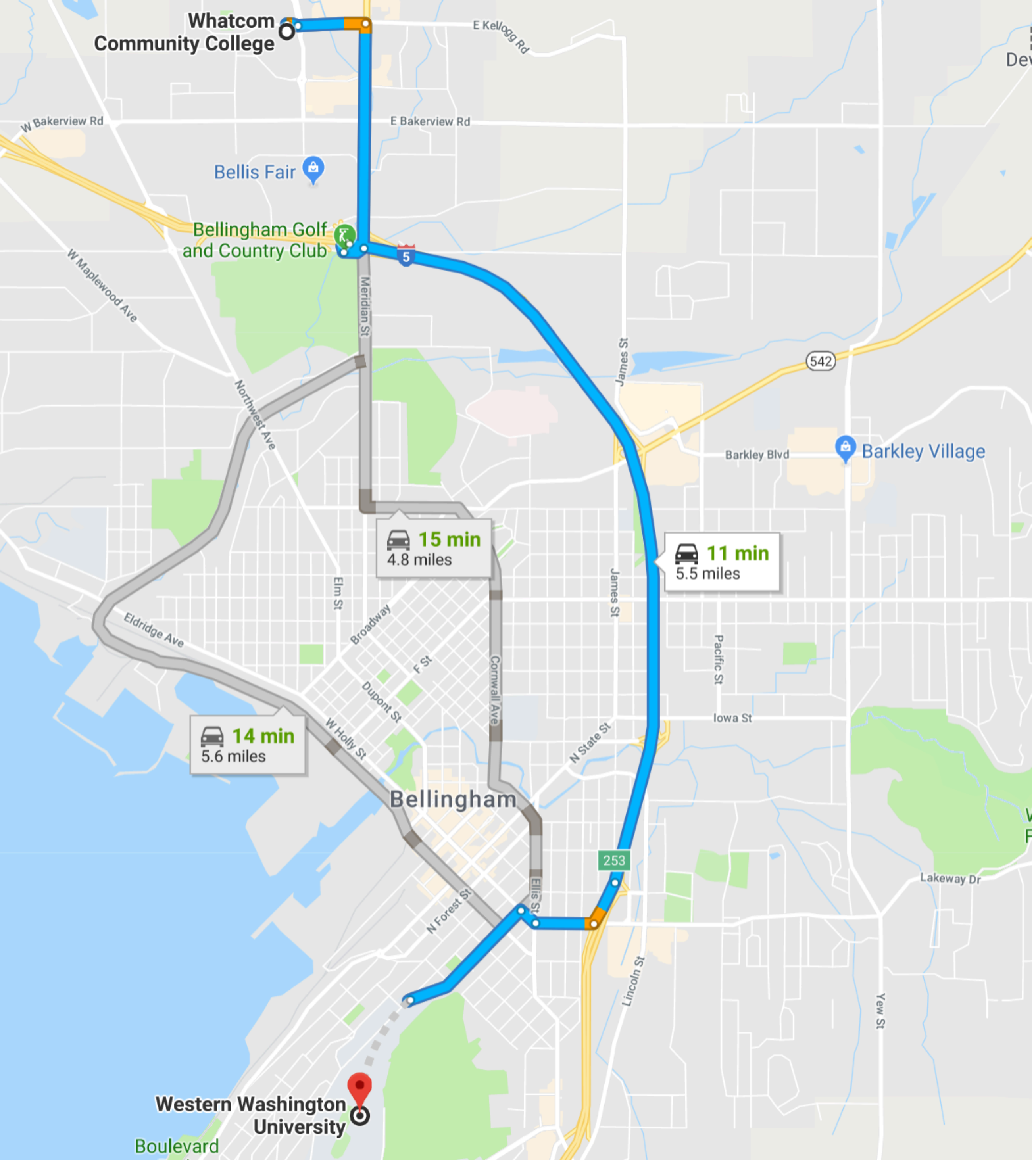


CSCI 241

Lecture 21

Dijkstra's Single-Source Shortest Paths Algorithm

Announcements

Announcements

- Lab 8 is out.

Announcements

- Lab 8 is out.
- A4 is out
  - I’ll post full slides for Dijkstra even if we don't get through all of them today.
  - I'll also post two sample graphs for you to run the algorithm on.

Announcements

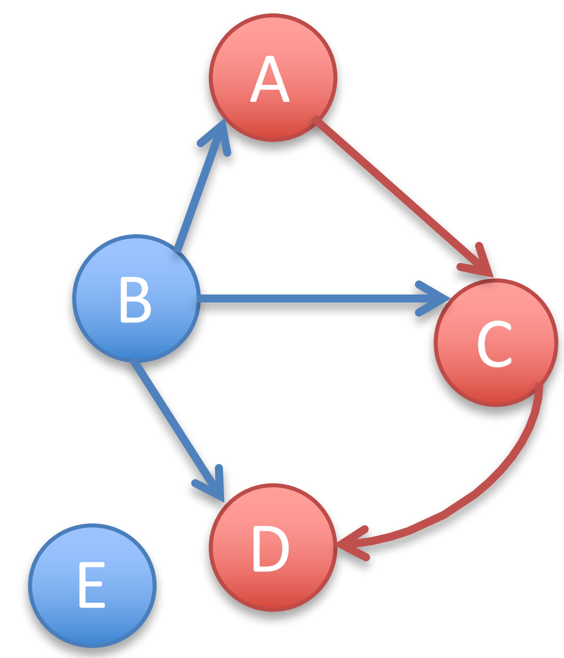
- Lab 8 is out.
- A4 is out
  - I’ll post full slides for Dijkstra even if we don't get through all of them today.
  - I'll also post two sample graphs for you to run the algorithm on.
- Quiz 5 is graded, video is posted.

Goals

- Know how to determine whether a graph is connected
- Know the definition of connected components.
- Know what a weighted graph is.
- Understand the intuition behind Dijkstra’s shortest paths algorithm.
- Be able to execute Dijkstra’s algorithm manually on a graph.
- Be prepared to implement Dijkstra's algorithm efficiently.
- Know how to augment the algorithm to keep backpointers in order to reconstruct the sequence of nodes in a shortest path.

Graph Terminology

- A graph is connected if there is a path between every pair of nodes.
  - A directed graph is strongly connected if there is a directed path between all pairs of nodes.
  - A directed graph is weakly connected if the graph becomes connected when all edges are converted to undirected edges.
- A graph can have multiple connected components: subsets of the vertices and edges that are connected.

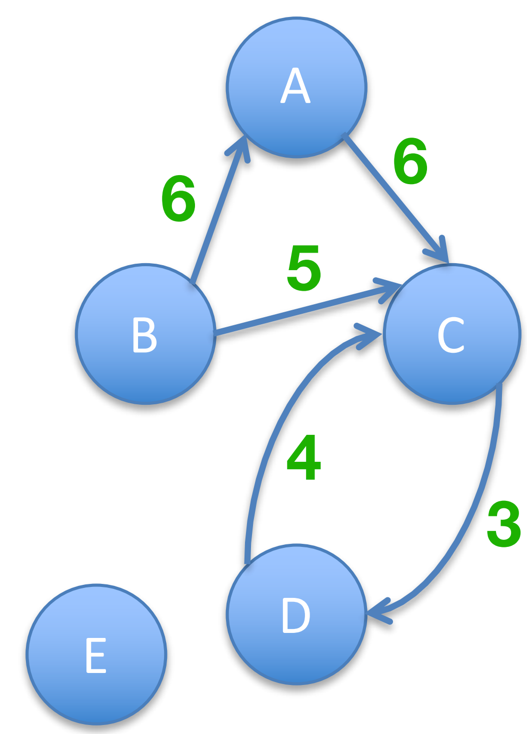


Not strongly connected

Not weakly connected

Weighted Graphs

- Like a normal graph, but edges have weights.
- Formally: a graph (V,E) with an accompanying weight function w: E -> $\mathbb{R}$
  - may be directed or undirected.

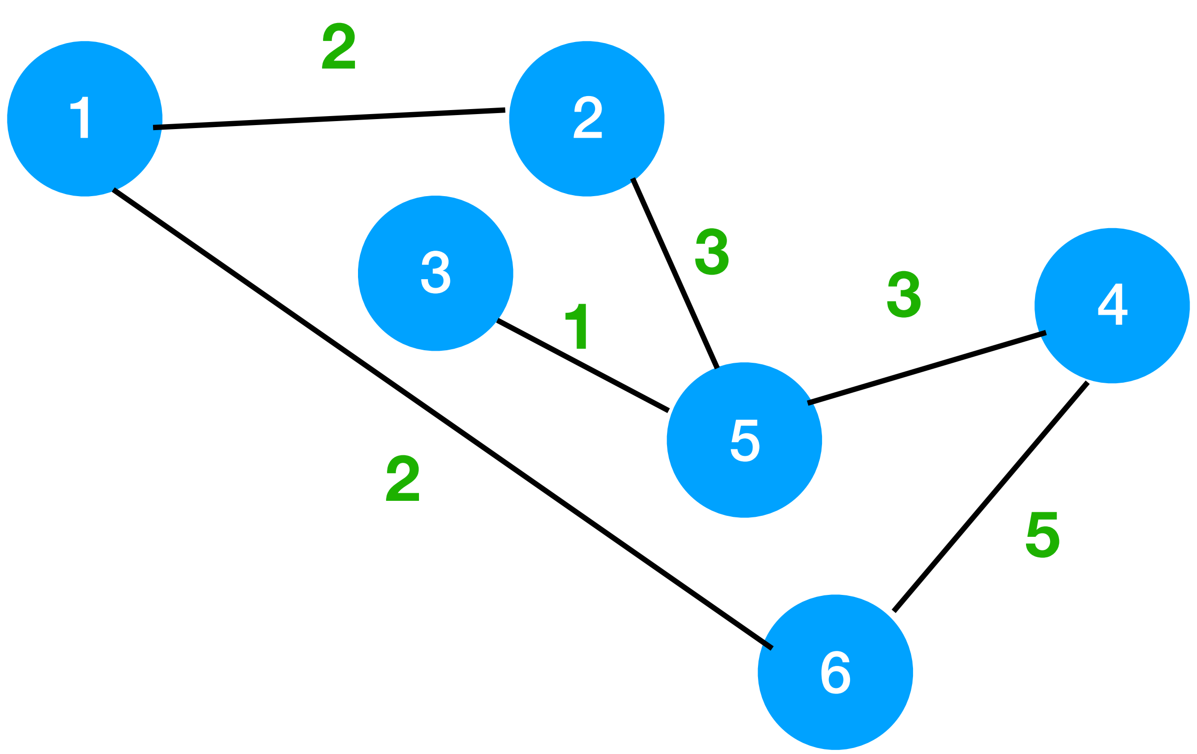


- Informally: label edges with their weights
- Representation:
  - adjacency list - store weight of (u,v) with v the node in u's list
  - adjacency matrix - store weight in matrix entry for (u,v)

Paths in Weighted Graphs

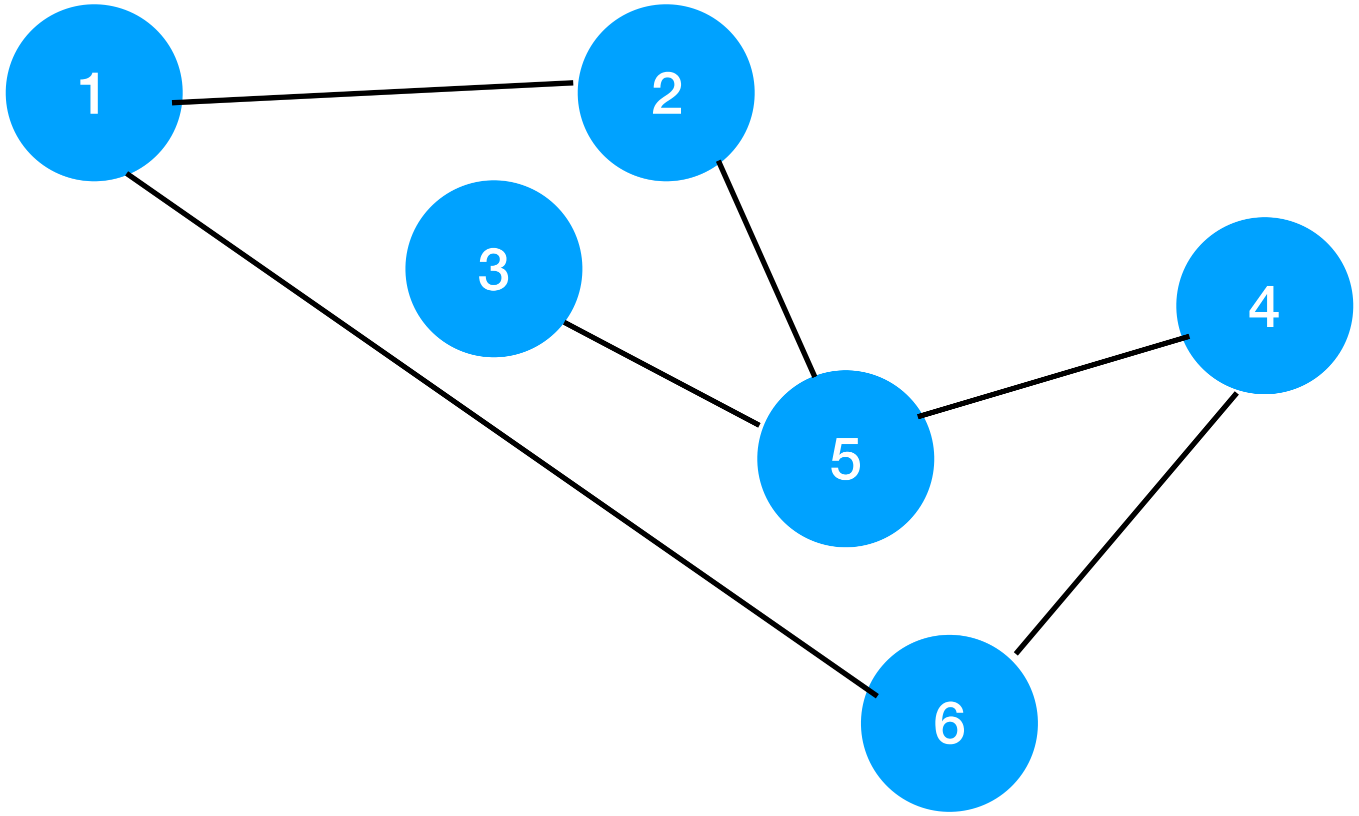
- The length (or weight) of a path in a weighted graph is the sum of the edge weights along that path.

- **ABCD**: What's the length of the shortest path from 3 to 6?
  A. 7
  B. 8
  C. 9
  D. 10



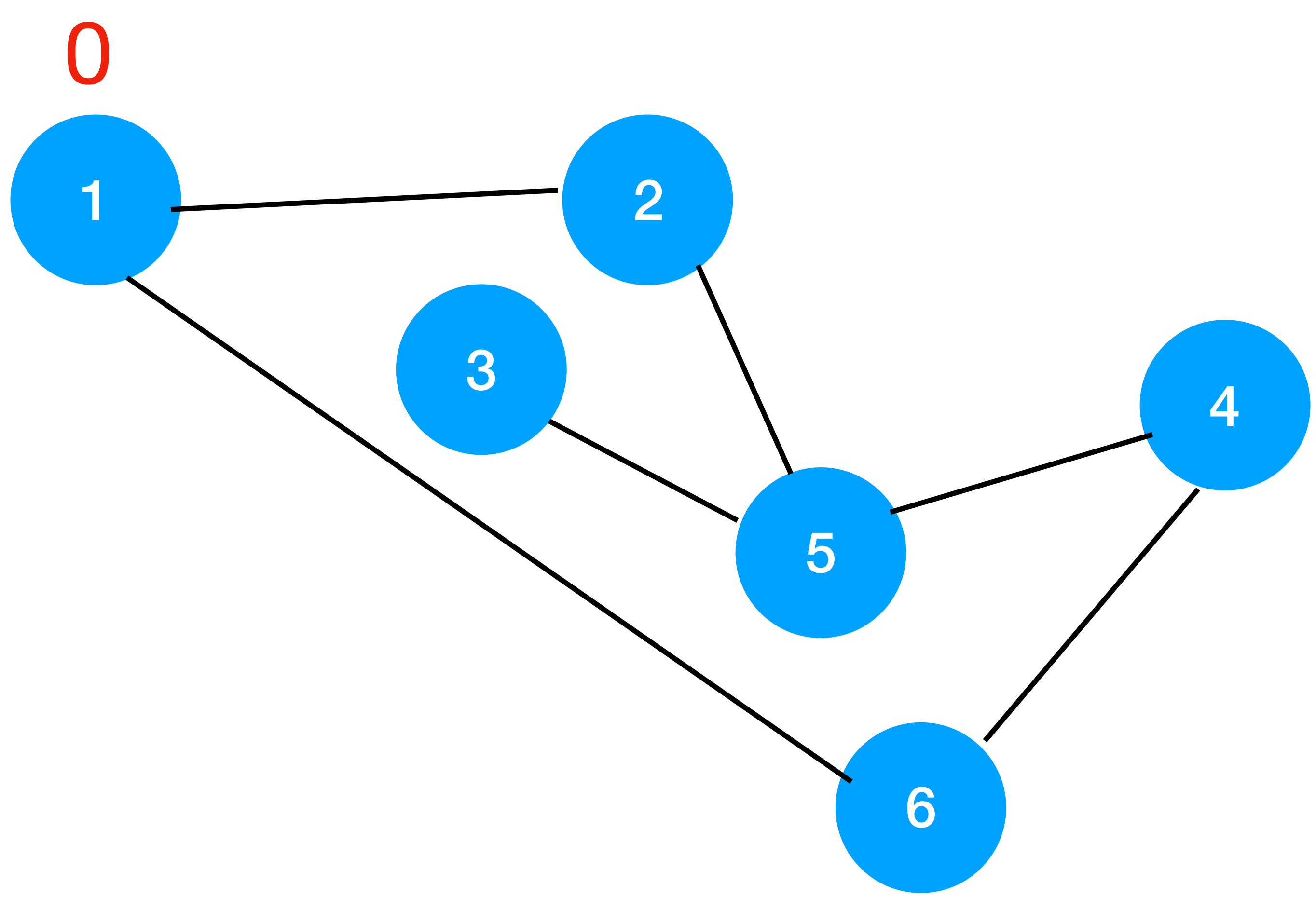
Computing Shortest Paths in Unweighted Graphs

- Perform a breadth-first search (that's it!)
- BFS visits nodes in order of "hop distance", or path length!
- BFS(1):



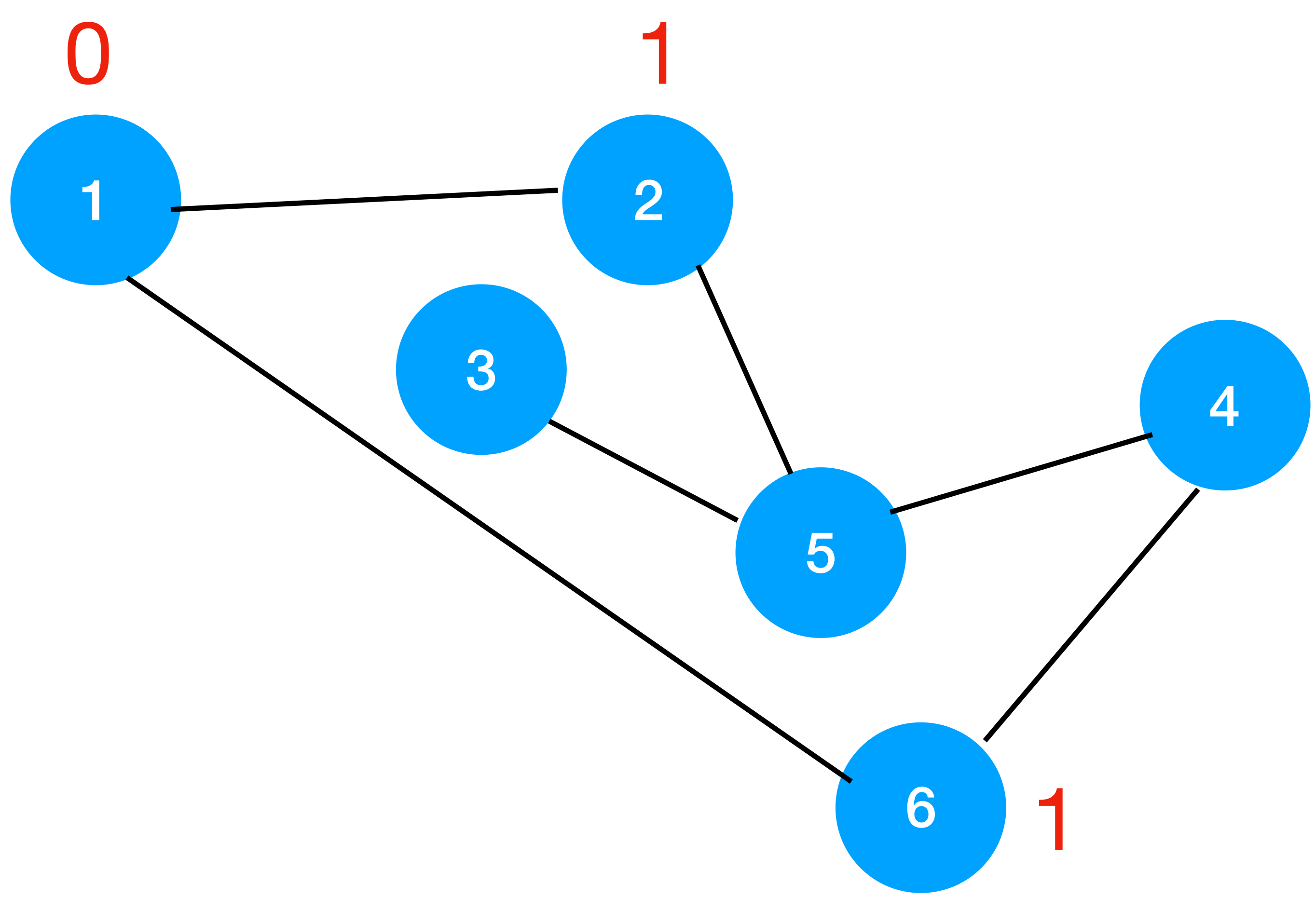
Computing Shortest Paths in Unweighted Graphs

- Perform a breadth-first search (that's it!)
- BFS visits nodes in order of "hop distance", or path length!
- BFS(1):



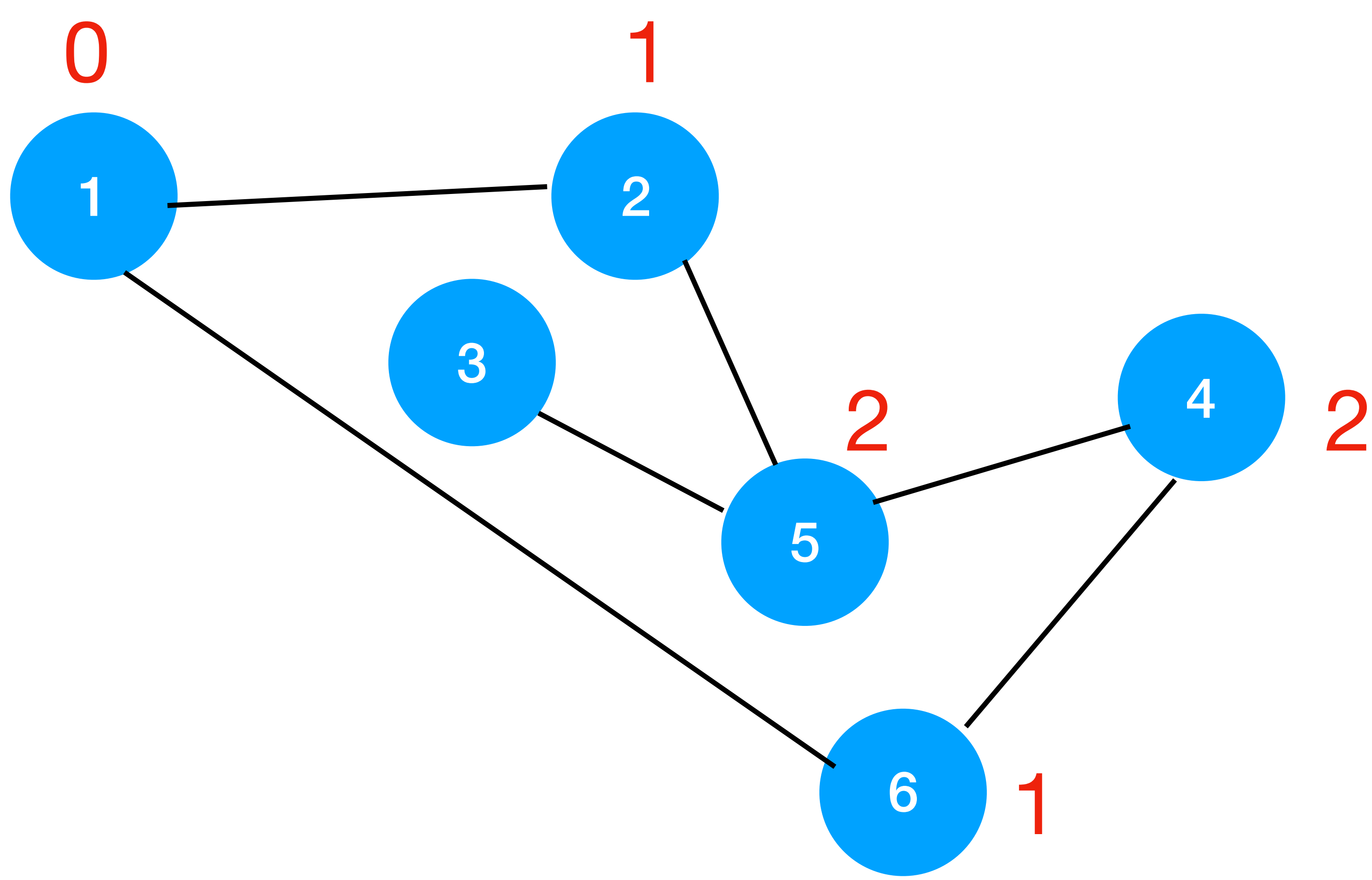
Computing Shortest Paths in Unweighted Graphs

- Perform a breadth-first search (that's it!)
- BFS visits nodes in order of "hop distance", or path length!
- BFS(1):



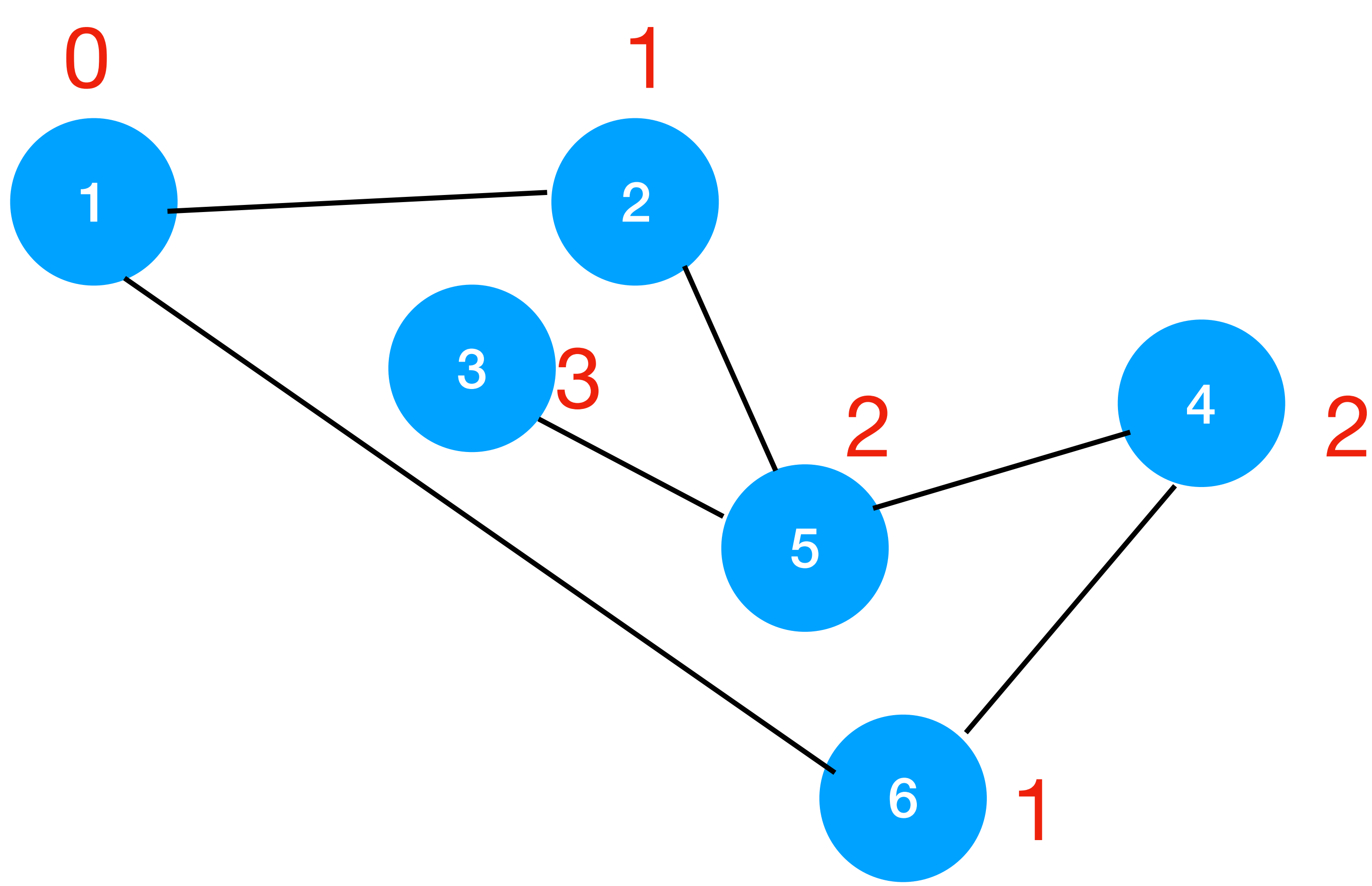
Computing Shortest Paths in Unweighted Graphs

- Perform a breadth-first search (that's it!)
- BFS visits nodes in order of "hop distance", or path length!
- BFS(1):



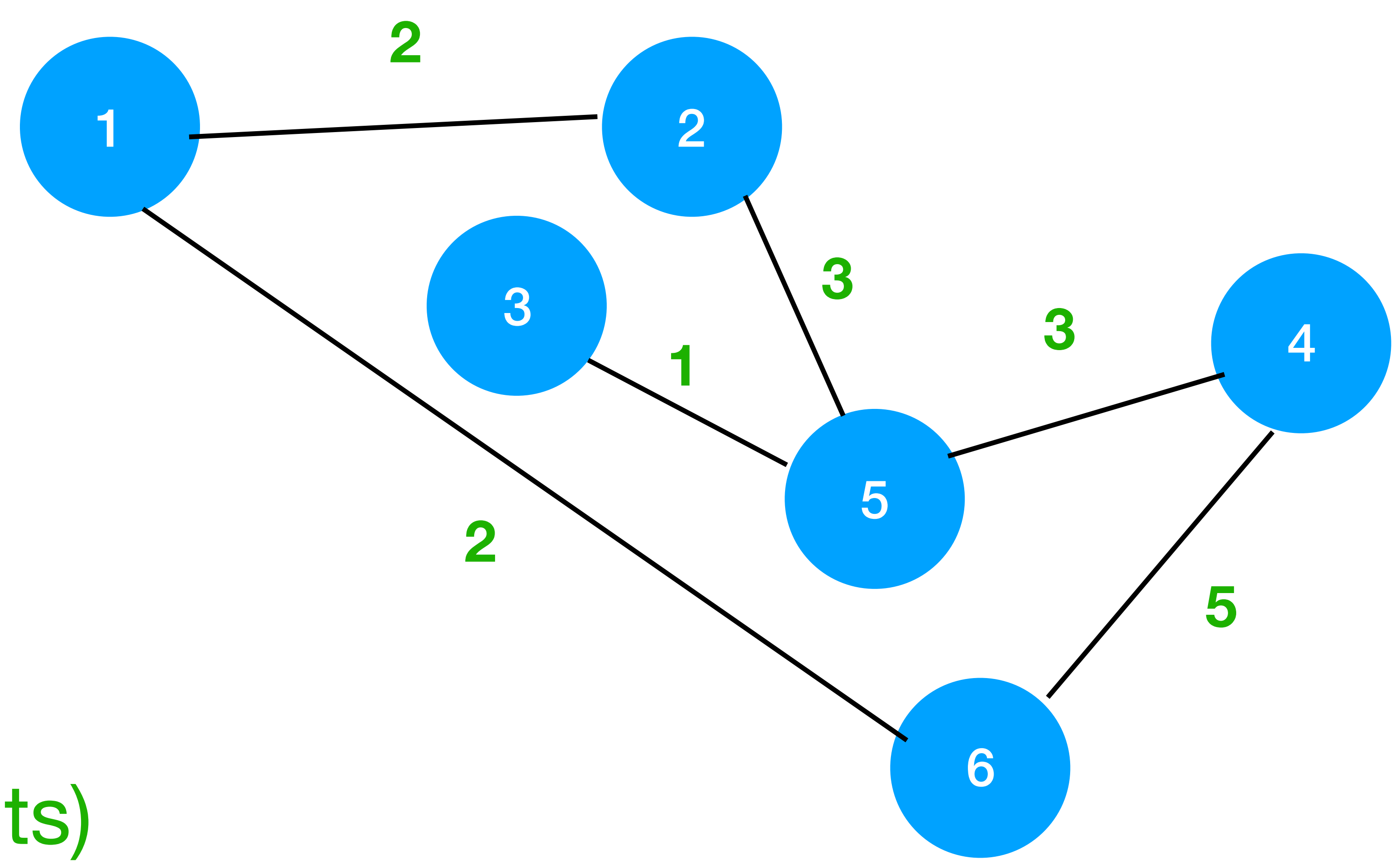
Computing Shortest Paths in Unweighted Graphs

- Perform a breadth-first search (that's it!)
- BFS visits nodes in order of "hop distance", or path length!
- BFS(1):



Computing Shortest Paths in Weighted Graphs

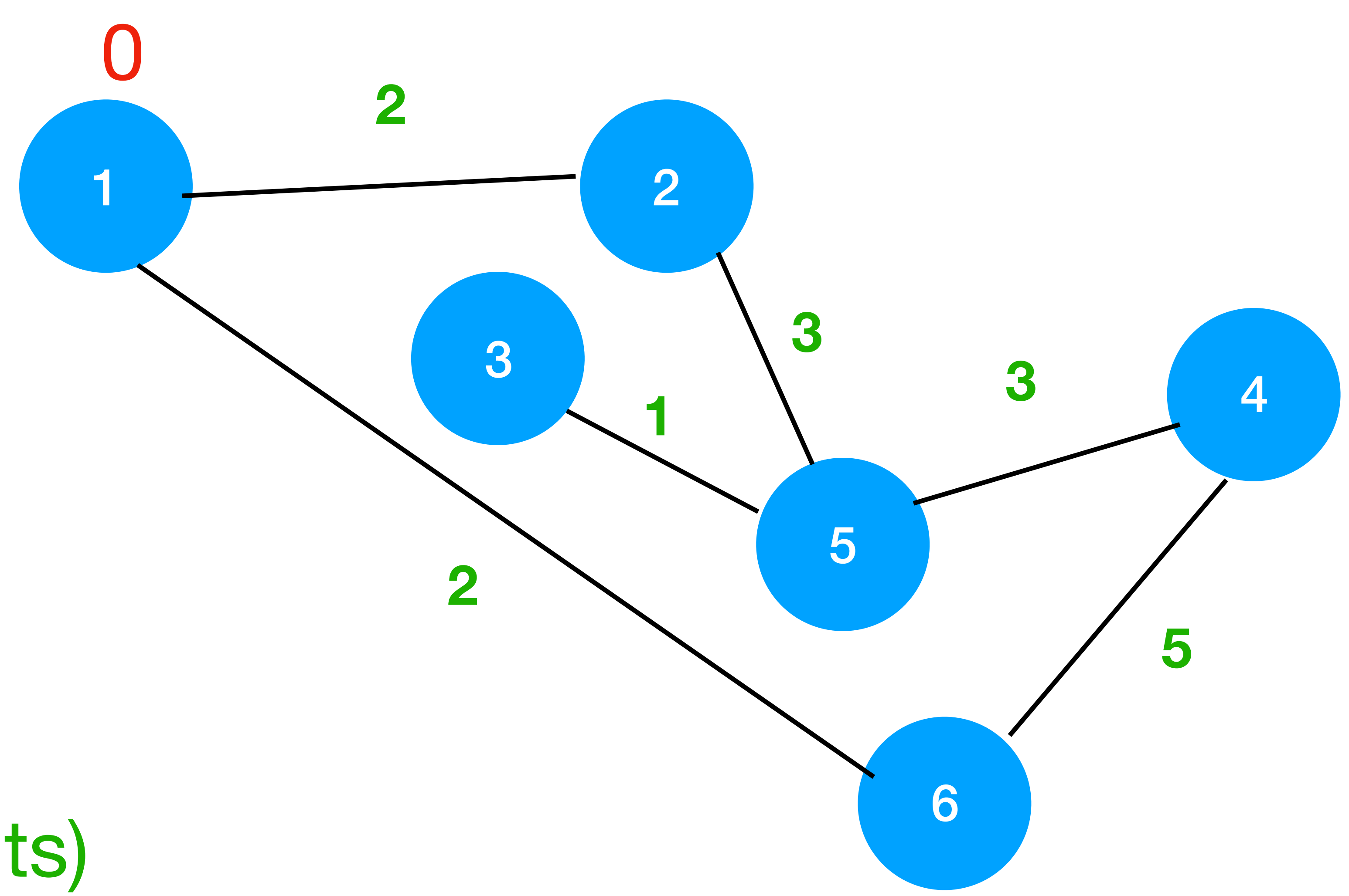
BFS doesn't visit nodes in order of shortest path length:



(edge weights)
(shortest path length from node 1)

Computing Shortest Paths in Weighted Graphs

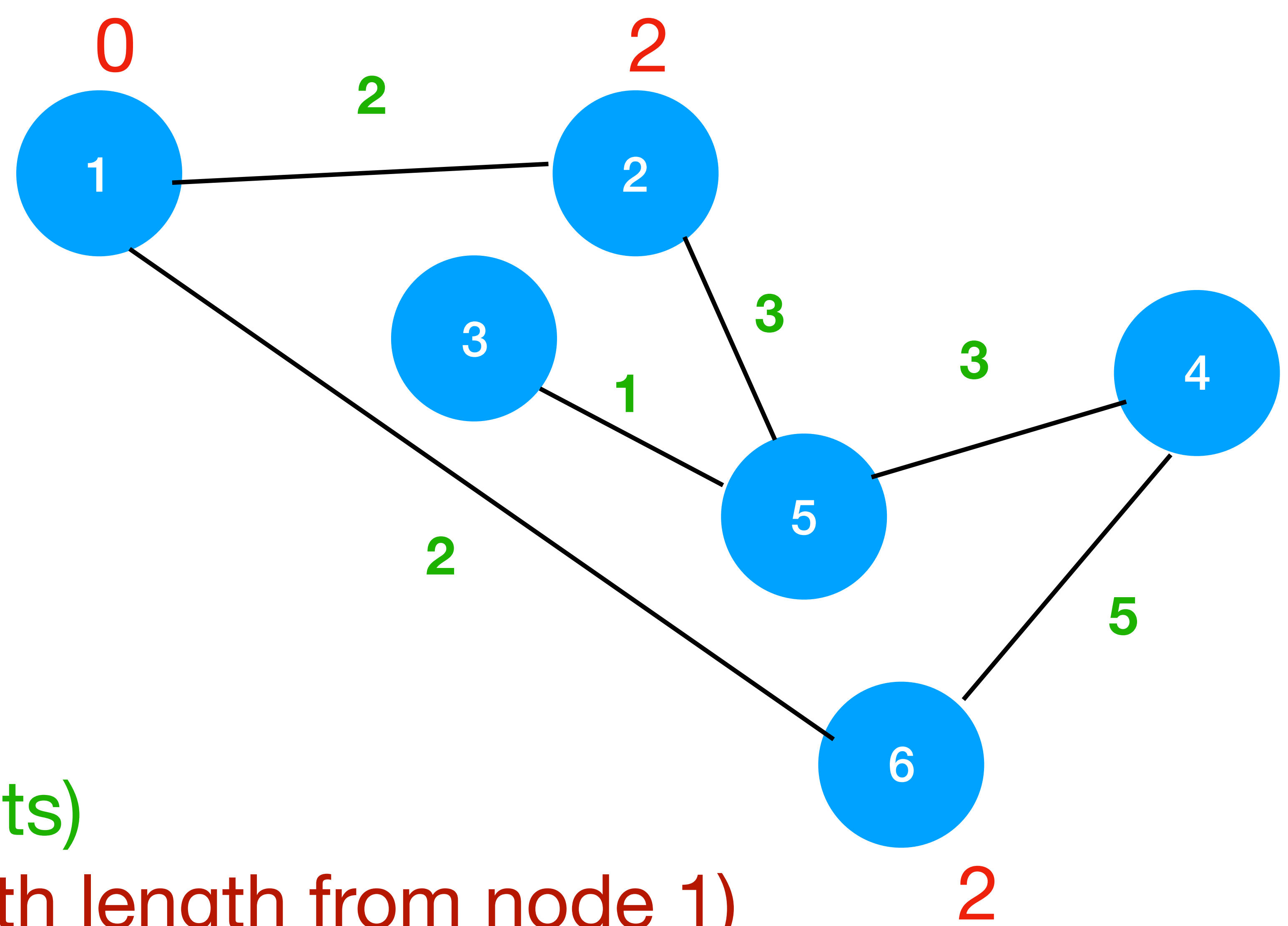
BFS doesn't visit nodes in order of shortest path length:



(edge weights)
(shortest path length from node 1)

Computing Shortest Paths in Weighted Graphs

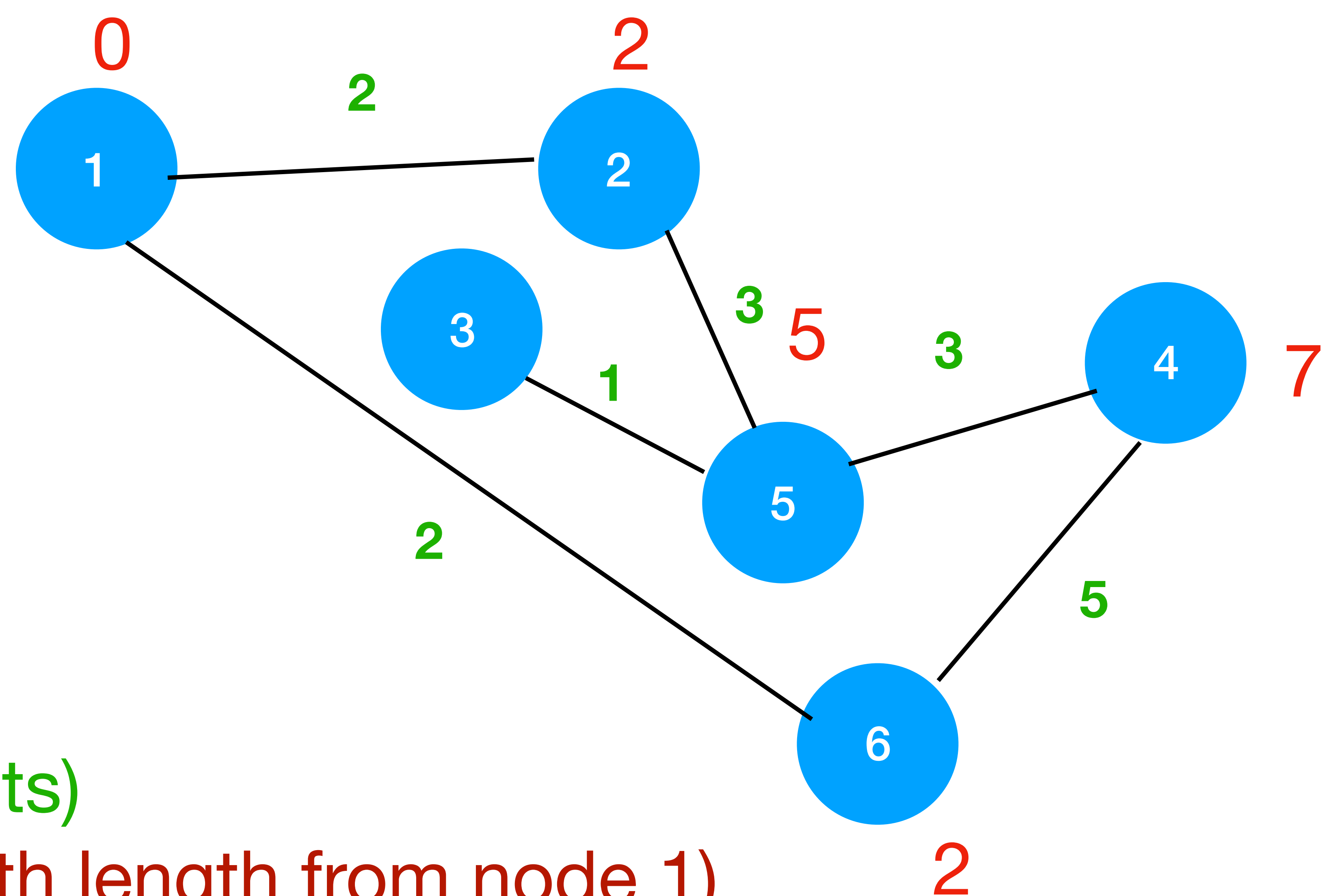
BFS doesn't visit nodes in order of shortest path length:



(edge weights)
(shortest path length from node 1)

Computing Shortest Paths in Weighted Graphs

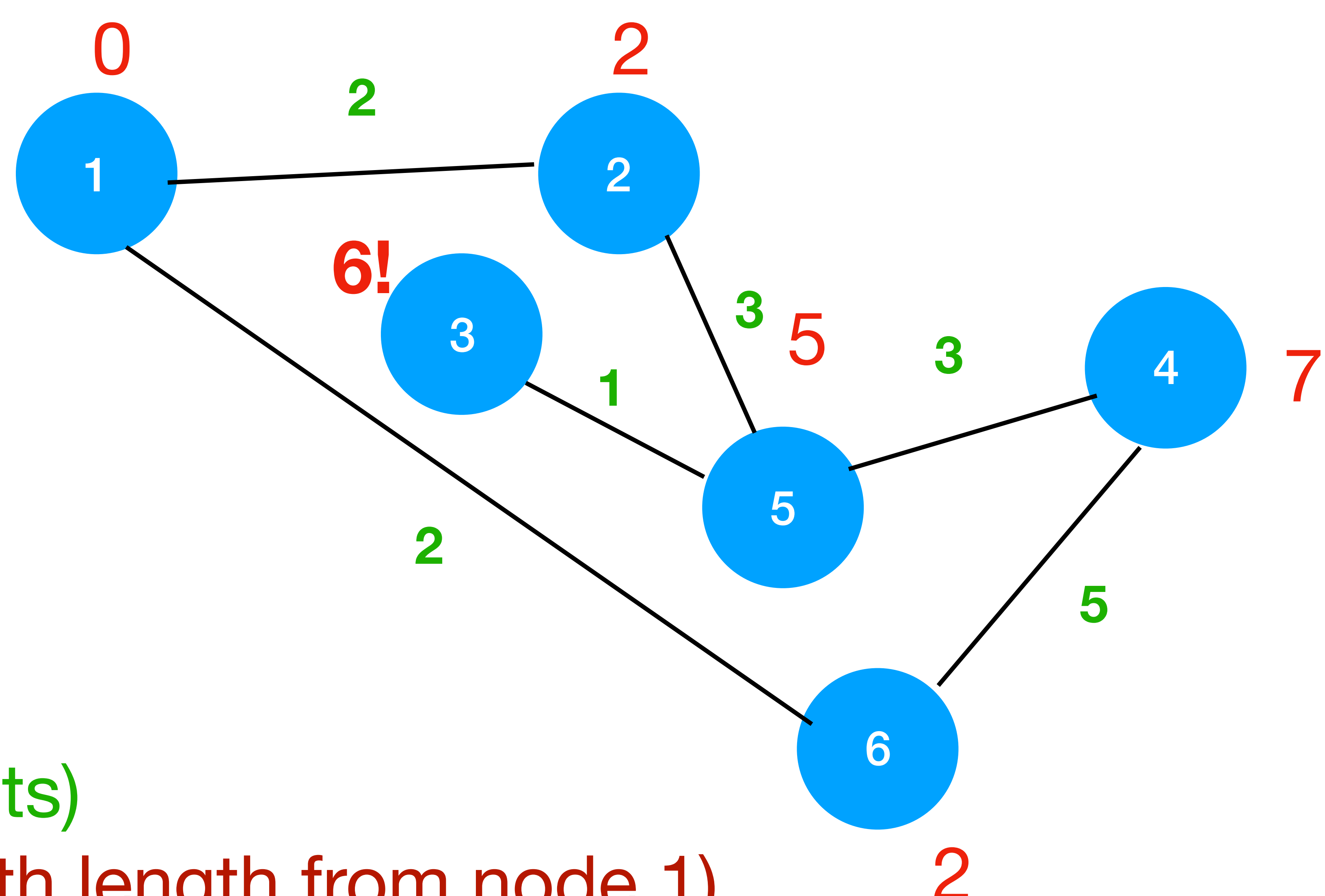
BFS doesn't visit nodes in order of shortest path length:



(edge weights)
(shortest path length from node 1)

Computing Shortest Paths in Weighted Graphs

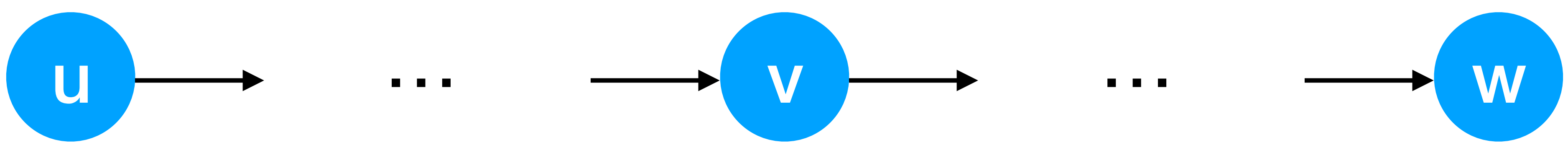
BFS doesn't visit nodes in order of shortest path length:



(edge weights)
(shortest path length from node 1)

Dijkstra’s Shortest Paths: Subpaths

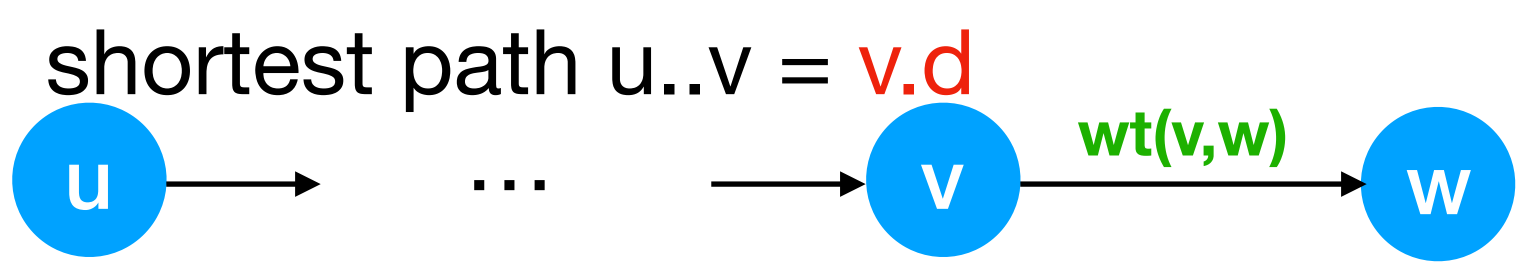
- Fact: **subpaths** of shortest paths are shortest paths



- Example: if the shortest path from u to w goes through v, then
  - the part of that path from u to v is the shortest path from u to v.
  - if there were some better path u..v, that would also be part of a better way to get from u to w.

Dijkstra's Shortest Paths: Subpaths

- Fact: **subpaths** of shortest paths are shortest paths
- Consequence: a **candidate** shortest path from start node **s** to some node **v**'s neighbor **w** is the shortest path from to **v** + the edge weight from **v** to **w**.



Dijkstra's Shortest Paths: Intuition

- Intuition: **explore nodes like BFS, but in order of path length instead of number of hops.**
- There are three kinds of nodes:
  - Settled - nodes for which we know the actual shortest path.
  - Frontier - nodes that have been visited but we don't necessarily have their actual shortest path
  - Unexplored - all other nodes.
- Each node n keeps track of `n.d`, the length of the shortest known known path from start.
- We may discover a shorter path to a frontier node than the one we've found already - if so, update `n.d`.

Dijkstra's Shortest Paths: Cartoon

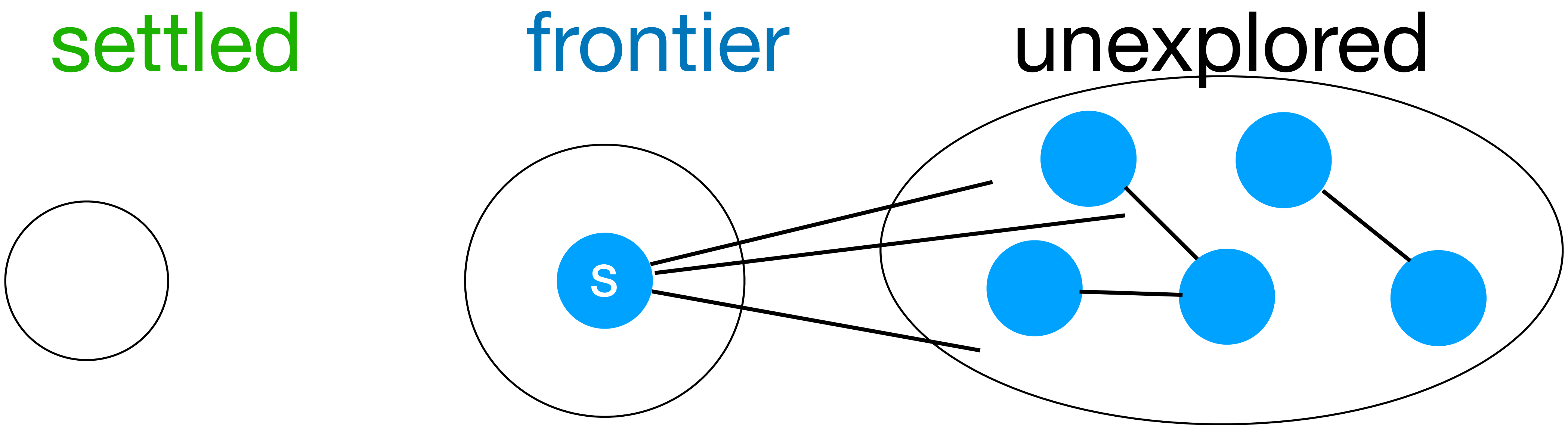
settled frontier unexplored

Before:

During:

After:

Dijkstra's Shortest Paths: Cartoon

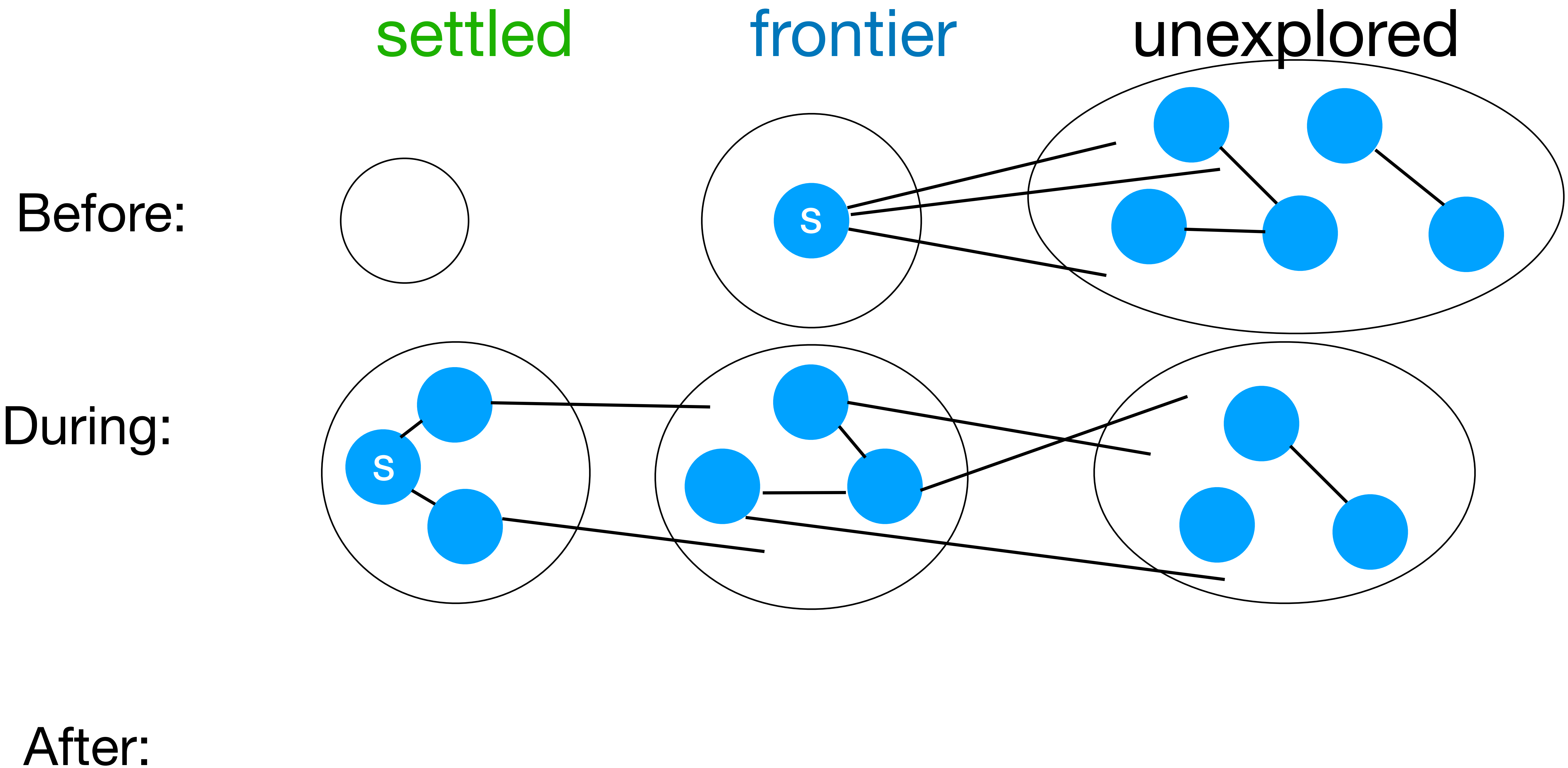


Before:

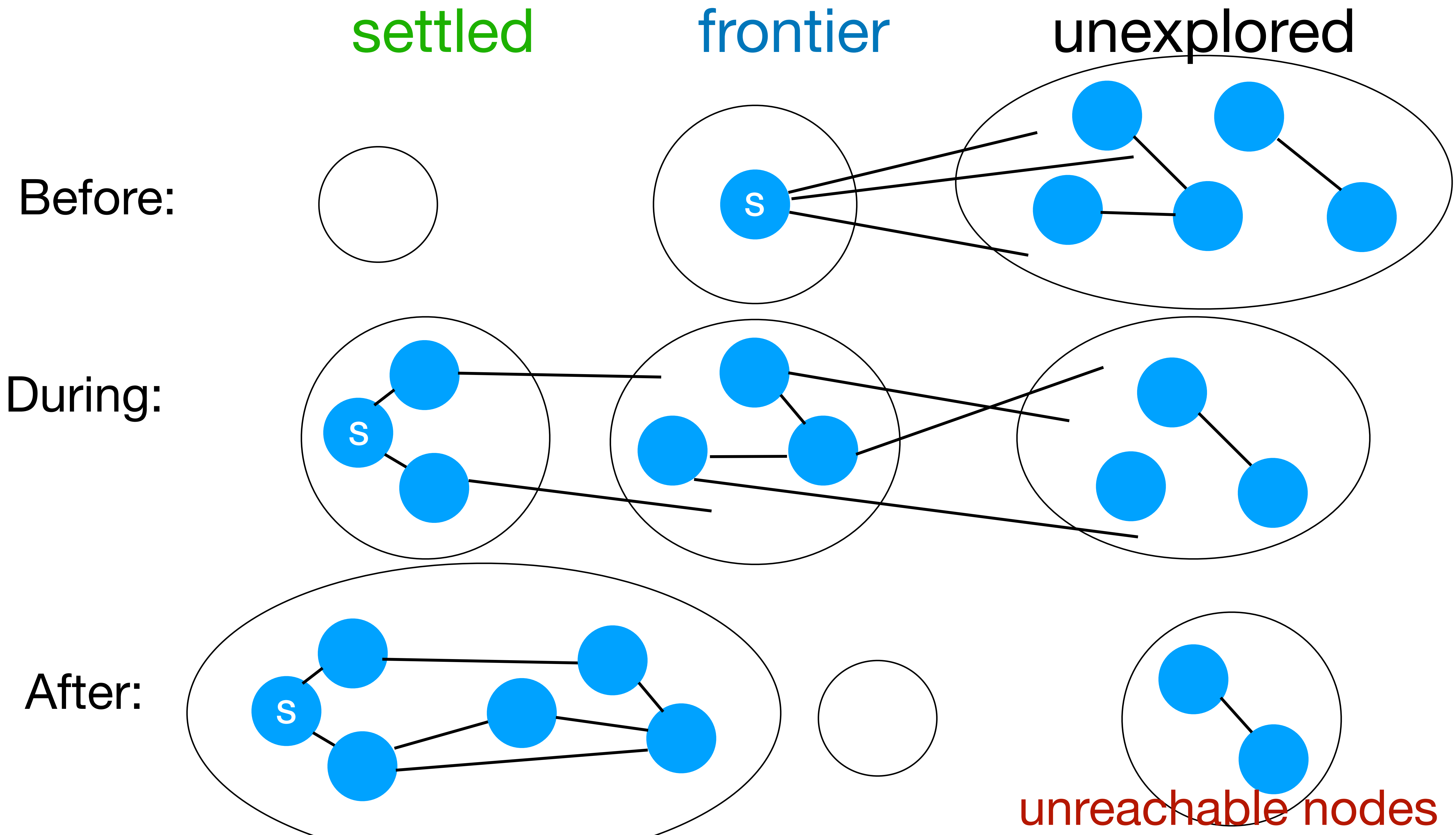
During:

After:

Dijkstra’s Shortest Paths: Cartoon



Dijkstra’s Shortest Paths: Cartoon

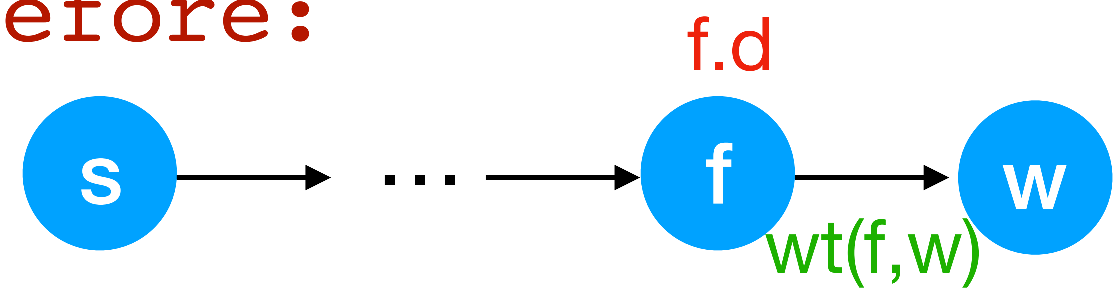


Dijkstra’s Shortest Paths: High-Level Algorithm

```
Initialize Settled to empty
Initialize Frontier to the start node
While the frontier isn’t empty:
  move the node f with smallest d from F to S
  For each neighbor w of f:
    if we’ve never seen w before:
      set its path length
      add it to frontier
  else if the path to w via f is shorter:
      update w’s shortest path length
```

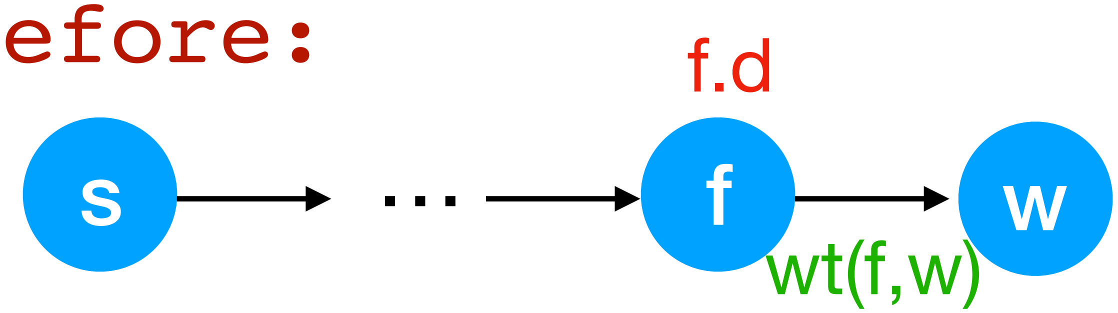
Dijkstra's Shortest Paths: High-Level Algorithm

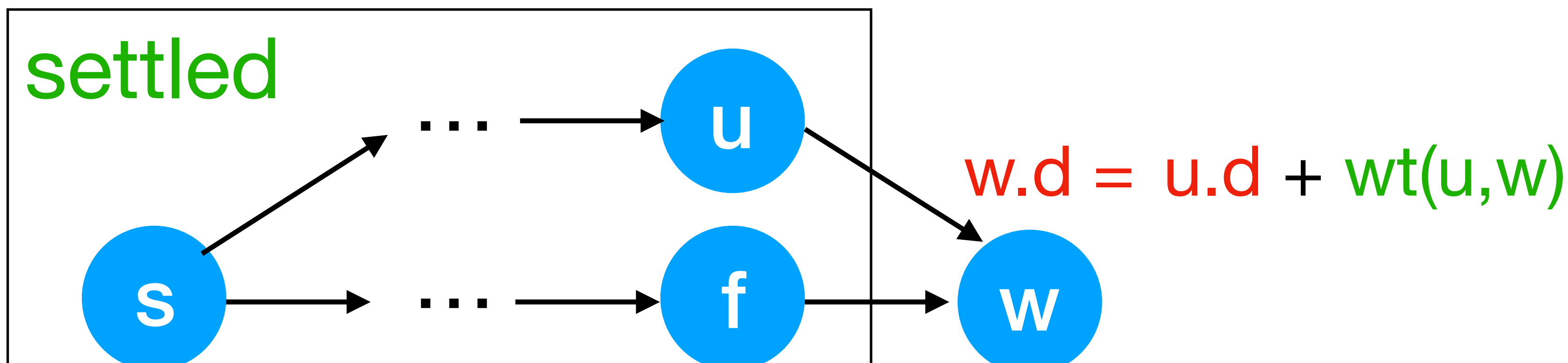
```
Initialize Settled to empty
Initialize Frontier to the start node
While the frontier isn't empty:
   move the node f with smallest d from F to S
   For each neighbor w of f:
      if we've never seen w before:
         set its path length
         add it to frontier
   else if the path to w via f is shorter:
         update w's shortest path length
```



Dijkstra's Shortest Paths: High-Level Algorithm

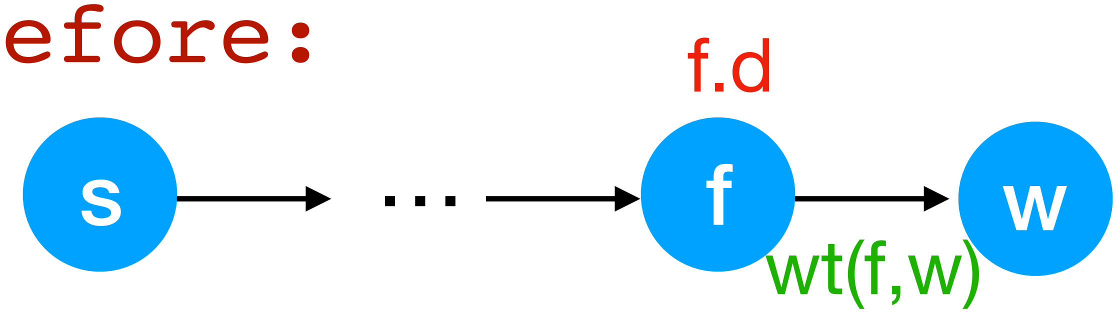
```
Initialize Settled to empty
Initialize Frontier to the start node
While the frontier isn't empty:
   move the node f with smallest d from F to S
   For each neighbor w of f:
      if we've never seen w before:
         set its path length
         add it to frontier
   else if the path to w via f is shorter:
         update w's shortest path length
```

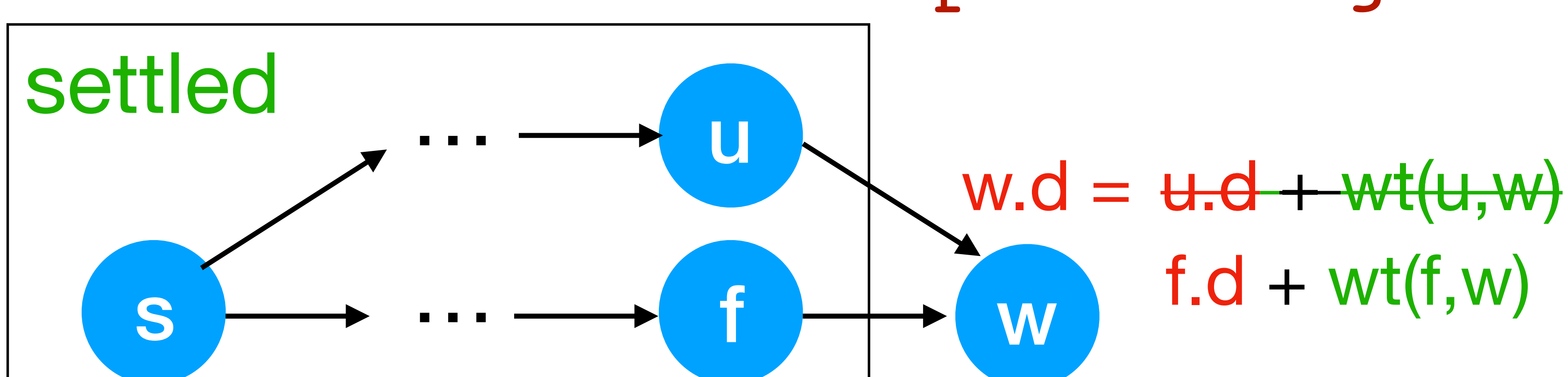




Dijkstra's Shortest Paths: High-Level Algorithm

```
Initialize Settled to empty
Initialize Frontier to the start node
While the frontier isn't empty:
   move the node f with smallest d from F to S
   For each neighbor w of f:
      if we've never seen w before:
         set its path length
         add it to frontier
   else if the path to w via f is shorter:
         update w's shortest path length
```





Dijkstra's Shortest Paths: Execution

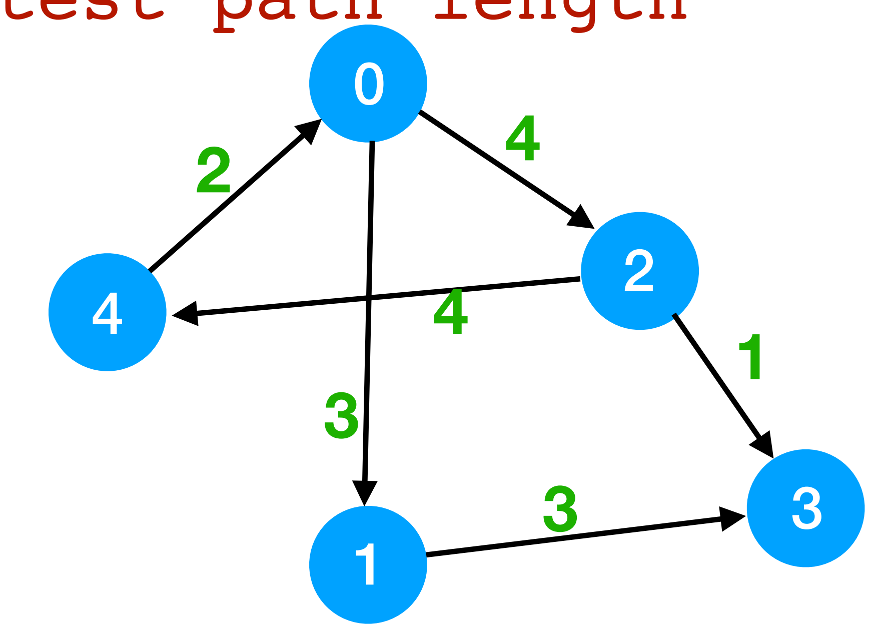
Best known distances:

Node	d
0	?
1	?
2	?
3	?
4	?

```
Initialize Settled to empty
Initialize Frontier to the start node
While the frontier isn't empty:
  move the node f with smallest d from F to S
  For each neighbor w of f:
    if we've never seen w before:
      set its path length to f.d + wt(f,w)
      add w to the frontier
    else if the path to w via f is shorter:
      update w's shortest path length
```

Settled set:

Frontier set:



shortest-paths(4)

Dijkstra's Shortest Paths: Execution

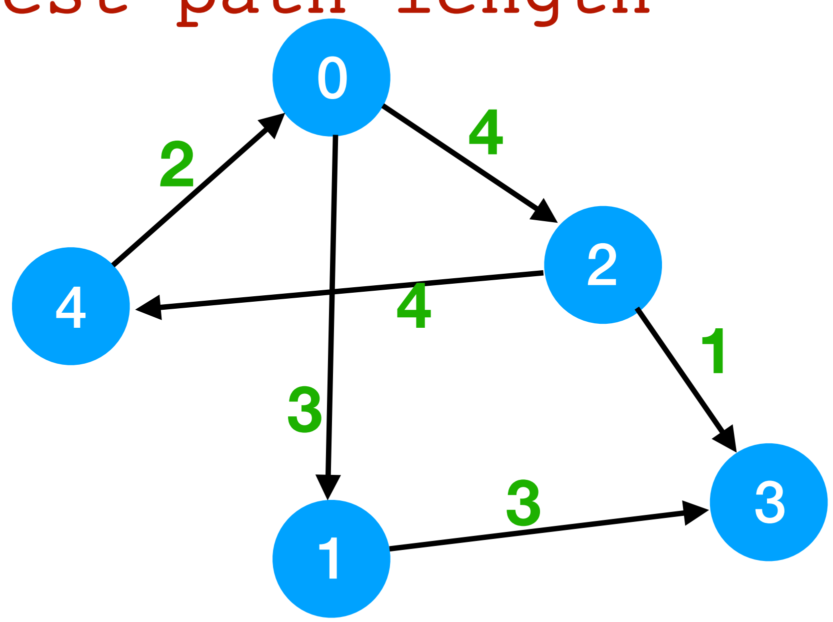
Best known distances:

Node	d
0	?
1	?
2	?
3	?
4	**0**

```
Initialize Settled to empty
Initialize Frontier to the start node
While the frontier isn't empty:
  move the node f with smallest d from F to S
  For each neighbor w of f:
    if we've never seen w before:
      set its path length to f.d + wt(f,w)
      add w to the frontier
    else if the path to w via f is shorter:
      update w's shortest path length
```

Settled set: {}

Frontier set: {4}



shortest-paths(4)

Dijkstra's Shortest Paths: Execution

Best known distances:

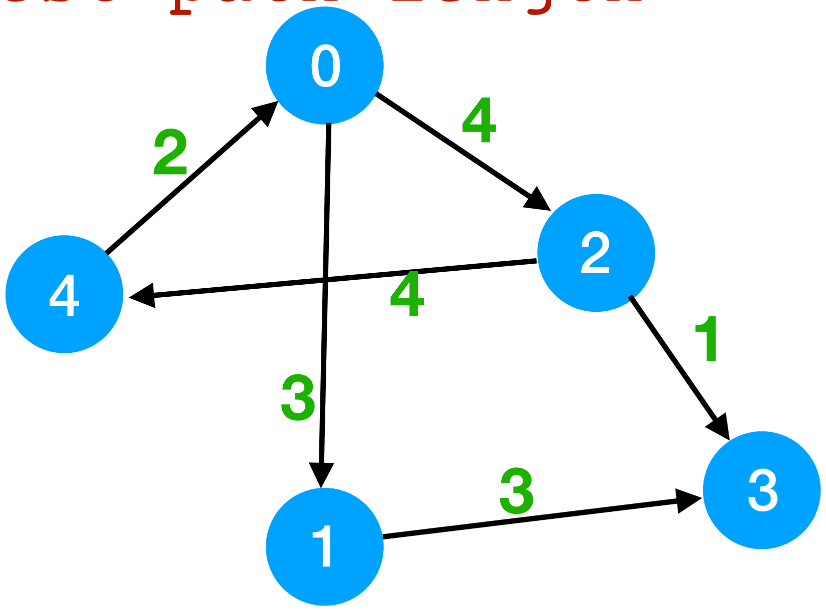
Node	d
0	?
1	?
2	?
3	?
4	0

```
Initialize Settled to empty
Initialize Frontier to the start node
While the frontier isn't empty:
  move the node f with smallest d from F to S
  For each neighbor w of f:
    if we've never seen w before:
      set its path length to f.d + wt(f,w)
      add w to the frontier
    else if the path to w via f is shorter:
      update w's shortest path length
```

f: 4

Settled set: {4}

Frontier set: {}



shortest-paths(4)

Dijkstra's Shortest Paths: Execution

Best known distances:

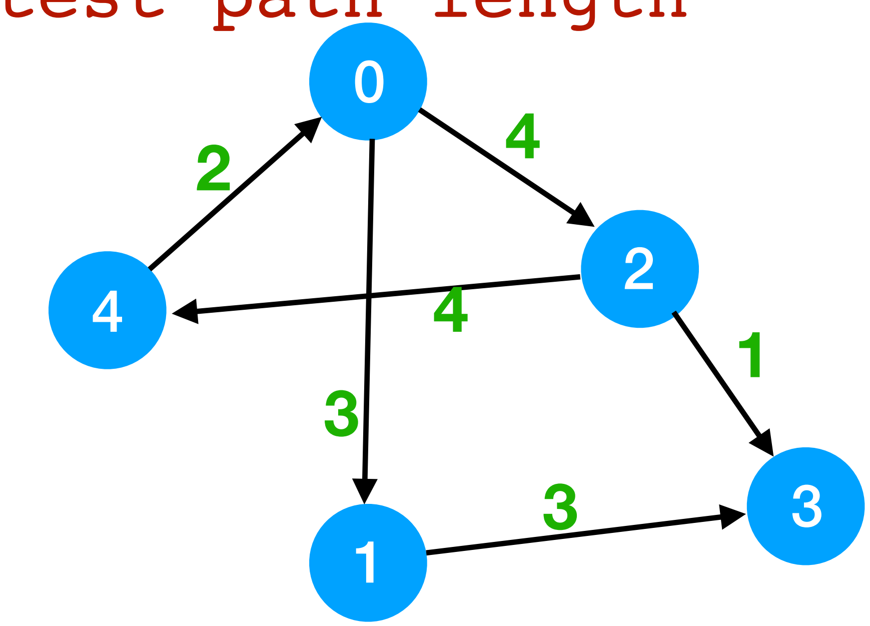
Node	d
0	2
1	?
2	?
3	?
4	0

```
Initialize Settled to empty
Initialize Frontier to the start node
While the frontier isn't empty:
  move the node f with smallest d from F to S
  For each neighbor w of f:
    if we've never seen w before:
      set its path length to f.d + wt(f,w)
      add w to the frontier
    else if the path to w via f is shorter:
      update w's shortest path length
```

f: 4
w: 0

Settled set: {4}

Frontier set: {0}



shortest-paths(4)

Dijkstra's Shortest Paths: Execution

Best known distances:

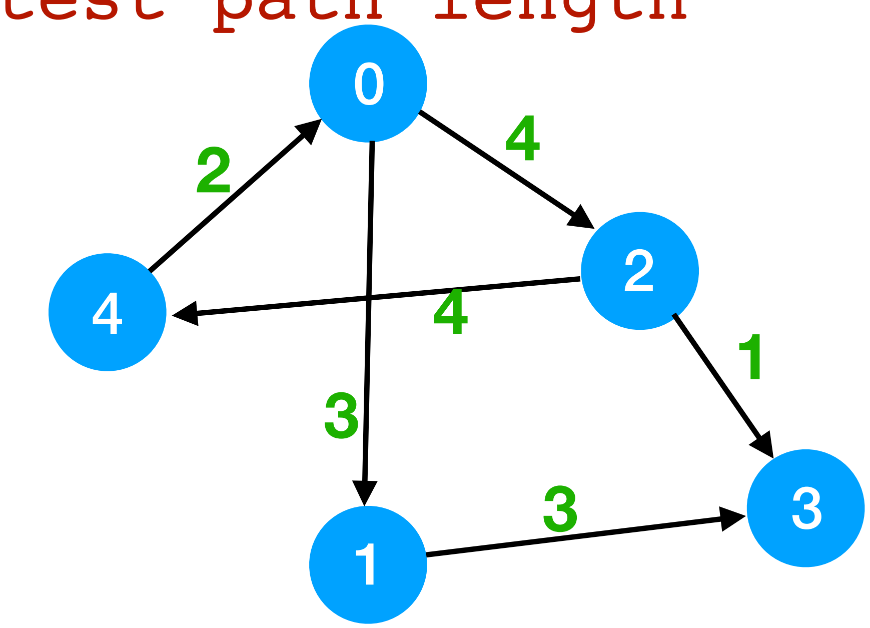
Node	d
0	2
1	?
2	?
3	?
4	0

```
Initialize Settled to empty
Initialize Frontier to the start node
While the frontier isn't empty:
   move the node f with smallest d from F to S
   For each neighbor w of f:
     if we've never seen w before:
       set its path length to f.d + wt(f,w)
       add w to the frontier
     else if the path to w via f is shorter:
       update w's shortest path length
```

f: 0

Settled set: {4, 0}

Frontier set: {}



shortest-paths(4)

Dijkstra's Shortest Paths: Execution

Best known distances:

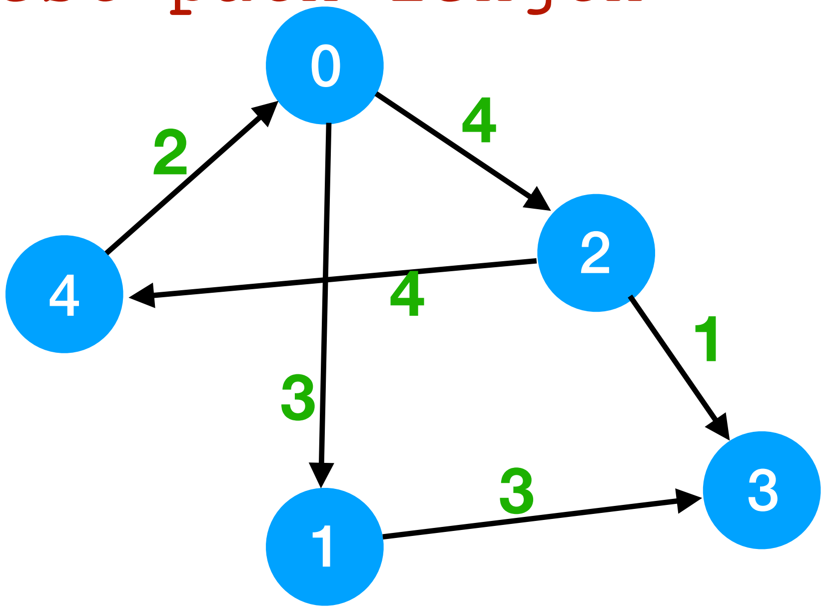
Node	d
0	2
1	**5**
2	?
3	?
4	0

```
Initialize Settled to empty
Initialize Frontier to the start node
While the frontier isn't empty:
   move the node f with smallest d from F to S
    For each neighbor w of f:
      if we've never seen w before:
        set its path length to f.d + wt(f,w)
        add w to the frontier
      else if the path to w via f is shorter:
        update w's shortest path length
```

f: 0
w: 1

Settled set: {4, 0}

Frontier set: {1}



shortest-paths(4)

Dijkstra's Shortest Paths: Execution

Best known distances:

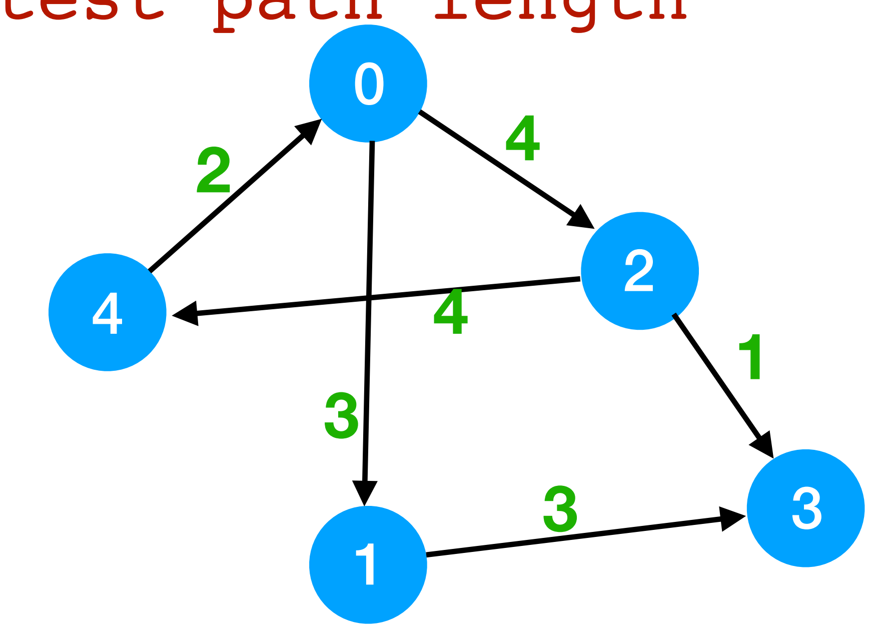
Node	d
0	2
1	5
2	**6**
3	?
4	0

```
Initialize Settled to empty
Initialize Frontier to the start node
While the frontier isn't empty:
  move the node f with smallest d from F to S
  For each neighbor w of f:
    if we've never seen w before:
      set its path length to f.d + wt(f,w)
      add w to the frontier
    else if the path to w via f is shorter:
      update w's shortest path length
```

f: 0
w: 2

Settled set: {4, 0}

Frontier set: {1, 2}



shortest-paths(4)

Dijkstra's Shortest Paths: Execution

Best known distances:

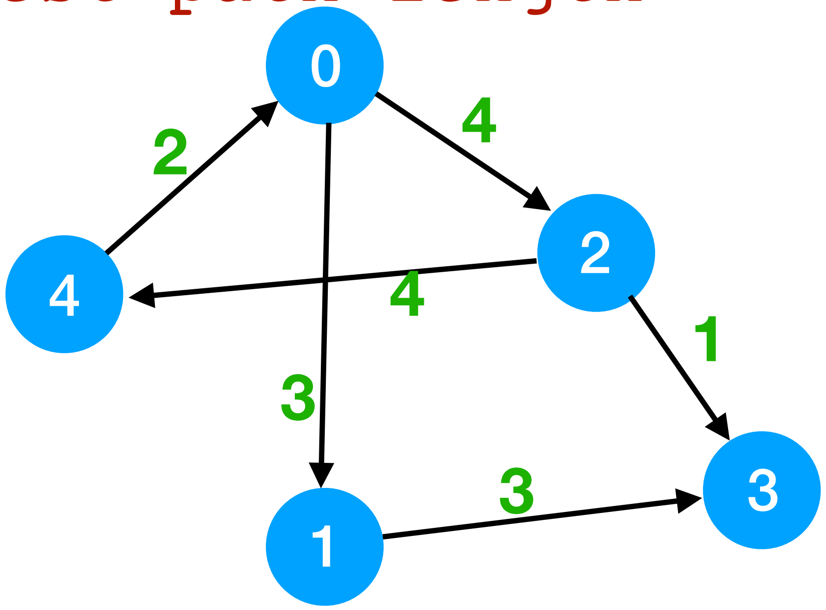
Node	d
0	2
1	5
2	6
3	8
4	0

```
Initialize Settled to empty
Initialize Frontier to the start node
While the frontier isn't empty:
  move the node f with smallest d from F to S
  For each neighbor w of f:
    if we've never seen w before:
      set its path length to f.d + wt(f,w)
      add w to the frontier
    else if the path to w via f is shorter:
      update w's shortest path length
```

f: 1

Settled set: {4, 0, 1}

Frontier set: {2}



shortest-paths(4)

Dijkstra's Shortest Paths: Execution

Best known distances:

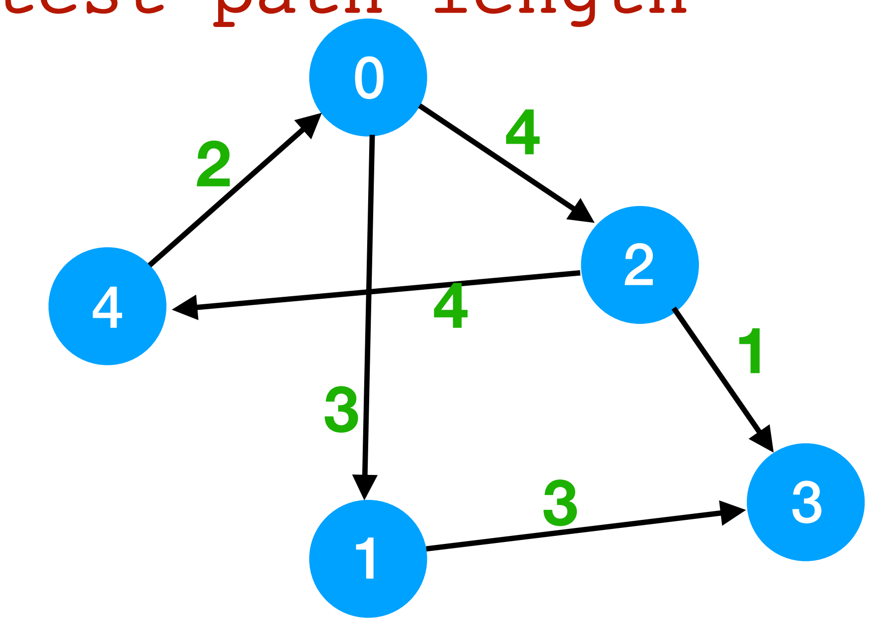
Node	d
0	2
1	5
2	6
3	8
4	0

```
Initialize Settled to empty
Initialize Frontier to the start node
While the frontier isn't empty:
   move the node f with smallest d from F to S
    For each neighbor w of f:
      if we've never seen w before:
        set its path length to f.d + wt(f,w)
        add w to the frontier
      else if the path to w via f is shorter:
        update w's shortest path length
```

f: 1
w: 3

Settled set: {4, 0, 1}

Frontier set: {2, 3}



shortest-paths(4)

Dijkstra's Shortest Paths: Execution

Best known distances:

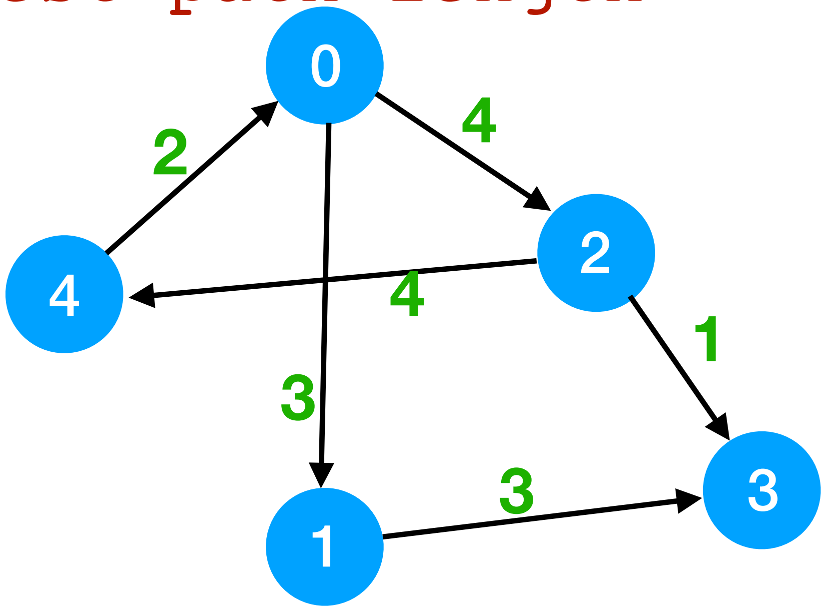
Node	d
0	2
1	5
2	6
3	8
4	0

```
Initialize Settled to empty
Initialize Frontier to the start node
While the frontier isn't empty:
  move the node f with smallest d from F to S
  For each neighbor w of f:
    if we've never seen w before:
      set its path length to f.d + wt(f,w)
      add w to the frontier
    else if the path to w via f is shorter:
      update w's shortest path length
```

f: 2

Settled set: {4, 0, 1, 2}

Frontier set: {3}



shortest-paths(4)

Dijkstra's Shortest Paths: Execution

Best known distances:

Node	d
0	2
1	5
2	6
3	**7**
4	0

```
Initialize Settled to empty
Initialize Frontier to the start node
While the frontier isn't empty:
  move the node f with smallest d from F to S
  For each neighbor w of f:
    if we've never seen w before:
      set its path length to f.d + wt(f,w)
      add w to the frontier
    else if the path to w via f is shorter:
      update w's shortest path length
```

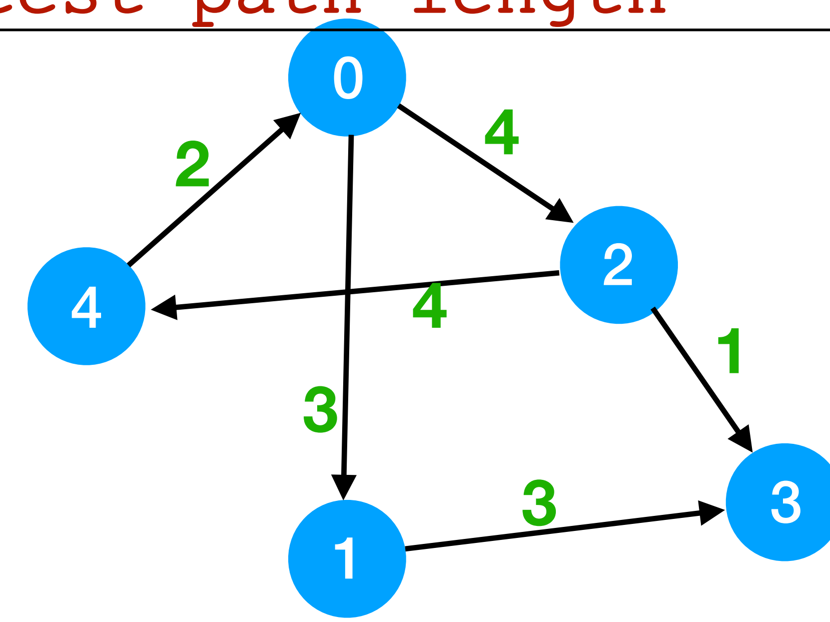
f: 2
w: 3

$$2.d + wt(2,3) < 3.d$$

$$7 < 8$$

Settled set: {4, 0, 1, 2}

Frontier set: {3}



`shortest-paths(4)`

Dijkstra's Shortest Paths: Execution

Best known distances:

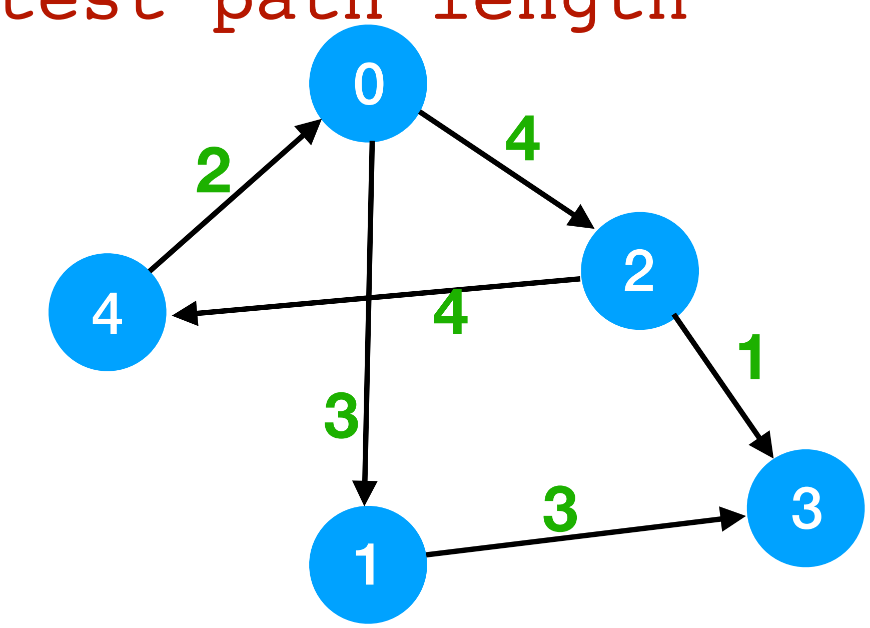
Node	d
0	2
1	5
2	6
3	7
4	0

```
Initialize Settled to empty
Initialize Frontier to the start node
While the frontier isn't empty:
  move the node f with smallest d from F to S
  For each neighbor w of f:
    if we've never seen w before:
      set its path length to f.d + wt(f,w)
      add w to the frontier
    else if the path to w via f is shorter:
      update w's shortest path length
```

f: 3

Settled set: {4, 0, 1, 2, 3}

Frontier set: {} Empty => done!



shortest-paths(4)

Dijkstra's Shortest Paths: Pseudocode

```
S = { }; F = {v};  v.d = 0;          Initialize Settled to empty
while  (F ≠ {})  {                   Initialize Frontier to the start node
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
          w.d =  f.d + weight(f, w);
          add w to F;
      } else if (f.d+weight(f,w) < w.d) {
          w.d = f.d+weight(f,w);
      }
   }
}
```

Dijkstra's Shortest Paths: Pseudocode

```
S = { }; F = {v};  v.d = 0;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         add w to F;
      } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
      }
   }
}
```

```
Initialize Settled to empty
Initialize Frontier to the start node

While the frontier isn't empty:
   move node f with smallest d
    from F to S
```

Dijkstra's Shortest Paths: Pseudocode

```
S = { }; F = {v};  v.d = 0;            Initialize Settled to empty
while  (F ≠ {})  {                     Initialize Frontier to the start node
   f = node in F with min d value;     While the frontier isn't empty:
   Remove f from F, add it to S;         move node f with smallest d
                                           from F to S
   for each neighbor w of f {          For each neighbor w of f:
     if (w not in S or F) {              if we've never seen w before:
         w.d =  f.d + weight(f, w);        set its path length
         add w to F;                       add it to frontier
     } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
     }
   }
}
```

Dijkstra's Shortest Paths: Pseudocode

$S = \{\ \}$; $F = \{v\}$; $v.d = 0$;
while $(F \neq \{\})$ {
  f = node in F with min d value;
  Remove f from F, add it to S;
  for each neighbor w of f {
    if (w not in S or F) {
      w.d = f.d + weight(f, w);
      add w to F;
    } **else if** (f.d+weight(f,w) < w.d) {
      w.d = f.d+weight(f,w);
    }
  }
}

```
Initialize Settled to empty
Initialize Frontier to the start node
While the frontier isn't empty:
   move node f with smallest d
    from F to S
For each neighbor w of f:
  if we've never seen w before:
    set its path length
    add it to frontier
  else if path to w via f is shorter:
      update w's shortest path length
```

What if we want to know the shortest path?

```
S = { }; F = {v};  v.d = 0;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         add w to F;
      } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
      }
   }
}
```

- At termination: for each reachable node n, n.d stores the **length** of the shortest path from v to n.
- We didn't keep track of **how** to get from v to n!

What if we want to know the shortest path?

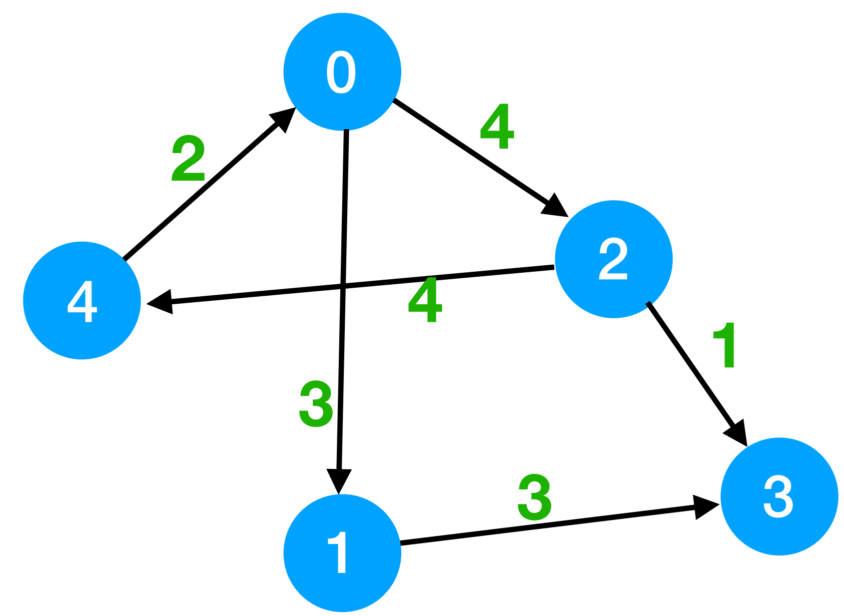
```
S = { }; F = {v};  v.d = 0; v.bp = null;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         w.bp = f;
         add w to F;
      } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
         w.bp = f
      }
   }
}
```

Each node could store the full path, but that would be expensive to keep updated.

Strategy: maintain a **backpointer** at each node pointing to the previous node in the shortest path.

What if we want to know the shortest path? Example

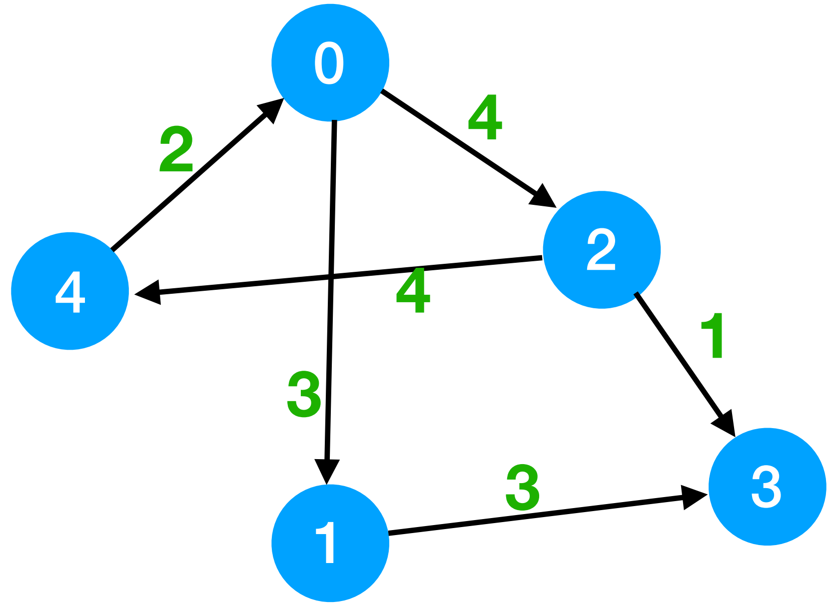
```
S = { }; F = {v};  v.d = 0; v.bp = null;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
          w.d =  f.d + weight(f, w);
          w.bp = f;
          add w to F;
     } else if (f.d+weight(f,w) < w.d) {
          w.d = f.d+weight(f,w);
          w.bp = f
     }
   }
}
```



shortest-paths(4)

Strategy: maintain a **backpointer** at each node pointing to the previous node in the shortest path.

```
S = { }; F = {v};  v.d = 0; v.bp = null;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         w.bp = f;
         add w to F;
      } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
         w.bp = f
      }
   }
}
```



shortest-paths(4)

S:

F:

Node	d	bp
0		
1		
2		
3		
4		

Questions?


The next slide very
important.

Implementing Dijkstra Efficiently (A4)

```
S = { }; F = {v};  v.d = 0; v.bp = null;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
  for each neighbor w of f {
     if (w not in S or F) {
        w.d =  f.d + weight(f, w);
        w.bp = f;
        add w to F;
    } else if (f.d+weight(f,w) < w.d) {
        w.d = f.d+weight(f,w);
        w.bp = f
    }
  }
}
```

1. Store Frontier in a min-heap priority queue with d-values as priorities.
2. To efficiently iterate over neighbors, use an adjacency list graph representation.
3. Could store w.d and w.bp in Node class; in A4, we use a HashMap<Node,PathData>
4. No need to explicitly store Settled or Unexplored sets: a node is in S or F iff it is in the map.

Implementing Dijkstra Efficiently (A4)

```
S = { }; F = {v};  v.d = 0; v.bp = null;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         w.bp = f;
         add w to F;
      } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
         w.bp = f
      }
   }
}
```

1. **Store Frontier in a min-heap priority queue with d-values as priorities.**
2. To efficiently iterate over neighbors, use an adjacency list graph representation.
3. Could store w.d and w.bp in Node class; in A4, we use a HashMap<Node,PathData>
4. No need to explicitly store Settled or Unexplored sets:
   a node is in S or F iff it is in the map.

Implementing Dijkstra Efficiently (A4)

```
S = { }; F = {v};  v.d = 0; v.bp = null;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
          w.d =  f.d + weight(f, w);
          w.bp = f;
          add w to F;
      } else if (f.d+weight(f,w) < w.d) {
          w.d = f.d+weight(f,w);
          w.bp = f
      }
   }
}
```

1. Store Frontier in a min-heap priority queue with d-values as priorities.
2. **To efficiently iterate over neighbors, use an adjacency list graph representation.**
3. Could store w.d and w.bp in Node class; in A4, we use a HashMap<Node,PathData>
4. No need to explicitly store Settled or Unexplored sets:
   a node is in S or F iff it is in the map.

Implementing Dijkstra Efficiently (A4)

```
S = { }; F = {v}; v.d = 0; v.bp = null;
while (F ≠ {}) {
  f = node in F with min d value;
  Remove f from F, add it to S;
  for each neighbor w of f {
    if (w not in S or F) {
      w.d = f.d + weight(f, w);
      w.bp = f;
      add w to F;
    } else if (f.d+weight(f,w) < w.d) {
      w.d = f.d+weight(f,w);
      w.bp = f
    }
  }
}
```

1. Store Frontier in a min-heap priority queue with d-values as priorities.
2. To efficiently iterate over neighbors, use an adjacency list graph representation.
3. **Could store w.d and w.bp in Node class; in A4, we use a HashMap<Node,PathData>**
4. No need to explicitly store Settled or Unexplored sets: a node is in S or F iff it is in the map.

Implementing Dijkstra Efficiently (A4)

```
S = { }; F = {v};  v.d = 0; v.bp = null;
while  (F ≠ {})  {
  f = node in F with min d value;
  Remove f from F, add it to S;
  for each neighbor w of f {
     if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         w.bp = f;
         add w to F;
     } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
         w.bp = f
     }
  }
}
```

1. Store Frontier in a min-heap priority queue with d-values as priorities.
2. To efficiently iterate over neighbors, use an adjacency list graph representation.
3. Could store w.d and w.bp in Node class; in A4, we use a HashMap<Node,PathData>
4. **No need to explicitly store Settled or Unexplored sets:**
   a node is in S or F iff it is in the map.

Implementing Dijkstra Efficiently (A4)

```
S = { }; F = {v};  v.d = 0; v.bp = null;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         w.bp = f;
         add w to F;
     } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
         w.bp = f
     }
   }
}
```

4. No need to explicitly store Settled or Unexplored sets:
w is in S or F <=> it is in the map.

The only time we need to check membership in S is **here**.

Implementing Dijkstra Efficiently (A4)

```
S = { }; F = {v};  v.d = 0; v.bp = null;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         w.bp = f;
         add w to F;
      } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
         w.bp = f
      }
   }
}
```

4. No need to explicitly store Settled or Unexplored sets:
w is in S or F <=> it is in the map.

The only time we need to check membership in S is **here**.

If w is not in S or F,
it must be in Unexplored.

Implementing Dijkstra Efficiently (A4)

```
S = { }; F = {v};  v.d = 0; v.bp = null;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         w.bp = f;
         add w to F;
      } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
         w.bp = f
      }
   }
}
```

4. No need to explicitly store Settled or Unexplored sets:
w is in S or F <=> it is in the map.

The only time we need to check membership in S is **here**.

If w is not in S or F,
it must be in Unexplored.

therefore,
we haven't found a path to it.

Implementing Dijkstra Efficiently (A4)

```
S = { }; F = {v};  v.d = 0; v.bp = null;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         w.bp = f;
         add w to F;
      } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
         w.bp = f
      }
   }
}
```

4. No need to explicitly store Settled or Unexplored sets:
w is in S or F <=> it is in the map.

The only time we need to check membership in S is **here**.

If w is not in S or F,
it must be in Unexplored.

therefore,
we haven't found a path to it.

therefore,
it has no d or bp yet.

Implementing Dijkstra Efficiently (A4)

```
S = { }; F = {v};  v.d = 0; v.bp = null;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         w.bp = f;
         add w to F;
     } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
         w.bp = f
     }
   }
}
```

4. No need to explicitly store Settled or Unexplored sets:
w is in S or F <=> it is in the map.

The only time we need to check membership in S is **here**.

If w is not in S or F,
it must be in Unexplored.

therefore,
we haven't found a path to it.

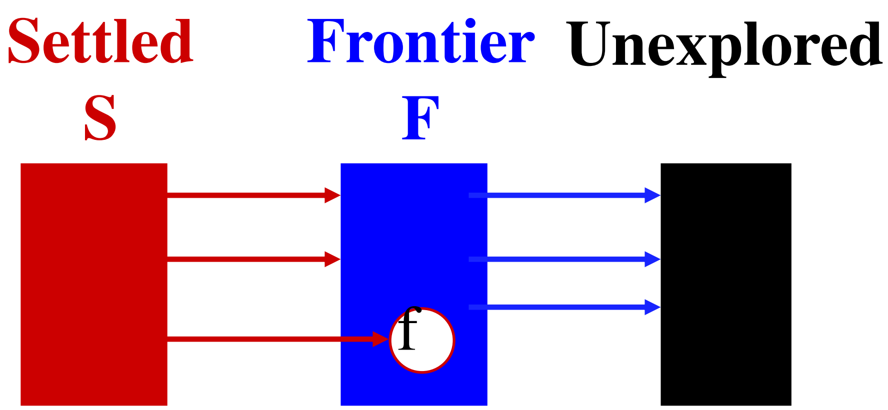
therefore,
it has no d or bp yet.

therefore,
it isn't in the map!

Proof of Correctness

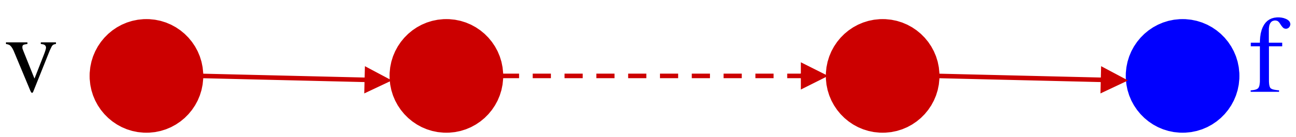
- Dijkstra’s algorithm is **greedy**: it makes a sequence of *locally* optimal moves, which results in the *globally* optimal solution.
    - Most algorithms don’t work like this - need to prove that it results in the global optimum.
- Specifically: It is not obvious that there cannot still be a shorter path to the Frontier node with smallest d-value.

Proof of Correctness: Invariant



The while loop in Dijkstra's algorithm maintains a 3-part invariant:

1. For a Settled node s, a shortest path from v to s contains only settled nodes and s.d is length of shortest v -> s path.



2. For a Frontier node f, at least one v -> f path contains only settled nodes (except perhaps for f) and f.d is the length of the shortest such path

3. All edges leaving S go to F (or: no edges from S to Unexplored)

Proof of Correctness: Theorem

```
S = { }; F = {v};  v.d = 0;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         add w to F;
     } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
     }
   }
}
```

Theorem: For a node f in the Frontier with minimum d value (over all nodes in the Frontier), f.d is the shortest-path distance from v to f.

Proof: Show that any other path from v to if has length >= f.d

Case 1: if v is in F, then S is empty and v.d = 0, which is trivially the shortest distance from v to v.

Proof of Correctness: Theorem

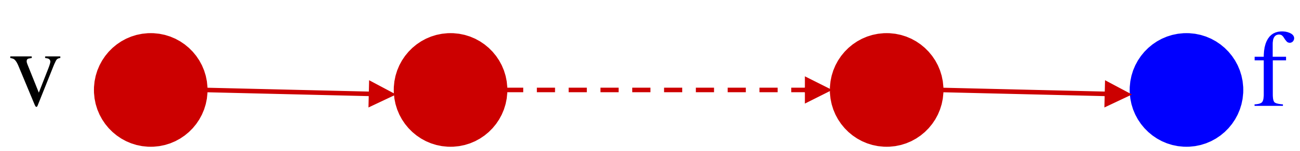
```
S = { }; F = {v};  v.d = 0;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         add w to F;
      } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
      }
   }
}
```

Theorem: For a node f in the Frontier with minimum d value (over all nodes in the Frontier), f.d is the shortest-path distance from v to f.

Proof: Show that any other path from v to if has length >= f.d

Case 2: v is in S. Part 2 of the invariant says:

- f.d is the length of the shortest path from v to f containing all settled nodes except f, and f.d is the length of such a path.



Proof of Correctness: Theorem

```
S = { }; F = {v};  v.d = 0;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         add w to F;
      } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
      }
   }
}
```

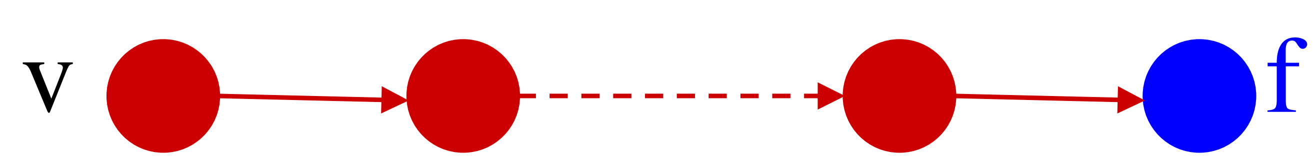
Theorem: For a node f in the Frontier with minimum d value (over all nodes in the Frontier), f.d is the shortest-path distance from v to f.

Proof: Show that any other path from v to if has length >= f.d

Case 2: v is in S. Part 2 of the invariant says:

- f.d is the length of the shortest path from v to f containing all settled nodes except f, and f.d is the length of such a path.

Any other v-f path must either be longer or go through another frontier node g then arrive at f:



Proof of Correctness: Theorem

```
S = { }; F = {v};  v.d = 0;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         add w to F;
      } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
      }
   }
}
```

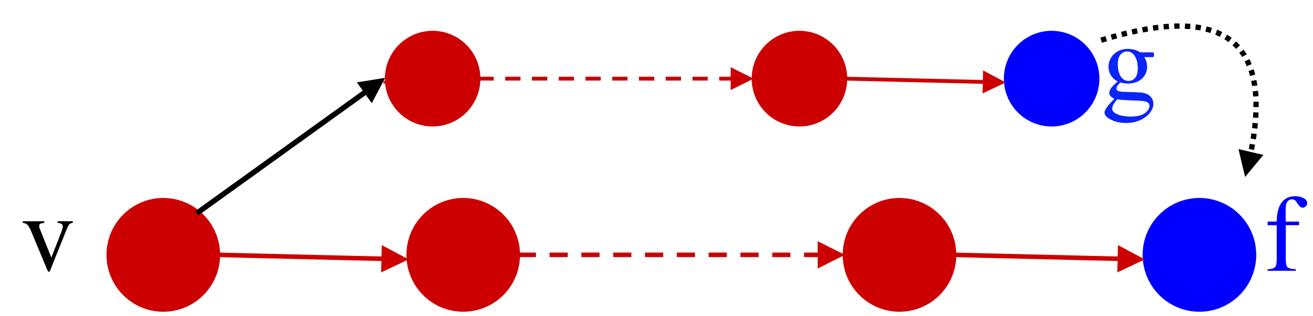
Theorem: For a node f in the Frontier with minimum d value (over all nodes in the Frontier), f.d is the shortest-path distance from v to f.

Proof: Show that any other path from v to if has length >= f.d

Case 2: v is in S. Part 2 of the invariant says:

- f.d is the length of the shortest path from v to f containing all settled nodes except f, and f.d is the length of such a path.

Any other v-f path must either be longer or go through another frontier node g then arrive at f:



Proof of Correctness: Theorem

```
S = { }; F = {v};  v.d = 0;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         add w to F;
      } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
      }
   }
}
```

Theorem: For a node f in the Frontier with minimum d value (over all nodes in the Frontier), f.d is the shortest-path distance from v to f.

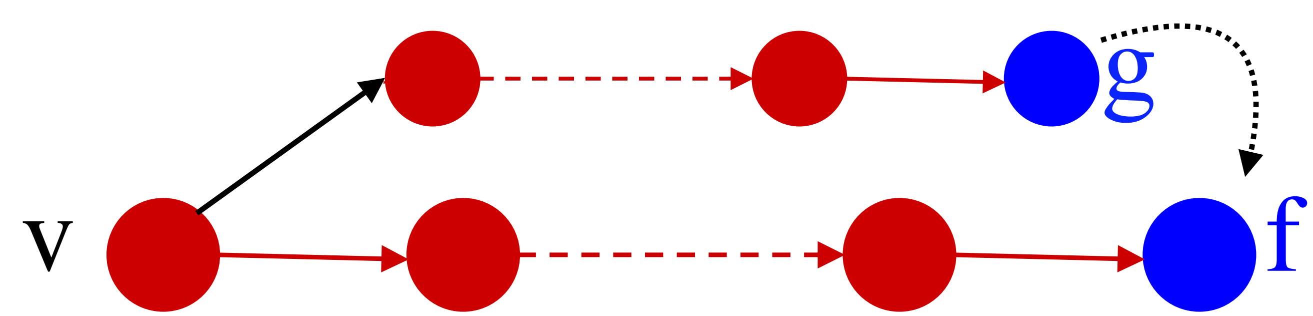
Proof: Show that any other path from v to if has length >= f.d

Case 2: v is in S. Part 2 of the invariant says:

- f.d is the length of the shortest path from v to f containing all settled nodes except f, and f.d is the length of such a path.

Any other v-f path must either be longer or go through another frontier node g then arrive at f:

d.f <= d.g,
so that path cannot be shorter



Proof of Correctness: Invariant Maintenance

```
S = { }; F = {v};  v.d = 0;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         add w to F;
      } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
      }
   }
}
```

1. For a Settled node s, a shortest path from v to s contains only settled nodes and s.d is length of shortest v -> s path.
2. For a Frontier node f, at least one v -> f path contains only settled nodes (except perhaps for f) and f.d is the length of the shortest such path
3. All edges leaving S go to F (or: no edges from S to Unexplored)

Proof of Correctness: Invariant Maintenance

```
S = { }; F = {v};  v.d = 0;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         add w to F;
      } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
      }
   }
}
```

1. For a Settled node s, a shortest path from v to s contains only settled nodes and s.d is length of shortest v -> s path.
2. For a Frontier node f, at least one v -> f path contains only settled nodes (except perhaps for f) and f.d is the length of the shortest such path
3. All edges leaving S go to F (or: no edges from S to Unexplored)

At initialization:

1. S is empty; trivially true.
2. v.d = 0, which is the shortest path.
3. S is empty, so no edges leave it.

Proof of Correctness: Invariant Maintenance

```
S = { }; F = {v};  v.d = 0;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         add w to F;
      } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);
      }
   }
}
```

1. For a Settled node s, a shortest path from v to s contains only settled nodes and s.d is length of shortest v -> s path.
2. For a Frontier node f, at least one v -> f path contains only settled nodes (except perhaps for f) and f.d is the length of the shortest such path
3. All edges leaving S go to F (or: no edges from S to Unexplored)

At each iteration:

1. Theorem says f.d is the shortest path, so it can safely move to S
2. Updating w.d maintains Part 2 of the invariant.
3. Each neighbor is either already in F or gets moved there.