Happenings

Today: AWC-organized panel discussion with recent female alumnae working in the tech industry. AW 210 at 4:00.

Tuesday, 3/12 – AWC Bake Sale – 10 am to 2 pm in the CF 1st Floor Lobby

Wednesday, 3/13 – Peer Lecture Series: Machine Learning – 5 pm in CF 420

Thursday, 3/14 – CS Study Break: Bingo and Brownies! – 3 pm in the CF 4th Floor Foyer
Join us for brownies and bingo!

3 - 4 PM
THURSDAY, MARCH 14
CF 4TH FLOOR FOYER

Bingo called by Aran Clauson

For disability resources, contact 360-650-3083
Announcements

• No quiz today.
• Quiz Monday instead.
Goals

• Be able to run Dijkstra’s algorithm on paper.

• Know how to implement Dijkstra efficiently (as in A4)

• Know how to augment the algorithm to keep backpointers in order to reconstruct the sequence of nodes in a shortest path.
Dijkstra’s Shortest Paths: Cartoon

Before:

During:

After:

settled  frontier  unexplored

unreachable nodes
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

\[
\text{settled} \quad \ldots \quad u \quad \ldots \quad w \quad s \quad \ldots \quad f \quad w
\]

\[
w.d = u.d + wt(u,w)
\]

\[
f.d = wt(f,w)
\]
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

\[ w.d = u.d + wt(u, w) \]
\[ f.d + wt(f, w) \]
Dijkstra’s Shortest Paths: Execution

Best known distances:

<table>
<thead>
<tr>
<th>Node</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>2</td>
<td>?</td>
</tr>
<tr>
<td>3</td>
<td>?</td>
</tr>
<tr>
<td>4</td>
<td>?</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Node</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>2</td>
<td>?</td>
</tr>
<tr>
<td>3</td>
<td>?</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Node</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>2</td>
<td>?</td>
</tr>
<tr>
<td>3</td>
<td>?</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

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: {4}
Frontier set: {}
Dijkstra's Shortest Paths: Execution

Best known distances:

<table>
<thead>
<tr>
<th>Node</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>2</td>
<td>?</td>
</tr>
<tr>
<td>3</td>
<td>?</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

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: \{4\}
Frontier set: \{0\}

shortest-paths(4)
**Dijkstra’s Shortest Paths: Execution**

Best known distances:

<table>
<thead>
<tr>
<th>Node</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>2</td>
<td>?</td>
</tr>
<tr>
<td>3</td>
<td>?</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Settled set: \{4, 0\}

Frontier set: \{\}

Initialize Settled to empty

Initialize Frontier to the start node

While the frontier isn’t empty:

1. Move the node \(f\) with smallest \(d\) from \(F\) to \(S\)
2. 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

```
shortest-paths(4)
```

```plaintext
f: 0
```
Dijkstra’s Shortest Paths: Execution

Best known distances:

<table>
<thead>
<tr>
<th>Node</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>?</td>
</tr>
<tr>
<td>3</td>
<td>?</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

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: {4, 0}
Frontier set: {1}

shortest-paths(4)
Dijkstra’s Shortest Paths: Execution

Best known distances:

<table>
<thead>
<tr>
<th>Node</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>?</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Settled set: {4, 0}

Frontier set: {1, 2}

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

shortest-paths(4)
Dijkstra’s Shortest Paths: Execution

Best known distances:

<table>
<thead>
<tr>
<th>Node</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

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: {4, 0, 1}

 Frontier set: {2}
Dijkstra’s Shortest Paths: Execution

Best known distances:

<table>
<thead>
<tr>
<th>Node</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

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: \( \{4, 0, 1\} \)

Frontier set: \( \{2, 3\} \)
Dijkstra’s Shortest Paths: Execution

**Best known distances:**

<table>
<thead>
<tr>
<th>Node</th>
<th>( d )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

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: \( \{4, 0, 1, 2\} \)

Frontier set: \( \{3\} \)
Dijkstra’s Shortest Paths: Execution

Best known distances:

<table>
<thead>
<tr>
<th>Node</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

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

2.d + wt(2,3) < 3.d
7 < 8

Settled set: {4, 0, 1, 2}
Frontier set: {3}
Dijkstra’s Shortest Paths: Execution

Best known distances:

<table>
<thead>
<tr>
<th>Node</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

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

Frontier set: {}  Empty => done!

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

shortest-paths(4)
Dijkstra’s Shortest Paths: Pseudocode

\[ S = \{ \}; \quad F = \{ v \}; \quad v.d = 0; \]

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
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 {
        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

\[ S = \{ \} \text{; } F = \{ v \} \text{; } v.d = 0; \]

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

- 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
Dijkstra’s Shortest Paths: Pseudocode

\[ S = \{ \}; \quad F = \{ v \}; \quad v.d = 0; \]

\[ \text{Initialize Settled to empty} \]
\[ \text{Initialize Frontier to the start node} \]

\[ \text{while } (F \neq \{ \}) \{ \]
\[ f = \text{node in F with min d value; } \]
\[ \text{Remove f from F, add it to S; } \]
\[ \text{for each neighbor w of f } \{ \]
\[ \quad \text{if (w not in S or F) } \{ \]
\[ \quad \quad w.d = f.d + \text{weight}(f, w); \]
\[ \quad \quad \text{add w to F;} \]
\[ \quad \} \text{ else if } (f.d + \text{weight}(f, w) < w.d) \{ \]
\[ \quad \quad w.d = f.d + \text{weight}(f, w); \]
\[ \} \]
\[ \} \]
\[ \text{While the frontier isn’t empty:} \]
\[ \quad \text{move node f with smallest d from F to S} \]
\[ \text{For each neighbor w of f:} \]
\[ \quad \text{if we’ve never seen w before:} \]
\[ \quad \quad \text{set its path length} \]
\[ \quad \quad \text{add it to frontier} \]
\[ \quad \text{else if path to w via f is shorter:} \]
\[ \quad \quad \text{update w’s shortest path length} \]
What if we want to know the shortest path?

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

- At termination: for each reachable node \( n \), \( n.d \) stores the \textbf{length} of the shortest path from \( v \) to \( n \).

- We didn’t keep track of \textbf{how} to get from \( v \) to \( n \)!
What if we want to know the shortest path?

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.

```plaintext
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
        }
    }
}
```
What if we want to know the shortest path? Example

```plaintext
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
        }
    }
}
```

Strategy: maintain a backpointer at each node pointing to the previous node in the shortest path.
Let’s Dijkstra

• Half get the algorithm, half get the graphs. Pair up with someone with the other thing.

• Run the algorithm on each graph:
  • (first) 5-node graph: start at node S
  • (second) Other graph: start at node D
The next slide is really 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)

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

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)

\[
S = \{\}; \quad F = \{v\}; \quad v.d = 0; \quad v.bp = \textbf{null};
\]

\[
\text{while} \ (F \neq \{\}) \ {\}$
\[
\quad f = \text{node in } F \text{ with min } d \text{ value};
\]

Remove \(f\) from \(F\), add it to \(S\);

for each neighbor \(w\) of \(f\) {
\[
\text{if} \ (w \not\in S \text{ or } F) \ {\}$
\[
\quad w.d = f.d + \text{weight}(f, w);
\]

\[
\quad w.bp = f;
\]

\[
\quad \text{add } w \text{ to } F;
\]

\[
\text{else if} \ (f.d + \text{weight}(f,w) < w.d) \ {\}$
\[
\quad w.d = f.d + \text{weight}(f,w);
\]

\[
\quad w.bp = f
\]

\]

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\(<\text{Node},\text{PathData}>\).

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)

\[ S = \{ \}; \quad F = \{ v \}; \quad v.d = 0; \quad v.bp = \text{null}; \]

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

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)

\[ S = \{ \} ; F = \{ v \} ; v.d = 0 ; v.bp = \text{null}; \]

\[ \text{while} \ (F \neq \{\}) \ {\}
\]

\[ f = \text{node in F with min d value}; \]

\[ \text{Remove f from F, add it to S}; \]

\[ \text{for each neighbor w of f} \ {\}
\]

\[ \text{if (w not in S or F)} \ {\}
\]

\[ w.d = f.d + \text{weight}(f, w); \]

\[ w.bp = f; \]

\[ \text{add w to F}; \]

\[ \text{else if (f.d+weight(f,w) < w.d)} \ {\}
\]

\[ w.d = f.d+weight(f,w); \]

\[ w.bp = f \]

\[ \}
\]

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\langle Node, PathData\rangle.

4. No need to explicitly store Settled 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 the length of the shortest \( v \rightarrow s \) path.

2. For a Frontier node \( f \), at least one \( v \rightarrow f \) 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:** 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 $f$ has length $\geq f.d$.

```plaintext
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) {
            $w.d = f.d + \text{weight}(f, w)$;
            add $w$ to F;
        } else if ($f.d+\text{weight}(f,w) < w.d$) {
            $w.d = f.d+\text{weight}(f,w)$;
        }
    }
    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**

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 \( f \) has length \( \geq f.d \)

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);
        }
    }
}

**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: 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 \( f \) has length \( \geq f.d \).

\[
S = \{ \}; \quad F = \{v\}; \quad v.d = 0;
\]

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 \((w \text{ not in } S \text{ or } F)\) {

\( w.d = f.d + \text{weight}(f, w); \)
add \( w \) to \( F \);
}
else if \((f.d+\text{weight}(f,w) < w.d)\) {

\( w.d = f.d+\text{weight}(f,w); \)
}
}

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**: 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 \( f \) has length \( \geq f.d \).

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 \( (w \text{ not in } S \text{ or } F) \) {
    - \( w.d = f.d + \text{weight}(f, w) \);
    - add \( w \) to \( F \);
  }
  - else if \( (f.d + \text{weight}(f,w) < w.d) \) {
    - \( w.d = f.d + \text{weight}(f,w) \);
  }
}

**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 \( f \):

\( d.f \leq d.g \),

so that path cannot be shorter.
Proof of Correctness:
Invariant Maintenance

\[ S = \{ \}; \quad F = \{v\}; \quad v.d = 0; \]

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

1. For a Settled node \( s \), a shortest path from \( v \) to \( s \) contains only settled nodes and \( s.d \) is length of shortest \( v \right\rightarrow s \) path.

2. For a Frontier node \( f \), at least one \( v \right\rightarrow f \) 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

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

2. For a Frontier node $f$, at least one $v \rightarrow f$ 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

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)

\[
S = \{ \}; F = \{v\}; \quad v.d = 0;
\]

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

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.