1

SHORTEST PATHS

READINGS? CHAPTER 28

Lecture 21
CS2110 --Spring 2017

A7. Implement shortest-path algorithm

2

Not due until after the prelim, but it's in your best interest to do it before because it's on the prelim.

Last semester: Average time was 3.3 hours.

This semester, because of the tutorial and today's lecture, you will be better prepared for the assignment.

We give you complete set of test cases and a GUI to play with.

Efficiency and simplicity of code will be graded.

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 Mathematik* 1, 269–271 (1959).

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

3

Dijkstra's shortest-path algorithm

Dijsktra 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."

Dijkstra, E.W. A note on two problems in Connexion with graphs. *Numerische Mathematik* 1, 269–271 (1959).

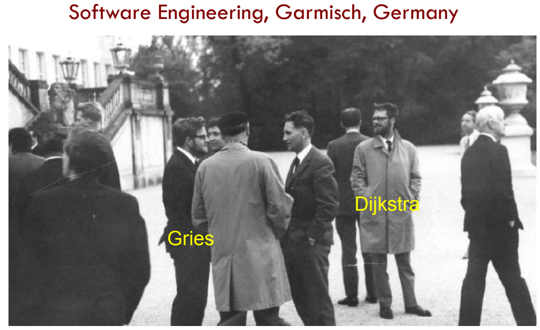
4

1968 NATO Conference on Software Engineering

5

- 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!

1968 NATO Conference on Software Engineering, Garmisch, Germany

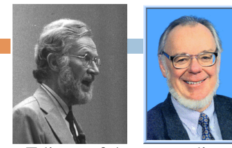


Term "software engineering" coined for this conference

6

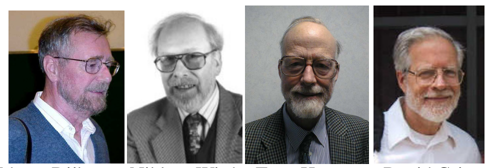
1968 NATO Conference on Software Engineering, Garmisch, Germany

1968/69 NATO Conferences on Software Engineering



Editors of the proceedings

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



Edsger Dijkstra Niklaus Wirth Tony Hoare David Gries

8

Dijkstra's shortest path algorithm

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

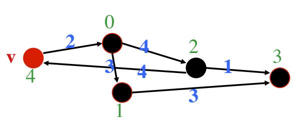
Each edge has a positive weight.

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.

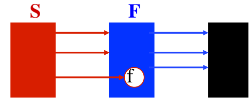


d[0] = 2
d[1] = 5
d[2] = 6
d[3] = 7
d[4] = 0

9

The loop invariant

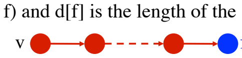
Settled S **Frontier F** **Far off**



(edges leaving the Far off set and edges from the Frontier to the Settled set are not shown)

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 → s path.

2. For a Frontier node f, at least one v → f path contains only settled nodes (except perhaps for f) and d[f] is the length of the shortest such path

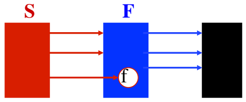


3. All edges leaving S go to F.

10

The loop invariant

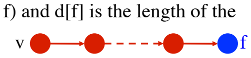
Settled S **Frontier F** **Far off**



(edges leaving the Far off set and edges from the Frontier to the Settled set are not shown)

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 → s path.

2. For a Frontier node f, at least one v → f path contains only settled nodes (except perhaps for f) and d[f] is the length of the shortest such path

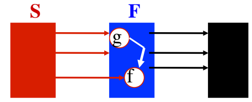


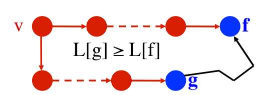
3. All edges leaving S go to F.

11

Theorem about the invariant

Settled S **Frontier F** **Far off**





1. For a Settled node s, d[s] is length of shortest v → s path.

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

3. All edges leaving S go to F.

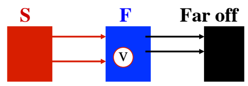
Theorem. For a node f in F with minimum d value (over nodes in F), d[f] is the length of a shortest path from v to f.

Case 1: v is in S.

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

12

The algorithm



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**.

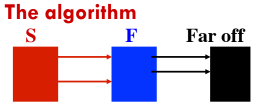
Theorem: For a node **f** in **F** with min d value, d[f] is shortest path length

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

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

13

The algorithm



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**.

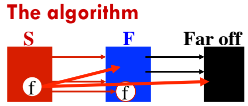
Theorem: For a node **f** in **F** with min d value, d[f] is shortest path length

```
S= { }; F= { v }; d[v]= 0;
while  F ≠ {}  {

}
```

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

14

The algorithm



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**.

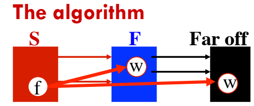
Theorem: For a node **f** in **F** with min d value, d[f] is shortest path length

```
S= { }; F= { v }; d[v]= 0;
while  F ≠ {} {
    f= node in F with min d value;
    Remove f from F, add it to S;

}
```

Loopy question 3: Progress toward termination?

15

The algorithm



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**.

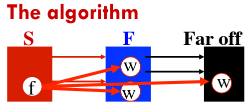
Theorem: For a node **f** in **F** with min d value, d[f] is shortest path length

```
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) {

        } else {

        }
    }
}
```

Loopy question 4: Maintain invariant?

16

The algorithm



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**.

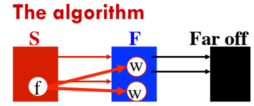
Theorem: For a node **f** in **F** with min d value, d[f] is shortest path length

```
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 {

        }
    }
}
```

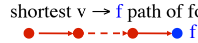
Loopy question 4: Maintain invariant?

17

The algorithm



1. **For s, d[s]** is length of shortest v → s path.
2. **For f, d[f]** is length of shortest v → 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

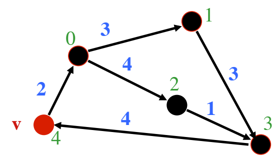
```
S= { }; F= { v }; d[v]= 0;
while  F ≠ {} {
    f= node in F with min d value;
    Remove f from F, add it to S;
    for each neighbor w of f {
        if (w not in S or F) {
            d[w]=  d[f] + wgt(f, w);
            add w to F;
        } else
            if (d[f] + wgt (f,w) < d[w]) {
                d[w]= d[f] + wgt(f, w);
            }
    }
}
```

Algorithm is finished!

18

Extend algorithm to include the shortest path

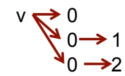
Let's extend the algorithm to calculate not only the length of the shortest path but the path itself.



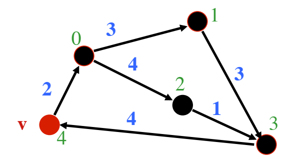
d[0] = 2
d[1] = 5
d[2] = 6
d[3] = 7
d[4] = 0

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?

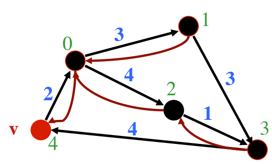


d[0] = 2
d[1] = 5
d[2] = 6
d[3] = 7
d[4] = 0

Extend algorithm to include the shortest path

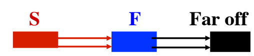
For each node, maintain the *backpointer* on the shortest path to that node.

Shortest path to 0 is v -> 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 -> 1. Node 3 backpointer is 2.



bk[w] is w's backpointer

d[0] = 2	bk[0] = 4
d[1] = 5	bk[1] = 0
d[2] = 6	bk[2] = 0
d[3] = 7	bk[3] = 2
d[4] = 0	bk[4] (none)

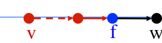


Maintain backpointers

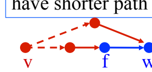
```
S= { }; F= {v};  d[v]= 0;
while  (F ≠ {}) {
   f= node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
          d[w]=  d[f] + wgt(f, w);
          add w to F;  bk[w]=  f;
      } else if (d[f] + wgt (f,w) < d[w]) {
              d[w]= d[f] + wgt(f, w);
              bk[w]=  f;
          }
  }}
```

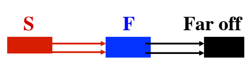
Wow! It's so easy to maintain backpointers!

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:

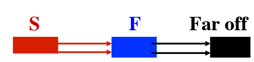





```
S= { }; F= {v};  d[v]= 0;
while  (F ≠ {}) {
   f= node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
          d[w]=  d[f] + wgt(f, w);
          add w to F; bk[w]=  f;
      } else if (d[f]+wgt (f,w) < d[w]) {
          d[w]= d[f] + 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?




```
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;
      }
}}
```

1. How do we implement F?

Implement F as a min-heap of Nodes!
The priority of a node in F is its d-value —the shortest distance known from v to that node.

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?

25

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?

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

Don't want to use arrays d and bk! Instead, keep information associated with a node. What data structure to use for the two values?

26

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?

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

```
public class SFdata {
    private node backPointer;
    private int distance;
    …
}
```

27

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;
    }
}}
```

F implemented as a heap of Nodes.
What data structure do we use to maintain an SFdata 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 SFdata {
    private node backPointer;
    private int distance;
    …
}
```

28

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 every node in S or F, we need an object of class SFdata. What data structure to use?

HashMap<Node, SFdata> map

You will implement the algorithm on this slide. S, d, and b are replaced by map. F is implemented as a min-heap.

```
public class SFdata {
    private node backPointer;
    private int distance;
    …
}
```

Algorithm to implement

29

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;
    }
}}
```

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

HashMap<Node, SFdata> map

```
public class SFdata {
    private node backPointer;
    private int distance; … }
```

30

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;
    }
}}
```

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

Example. How many times does F ≠ {} evaluate to true?

Yes! n times. v is in F initially. Every node is put into F once and taken out once.

HashMap<Node, SFdata> map

```
public class SFdata {
  private node backPointer;
  private int distance; … }
```

31

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;
    }
}}
```

Directed graph
n nodes reachable from v
e edges leaving those n nodes
F ≠ {} is true n times

Harder: In total, how many times does the loop

for each neighbor w of f

find a neighbor and execute the if-statement?

HashMap<Node, SFdata> map

```
public class SFdata {
  private node backPointer;
  private int distance; … }
```

32

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;
    }
}}
```

Directed graph
n nodes reachable from v
e edges leaving those n nodes
F ≠ {} is true n times
First if-statement: done e times

How many times does

w not in S or F

evaluate to true?

HashMap<Node, SFdata> map

```
public class SFdata {
  private node backPointer;
  private int distance; … }
```

33

S F Far off

Number of times (x) each part is executed/evaluated

S= { }; F= {v}; d[v]= 0;	1 x
while (F ≠ {}) {	true n x, false 1 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 e x, false n x
if (w not in S or F) {	done e x, true n-1 x, false e-(n-1) x
d[w]= d[f] + wgt(f, w);	n-1 x
add w to F; bk[w]= f;	n-1 x
} else if (d[f]+wgt (f,w) < d[w]) {	done e-(n-1) x, true ?
d[w]= d[f] + wgt(f, w);	done at most e-(n-1) x
bk[w]= f;	done at most e-(n-1) x
}	
}}	

Assume: directed graph, using adjacency list
n nodes reachable from v
e edges leaving those n nodes

34

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;
    }
}}
```

To find an upper bound on time complexity, multiply complexity of each part by the number of times its executed.

Then add them up.

Assume: directed graph, using adjacency list
n nodes reachable from v
e edges leaving those n nodes

35

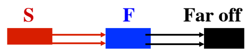
S F Far off

Expected time

	Expected time
S= { }; F= {v}; d[v]= 0;	1 * O(1)
while (F ≠ {}) {	(n+1) * O(1)
f= node in F with min d value;	n * O(1)
Remove f from F, add it to S;	n * (O(log n) + O(1))
for each neighbor w of f {	(e+n) * O(1)
if (w not in S or F) {	e * O(1)
d[w]= d[f] + wgt(f, w);	(n-1) * O(1)
add w to F; bk[w]= f;	(n-1) * O(log n)
} else if (d[f]+wgt (f,w) < d[w]) {	(e-(n-1)) * O(1)
d[w]= d[f] + wgt(f, w);	(e-(n-1)) * O(log n)
bk[w]= f;	(e-(n-1)) * O(1)
}	
}}	

Assume: directed graph, using adjacency list
n nodes reachable from v
e edges leaving those n nodes

36



Code	Expected time	
S= { }; F= {v}; d[v]= 0;	1 * O(1)	1
while (F ≠ {}) {	(n+1) * O(1)	2
f= node in F with min d value;	n * O(1)	3
Remove f from F, add it to S;	n * (O(log n) + O(1))	4
for each neighbor w of f {	(e+n) * O(1)	5
if (w not in S or F) {	e * O(1)	6
d[w]= d[f] + wgt(f, w);	(n-1) * O(1)	7
add w to F; bk[w]= f;	(n-1) * O(log n)	8
} **else if** (d[f]+wgt (f,w) < d[w]) {	(e-(n-1)) * O(1)	9
d[w]= d[f] + wgt(f, w);	(e-(n-1)) * O(log n)	10
bk[w]= f;	(e-(n-1)) * O(1)	11
}		
}}		

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)

37