A6. Implement shortest-path algorithm

One semester: mean time: 4.2 hrs, median time: 4.5 hrs.
max: 30 hours !!!!

We give you complete set of test cases and a GUI to play with.
Don’t wait until the last minute. It’s easy to make a mistake, and
you may not be able to get help to find it.

Efficiency and simplicity of code will be graded.

Read handout carefully:
2. Important! Grading guidelines.

We demo it.

---

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]

Dijkstra, E.W. A note on two problems in Connexion with graphs. Numerische

Visit http://www.dijkstrascry.com for all sorts of information on Dijkstra and
his contributions. As a historical record, this is a gold mine.

---

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:
  http://homepages.cs.ncl.ac.uk/~brian.randell/NATO/index.html
  Get a good sense of the times by reading these reports!
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 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 \( w \).

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

**Theorem about the invariant**

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

**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 \( f \).
The theorem tells us that the shortest \( v \rightarrow b \) path over all paths has length 2.
The theorem gives us no additional information about \( v \rightarrow c \) paths.

The loop invariant

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.
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)    (Dryden, 11 miles)
(Enfield, 10 miles)       (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.

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

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

Loopy question 3: Progress toward termination?

Loopy question 4: Maintain invariant?
The algorithm

\[ S = \{ \}; F = \{ v \}; d[v]= 0; \]
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;
    } else {
      if (d[f]+wgt(f,w) < d[w]) {
        d[w]= d[f] + wgt(f, w);
        bk[w]= f;
      }
    }
  }
}

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.

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?

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

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

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:

Extend algorithm to include the shortest path

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

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 info 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?
while 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[w]+wgt(f,w) < d[w]) {
        d[w]=d[f] + wgt(f,w); bk[w]= f;
    }
}

Far off

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

L. How do we implement F?

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

Distances ---priorities--- will change. That’s why we need changePriority in Heap.java

1. How do we implement F?

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[w]+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 backpointer.

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

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

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

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

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

F implemented as a heap of Nodes. What data structure to use to maintain a DB object for each node in S and F?

For every node in S or F we need to get its DB object quickly. What data structure to use?

HashMap<Node, DB> info

public class DB {
    private int dist;
    private node bkptr;
    …
}

Given a node in S or F, we need to get its DB object quickly. What data structure to use?

Implement this algorithm. F: implemented as a min-heap. info: replaces S, d, b

public class DB {
    private int dist;
    private node bkptr;
    …
}

Final abstract algorithm

public class DB {
    private int dist;
    private node bkptr;
    …
}
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;
        }
    }
}} HashMap<node, DB> info

for each neighbor w of f {
    if (w not in S or F) {
        if (w not in 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;
        }
    }
}} HashMap<node, DB> info

Directed graph n nodes reachable from v e edges leaving the n nodes
Public class DB {  
    private int dist;  
    private node bkptr;  
} ...

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

\[ S = \{ \} \; F = \{ v \}; \; d[v] = 0; \]
\[
\text{while } (F \neq \{ \}) \{
\quad \text{true x}
\quad f = \text{node in } F \text{ with min } d \text{ value;}
\quad \text{Remove } f \text{ from } F, \text{ add it to } S; \]
\[
\quad \text{for each neighbor } w \text{ of } f \{
\quad \quad \text{true x}
\quad \quad \text{if } (w \text{ not in } S \text{ or } F) \{
\quad \quad \quad \text{true x}
\quad \quad \quad d[w] = d[f] + \text{wgt}(f, w); \quad \text{n-1 x}
\quad \quad \quad \text{add } w \text{ to } F; \; \text{bk}[w] = f; \quad \text{n-1 x}
\quad \quad \} \quad \text{else if } (d[f] + \text{wgt}(f, w) < d[w]) \{
\quad \quad \quad d[w] = d[f] + \text{wgt}(f, w); \quad \text{e+1-n x}
\quad \quad \quad \text{bk}[w] = f; \quad \text{e+1-n x}
\quad \quad \}
\quad \}
\}
\]

\text{We don’t know. Varies.}
\text{expected case: } e+1-x \text{ times.}

\text{Answer: We don’t know. Varies.}
\text{expected case: } e+1-x \text{ times.}

\[ S = \{ \} \; F = \{ v \}; \; d[v] = 0; \]
\[
\text{while } (F \neq \{ \}) \{
\quad \text{true x}
\quad f = \text{node in } F \text{ with min } d \text{ value;}
\quad \text{Remove } f \text{ from } F, \text{ add it to } S; \]
\[
\quad \text{for each neighbor } w \text{ of } f \{
\quad \quad \text{true x}
\quad \quad \text{if } (w \text{ not in } S \text{ or } F) \{
\quad \quad \quad \text{true x}
\quad \quad \quad d[w] = d[f] + \text{wgt}(f, w); \quad \text{n-1 x}
\quad \quad \quad \text{add } w \text{ to } F; \; \text{bk}[w] = f; \quad \text{n-1 x}
\quad \quad \} \quad \text{else if } (d[f] + \text{wgt}(f, w) < d[w]) \{
\quad \quad \quad d[w] = d[f] + \text{wgt}(f, w); \quad \text{e+1-n x}
\quad \quad \quad \text{bk}[w] = f; \quad \text{e+1-n x}
\quad \quad \}
\quad \}
\}
\]

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

\text{Dense graph, so } e \text{ close to } n^2: \text{ Line 10 gives } O(n^2 \log n)
\text{Sparse graph, so } e \text{ close to } n: \text{ Line 4 gives } O(n \log n)