Dijkstra's Algorithm: Efficient Implementation
Goals

Be prepared to implement Dijkstra's algorithm efficiently.
This bit is very important.
Implementing Dijkstra Efficiently (A4)

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

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. To store w.d and w.bp, we will use a HashMap<Node,PathData>

4. We don't 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. To store w.d and w.bp, we will use a HashMap<Node,PathData>

4. We don't 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)

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

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. To store w.d and w.bp, we will use a HashMap<Node,PathData>

4. We don't 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. To store w.d and w.bp, we will use a HashMap<Node,PathData>

4. We don't 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)

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

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

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. To store \( w.d \) and \( w.bp \), we will use a HashMap<\text{Node}, PathData>.

4. \textbf{We don't need to explicitly store Settled or Unexplored sets: a node is in } S \text{ or } F \text{ iff it is in the map.}
Implementing Dijkstra Efficiently (A4)

\begin{verbatim}
S = {}; F = {v}; v.d = 0; v.bp = null;
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);
            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
        }
    }
}
\end{verbatim}

4. We don't need to explicitly store Settled or Unexplored sets: w is in S or F if and only if 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!