# COMP90038
# Algorithms and Complexity

Lecture 20: Greedy Techniques – Prim and Dijkstra

(with thanks to Harald Søndergaard & Michael Kirley)

Andres Munoz-Acosta

munoz.m@unimelb.edu.au

Peter Hall Building G.83

# On the previous lecture

- We have talked a lot about **dynamic programming:**
  - DP is bottom-up problem solving technique
  - Similar to divide-and-conquer; however, problems are overlapping
  - Solutions often involve recursion
- We applied this idea to two graph problems:
  - Computing the **transitive closure** of a directed graph
  - **Finding shortest distances** in weighted directed graphs

# Greedy algorithms

- A problem solving strategy is to take the **locally best** choice among all feasible ones
  - Once we do this, our decision is **irrevocable**

- We want to change 30 cents using the smallest number of coins
  - Assuming denominations of $\{25, 10, 5, 1\}$, we could use as many $25$-cent pieces as we can, then do the same for $10$-cent pieces, and so on, until we have reached $30$ cents $(25+5)$

  - This **greedy** strategy would not work for denominations $\{25, 10, 1\}$ ($25+1+1+1+1+1$ compared to $10+10+10$)

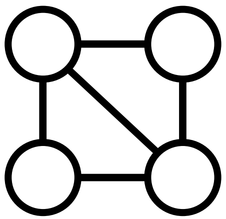
# Greedy algorithms

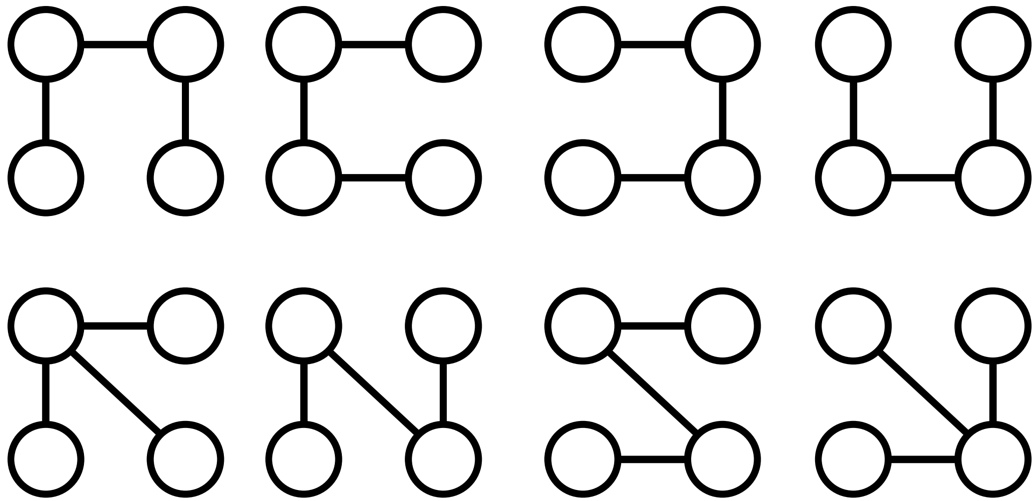
- In general, it is unusual that **locally best** choices yield **global best** results
  - However, there are problems for which **a greedy algorithm is correct and fast**
  - In some other problems, a greedy algorithm is an acceptable **approximation algorithm**

- Here we shall look at:
  - Prim's algorithm for finding **minimum spanning trees**
  - Dijkstra's algorithm for **single-source shortest paths**

# What is a spanning tree?

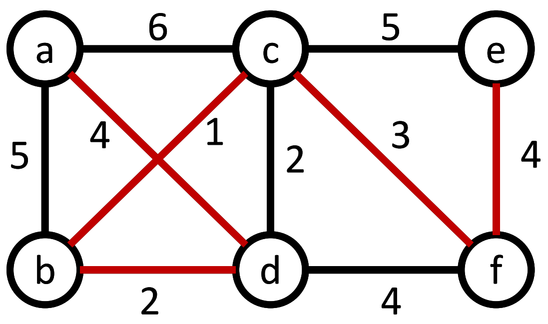
- Recall that a **tree** is a connected graph with no cycles
- A **spanning tree** of a graph $\langle V,E\rangle$ is a tree $\langle V,E'\rangle$ where $E'$ is a subset of $E$
- For example, the graph on the left has eight different spanning trees:





# Minimum spanning trees of weighted graphs

- For a **weighted graph**, some spanning trees are more desirable than others
  - For example, suppose we have a set of “stations” to connect in a network, and also some possible connections, each with its own **cost**
- This is the problem of finding a spanning tree with the smallest possible cost
  - Such tree is a **minimum spanning tree** for the graph



# Prim’s algorithm

- Prim's algorithm is an example of a greedy algorithm
  - It constructs a sequence of subtrees $T$, by **adding to the latest tree the closest node not currently on it**

- A simple version:

$$
\begin{array}{l}
\textbf{function } \textsc{Prim}(\langle V, E\rangle) \\
\quad V_T \leftarrow \{v_0\} \\
\quad E_T \leftarrow \emptyset \\
\quad \textbf{for } i \leftarrow 1 \text{ to } |V|-1 \textbf{ do} \\
\quad\quad \boxed{\text{find a minimum-weight edge } (v,u) \in V_T \times (V \setminus V_T)} \\
\quad\quad V_T \leftarrow V_T \cup \{u\} \\
\quad\quad E_T \leftarrow E_T \cup \{(v,u)\} \\
\quad \textbf{return } E_T
\end{array}
$$

# Prim’s algorithm

- But how to find the **minimum-weight edge** $(v,u)$?

- An approach is to organise the nodes that are not yet included in the spanning tree $T$ as a **priority queue**, using a **min-heap** by edge **cost**

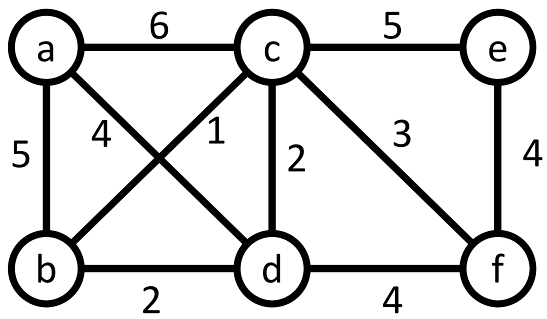
- Which nodes are connected in $T$ is stored by an array $prev$ of nodes, indexed by $V$. Namely, when $(v,u)$ is included, this is stored by setting $prev[u] = v$

# Prim's algorithm

- The complete algorithm is:

$$
\begin{array}{ll}
\textbf{function } \textsc{Prim}(\langle V, E \rangle) & \\
\quad \textbf{for } \text{each } v \in V \textbf{ do} & \\
\quad\quad cost[v] \leftarrow \infty & \\
\quad\quad prev[v] \leftarrow nil & \\
\quad \text{pick initial node } v_0 & \\
\quad cost[v_0] \leftarrow 0 & \\
\quad \boxed{Q \leftarrow \textsc{InitPriorityQueue}(V)} & \triangleright \text{ priorities are cost values} \\
\quad \textbf{while } Q \text{ is non-empty } \textbf{do} & \\
\quad\quad \boxed{u \leftarrow \textsc{EjectMin}(Q)} & \\
\quad\quad \textbf{for } \text{each } (u, w) \in E \textbf{ do} & \\
\quad\quad\quad \textbf{if } weight(u, w) < cost[w] \textbf{ then} & \\
\quad\quad\quad\quad cost[w] \leftarrow weight(u, w) & \\
\quad\quad\quad\quad prev[w] \leftarrow u & \\
\quad\quad\quad\quad \boxed{\textsc{Update}(Q, w, cost[w])} & \triangleright \text{ rearranges priority queue}
\end{array}
$$

# Prim’s algorithm




- On the first loop, we only create the table

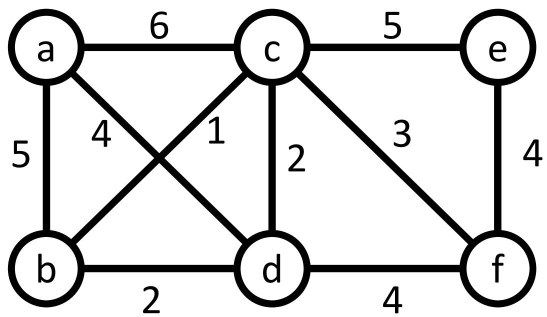
**function** $\textsc{Prim}(\langle V, E \rangle)$
    **for** each $v \in V$ **do**
        $cost[v] \leftarrow \infty$
        $prev[v] \leftarrow nil$
    pick initial node $v_0$
    $cost[v_0] \leftarrow 0$
    $Q \leftarrow \textsc{InitPriorityQueue}(V)$
    **while** $Q$ is non-empty **do**
        $u \leftarrow \textsc{EjectMin}(Q)$
        **for** each $(u, w) \in E$ **do**
            **if** $weight(u, w) < cost[w]$ **then**
                $cost[w] \leftarrow weight(u, w)$
                $prev[w] \leftarrow u$
                $\textsc{Update}(Q, w, cost[w])$

| Tree T | | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|
| | cost | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

# Prim’s algorithm

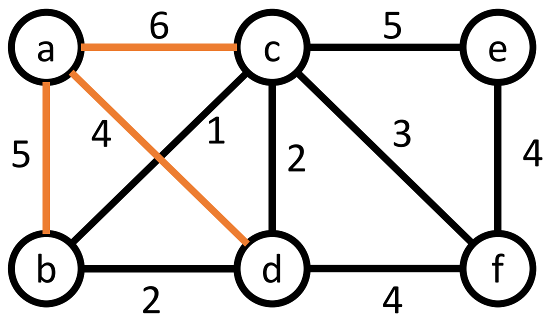
- Then we pick the first node as the initial one

**function** PRIM($\langle V, E\rangle$)
  **for** each $v \in V$ **do**
    $cost[v] \leftarrow \infty$
    $prev[v] \leftarrow nil$
  $\boxed{\text{pick initial node } v_0 \quad cost[v_0] \leftarrow 0}$
  $Q \leftarrow$ INITPRIORITYQUEUE($V$)
  **while** $Q$ is non-empty **do**
    $u \leftarrow$ EJECTMIN($Q$)
    **for** each $(u, w) \in E$ **do**
      **if** $weight(u, w) < cost[w]$ **then**
        $cost[w] \leftarrow weight(u, w)$
        $prev[w] \leftarrow u$
        UPDATE($Q, w, cost[w]$)




| Tree T | | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|
| | cost | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| | cost | 0 | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

# Prim’s algorithm




- We take the first node out of the queue and update the costs

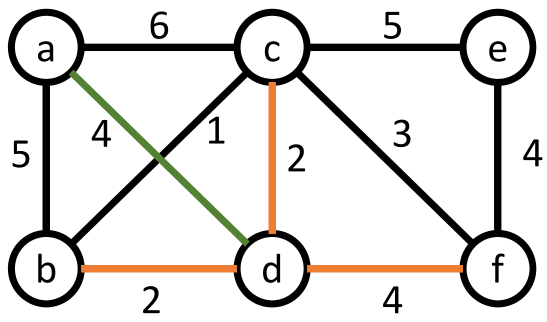
**function** $\textsc{Prim}(\langle V, E\rangle)$
  **for** each $v \in V$ **do**
    $cost[v] \leftarrow \infty$
    $prev[v] \leftarrow nil$
  pick initial node $v_0$
  $cost[v_0] \leftarrow 0$
  $Q \leftarrow \textsc{InitPriorityQueue}(V)$
  **while** $Q$ is non-empty **do**
    $u \leftarrow \textsc{EjectMin}(Q)$
    **for** each $(u, w) \in E$ **do**
      **if** $weight(u, w) < cost[w]$ **then**
        $cost[w] \leftarrow weight(u, w)$
        $prev[w] \leftarrow u$
        $\textsc{Update}(Q, w, cost[w])$

| Tree T | | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|
| | cost | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| | cost | 0 | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| a | cost | | 5 | 6 | 4 | ∞ | ∞ |
| | prev | | a | a | a | nil | nil |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

# Prim's algorithm

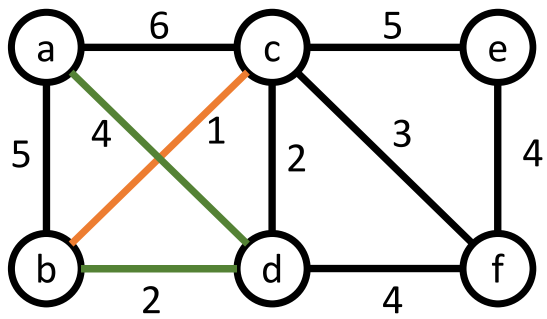
- We eject the node with the lowest cost and update the queue

**function** $\textsc{Prim}(\langle V, E \rangle)$
    **for** each $v \in V$ **do**
        $cost[v] \leftarrow \infty$
        $prev[v] \leftarrow nil$
    pick initial node $v_0$
    $cost[v_0] \leftarrow 0$
    $Q \leftarrow \textsc{InitPriorityQueue}(V)$
    **while** $Q$ is non-empty **do**
        $u \leftarrow \textsc{EjectMin}(Q)$
        **for** each $(u, w) \in E$ **do**
            **if** $weight(u, w) < cost[w]$ **then**
                $cost[w] \leftarrow weight(u, w)$
                $prev[w] \leftarrow u$
                $\textsc{Update}(Q, w, cost[w])$




| Tree T | | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|
| | cost | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| | cost | 0 | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| a | cost | | 5 | 6 | 4 | ∞ | ∞ |
| | prev | | a | a | a | nil | nil |
| a,d | cost | | 2 | 2 | | ∞ | 4 |
| | prev | | d | d | | nil | d |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

# Prim’s algorithm




- We eject the next node based on alphabetical order.
  **Why is `(f)` not updated?**

**function** $\textsc{Prim}(\langle V, E\rangle)$

    **for** each $v \in V$ **do**

        $cost[v] \leftarrow \infty$

        $prev[v] \leftarrow nil$

    pick initial node $v_0$

    $cost[v_0] \leftarrow 0$

    $Q \leftarrow \textsc{InitPriorityQueue}(V)$

    **while** $Q$ is non-empty **do**

        $u \leftarrow \textsc{EjectMin}(Q)$

        **for** each $(u, w) \in E$ **do**

            **if** $weight(u, w) < cost[w]$ **then**

                $cost[w] \leftarrow weight(u, w)$

                $prev[w] \leftarrow u$

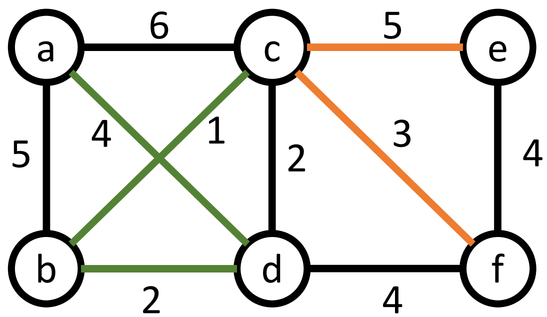
                $\textsc{Update}(Q, w, cost[w])$

| Tree T | | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|
| | cost | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| | cost | 0 | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| a | cost | | 5 | 6 | 4 | ∞ | ∞ |
| | prev | | a | a | a | nil | nil |
| a,d | cost | | 2 | 2 | | ∞ | 4 |
| | prev | | d | d | | nil | d |
| a,d,b | cost | | | 1 | | ∞ | 4 |
| | prev | | | b | | nil | d |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

# Prim's algorithm

- We now update `(f)`

**function** $\textsc{Prim}(\langle V, E\rangle)$
    **for** each $v \in V$ **do**
        $cost[v] \leftarrow \infty$
        $prev[v] \leftarrow nil$
    pick initial node $v_0$
    $cost[v_0] \leftarrow 0$
    $Q \leftarrow \textsc{InitPriorityQueue}(V)$
    **while** $Q$ is non-empty **do**
        $u \leftarrow \textsc{EjectMin}(Q)$
        **for** each $(u, w) \in E$ **do**
            **if** $weight(u, w) < cost[w]$ **then**
                $cost[w] \leftarrow weight(u, w)$
                $prev[w] \leftarrow u$
                $\textsc{Update}(Q, w, cost[w])$

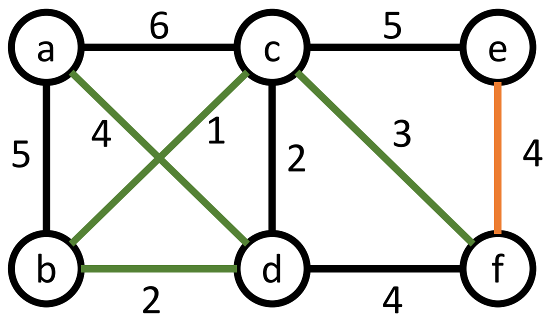



| Tree T | | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|
| | cost | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| | cost | 0 | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| a | cost | | 5 | 6 | **4** | ∞ | ∞ |
| | prev | | a | a | **a** | nil | nil |
| a,d | cost | | **2** | 2 | | ∞ | 4 |
| | prev | | **d** | d | | nil | d |
| a,d,b | cost | | | **1** | | ∞ | 4 |
| | prev | | | **b** | | nil | d |
| a,d,b,c | cost | | | | | **5** | **3** |
| | prev | | | | | **c** | **c** |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

# Prim’s algorithm

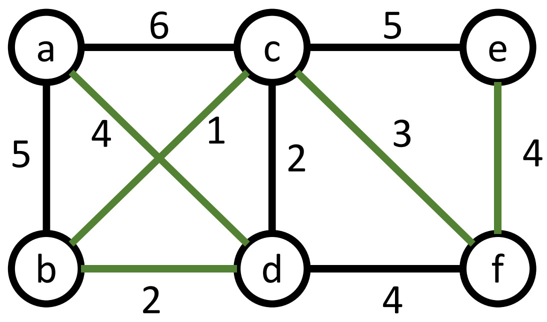
- We reach the last choice

**function** $\textsc{Prim}(\langle V, E \rangle)$
    **for** each $v \in V$ **do**
        $cost[v] \leftarrow \infty$
        $prev[v] \leftarrow nil$
    pick initial node $v_0$
    $cost[v_0] \leftarrow 0$
    $Q \leftarrow \textsc{InitPriorityQueue}(V)$
    **while** $Q$ is non-empty **do**
        $u \leftarrow \textsc{EjectMin}(Q)$
        **for** each $(u, w) \in E$ **do**
            **if** $weight(u, w) < cost[w]$ **then**
                $cost[w] \leftarrow weight(u, w)$
                $prev[w] \leftarrow u$
                $\textsc{Update}(Q, w, cost[w])$




| Tree T | | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|
| | cost | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| | cost | 0 | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| a | cost | | 5 | 6 | 4 | ∞ | ∞ |
| | prev | | a | a | a | nil | nil |
| a,d | cost | | 2 | 2 | | ∞ | 4 |
| | prev | | d | d | | nil | d |
| a,d,b | cost | | | 1 | | ∞ | 4 |
| | prev | | | b | | nil | d |
| a,d,b,c | cost | | | | | 5 | 3 |
| | prev | | | | | c | c |
| a,d,b,c,f | cost | | | | | 4 | |
| | prev | | | | | f | |
| | | | | | | | |
| | | | | | | | |

# Prim's algorithm




- The resulting tree is `{a,d,b,c,f,e}`

**function** $\textsc{Prim}(\langle V, E\rangle)$
    **for** each $v \in V$ **do**
        $cost[v] \leftarrow \infty$
        $prev[v] \leftarrow nil$
    pick initial node $v_0$
    $cost[v_0] \leftarrow 0$
    $Q \leftarrow \textsc{InitPriorityQueue}(V)$
    **while** $Q$ is non-empty **do**
        $u \leftarrow \textsc{EjectMin}(Q)$
        **for** each $(u, w) \in E$ **do**
            **if** $weight(u, w) < cost[w]$ **then**
                $cost[w] \leftarrow weight(u, w)$
                $prev[w] \leftarrow u$
                $\textsc{Update}(Q, w, cost[w])$

| Tree T | | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|
| | cost | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| | cost | 0 | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| a | cost | | 5 | 6 | **4** | ∞ | ∞ |
| | prev | | a | a | **a** | nil | nil |
| a,d | cost | | **2** | 2 | | ∞ | 4 |
| | prev | | **d** | d | | nil | d |
| a,d,b | cost | | | **1** | | ∞ | 4 |
| | prev | | | **b** | | nil | d |
| a,d,b,c | cost | | | | | 5 | **3** |
| | prev | | | | | c | **c** |
| a,d,b,c,f | cost | | | | | **4** | |
| | prev | | | | | **f** | |
| a,d,b,c,f,e | cost | | | | | | |
| | prev | | | | | | |

# Analysis of Prim's algorithm

- First, a crude analysis: For each node, we look through the edges to find those incident to the node, and pick the one with smallest cost. Thus we get $O(|V|\times|E|)$. However, we are using cleverer data structures.

- Using adjacency lists for the graph and a min-heap for the priority queue, we perform $|V|$ - $1$ heap deletions (each at cost $O(\log |V|)$) and $|E|$ updates of priorities (each at cost $O(\log |V|)$).

- Altogether $(|V|-1+|E|)\ O(\log |V|)$.

- Since, in a connected graph, $|V|-1 \leq |E|$, this is $O(|E| \log |\mathrm{V}|)$.

# Dijkstra’s algorithm

- Another classical greedy weighted-graph algorithm is **Dijkstra's algorithm**, whose overall structure is the same as Prim's

- On **Lecture 19** we talked about Floyd's algorithm:
  - It gave us the shortest paths, **for every pair of nodes**, in a (directed or undirected) weighted graph.
  - Assumes an adjacency matrix representation and had complexity $O(|V|^3)$

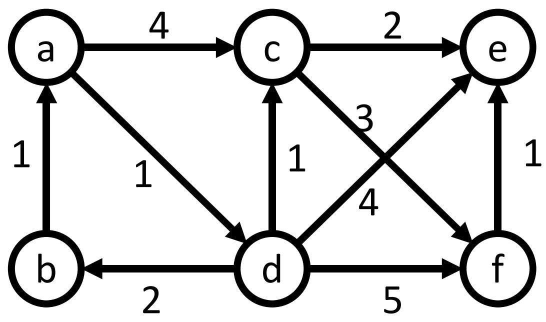
- **Dijkstra's algorithm** is also a shortest-path algorithm for (directed or undirected) weighted graphs
  - It finds all shortest paths **from a fixed start node**
  - Its complexity is the same as that of Prim's algorithm

# Dijkstra's algorithm

- The complete algorithm is:

**function** $\textsc{Dijkstra}(\langle V, E\rangle, v_0)$
    **for** each $v \in V$ **do**
        $\mathit{dist}[v] \leftarrow \infty$
        $\mathit{prev}[v] \leftarrow \mathit{nil}$
    $\mathit{dist}[v_0] \leftarrow 0$
    $\boxed{Q \leftarrow \textsc{InitPriorityQueue}(V)}$      ▷ priorities are distances
    **while** $Q$ is non-empty **do**
        $\boxed{u \leftarrow \textsc{EjectMin}(Q)}$
        **for** each $(u, w) \in E$ **do**
            **if** $\mathit{dist}[u] + \mathit{weight}(u, w) < \mathit{dist}[w]$ **then**
                $\mathit{dist}[w] \leftarrow \mathit{dist}[u] + \mathit{weight}(u, w)$
                $\mathit{prev}[w] \leftarrow u$
                $\boxed{\textsc{Update}(Q, w, \mathit{dist}[w])}$      ▷ rearranges priority queue

# Dijkstra’s algorithm




- On the first loop, we only create the table

$$
\begin{array}{l}
\textbf{function } \textsc{Dijkstra}(\langle V, E\rangle, v_0) \\
\quad \boxed{\begin{array}{l} \textbf{for } \text{each } v \in V \textbf{ do} \\ \quad \mathit{dist}[v] \leftarrow \infty \\ \quad \mathit{prev}[v] \leftarrow \mathit{nil} \end{array}} \\
\quad \mathit{dist}[v_0] \leftarrow 0 \\
\quad Q \leftarrow \textsc{InitPriorityQueue}(V) \\
\quad \textbf{while } Q \text{ is non-empty } \textbf{do} \\
\quad\quad u \leftarrow \textsc{EjectMin}(Q) \\
\quad\quad \textbf{for } \text{each } (u, w) \in E \textbf{ do} \\
\quad\quad\quad \textbf{if } \mathit{dist}[u] + \mathit{weight}(u, w) < \mathit{dist}[w] \textbf{ then} \\
\quad\quad\quad\quad \mathit{dist}[w] \leftarrow \mathit{dist}[u] + \mathit{weight}(u, w) \\
\quad\quad\quad\quad \mathit{prev}[w] \leftarrow u \\
\quad\quad\quad\quad \textsc{Update}(Q, w, \mathit{dist}[w])
\end{array}
$$

| Covered | | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|
| | cost | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

# Dijkstra’s algorithm

- Then we pick the first node as the initial one

$\textbf{function } \textsc{Dijkstra}(\langle V, E\rangle, v_0)$

$\quad \textbf{for } \text{each } v \in V \textbf{ do}$

$\quad\quad \mathit{dist}[v] \leftarrow \infty$

$\quad\quad \mathit{prev}[v] \leftarrow \mathit{nil}$

$\quad \boxed{\mathit{dist}[v_0] \leftarrow 0}$

$\quad Q \leftarrow \textsc{InitPriorityQueue}(V)$

$\quad \textbf{while } Q \text{ is non-empty } \textbf{do}$

$\quad\quad u \leftarrow \textsc{EjectMin}(Q)$

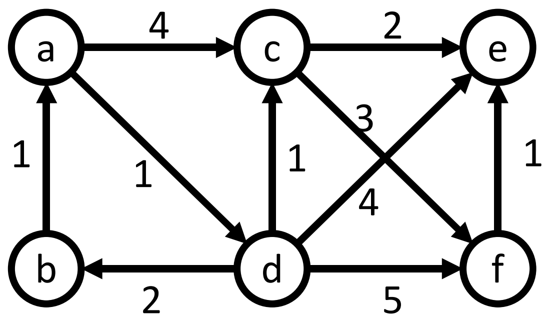
$\quad\quad \textbf{for } \text{each } (u, w) \in E \textbf{ do}$

$\quad\quad\quad \textbf{if } \mathit{dist}[u] + \mathit{weight}(u, w) < \mathit{dist}[w] \textbf{ then}$

$\quad\quad\quad\quad \mathit{dist}[w] \leftarrow \mathit{dist}[u] + \mathit{weight}(u, w)$

$\quad\quad\quad\quad \mathit{prev}[w] \leftarrow u$

$\quad\quad\quad\quad \textsc{Update}(Q, w, \mathit{dist}[w])$

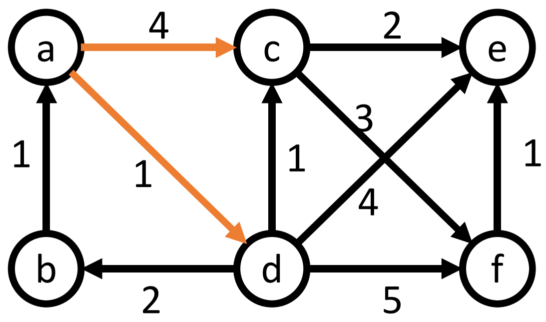



| Covered | | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|
| | cost | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| | cost | **0** | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | **nil** | nil | nil | nil | nil | nil |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

# Dijkstra's algorithm

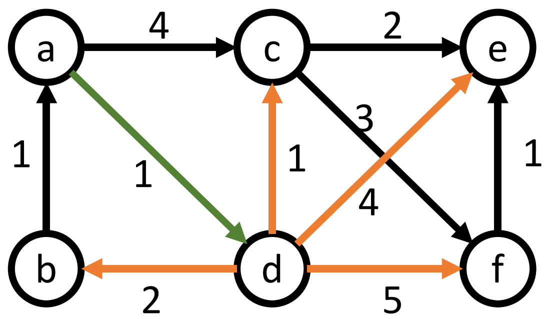
- Then we pick the first node as the initial one

$$
\begin{array}{l}
\textbf{function } \textsc{Dijkstra}(\langle V, E\rangle, v_0) \\
\quad \textbf{for } \text{each } v \in V \textbf{ do} \\
\quad\quad \mathit{dist}[v] \leftarrow \infty \\
\quad\quad \mathit{prev}[v] \leftarrow \mathit{nil} \\
\quad \mathit{dist}[v_0] \leftarrow 0 \\
\quad Q \leftarrow \textsc{InitPriorityQueue}(V) \\
\quad \boxed{\begin{array}{l}
\textbf{while } Q \text{ is non-empty } \textbf{do} \\
\quad u \leftarrow \textsc{EjectMin}(Q) \\
\quad \textbf{for } \text{each } (u, w) \in E \textbf{ do} \\
\quad\quad \textbf{if } \mathit{dist}[u] + \mathit{weight}(u, w) < \mathit{dist}[w] \textbf{ then} \\
\quad\quad\quad \mathit{dist}[w] \leftarrow \mathit{dist}[u] + \mathit{weight}(u, w) \\
\quad\quad\quad \mathit{prev}[w] \leftarrow u \\
\quad\quad\quad \textsc{Update}(Q, w, \mathit{dist}[w])
\end{array}}
\end{array}
$$




| Covered | | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|
| | cost | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| | cost | 0 | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| a | cost | | ∞ | 4 | 1 | ∞ | ∞ |
| | prev | | nil | a | a | nil | nil |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

# Dijkstra's algorithm

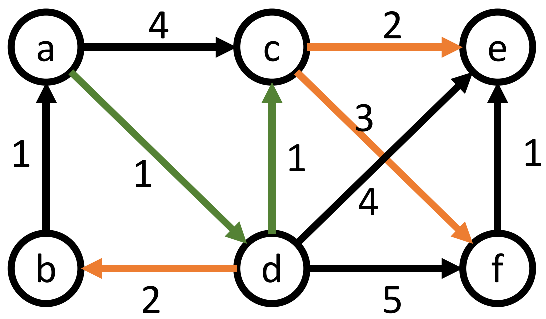



- Then eject the node with the shortest distance from the queue. Then, **we update all the paths by adding 1.**

**function** $\textsc{Dijkstra}(\langle V, E\rangle, v_0)$
    **for** each $v \in V$ **do**
        $dist[v] \leftarrow \infty$
        $prev[v] \leftarrow nil$
    $dist[v_0] \leftarrow 0$
    $Q \leftarrow \textsc{InitPriorityQueue}(V)$
    **while** $Q$ is non-empty **do**
        $u \leftarrow \textsc{EjectMin}(Q)$
        **for** each $(u, w) \in E$ **do**
            **if** $dist[u] + weight(u, w) < dist[w]$ **then**
                $dist[w] \leftarrow dist[u] + weight(u, w)$
                $prev[w] \leftarrow u$
                $\textsc{Update}(Q, w, dist[w])$

| Covered | | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|
| | cost | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| | cost | 0 | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| a | cost | | ∞ | 4 | 1 | ∞ | ∞ |
| | prev | | nil | a | a | nil | nil |
| a,d | cost | | 3 | 2 | | 5 | 6 |
| | prev | | d | d | | d | d |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

# Dijkstra's algorithm




- Our next node will be the one with the shortest path in overall (b)

**function** $\textsc{Dijkstra}(\langle V, E\rangle, v_0)$
  **for** each $v \in V$ **do**
    $dist[v] \leftarrow \infty$
    $prev[v] \leftarrow nil$
  $dist[v_0] \leftarrow 0$
  $Q \leftarrow \textsc{InitPriorityQueue}(V)$
  **while** $Q$ is non-empty **do**
    $u \leftarrow \textsc{EjectMin}(Q)$
    **for** each $(u, w) \in E$ **do**
      **if** $dist[u] + weight(u, w) < dist[w]$ **then**
        $dist[w] \leftarrow dist[u] + weight(u, w)$
        $prev[w] \leftarrow u$
        $\textsc{Update}(Q, w, dist[w])$

| Covered | | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|
| | cost | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| | cost | 0 | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| a | cost | | ∞ | 4 | 1 | ∞ | ∞ |
| | prev | | nil | a | a | nil | nil |
| a,d | cost | | 3 | 2 | | 5 | 6 |
| | prev | | d | d | | d | d |
| a,d,c | cost | | 3 | | | 4 | 5 |
| | prev | | d | | | c | c |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

# Dijkstra's algorithm

- Now, we continue evaluating from (c)

**function** $\textsc{Dijkstra}(\langle V, E\rangle, v_0)$

  **for** each $v \in V$ **do**

    $dist[v] \leftarrow \infty$

    $prev[v] \leftarrow nil$

  $dist[v_0] \leftarrow 0$

  $Q \leftarrow \textsc{InitPriorityQueue}(V)$

  **while** $Q$ is non-empty **do**

    $u \leftarrow \textsc{EjectMin}(Q)$

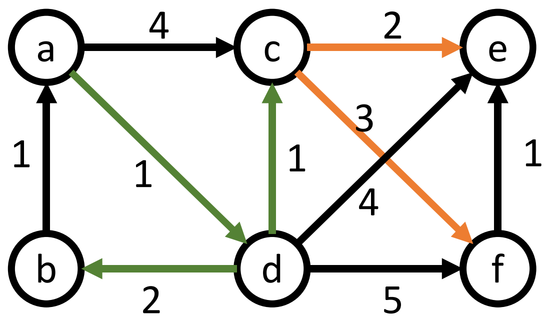
    **for** each $(u, w) \in E$ **do**

      **if** $dist[u] + weight(u, w) < dist[w]$ **then**

        $dist[w] \leftarrow dist[u] + weight(u, w)$

        $prev[w] \leftarrow u$

        $\textsc{Update}(Q, w, dist[w])$

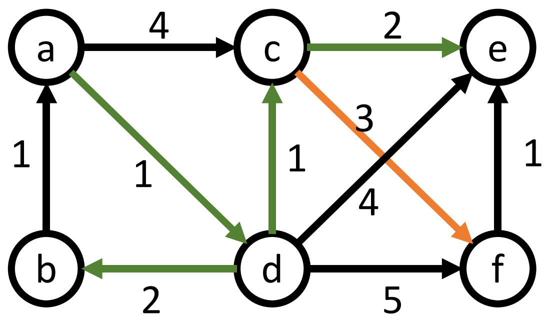



| Covered | | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|
| | cost | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| | cost | 0 | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| a | cost | | ∞ | 4 | 1 | ∞ | ∞ |
| | prev | | nil | a | a | nil | nil |
| a,d | cost | | 3 | 2 | | 5 | 6 |
| | prev | | d | d | | d | d |
| a,d,c | cost | | 3 | | | 4 | 5 |
| | prev | | d | | | c | c |
| a,d,c,b | cost | | | | | 4 | 5 |
| | prev | | | | | c | c |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

# Dijkstra's algorithm

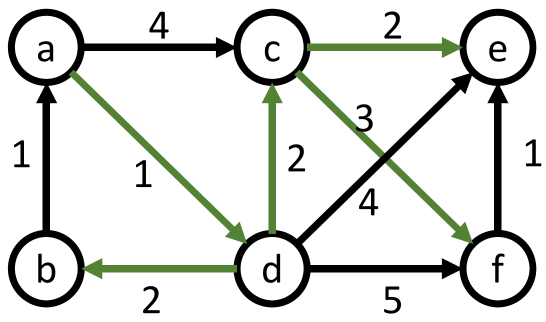
- We arrive at our last decision.

**function** $\textsc{Dijkstra}(\langle V, E\rangle, v_0)$
  **for** each $v \in V$ **do**
    $dist[v] \leftarrow \infty$
    $prev[v] \leftarrow nil$
  $dist[v_0] \leftarrow 0$
  $Q \leftarrow \textsc{InitPriorityQueue}(V)$
  **while** $Q$ is non-empty **do**
    $u \leftarrow \textsc{EjectMin}(Q)$
    **for** each $(u, w) \in E$ **do**
      **if** $dist[u] + weight(u, w) < dist[w]$ **then**
        $dist[w] \leftarrow dist[u] + weight(u, w)$
        $prev[w] \leftarrow u$
        $\textsc{Update}(Q, w, dist[w])$




| Covered | | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|
| | cost | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| | cost | 0 | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| a | cost | | ∞ | 4 | 1 | ∞ | ∞ |
| | prev | | nil | a | a | nil | nil |
| a,d | cost | | 3 | 2 | | 5 | 6 |
| | prev | | d | d | | d | d |
| a,d,c | cost | | 3 | | | 4 | 5 |
| | prev | | d | | | c | c |
| a,d,c,b | cost | | | | | 4 | 5 |
| | prev | | | | | c | c |
| a,d,c,b,e | cost | | | | | | 5 |
| | prev | | | | | | c |
| | | | | | | | |
| | | | | | | | |

# Dijkstra's algorithm



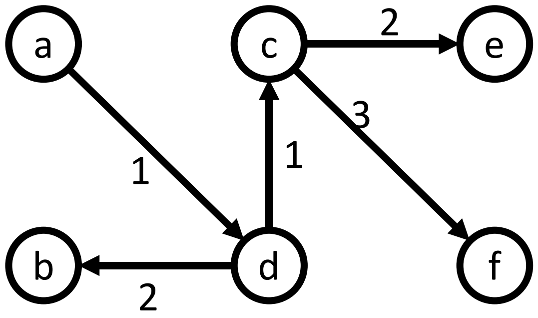

- Our complete tree is {a,d,c,b,e,f}

**function** $\textsc{Dijkstra}(\langle V, E\rangle, v_0)$
    **for** each $v \in V$ **do**
        $dist[v] \leftarrow \infty$
        $prev[v] \leftarrow nil$
    $dist[v_0] \leftarrow 0$
    $Q \leftarrow \textsc{InitPriorityQueue}(V)$
    **while** $Q$ is non-empty **do**
        $u \leftarrow \textsc{EjectMin}(Q)$
        **for** each $(u, w) \in E$ **do**
            **if** $dist[u] + weight(u, w) < dist[w]$ **then**
                $dist[w] \leftarrow dist[u] + weight(u, w)$
                $prev[w] \leftarrow u$
                $\textsc{Update}(Q, w, dist[w])$

| Covered | | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|
| | cost | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| | cost | 0 | ∞ | ∞ | ∞ | ∞ | ∞ |
| | prev | nil | nil | nil | nil | nil | nil |
| a | cost | | ∞ | 4 | 1 | ∞ | ∞ |
| | prev | | nil | a | a | nil | nil |
| a,d | cost | | 3 | 2 | | 5 | 6 |
| | prev | | d | d | | d | d |
| a,d,c | cost | | 3 | | | 4 | 5 |
| | prev | | d | | | c | c |
| a,d,c,b | cost | | | | | 4 | 5 |
| | prev | | | | | c | c |
| a,d,c,b,e | cost | | | | | | 5 |
| | prev | | | | | | c |
| a,d,c,b,e,f | cost | | | | | | |
| | prev | | | | | | |

# Tracing paths

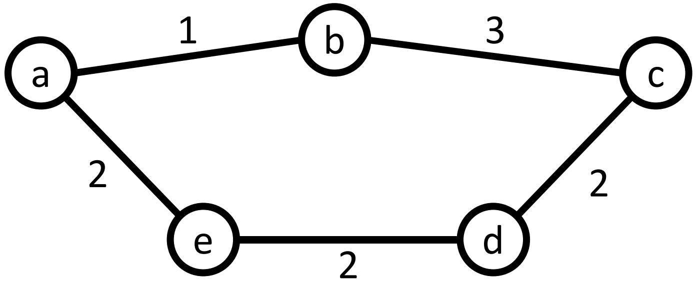
- The array `prev` is not really needed, unless we want to retrace the shortest paths from node `a`




- This tree is referred as the **shortest-path tree**

# Spanning trees and shortest-path trees

- The shortest-path tree that results from Dijkstra’s algorithm is very similar to the minimum spaning tree.




- Exercise:
  - Assume that you started from node a.
  - Which edge is missing in the minimal spanning tree?
  - Which edge is missing from the shortest-path tree?

# Next lecture

- Huffman trees and codes (Levitin Section 9.4)