COMP20007 Design of Algorithms

Graph Traversal

Lars Kulik

Lecture 7

Semester 1, 2023

Breadth-First and Depth-First Traversal

There are two natural approaches to the traversal of a graph.

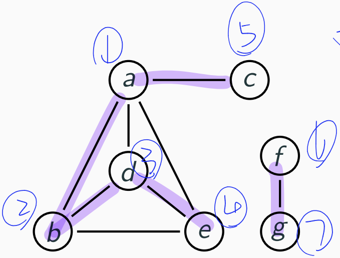
Suppose we have a graph and we want to explore all its nodes systematically. Suppose we start from node v and v has neighbouring nodes x, y and z.

In a breadth-first approach we, roughly, explore x, y and z before exploring any of their neighboring nodes.

In a depth-first approach, we may explore, say, x first, but then, before exploring y and z, we first explore one of x's neighbours, then one of its neighbours, and so on.

(This is really hard to express in English—we do need pseudo-code!)

Depth-First Search



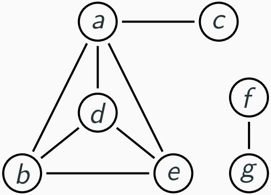
Both graph traversal methods rely on marking nodes as they are visited—so that we can avoid revisiting nodes.

Depth-first search is based on backtracking.

Neighbouring nodes are considered in, say, alphabetical order.

For the example graph, nodes are visited in the order a, b, d, e, c, f, g.

Depth-First Search: The Traversal Stack

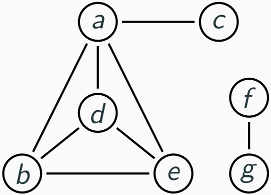


DFS corresponds to using a stack discipline for keeping track of where we are in the overall process.

Here is how the “where-we-came-from” stack develops for the example:

			e									
		d	*d*	*d*								
	b	*b*	*b*	*b*	*b*		*c*				*g*	
a	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*		*f*	*f*	*f*
—	—	—	—	—	—	—	—	—	—	—	—	—

Depth-First Search: The Traversal Stack

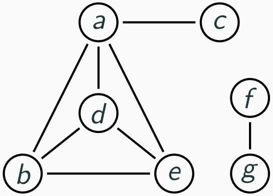


Levitin uses a more compact notation for the stack's history. Here is how the stack develops, in Levitin's notation:

$$\begin{array}{lll} e_{4,1} & & \\ d_{3,2} & & \\ b_{2,3} & c_{5,4} & g_{7,6} \\ a_{1,5} & & f_{6,7} \end{array}$$

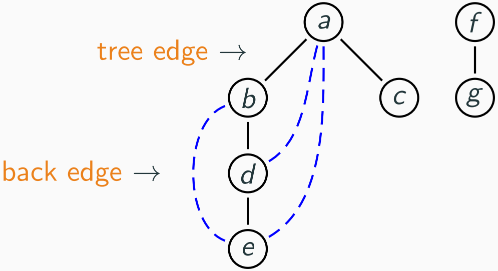
The first subscripts give the order in which nodes are pushed, the second the order in which they are popped off the stack.

Depth-First Search: The Depth-First Search Forest



Another useful tool for depicting a DF traversal is the DFS tree (for a connected graph).

More generally, we get a DFS forest:


tree edge →

back edge →

Depth-First Search: The Algorithm

```
function $\mathrm{DFS}(\langle V, E \rangle)$
    mark each node in $V$ with 0
    $count \leftarrow 0$
    for each $v$ in $V$ do
        if $v$ is marked 0 then
            $\textsc{DfsExplore}(v)$

function $\textsc{DfsExplore}(v)$
    $count \leftarrow count + 1$
    mark $v$ with $count$
    for each edge $(v, w)$ do                    ▷ $w$ is $v$'s neighbour
        if $w$ is marked with 0 then
            $\textsc{DfsExplore}(w)$
```

This works both for directed and undirected graphs.

Depth-First Search: The Algorithm

The "marking" of nodes is usually done by maintaining a separate array, `mark`, indexed by V.

For example, when we wrote "mark v with *count*", that would be implemented as "`mark[v] := count`".

How to find the nodes adjacent to v depends on the graph representation used.

Using an adjacency matrix `adj`, we need to consider `adj[v,w]` for each `w` in V. Here the complexity of graph traversal is $\Theta(|V|^2)$.

Using adjacency lists, for each v, we traverse the list `adj[v]`. In this case, the complexity of traversal is $\Theta(|V| + |E|)$. Why? ✎

Applications of Depth-First Search

It is easy to adapt the DFS algorithm so that it can decide whether a graph is connected.

How? ✎

check whether all the nodes have been marked. once the first time of DFS have been finished.

Applications of Depth-First Search

It is easy to adapt the DFS algorithm so that it can decide whether a graph is connected.

How? ✎

It is also easy to adapt it so that it can decide whether a graph has a cycle.

How? ✎

Once a node has been checked, there must exists a cycle.

Applications of Depth-First Search

It is easy to adapt the DFS algorithm so that it can decide whether a graph is connected.

How? ✎

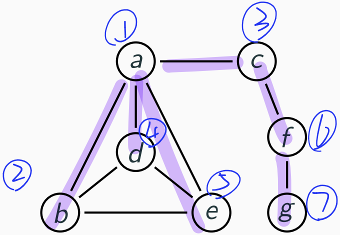
It is also easy to adapt it so that it can decide whether a graph has a cycle.

How? ✎

In terms of DFS forests, how can we tell if we have traversed a dag?

Yeap

Breadth-First Search



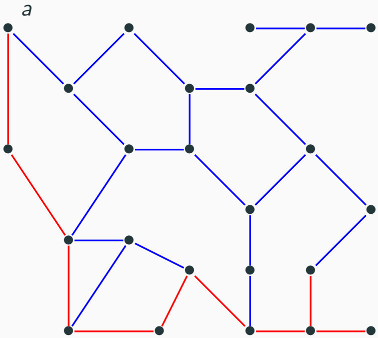
Breadth-first search proceeds in a concentric manner, visiting all nodes that are one step away from the start node, then all those that are two steps away (except those that were already visited), and so on.

Again, neighbouring nodes are considered in, say, alphabetical order.

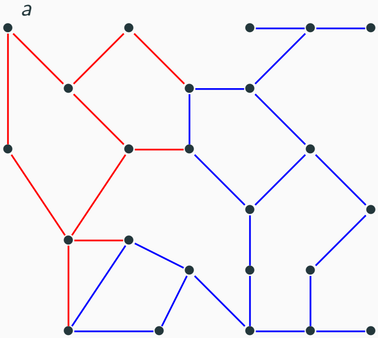
For the example graph, nodes are visited in the order a, b, c, d, e, f, g.

Depth-First Search vs Breadth-First

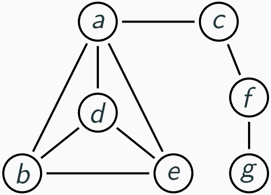
Typical depth-first search:



Typical breadth-first search:



Breadth-First Search: The Traversal Queue



BFS uses a queue discipline for keeping track of pending tasks.

How the queue develops for the example:

a_1

$b_2 \quad c_3 \quad d_4 \quad e_5$

$c_3 \quad d_4 \quad e_5$

$d_4 \quad e_5 \quad f_6$

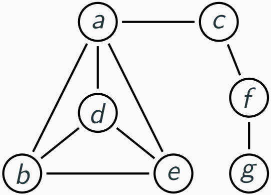
$e_5 \quad f_6$

f_6

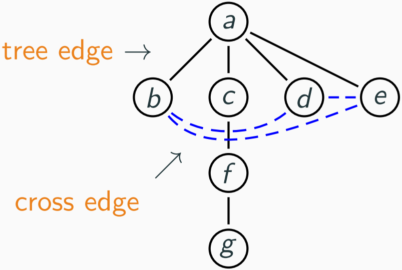
g_7

The subscript again is Levitin's; it gives the order in which nodes are processed.

The Breadth-First Search Forest



Here is the BFS tree for the example:



In general, we may get a BFS forest.

Breadth-First Search: The Algorithm

```
function $\mathrm{BFS}(\langle V, E \rangle)$
    mark each node in $V$ with 0
    $count \leftarrow 0$, $init$(queue)                    ▷ create an empty queue
    for each $v$ in $V$ do
        if $v$ is marked 0 then
            $count \leftarrow count + 1$
            mark $v$ with $count$
            $inject(queue, v)$                       ▷ queue containing just $v$
            while $queue$ is non-empty do
                $u \leftarrow eject(queue)$                  ▷ dequeues $u$
                for each edge $(u, w)$ adjacent to $u$ do
                    if $w$ is marked with 0 then
                        $count \leftarrow count + 1$
                        mark $w$ with $count$
                        $inject(queue, w)$           ▷ enqueues $w$
```

Breadth-First Search: The Algorithm

BFS has the same complexity as DFS.

Again, the same algorithm works for directed graphs as well.

Certain problems are most easily solved by adapting BFS.

For example, given a graph and two nodes, *a* and *b* in the graph, how would you find the fewest number of edges between two given vertices *a* and *b*?

✎

Topological Sorting

We mentioned scheduling problems and their representation by directed graphs.

Assume a directed edge from a to b means that task a must be completed before b can be started.

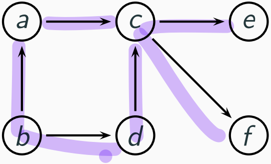
Then the graph has to be a dag.

Assume the tasks are carried out by a single person, unable to multi-task.

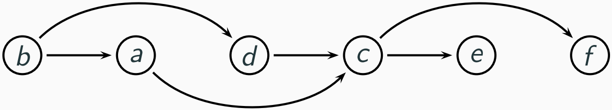
Then we should try to linearize the graph, that is, order the nodes in a sequence $v_1, v_2, \ldots, v_n$ such that for each edge $(v_i, v_j) \in E$, we have $i < j$.

Topological Sorting: Example

There are four different ways to linearize the following graph.



Here is one:



Topological Sorting Algorithm 1

We can solve the top-sort problem with depth-first search:

1. Perform DFS and note the order in which nodes are popped off the stack.
2. List the nodes in the reverse of that order.

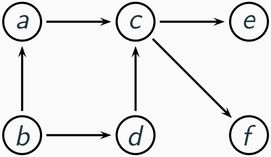
This works because of the stack discipline.

If (u, v) is an edge then it is possible (for some way of deciding ties) to arrive at a DFS stack with u sitting below v.

Taking the "reverse popping order" ensures that u is listed before v.

Topological Sorting Example Again

Using the DFS method and resolving ties by using alphabetical order, the graph gives rise to the traversal stack shown on the right (the popping order shown in red):



			e		f					
		c	c	c	c	c				
	a	a	a	a	a	a	a		d	
b	b	b	b	b	b	b	b	b	b	b

$e_{3,1}$	$f_{4,2}$	
$c_{2,3}$		$d_{6,5}$
$a_{1,4}$		$b_{5,6}$

e f c a d b ⇒ b d a c f e

Taking the nodes in reverse popping order yields b, d, a, c, f, e.

Topological Sorting Algorithm 2

An alternative method would be to repeatedly select a random source in the graph (that is, a node with no incoming edges), list it, and remove it from the graph.

This is a very natural approach, but it has the drawback that we repeatedly need to scan the graph for a source.

However, it exemplifies the general principle of decrease-and-conquer.

COMP20007 Design of Algorithms

Greedy Algorithms: Prim and Dijkstra

Lars Kulik

Lecture 8

Semester 1, 2023

Greedy Algorithms

A natural strategy to problem solving is to make decisions based on what is the locally best choice.

Suppose we have coin denominations 25, 10, 5, and 1, and we want to change 30 cents using the smallest number of coins.

In general we will want to 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. (In this case we use 25+5 cents.)

This greedy strategy will work for the given denominations, but not for, say, 25, 10, 1.

Greedy Algorithms

In general we cannot expect locally best choices to yield globally best outcomes.

However, there are some well-known algorithms that rely on the greedy approach, being both correct and fast.

In other cases, for hard problems, a greedy algorithm can sometimes serve 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

The Priority Queue

A priority queue is a set (or pool) of elements.

An element is injected into the priority queue together with a priority (often the key value itself) and elements are ejected according to priority.

As an abstract data type, the priority queue supports the following operations on a “pool” of elements (ordered by some linear order):

- find an item with maximal priority
- insert a new item with associated priority
- test whether a priority queue is empty
- eject the largest element

Stacks and Queues as Priority Queues

Special instances are obtained when we use time for priority:

- If “large” means “late” we obtain the stack.
- If “large” means “early” we obtain the queue.

Possible Implementations of the Priority Queue

Assume priority = key.

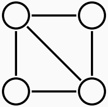
	Inject(e)	Eject()
Unsorted array or list		
Sorted array or list		

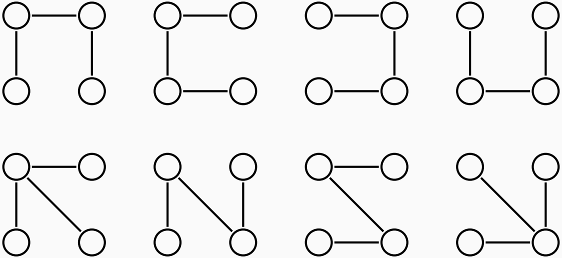
✎

Spanning Trees

Recall that a tree is a connected graph with no cycle.

A spanning tree of a graph $\langle V, E \rangle$ is a tree $\langle V, E' \rangle$ with $E' \subseteq E$.

The graph  has eight different spanning trees:

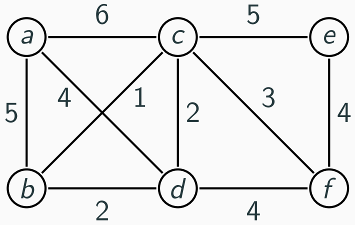


Minimum Spanning Trees of Weighted Graphs

In applications where the edges correspond to distances, or cost, some spanning trees will be more desirable than others.

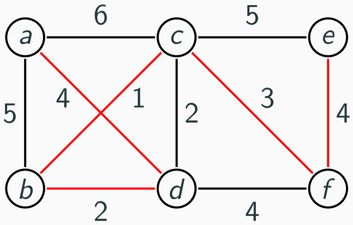
Suppose we have a set of 'stations' to connect in a network, and also some possible connections, together with the cost of each connection.

Then we have a weighted graph problem, of finding a spanning tree with the smallest possible cost.



Minimum Spanning Trees

Given a weighted graph, a sub-graph which is a tree with minimal weight is a minimum spanning tree for the graph.



Minimum Spanning Trees: Prim's Algorithm

Prim's algorithm is an example of a greedy algorithm.

It constructs a sequence of subtrees T, each adding a node together with an edge to a node in the previous subtree. In each step it picks a closest node from outside the tree and adds that. A sketch:

```
function PRIM($\langle V, E \rangle$)
    $V_T \leftarrow \{v_0\}$
    $E_T \leftarrow \emptyset$
    for $i \leftarrow 1$ to $|V| - 1$ do
        find a minimum-weight edge $(v, u) \in V_T \times (V \setminus V_T)$
        $V_T \leftarrow V_T \cup \{u\}$
        $E_T \leftarrow E_T \cup \{(v, u)\}$
    return $E_T$
```

Prim's Algorithm

Note that in each iteration, the tree grows by one edge.

Or, we can say that the tree grows to include the node from outside that has the smallest cost.

But how do we find the minimum-weight edge (v, u)?

A standard way to do this is to organise the nodes that are not yet included in the spanning tree T as a priority queue organised by edge cost.

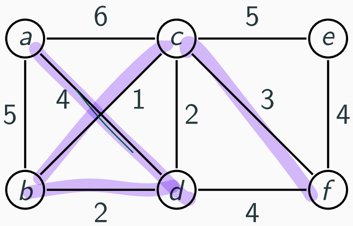
$O(\log n)$

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

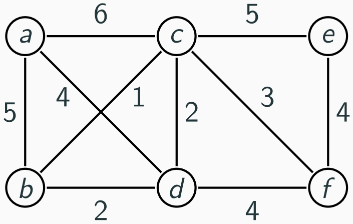
```
function PRIM($\langle V, E \rangle$)
    for each $v \in V$ do
        $cost[v] \leftarrow \infty$
        $prev[v] \leftarrow nil$
    pick initial node $v_0$
    $cost[v_0] \leftarrow 0$
    $Q \leftarrow$ INITPRIORITYQUEUE($V$)              ▷ priorities are cost values
    while $Q$ is non-empty do
        $u \leftarrow$ EJECTMIN($Q$)
        for each $(u, w) \in E$ do
            if $w \in Q$ and $weight(u, w) < cost[w]$ then
                $cost[w] \leftarrow weight(u, w)$
                $prev[w] \leftarrow u$
                UPDATE($Q, w, cost[w]$)          ▷ rearranges priority queue
```

Prim's Algorithm: Example



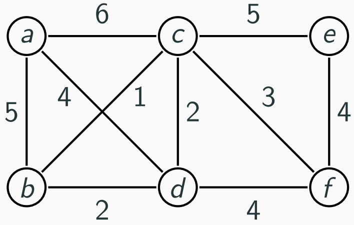
Tree T	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		5/a	6/a	4/a	∞/nil	∞/nil
a,d		2/d	2/d		∞/nil	4/d
a,b,d			1/b		∞/nil	4/d
a,b,c,d					5/c	3/c
a.b,c.d.f					4/f	
a.b,c.d,e f						

Prim's Algorithm: Example



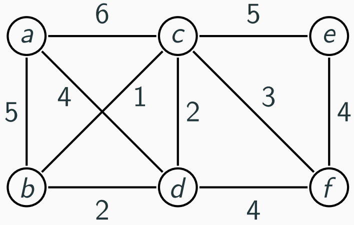
Tree T	a	b	c	d	e	f
$-$	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		$5/a$	$6/a$	$4/a$	∞/nil	∞/nil

Prim's Algorithm: Example



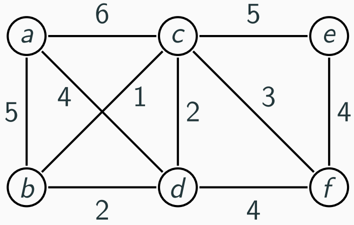
Tree T	a	b	c	d	e	f
$-$	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		$5/a$	$6/a$	$4/a$	∞/nil	∞/nil
a, d		$2/d$	$2/d$		∞/nil	$4/d$

Prim's Algorithm: Example



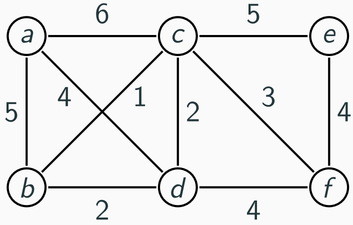
Tree T	a	b	c	d	e	f
$-$	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		$5/a$	$6/a$	$4/a$	∞/nil	∞/nil
a, d		$2/d$	$2/d$		∞/nil	$4/d$
a, d, b			$1/b$		∞/nil	$4/d$

Prim's Algorithm: Example



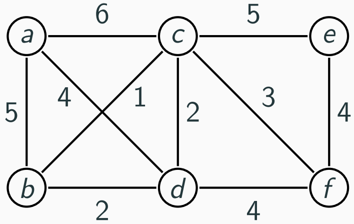
Tree T	a	b	c	d	e	f
–	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		$5/a$	$6/a$	$4/a$	∞/nil	∞/nil
a, d		$2/d$	$2/d$		∞/nil	$4/d$
a, d, b			$1/b$		∞/nil	$4/d$
a, d, b, c					$5/c$	$3/c$

Prim's Algorithm: Example



Tree T	a	b	c	d	e	f
–	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		$5/a$	$6/a$	$4/a$	∞/nil	∞/nil
a, d		$2/d$	$2/d$		∞/nil	$4/d$
a, d, b			$1/b$		∞/nil	$4/d$
a, d, b, c					$5/c$	$3/c$
a, d, b, c, f					$4/f$	

Prim's Algorithm: Example



Tree T	a	b	c	d	e	f
–	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		$5/a$	$6/a$	$4/a$	∞/nil	∞/nil
a, d		$2/d$	$2/d$		∞/nil	$4/d$
a, d, b			$1/b$		∞/nil	$4/d$
a, d, b, c					$5/c$	$3/c$
a, d, b, c, f					$4/f$	
a, d, b, c, f, e						

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 $O(|V| \cdot |E|)$. However, we are using cleverer data structures.

Using adjacency lists for the graph and a min-heap for the priority queue, we can do better! We will discuss this later.

Dijkstra's Algorithm

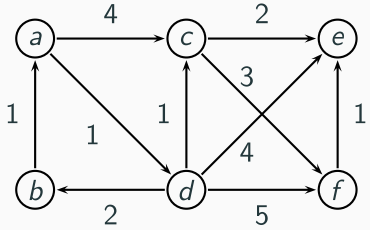
Another classical greedy weighted-graph algorithm is Dijkstra's algorithm, whose overall structure is the same as Prim's.

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

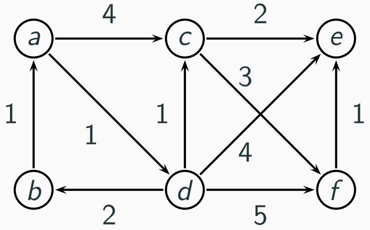
```
function DIJKSTRA($\langle V, E\rangle, v_0$)
    for each $v \in V$ do
        $dist[v] \leftarrow \infty$
        $prev[v] \leftarrow nil$
    $dist[v_0] \leftarrow 0$
    $Q \leftarrow$ INITPRIORITYQUEUE($V$)                ▷ priorities are distances
    while $Q$ is non-empty do
        $u \leftarrow$ EJECTMIN($Q$)
        for each $(u, w) \in E$ do
            if $w \in Q$ and $dist[u] + weight(u, w) < dist[w]$ then
                $dist[w] \leftarrow dist[u] + weight(u, w)$
                $prev[w] \leftarrow u$
                UPDATE($Q, w, dist[w]$)          ▷ rearranges priority queue
```

Dijkstra's Algorithm: Example



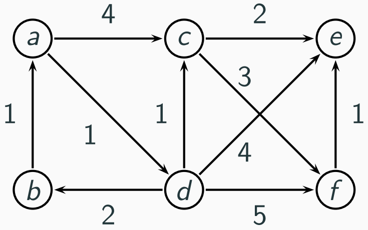
Covered	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil

Dijkstra's Algorithm: Example



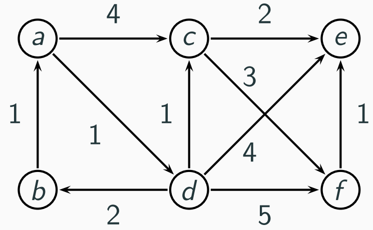
Covered	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		∞/nil	$4/a$	$1/a$	∞/nil	∞/nil

Dijkstra's Algorithm: Example



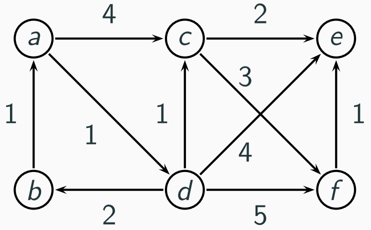
Covered	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		∞/nil	$4/a$	$1/a$	∞/nil	∞/nil
a, d		$3/d$	$2/d$		$5/d$	$6/d$

Dijkstra's Algorithm: Example



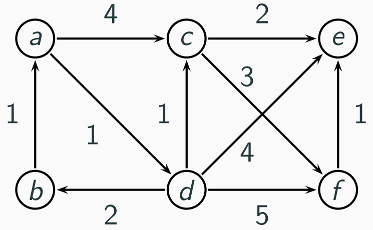
Covered	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		∞/nil	$4/a$	$1/a$	∞/nil	∞/nil
a, d		$3/d$	$2/d$		$5/d$	$6/d$
a, d, c		$3/d$			$4/c$	$5/c$

Dijkstra's Algorithm: Example



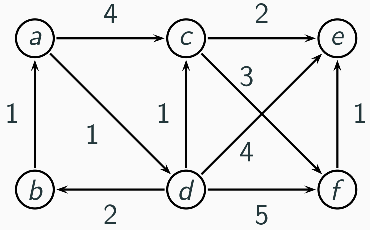
Covered	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		∞/nil	$4/a$	$1/a$	∞/nil	∞/nil
a, d		$3/d$	$2/d$		$5/d$	$6/d$
a, d, c		$3/d$			$4/c$	$5/c$
a, d, c, b					$4/c$	$5/c$

Dijkstra's Algorithm: Example



Covered	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		∞/nil	$4/a$	$1/a$	∞/nil	∞/nil
a, d		$3/d$	$2/d$		$5/d$	$6/d$
a, d, c		$3/d$			$4/c$	$5/c$
a, d, c, b					$4/c$	$5/c$
a, d, c, b, e						$5/c$

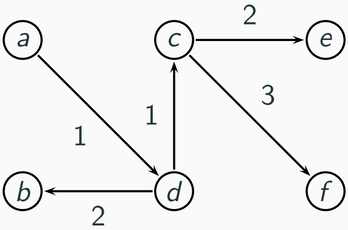
Dijkstra's Algorithm: Example



Covered	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		∞/nil	$4/a$	$1/a$	∞/nil	∞/nil
a, d		$3/d$	$2/d$		$5/d$	$6/d$
a, d, c		$3/d$			$4/c$	$5/c$
a, d, c, b					$4/c$	$5/c$
a, d, c, b, e						$5/c$
a, d, c, b, e, f						

Dijkstra's Algorithm: Tracing Paths

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



Negative Weights

In our example, we used positive weights, and for a good reason: Dijkstra's algorithm may not work otherwise!

In this example, the greedy pick—choosing the edge from *a* to *b*—is clearly the wrong one.