Algorithms Week 9

Ljubomir Perković, DePaul University

Vertex u can reach another vertex v in a directed graph G if G contains a directed path from u to v.

Vertex u can *reach* another vertex v in a directed graph G if G contains a directed path from u to v.

Let $reach(u)$ denote the set of all vertices in G that u can reach.

Vertex u can reach another vertex v in a directed graph G if G contains a directed path from u to v.

Let $reach(u)$ denote the set of all vertices in G that u can reach.

Two vertices u and v are strongly connected if u can reach v and v can reach u.

Vertex u can reach another vertex v in a directed graph G if G contains a directed path from u to v.

Let $reach(u)$ denote the set of all vertices in G that u can reach.

Two vertices u and v are strongly connected if u can reach v and v can reach u.

A directed graph is strongly connected if and only if every pair of vertices is strongly connected.

A strongly connected component of a directed graph G is a maximal strongly connected subgraph of G.

A strongly connected component of a directed graph G is a maximal strongly connected subgraph of G.

Some special cases:

A strongly connected component of a directed graph G is a maximal strongly connected subgraph of G.

Some special cases:

- A directed graph G is strongly connected if and only if G has exactly one strongly connected component

A strongly connected component of a directed graph G is a maximal strongly connected subgraph of G.

Some special cases:

- A directed graph G is strongly connected if and only if G has exactly one strongly connected component

- G is a dag if and only if every strongly connected component of G consists of a single vertex.

A strongly connected component of a directed graph G is a maximal strongly connected subgraph of G.

Some special cases:

- A directed graph G is strongly connected if and only if G has exactly one strongly connected component

- G is a dag if and only if every strongly connected component of G consists of a single vertex.

The strongly connected component graph $scc(G)$ is another directed graph obtained from G by contracting each strongly connected component to a single vertex and collapsing parallel edges.

A strongly connected component of a directed graph G is a maximal strongly connected subgraph of G.

Some special cases:

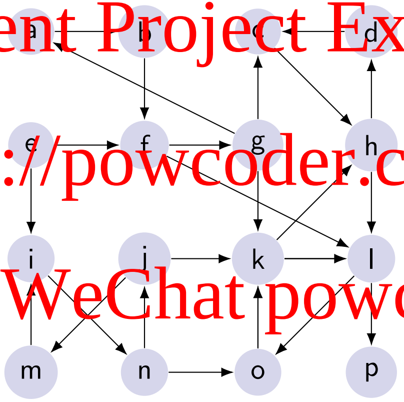
- A directed graph G is strongly connected if and only if G has exactly one strongly connected component

- G is a dag if and only if every strongly connected component of G consists of a single vertex.

The strongly connected component graph $scc(G)$ is another directed graph obtained from G by contracting each strongly connected component to a single vertex and collapsing parallel edges.

$scc(G)$ is always a dag.

Strongly connected components



Finding all strongly connected components of a directed graph

To find the strongly connected component that a vertex v is part of:

- First we compute $reach(v)$ via whatever-first search
- Then compute $reach^{-1}(v) = \{u | v \in reach(u)\}$ by searching, from v, the directed graph obtained by reversing the direction of edges of G.

The strongly connected component of v is $reach(v) \cap reach^{-1}(v)$.

Finding all strongly connected components of a directed graph

To find the strongly connected component that a vertex v is part of:

- First we compute $reach(v)$ via whatever-first search
- Then compute $reach^{-1}(v) = \{u | v \in reach(u)\}$ by searching, from v, the directed graph obtained by reversing the direction of edges of G.

The strongly connected component of v is $reach(v) \cap reach^{-1}(v)$.

Running time: $O(V + E)$

Finding all strongly connected components of a directed graph

To find the strongly connected component that a vertex v is part of:

- First we compute $\mathit{reach}(v)$ via whatever-first search
- Then compute $\mathit{reach}^{-1}(v) = \{u | v \in \mathit{reach}(u)\}$ by searching, from v, the directed graph obtained by reversing the direction of edges of G.

The strongly connected component of v is $\mathit{reach}(v) \cap \mathit{reach}^{-1}(v)$.

Running time: $O(V + E)$

To find all strongly connected components in a directed graph we can repeat the above using the standard DFS wrapper function.

Finding all strongly connected components of a directed graph

To find the strongly connected component that a vertex v is part of:

- First we compute $reach(v)$ via whatever-first search
- Then compute $reach^{-1}(v) = \{u | v \in reach(u)\}$ by searching, from v, the directed graph obtained by reversing the direction of edges of G.

The strongly connected component of v is $reach(v) \cap reach^{-1}(v)$.

Running time: $O(V + E)$

To find all strongly connected components in a directed graph we can repeat the above using the standard DFS wrapper function. However, the resulting algorithm runs in $O(VE)$ time: there are at most V strong components, and each requires $O(E)$ time to discover.

Algorithms to compute all strongly connected components in $O(V + E)$ time rely on the following observation:

Lemma

Fix a depth-first traversal of directed graph G. Each strongly connected component C of G contains exactly one node that does not have a parent in C.

Algorithms to compute all strongly connected components in $O(V + E)$ time rely on the following observation:

Lemma

Fix a depth-first traversal of directed graph G. Each strongly connected component C of G contains exactly one node that does not have a parent in C.

In other words, either this node has a parent in another strongly connected component, or it has no parent.

Strongly connected components in linear time

Algorithms to compute all strongly connected components in $O(V+E)$ time rely on the following observation:

Lemma

Fix a depth-first traversal of directed graph G. Each strongly connected component C of G contains exactly one node that does not have a parent in C.

In other words, either this node has a parent in another strongly connected component, or it has no parent.

The lemma implies that each strongly connected component of a directed graph G defines a connected subtree of any depth-first forest of G.

Algorithms to compute all strongly connected components in $O(V + E)$ time rely on the following observation:

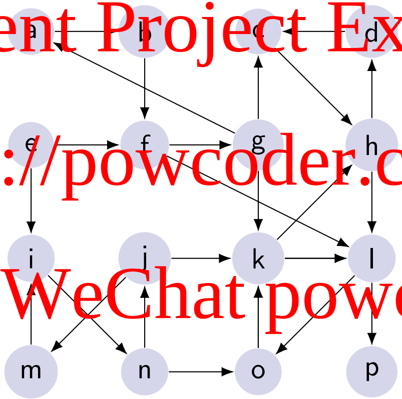
Lemma

Fix a depth-first traversal of directed graph G. Each strongly connected component C of G contains exactly one node that does not have a parent in C.

In other words, either this node has a parent in another strongly connected component, or it has no parent.

The lemma implies that each strongly connected component of a directed graph G defines a connected subtree of any depth-first forest of G. In particular, for any strongly connected component C, the vertex in C with the earliest preorder time is the lowest common ancestor of all vertices in C; we call this vertex the root of C.

Strongly connected components in linear time



Let C be any strongly connected component of G that is a sink component, defined as a component in which the reach of any vertex in C is precisely C.

Let C be any strongly connected component of G that is a sink component, defined as a component in which the reach of any vertex in C is precisely C.

We can find all the strongly connected components in G by repeatedly:

Let C be any strongly connected component of G that is a sink component, defined as a component in which the reach of any vertex in C is precisely C.

We can find all the strongly connected components in G by repeatedly:

1. finding a vertex v in some sink component (somehow),

Strongly connected components in linear time

Let C be any strongly connected component of G that is a sink component, defined as a component in which the reach of any vertex in C is precisely C.

We can find all the strongly connected components in G by repeatedly:

1. finding a vertex v in some sink component (somehow),
2. finding the vertices reachable from v, and

Let C be any strongly connected component of G that is a sink component, defined as a component in which the reach of any vertex in C is precisely C.

We can find all the strongly connected components in G by repeatedly:

1. finding a vertex v in some sink component (somehow),
2. finding the vertices reachable from v, and
3. removing that sink component from the input graph,

Strongly connected components in linear time

Let C be any strongly connected component of G that is a sink component, defined as a component in which the reach of any vertex in C is precisely C.

We can find all the strongly connected components in G by repeatedly:

1. finding a vertex v in some sink component (somehow),
2. finding the vertices reachable from v, and
3. removing that sink component from the input graph,

until no vertices remain.

Strongly connected components in linear time


```
StrongComponents(G):
  count ← 0
  while G is non-empty
    C ← ∅
    count ← count + 1
    v ← any vertex in a sink component of G
    for all vertices w in reach(v):
      w.label ← count
      add w to C
    remove C and its incoming edges from G
```

How to find a vertex in a sink component?

Lemma

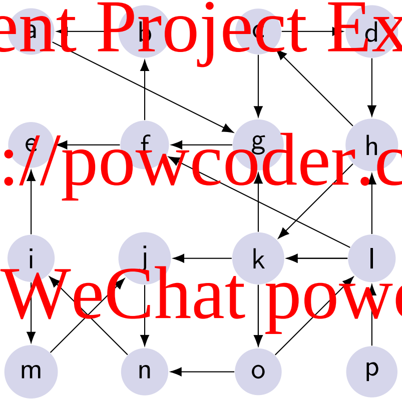
The last vertex in any postordering of the graph obtained by reversing the edges of G lies in a sink component of G.

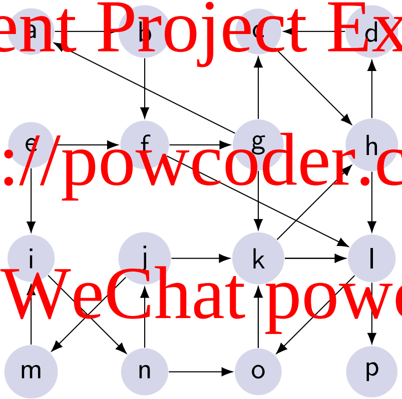
How to find a vertex in a sink component?

Lemma

The last vertex in any postordering of the graph obtained by reversing the edges of G lies in a sink component of G.

Then, if we traverse the graph a second time, where the wrapper function follows a reverse postordering of rev(G), each call to DFS in the wrapper function visits exactly one strongly connected component.


a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p


a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p

Given a connected and undirected graph G with weights on the edges given by function $w : E \to \mathbb{R}$...

Given a connected and undirected graph G with weights on the edges given by function $w : E \to \mathbb{R}$...

... we are interested in finding the minimum spanning tree of G, that is, the spanning tree T that minimizes the function

$$w(T) = \sum_{e \in T} w(e).$$

The generic minimum spanning tree algorithm maintains an acyclic subgraph F of the input graph G: the intermediate spanning forest.

The generic minimum spanning tree algorithm maintains an acyclic subgraph F of the input graph G: the intermediate spanning forest.

At all times, F satisfies the following invariant:

The generic minimum spanning tree algorithm maintains an acyclic subgraph F of the input graph G: the intermediate spanning forest.

At all times, F satisfies the following invariant:

Invariant

F is a subgraph of the minimum spanning tree of G.

The generic minimum spanning tree algorithm maintains an acyclic subgraph F of the input graph G: the intermediate spanning forest.

At all times, F satisfies the following invariant:

Invariant

F is a subgraph of the minimum spanning tree of G.

Initially, F consists of V one-vertex trees.

The only minimum spanning tree algorithm

The generic minimum spanning tree algorithm maintains an acyclic subgraph F of the input graph G: the intermediate spanning forest.

At all times, F satisfies the following invariant:

Invariant

F is a subgraph of the minimum spanning tree of G.

Initially, F consists of V one-vertex trees. The generic algorithm connects trees in F by adding certain edges between them.

The generic minimum spanning tree algorithm maintains an acyclic subgraph F of the input graph G, the intermediate spanning forest.

At all times, F satisfies the following invariant:

Invariant:

F is a subgraph of the minimum spanning tree of G.

Initially, F consists of V one-vertex trees. The generic algorithm connects trees in F by adding certain edges between them.

When the algorithm halts, F consists of a single spanning tree;

The generic minimum spanning tree algorithm maintains an acyclic subgraph F of the input graph G: the intermediate spanning forest.

At all times, F satisfies the following invariant:

Invariant

F is a subgraph of the minimum spanning tree of G.

Initially, F consists of V one-vertex trees. The generic algorithm connects trees in F by adding certain edges between them.

When the algorithm halts, F consists of a single spanning tree; the invariant implies that this must be the minimum spanning tree of G.

At any stage of its evolution, the intermediate spanning forest F induces two special types of edges in the rest of the graph:

At any stage of its evolution, the intermediate spanning forest F induces two special types of edges in the rest of the graph:

- An edge is useless if it is not an edge of F, but both its endpoints are in the same component of F.

At any stage of its evolution, the intermediate spanning forest F induces two special types of edges in the rest of the graph:

- An edge is useless if it is not an edge of F, but both its endpoints are in the same component of F.

- An edge is safe if it is the minimum-weight edge with exactly one endpoint in some component of F.

The only minimum spanning tree algorithm

At any stage of its evolution, the intermediate spanning forest F induces two special types of edges in the rest of the graph:

- An edge is useless if it is not an edge of F, but both its endpoints are in the same component of F.

- An edge is safe if it is the minimum-weight edge with exactly one endpoint in some component of F.

The same edge could be safe for two different components of F.

At any stage of its evolution, the intermediate spanning forest F induces two special types of edges in the rest of the graph:

- An edge is useless if it is not an edge of F, but both its endpoints are in the same component of F.

- An edge is safe if it is the minimum-weight edge with exactly one endpoint in some component of F.

The same edge could be safe for two different components of F.

Some edges of $G \setminus F$ are neither safe nor useless; we call these edges undecided.

All minimum spanning tree algorithms are based on two simple observations:

All minimum spanning tree algorithms are based on two simple observations:

- The minimum spanning tree of G contains every safe edge

All minimum spanning tree algorithms are based on two simple observations:

1. The minimum spanning tree of G contains every safe edge
2. The minimum spanning tree contains no useless edge

All minimum spanning tree algorithms are based on two simple observations:

1. The minimum spanning tree of G contains every safe edge
2. The minimum spanning tree contains no useless edge

The generic minimum spanning tree algorithm repeatedly adds safe edges to the evolving forest F.

The only minimum spanning tree algorithm

All minimum spanning tree algorithms are based on two simple observations:

1. The minimum spanning tree of G contains every safe edge
2. The minimum spanning tree contains no useless edge

The generic minimum spanning tree algorithm repeatedly adds safe edges to the evolving forest F.

If F is not yet connected, there must be at least one safe edge, because the input graph G is connected.

The only minimum spanning tree algorithm

All minimum spanning tree algorithms are based on two simple observations:

1. The minimum spanning tree of G contains every safe edge
2. The minimum spanning tree contains no useless edge

The generic minimum spanning tree algorithm repeatedly adds safe edges to the evolving forest F.

If F is not yet connected, there must be at least one safe edge, because the input graph G is connected. Thus, no matter which safe edges we add in each iteration, our generic algorithm eventually connects F.

The only minimum spanning tree algorithm

All minimum spanning tree algorithms are based on two simple observations:

1. The minimum spanning tree of G contains every safe edge
2. The minimum spanning tree contains no useless edge

The generic minimum spanning tree algorithm repeatedly adds safe edges to the evolving forest F.

If F is not yet connected, there must be at least one safe edge, because the input graph G is connected. Thus, no matter which safe edges we add in each iteration, our generic algorithm eventually connects F. Observation 1 implies that the resulting tree is the minimum spanning tree.

All minimum spanning tree algorithms are based on two simple observations:

1. The minimum spanning tree of G contains every safe edge
2. The minimum spanning tree contains no useless edge

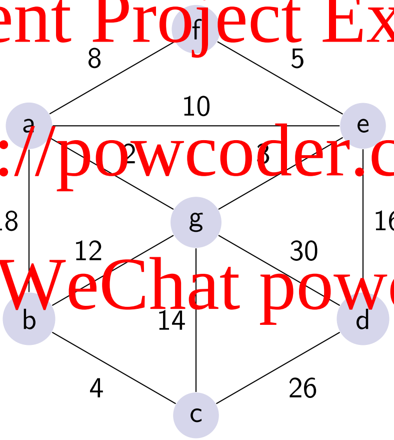
The generic minimum spanning tree algorithm repeatedly adds safe edges to the evolving forest F.

If F is not yet connected, there must be at least one safe edge, because the input graph G is connected. Thus, no matter which safe edges we add in each iteration, our generic algorithm eventually connects F. Observation 1 implies that the resulting tree is the minimum spanning tree.

To fully specify a particular algorithm, we must decide which safe edge(s) to add in each iteration and how to find them.

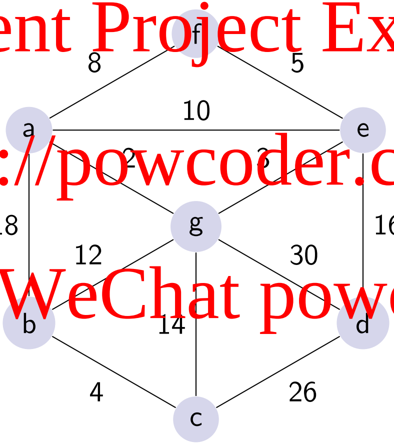
Boruvka's algorithm

Add ALL the safe edges and recurse.



Jarnik's ("Prim's") Algorithm

Start with T being an arbitrary vertex, then repeatedly add T's safe edge to T.



Start with T being an arbitrary vertex, then repeatedly add T's safe edge to T.

To implement Jarnik's algorithm, we keep all the edges adjacent to T in a priority queue:

Start with T being an arbitrary vertex, then repeatedly add T's safe edge to T.

To implement Jarnik's algorithm, we keep all the edges adjacent to T in a priority queue:

1. When we pull the minimum-weight edge out of the priority queue, we first check whether both of its endpoints are in T.

Jarnik's ("Prim's") Algorithm

Start with T being an arbitrary vertex, then repeatedly add T's safe edge to T.

To implement Jarnik's algorithm, we keep all the edges adjacent to T in a priority queue:

1. When we pull the minimum-weight edge out of the priority queue, we first check whether both of its endpoints are in T.

2. If not, we add the edge to T and then add the new neighboring edges to the priority queue.

Start with T being an arbitrary vertex, then repeatedly add T's safe edge to T.

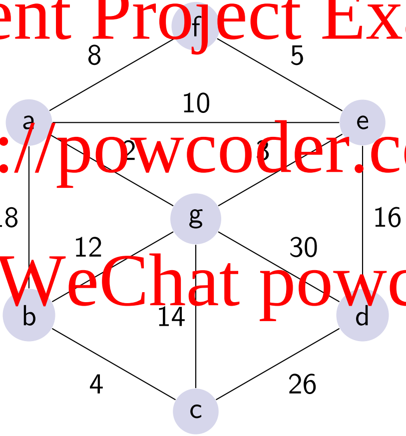
To implement Jarnik's algorithm, we keep all the edges adjacent to T in a priority queue:

1. When we pull the minimum-weight edge out of the priority queue, we first check whether both of its endpoints are in T.

2. If not, we add the edge to T and then add the new neighboring edges to the priority queue.

Jarnik's algorithm is a variant of "best-first search" which runs in $O(E \log E) = O(E \log V)$ time if a binary heap is used to implemenent the priority queue.

Scan all edges by increasing weight; if an edge is safe, add it to F.



Given a weighted directed graph $G = (V, E, w)$, want to find the shortest path from a source vertex s to a target vertex t.

Given a weighted directed graph $G = (V, E, w)$, want to find the shortest path from a source vertex s to a target vertex t.

That is, we want to find the directed path P starting at s and ending at t that minimizes the function

$$w(P) = \sum_{u \to v \in P} w(u \to v).$$

Given a weighted directed graph $G = (V, E, w)$, want to find the shortest path from a source vertex s to a target vertex t.

That is, we want to find the directed path P starting at s and ending at t that minimizes the function

$$w(P) = \sum_{u \to v \in P} w(u \to v).$$

Most algorithms for computing shortest paths from one vertex to another actually solve a bigger problem.

Given a weighted directed graph $G = (V, E, w)$, want to find the shortest path from a source vertex s to a target vertex t.

That is, we want to find the directed path P starting at s and ending at t that minimizes the function

$$w(P) = \sum_{u \to v \in P} w(u \to v).$$

Most algorithms for computing shortest paths from one vertex to another actually solve a bigger problem:

> *SSSP: Find the shortest paths from the source vertex s to every other vertex in the graph.*

Each vertex v in G stores two values:

Each vertex v in G stores two values:

- $dist(v)$ is the length of the tentative shortest $s \rightsquigarrow v$ path, or ∞ if there is no such path.

Each vertex v in G stores two values:

- $dist(v)$ is the length of the tentative shortest $s \rightsquigarrow v$ path, or ∞ if there is no such path.
- $pred(v)$ is the predecessor of v in the tentative shortest $s \rightsquigarrow v$ path, or $Null$ if there is no such vertex.

Each vertex v in G stores two values:

- $dist(v)$ is the length of the tentative shortest $s \rightsquigarrow v$ path, or ∞ if there is no such path.
- $pred(v)$ is the predecessor of v in the tentative shortest $s \rightsquigarrow v$ path, or $Null$ if there is no such vertex.

At the beginning of the algorithm, we initialize the distances and predecessors as follows:


```
InitSSSP(s):
  dist(s) ← 0
  pred(s) ← Null
  for all vertices v ≠ s
    dist(v) ← ∞
    pred(v) ← Null
```

During the execution of the algorithm, an edge $u \to v$ is said to be **tense** if $dist(u) + w(u \to v) < dist(v)$.

During the execution of the algorithm, an edge $u \rightarrow v$ is said to be tense if $dist(u) + w(u \rightarrow v) < dist(v)$.

If $u \rightarrow v$ is tense, the tentative shortest path $s \rightsquigarrow v$ is clearly incorrect, because the path $s \rightsquigarrow u \rightarrow v$ is shorter.

During the execution of the algorithm, an edge $u \rightarrow v$ is said to be tense if $dist(u) + w(u \rightarrow v) < dist(v)$.

If $u \rightarrow v$ is tense, the tentative shortest path $s \rightsquigarrow v$ is clearly incorrect, because the path $s \rightsquigarrow u \rightarrow v$ is shorter.

We improve this overestimate by relaxing the edge as follows:

```
Relax(u→v):
  dist(v) ← dist(u) + w(u→v)
  pred(v) ← u
```

Repeatedly relax tense edges, until there are no more tense edges.

Repeatedly relax tense edges, until there are no more tense edges.


```
FordSSSP(s):
  InitSSSP(s)
  while there is at least one tense edge
    Relax any tense edge
```

Repeatedly relax tense edges, until there are no more tense edges.


```
FordSSSP(s):
  InitSSSP(s)
  while there is at least one tense edge
    Relax any tense edge
```


We need to a method to find tense edges or which tense edge(s) to relax if there is more than one.

Repeatedly relax tense edges, until there are no more tense edges.


```
FordSSSP(s):
  InitSSSP(s)
  while there is at least one tense edge
    Relax any tense edge
```


We need to a method to find tense edges or which tense edge(s) to relax if there is more than one.

There are several ways to do that, depending on the structure of the input graph, which leads to different algorithms.

If all edges have weight 1 and the length of a path is just the number of edges then we can just use BFS:

Unweighted Graphs: Breadth-First Search

If all edges have weight 1 and the length of a path is just the number of edges then we can just use BFS:


```
BFS(s):
  InitSSSP(s)
  Push(s)
  while the queue is not empty
    u ← Pull()
    for all edges u→v
      if dist(v) > dist(u) + 1
        dist(v) ← dist(u) + 1
        pred(v) ← u
        Push(v)
```

If all edges have weight 1 and the length of a path is just the number of edges then we can just use BFS:

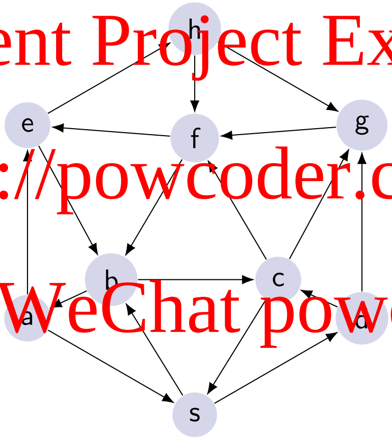

```
BFS(s):
  InitSSSP(s)
  Push(s)
  while the queue is not empty
    u ← Pull()
    for all edges u→v
      if dist(v) > dist(u) + 1
        dist(v) ← dist(u) + 1
        pred(v) ← u
        Push(v)
```


Running time:

If all edges have weight 1 and the length of a path is just the number of edges then we can just use BFS:

```
BFS(s):
  InitSSSP(s)
  Push(s)
  while the queue is not empty
    u ← Pull()
    for all edges u→v
      if dist(v) > dist(u) + 1
        dist(v) ← dist(u) + 1
        pred(v) ← u
        Push(v)
```

Running time: $O(V + E)$


h
e
f
g
b
c
a
d
s

If the FIFO queue in BFS is replaced with a priority queue, where the key of a vertex v is its tentative distance $dist(v)$, we get Dijkstra's algorithm.

No negative edges: Dijkstra's algorithm

If the FIFO queue in BFS is replaced with a priority queue, where the key of a vertex v is its tentative distance $dist(v)$, we get Dijkstra's algorithm.

Works well for arbitrary weighted graphs as longs as the edge weights are not negative.

```
NonnegativeDijkstra(s):
  InitSSSP(s)
  for all vertices v
    Insert(v,dist(v))
  while the priority queue is not empty
    u ← ExtractMin( )
    for all edges u→v
      if u→v is tense
        Relax(u→v)
        DecreaseKey(v,dist(v))
```

```
NonnegativeDijkstra(s):
  InitSSSP(s)
  for all vertices v
    Insert(v,dist(v))
  while the priority queue is not empty
    u ← ExtractMin( )
    for all edges u→v
      if u→v is tense
        Relax(u→v)
        DecreaseKey(v,dist(v))
```

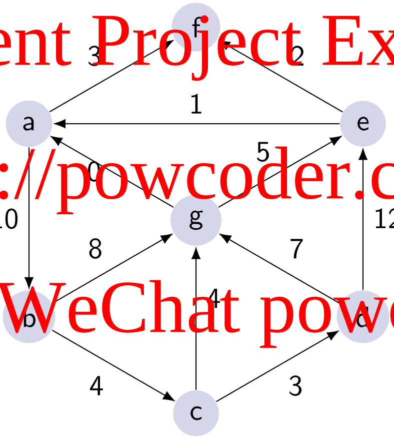

Running time:

No negative edges: Dijkstra's algorithm


```
NonnegativeDijkstra(s):
  InitSSSP(s)
  for all vertices v
    Insert(v,dist(v))
  while the priority queue is not empty
    u ← ExtractMin( )
    for all edges u→v
      if u→v is tense
        Relax(u→v)
        DecreaseKey(v,dist(v))
```

Running time: $O(E \log V)$

No negative edges: Dijkstra's algorithm



Directed Acyclic Graphs: DFS + dynamic programming

The following holds for all directed graphs

$$dist(v) = \begin{cases} 0, & \text{if } v = s \\ min_{u \to v}(dist(u) + w(u \to v)), & \text{otherwise} \end{cases}$$

The following holds for all directed graphs

$$dist(v) = \begin{cases} 0, & \text{if } v = s \\ min_{u \to v}(dist(u) + w(u \to v)), & \text{otherwise} \end{cases}$$

... but it is only a recurrence for directed acyclic graphs.

The following holds for all directed graphs

$$dist(v) = \begin{cases} 0, & \text{if } v = s \\ min_{u \to v}(dist(u) + w(u \to v)), & \text{otherwise} \end{cases}$$

... but it is only a recurrence for directed acyclic graphs.

Why?

Directed Acyclic Graphs: DFS + dynamic programming

The following holds for all directed graphs

$$dist(v) = \begin{cases} 0, & \text{if } v = s \\ \min_{u \to v} (dist(u) + w(u \to v)), & \text{otherwise} \end{cases}$$

... but it is only a recurrence for directed acyclic graphs.

Why? If the input graph G contained a cycle, a recursive evaluation of this function would fall into an infinite loop;

The following holds for all directed graphs

$$dist(v) = \begin{cases} 0, & \text{if } v = s \\ min_{u \to v}(dist(u) + w(u \to v)), & \text{otherwise} \end{cases}$$

... but it is only a recurrence for directed acyclic graphs.

Why? If the input graph G contained a cycle, a recursive evaluation of this function would fall into an infinite loop; if G is a dag, each recursive call visits an earlier vertex in topological order.

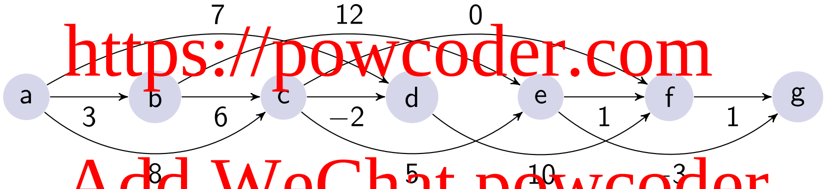
Note that subproblem $dist(v)$ depends on $dist(u)$ if and only if $u \to v$ is an edge in G.

Note that subproblem $dist(v)$ depends on $dist(u)$ if and only if $u \to v$ is an edge in G.

Thus, we compute the distance from s of every vertex v in G by first computing (using DFS) a topological ordering of the vertices in G and then applying dynamic programming to compute $dist(v)$ in topological order.

```
DagSSSP(s):
  for all vertices v in topological order
    if v = s
      dist(v) ← 0
    else
      dist(v) ← ∞
      for all edges u→v
        if dist(v) > dist(u) + w(u→v)
          dist(v) ← dist(u) + w(u→v)
```

Directed Acyclic Graphs: DFS + dynamic programming



A refactoring of the algorithm to fit the SSSP algorithm format:


```
PushDagSSSP(s):
  InitSSSP(s)
  for all vertices u in topological order
    for all outgoing edges u→v
      if u→v is tense
        Relax(u→v)
```