SHORTEST PATH ALGORITHM
A7. Implement shortest-path algorithm

Last semester: mean time: 3.7 hrs, median time: 4.0 hrs.

We give you complete set of test cases and a GUI to play with. Efficiency and simplicity of code will be graded.

Read pinned Assignment A7 note carefully:

2. Important! Grading guidelines.

We demo it.

A6 due Thursday.
Working with a partner? Group before submitting!!
Dijkstra’s shortest-path algorithm

Edsger Dijkstra, in an interview in 2010 (CACM):

... the algorithm for the shortest path, which I designed in about 20 minutes. One morning I was shopping in Amsterdam with my young fiance, and tired, we sat down on the cafe terrace to drink a cup of coffee, and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a 20-minute invention. [Took place in 1956]


Visit http://www.dijkstraescry.com for all sorts of information on Dijkstra and his contributions. As a historical record, this is a gold mine.
Dijkstra’s shortest-path algorithm

Dijkstra describes the algorithm in English:

- When he designed it in 1956 (he was 26 years old), most people were programming in assembly language.
- Only one high-level language: Fortran, developed by John Backus at IBM and not quite finished.

No theory of order-of-execution time — topic yet to be developed. In paper, Dijkstra says, “my solution is preferred to another one … “the amount of work to be done seems considerably less.”

1968 NATO Conference on Software Engineering

- In Garmisch, Germany
- Academicians and industry people attended
- For first time, people admitted they did not know what they were doing when developing/testing software. Concepts, methodologies, tools were inadequate, missing
- The term *software engineering* was born at this conference.

Get a good sense of the times by reading these reports!
1968 NATO Conference on Software Engineering, Garmisch, Germany

Term “software engineering” coined for this conference
1968 NATO Conference on Software Engineering, Garmisch, Germany
Beards

The reason why some people grow aggressive tufts of facial hair is that they do not like to show the chin that isn't there.

*a grook by Piet Hein*
Dijkstra’s shortest path algorithm

The n (> 0) nodes of a graph numbered 0..n-1.

Each edge has a positive weight.

wgt(v1, v2) is the weight of the edge from node v1 to v2.

Some node v be selected as the start node.

Calculate length of shortest path from v to each node.

Use an array d[0..n-1]: for each node w, store in d[w] the length of the shortest path from v to w.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

![Diagram of graph with node labels and edge weights]
1. For a Settled node $s$, a shortest path from $v$ to $s$ contains only settled nodes and $d[s]$ 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 $d[f]$ is the length of the shortest such path.

3. All edges leaving $S$ go to $F$.

Another way of saying 3: There are no edges from $S$ to the far-off set.
1. For a Settled node $s$, $d[s]$ is length of shortest $v \rightarrow s$ path.

2. For a Frontier node $f$, $d[f]$ is length of shortest $v \rightarrow f$ path using only Settled nodes (except for $f$).

3. All edges leaving $S$ go to $F$.

**Theorem.** For a node $f$ in $F$ with minimum $d$ value (over nodes in $F$), $d[f]$ is the length of a shortest path from $v$ to $f$.

**Case 1:** $v$ is in $S$.

**Case 2:** $v$ is in $F$. Note that $d[v]$ is 0; it has minimum $d$ value
The algorithm

1. For $s$, $d[s]$ is length of shortest $v \rightarrow s$ path.
2. For $f$, $d[f]$ is length of shortest $v \rightarrow f$ path using red nodes (except for $f$).
3. Edges leaving $S$ go to $F$.

**Theorem:** For a node $f$ in $F$ with min $d$ value, $d[f]$ is shortest path length

Loopy question 1:
How does the loop start? What is done to truthify the invariant?
The algorithm

1. For s, \( d[s] \) is length of shortest \( v \rightarrow s \) path.
2. For f, \( d[f] \) is length of shortest \( v \rightarrow f \) path using red nodes (except for f).
3. Edges leaving S go to F.

Theorem: For a node \( f \) in F with min d value, \( d[f] \) is shortest path length.

S = \{ \}; F = \{ v \}; \( d[v] = 0 \);
while ( F ≠ \{ \} ) {

Loopy question 2:
When does loop stop? When is array \( d \) completely calculated?
The algorithm

\[ S = \{ \}; F = \{ v \}; d[v] = 0; \]
\[ \text{while}( F \neq \{\} ) \{ \]
\[ f = \text{node in } F \text{ with min } d \text{ value; } \]
\[ \text{Remove } f \text{ from } F, \text{ add it to } S; \]

1. For \( s \), \( d[s] \) is length of shortest \( v \rightarrow s \) path.

2. For \( f \), \( d[f] \) is length of shortest \( v \rightarrow f \) path using red nodes (except for \( f \)).

3. Edges leaving \( S \) go to \( F \).

Theorem: For a node \( f \) in \( F \) with min \( d \) value, \( d[f] \) is shortest path length

Loopy question 3: Progress toward termination?
The algorithm

S = \{ \}; F = \{ v \}; d[v] = 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 {
        \text{if} (w \text{ not in } S \text{ or } F) {
            \}
        \text{else} {
            \}
    }
}

1. For s, d[s] is length of shortest v \to s path.

2. For f, d[f] is length of shortest v \to f path using red nodes (except for f).

3. Edges leaving S go to F.

Theorem: For a node f in F with min d value, d[f] is shortest path length

Loopy question 4: Maintain invariant?
The algorithm

S

F

Far off

1. For s, \(d[s]\) is length of shortest \(v \rightarrow s\) path.

2. For \(f\), \(d[f]\) is length of shortest \(v \rightarrow f\) path using red nodes (except for \(f\)).

3. Edges leaving \(S\) go to \(F\).

Theorem: For a node \(f\) in \(F\) with min d value, \(d[f]\) is shortest path length

Loopy question 4: Maintain invariant?

```cpp
S = \{ \}\; ; \; F = \{ v \}\; ; \; d[v] = 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) {
            d[w] = d[f] + wgt(f, w);
            add w to F;
        } else {
        }
    }
}
```
The algorithm

1. For $s$, $d[s]$ is length of shortest $v \rightarrow s$ path.

2. For $f$, $d[f]$ is length of shortest $v \rightarrow f$ path of form

3. Edges leaving $S$ go to $F$.

Theorem: For a node $f$ in $F$ with min $d$ value, $d[f]$ is its shortest path length

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

while ($F \neq \{ \}$)

\begin{align*}
  &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 \{ \\
  &\quad \text{if } (w \text{ not in } S \text{ or } F) \{ \\
  &\qquad d[w] = d[f] + \text{wgt}(f, w); \\
  &\qquad \text{add } w \text{ to } F; \\
  &\quad \} \text{ else} \\
  &\qquad \text{if } (d[f] + \text{wgt} (f,w) < d[w]) \{ \\
  &\qquad\qquad d[w] = d[f] + \text{wgt}(f, w); \\
  &\qquad\} \\
  &\}\}
\end{align*}

Algorithm is finished!
Extend algorithm to include the shortest path

Let’s extend the algorithm to calculate not only the length of the shortest path but the path itself.

d[0] = 2
d[1] = 5
d[2] = 6
d[3] = 7
d[4] = 0
Extend algorithm to include the shortest path

Question: should we store in v itself the shortest path from v to every node? Or do we need another data structure to record these paths?

Not finished!
And how do we maintain it?

d[0] = 2
d[1] = 5
d[2] = 6
d[3] = 7
d[4] = 0
Extend algorithm to include the shortest path

For each node, maintain the backpointer on the shortest path to that node.

Shortest path to 0 is \( v \rightarrow 0 \). Node 0 backpointer is 4.

Shortest path to 1 is \( v \rightarrow 0 \rightarrow 1 \). Node 1 backpointer is 0.

Shortest path to 2 is \( v \rightarrow 0 \rightarrow 2 \). Node 2 backpointer is 0.

Shortest path to 3 is \( v \rightarrow 0 \rightarrow 2 \rightarrow 1 \). Node 3 backpointer is 2.

\[
\begin{align*}
\text{bk}[w] & \text{ is w’s backpointer} \\
\text{d}[0] &= 2 & \text{bk}[0] &= 4 \\
\text{d}[1] &= 5 & \text{bk}[1] &= 0 \\
\text{d}[2] &= 6 & \text{bk}[2] &= 0 \\
\text{d}[3] &= 7 & \text{bk}[3] &= 2 \\
\text{d}[4] &= 0 & \text{bk}[4] &\text{ (none)} \\
\end{align*}
\]
S = { }; F = {v}; d[v] = 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) {
            d[w] = d[f] + wgt(f, w);
            add w to F; bk[w] = f;
        } else if (d[f] + wgt(f, w) < d[w]) {
            d[w] = d[f] + wgt(f, w);
            bk[w] = f;
        }
    }
}

Maintain backpointers

Wow! It’s so easy to maintain backpointers!

When w not in S or F:
Getting first shortest path so far:

When w in S or F and have shorter path to w:
S = \{ \}; F = \{ v \}; d[v] = 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) {
            d[w] = d[f] + wgt(f, w);
            add w to F; bk[w] = f;
        } else if (d[f]+wgt (f,w) < d[w]) {
            d[w] = d[f] + wgt(f, w);
            bk[w] = f;
        }
    }
}

This is our final high-level algorithm. These issues and questions remain:
1. How do we implement F?
2. The nodes of the graph will be objects of class Node, not ints. How will we maintain the data in arrays d and bk?
3. How do we tell quickly whether w is in S or F?
4. How do we analyze execution time of the algorithm?
1. How do we implement F?

S = \{ \}; F = \{v\}; d[v] = 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) {
            d[w] = d[f] + wgt(f, w);
            add w to F; bk[w] = f;
        } else if (d[f] + wgt(f, w) < d[w]) {
            d[w] = d[f] + wgt(f, w);
            bk[w] = f;
        }
    }
}

23
S  F  Far off

S= {}; F= {v}; d[v]= 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) {
            d[w]= d[f] + wgt(f, w);
            add w to F; bk[w]= f;
        } else if (d[f]+wgt (f,w) < d[w]) {
            d[w]= d[f] + wgt(f, w);
            bk[w]= f;
        }
    }
}

For what nodes do we need a distance and a backpointer?
S  F

Far off

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

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

For what nodes do we need a distance and a backpointer?

For every node in S or F we need both its d-value and its backpointer (null for v)

Instead of arrays d and b, keep information associated with a node. Use what data structure for the two values?
Far off

S= \{ \}; F= \{v\}; d[v] = 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) {
            d[w] = d[f] + wgt(f, w);
            add w to F; bk[w] = f;
        } else if (d[f]+wgt (f,w) < d[w]) {
            d[w] = d[f] + wgt(f, w);
            bk[w] = f;
        }
    }
}

For what nodes do we need a distance and a backpointer?

For every node in S or F we need both its d-value and its backpointer (null for v)

public class SF {
    private int distance;
    private node backPtr;
    ...
}
F implemented as a heap of Nodes.
What data structure do we use to maintain an SF object for each node in S and F?

For every node in S or F we need both its d-value and its backpointer (null for v):

```java
public class SF {
    private int distance;
    private node backPtr;
    ...
}
```
For every node in S or F, we need an object of class SF. What data structure to use?

```
public class SF {
    private int distance;
    private node backPtr;
    ...
}
```

Implement this algorithm.
F: implemented as a min-heap.
S, d, b: replaced by data.

Algorithm to implement
Investigate execution time.
Important: understand algorithm well enough to easily determine the total number of times each part is executed/evaluated

Assume:
- n nodes reachable from v
- e edges leaving those n nodes

public class SFinfo {
    private int distance;
    private node backPtr;  …
}

HashMap<Node, SFinfo> data