Breadth First Search, Dijkstra's Algorithm for Shortest Paths

Lecture 17

March 28, 2023

Part I

Breadth First Search

Breadth First Search (BFS)

Overview

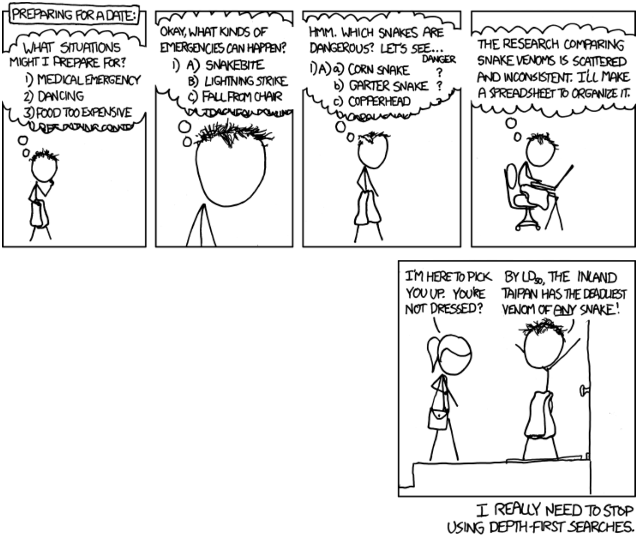
(A) **BFS** is obtained from **BasicSearch** by processing edges using a data structure called a **queue**.

(B) It processes the vertices in the graph in the order of their shortest distance from the vertex s (the start vertex).

As such...

1. **DFS** good for exploring graph structure
2. **BFS** good for exploring *distances*

xkcd take on DFS



Distances in Graphs

Given a graph $G = (V, E)$ and two nodes s, t the *distance* $\mathbf{dist}(s, t)$ is the length of the shortest path from s to t in G

Distances in Graphs

Given a graph $G = (V, E)$ and two nodes s, t the *distance* $\mathbf{dist}(s, t)$ is the length of the shortest path from s to t in G

- $\mathbf{dist}(s, t) = \mathbf{dist}(t, s)$ in *undirected* graphs while $\mathbf{dist}(s, t)$ and $\mathbf{dist}(t, s)$ may be different in *directed* graphs

Distances in Graphs

Given a graph $G = (V, E)$ and two nodes s, t the *distance* $\mathbf{dist}(s, t)$ is the length of the shortest path from s to t in G

- $\mathbf{dist}(s, t) = \mathbf{dist}(t, s)$ in *undirected* graphs while $\mathbf{dist}(s, t)$ and $\mathbf{dist}(t, s)$ may be different in *directed* graphs
- Triangle inequality: $\mathbf{dist}(u, v) + \mathbf{dist}(v, w) \geq \mathbf{dist}(u, w)$ for all $u, v, w \in V$

Shortest Path Problems

Shortest Path Problems

Input A (undirected or directed) graph $G = (V, E)$

1. Given nodes s, t find shortest path from s to t.
2. Given node s find shortest path from s to all other nodes.
3. Find shortest paths for all pairs of nodes.

Many applications!

Shortest Path Problems

Shortest Path Problems

Input A (undirected or directed) graph $G = (V, E)$

1. Given nodes s, t find shortest path from s to t.
2. Given node s find shortest path from s to all other nodes.
3. Find shortest paths for all pairs of nodes.

Many applications!

These are *unweighted* problems. More general problem when edges have lengths which can potentially be negative! Will discuss later.

Single-Source Shortest Paths

Single-Source Shortest Path Problems

Input A (undirected or directed) graph $G = (V, E)$

1. Given nodes s, t find shortest path from s to t.
2. Given node s find shortest path from s to all other nodes.

Single-Source Shortest Paths

Single-Source Shortest Path Problems

Input A (undirected or directed) graph $G = (V, E)$

1. Given nodes s, t find shortest path from s to t.
2. Given node s find shortest path from s to all other nodes.

Notation: If s is clear from context we may use $\mathrm{dist}(u)$ as short hand for $\mathrm{dist}(s, u)$.

Single-Source Shortest Paths

Single-Source Shortest Path Problems

Input A (undirected or directed) graph $G = (V, E)$

1. Given nodes s, t find shortest path from s to t.
2. Given node s find shortest path from s to all other nodes.

Notation: If s is clear from context we may use $\text{dist}(u)$ as short hand for $\text{dist}(s, u)$.

- **BFS** solves single-source shortest path problems in unweighted graphs (both undirected and directed) in $O(n + m)$ time.
- **BFS** is obtained from Basic Search by using a Queue data structure

Queue Data Structure

Queues

A **queue** is a list of elements which supports the operations:

1. **enqueue**: Adds an element to the end of the list
2. **dequeue**: Removes an element from the front of the list

Elements are extracted in **first-in first-out (FIFO)** order, i.e., elements are picked in the order in which they were inserted.

BFS Algorithm

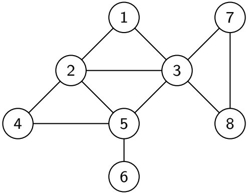
Given (undirected or directed) graph $G = (V, E)$ and node $s \in V$

```
BFS(s)
    Mark all vertices as unvisited
    Initialize search tree T to be empty
    Mark vertex s as visited
    set Q to be the empty queue
    enq(s)
    while Q is nonempty do
        u = deq(Q)
        for each vertex v ∈ Adj(u)
            if v is not visited then
                add edge (u, v) to T
                Mark v as visited and enq(v)
```

Proposition

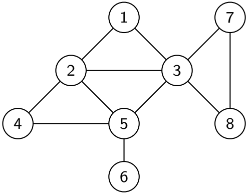
BFS(s) *runs in* $O(n + m)$ *time.*

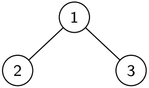
BFS: An Example in Undirected Graphs



1. [1]

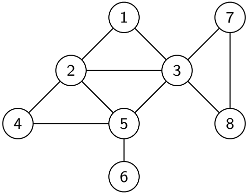
BFS: An Example in Undirected Graphs

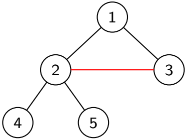




1. [1]
2. [2,3]

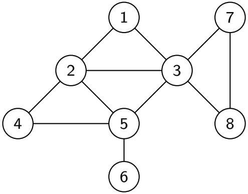
BFS: An Example in Undirected Graphs

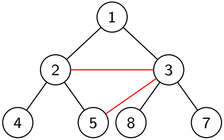




1. [1]
2. [2,3]
3. [3,4,5]

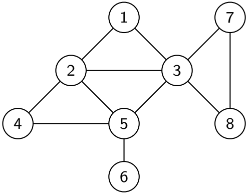
BFS: An Example in Undirected Graphs

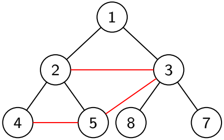




1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]

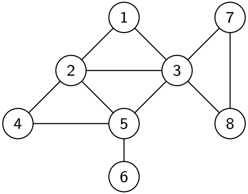
BFS: An Example in Undirected Graphs

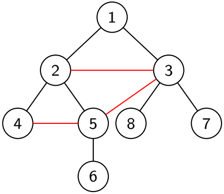




1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]

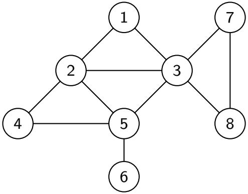
BFS: An Example in Undirected Graphs

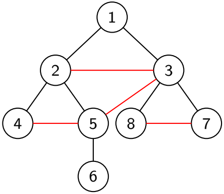




1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]

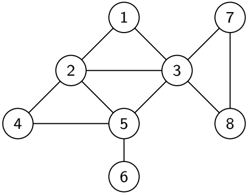
BFS: An Example in Undirected Graphs

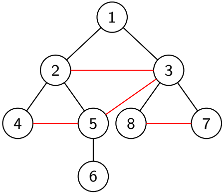




1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]

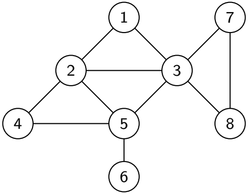
BFS: An Example in Undirected Graphs

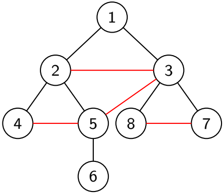




1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]

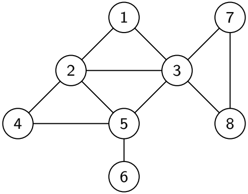
BFS: An Example in Undirected Graphs

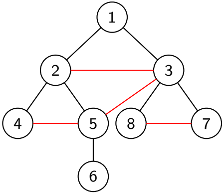




1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

BFS: An Example in Undirected Graphs

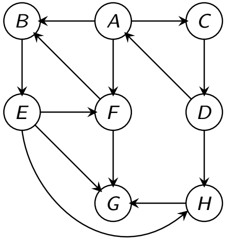




1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

BFS tree is the set of black edges.

BFS: An Example in Directed Graphs



BFS with Distance

```
BFS(s)
    Mark all vertices as unvisited; for each v set dist(v) = ∞
    Initialize search tree T to be empty
    Mark vertex s as visited and set dist(s) = 0
    set Q to be the empty queue
    enq(s)
    while Q is nonempty do
        u = deq(Q)
        for each vertex v ∈ Adj(u) do
            if v is not visited do
                add edge (u, v) to T
                Mark v as visited, enq(v)
                and set dist(v) = dist(u) + 1
```

Properties of BFS: Undirected Graphs

Theorem

The following properties hold upon termination of **BFS**(s)

- **(A)** *The search tree contains exactly the set of vertices in the connected component of* s.
- **(B)** *If* $\mathrm{dist}(u) < \mathrm{dist}(v)$ *then* u *is visited before* v.
- **(C)** *For every vertex* u, $\mathrm{dist}(u)$ *is the length of a shortest path (in terms of number of edges) from* s *to* u.
- **(D)** *If* u, v *are in connected component of* s *and* $e = \{u, v\}$ *is an edge of* G, *then* $|\mathrm{dist}(u) - \mathrm{dist}(v)| \leq 1$.

Properties of BFS: Directed Graphs

Theorem

The following properties hold upon termination of **BFS**(s):

- (A) *The search tree contains exactly the set of vertices reachable from* s
- (B) *If* $\mathrm{dist}(u) < \mathrm{dist}(v)$ *then* u *is visited before* v
- (C) *For every vertex* u, $\mathrm{dist}(u)$ *is indeed the length of shortest path from* s *to* u
- (D) *If* u *is reachable from* s *and* $e = (u, v)$ *is an edge of* G, *then* $\mathrm{dist}(v) - \mathrm{dist}(u) \leq 1$.
  Not necessarily the case that $\mathrm{dist}(u) - \mathrm{dist}(v) \leq 1$.

BFS with Layers

BFS is a simple algorithm but proving its properties formally is not straight forward

BFS explores graph in increasing order of distance from source s

There is a simpler variant that makes **BFS** exploration transparent and easier to understand.

BFS with Layers

BFS is a simple algorithm but proving its properties formally is not straight forward

BFS explores graph in increasing order of distance from source s

There is a simpler variant that makes **BFS** exploration transparent and easier to understand.

- Given G and $s \in V$ define $L_i = \{v \mid \mathrm{dist}(s, v) = i\}$. The "layer" of all vertices at exactly distance i from s

BFS with Layers

BFS is a simple algorithm but proving its properties formally is not straight forward

BFS explores graph in increasing order of distance from source s

There is a simpler variant that makes **BFS** exploration transparent and easier to understand.

- Given G and $s \in V$ define $L_i = \{v \mid \mathrm{dist}(s, v) = i\}$. The "layer" of all vertices at exactly distance i from s
- $L_0 = \{s\}$

BFS with Layers

BFS is a simple algorithm but proving its properties formally is not straight forward

BFS explores graph in increasing order of distance from source s

There is a simpler variant that makes **BFS** exploration transparent and easier to understand.

- Given G and $s \in V$ define $L_i = \{v \mid \mathrm{dist}(s, v) = i\}$. The "layer" of all vertices at exactly distance i from s
- $L_0 = \{s\}$
- Can find L_i from $L_0, L_2, \ldots, L_{i-1}$ inductively and easily.

BFS with Layers

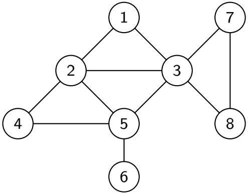
```
BFSLayers(s):
    Mark all vertices as unvisited and initialize T to be empty
    Mark s as visited and set L_0 = {s}
    i = 0
    while L_i is not empty do
            initialize L_{i+1} to be an empty list
            for each u in L_i do
                for each edge (u, v) ∈ Adj(u) do
                if v is not visited
                        mark v as visited
                        add (u, v) to tree T
                        add v to L_{i+1}
            i = i + 1
```

BFS with Layers

```
BFSLayers(s):
    Mark all vertices as unvisited and initialize T to be empty
    Mark s as visited and set L_0 = {s}
    i = 0
    while L_i is not empty do
            initialize L_{i+1} to be an empty list
            for each u in L_i do
                for each edge (u, v) ∈ Adj(u) do
                if v is not visited
                        mark v as visited
                        add (u, v) to tree T
                        add v to L_{i+1}
            i = i + 1
```

Running time: $O(n + m)$

Example



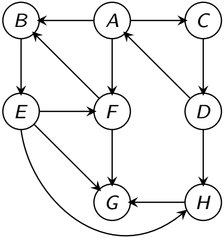
BFS with Layers: Properties

Proposition

The following properties hold on termination of **BFSLayers**(s).

1. **BFSLayers**(s) *outputs a* **BFS** *tree*
2. L_i *is the set of vertices at distance exactly* i *from* s
3. *If* G *is undirected, each edge* $e = \{u, v\}$ *is one of three types:*
   1. **tree** *edge between two* consecutive *layers*
   2. *non-tree* **forward**/**backward** *edge between two consecutive layers*
   3. *non-tree* **cross-edge** *with both* u, v *in same layer*
   4. $\implies$ *Every edge in the graph is either between two vertices that are either (i) in the same layer, or (ii) in two consecutive layers.*

Example



BFS with Layers: Properties

For directed graphs

Proposition

The following properties hold on termination of **BFSLayers**(s)*, if* G *is directed.*

For each edge $e = (u, v)$ *is one of four types:*

1. *a* **tree** *edge between consecutive layers,* $u \in L_i, v \in L_{i+1}$ *for some* $i \geq 0$
2. *a non-tree* **forward** *edge between consecutive layers*
3. *a non-tree* **backward** *edge*
4. *a* **cross-edge** *with both* u, v *in same layer*

Part II

Shortest Paths and Dijkstra's Algorithm

Shortest Path Problems

Shortest Path Problems

Input A (undirected or directed) graph $G = (V, E)$ with edge lengths (or costs). For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Given nodes s, t find shortest path from s to t.
2. Given node s find shortest path from s to all other nodes.
3. Find shortest paths for all pairs of nodes.

Shortest Walk Problems

Given a graph $G = (V, E)$:

1. A path is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \le i \le k-1$.
2. A walk is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \le i \le k-1$. Vertices are allowed to repeat.

Shortest Walk Problems

Given a graph $G = (V, E)$:

1. A path is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$.
2. A walk is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$. Vertices are allowed to repeat.

Finding walks is often easier and more natural than finding paths. Why?

Shortest Walk Problems

Given a graph $G = (V, E)$:

1. A path is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$.
2. A walk is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$. Vertices are allowed to repeat.

Finding walks is often easier and more natural than finding paths. Why? Concatenating two walks gives a walk while concatenating two paths may give a walk, and not necessarily a path.

Shortest Walk Problems

Given a graph $G = (V, E)$:

1. A path is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$.
2. A walk is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$. Vertices are allowed to repeat.

Finding walks is often easier and more natural than finding paths. Why? Concatenating two walks gives a walk while concatenating two paths may give a walk, and not necessarily a path.

When edges have non-negative lenghts, finding a shortest s-t walk is the same as finding a shortest s-t path. Why?

Shortest Walk Problems

Given a graph $G = (V, E)$:

1. A path is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$.
2. A walk is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$. Vertices are allowed to repeat.

Finding walks is often easier and more natural than finding paths. Why? Concatenating two walks gives a walk while concatenating two paths may give a walk, and not necessarily a path.

When edges have non-negative lenghts, finding a shortest s-t walk is the same as finding a shortest s-t path. Why?

In more general settings walks are easier to work with.

Single-Source Shortest Paths:

Non-Negative Edge Lengths

Single-Source Shortest Path Problems

1. Input: A (undirected or directed) graph $G = (V, E)$ with non-negative edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.
2. Given nodes s, t find shortest path from s to t.
3. Given node s find shortest path from s to all other nodes.

Single-Source Shortest Paths:

Non-Negative Edge Lengths

Single-Source Shortest Path Problems

1. Input: A (undirected or directed) graph $G = (V, E)$ with non-negative edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.
2. Given nodes s, t find shortest path from s to t.
3. Given node s find shortest path from s to all other nodes.

1. Restrict attention to directed graphs
2. Undirected graph problem can be reduced to directed graph problem - how?

Single-Source Shortest Paths:

Non-Negative Edge Lengths

Single-Source Shortest Path Problems

1. Input: A (undirected or directed) graph $G = (V, E)$ with non-negative edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.
2. Given nodes s, t find shortest path from s to t.
3. Given node s find shortest path from s to all other nodes.

1. Restrict attention to directed graphs
2. Undirected graph problem can be reduced to directed graph problem - how?
   1. Given undirected graph G, create a new directed graph G' by replacing each edge $\{u, v\}$ in G by (u, v) and (v, u) in G'.
   2. set $\ell(u, v) = \ell(v, u) = \ell(\{u, v\})$
   3. Exercise: show reduction works. Relies on non-negativity!

Single-Source Shortest Paths via BFS

Special case: All edge lengths are 1.

Single-Source Shortest Paths via BFS

Special case: All edge lengths are 1.

1. Run **BFS**(s) to get shortest path distances from s to all other nodes.
2. $O(m + n)$ time algorithm.

Single-Source Shortest Paths via BFS

Special case: All edge lengths are 1.

1. Run **BFS**(s) to get shortest path distances from s to all other nodes.
2. $O(m + n)$ time algorithm.

Special case: Suppose $\ell(e)$ is an integer for all e?
Can we use **BFS**?

Single-Source Shortest Paths via BFS

Special case: All edge lengths are 1.

1. Run **BFS**(s) to get shortest path distances from s to all other nodes.
2. $O(m + n)$ time algorithm.

Special case: Suppose $\ell(e)$ is an integer for all e?
Can we use **BFS**? Reduce to unit edge-length problem by placing $\ell(e) - 1$ dummy nodes on e

Single-Source Shortest Paths via BFS

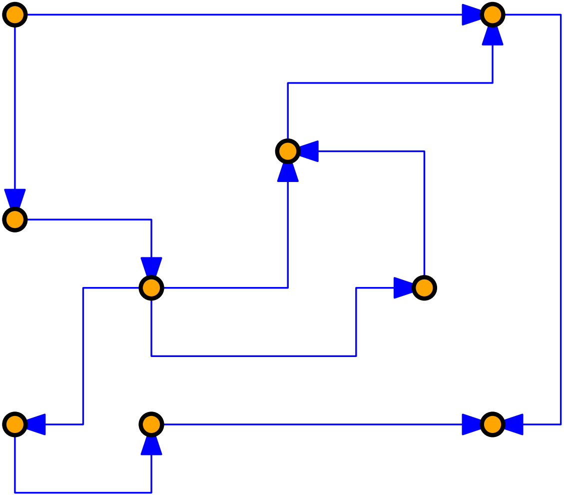
Special case: All edge lengths are 1.

1. Run **BFS**(s) to get shortest path distances from s to all other nodes.
2. $O(m+n)$ time algorithm.

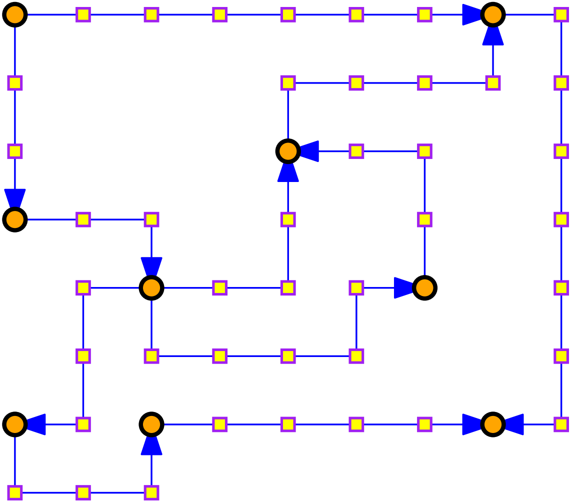
Special case: Suppose $\ell(e)$ is an integer for all e?
Can we use **BFS**? Reduce to unit edge-length problem by placing $\ell(e)-1$ dummy nodes on e

Let $L = \max_e \ell(e)$. New graph has $O(mL)$ edges and $O(mL+n)$ nodes. **BFS** takes $O(mL+n)$ time. Not efficient if L is large.

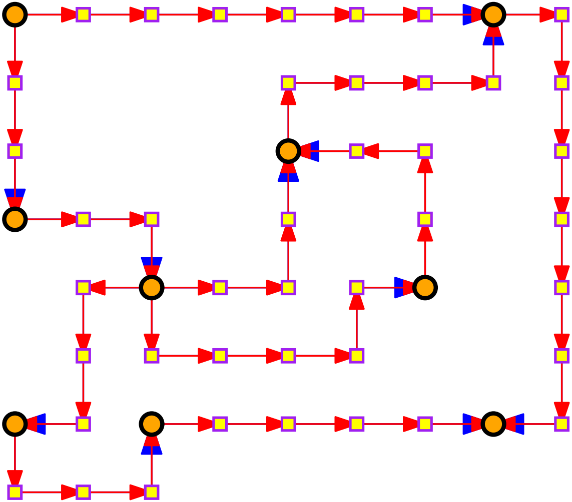
Example of edge refinement



Example of edge refinement



Example of edge refinement



Towards an algorithm

Why does **BFS** work?

Towards an algorithm

Why does **BFS** work?
BFS(s) explores nodes in increasing distance from s

Towards an algorithm

Why does **BFS** work?

BFS(s) explores nodes in increasing distance from s

Lemma

Let G be a directed graph with non-negative edge lengths. If $s = v_0 \to v_1 \to v_2 \to \ldots \to v_i$ is a shortest path from s to v_i then for $1 \leq j < i$:

1. $s = v_0 \to v_1 \to \ldots \to v_j$ *is a shortest path from s to v_j*
2. $\text{dist}(s, v_j) \leq \text{dist}(s, v_i)$. *Relies on non-neg edge lengths.*

Towards an algorithm

Lemma

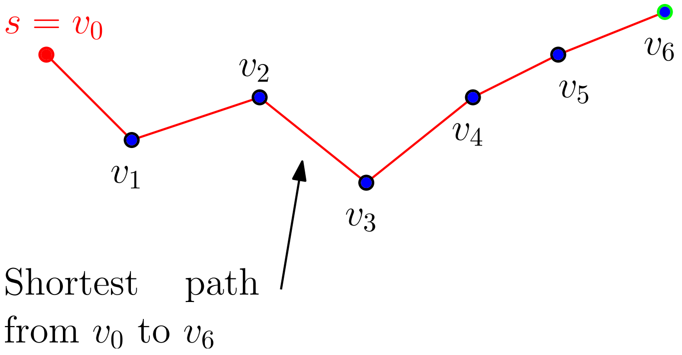
Let G be a directed graph with non-negative edge lengths. If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i$ is a shortest path from s to v_i then for $1 \leq j < i$:

1. $s = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_j$ *is a shortest path from* s *to* v_j
2. $\mathrm{dist}(s, v_j) \leq \mathrm{dist}(s, v_i)$. *Relies on non-neg edge lengths.*

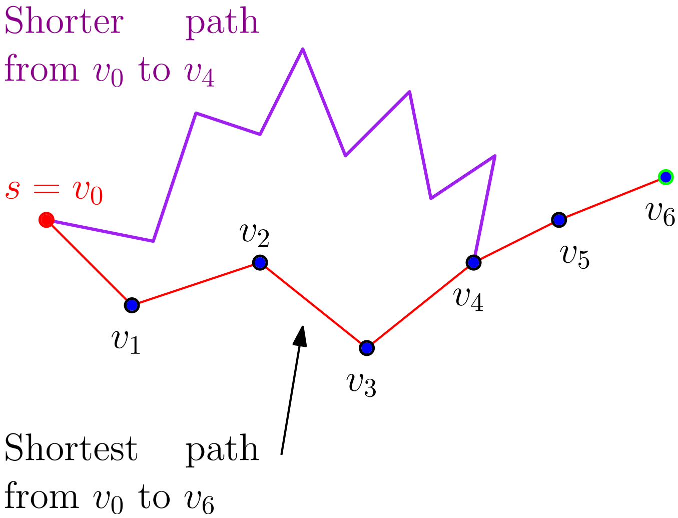
Proof.

Suppose not. Then for some $j < i$ there is a path P' from s to v_j of length strictly less than that of $s = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_j$. Then P' concatenated with $v_j \rightarrow v_{j+1} \ldots \rightarrow v_i$ contains a strictly shorter path to v_i than $s = v_0 \rightarrow v_1 \ldots \rightarrow v_i$. For the second part, observe that edge lengths are non-negative. □

A proof by picture

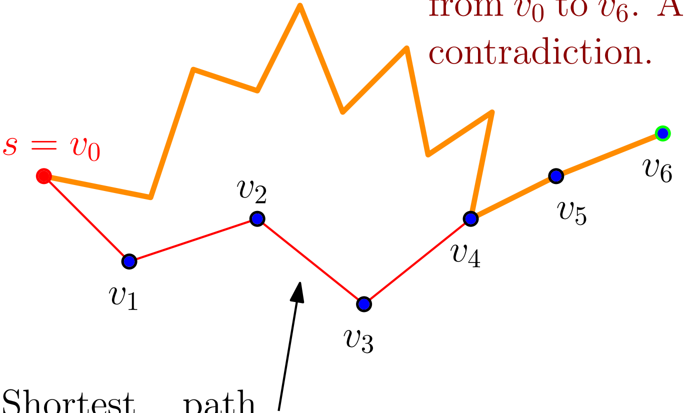


A proof by picture



A proof by picture

A shorter path from v_0 to v_6. A contradiction.



Shortest path from v_0 to v_6

A Basic Strategy

Explore vertices in increasing order of distance from s:
(For simplicity assume that nodes are at different distances from s and that no edge has zero length)

```
Initialize for each node v, dist(s, v) = ∞
Initialize X = {s},
for i = 2 to |V| do
    (* Invariant: X contains the i − 1 closest nodes to s *)
    Among nodes in V − X, find the node v that is the
            i'th closest to s
    Update dist(s, v)
    X = X ∪ {v}
```

A Basic Strategy

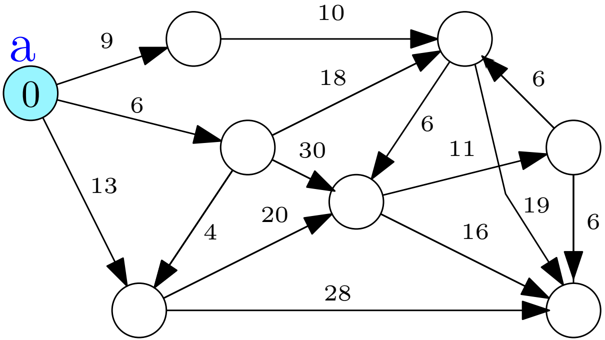
Explore vertices in increasing order of distance from s:
(For simplicity assume that nodes are at different distances from s and that no edge has zero length)

```
Initialize for each node v, dist(s, v) = ∞
Initialize X = {s},
for i = 2 to |V| do
    (* Invariant:  X contains the i − 1 closest nodes to s *)
    Among nodes in V − X, find the node v that is the
            i'th closest to s
    Update dist(s, v)
    X = X ∪ {v}
```

How can we implement the step in the for loop?

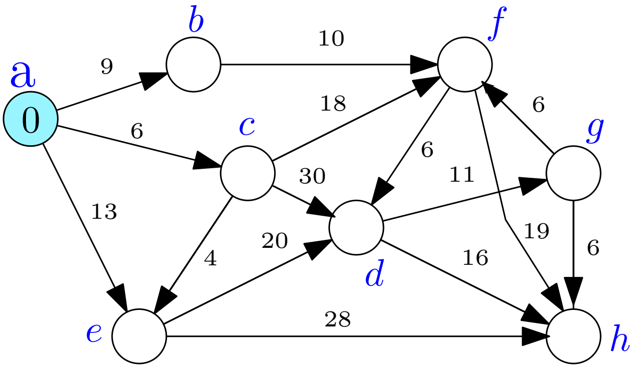
Finding the *i*'th closest node repeatedly

An example



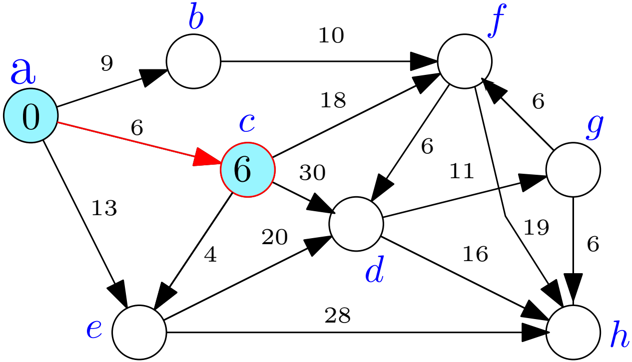
Finding the *i*'th closest node repeatedly

An example



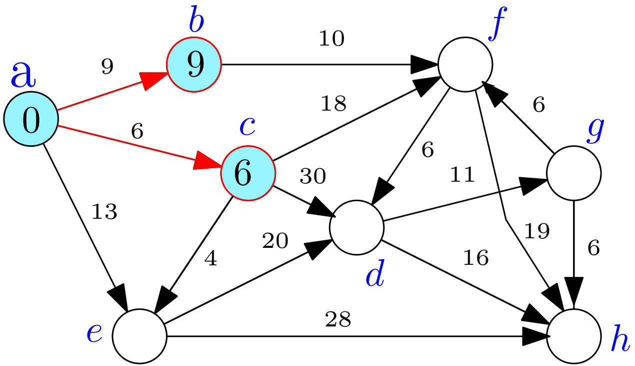
Finding the *i*'th closest node repeatedly

An example



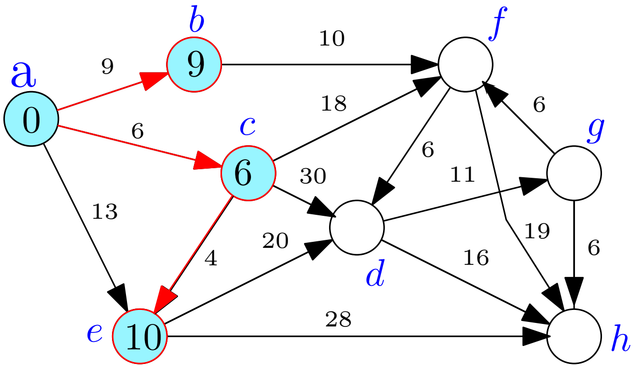
Finding the i'th closest node repeatedly

An example



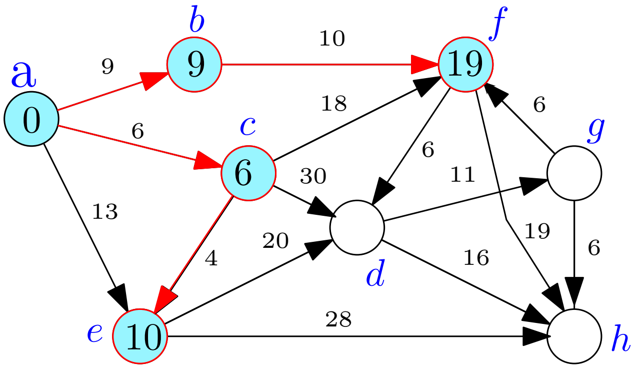
Finding the *i*'th closest node repeatedly

An example



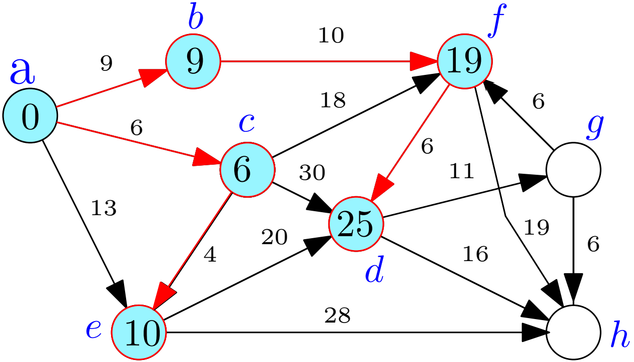
Finding the *i*'th closest node repeatedly

An example



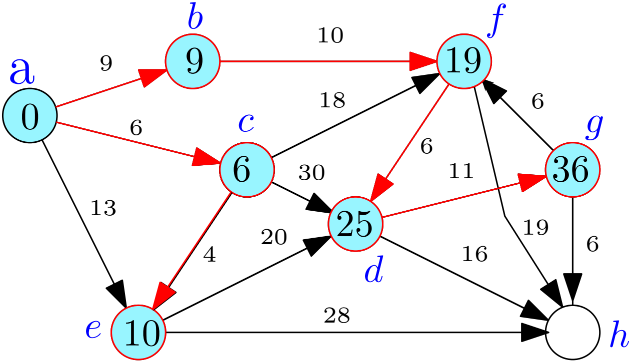
Finding the *i*'th closest node repeatedly

An example



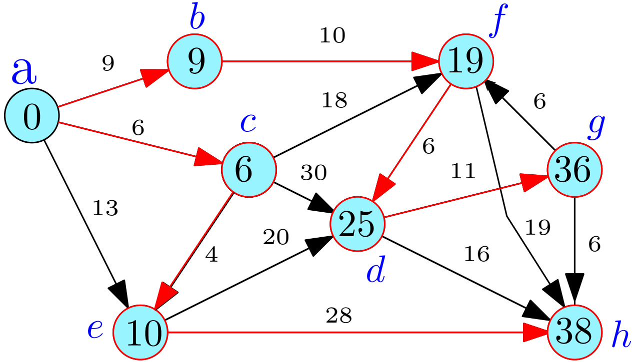
Finding the i'th closest node repeatedly

An example



Finding the *i*'th closest node repeatedly

An example



Finding the i'th closest node

1. X contains the $i - 1$ closest nodes to s
2. Want to find the ith closest node from $V - X$.

What do we know about the ith closest node?

Finding the i'th closest node

1. X contains the $i-1$ closest nodes to s
2. Want to find the ith closest node from $V-X$.

What do we know about the ith closest node?

Claim

Let P be a shortest path from s to v where v is the ith closest node. Then, all intermediate nodes in P belong to X.

Finding the i'th closest node

1. X contains the $i - 1$ closest nodes to s
2. Want to find the ith closest node from $V - X$.

What do we know about the ith closest node?

Claim

Let P be a shortest path from s to v where v is the ith closest node. Then, all intermediate nodes in P belong to X.

Proof.

If P had an intermediate node u not in X then u will be closer to s than v. Implies v is not the i'th closest node to s; recall that X already has the $i - 1$ closest nodes. □

Finding the i'th closest node

1. X contains the $i - 1$ closest nodes to s
2. Want to find the ith closest node from $V - X$.

1. For each $u \in V - X$ let $P(s, u, X)$ be a shortest path from s to u using only nodes in X as intermediate vertices.
2. Let $d'(s, u)$ be the length of $P(s, u, X)$

Finding the i'th closest node

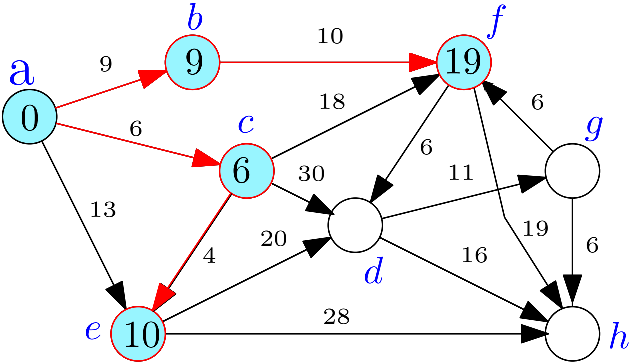
1. X contains the $i-1$ closest nodes to s
2. Want to find the ith closest node from $V - X$.

1. For each $u \in V - X$ let $P(s, u, X)$ be a shortest path from s to u using only nodes in X as intermediate vertices.
2. Let $d'(s, u)$ be the length of $P(s, u, X)$

Claim

For each $u \in V - X$, $d'(s, u) = \min_{t \in X}(\text{dist}(s, t) + \ell(t, u))$.

Understanding $d'(s, u)$ values



Finding the i'th closest node

1. X contains the $i - 1$ closest nodes to s
2. Want to find the ith closest node from $V - X$.

1. For each $u \in V - X$ let $P(s, u, X)$ be a shortest path from s to u using only nodes in X as intermediate vertices.
2. Let $d'(s, u)$ be the length of $P(s, u, X)$
3. Can compute all $d'(s, u)$ values

Main claim:

Lemma

The ith closest node to s is the node $v \in V - X$ with the smallest d' value, that is, $d'(s, v) = \min_{u \in V - X} d'(s, u)$.

Finding the i'th closest node

1. X contains the $i-1$ closest nodes to s
2. Want to find the ith closest node from $V-X$.

1. For each $u \in V-X$ let $P(s,u,X)$ be a shortest path from s to u using only nodes in X as intermediate vertices.
2. Let $d'(s,u)$ be the length of $P(s,u,X)$
3. Can compute all $d'(s,u)$ values

Main claim:

Lemma

The ith closest node to s is the node $v \in V-X$ with the smallest d' value, that is, $d'(s,v) = \min_{u \in V-X} d'(s,u)$.

Assuming claim, inductive algorithm follows.

Finding the i'th closest node: proof

Auxiliary lemma:

Lemma

If v is an ith closest node to s, then $d'(s, v) = \text{dist}(s, v)$.

Proof.

Let v be the ith closest node to s. Then there is a shortest path P from s to v that contains only nodes in X as intermediate nodes (see previous claim). Therefore $d'(s, v) = \text{dist}(s, v)$. □

Finding the i'th closest node: proof

Lemma

If v is an ith closest node to s, then $d'(s, v) = \mathrm{dist}(s, v)$.

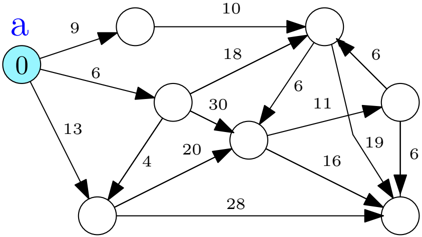
Lemma

The ith closest node to s is the node $v \in V - X$ with the smallest d' value, that is, $d'(s, v) = \min_{u \in V - X} d'(s, u)$.

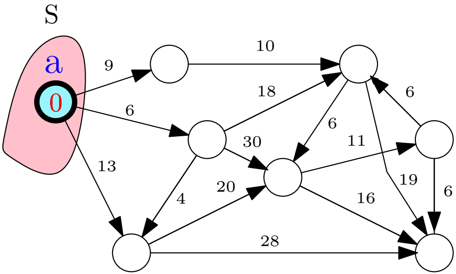
Proof.

Assume distances are unique for simplicity. Let $v^* \in V - X$ be the i'th closest node to s. Implies for every other $u \in V - X$, $d'(s, u) \geq \mathrm{dist}(s, u) > \mathrm{dist}(s, v^*)$. But Lemma says $d'(s, v^*) = \mathrm{dist}(s, v^*)$. Hence node v that minimizes $d'(s, v)$ value must be v^*. □

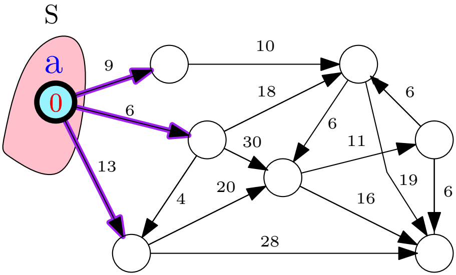
Example: Dijkstra algorithm in action



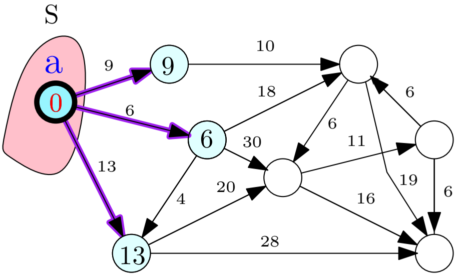
Example: Dijkstra algorithm in action



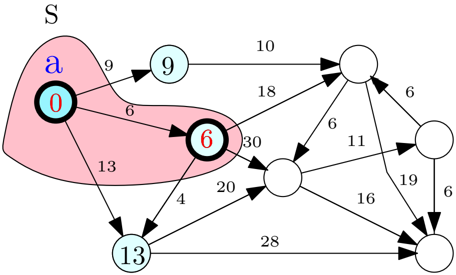
Example: Dijkstra algorithm in action



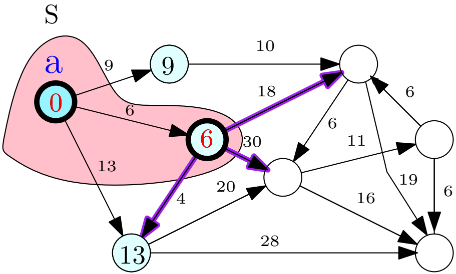
Example: Dijkstra algorithm in action



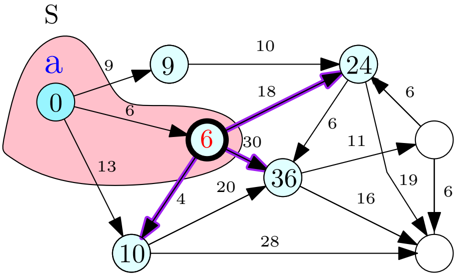
Example: Dijkstra algorithm in action



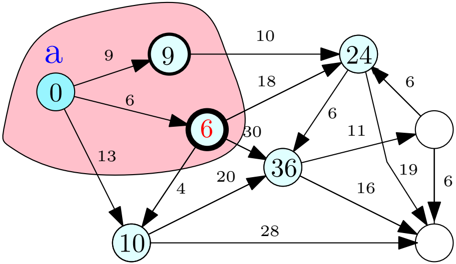
Example: Dijkstra algorithm in action



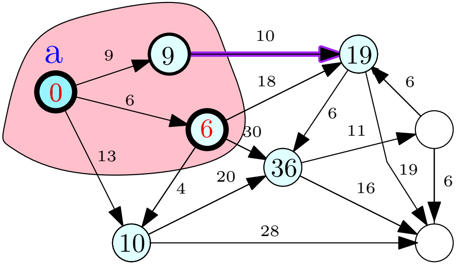
Example: Dijkstra algorithm in action



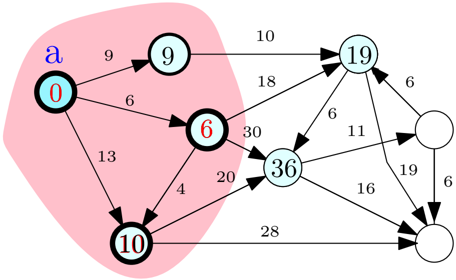
Example: Dijkstra algorithm in action



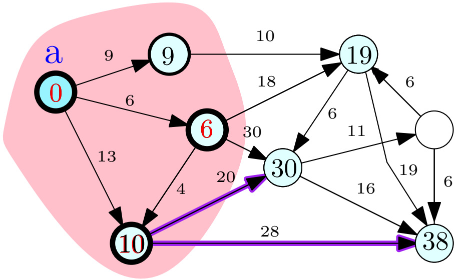
Example: Dijkstra algorithm in action



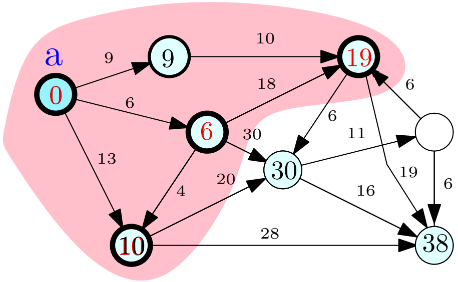
Example: Dijkstra algorithm in action



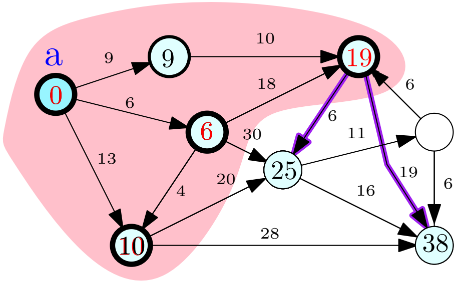
Example: Dijkstra algorithm in action



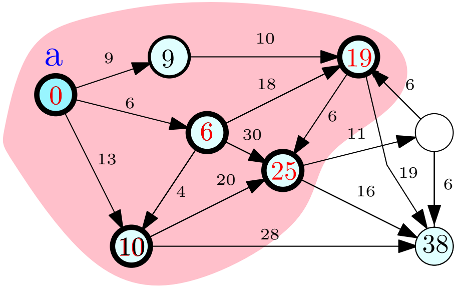
Example: Dijkstra algorithm in action



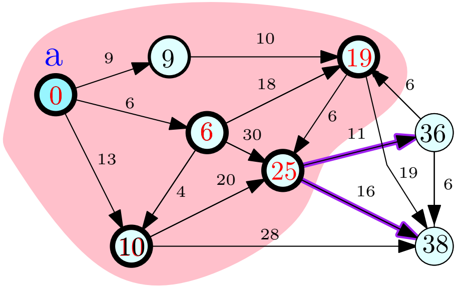
Example: Dijkstra algorithm in action



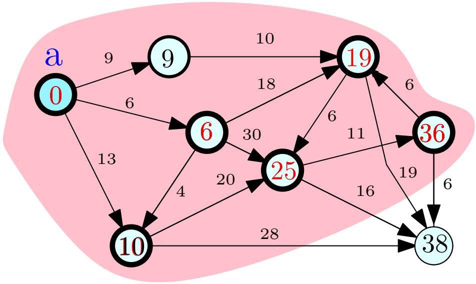
Example: Dijkstra algorithm in action



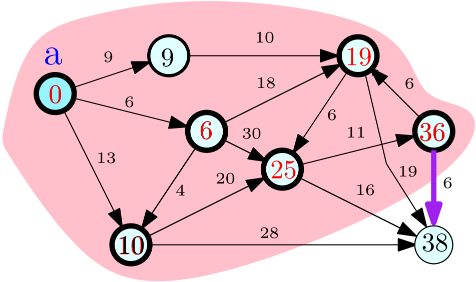
Example: Dijkstra algorithm in action



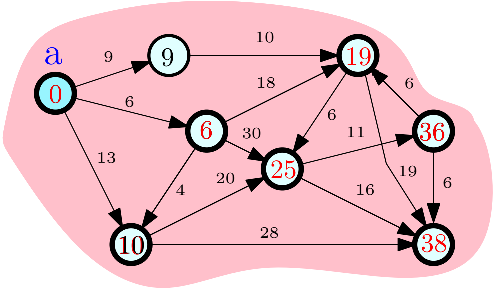
Example: Dijkstra algorithm in action



Example: Dijkstra algorithm in action



Example: Dijkstra algorithm in action



Algorithm

```
Initialize for each node v:  dist(s, v) = ∞
Initialize X = ∅,  d'(s, s) = 0
for i = 1 to |V| do
    (* Invariant:  X contains the i − 1 closest nodes to s *)
    (* Invariant:  d'(s, u) is shortest path distance from u to s
     using only X as intermediate nodes*)
    Let v be such that d'(s, v) = min_{u∈V−X} d'(s, u)
    dist(s, v) = d'(s, v)
    X = X ∪ {v}
    for each node u in V − X do
        d'(s, u) = min_{t∈X} (dist(s, t) + ℓ(t, u))
```

Algorithm

```
Initialize for each node v: dist(s, v) = ∞
Initialize X = ∅, d'(s, s) = 0
for i = 1 to |V| do
    (* Invariant: X contains the i − 1 closest nodes to s *)
    (* Invariant: d'(s, u) is shortest path distance from u to s
     using only X as intermediate nodes*)
    Let v be such that d'(s, v) = min_{u∈V−X} d'(s, u)
    dist(s, v) = d'(s, v)
    X = X ∪ {v}
    for each node u in V − X do
        d'(s, u) = min_{t∈X} (dist(s, t) + ℓ(t, u))
```

Correctness: By induction on i using previous lemmas.

Algorithm

```
Initialize for each node v: dist(s, v) = ∞
Initialize X = ∅, d'(s, s) = 0
for i = 1 to |V| do
    (* Invariant: X contains the i − 1 closest nodes to s *)
    (* Invariant: d'(s, u) is shortest path distance from u to s
     using only X as intermediate nodes*)
    Let v be such that d'(s, v) = min_{u∈V−X} d'(s, u)
    dist(s, v) = d'(s, v)
    X = X ∪ {v}
    for each node u in V − X do
        d'(s, u) = min_{t∈X} (dist(s, t) + ℓ(t, u))
```

Correctness: By induction on i using previous lemmas.

Running time:

Algorithm

```
Initialize for each node v:  dist(s, v) = ∞
Initialize X = ∅,  d'(s, s) = 0
for i = 1 to |V| do
    (* Invariant:  X contains the i − 1 closest nodes to s *)
    (* Invariant:  d'(s, u) is shortest path distance from u to s
     using only X as intermediate nodes*)
    Let v be such that d'(s, v) = min_{u∈V−X} d'(s, u)
    dist(s, v) = d'(s, v)
    X = X ∪ {v}
    for each node u in V − X do
        d'(s, u) = min_{t∈X} (dist(s, t) + ℓ(t, u))
```

Correctness: By induction on i using previous lemmas.

Running time: $O(n \cdot (n + m))$ time.

1. n outer iterations. In each iteration, $d'(s, u)$ for each u by scanning all edges out of nodes in X; $O(m + n)$ time/iteration.

Improved Algorithm

1. Main work is to compute the $d'(s, u)$ values in each iteration
2. $d'(s, u)$ changes from iteration i to $i + 1$ only because of the node v that is added to X in iteration i.

Dijkstra's Algorithm

1. eliminate $d'(s,u)$ and let $\mathrm{dist}(s,u)$ maintain it
2. update *dist* values after adding v by scanning edges out of v

```
Initialize for each node v, dist(s, v) = ∞
Initialize X = ∅, dist(s, s) = 0
for i = 1 to |V| do
    Let v be such that dist(s, v) = min_{u∈V−X} dist(s, u)
    X = X ∪ {v}
    for each u in Adj(v) do
        dist(s, u) = min(dist(s, u), dist(s, v) + ℓ(v, u))
```

Running time:

Dijkstra's Algorithm

1. eliminate $d'(s, u)$ and let $\mathrm{dist}(s, u)$ maintain it
2. update *dist* values after adding v by scanning edges out of v

```
Initialize for each node v, dist(s, v) = ∞
Initialize X = ∅, dist(s, s) = 0
for i = 1 to |V| do
    Let v be such that dist(s, v) = min_{u∈V−X} dist(s, u)
    X = X ∪ {v}
    for each u in Adj(v) do
        dist(s, u) = min(dist(s, u), dist(s, v) + ℓ(v, u))
```

Running time: $O(m + n^2)$ time.

1. n outer iterations and in each iteration following steps
2. updating $d'(s, u)$ after v is added takes $O(deg(v))$ time so total work is $O(m)$ since a node enters X only once
3. Finding v from $d'(s, u)$ values is $O(n)$ time

Dijkstra's Algorithm

1. eliminate $d'(s,u)$ and let $\mathrm{dist}(s,u)$ maintain it
2. update *dist* values after adding v by scanning edges out of v

```
Initialize for each node v, dist(s, v) = ∞
Initialize X = ∅, dist(s, s) = 0
for i = 1 to |V| do
    Let v be such that dist(s, v) = min_{u∈V−X} dist(s, u)
    X = X ∪ {v}
    for each u in Adj(v) do
        dist(s, u) = min(dist(s, u), dist(s, v) + ℓ(v, u))
```

Priority Queues to maintain *dist* values for faster running time

Dijkstra's Algorithm

1. eliminate $d'(s, u)$ and let $\text{dist}(s, u)$ maintain it
2. update *dist* values after adding v by scanning edges out of v

```
Initialize for each node v, dist(s, v) = ∞
Initialize X = ∅, dist(s, s) = 0
for i = 1 to |V| do
    Let v be such that dist(s, v) = min_{u∈V−X} dist(s, u)
    X = X ∪ {v}
    for each u in Adj(v) do
        dist(s, u) = min(dist(s, u), dist(s, v) + ℓ(v, u))
```

Priority Queues to maintain *dist* values for faster running time

1. Using heaps and standard priority queues: $O((m + n) \log n)$
2. Using Fibonacci heaps: $O(m + n \log n)$.

Priority Queues

Data structure to store a set S of n elements where each element $v \in S$ has an associated real/integer key $k(v)$ such that the following operations:

1. **makePQ**: create an empty queue.
2. **findMin**: find the minimum key in S.
3. **extractMin**: Remove $v \in S$ with smallest key and return it.
4. **insert**$(v, k(v))$: Add new element v with key $k(v)$ to S.
5. **delete**(v): Remove element v from S.

Priority Queues

Data structure to store a set S of n elements where each element $v \in S$ has an associated real/integer key $k(v)$ such that the following operations:

1. **makePQ**: create an empty queue.
2. **findMin**: find the minimum key in S.
3. **extractMin**: Remove $v \in S$ with smallest key and return it.
4. **insert**$(v, k(v))$: Add new element v with key $k(v)$ to S.
5. **delete**(v): Remove element v from S.
6. **decreaseKey**$(v, k'(v))$: *decrease* key of v from $k(v)$ (current key) to $k'(v)$ (new key). Assumption: $k'(v) \leq k(v)$.
7. **meld**: merge two separate priority queues into one.

Priority Queues

Data structure to store a set S of n elements where each element $v \in S$ has an associated real/integer key $k(v)$ such that the following operations:

1. **makePQ**: create an empty queue.
2. **findMin**: find the minimum key in S.
3. **extractMin**: Remove $v \in S$ with smallest key and return it.
4. **insert**$(v, k(v))$: Add new element v with key $k(v)$ to S.
5. **delete**(v): Remove element v from S.
6. **decreaseKey**$(v, k'(v))$: *decrease* key of v from $k(v)$ (current key) to $k'(v)$ (new key). Assumption: $k'(v) \leq k(v)$.
7. **meld**: merge two separate priority queues into one.

All operations can be performed in $O(\log n)$ time.

decreaseKey is implemented via **delete** and **insert**.

Dijkstra's Algorithm using Priority Queues

```
Q ← makePQ()
insert(Q, (s,0))
for each node u ≠ s do
    insert(Q, (u,∞))
X ← ∅
for i = 1 to |V| do
    (v, dist(s, v)) = extractMin(Q)
    X = X ∪ {v}
    for each u in Adj(v) do
        If (u ∉ X) do
            decreaseKey(Q, (u, min(k(u), dist(s, v) + ℓ(v, u)))).
```

Priority Queue operations:

1. $O(n)$ **insert** operations
2. $O(n)$ **extractMin** operations
3. $O(m)$ **decreaseKey** operations

Implementing Priority Queues via Heaps

Using Heaps

Store elements in a heap based on the key value

1. All operations can be done in $O(\log n)$ time

Implementing Priority Queues via Heaps

Using Heaps

Store elements in a heap based on the key value

1. All operations can be done in $O(\log n)$ time

Dijkstra's algorithm can be implemented in $O((n+m)\log n)$ time.

Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

1. **extractMin**, **insert**, **delete**, **meld** in $O(\log n)$ time
2. **decreaseKey** in $O(1)$ *amortized* time:

Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

1. **extractMin**, **insert**, **delete**, **meld** in $O(\log n)$ time
2. **decreaseKey** in $O(1)$ *amortized* time: ℓ **decreaseKey** operations for $\ell \geq n$ take *together* $O(\ell)$ time
3. Relaxed Heaps: **decreaseKey** in $O(1)$ worst case time but at the expense of **meld** (not necessary for Dijkstra's algorithm)

Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

1. **extractMin**, **insert**, **delete**, **meld** in $O(\log n)$ time
2. **decreaseKey** in $O(1)$ *amortized* time: ℓ **decreaseKey** operations for $\ell \geq n$ take *together* $O(\ell)$ time
3. Relaxed Heaps: **decreaseKey** in $O(1)$ worst case time but at the expense of **meld** (not necessary for Dijkstra's algorithm)

1. Dijkstra's algorithm can be implemented in $O(n \log n + m)$ time. If $m = \Omega(n \log n)$, running time is linear in input size.

Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

1. **extractMin**, **insert**, **delete**, **meld** in $O(\log n)$ time
2. **decreaseKey** in $O(1)$ *amortized* time: ℓ **decreaseKey** operations for $\ell \geq n$ take *together* $O(\ell)$ time
3. Relaxed Heaps: **decreaseKey** in $O(1)$ worst case time but at the expense of **meld** (not necessary for Dijkstra's algorithm)

1. Dijkstra's algorithm can be implemented in $O(n \log n + m)$ time. If $m = \Omega(n \log n)$, running time is linear in input size.
2. Data structures are complicated to analyze/implement. Recent work has obtained data structures that are easier to analyze and implement, and perform well in practice. Rank-Pairing Heaps (European Symposium on Algorithms, September 2009!)

Shortest Path Tree

Dijkstra's algorithm finds the shortest path distances from s to V.
Question: How do we find the paths themselves?

Shortest Path Tree

Dijkstra's algorithm finds the shortest path distances from s to V.

Question: How do we find the paths themselves?

```
Q = makePQ()
insert(Q, (s,0))
prev(s) ← null
for each node u ≠ s do
    insert(Q, (u,∞) )
    prev(u) ← null

X = ∅
for i = 1 to |V| do
    (v, dist(s,v)) = extractMin(Q)
    X = X ∪ {v}
    for each u in Adj(v) do
        if (dist(s,v) + ℓ(v,u) < dist(s,u)) then
            decreaseKey(Q, (u, dist(s,v) + ℓ(v,u)))
            prev(u) = v
```

Shortest Path Tree

Lemma

The edge set $(u, \mathrm{prev}(u))$ *is the* reverse *of a shortest path tree rooted at* s*. For each* u*, the reverse of the path from* u *to* s *in the tree is a shortest path from* s *to* u*.*

Proof Sketch.

1. The edge set $\{(u, \mathrm{prev}(u)) \mid u \in V\}$ induces a directed in-tree rooted at s (Why?)
2. Use induction on $|X|$ to argue that the tree is a shortest path tree for nodes in V.

□

Shortest paths to s

Dijkstra's algorithm gives shortest paths from s to all nodes in V.
How do we find shortest paths from all of V to s?

Shortest paths to s

Dijkstra's algorithm gives shortest paths from s to all nodes in V.
How do we find shortest paths from all of V to s?

1. In undirected graphs shortest path from s to u is a shortest path from u to s so there is no need to distinguish.
2. In directed graphs, use Dijkstra's algorithm in G^{rev}!

Shortest paths between sets of nodes

Suppose we are given $S \subset V$ and $T \subset V$. Want to find shortest path from S to T defined as:

$$\mathbf{dist}(S, T) = \min_{s \in S, t \in T} \mathbf{dist}(s, t)$$

How do we find $\mathbf{dist}(S, T)$?

Example Problem

You want to go from your house to a friend's house. Need to pick up some dessert along the way and hence need to stop at one of the many potential stores along the way. How do you calculate the "shortest" trip if you include this stop?

Example Problem

You want to go from your house to a friend's house. Need to pick up some dessert along the way and hence need to stop at one of the many potential stores along the way. How do you calculate the "shortest" trip if you include this stop?

Given $G = (V, E)$ and edge lengths $\ell(e), e \in E$. Want to go from s to t. A subset $X \subset V$ that corresponds to stores. Want to find $\min_{x \in X} d(s, x) + d(x, t)$.

Example Problem

You want to go from your house to a friend's house. Need to pick up some dessert along the way and hence need to stop at one of the many potential stores along the way. How do you calculate the "shortest" trip if you include this stop?
Given $G = (V, E)$ and edge lengths $\ell(e), e \in E$. Want to go from s to t. A subset $X \subset V$ that corresponds to stores. Want to find $\min_{x \in X} d(s, x) + d(x, t)$.

Basic solution: Compute for each $x \in X$, $d(s, x)$ and $d(x, t)$ and take minimum. $2|X|$ shortest path computations.
$O(|X|(m + n \log n))$.

Example Problem

You want to go from your house to a friend's house. Need to pick up some dessert along the way and hence need to stop at one of the many potential stores along the way. How do you calculate the "shortest" trip if you include this stop?
Given $G = (V, E)$ and edge lengths $\ell(e), e \in E$. Want to go from s to t. A subset $X \subset V$ that corresponds to stores. Want to find $\min_{x \in X} d(s, x) + d(x, t)$.

Basic solution: Compute for each $x \in X$, $d(s, x)$ and $d(x, t)$ and take minimum. $2|X|$ shortest path computations.
$O(|X|(m + n \log n))$.

Better solution: Compute shortest path distances from s to every node $v \in V$ with one Dijkstra. Compute from every node $v \in V$ shortest path distance to t with one Dijkstra.