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 FRIDAY. Late deadline still Sunday Working with a partner? Group before submitting!!

We will talk about prelim 2 (24 April) on Thursday.
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.dijkstra59.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.
- The NATO Software Engineering Conferences:

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.

$\text{wgt}(v_1, v_2)$ is the weight of the edge from node $v_1$ to $v_2$.

Some node $v$ be selected as the \textit{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 $w$.

\begin{align*}
  d[0] &= 2 \\
  d[1] &= 5 \\
  d[2] &= 6 \\
  d[3] &= 7 \\
  d[4] &= 0
\end{align*}
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
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 \( f \).

What does the theorem tell us about this frontier set?

(Cortland, 20 miles) \hspace{2em} (Dryden, 11 miles)  
(Enfield, 10 miles) \hspace{2em} (Tburg, 15 miles)

Answer: The shortest path from the start node to Enfield has length 10 miles.

Note: the following answer is incorrect because we haven’t said a word about the algorithm! We are just investigating properties of the invariant:

Enfield can be moved to the settled set.
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?
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 2:
When does loop stop? When is array d completely calculated?
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 3: Progress toward termination?

\[
\text{S= } \{ \} ; \text{F= } \{ v \} ; \text{d[v]}= 0; \\
\text{while ( F } \neq \{ \} ) \{ \\
f= \text{ node in F with min d value;} \\
\text{Remove f from F, add it to S;}
\]
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 $f$ {
    if ($w$ not in $S$ or $F$) {
      }
    } else {
      }
    }
}

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?
The algorithm

S

F

Far off

1. For s, d[s] is length of shortest v → s path.
2. For f, d[f] is length of shortest v → 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?
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 $\bullet \cdots \bullet \rightarrow f$

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 \{ \} )  
   
   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
      if (d[f] + wgt(f,w) < d[w]) {
         d[w]= d[f] + wgt(f, w);
      }
   }

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 \textit{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.

\( bk[w] \) is \( w \)'s backpointer
\[ d[0] = 2 \quad bk[0] = 4 \]
\[ d[1] = 5 \quad bk[1] = 0 \]
\[ d[2] = 6 \quad bk[2] = 0 \]
\[ d[3] = 7 \quad bk[3] = 2 \]
\[ d[4] = 0 \quad bk[4] \text{ (none)} \]
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;
        }
    }
}

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:

Wow! It’s so easy to maintain backpointers!
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?

Use a min-heap, with the priorities being the distances!

Distances ---priorities--- will change. That’s why we need updatePriority in Heap.java
For what nodes do we need a distance and a backpointer?

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

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 $F$) {
            $d[w] = d[f] + \text{wgt}(f, w)$;
            add $w$ to $F$; $bk[w] = f$;
        } 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?
Far off

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

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

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 ≠ \{\}) {
    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] + \text{wgt}(f, w);
            add w to F; bk[w] = f;
        } 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)

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

27
S  F  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;
        }
    }
}

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):

```
public class SF {
    private int distance;
    private node backPtr;
    ...
}
```
Given a node in S or F, we need to get its SF object quickly. What data structure to use?

Implement this algorithm.

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

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 ($w$ not in $S$ or $F$) {
            $d[w] = d[f] + \text{wgt}(f, w)$;
            add $w$ to $F$; $bk[w] = f$;
        } else if ($d[f] + \text{wgt}(f, w) < d[w]$) {
            $d[w] = d[f] + \text{wgt}(f, w)$;
            $bk[w] = f$;
        }
    }
}

Algorithm to implement

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

HashMap<Node, SF> data

$F$: implemented as a min-heap.

$S$, $d$, $b$ data: replaces $S$, $d$, $b$
S

F

Far off

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

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

\text{public class SFinfo} \ { 
\text{private int distance; \\
private node backPtr; 
}\}

\text{HashMap<} \text{Node, SFinfo}> \text{ data}

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

\text{Assume:} \\
\text{n nodes reachable from v} \\
\text{e edges leaving those n nodes}
S \quad F \quad \text{Far off}

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

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

\text{public class } SF \{
    \text{private int } distance;
    \text{private Node backptr;}
\}
S  F  Far off

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

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

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

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

\[ \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]) \ { \}

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

\[ \ bk[w] = f; \]

\[ \}

\} \]

\} \]

\} \]

\]HashMap<\text{Node, SF}> \text{ data} \]

Directed graph
n nodes reachable from v
e edges leaving the n nodes

Harder: In total, how many
times does the loop
for each neighbor w of f
find a neighbor and execute
the repetend?

public class SF {
    private int distance;
    private Node backPtr;
}

32
Directed graph
n nodes reachable from v
e edges leaving the n nodes

S  F  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;
    }
  }
}

Harder: In total, how many times does the loop
for each neighbor w of f
find a neighbor and execute the repetend?

Answer: The for-each statement
is executed ONCE for each node. During that execution, the repetend is executed once for each neighbor. In total then, the repetend is executed once for each neighbor of each node. A total of e times.
S        F        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;  
}  
}  
}  

Directed graph  
n nodes reachable from v  
e edges leaving the n nodes  
How many times does w not in S or F evaluate to true?  

**Answer:** If w is not in S or F, it is in the far-off set. When the main loop starts, n-1 nodes are in the far-off set. If w is in the far-off set, it is immediately put into w. **Answer:** n-1 times.
Directed graph
n nodes reachable from v
e edges leaving the n nodes

Far off

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

while (\( F \neq \{ \} \)) {
    \( f = \text{node in } F \text{ with min } d \text{ value; } \)
    Remove \( f \) from \( F \), add it to \( S \); \n
    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 (\( d[f] + wgt(f, w) < d[w] \)) {
            \( d[w] = d[f] + wgt(f, w); \)
            \( bk[w] = f; \)
        }
    }
}

How many times is the if-statement executed?

Answer: The repetend is executed \( e \) times. The if-condition in the repetend is true \( n-1 \) times. So the else-part is executed \( e-(n-1) \) times.

Answer: \( e+1-n \) times.
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; 
        }
    }
}

Directed graph 
n nodes reachable from v 
e edges leaving the n nodes

Answer: We don’t know. Varies. 
expected case: e+1-x times.
S = \{\} \; F = \{v\}; \; d[v] = 0; \\
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; \\
for \text{each neighbor } w \text{ of } f \\
    \text{if (} w \text{ not in } S \text{ or } F \text{)} \\
    \quad d[w] = d[f] + wgt(f, w); \quad n-1 \ \\
    \quad \text{add } w \text{ to } F; \; \text{bk}[w] = f; \quad n-1 \ \\
\text{else if (} d[f]+wgt(f,w) < d[w] \text{)} \\
\quad d[w] = d[f] + wgt(f, w); \quad e+1-n \ \\
\quad \text{bk}[w] = f; \quad e+1-n \ \\
\}}

Directed graph 
\( n \) nodes reachable from \( v \) 
\( e \) edges leaving the \( n \) nodes 

Expected-case analysis

We know how often each statement is executed. 
Multiply by its \( O(\ldots) \) time
\[ S = \{ \} ; F = \{ v \} ; \quad d[v] = 0; \]

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

\[ d[w] = d[f] + \text{wgt}(f, w); \quad n-1 \times O(n) \]

add \(w\) to \(F\); \(bk[w]\) = \(f\); 

\[ \text{e x O(e)} \]

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

\[ d[w] = d[f] + \text{wgt}(f, w); \quad e+1-n \times O((e-n) \log n) \]

\[ bk[w] = f; \quad e+1-n \times O(e-n) \]

} 

}\}

Directed graph

\(n\) nodes reachable from \(v\)

\(e\) edges leaving the \(n\) nodes

Expected-case analysis

We know how often each statement is executed. Multiply by its \(O(\ldots)\) time
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; 
        } 
    } 
}

Dense graph, so e close to n*n: Line 10 gives O(n^2 log n)

Sparse graph, so e close to n: Line 4 gives O(n log n)