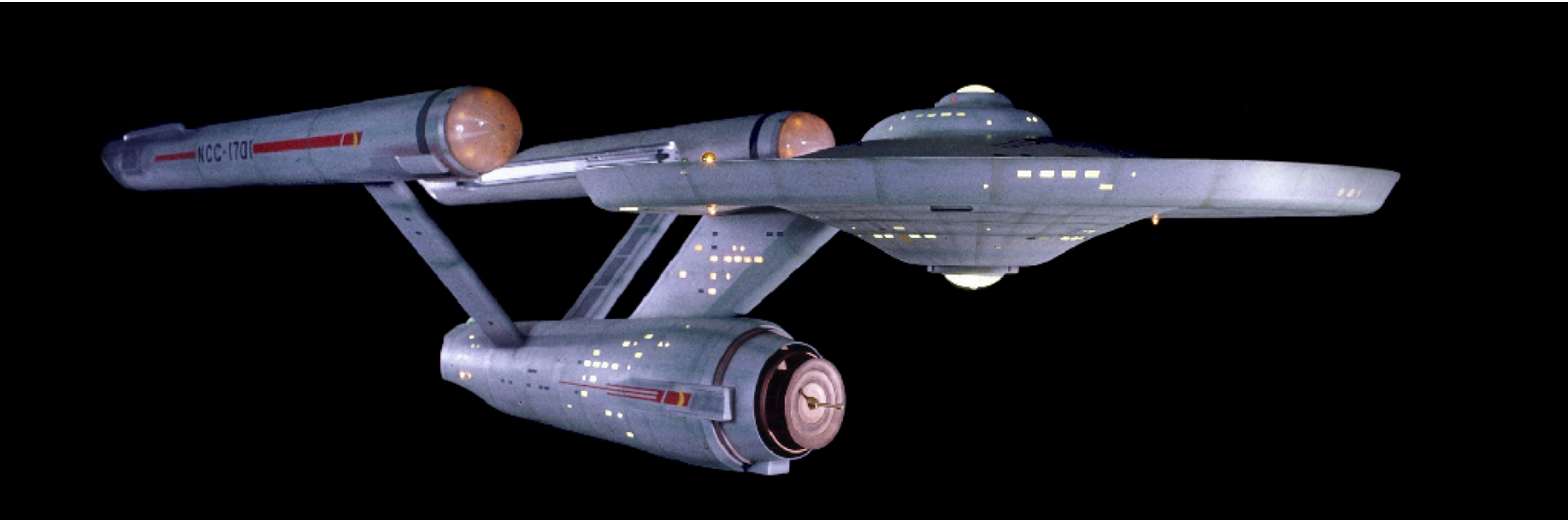


CSCI 241

Lecture 23
Dijkstra’s Algorithm:
Implementation, Proof of Correctness

Announcements

- A4: Implement Dijkstra.
  - Out this afternoon; due Sunday 12/2.
- A2: You can revise your code for half-credit back on unit test correctness points.

Goals

- Know how to implement Dijkstra efficiently.
- Know how to augment the algorithm to keep backpointers in order to reconstruct the sequence of nodes in a shortest path.
- Understand a proof that Dijkstra’s algorithm is correct.

Dijkstra's Shortest Paths: Intuition

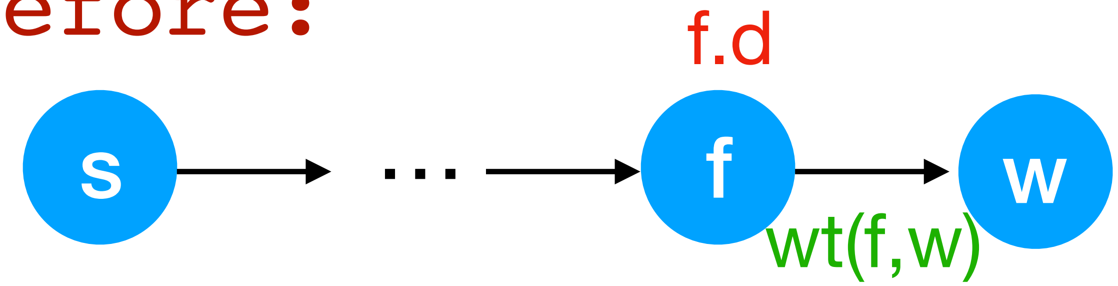
- Intuition: explore nodes kinda like BFS.
- There are three kinds of nodes:
  - Settled - nodes for which we know the actual shortest path.
  - Frontier - nodes that have been visited but we don't necessarily have their actual shortest path
  - Unexplored - all other nodes.
- Each node n keeps track 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: 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
```

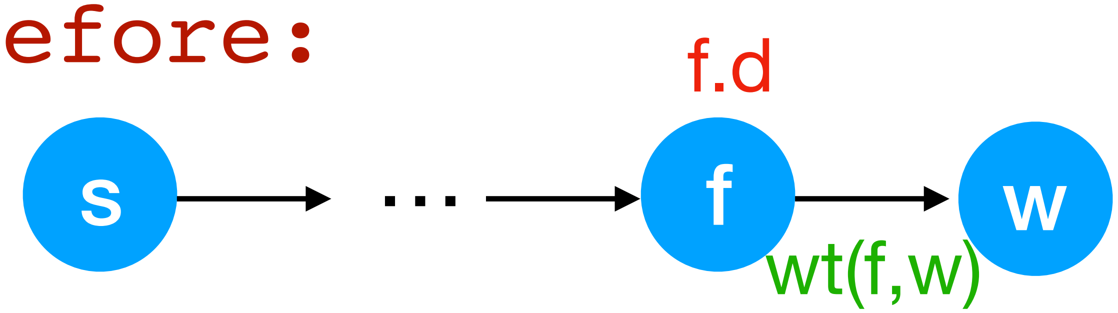
Dijkstra's Shortest Paths: High-Level Algorithm

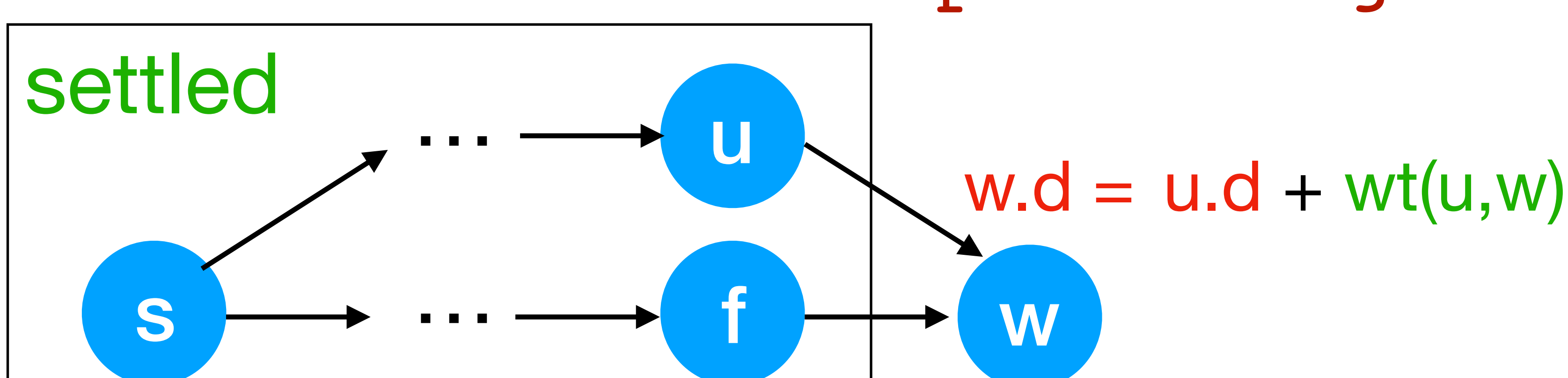
```
Initialize Settled to empty
Initialize Frontier to the start node
While the frontier isn't empty:
  move the node f with smallest d from F to S
  For each neighbor w of f:
    if we've never seen w before:
      set its path length
      add it to frontier
  else if the path to w via f is shorter: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



Dijkstra's Shortest Paths: High-Level Algorithm

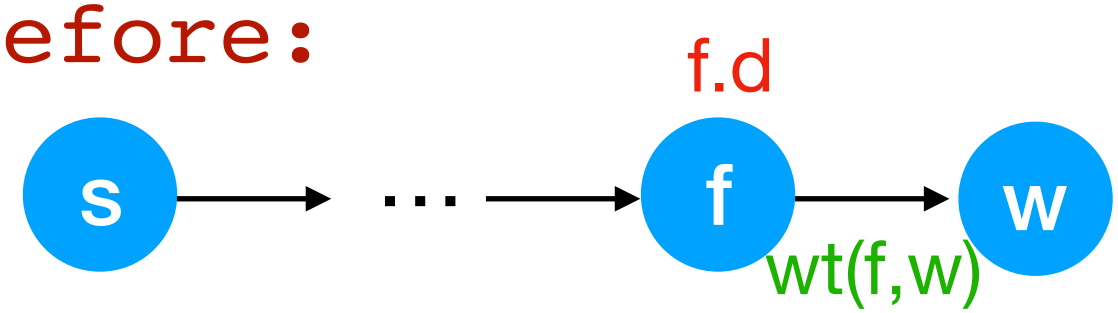
```
Initialize Settled to empty
Initialize Frontier to the start node
While the frontier isn't empty:
   move the node f with smallest d from F to S
   For each neighbor w of f:
      if we've never seen w before:
         set its path length
         add it to frontier
   else if the path to w via f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

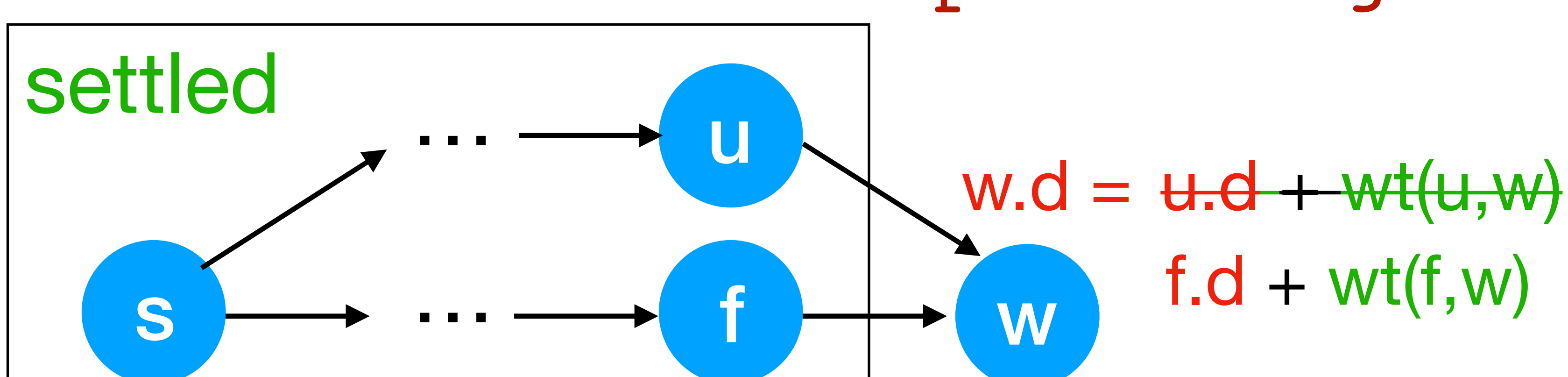




Dijkstra’s Shortest Paths: High-Level Algorithm

```
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: 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;               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
   for each neighbor w of f {                 from F to S
      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;            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;                      Initialize Settled to empty
                                                 Initialize Frontier to the start node
while  (F ≠ {})  {                               While the frontier isn't empty:
   f = node in F with min d value;                  move node f with smallest d
   Remove f from F, add it to S;                      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) {            else if path to w via f is shorter:
         w.d = f.d+weight(f,w);                          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?

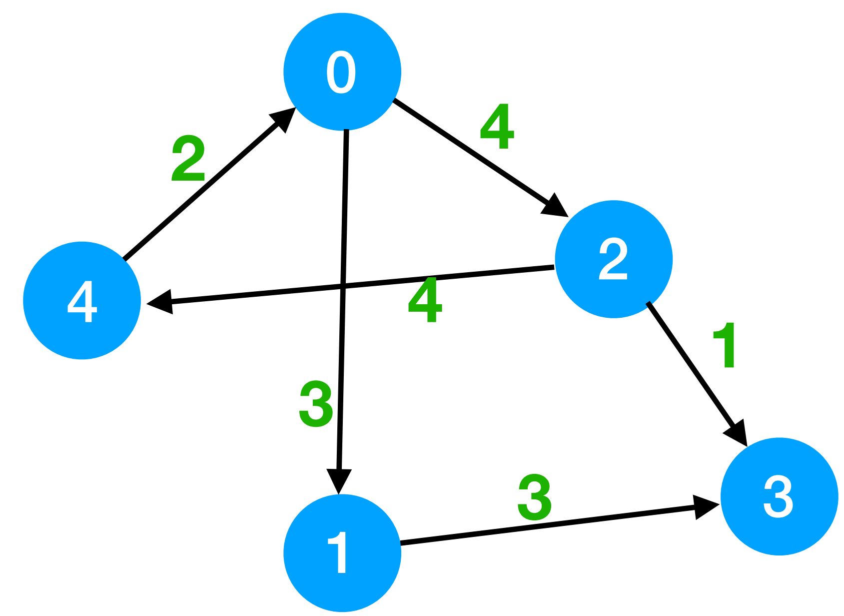
```
S = { }; F = {v};  v.d = 0; v.bp = null;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         w.bp = f;
         add w to F;
      } else if (f.d+weight(f,w) 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

Implementing Dijkstra Efficiently

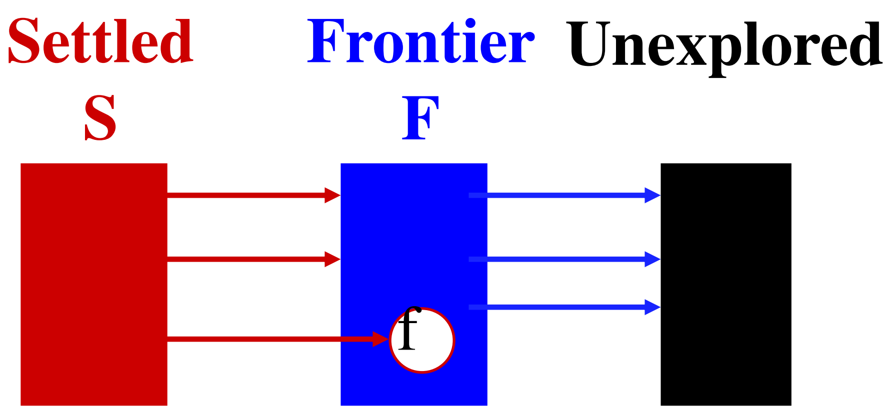
```
S = { }; F = {v};  v.d = 0; v.bp = null;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
          w.d =  f.d + weight(f, w);
          w.bp = f;
          add w to F;
      } else if (f.d+weight(f,w)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

1. Store F 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'll use a HashMap<Node,PathData>
4. Don't need to explicitly store S or Unexplored sets:
   a node is in S or F iff it is in the map.

Proof of Correctness

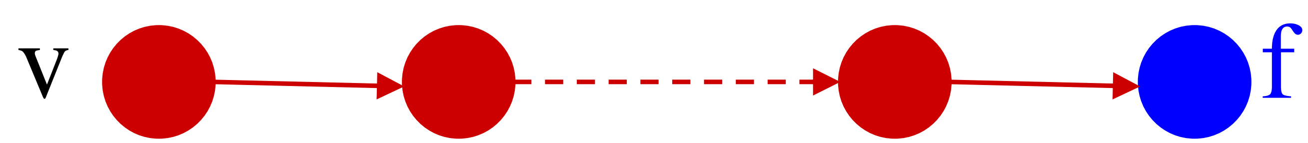
- Dijkstra's algorithm is **greedy**: it makes a sequence of *locally* optimal moves, which results in the *globally* optimal solution.
  - Most algorithms don't work like this - need to prove that it results in the global optimum.
- Specifically: It is not obvious that there cannot still be a shorter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

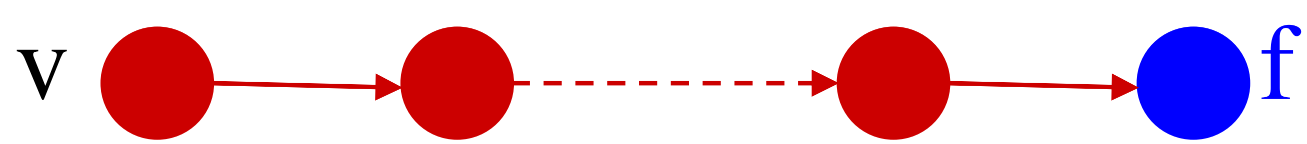
```
S = { }; F = {v};  v.d = 0;
while  (F ≠ {})  {
   f = node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         w.d =  f.d + weight(f, w);
         add w to F;
      } else if (f.d+weight(f,w) < w.d) {
         w.d = f.d+weight(f,w);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

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);
      }
   }
}
```

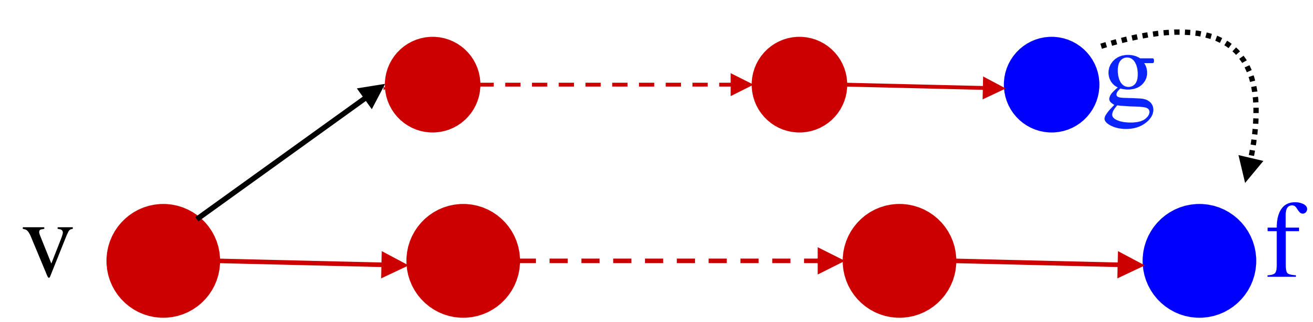
Theorem: For a node f in the Frontier with minimum d value (over all nodes in the Frontier), f.d is the shortest-path distance from v to f.

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.

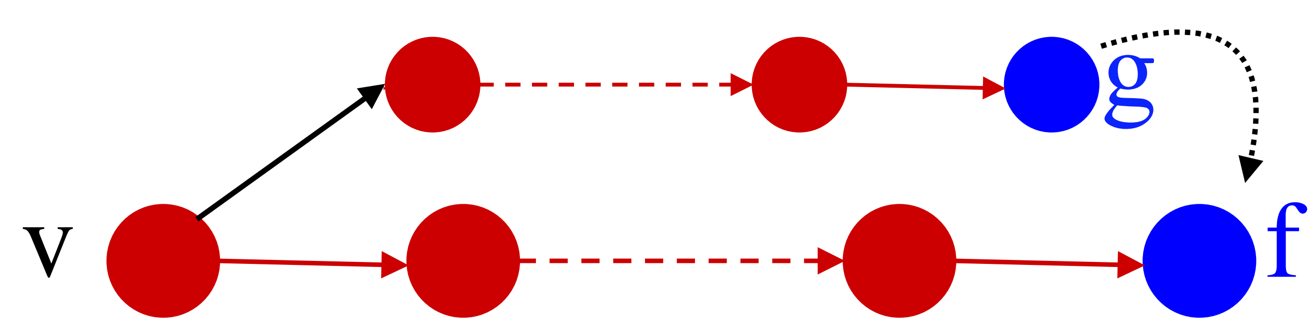
Proof: Show that any other path from v to if has length >= f.d

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:

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. Update the w.d to maintain Part 2.
3. Added each neighbor is either already in F or gets moved there.