COMP90038

Algorithms and Complexity

Lecture 20: Greedy Algorithms and Dijkstra

(with thanks to Harald Søndergaard and Michael Kirley)

Andres Munoz-Acosta

munoz.m@unimelb.edu.au

Peter Hall Building G.83

Recap

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

A practice challenge

- Can you solve the problem in the figure?
  - W = 15
  - w = [1 1 2 4 12]
  - v = [1 2 2 10 4]
- Because it is a larger instance, **memoing** is preferable.
  - How many states do we need to evaluate?
- FYI the answer is $15 {1,2,3,4}

The table

		j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
w	*v*	*i*																
		0																
1	1	1		**1**	**1**							**1**	**1**	**1**	**1**	**1**	**1**	**1**
1	2	2		**2**	-1	**3**	-1	-1	-1			**3**	-1	**3**	-1	**3**	-1	**3**
2	2	3		-1	-1	**4**	-1	-1	-1			-1	-1	**5**	-1	-1	-1	**5**
4	10	4		-1	-1	**4**	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	**15**
12	4	5		-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	**15**

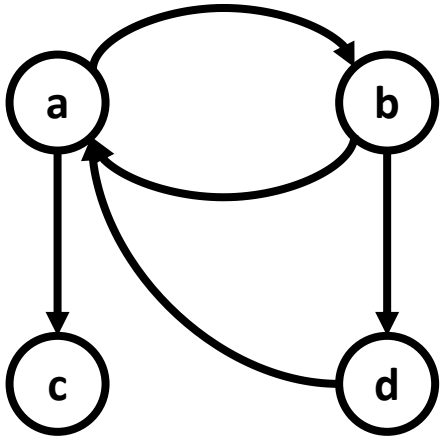
- We know that we include all the elements up to 4 because the last column (15) is the cumulative sum of the values.

Warshall's algorithm

- Warshall's algorithm computes the **transitive closure** of a directed graph.
  - An edge (*a*,*d*) is in the transitive closure of graph *G* iff there is a path in *G* from *a* to *d*.

- **Is there a path** from node *i* to node *j* using [1 … *k*] as "stepping stones"?

- Such path will exist if and only if we can:
  - step from *i* to *j* using only nodes [1 … *k*-1], or
  - step from *i* to *k* using only nodes [1 … *k*-1], and then step from *k* to *j* using only nodes [1 … *k*-1].



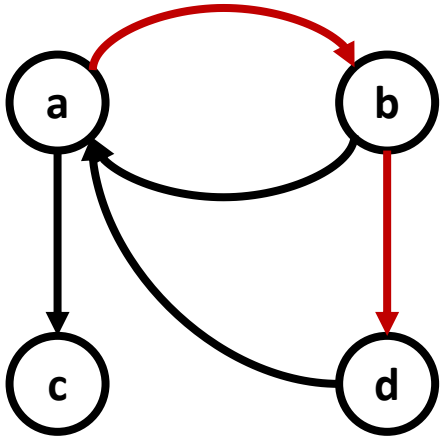
$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Warshall's algorithm

- Warshall's algorithm computes the **transitive closure** of a directed graph.
  - An edge (*a*,*d*) is in the transitive closure of graph *G* iff there is a path in *G* from *a* to *d*.

- **Is there a path** from node *i* to node *j* using [1 … *k*] as "stepping stones"?

- Such path will exist if and only if we can:
  - step from *i* to *j* using only nodes [1 … *k*-1], or
  - step from *i* to *k* using only nodes [1 … *k*-1], and then step from *k* to *j* using only nodes [1 … *k*-1].



$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Warshall's Algorithm

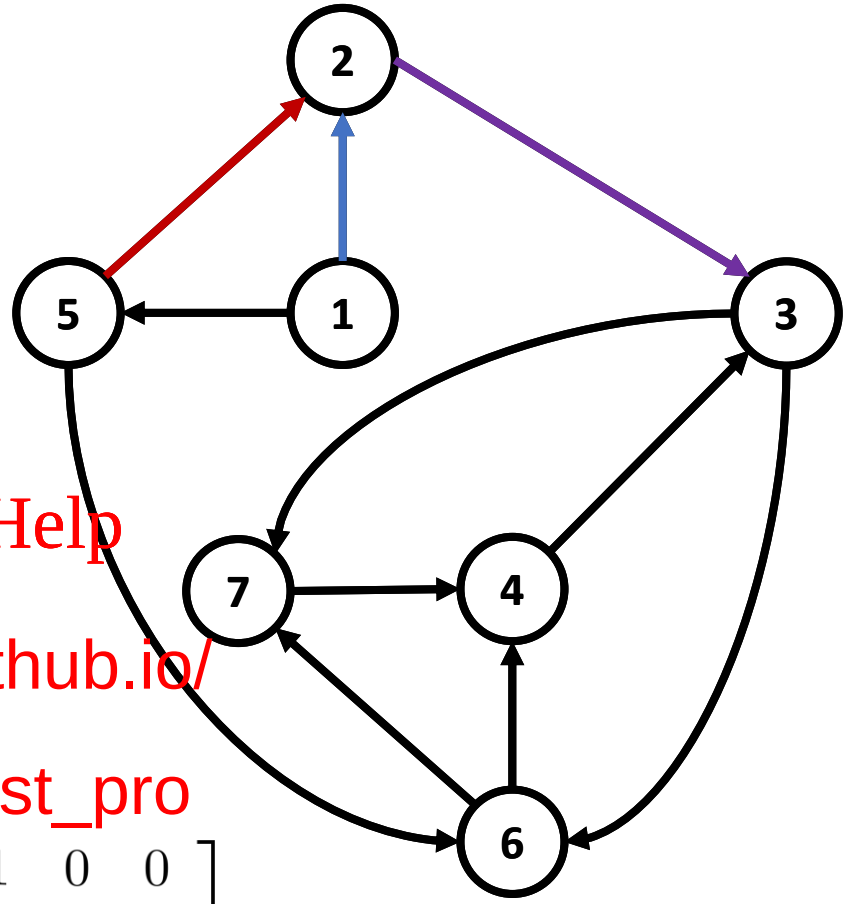
- If *G*'s adjacency matrix is *A* then we can express the recurrence relation as:

- We examined the simplest version with n.

```
for k ← 1 to n do
    for i ← 1 to n do
        if A[i, k] then
            for j ← 1 to n do
                if A[k, j] then
                    A[i, j] ← 1
```

Warshall's Algorithm

- Let's visualize the steps.



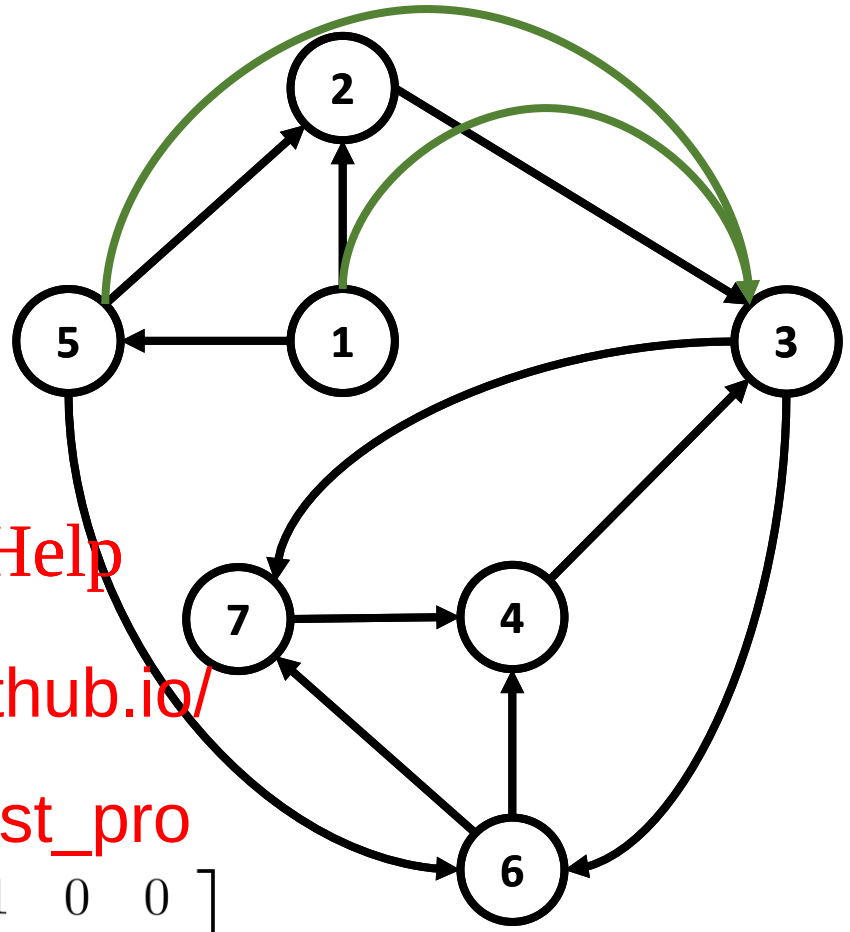
- Using node 2 (k=2), we reach node 3 from no 5.

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Warshall's Algorithm

- Let's visualize the steps.

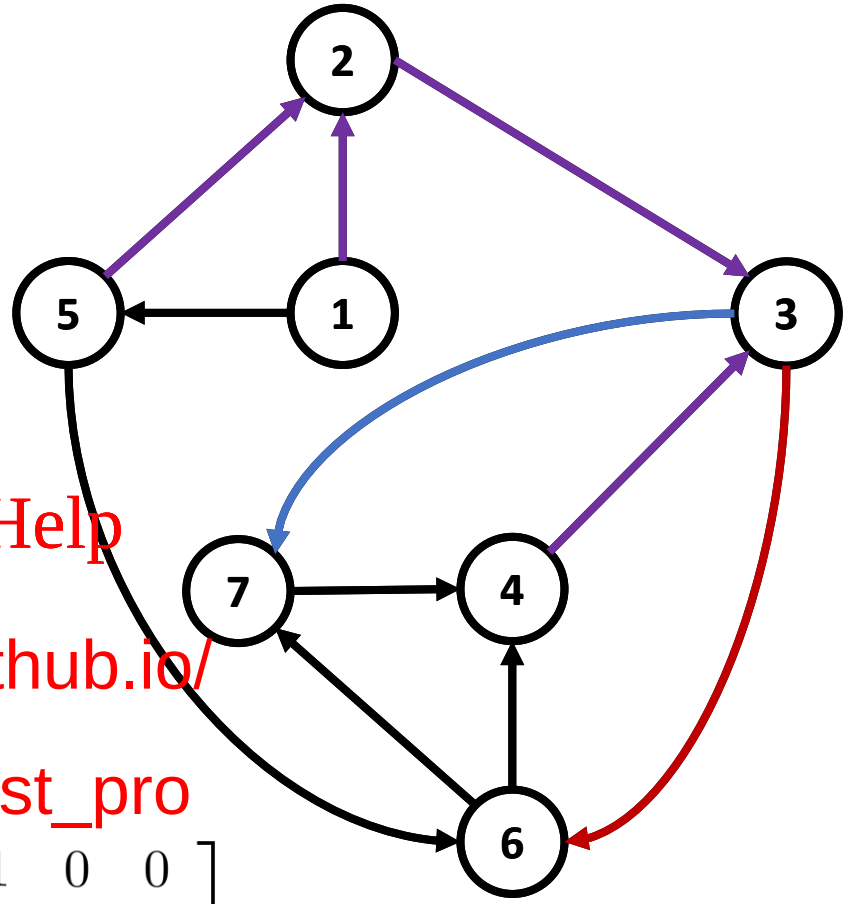
- Using node 2 (*k*=2), we reach node 3 from no 5.



$$\begin{bmatrix} 0 & 1 & \boxed{1} & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & \boxed{1} & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Warshall's Algorithm

- Let's visualize the steps.



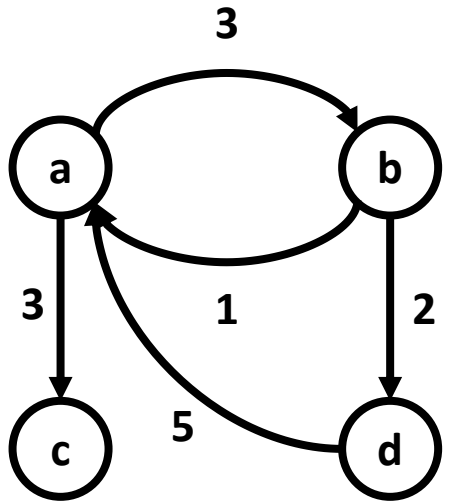
- Using node 2 (*k*=2), we reach node 3 from no 5.

- Using node 3 (k=3) we can reach: Nodes [6 7] from nodes [1,2,5]

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Floyd's Algorithm

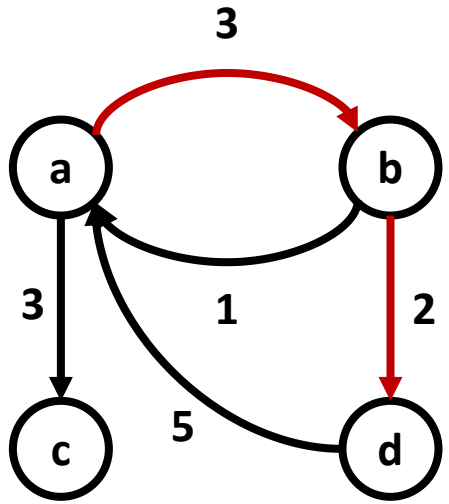
- Floyd's algorithm solves the **all-pairs shortest-path** problem for weighted graphs with **positive weights**.
  - It works for **directed** as well as **undirected** graphs.
- **What is the shortest pa** ing nodes [1 … *k*] as "stepping stones"?
- Such path will exist if and only if we can:
  - step from *i* to *j* using only nodes [1 … *k*-1], or
  - step from *i* to *k* using only nodes [1 … *k*-1], and then step from *k* to *j* using only nodes [1 … *k*-1].



$$\begin{bmatrix} \infty & 3 & 3 & \infty \\ 1 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty \\ 5 & \infty & \infty & \infty \end{bmatrix}$$

Floyd's Algorithm

- Floyd's algorithm solves the **all-pairs shortest-path** problem for weighted graphs with **positive weights**.
  - It works for **directed** as well as **undirected** graphs.
- **What is the shortest pa** ing nodes [1 … *k*] as "stepping stones"?
- Such path will exist if and only if we can:
  - step from *i* to *j* using only nodes [1 … *k*-1], or
  - step from *i* to *k* using only nodes [1 … *k*-1], and then step from *k* to *j* using only nodes [1 … *k*-1].



$$\begin{bmatrix} \infty & 3 & 3 & \boxed{5} \\ 1 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty \\ 5 & \infty & \infty & \infty \end{bmatrix}$$

Floyd's Algorithm

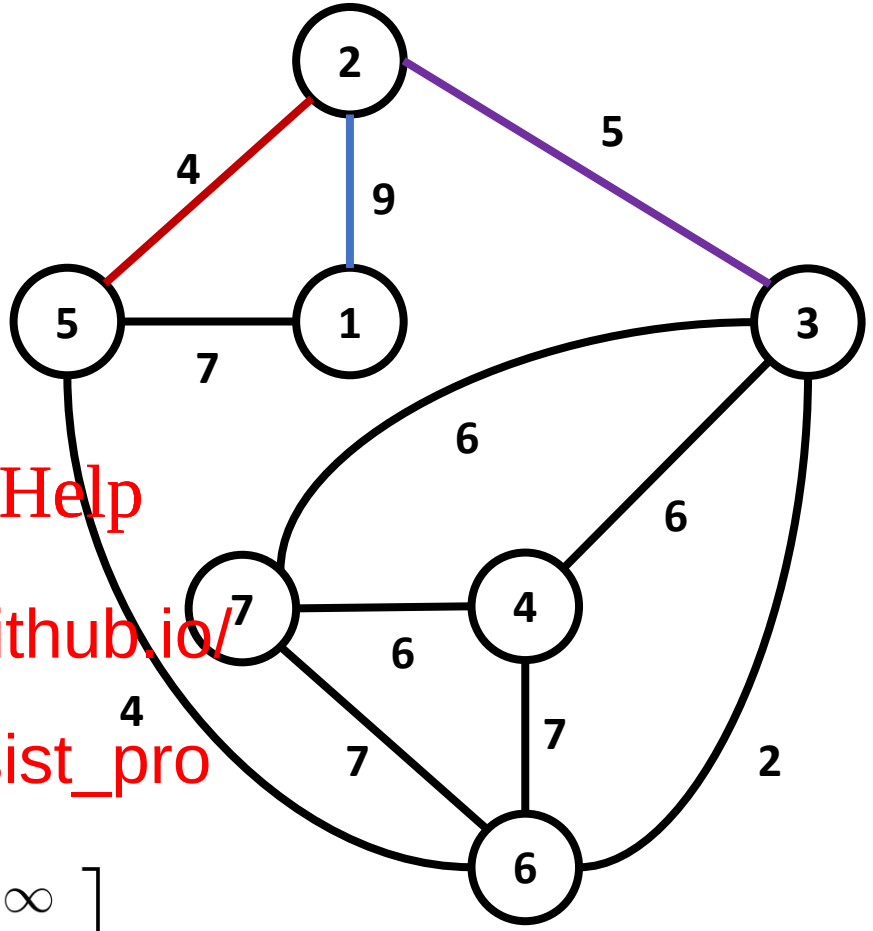
- If *G*'s weight matrix is *W* then we can express the recurrence relation as:

- A simpler version updating D:

```
function FLOYD(W[·,·], n)
    D ← W
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                D[i,j] ← min(D[i,j], D[i,k] + D[k,j])
    return D
```

Floyd's Algorithm

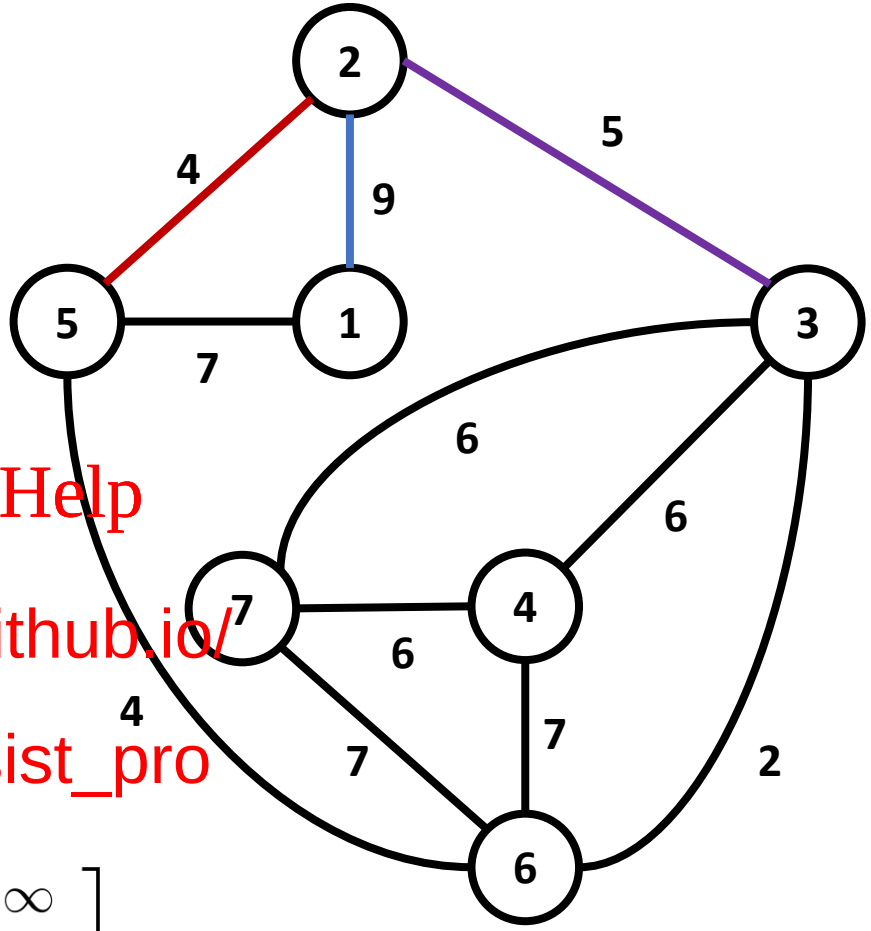
- For *k*=2
  - We can go 1 → 2 → 3, the distance 1 → 3 is 9 + 5 = 14
  - We can go 5 → 2 → 3, of 5 → 3 is 4 + 5 = 9



$$\begin{bmatrix} 0 & 9 & \infty & \infty & 7 & \infty & \infty \\ 9 & 0 & 5 & \infty & 4 & \infty & \infty \\ \infty & 5 & 0 & 6 & \infty & 2 & 6 \\ \infty & \infty & 6 & 0 & \infty & 7 & 6 \\ 7 & 4 & \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 7 & 4 & 0 & 7 \\ \infty & \infty & 6 & 6 & \infty & 7 & 0 \end{bmatrix}$$

Floyd's Algorithm

- For *k*=2
  - We can go 1 → 2 → 3, the distance 1 → 3 is 9 + 5 = 14
  - We can go 5 → 2 → 3, of 5 → 3 is 4 + 5 = 9
- The distance matrix gets updated to:



$$\begin{bmatrix} 0 & 9 & 14 & \infty & 7 & \infty & \infty \\ 9 & 0 & 5 & \infty & 4 & \infty & \infty \\ 14 & 5 & 0 & 6 & 9 & 2 & 6 \\ \infty & \infty & 6 & 0 & \infty & 7 & 6 \\ 7 & 4 & 9 & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 7 & 4 & 0 & 7 \\ \infty & \infty & 6 & 6 & \infty & 7 & 0 \end{bmatrix}$$

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 t t number of coins.
  - If we assume coin denominations of {2 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).

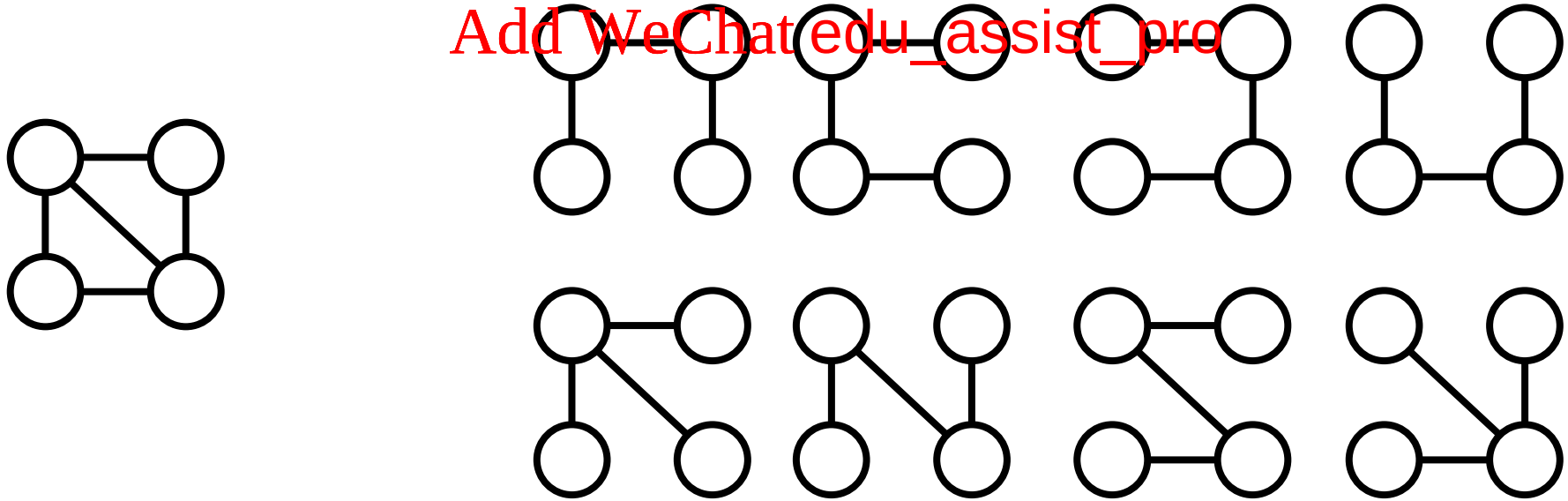
Greedy Algorithms

- 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 proble ve as 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**

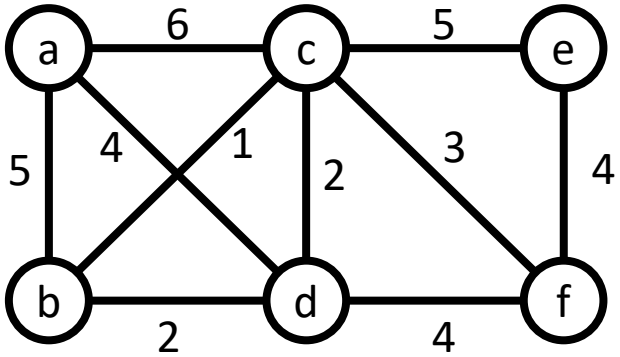
What is an Spanning Tree?

- 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 grap different spanning trees:



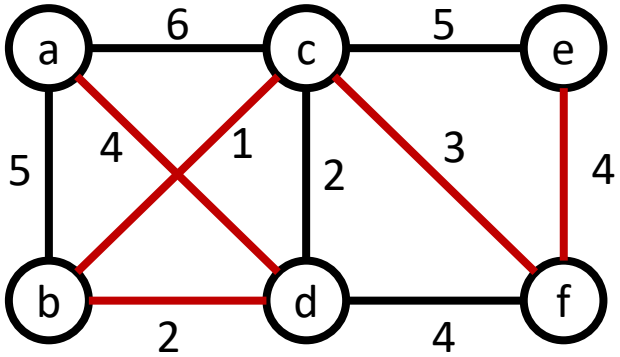
Minimum Spanning Trees of Weighted Graphs

- 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 fi ith the smallest possible cost.
  - Such tree is a **minimum spanning tree** for



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 fi ith the smallest possible cost.
  - Such tree is a **minimum spanning tree** for



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 in it.**

- A simple version:

Prim's Algorithm

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

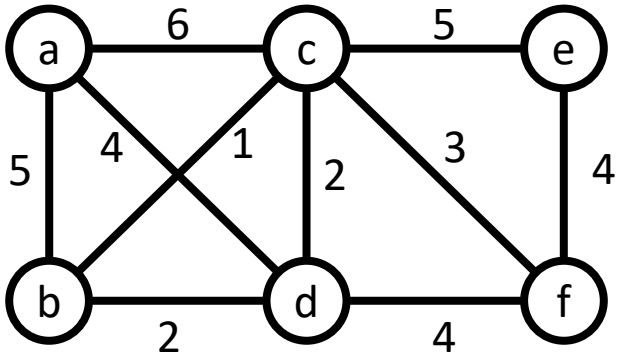
- A standard way to do t odes that are not yet included in the spanni **queue**, organised in a **min-heap** by edge **cost**.

- The information about which nodes are connected in *T* can be captured by an array *prev* of nodes, indexed by *V*. Namely, when (*v*,*u*) is included, this is captured by setting *prev*[*u*] = v.

Prim's Algorithm

- The complete algorithm is:

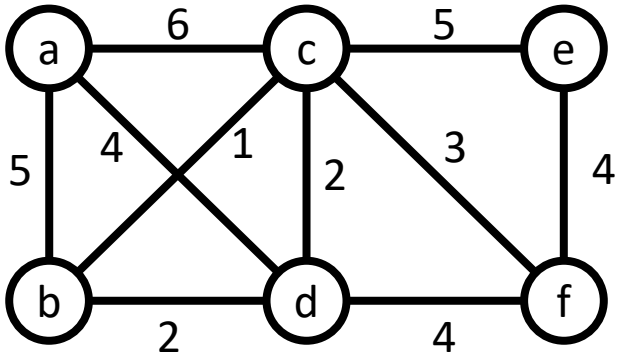
Prim's Algorithm



- On the first loop, we only create the table

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

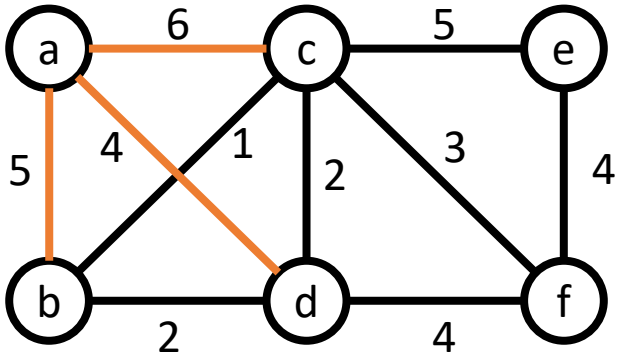
Prim's Algorithm



- Then we pick the first node as the initial one

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

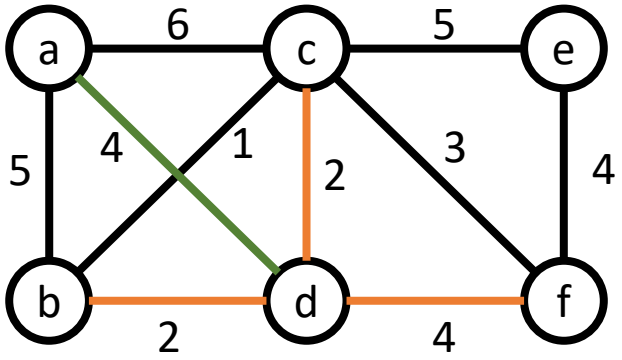
Prim's Algorithm



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

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
	cost		5	6	4	∞	∞
	prev		a	a	a	nil	nil

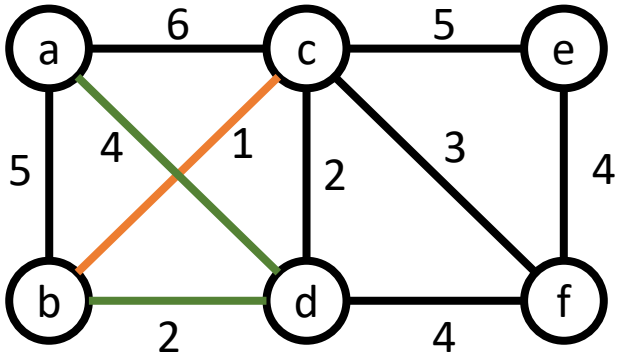
Prim's Algorithm



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

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
	cost		5	6	4	∞	∞
	prev		a	a	a	nil	nil
a,d	cost		2	2		∞	4
	prev		d	d		nil	d

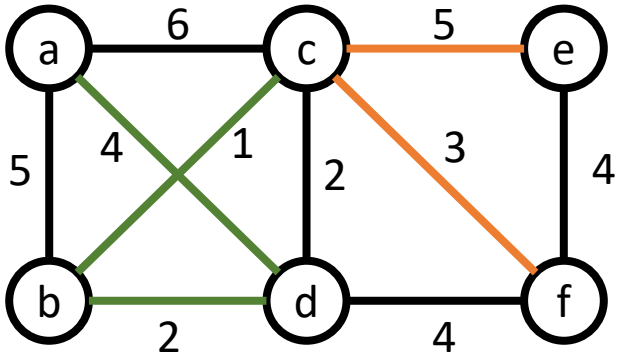
Prim's Algorithm



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

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
	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

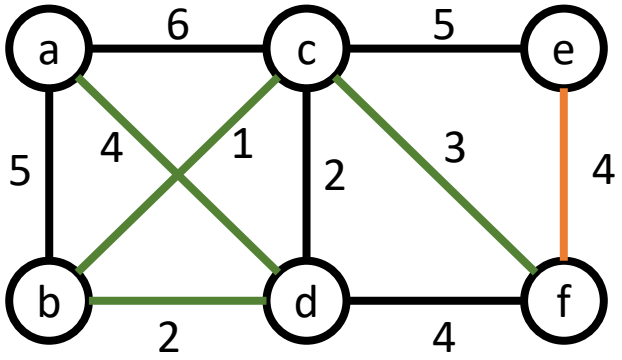
Prim's Algorithm



- We now update f

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
	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**

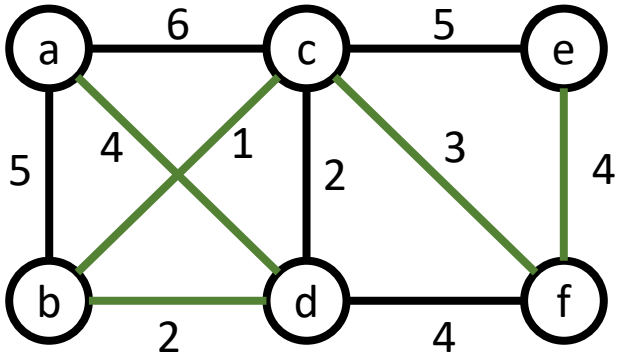
Prim's Algorithm



- We reach the last choice

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
	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}

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
	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 ap for the priority queue, we perform $|V|$ - 1 heap deletions (ea (log $|V|$)) and $|E|$ updates of priorities (each at cost O(lo

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

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

Dijkstra's Algorithm

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

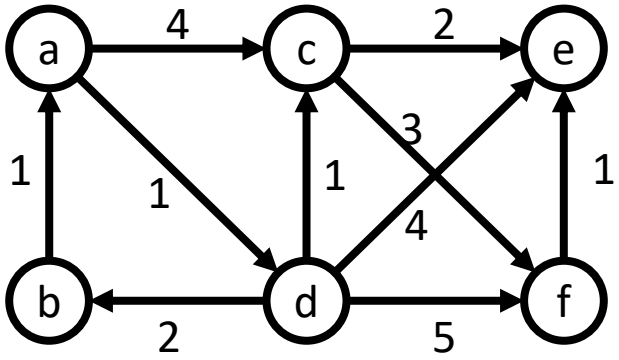
- Recall that Floyd's alg rtest paths, **for every pair of nodes**, in a (directe ted graph. It assumed an adjacency matrix representation an plexity $O(|V|^3)$.

- **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:

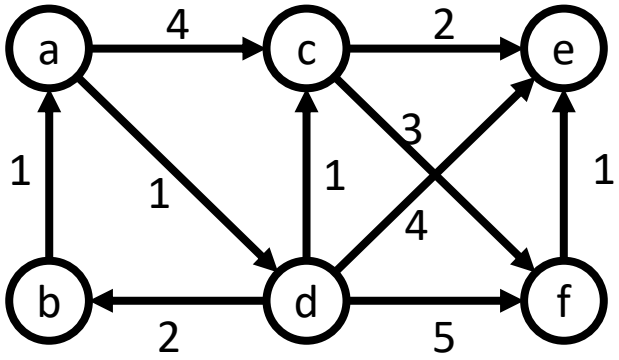
Dijkstra's Algorithm



- On the first loop, we only create the table

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

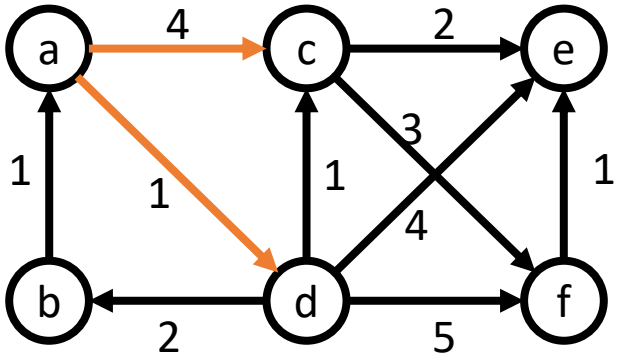
Dijkstra's Algorithm



- Then we pick the first node as the initial one

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

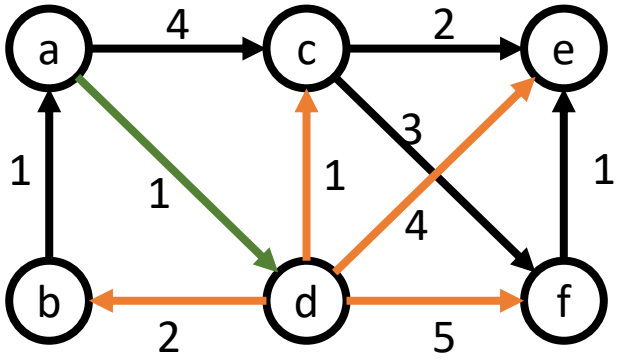
Dijkstra's Algorithm



- Then we pick the first node as the initial one

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

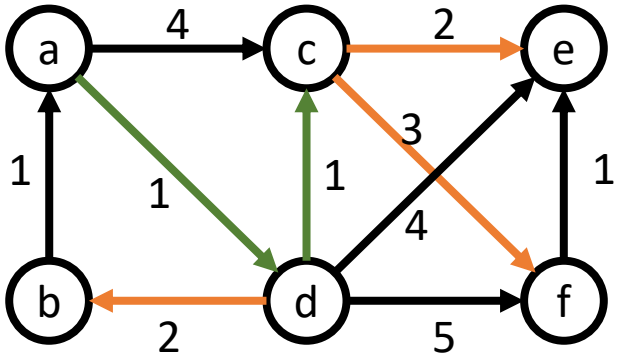
Dijkstra's Algorithm



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

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

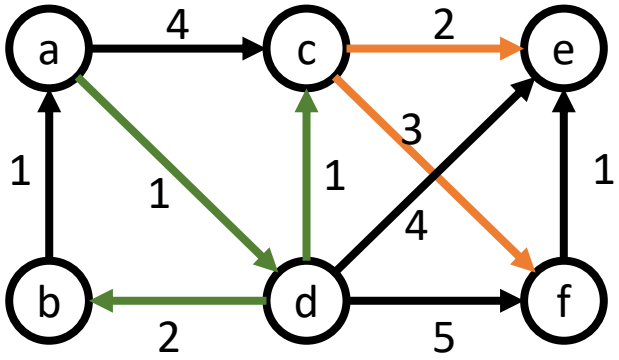
Dijkstra's Algorithm



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

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

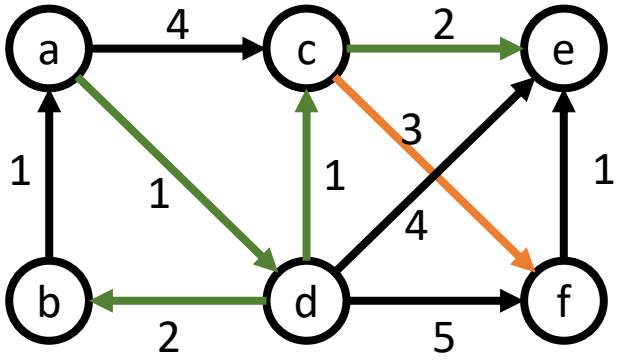
Dijkstra's Algorithm



- Now, we continue evaluating from (c)

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

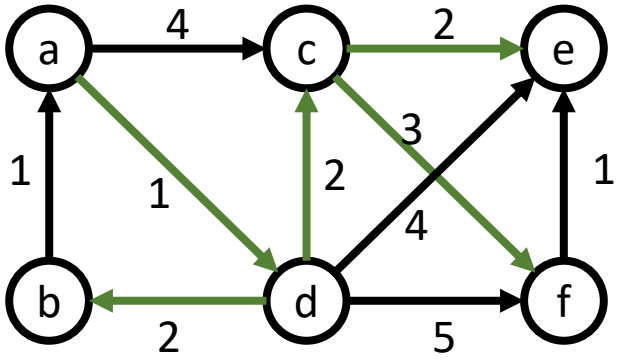
Dijkstra's Algorithm



- We arrive at our last decision.

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

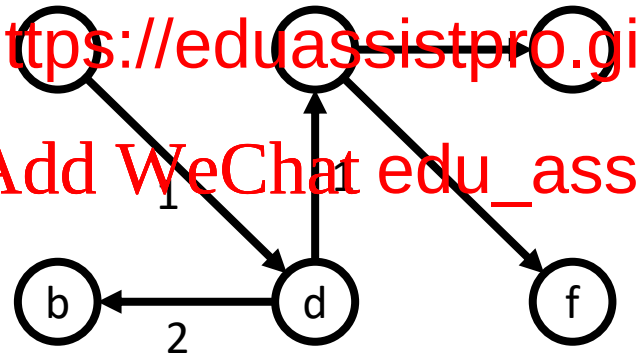


- Our complete tree is {a,d,c,b,e,f}

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

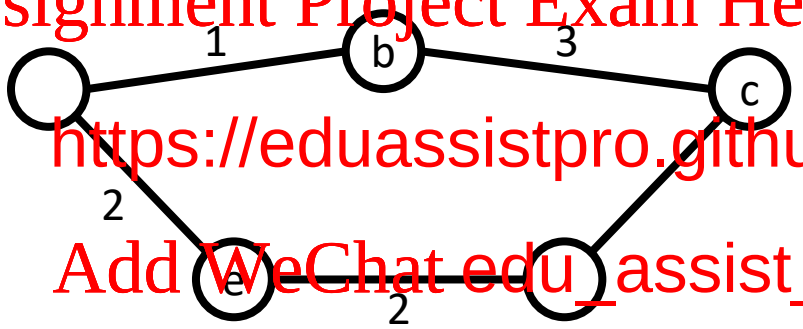
- The array `prev` is not really needed, unless we want to retrace the shortest paths from node *a*



- 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 minima spaning tree.



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

Next lecture

- We will have a look to data compression