A4 and A5 grades

A4 grades released. Read the feedback.
Mean time: 6.9 hours
Median time: 6.0 hours
Assignment A6 Piazza note contains a file with comments extracted from your submissions.

A5 grades released early tomorrow morning but will contain only the grade for correctness. The grade may be reduced during this week (until Sunday) as graders check over your solution.

**Reason for this process:** If you got 100, you can use your A5 in A6; otherwise, use our solution—it will be made available tomorrow.

So far, 453/489 students got 100. Late ones not graded yet.

---

A6. Implement shortest-path algorithm

Last semester: mean time: 3.7 hrs, median time: 3.0 hrs.
max: 30 hours !!!!

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

**Read pinned note Assignment A6 note carefully:**

2. Important! Grading guidelines.

We demo it.

We will talk about prelim 2 (15 November) 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.dijkstrascry.com](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:
  Get a good sense of the times by reading these reports!

---

**1968 NATO Conference on Software Engineering, Garmisch, Germany**

---

**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 $w$.

- $d[0] = 2$
- $d[1] = 5$
- $d[2] = 6$
- $d[3] = 7$
- $d[4] = 0$

Edsger Dijkstra, Niklaus Wirth, Tony Hoare, David Gries

---

**1968/69 NATO Conferences on Software Engineering**

- 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

---

**Settled S Frontier F Far off**

- 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.
For a Settled node $s$, $d[s]$ is length of shortest $v \rightarrow s$ path.

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

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

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

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 {
        }
    }
}

Loopy question 4: Maintain invariant?

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

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 {
        }
    }
}

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.

For each node, maintain the backpointer on the shortest path to that node.
Shortest path to 0 is v -> 0. Node 0 backpointer is 4.
Shortest path to 1 is v -> 0 -> 1. Node 1 backpointer is 0.
Shortest path to 2 is v -> 0 -> 2. Node 2 backpointer is 0.
Shortest path to 3 is v -> 0 -> 2 -> 3. Node 3 backpointer is 2.

bk[w] is w’s backpointer

d[0] = 2  bk[0] = 4

d[1] = 5  bk[1] = 0


Wow! It’s so easy to maintain backpointers!

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

Maintain backpointers

When w in S or F and have shorter path to w:
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 b(k)
3. How do we tell quickly whether w is in S or F?
4. How do we analyze execution time of the algorithm?

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)

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

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

```
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;
    }
  }
}
```

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

```
private Node backPtr;
```

Final abstract algorithm

```
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, DistBack> info

```
public class DistBack {
  private int distance;
  private Node backPtr;
  ...
}
```

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.

```
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, DistBack> info

```
public class DistBack {
  private int distance;
  private Node backPtr;
  ...
}
```

Assume: n nodes reachable from v e edges leaving the n nodes

```
for each neighbor w of f {
  (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, DistBack> info

```
public class DistBack {
  private int distance;
  private Node backPtr;
  ...
}
```

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.
while

S

F

Far off

S= { }; F= {v}; d[v]=0; 1 x

while (F ≠ {}) { true n x

f= node in F with min d value; n x

Remove f from F, add it to S; n x

for each neighbor w of f { true n x

if (w not in S or F) { ex n x
d[w]= d[f] + wgt(f,w); n-l x

add w to F; bk[w]= f; n-l x
}

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

d[w]= d[f] + wgt(f,w); e+1-n x

bk[w]= f;
}

}

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

S

F

Far off

S= { }; F= {v}; d[v]=0; 1 x

while (F ≠ {}) { true n x

f= node in F with min d value; n x

Remove f from F, add it to S; n x

for each neighbor w of f { true n x

if (w not in S or F) { ex n x
d[w]= d[f] + wgt(f,w); n-l x

add w to F; bk[w]= f; n-l x
}

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

d[w]= d[f] + wgt(f,w); e+1-n x

bk[w]= f;
}

}

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

S

F

Far off

S= { }; F= {v}; d[v]=0; 1 x O(1)

while (F ≠ {}) { true n x O(n)

f= node in F with min d value; n x O(n)

Remove f from F, add it to S; n x O(n log n)

for each neighbor w of f { true n x O(e)

if (w not in S or F) { ex O(e)
d[w]= d[f] + wgt(f,w); n-l x O(n)

add w to F; bk[w]= f; n-l x O(n log n)
}

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

d[w]= d[f] + wgt(f,w); e+1-n x O(e-n)

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)

Directed graph

n nodes reachable from v e edges leaving the n nodes

Expected-case analysis

Answer: We don’t know. Varies. Expected case: e+1-n x times.