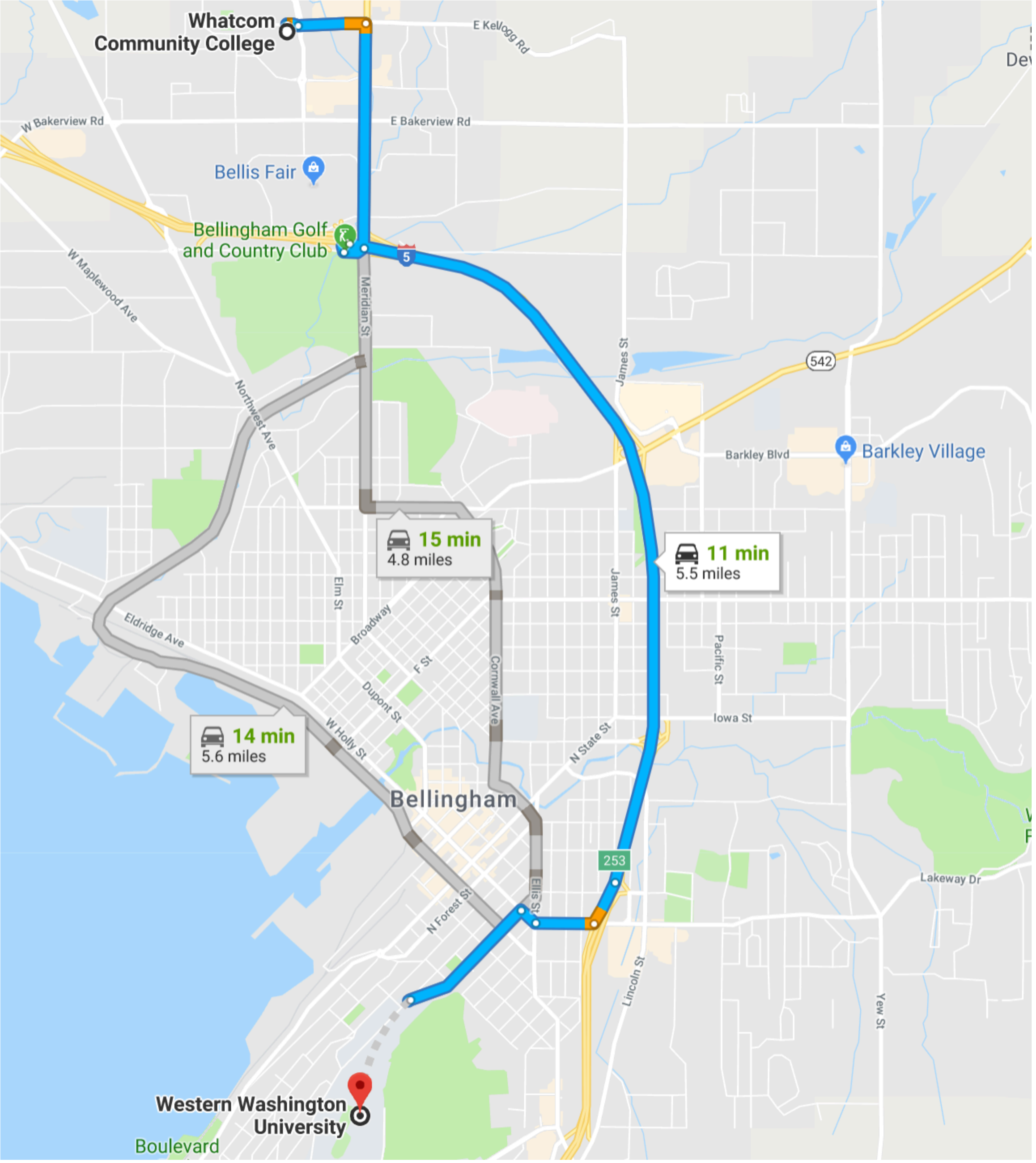


CSCI 241

Lecture 19

Dijkstra’s Single-Source Shortest Paths Algorithm

Announcements

- A4 out today
- I’ll post full slides for Dijkstra even though we won’t get through all of them today.

Goals

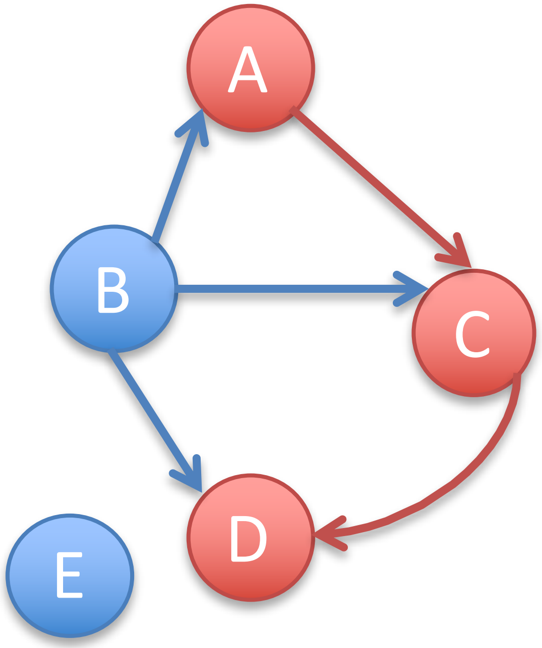
- Know how to determine whether a graph is connected
- Know the definition of connected components.
- Understand and be able to implement graph traversal/search algorithms:
  - Depth-first search
  - Breadth-first search
- Know what a weighted graph is.
- Understand the intuition behind Dijkstra’s shortest paths algorithm.
- Be able to execute Dijkstra’s algorithm manually on a graph.

Graph Terminology: Adjacency, Degree

- Two vertices are adjacent if they are connected by an edge
- Nodes u and v are called the source and sink of the **directed** edge (u, v)
- Nodes u and v are endpoints of an edge (u, v) (directed or undirected)
- The outdegree of a vertex u in a **directed** graph is the number of edges for which u is the source
- The indegree of a vertex v in a **directed** graph is the number of edges for which v is the sink
- The degree of a vertex u in an **undirected** graph is the number of edges of which u is an endpoint

Graph Terminology

- A path is a sequence of vertices where each consecutive pair are adjacent.
  - In a directed graph, paths must follow the direction of the edges.
- A cycle is a path that ends where it started, e.g.: x, y, z, x
- A graph is acyclic if it has no cycles.

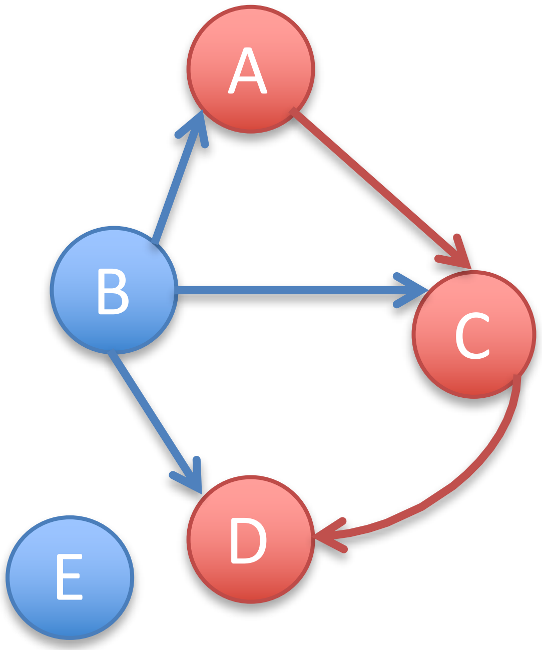


Path A,C,D

Acyclic

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.

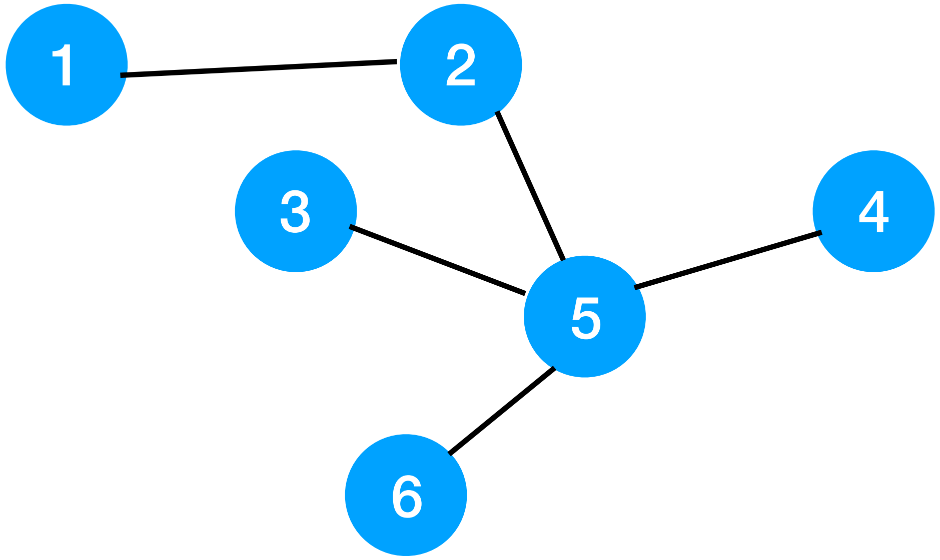


Not strongly connected

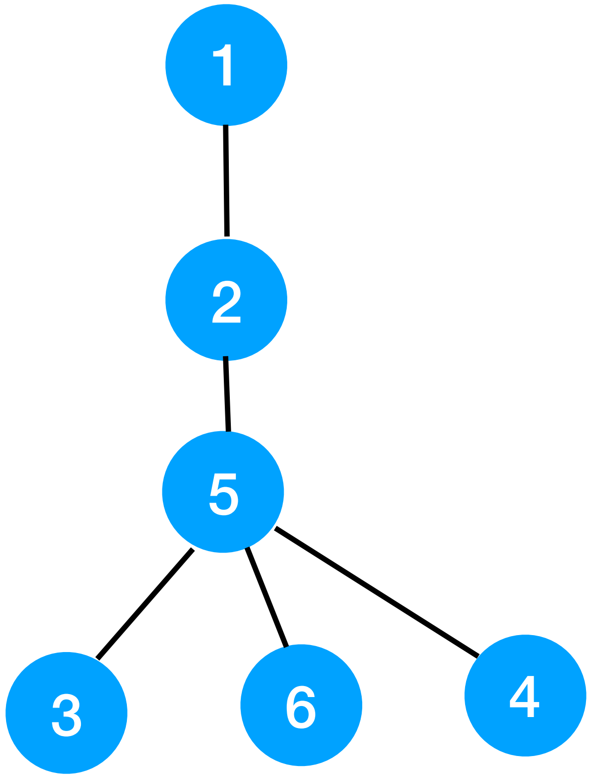
Not weakly connected

Trees vs Graphs

- Trees are graphs!
- A tree is an undirected graph with exactly 1 path between all pairs of nodes.
- Implication: no cycles!



V = {1,2,3,4,5,6}
E = {(1,2), (2,5), (3,5)
(4,5), (5,6)}



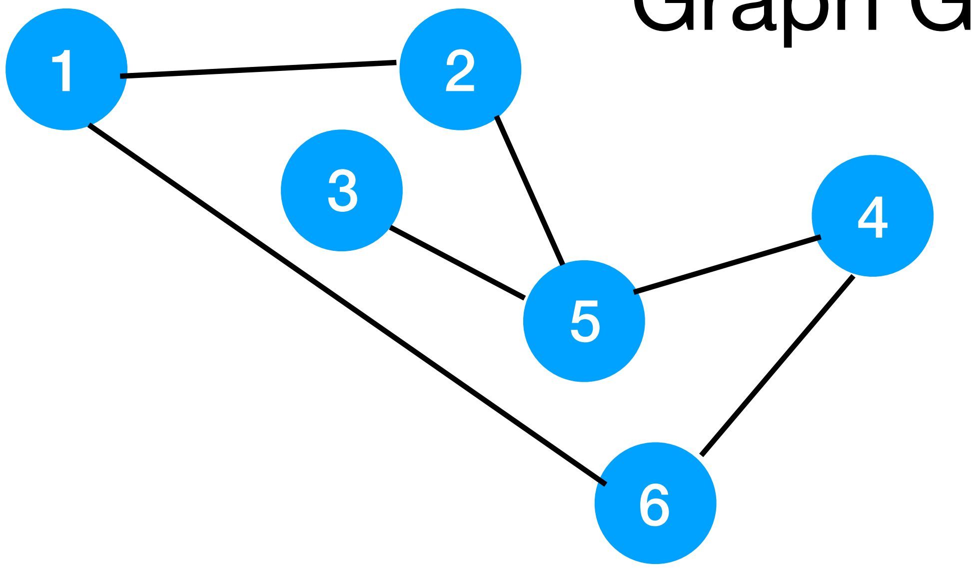
Many problems are easy in trees and harder in graphs.

Graph terminology: Lightning Round!

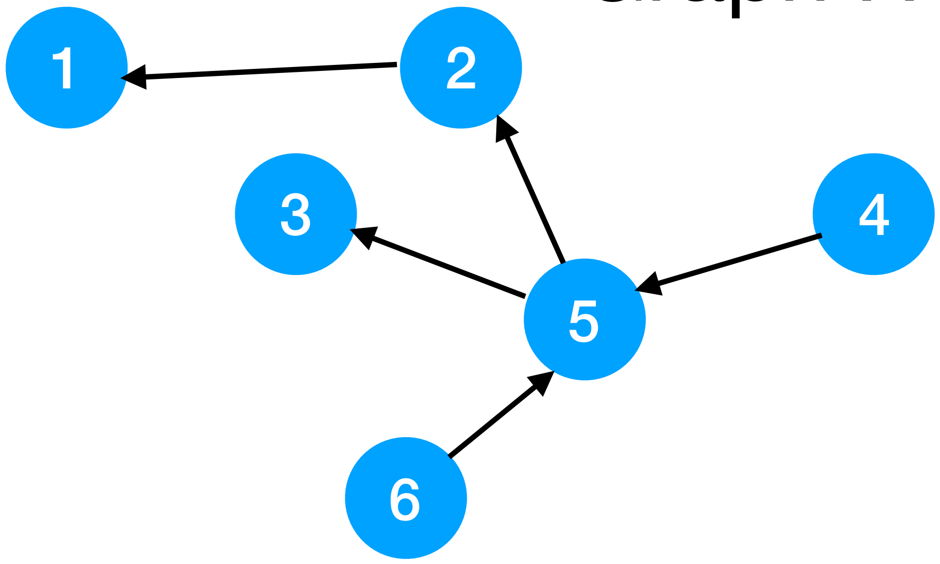
A: No B: Yes

- Is graph G acyclic?
- Is there a path from 3 to 5 in graph H?
- Is graph H directed?
- Is (1,2) an edge in H?

Graph G



Graph H



Graph terminology: Lightning Round!

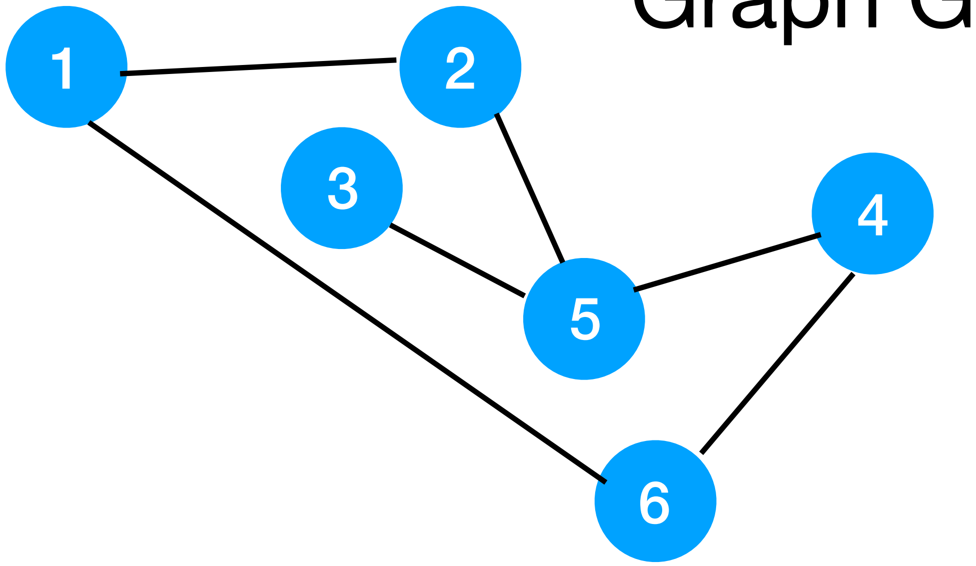
- What's the degree of node 5 in graph G?
  A: 1 B: 2 C: 3 D: 4

- What is |V| in graph G?
  A: 3 B: 4 C: 5 D: 6

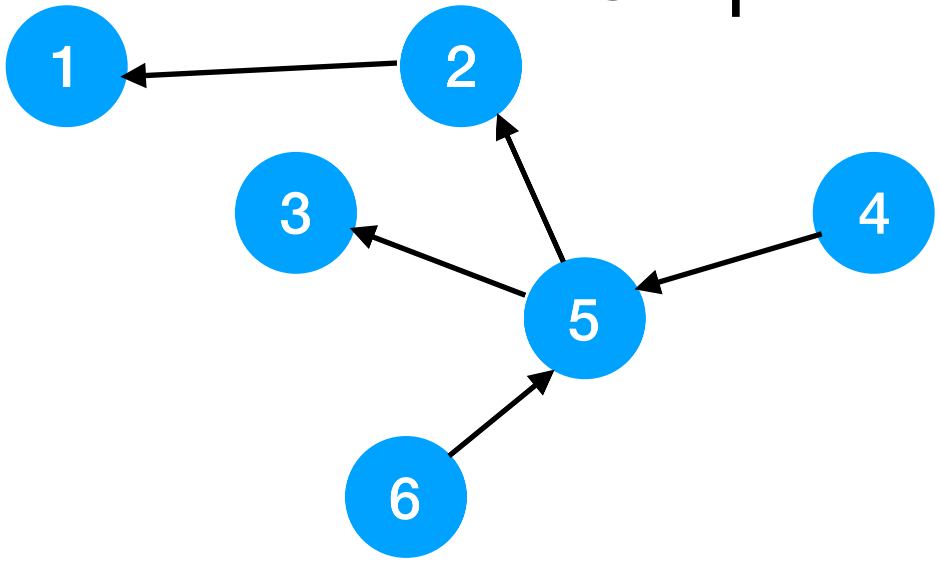
- What is |E| in graph H?
  A: 4 B: 5 C: 6 D: 7

- Is H connected?
  A: no B: yes

Graph G



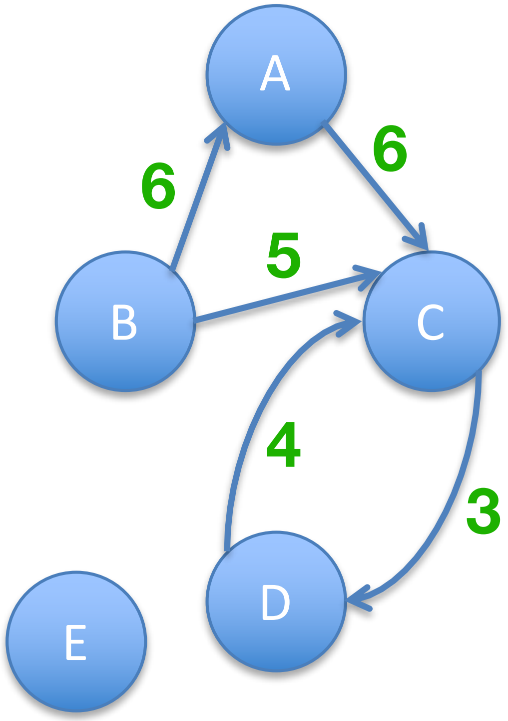
Graph H



Back to graph traversals…

Weighted Graphs

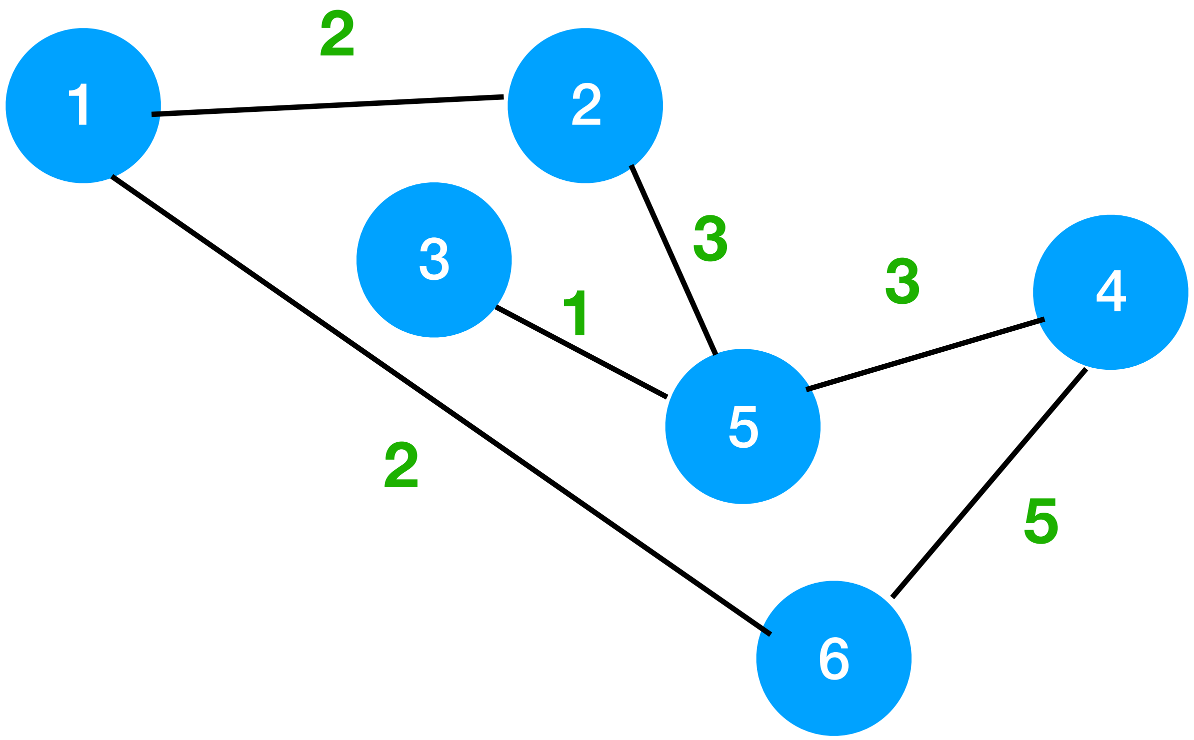
- Like a normal graph, but edges have weights.
- Formally: a graph (V,E) with an accompanying weight function w: E -> $\mathbb{R}$
  - may be directed or undirected.
- Informally: label edges with their weights



- 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)

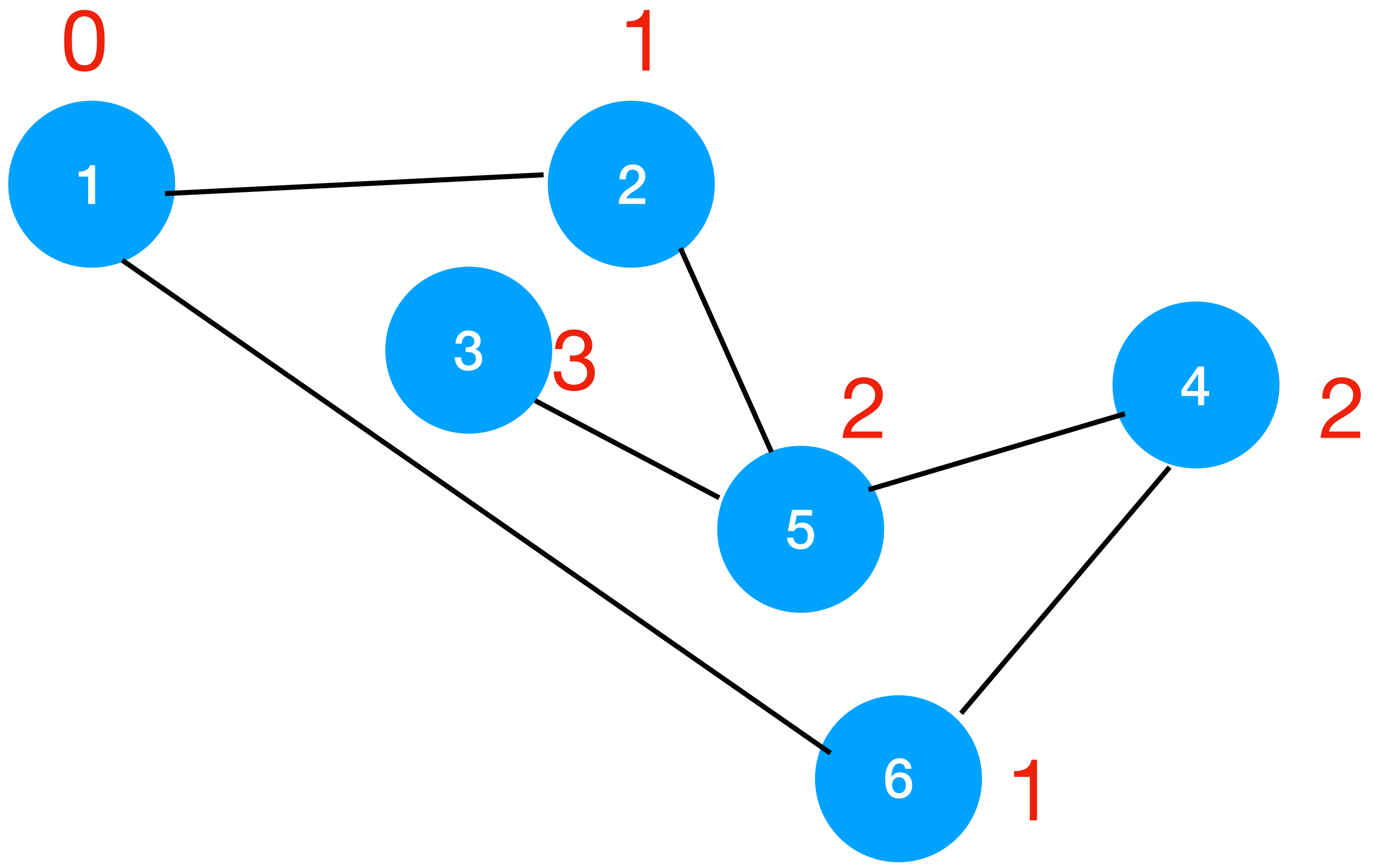
Paths in Weighted Graphs

- 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



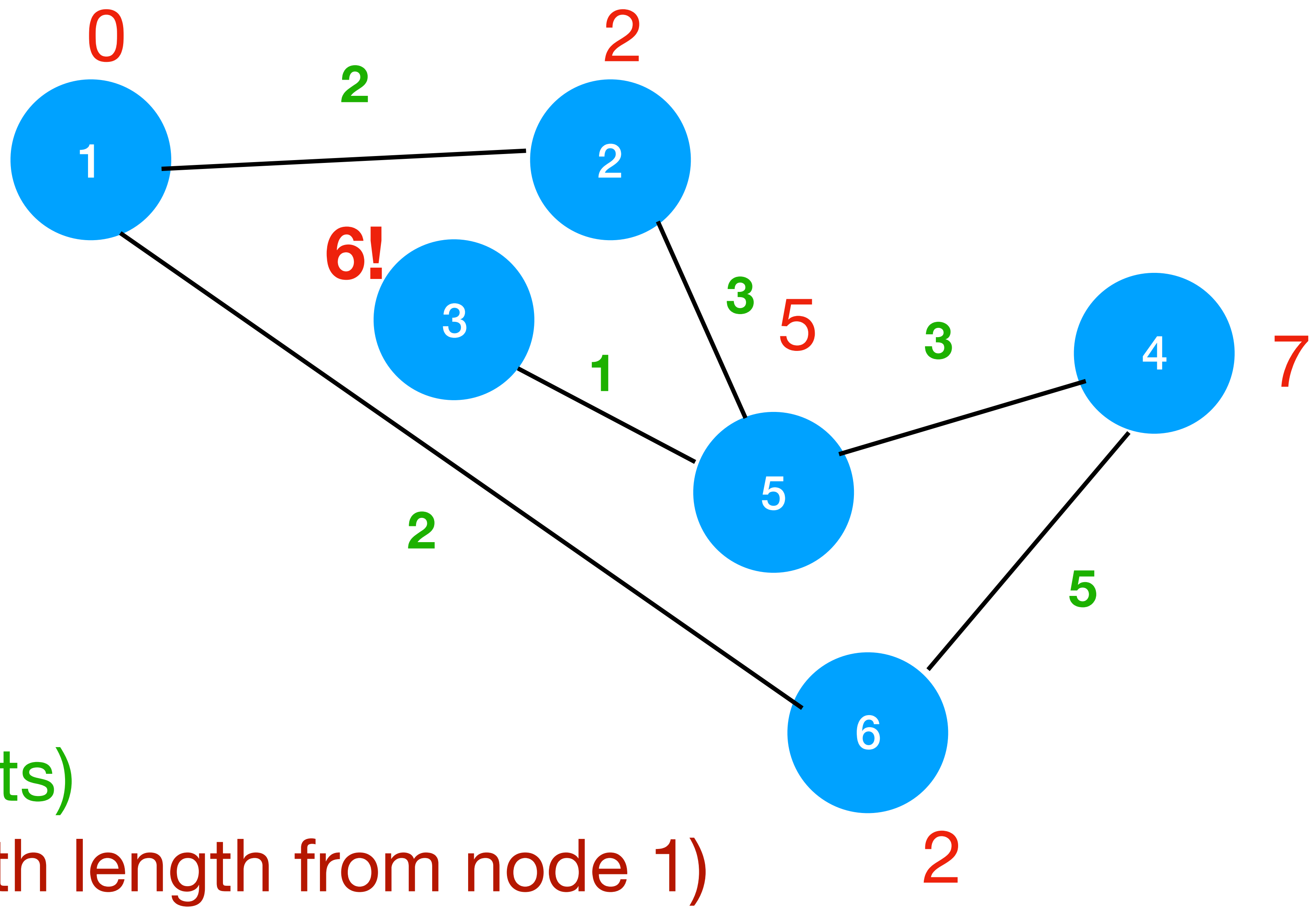
Computing Shortest Paths in Unweighted Graphs

- 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 Weighted Graphs

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



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

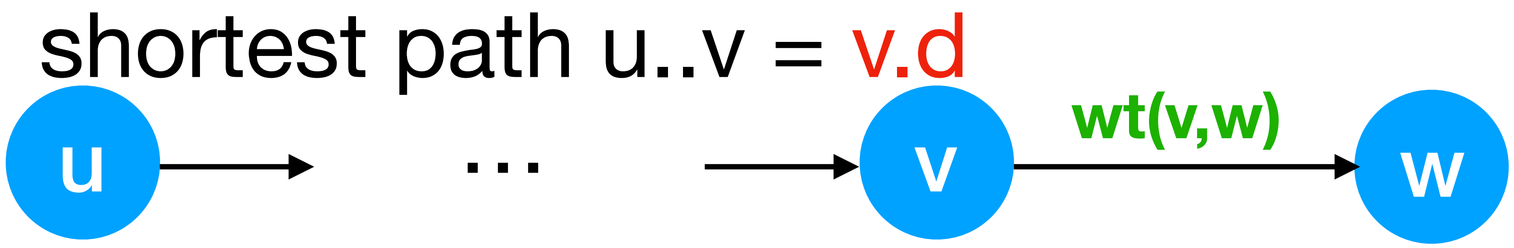
Dijkstra’s Shortest Paths: Subpaths

- 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

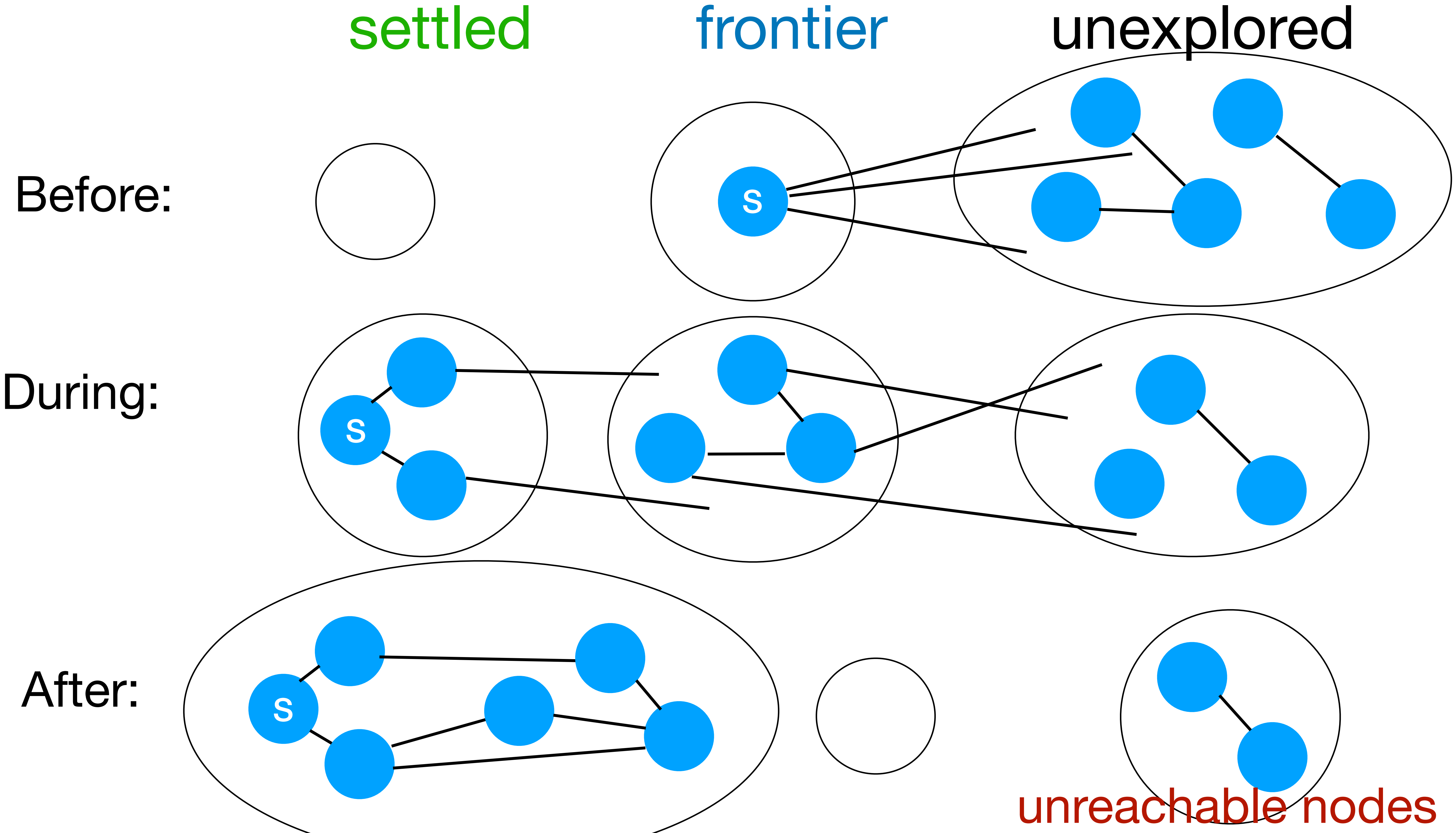
- Fact: **subpaths** of shortest paths are shortest paths
- Consequence: a **candidate** shortest path from start node **s** to some node **v**'s neighbor **w** is the shortest path from to **v** + the edge weight from **v** to **w**.



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:

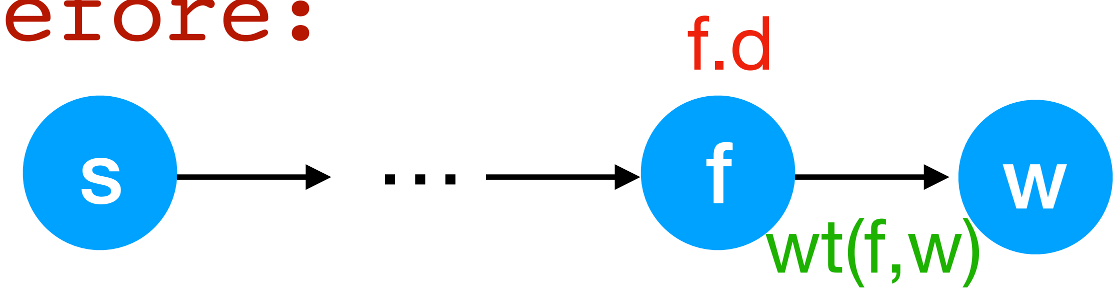
unreachable nodes

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
```

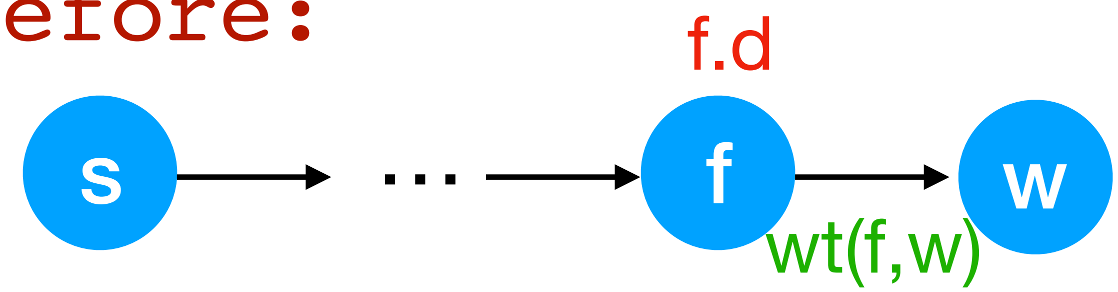
Dijkstra’s Shortest Paths: High-Level Algorithm

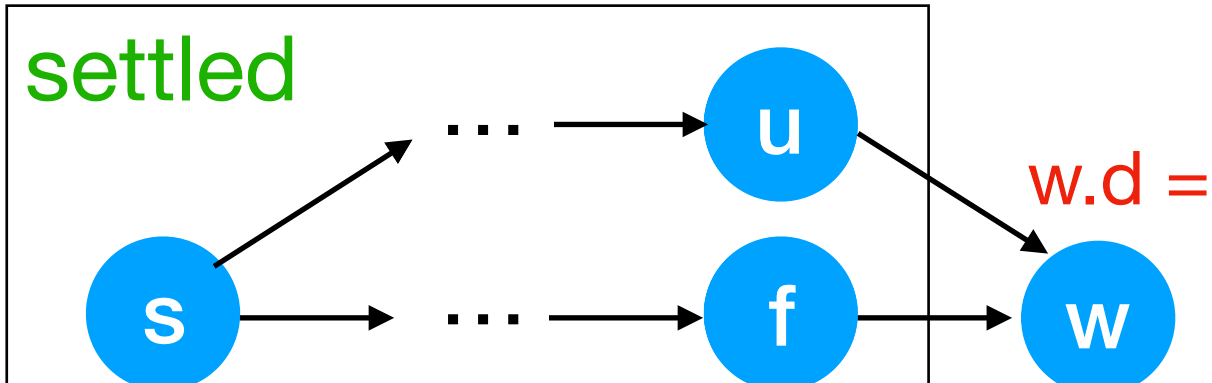
```
Initialize Settled to empty
Initialize Frontier to the start node
While the frontier isn’t empty:
   move the node f with smallest d from F to S
   For each neighbor w of f:
      if we’ve never seen w before:
         set its path length
         add it to frontier
   else if the path to w via f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

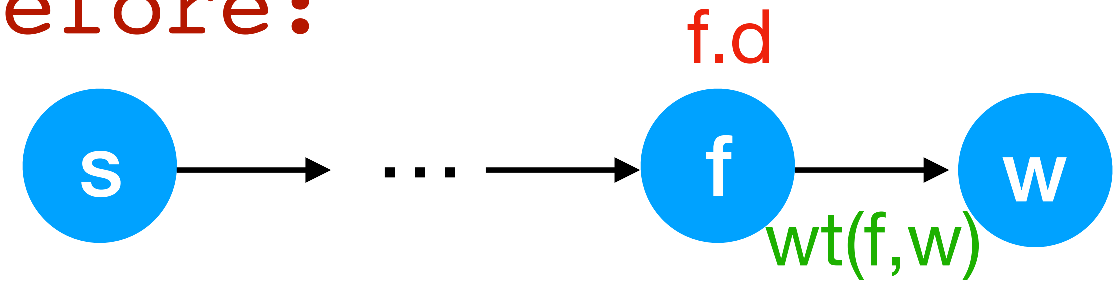


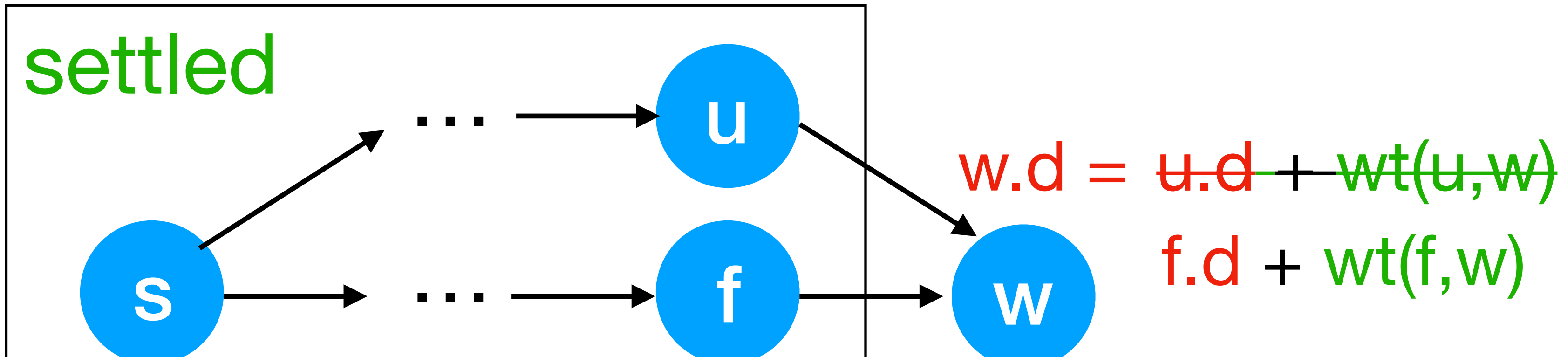


w.d = u.d + wt(u,w)

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

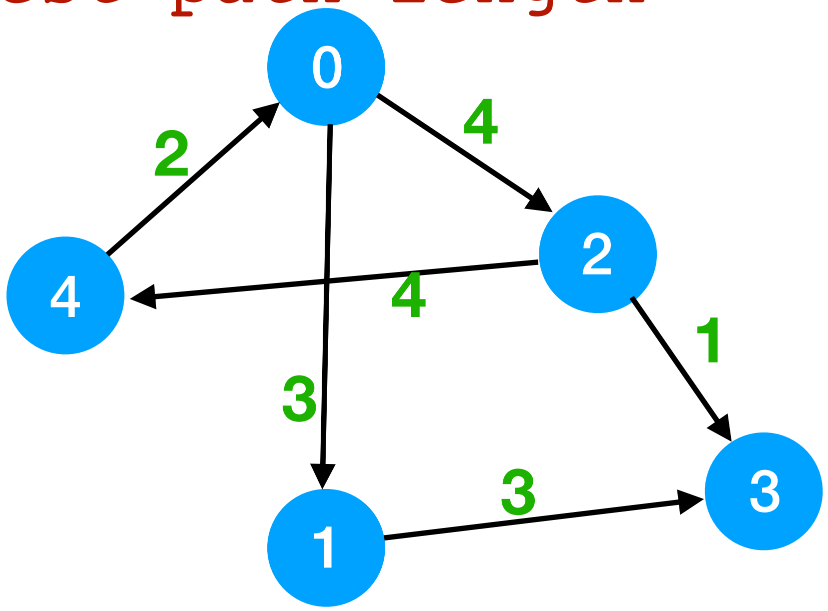
Best known distances:

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

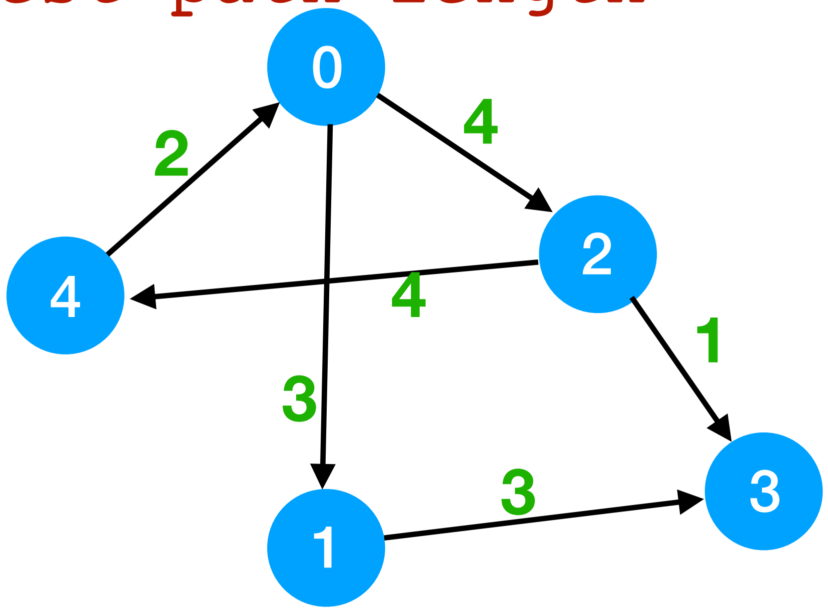
Best known distances:

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:

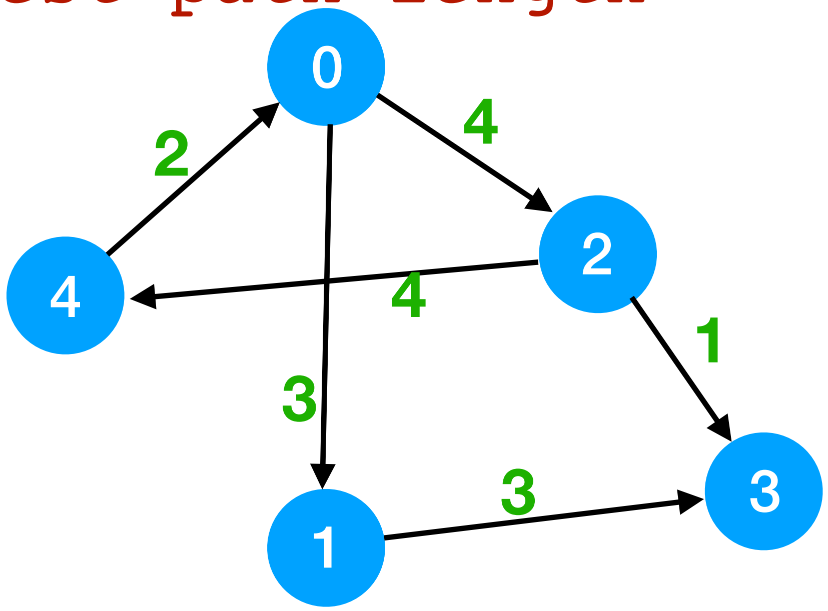
Node	d
0	?
1	?
2	?
3	?
4	0

```
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:

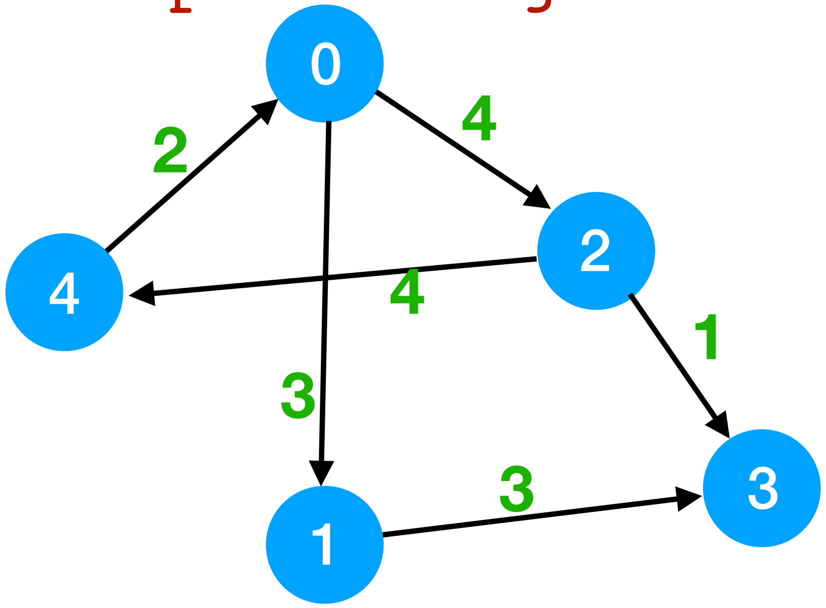
Node	d
0	2
1	?
2	?
3	?
4	0

```
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:

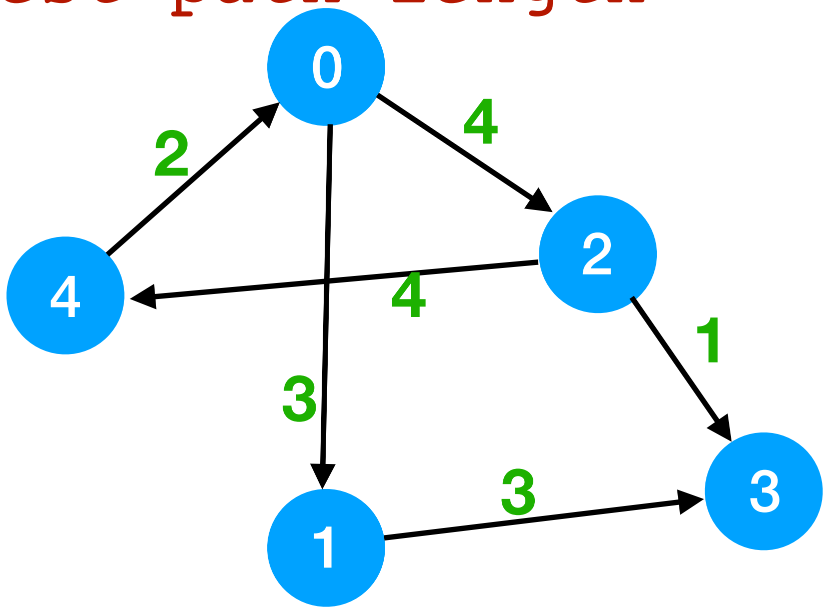
Node	d
0	2
1	?
2	?
3	?
4	0

```
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:

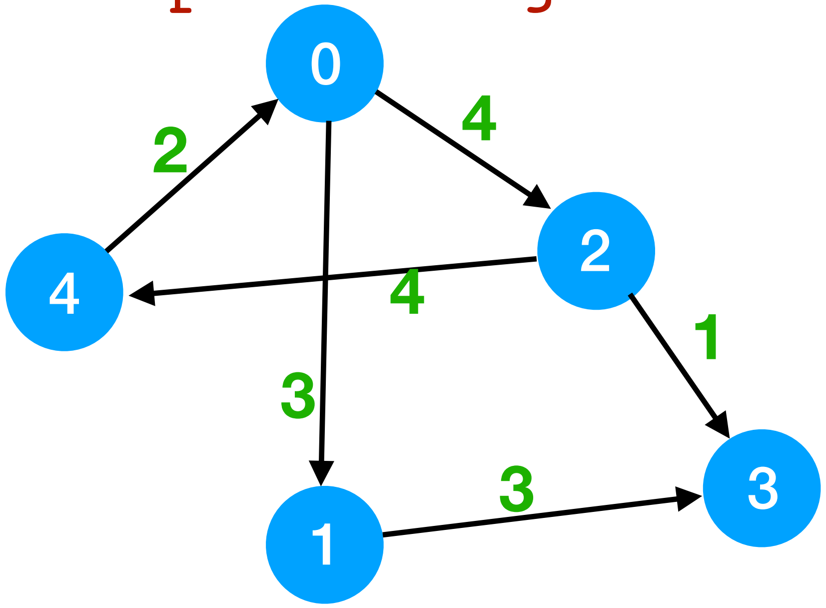
Node	d
0	2
1	**5**
2	?
3	?
4	0

```
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:

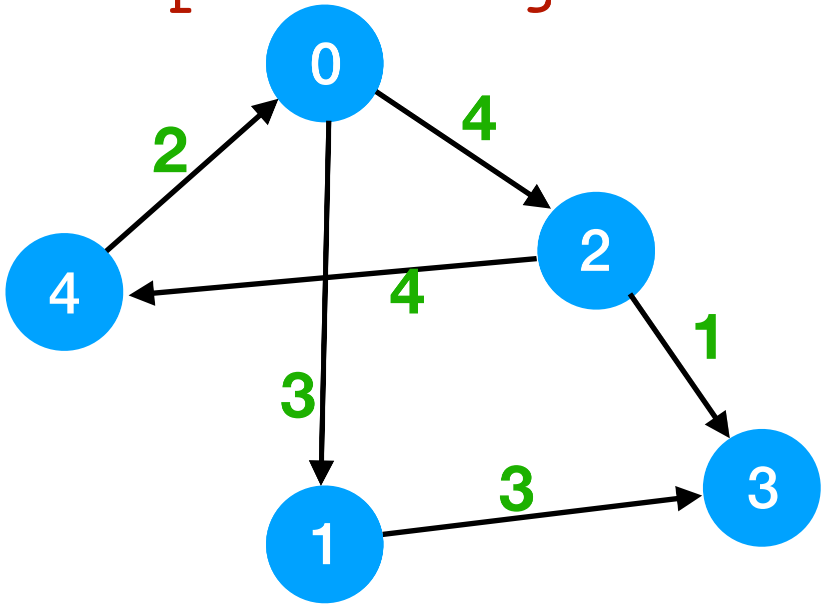
Node	d
0	2
1	5
2	6
3	?
4	0

```
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:

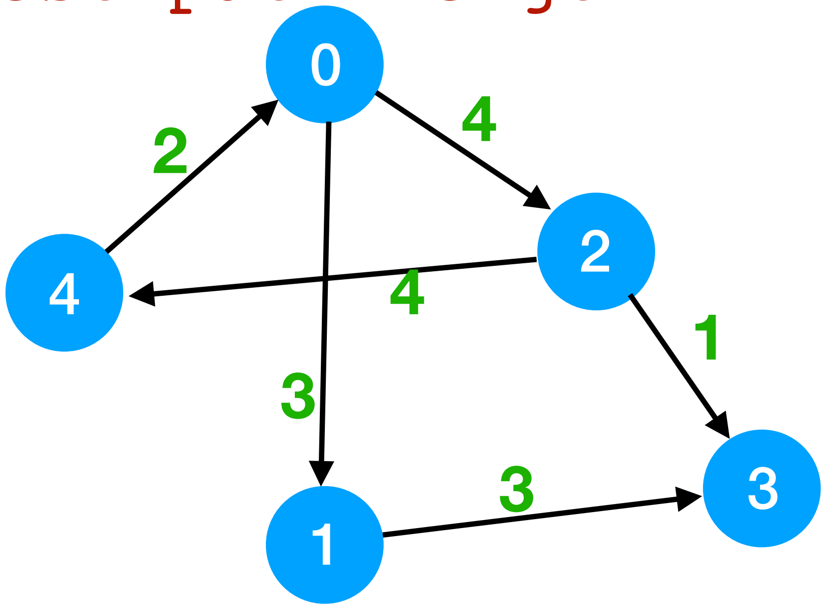
Node	d
0	2
1	5
2	6
3	**8**
4	0

```
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:

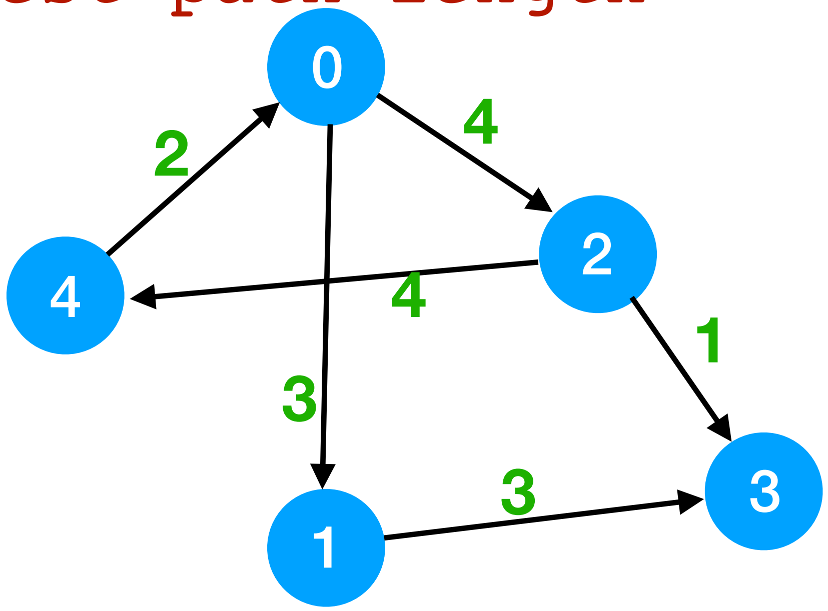
Node	d
0	2
1	5
2	6
3	8
4	0

```
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:

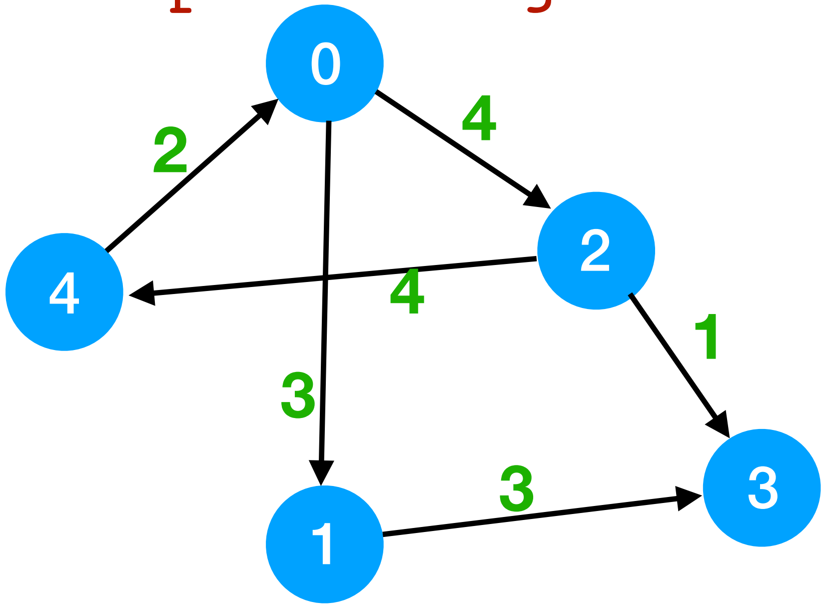
Node	d
0	2
1	5
2	6
3	8
4	0

```
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
```

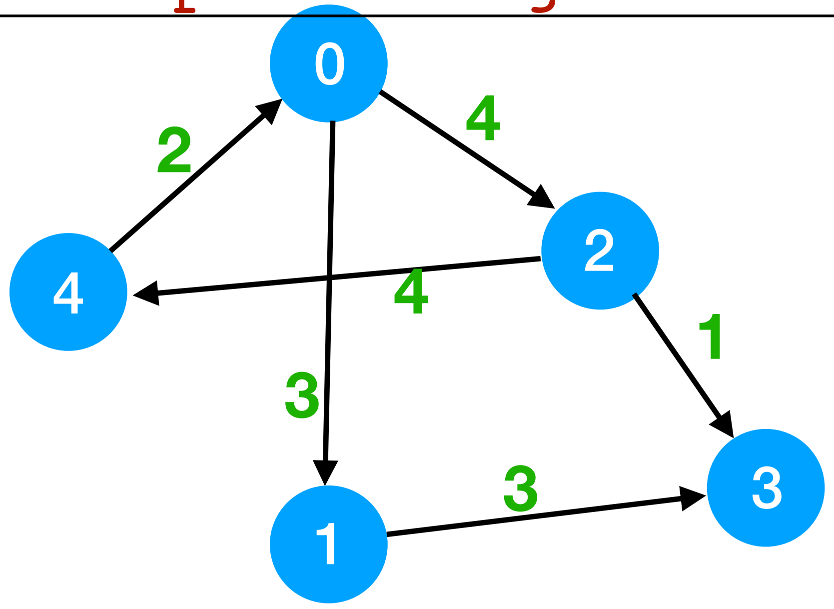
f: 2
w: 3

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

$7 < 8$

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

Frontier set: {3}



`shortest-paths(4)`

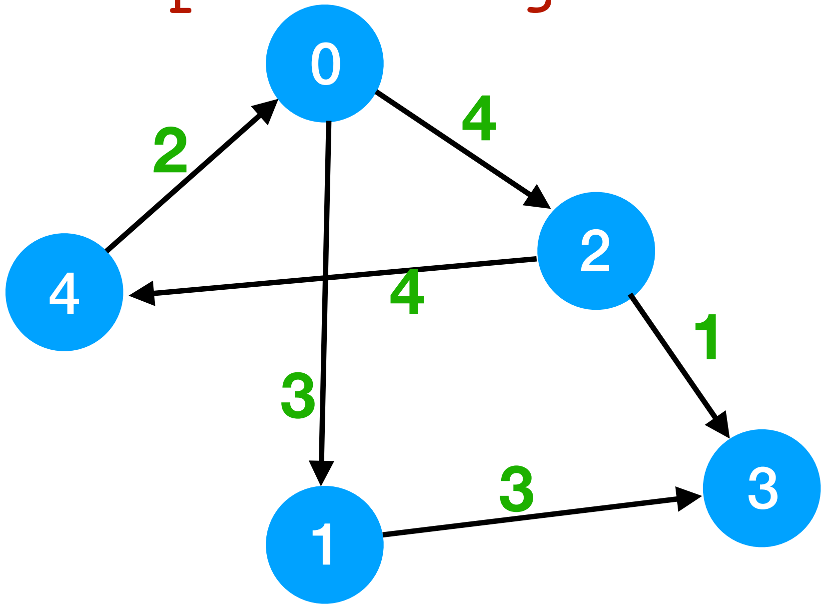
Dijkstra’s Shortest Paths: Execution

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



shortest-paths(4)

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

Frontier set: {} Empty => done!

Unanswered Questions

- Does this always work?
- How do you get the path, not just its length?
- How do you implement it efficiently?
- What’s the runtime?