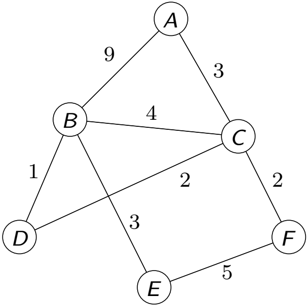
Graphs: Dijkstra's Algorithm

March 4, 2011

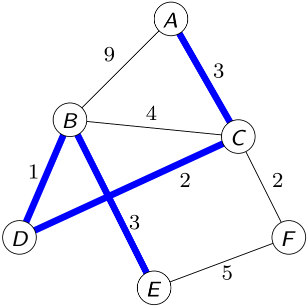
Consider a graph with weighted edges.



Which path from A to E has smallest total weight?

An Example Graph

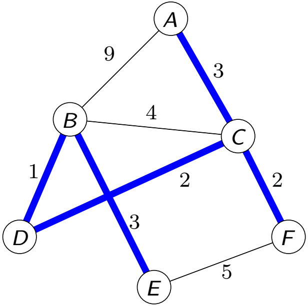
Consider a graph with weighted edges.



Minimal path from A to E has weight 9.

Single Source Shortest Paths

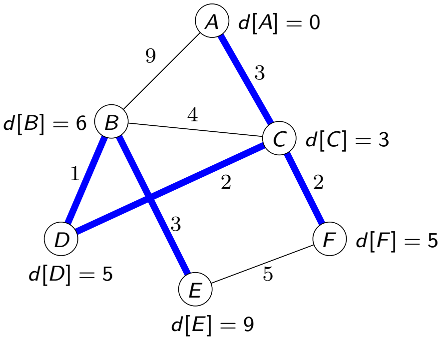
Dijkstra's algorithm will build a tree of shortest paths from A (source) to each other node.



It will also compute shortest distances to each node from A.

Single Source Shortest Paths

Dijkstra's algorithm will build a tree of shortest paths from A (source) to each other node.



It will also compute shortest distances to each node from A.

Dijkstra: The Main Idea

Same basic idea as we used for Breadth-First Search:

- T stores nodes we have finished handling, and
- Q stores nodes we would like to handle next.

Initially, $T = \{\}$ and $Q = \{s\}$, where is s is the source.

Repeatedly remove a node from Q, process it, and add it to T.

Main difference

- BFS used a FIFO queue for Q
- Dijkstra uses a priority queue for Q

Dijkstra: The Main Idea

Same basic idea as we used for Breadth-First Search:

- T stores nodes we have finished handling, and
- Q stores nodes we would like to handle next.

Initially, $T = \{\}$ and $Q = \{s\}$, where is s is the source.

Repeatedly remove a node from Q, process it, and add it to T.

Main difference

- BFS used a FIFO queue for Q
- Dijkstra uses a priority queue for Q

Dijkstra: The Main Idea

Same basic idea as we used for Breadth-First Search:

- T stores nodes we have finished handling, and
- Q stores nodes we would like to handle next.

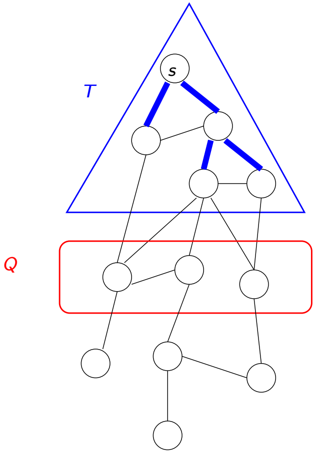
Initially, $T = \{\}$ and $Q = \{s\}$, where is s is the source.

Repeatedly remove a node from Q, process it, and add it to T.

Main difference

- BFS used a FIFO queue for Q
- Dijkstra uses a priority queue for Q

Invariant

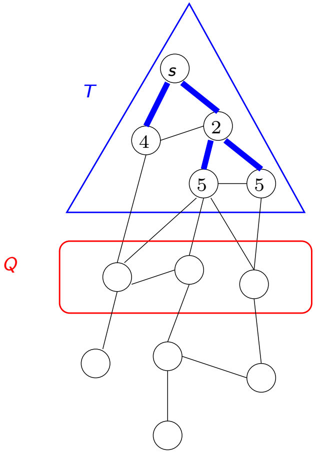


Invariants:
We know the correct distance from source to each node in T.

For each node u in Q, we know the length of the shortest path from s to u that stays inside T.

Claim: For the u with the smallest such length, that length really is the shortest distance from s to u.
$\Rightarrow$ Process that node u next.

Invariant



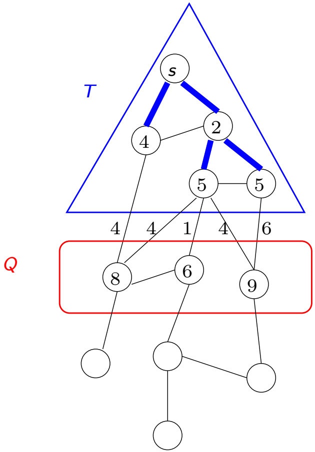
Invariants:

We know the correct distance from source to each node in T.

For each node u in Q, we know the length of the shortest path from s to u that stays inside T.

Claim: For the u with the smallest such length, that length really is the shortest distance from s to u.

$\Rightarrow$ Process that node u next.

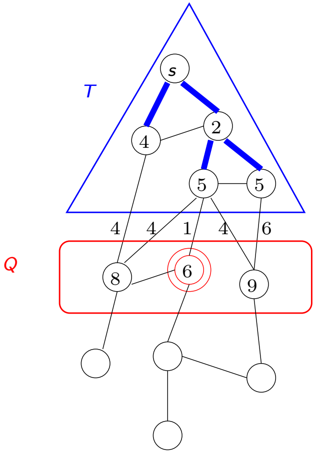


Invariants:
We know the correct distance from source to each node in T.

For each node u in Q, we know the length of the shortest path from s to u that stays inside T.

Claim: For the u with the smallest such length, that length really is the shortest distance from s to u.
$\Rightarrow$ Process that node u next.

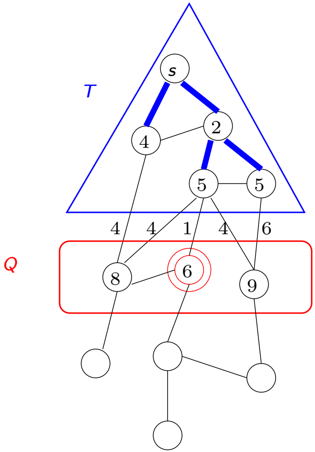
Invariant



Invariants:
We know the correct distance from source to each node in T.

For each node u in Q, we know the length of the shortest path from s to u that stays inside T.

Claim: For the u with the smallest such length, that length really is the shortest distance from s to u.
$\Rightarrow$ Process that node u next.



Invariants:
We know the correct distance from source to each node in T.

For each node u in Q, we know the length of the shortest path from s to u that stays inside T.

Claim: For the u with the smallest such length, that length really is the shortest distance from s to u.
$\Rightarrow$ Process that node u next.

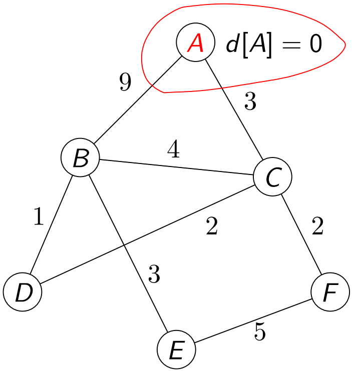
Pseudocode for Dijkstra's Algorithm

// $d[v]$ will eventually store distance from s to v

$d[s] = 0$
$Q = \{s\}$
$T = \{\}$
while Q is not empty
    remove from Q the node u with the lowest d-value
    add u to T
    update d-values of neighbours of u in Q
    for each edge $u \rightarrow v$
        if v not already in $Q \cup T$ then
            add v to Q with $d[v] = d[u] + weight[u, v]$
        else if v is already in Q and $d[v] > d[u] + weight[u, v]$ then
            change $d[v]$ to $d[u] + weight[u, v]$
    end for
end while

Pseudocode for Dijkstra's Algorithm

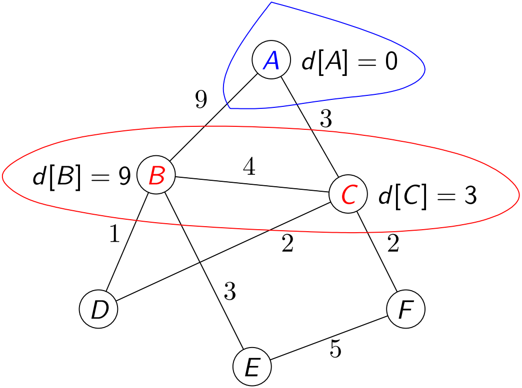
```
// d[v] will eventually store distance from s to v

d[s] = 0
Q = {s}
T = {}
while Q is not empty
    remove from Q the node u with the lowest d-value
    add u to T
    // update d-values of neighbours of u in Q
    for each edge u → v
        if v not already in Q ∪ T then
            add v to Q with d[v] = d[u] + weight[u, v]
        else if v is already in Q and d[v] > d[u] + weight[u, v] then
            change d[v] to d[u] + weight[u, v]
    end for
end while
```



$T = \{\}$

$Q = \{A\}$

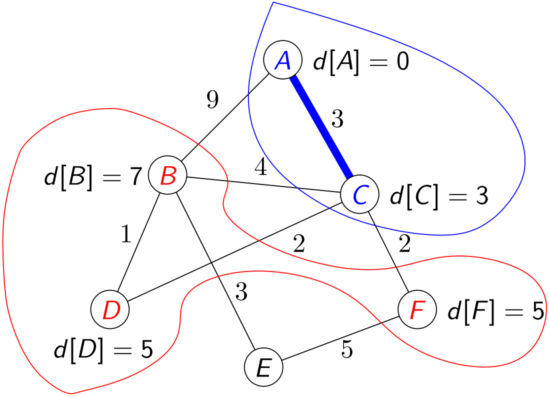
An Example Execution



$T = \{A\}$

$Q = \{B, C\}$

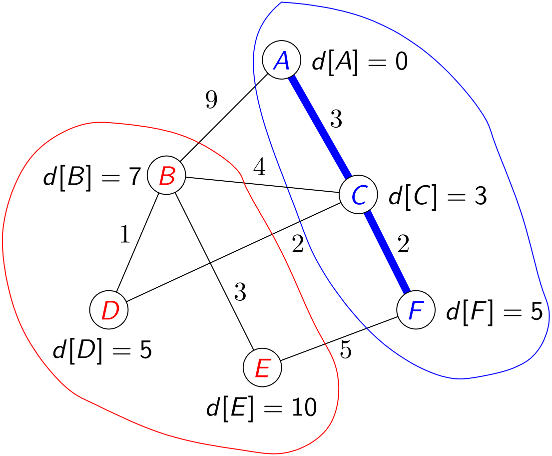
An Example Execution



$T = \{A, C\}$

$Q = \{B, D, F\}$

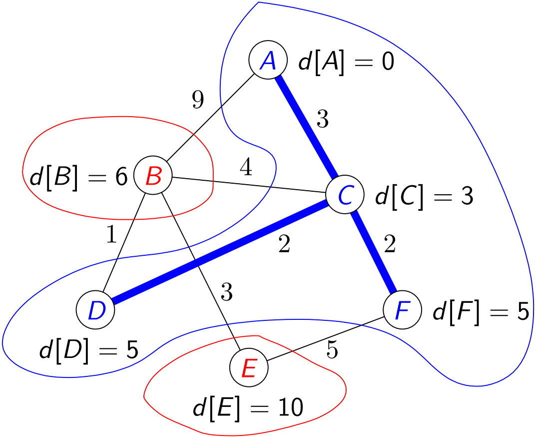
An Example Execution



$T = \{A, C, F\}$

$Q = \{B, D, E\}$

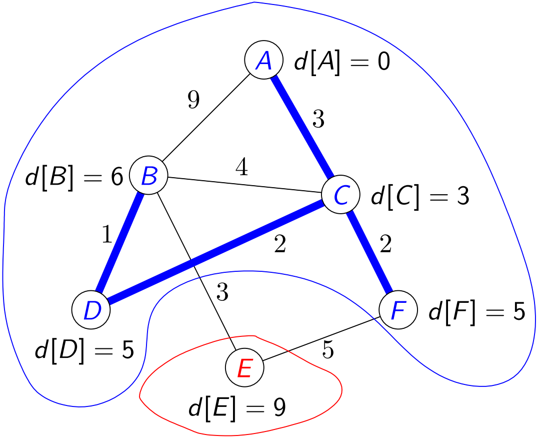
An Example Execution



$T = \{A, C, D, F\}$

$Q = \{B, E\}$

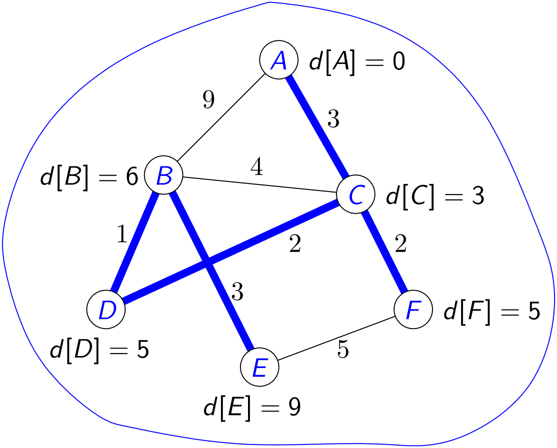
An Example Execution



$T = \{A, B, C, D, F\}$

$Q = \{E\}$

An Example Execution



$T = \{A, B, C, D, E, F\}$

$Q = \{\}$

Java Implementation

We use a PriorityQueue to pick next node in each iteration.

- This requires creating Node objects that can be stored in the PriorityQueue.
- We also design comparison operators for Node objects that compare them according to their *d*-values (so that the PriorityQueue knows how to order nodes).

We also store parent of each node in shortest path tree.
This information can be used to reconstruct shortest paths.
(Same as for BFS a few weeks ago.)

The algorithm given is designed for directed graphs.
For undirected graphs, just add edges in both directions.

Java Implementation

We use a PriorityQueue to pick next node in each iteration.

- This requires creating Node objects that can be stored in the PriorityQueue.
- We also design comparison operators for Node objects that compare them according to their d-values (so that the PriorityQueue knows how to order nodes).

We also store parent of each node in shortest path tree.
This information can be used to reconstruct shortest paths.
(Same as for BFS a few weeks ago.)

The algorithm given is designed for directed graphs.
For undirected graphs, just add edges in both directions.

Java Implementation

We use a PriorityQueue to pick next node in each iteration.

- This requires creating Node objects that can be stored in the PriorityQueue.
- We also design comparison operators for Node objects that compare them according to their d-values (so that the PriorityQueue knows how to order nodes).

We also store parent of each node in shortest path tree.
This information can be used to reconstruct shortest paths.
(Same as for BFS a few weeks ago.)

The algorithm given is designed for directed graphs.
For undirected graphs, just add edges in both directions.