9 Graphs

9.1 Trees

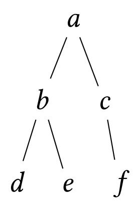
Frequently the relations we draw between objects are hierarchical in nature. That is the objects have a parent-to-child relationship, for example:

- A literal parent and their children.
- A manager and the employees that report to them.
- A folder and the files it contains.

We represent these relationships with a specialized kind of graph called a *tree*.

Definition 1 (Tree). *A tree is an undirected graph that contains no cycles.*

Here is an artficial example of a tree with five nodes, a–f:



We distinguish a vertex of the tree as its *root*. Here we'll consider a to be the root of the tree although any of the vertices could be considered the root. By convention, we draw trees "upside down" with the root at the top and the tree growing downwards.

The root allows us to categorize the vertices of the tree by their distance from the root. We call a collection of vertices that are the same distance away from the root a *level*.

Definition 2 (Level (Tree)). *Let $T = (V, E)$ be a tree with a distinguished root r. Define the ith level of a tree, denoted L_i to be the set of vertices that are i away from r:*

$$L_i = \{ v \mid v \in V \text{ and there exists a path of length } i \text{ from } r \text{ to } v \}.$$

Definition 3 (Height). *The* height *of at tree is the maximum level of any of its vertices.*

In our above example:

$$\begin{aligned} L_0 &= \{ a \} \\ L_1 &= \{ b, c \} \\ L_2 &= \{ d, e, f \} \end{aligned}$$

and the tree has height 2. Note that L_k for any $k > 2$ is $\emptyset$ since there are no nodes greater than 2 away from A.

With levels defined, we can now formally define the parent-child relationship that we characterizes trees:

Definition 4 (Parent). *The* parent *of a vertex v at level i of a tree is the node u for which the edge (u, v) is in the tree and u is at level $i - 1$.*

Definition 5 (Children). *The* children *of a vertex u at level i of a tree are the nodes V for which each $v \in V$, the edge (u, v) is in the tree and v is at level $i + 1$.*

Because a tree contains no cycles and the tree is rooted at a particular node, it follows that every vertex of a tree except the root has exactly one parent. (This is a worthwhile claim to prove yourself for practice!)

We can categorize trees by the maximal number of children any one node possesses. We call this value the tree's *fan-out*:

Definition 6 (Fan-Out). *The* fan-out *of a tree k is the maximum number of children that any one vertex of the tree possesses. We call such a tree a k-ary tree or a k-tree for short. Notably a* 1*-tree is a* sequence *or a* list*, and a* 2*-tere is a* binary tree*.*

Finally, we've restricted ourselves to *connected* trees, trees in which all its vertices are mutually reachable. If a graph is unconnected, but all of its connected components themselves are trees, then we call the graph a *forest*:

Definition 7 (Connected Components). *Call an undirected graph $G = (V, E)$* connected *if there exists a path between every pair of vertices in G. A* connected component *G' of G is a sub-graph of G that is, itself, connected.*

Definition 8 (Forest). *A graph is a* forest *if all of its connected components are trees.*

Directed Acyclic Graphs We generally assume that a tree is an undirected graph. However, we can apply the same concepts to a directed graph. This results in a kind of graph called a *directed acyclic graph* (DAG):

Definition 9. *A* directed acyclic graph *(DAG) is a directed graph that contains no cycles.*

While we do not have time to discuss DAGs in sufficient depth, be aware that DAGs have their own interesting properties and operations distinct from trees.

9.1.1 Depth-First Tree Traversals

Previously, we learned about depth-first and breadth-first traversals for graphs. Breadth-first traversals remain the same: we traverse the vertices of the tree by order of increasing level. However, the specialized nature of trees lets us specify different sorts of depth-first traversals, in particular, for *binary trees* where every node possesses at most two children In such a tree, we call one child the *left child* and the other child the *right child*.

With this in mind, we can describe a recursive algorithm for depth-first search specialized to binary trees. In the code below, we use dot-notation to denote children, *e.g.*, `v.l` for `v`'s left child and `v.r` for `v`'s right child.

```
def preDFS(u):
    visit(u)
    preDFS(u.l)
    preDFS(u.r)
```

Note that we visit the node u before visiting its children. Performing this traversal on our example tree from the beginning of this section yields the sequence:

$$a, b, d, e, c, f.$$

This kind of depth-first traversal of the tree is called a *pre-order* traversal of the tree. We first visit the current element and then visit its children.

In contrast, a *post-order* traversal of the tree visits the children first and then the current node last.

```
def postDFS(u):
    postDFS(u.l)
    postDFS(u.r)
    visit(u)
```

A post-order traversal of the graph yields the following sequence:

$$d, e, b, f, c, a.$$

Finally, we can intermix visiting children and visiting the current node with an *in-order* traversal:

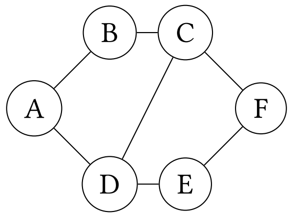
```
def inDFS(u):
    inDFS(u.l)
    inDFS(u.r)
    visit(u)
```

An in-order traversal yields the following sequence:

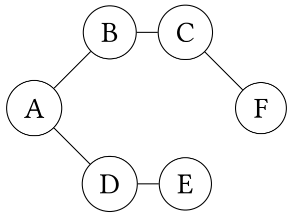
$$d, b, e, a, c, f.$$

9.2 Spanning Trees

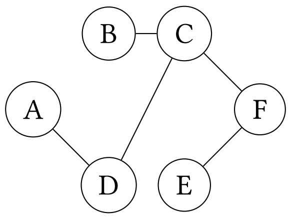
Consider a graph $G = (V, E)$. A *spanning tree* of G is a subgraph $G' = (V', E')$ (with $V' \subseteq V$ and $E' \subseteq E$) that is a tree (*i.e.*, contains no cycles) that covers every vertex of G. That is $V' = V$. For example, consider the following graph:



A spanning tree for the graph is given below:



Spanning trees are not necessarily unique. For example, here is a different spanning tree for the graph.



Constructing a spanning tree for a graph is useful in many situations, for example:

- Consider a electrical network for a neighborhood where nodes represents houses and edges represent potential electrical connections between houses. A spanning tree in this context represents a minimal set of electrical connections used to connect all of the houses to the power grid.
- Consider a collection of networked computers where nodes represent computers and edges represent physical connections between computers. A spanning tree in this context represents a minimal collection of connections that allow one machine to communicate with another without the fear of encountering a *routing loop* where a message is relayed between a collection of machines in perpetuity.

Exercise (Sizes of Spanning Trees) Say that we have a graph $G = (V, E)$. What is the number of edges of any spanning tree of G? Prove your claim.

9.2.1 Constructing Spanning Trees

To construct a spanning tree, we can employ either traversal algorithm we have discussed for graphs—*breadth-first* or *depth-first* search—to reach every vertex from some arbitrary starting vertex. When processing a vertex v, we add to our working set each vertex v' connected to v (*i.e.*, $(v, v') \in E$) that we haven't already seen. If we are performing breadth-first search, we treat the working set like a queue, processing the oldest entry in the working set first. If we are performing depth-first search, we treat the working set like a stack, processing the newest entry in the working set first.

For example, the first spanning tree we discussed previously is the result of a breadth-first traversal, namely:

$$A, B, D, C, E, F.$$

The second spanning tree is the result of a depth-first traversal, namely:

$$A, D, C, B, F, E.$$

Exercise (Runtime of Spanning Tree Construction) Prove that for a graph $G = (V, E)$, we must process exactly $V - 1$ vertices when constructing the spanning tree, irrespective of the traversal used to generate the tree.

9.3 Minimum Spanning Trees

Now, let's consider extending our graph edges with *weights*. Each weight represents a "cost" associated with that edge, for example:

- Distances if the edges represent connections between physical places.
- Monetary amounts if the edges represent a transformation from one kind of object to another.

Let

$$\mathsf{weight}(E) = \sum_{(v_1, v_2, w) \in E} w$$

be the sum of the weights of all the edges in E.

With a weighted graph, we can refine our notion of spanning tree. We can now consider *minimum spanning trees* (MST), a spanning tree with minimal cost. Formally defined:

Definition 10 (Minimum Spanning Tree). *Let $G = (V, E)$ be a weighted graph. Then a* minimum spanning tree, *$T_1 = (V_1, E_1)$, be a spanning tree of G such that for any other spanning tree $T_2 = (V_2, E_2)$ of G, it is the case that* $\mathsf{weight}(E_1) \leq \mathsf{weight}(E_2)$.

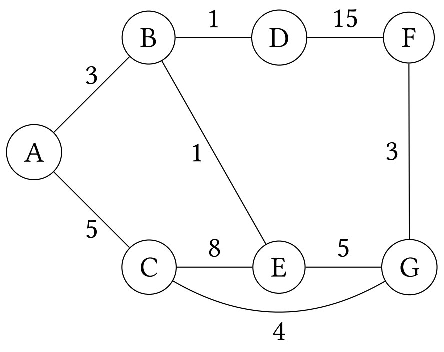
In our above examples, we could attach an appropriate cost to each graph:

- Weights in the electrical network represent physical distance between houses.
- Weights in the computer network represent the average time it takes for two computers to communicate with each other.

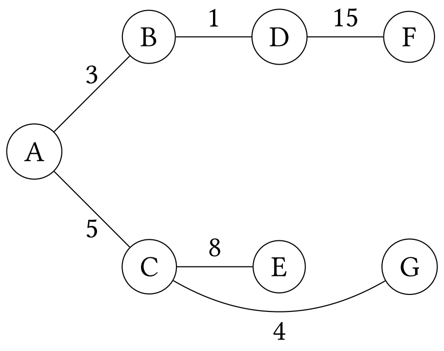
Minimum spanning trees for each of these examples then also minimize the values of these trees—physical distances and average communication time, respectively—in addition to "spanning" the graphs.

9.3.1 Algorithms for Minimum Spanning Trees

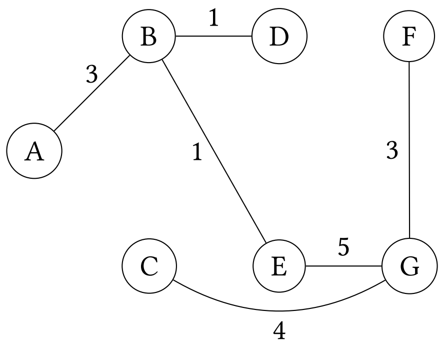
Note that our current methods for constructing minimum spanning trees are agnostic to the weights of the edges. Even though such algorithms choose a minimal number of edges, this doesn't guarantee that the weight of the resulting spanning tree is minimized. For example, consider the following weighted graph:



A BFS traversal of the graph starting at A produces the following weighted graph:

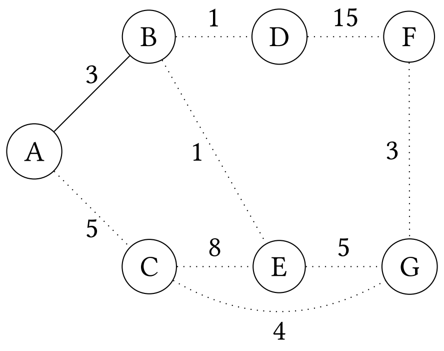


Its weight is $3 + 1 + 15 + 5 + 8 + 4 = 36$ but it is not minimal. The following minimum spanning tree is minimal for our graph:



The weight of this MST is $3 + 1 + 1 + 3 + 5 + 4 = 17$. We clearly need another method of calculating a MST that takes into account (a) the vertices we have yet to explore and (b) the weights of the chosen edges.

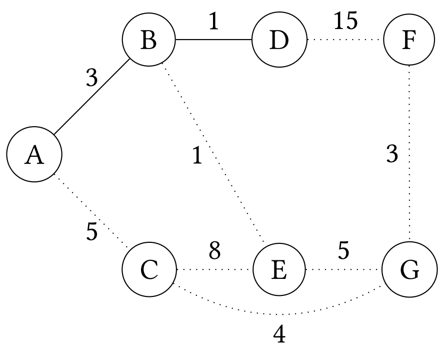
There are several algorithms that we could consider. Here, we will consider *Prim's Algorithm* which we will present as a modification of breadth-first search for this setting. To gain some intuition about how to proceed, let's see how naive breadth-first search failed to produce the MST and then modify the algorithm to obtain the desired result.



Initially, we begin our BFS at A and then add B. We can think of A and B and the edge AB as part of our MST. Our goal is to figure out which edge to add to the MST next. Note that we must pick an edge that does not create a loop in the MST, thus we must only consider edges that connect the MST to a vertex not already in the MST.

At this point, BFS would consider C next. However, we see that the edge AC is not in the MST. This is because it turns out it is more profitable to instead include vertex C by way of edge CG instead. How can we avoid making this choice? Note that we have three edges to choose from at this point—AC, BD, and BE—and the last two edges have a lower weight (1) than AC (5). It seems like we should consider one of these edges first since its weight is smaller.

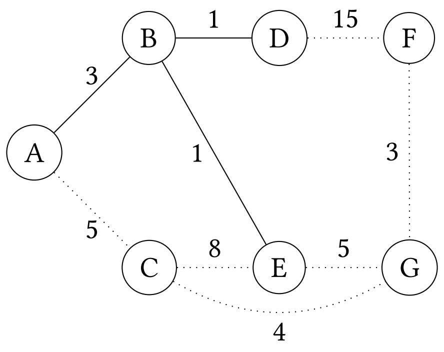
If we choose BD, we obtain the following extended MST that includes A, B, and D:



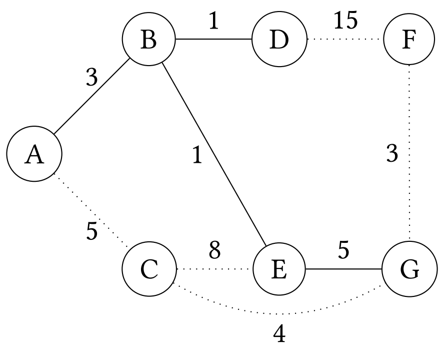
Now we can consider the following edges:

- *AC*—cost 5,
- *BE*—cost 1, and
- *DF*—cost 15.

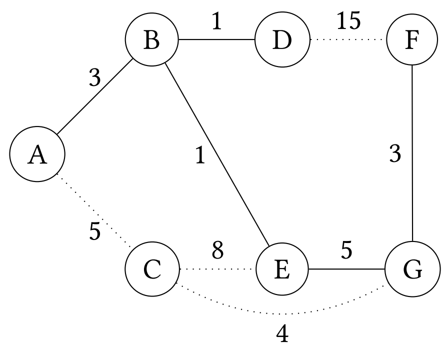
By considering the *lowest weight edge* next, we then add *BE* to the MST:



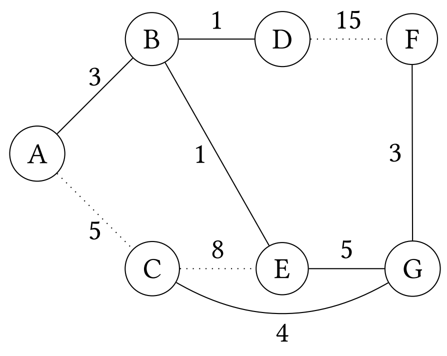
We continue in this manner, considering the lowest weight edge that expands the MST by one vertex. We would next add *EG* (cost 5):



Next we would add *FG* (cost 3):



And then finally *CG* (cost 4):



At this point, we know we are done because there are no other vertices left to add to the graph. Alternatively, we know that any tree of the graph contains $|V| - 1$ edges, so once we add this number of edges, we can safely stop.

To summarize, Prim's algorithm proceeds as follows on a graph $G = (V, E)$.

- Choose a start vertex $s \in V$ as an initial T consisting of just s.
- For $|V| - 1$ times, do:
    - Choose the minimum weight edge $e \in E$ that connects T with a vertex not in T and add e to T.
- Afterwards T is a MST for G.

Prim's algorithm is an example of a *greedy algorithm*. A greedy algorithm is one that, for each iteration, makes its next choice by choosing the minimum or maximum option available. Here, that choice is the minimum weight edge that expands the current MST. However, how do we know this choice is correct?

To prove that our greedy choice is correct, we must first introduce the notion of a *cut*.

Definition 11. *Let $G = (V, E)$ be a graph. A* cut *of the graph is a partition of its vertices $(S, V - S)$ with $S \subseteq V$. A* non-trivial *cut is one where $S \neq \emptyset$ and $S \neq V$.*

Since S in the above definition uniquely identifies the cut, we will refer to the cut by S for notational convenience.

Cuts allow us to talk about the state of Prim's algorithm precisely. At any iteration of the algorithm, we may view the MST $T = (V', E')$ as inducing a cut V' of G. The algorithm then considers the minimum weight edge that flows between vertices of the cut, *i.e.*, vertices of the form u, v with $u \in V'$ and $v \in V - V'$. We must show that this edge belongs to some MST of G. In particular, we'll show that it can belong to the *eventual* MST T^* of the graph grown from the current MST T, that is, $T \subseteq T^*$.

Claim 1 (Cut Property). *Let $G = (V, E)$ be a graph and $T = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$ be a minimum spanning tree for V'. Consider the set of edges D that connect a vertex in T to a vertex not in T,* i.e.,

$$D = \{ (u, v) \mid (u, v) \in E, u \in V', v \in V - V' \},$$

and let e be an edge with minimum weight of D. e belongs to some MST T^* of G where $T \subseteq T^*$.

Proof. We prove this fact by contradiction. Let the cut induced by $T = (V', E')$ be $(V', V - V')$ and the minimum weight edge under consideration be $e = (u, v)$ with $u \in V'$ and $v \in V - V'$. Suppose for the sake of contradiction that e does not belong to any MST of G of which T is a subtree. Now consider an arbitrary MST of G, call it T_2^*. Note that because v must be connected in T_2^*, if e does not connect e in T_2^*, there must exist some other path between T and v in T_2^*. Let the edge in this path that flows across the cut induced by T be e'.

Now, consider the alternative tree of G where we replace e' with e, call it T_1^*. T_1^* is a spanning tree because any vertex that was reachable through e' is now reachable through e. Furthermore, note that e and e' are both edges across the cut induced by T, but e is assumed to have minimum weight among such edges. Therefore, weight(e) $\leq$ weight(e'). But T_1^* only differs from T_2^* in this edge, so we can conclude that weight(T_1^*) $\leq$ weight(T_2^*) which implies that weight(T_1^*) is a MST for G, a contradiction since e belongs to it. □

This argument is kind of *exchange argument* for greedy algorithms. We justify the greedy choice by arguing that any "sub-optimal" choice could be substituted by the greedy choice to obtain a solution at least as good as the original one. Our argument accounts for the fact that since edge weights are not necessarily distinct that e and e' could be both valid minimal choices. The resulting MSTs are, therefore, potentially different, but both have the same (minimal) weight. Note that if the edge weights of G are distinct, then e is the minimum weight edge of the cut and thus T_2^* in the proof above is not a MST; T_1^*. This implies that in the case where edge weights are distinct, the minimum weight edge e must belong to *any* MST of G, not just one of them.

We can use this lemma to prove Prim's algorithm easily. The algorithm maintains a MST for the vertices it contains so far, extending the MST by one edge (and thus, one vertex) on each iteration. We therefore prove the correctness of Prim's algorithm by induction of the number of iterations of its loop, claiming that T is MST for the vertices it contains so far.

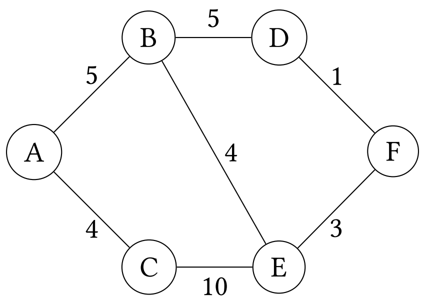
Claim 2. *On the n-th iteration of Prim's on a graph $G = (V, E)$, $T = (V', E')$ with $V' \subseteq V$ and $E \subseteq E'$ is a MST for V'.*

Proof. By induction on n.

- $n = 0$. Initially T contains a single vertex and is trivially a MST for that one vertex.
- $n = k + 1$. Our induction hypothesis says that on iteration k, $T = (V', E')$ is a MST for V' In the $(k + 1)$-st iteration, we extend T with an edge e across the cut induced by T. By the Cut Property, we know that e can belong some MST T^* of G with $T^* \subseteq T$. Therefore, we know that T extended with e is a MST for its vertices. □

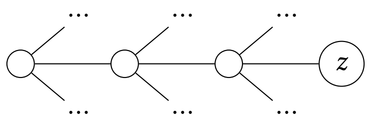
9.3.2 Shortest Paths

A related problem to minimum spanning trees is *shortest paths*. That is, what is the shortest path between two vertices in a graph, say a and b. A naive greedy approach seems compelling given our exploration of minimum spanning trees. For example, consider the following graph:



Suppose we want to find the shortest path from A to F. We could try to start from A, and choose the edge of least weight to traverse next, eventually arriving at F. However, this yields the path A, C, E, F which has total weight $4 + 10 + 3 = 17$. The optimal path instead is the top path of the graph: A, B, D, F with total weight $5 + 5 + 1 = 11$. Note that even if we were to divine that it is better to go to B from A, we encounter the same problem at B: a greedy choice will send us down to E (at cost 4) when traversing D would have been the correct move!

More generally, this approach has the problem where it may send us down a sub-optimal path:

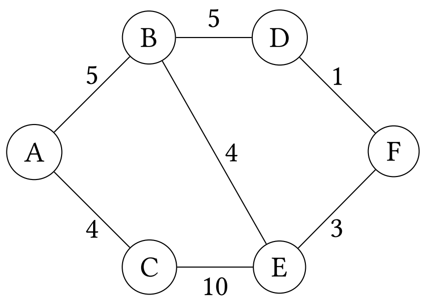


For example, we may reach vertex z in the graph above only to find that we are either in a sub-optimal path or worse yet, that the target node of our path is unreachable from z! In this case, we would be forced to backtrack to explore other possible paths. In general, this is what occurs when a greedy choice fails: we have to "undo" our optimal choice and try other possibilities instead. We may end up exploring all such possibilities exhaustively which is problematic if there are a large number of these possibilities to consider! This realization might motivate us to dismiss any sort of greedy algorithm for this problem. However, if we are smart in tracking enough information so that we never need to backtrack, we can retain a greedy approach!

The algorithm we'll consider is *Dijkstra's algorithm* which we can think of as a refinement of Prim's algorithm for minimal path searching. Note that Prim's proceeded by growing a minimal spanning tree from a single node. Dijkstra's proceeds similarly, growing an optimal path from the start vertex under consideration. However, unlike Prim's which only tracks the growing MST of interest, Dijkstra's does not just record the current optimal path to the desired end vertex but *all* such optimal paths to every vertex in the graph from the start vertex. This additional information is sufficient for us to make greedy choices that always lead to the discovery of the optimal path.

Suppose we are interested in finding the shortest path from A to F in our example graph for this section. Dijkstra's ultimately tracks the shortest path from A *through the nodes it has visited so far* to *all nodes in the graph*, refining these paths as it greedily consumes vertices in the graph. Initially, we know that the shortest path from A to itself is simply staying at A for a cost of 0. For every other node, we don't know a path—indeed, such a path may not exist!—so we assign the value ∞ to these paths.

Destination	Path	Cost
A	A	0
B	?	∞
C	?	∞
D	?	∞
E	?	∞
F	?	∞

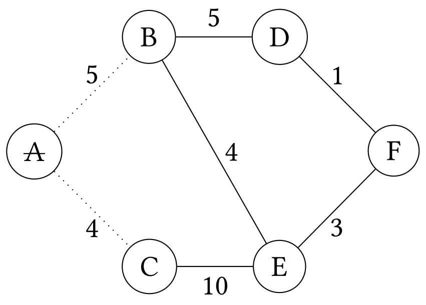


We begin by considering A and its edges. We look at each edge incident to A and update our table based on these edges. When looking at these edges, we ask the question

Can taking this edge to a vertex v give us a new optimal path from A to v?

There are two such edges to consider: AB and AC. Since we don't know of any paths from A to either of these edge's endpoints—represented by the fact that their entries in the table are ∞—we can update our shortest paths entries for these vertices with these edges.

Destination	Path	Cost
A	A	0
B	AB	5
C	AC	4
D	?	∞
E	?	∞
F	?	∞

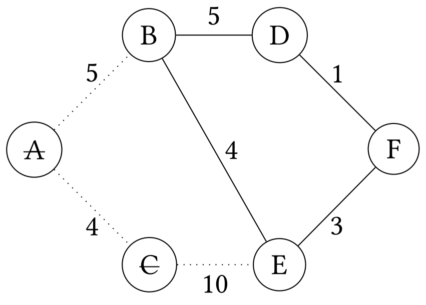


After processing these edges, we are done with A. An invariant of the algorithm is that we never need to reconsider these edges again. The important information about them has been recorded in the table, allowing us to avoid backtracking if an optimal path needs to be updated!

We now repeat the process by choosing a vertex that we have not yet visited and updating the table based on its incident edges that we have not yet considered. Which vertex do we consider next? This is where we'll make our greedy choice: we'll consider the vertex with *minimum cost* according to the table that we have not yet visited.

In our running example, this is node C with current path cost 4. We now update our table with C's additional edge: CE. Note that this edge gives us a path from A to E; what is its length? It is the length of the *optimal* path from A to C plus the cost of traversing CE! This quantity is $4 + 10 = 14$ corresponding to the path ACE.

Destination	Path	Cost
A	A	0
B	AB	5
C	AC	4
D	?	∞
E	ACE	14
F	?	∞

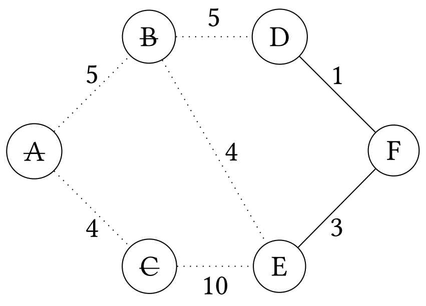


Next we consider node *B* since its current shortest path cost 4 is lower than *E*'s cost 14. We thus consider edges *BD* and *BE* next. Edge *BD* updates the path to *D* as expected. However, edge *BE* introduces a choice between two paths:

- The *current shortest path* in the table: *ACE* with cost 14.
- The *candidate shortest path* through *BE*. The cost of this path is the minimal cost of reaching *B* from *A* plus the cost of traversing *BE*: $5 + 4 = 9$.

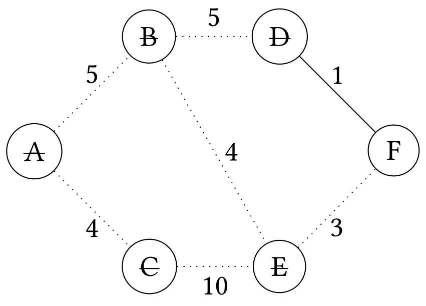
We note that $9 < 14$ and thus the candidate path is shorter than our current best known path. We therefore update the table to reflect the fact, recording a new best shortest path for *E*!

Destination	**Path**	**Cost**
A	*A*	0
B	*AB*	5
C	*AC*	4
D	*ABD*	10
E	*ABE*	9
F	?	∞



We next consider *E* with current shortest path length 9. It has one unvisited incident edge, *EF*, allowing us to final reach our intended endpoint, *F*, with cost $9 + 3 = 12$.

Destination	**Path**	**Cost**
A	*A*	0
B	*AB*	5
C	*AC*	4
D	*ABD*	10
E	*ABE*	9
F	*ABEF*	12

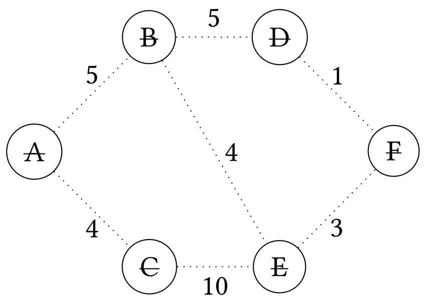


Now we will consider vertex *D* with edge *DF*. We employ the same logic here as with *B*. We need to compare the:

- The current shortest path recorded in the table: *ABEF* of length 12.
- The candidate shortest path through *D*: *ABDF* of length $10 + 1 = 11$.

The candidate shortest path is shorter, so we update the table with the new path for *F*.

Destination	**Path**	**Cost**
A	*A*	0
B	*AB*	5
C	*AC*	4
D	*ABD*	10
E	*ABE*	9
F	*ABDF*	11



Since F has no unvisited edges to process, we don't need to do anything to process it, completing the search procedure. We can now inspect the table to find the shortest path from A to F which is $ABEF$ of length 11 as desired!

With this example, we see that the sailent parts of Dijkstra's algorithm are:

- Repeatedly choosing vertices to process based on their current, best-known shortest paths from the start vertex.
- Comparing our current known shortest paths with paths through the current vertex under consideration and choosing the better of the two.

Let $G = (V, E)$ be an undirected graph. Suppose that we are interested in finding all shortest paths starting from vertex $s \in V$. Let $\text{cost}(x)$ be the cost of the shortest known path from s to x. Dijkstra's algorithm proceeds as follows:

1. Initially, let $\text{cost}(s) = 0$ and $\text{cost}(v) = \infty$ for all $u \in V$ that are not s.
2. Repeatedly choose vertex u that has minimal $\text{cost}(v)$ among all vertices that have not yet been processed in V.
    - For each edge $(u, v) \in E$ that has not yet been processed by the algorithm, update the shortest known path to v as follows:

$$\text{cost}(v) \leftarrow \min(\text{cost}(v), \text{cost}(u) + \text{weight}(uv))$$

After Dijkstra's algorithm completes, $\text{cost}(t)$ records the shortest path from s to t in G for all $t \in V$.

Note that our choice of initially assigning unknown paths the value ∞ makes the description of our algorithm concise. We don't need to define a special case for when we first find a path to a target vertex. We will always choose the found path because ∞ is effectively larger than the length of any known path length we would consider during execution of the algorithm.