored with blue to create a 2-coloring, as on the right.

In the middle, a 4-coloring is minimAL. We cannot remove any colors from it to make a 3-coloring, even though the cube has a minimUM coloring with only 2 colors (on the right).

Greedy algorithms find solutions which are maximAL/minimAL. They do not always necessarily find the maximUM/minimUM solution.

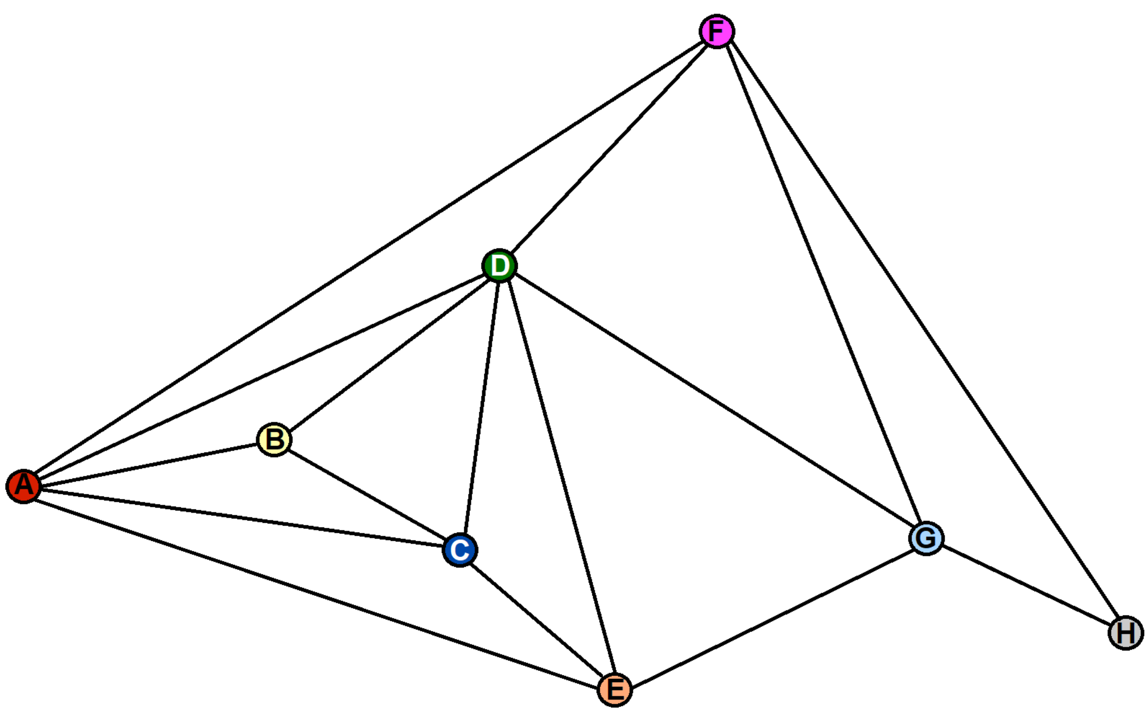
1 Interval Scheduling

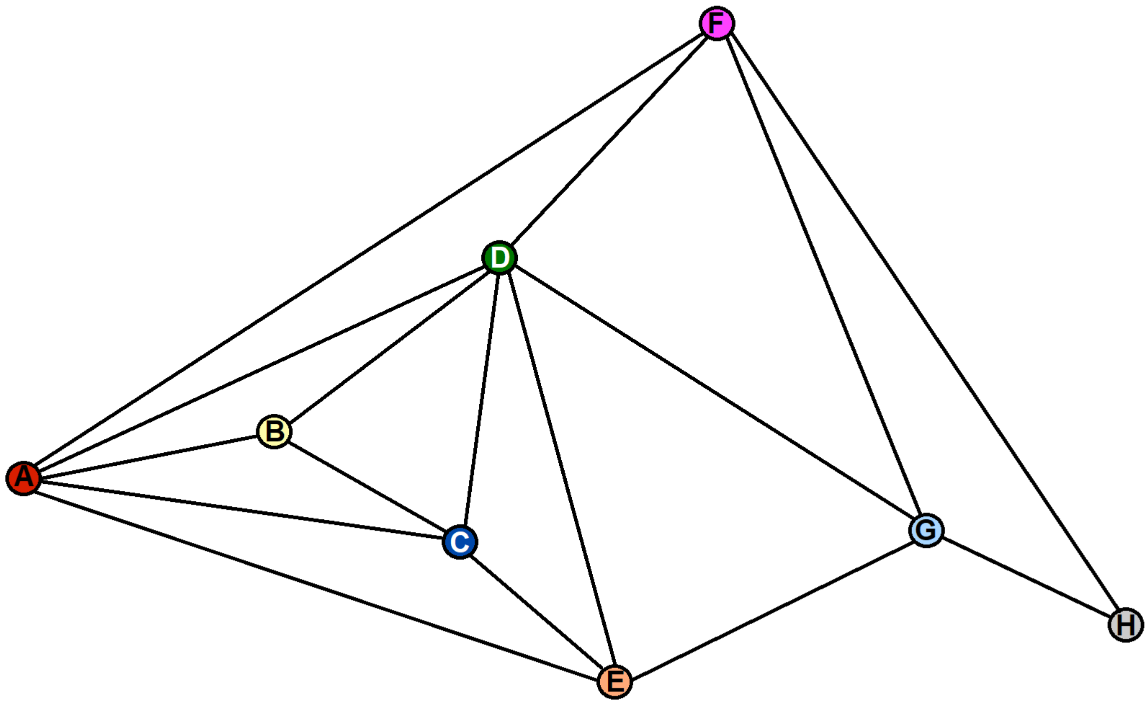
You all saw on HW 2 that the Interval Scheduling problem does not succumb to ALL greedy algorithms, only very specific ones. This is because there are sets of non-overlapping intervals which are maximAL but not maximUM. For example:



In this instance, the set $\{C, F\}$ is maximAL in the sense that it cannot be expanded, even though the set $\{B, E, H\}$ contains more intervals and is maximUM.

Interval Scheduling is actually a special case of the Maximum Independent Set problem. Specifically, we can make a graph whose nodes are the intervals, and which has an edge between two nodes if the corresponding intervals overlap. **An Independent Set in the graph below corresponds to a non-overlapping set of intervals in the problem above.**

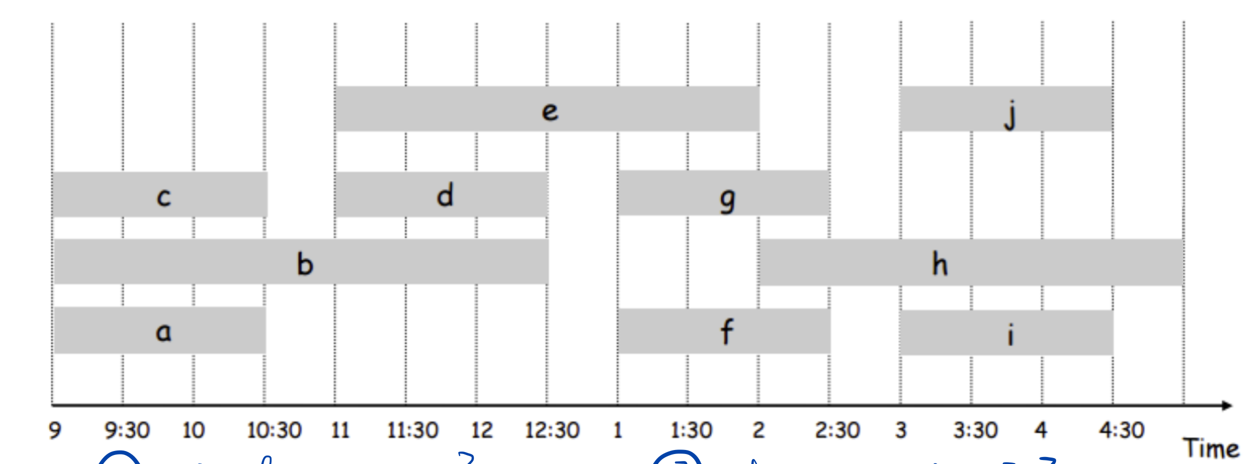




The problem of finding a minimum number of machines to do all of the jobs, assuming that each machine can do one job at a time, is actually the Minimum Graph Coloring problem. **Each color in a graph coloring would correspond to a single machine's job assignments.**

1.1 Exercise:

Consider the interval scheduling problem below. We will consider a greedy algorithm for interval scheduling which assigns priority to intervals based on latest end time.



① { h, e, a }   ② { a, d, f, i } is maximum.

1. Write down the maximal set of intervals that the latest-end-time greedy algorithm will find.
2. Why isn't the solution maximUM? Find a maximum solution which is contains strictly more intervals than the maximal solution you just found.
3. Draw the graph corresponding to the set of intervals given. Find the independent set which corresponds to your maximal solution, and the independent set which corresponds to your maximum solution.
4. How many colors do you need to color this graph? In other words, how many machines do you need to do all of the jobs? A 4-coloring should be obvious from the interval diagram. Can you find a 3-coloring? The maximum set always cover the interval
5. Discuss why the maximal solution found by the earliest-end-time greedy algorithm is always maximum, but the maximal solution found by the latest-end-time greedy algorithm isn't.

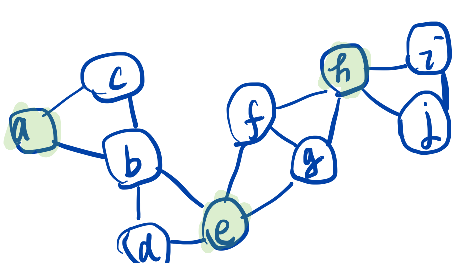
The proof of correctness of a greedy algorithm is a proof that the maximAL solution that the algorithm finds is always maximUM. This is what exchange arguments do: they prove that maximAL = maximUM.

(respectively minimAL and minimUM for minimization problems)

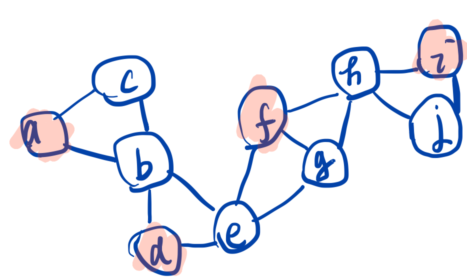
④

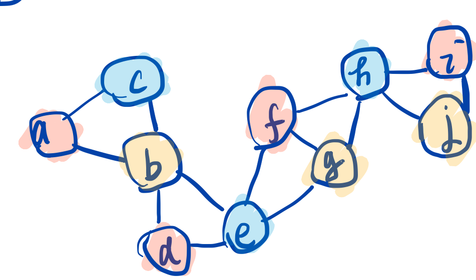
③

the maximal solution



the maximum





3-coloring is enough.

we can't add any more nodes to the set.

5

The interval with the earliest end time has the smallest probability of overlapping with others. By picking the one, we minimize the probability of hitting others. In other words, by picking it, we maximize the available time slots in the future and so can have more spaces to cover more intervals. Moreover, we eliminate the ones that overlapped with it after picking it. If we pick others, the available spaces must be smaller than or equal to the case in which we pick the one with the earliest end time. Then, in future iterations, we pick the one with the earliest end time from the remaining choices and eliminate the ones that overlapped with it. By doing so, we pick the one which maximizes the available time in each iteration

2 Exchange Arguments: Independent Set on Trees

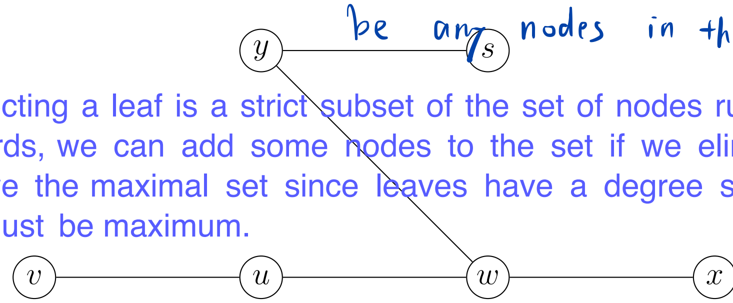
This week, in the homework, you made an exchange argument for Vertex Covers on trees. Now, we will explore an exchange argument for the dual problem, Independent Sets on trees.

1. Let $T = \big(V(T), E(T)\big)$ be the graph

① $\{u, x, s\}$ v has only one neighbor u. If we remove u, then there won't be any nodes in the set, connecting to v.

④

The nodes ruled out by selecting a leaf is a strict subset of the set of nodes ruled out by selecting the parent of a leaf. In other words, we can add some nodes to the set if we eliminate leaves, we can't make the set larger than we the maximal set since leaves have a degree smaller than or equal to one. Hence, the maximal must be maximum.



   Find an independent set $S \subseteq V(T)$ that includes the vertex u. Now, explain why the set $S' = (S \setminus \{u\}) \cup \{v\}$ obtained from your S by removing u and adding v is also an independent set of T.

2. Prove this property in general. That is, let T be any tree, suppose that S is an independent set of T, and suppose that S contains the parent u of a leaf vertex v. Let $S' = (S \setminus \{u\}) \cup \{v\}$ be the set of vertices obtained from S by removing u and adding v. Carefully explain why S' is an independent set of T. ② ?

3. With the above fact in mind, come up with a greedy algorithm which finds a maximal independent set in a given tree T. Hint: it should involve finding the leaves of T, adding them to S, and then reducing the size of T.

4. Use the exchange argument above to prove that any maximal independent set which is found by your algorithm must be a maximUM independent set.

③

In each iteration, find all leaves and add them to the set. In addition, eliminate them and the ones connecting to them from the graph. The algorithm stops when there are no nodes in the graph.

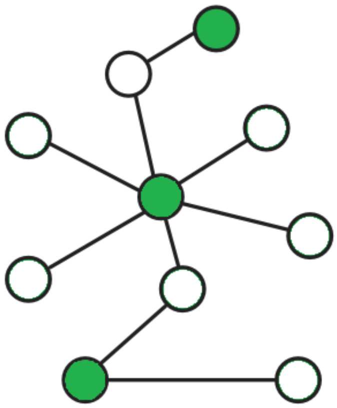
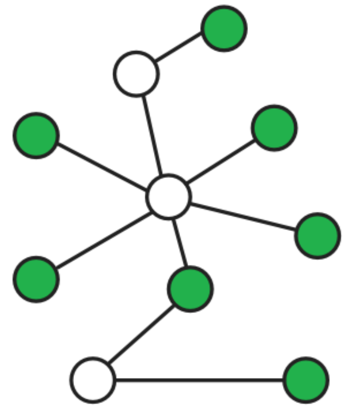
2.1 Duality

Remark 2.1. You may notice that the above problem has a striking symmetry with problem 6 on your last problem set. This is not a coincidence. Independent Set is the "exact opposite" problem of Vertex Cover, in some sense. The following theorem makes that sense precise.

Theorem 2.2. *Suppose $G = (V, E)$ is a graph. Let $S \subseteq V$. Then S is an independent set if and only if $V \setminus S$ is a vertex cover.*

In other words, any time you have a vertex cover, all of the rest of the vertices form an independent set. And any time you find an independent set, all the rest of the vertices form a vertex cover!

Take some time to intuitively verify this for yourself with the examples below:

Pictured above in green are two independent sets. For each independent set, the remaining vertices (in white) form a vertex cover.

The following pages are referenced from Michael Levet's algorithm note page 43, 45-47, 51-54.

Definition 58. Let $G(V, E, w)$ be a weighted graph, and let $\mathcal{F}$ be an intermediate spanning forest of G. Let $e = \{u, v\}$ be an edge of G such that $e \notin \mathcal{F}$.

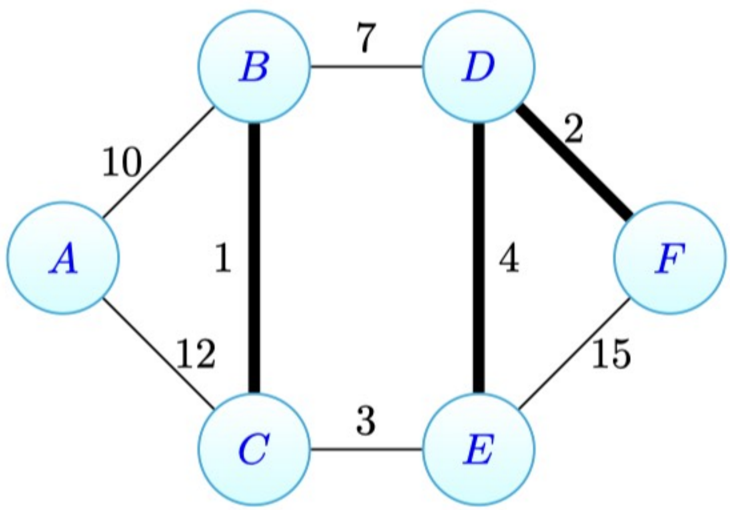
(a) We say that e is *safe* with respect to $\mathcal{F}$ if $\mathcal{F} \cup e$ is a subgraph of some minimum-weight spanning tree of G.

(b) Let $S \subseteq V(G)$ be a set of vertices such that every edge $\{x, y\}$ in $\mathcal{F}$ has either $x, y \in S$ or $x, y \in V(G) \setminus S$. We say that e is a *light* edge e is a minimum weight edge with one endpoint in S and the other endpoint in $V(G) \setminus S$.

(c) We say that e is *useless* with respect to $\mathcal{F}$ if both u and v lie on the same tree in $\mathcal{F}$. Note that in this case, adding e to $\mathcal{F}$ creates a cycle.

(d) We say that e is *undecided* with respect to $\mathcal{F}$ if e is neither safe nor useless.

Corollary 61. Let $G(V, E, w)$ be a weighted graph, and let $\mathcal{F}$ be an intermediary spanning forest. Fix a tree T_i, and tet $e \in E(G)$ be a light edge with exactly one endpoint in T_i. Then e is safe.



We have the following.

- $\{A, B\}$ is the minimum-weight edge incident to $\{A\}$. Therefore, as $\{A, B\}$ is a light edge with exactly one endpoint belonging to $\{A\}$, we have by Corollary 61 that $\{A, B\}$ is **safe** with respect to $\mathcal{F}$.

- $\{C, E\}$ is the minimum-weight edge with exactly one endpoint in the component $\{B, C\}$ (as well as the minimum-weight edge with exactly one endpoint in the component $\{D, E, F\}$). Therefore, we have by Corollary 61 that $\{C, E\}$ is **safe** with respect to $\mathcal{F}$.

- While the edge $\{A, C\}$ connects the components $\{A\}$ and $\{B, C\}$, $\{A, C\}$ is not a minimum-weight edge doing so. Therefore, $\{A, C\}$ is **undecided** with respect to $\mathcal{F}$.

- While the edge $\{B, D\}$ connects the components $\{B, C\}$ and $\{D, E, F\}$, $\{B, D\}$ is not a minimum-weight edge doing so. Therefore, $\{B, D\}$ is **undecided** with respect to $\mathcal{F}$.

- The edge $\{E, F\}$ creates has both endpoints in the component $\{D, E, F\}$. So $\{E, F\}$ is **useless** with respect to $\mathcal{F}$.

Kruskal

4.3 Kruskal's Algorithm

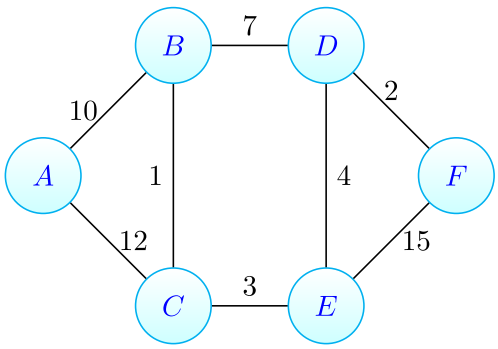
We briefly introduced Kruskal's algorithm in Section 4.1. In this section, we will examine Kruskal's algorithm in more detail. Recall that Kruskal's algorithm places the edges of the input graph into a priority queue. It then polls the edges one at a time, adding the edge e currently being considered to the intermediate spanning forest precisely if e connects two disjoint components. As the edges are sorted from lowest weight to highest weight, it follows that e is added precisely if there exists a component T where e is a light edge with exactly one endpoint in T. So by Corollary 61, e is added precisely if e is safe.

We formalize Kruskal's algorithm below.

Algorithm 6 Kruskal's Algorithm

```
procedure KRUSKAL(ConnectedWeightedGraph G(V, E, w))
    𝓕 ← (V(G), ∅)                          ▷ Initialize the Intermediate Spanning Forest to contain no edges.
    PriorityQueue Q ← []
    Q.addAll(E(G))
    while 𝓕.numEdges() < |V(G)| − 1 do
        {u, v} ← Q.poll()                  ▷ Poll an edge and call the endpoints u and v
        if u and v are on different components of 𝓕 then
            𝓕.addEdge({u, v})
    return 𝓕
```

Example 64. We now work through Kruskal's algorithm on the following graph.

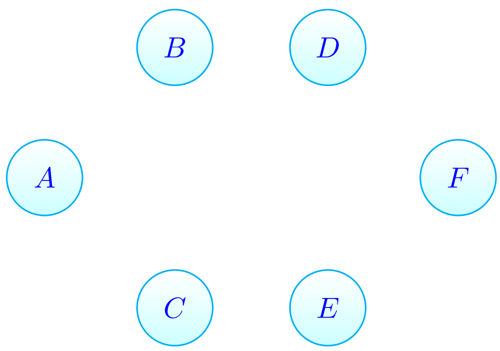


We proceed as follows.

1. We initialize the intermediate spanning forest $\mathcal{F}$ to be the empty graph (the graph on no edges). We also place the edges of G into a priority queue, which we call Q. So:

   $$Q = [(\{B, C\}, 1), (\{D, F\}, 2), (\{C, E\}, 3), (\{D, E\}, 4), (\{B, D\}, 7), (\{B, A\}, 10), (\{A, C\}, 12), (\{E, F\}, 15)].$$

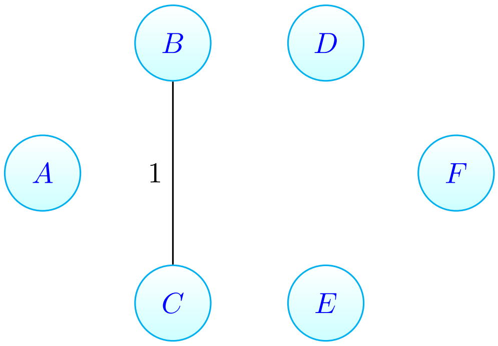
   Here, $(\{B, C\}, 1)$ indicates the edge $\{B, C\}$ has weight 1. The intermediate spanning forest $\mathcal{F}$ is pictured below.



2. We poll from Q, which returns the edge $\{B, C\}$. Note that $w(\{B, C\}) = 1$. As B and C are on different components of $\mathcal{F}$, we add the edge $\{B, C\}$ to $\mathcal{F}$. So:

   $$Q = [(\{D, F\}, 2), (\{C, E\}, 3), (\{D, E\}, 4), (\{B, D\}, 7), (\{B, A\}, 10), (\{A, C\}, 12), (\{E, F\}, 15)],$$

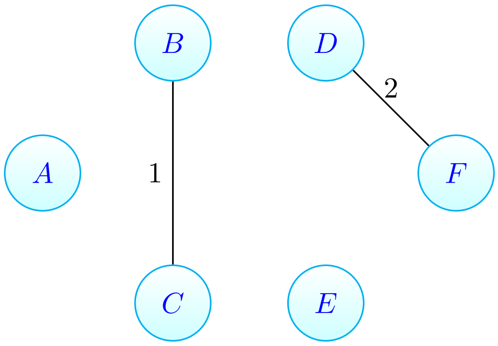
   and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



3. We poll from Q, which returns the edge $\{D, F\}$. Note that $w(\{D, F\}) = 2$. As B and C are on different components of $\mathcal{F}$, we add the edge $\{D, F\}$ to $\mathcal{F}$. So:

$$Q = [(\{C, E\}, 3), (\{D, E\}, 4), (\{B, D\}, 7), (\{B, A\}, 10), (\{A, C\}, 12), (\{E, F\}, 15)],$$

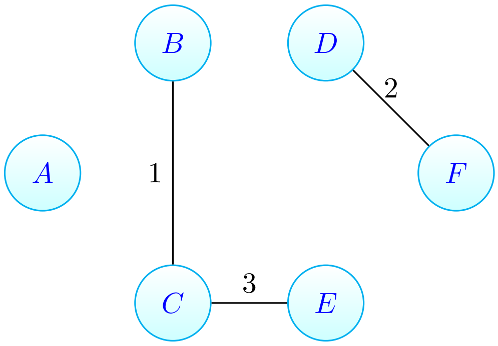
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



4. We poll from Q, which returns the edge $\{C, E\}$. Note that $w(\{C, E\}) = 3$. As C and E are on different components of $\mathcal{F}$, we add the edge $\{C, E\}$ to $\mathcal{F}$. So:

$$Q = [(\{D, E\}, 4), (\{B, D\}, 7), (\{B, A\}, 10), (\{A, C\}, 12), (\{E, F\}, 15)],$$

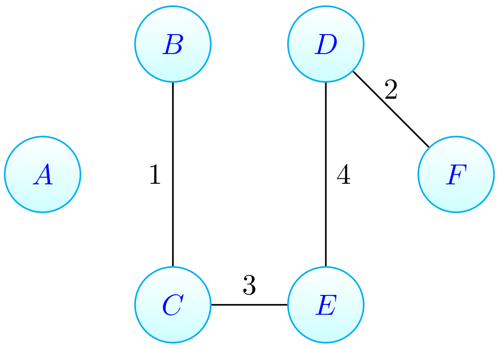
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



5. We poll from Q, which returns the edge $\{D, E\}$. Note that $w(\{D, E\}) = 4$. As D and E are on different components of $\mathcal{F}$, we add the edge $\{D, E\}$ to $\mathcal{F}$. So:

$$Q = [(\{B, D\}, 7), (\{B, A\}, 10), (\{A, C\}, 12), (\{E, F\}, 15)],$$

and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



6. We poll from Q, which returns the edge $\{B, D\}$. Note that $w(\{B, D\}) = 7$. As B and D are on the same component of $\mathcal{F}$, we do **not** add the edge $\{B, D\}$ to $\mathcal{F}$. So:

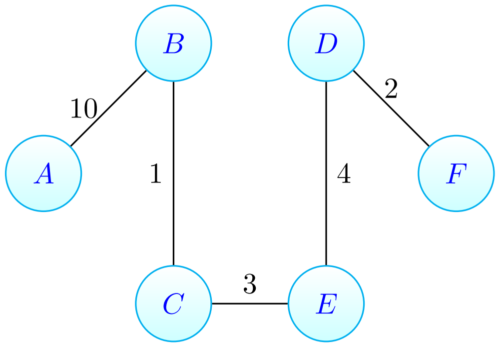
$$Q = [(\{B, A\}, 10), (\{A, C\}, 12), (\{E, F\}, 15)],$$

and the intermediate spanning forest $\mathcal{F}$ remains unchanged from the previous iteration.

7. We poll from Q, which returns the edge $\{B, A\}$. Note that $w(\{B, A\}) = 10$. As B and A are on different components of $\mathcal{F}$, we add the edge $\{B, A\}$ to $\mathcal{F}$. So:

$$Q = [(\{A, C\}, 12), (\{E, F\}, 15)],$$

and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



As there are 6 vertices and $\mathcal{F}$ has 5 edges, Kruskal's algorithm terminates and returns $\mathcal{F}$, which is our minimum-weight spanning tree.

Remark 65. Now that we have worked through an example of Kruskal's algorithm, we wish to comment a bit about the algorithm provided (Algorithm 6). On line 7 of Algorithm 6, there is an `if` statement checking whether two vertices belong to the same connected component. For the purposes of this class, we will not examine the details associated with implementing this functionality. In practice, a Union-Find data structure is used to manage the intermediate spanning forest. We direct the reader to [CLRS09, Chapter 21] for details regarding the Union-Find data structure.

Prim

4.4 Prim's Algorithm

In this section, we examine a second technique to construct minimum-weight spanning trees; namely, Prim's algorithm. We again start with the intermediate spanning forest $\mathcal{F}$ that contains all the vertices of our input graph $G(V, E, w)$, but none of the edges. While Kruskal's algorithm determines which edges to add to $\mathcal{F}$ by examining the entire graph, Prim's algorithm takes a more local perspective. We provide as input a specified source vertex $s \in V(G)$. Let T^* be the component of $\mathcal{F}$ that contains s. Prim's algorithm examines the edges of G that have exactly one endpoint in T^* and select a light edge e from these to add to $\mathcal{F}$. As e has exactly one endpoint in T^*, e connects two distinct components of $\mathcal{F}$. So by Corollary 61, e is a safe edge with respect to $\mathcal{F}$. This is the key observation in establishing that Prim's algorithm returns a minimum-weight spanning tree.

We now turn to formalizing Prim's algorithm.

Algorithm 7 Prim's Algorithm

```
1: procedure Prim(ConnectedWeightedGraph G(V, E, w), Vertex source)
2:     F ← (V(G), ∅)                     ▷ Initialize the Intermediate Spanning Forest to contain no edges.
3:     PriorityQueue Q ← []

4:     for each edge e ∈ E(G) do
5:         e.processed ← false

6:     for each edge e incident to source do
7:         Q.add(e)
8:         e.processed ← true

9:     while F.numEdges() < |V(G)| − 1 do
10:        {u, v} ← Q.poll()                          ▷ Poll an edge and call the endpoints u and v
11:        T_u ← F.componentContaining(u)
12:        T_v ← F.componentContaining(v)

13:        if T_u ≠ T_v then                  ▷ Check that u and v belong to different components
14:            F.addEdge({u, v})

15:            if source ∈ T_u then           ▷ If v was added to the component containing source
16:                for each unprocessed edge e incident to v do
17:                    Q.add(e)               ▷ Then add to Q each unprocessed edge incident to v

18:            else                           ▷ If u was added to the component containing source
19:                for each unprocessed edge e incident to u do
20:                    Q.add(e)               ▷ Then add to Q each unprocessed edge incident to u

       return F
```

We associate to each edge an attribute `processed` to indicate whether that edge has been placed into the priority queue. This ensures that each edge is considered at most once, which helps ensure that the algorithm will terminate.

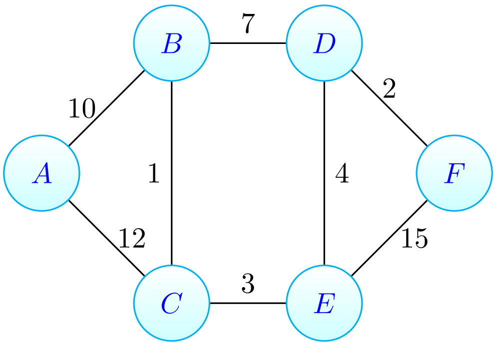
Now at lines 6-8, we initialize the priority queue to contain only edges that are incident to the source vertex. This ensures that the first edge placed into the intermediate spanning forest is incident to the source vertex. Now by adding an edge to $\mathcal{F}$, we introduce a new vertex v to the component containing our source vertex. Prim's algorithm then adds to the prioritiy queue the edges incident to v, provided such edges have not already been polled from the queue. So the `while` loop at line 9 preserves the invariant that every edge in the priority queue has at least one endpoint in the component containing our source vertex.

Prim's algorithm only adds an edge if it connects two components. Such an edge e is polled from the priority queue, and so (i) has an endpoint in the component containing the source vertex, and (ii) is a minimum-weight edge connecting two distinct components. Therefore, e is a safe edge.

4.4.1 Prim's Algorithm: Example 1

We now work through an example of Prim's algorithm.

Example 71. Consider the following graph $G(V, E, w)$ pictured below. Suppose we select the source vertex A.

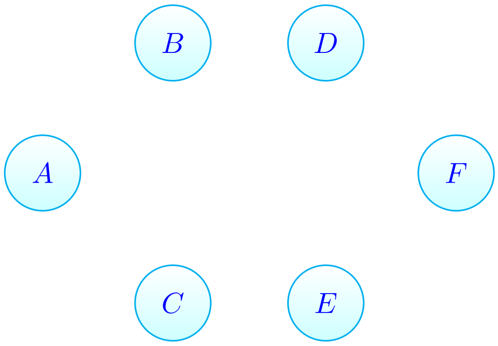


Prim's algorithm proceeds as follows.

1. We initialize the intermediate spanning forest to contain all the vertices of G, but no edges. We then initialize the priority queue to contain the edges incident to our source vertex A. So:

$$Q = [(\{A, B\}, 10), (\{A, C\}, 12)],$$

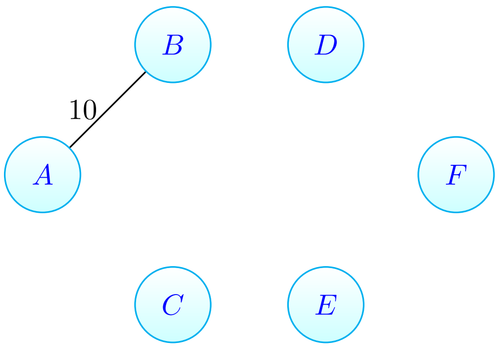
and our intermediate spanning forest $\mathcal{F}$ is pictured below.



2. We poll the edge $\{A, B\}$ from the queue and mark $\{A, B\}$ as processed. Note that $w(\{A, B\}) = 10$. As $\{A, B\}$ has exactly one endpoint on the component containing A (which is the isolated vertex A), we add $\{A, B\}$ to $\mathcal{F}$. We then push into the priority queue the unprocessed edges incident to B. So:

$$Q = [(\{B, C\}, 1), (\{B, D\}, 7), (\{A, C\}, 12)],$$

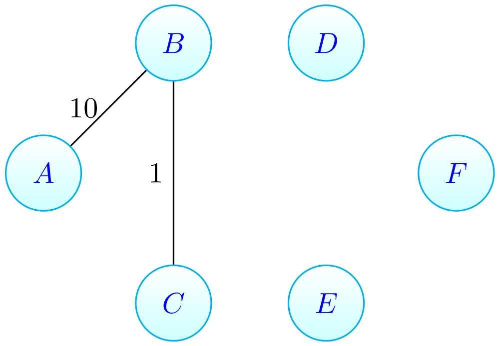
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



3. We poll the edge $\{B, C\}$ from the queue and mark $\{B, C\}$ as processed. Note that $w(\{B, C\}) = 1$. As $\{B, C\}$ has exactly one endpoint on the component containing A (which is the isolated vertex $\{A, B\}$), we add $\{B, C\}$ to $\mathcal{F}$. We then push into the priority queue the unprocessed edges incident to C (provided said edges are not already in the priority queue). So:

$$Q = [(\{C, E\}, 3), (\{B, D\}, 7), (\{A, C\}, 12)],$$

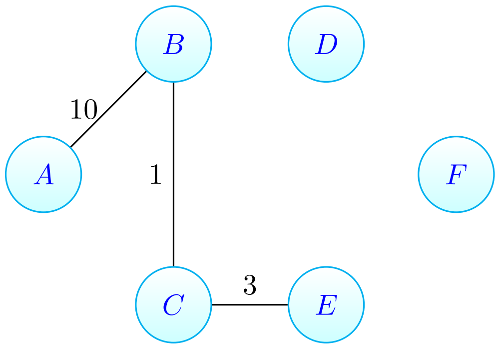
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



4. We poll the edge $\{C, E\}$ from the queue and mark $\{C, E\}$ as processed. Note that $w(\{C, E\}) = 3$. As $\{C, E\}$ has exactly one endpoint on the component containing A, we add $\{C, E\}$ to $\mathcal{F}$. We then push into the priority queue the unprocessed edges incident to E (provided said edges are not already in the priority queue). So:

$$Q = [(\{E, D\}, 4), (\{B, D\}, 7), (\{A, C\}, 12), (\{E, F\}, 15)],$$

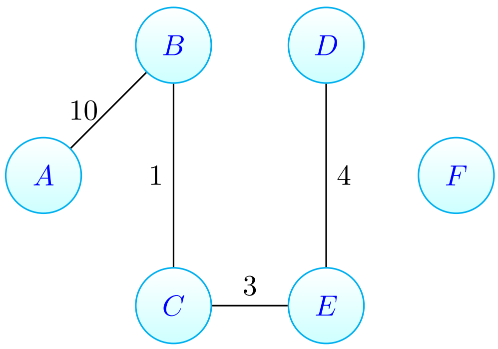
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



5. We poll the edge $\{E, D\}$ from the queue and mark $\{E, D\}$ as processed. Note that $w(\{E, D\}) = 4$. As $\{E, D\}$ has exactly one endpoint on the component containing A, we add $\{E, D\}$ to $\mathcal{F}$. We then push into the priority queue the unprocessed edges incident to D (provided said edges are not already in the priority queue). So:

$$Q = [(\{D, F\}, 2), (\{B, D\}, 7), (\{A, C\}, 12), (\{E, F\}, 15)],$$

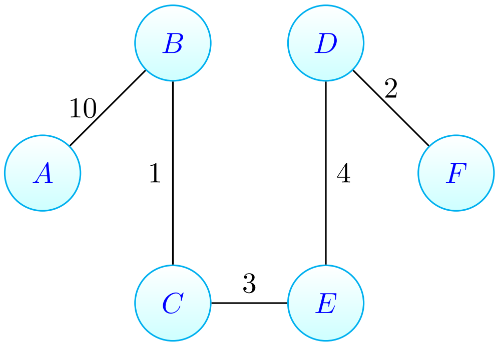
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



6. We poll the edge $\{D, F\}$ from the queue and mark $\{D, F\}$ as processed. Note that $w(\{D, F\}) = 2$. As $\{D, F\}$ has exactly one endpoint on the component containing A, we add $\{D, F\}$ to $\mathcal{F}$. We then push into the priority queue the unprocessed edges incident to D (provided said edges are not already in the priority queue). So:

$$Q = [(\{B, D\}, 7), (\{A, C\}, 12), (\{E, F\}, 15)],$$

and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



7. As $\mathcal{F}$ has $|V(G)| - 1 = 6 - 1 = 5$ edges, the algorithm terminates and returns $\mathcal{F}$, pictured in Step 6 immediately above.

Remark 72. We note that the minimum-weight spanning tree constructed by Kruskal's algorithm in Example 64 is the same tree that Prim's algorithm constructed in Example 71. For this input graph, the edge weights were distinct. Therefore, the graph had only one minimum-weight spanning tree. In general, Prim's algorithm and Kruskal's do not construct the same minimum-weight spanning tree.

The coming pages are refered from Introduction to Algorithm in page 431, 432.

Huffman

codes.

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n − 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)    // return the root of the tree
```