SHORTEST PATH ALGORITHM

Lecture 19

CS2110. Fall 2018

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.

Tomorrow is Halloween (Hallowed Eve)

1	1	13	15
2	2	14	16
3	3	15	17
4	4	*16*	*20*
5	5	17	21
6	6	18	22
7	7	19	23
8	10	20	24
9	11	21	25
10	12	22	26
11	13	23	27
12	14	24	30
		25	31

Last year, why did I get a Christmas card on Halloween?

Because Dec 25 is Oct 31

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.

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).

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!

1968 NATO Conference on Software Engineering, Garmisch, Germany

Term “software engineering” coined for this conference

1968 NATO Conference on
Software Engineering, Garmisch, Germany

1968/69 NATO Conferences on Software Engineering


Editors of the proceedings

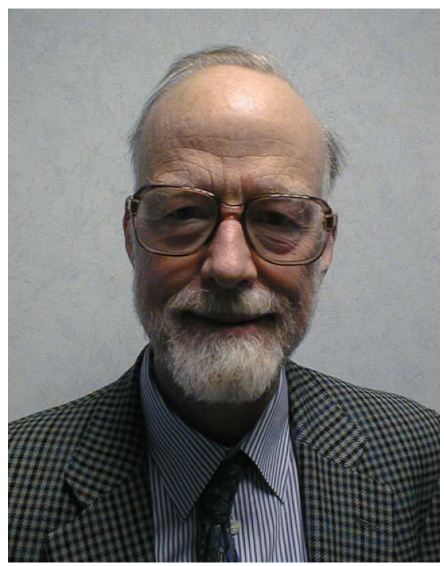
Beards

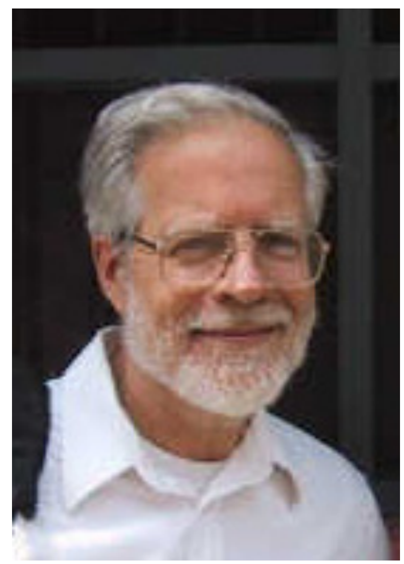
The reason why some people grow
aggressive tufts of facial hair
Is that they do not like to show
the chin that isn't there.

a grook by Piet Hein


Edsger Dijkstra


Niklaus Wirth


Tony Hoare


David Gries

Dijkstra's shortest path algorithm

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

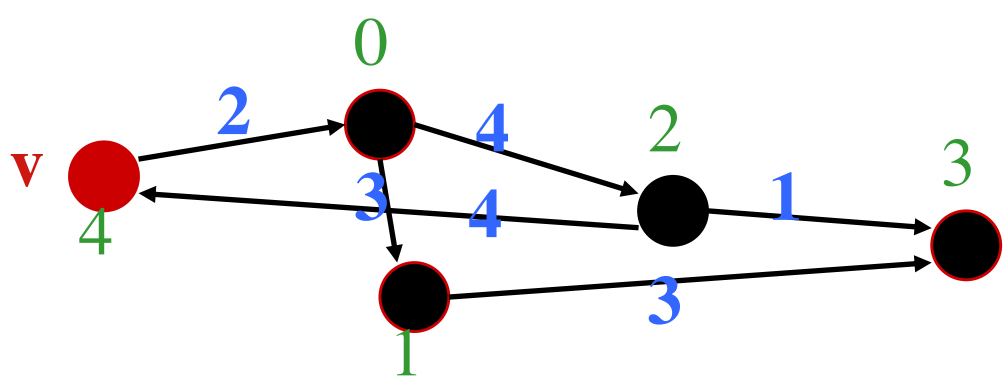
Each edge has a positive weight.

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

Some node v be selected as the *start* node.

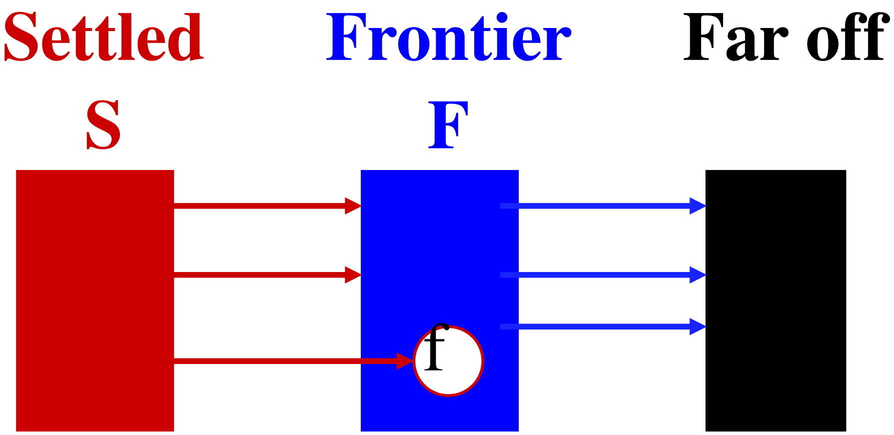
Calculate length of shortest path from v to each node.

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

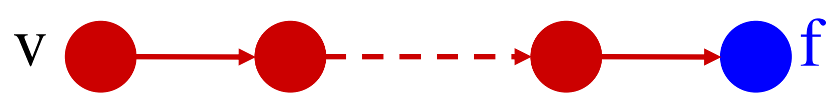
The loop invariant



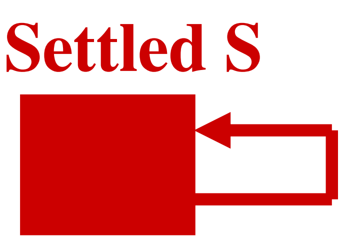
(edges leaving the Far off set and edges from the Frontier to the Settled set are not shown)

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.

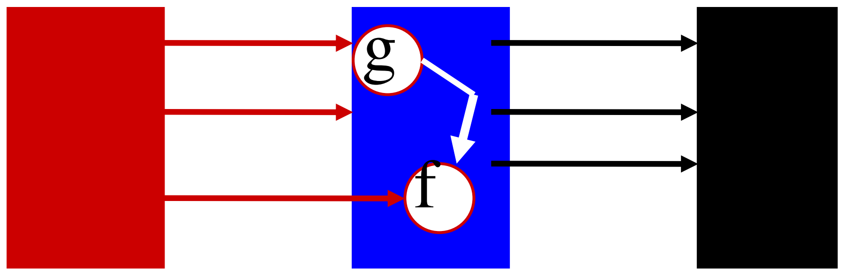


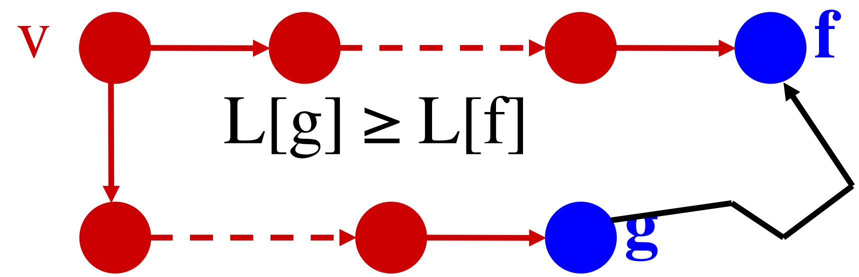
This edge does not leave S!

Another way of saying 3: There are no edges from S to the far-off set.

Settled **Frontier** **Far off** **Theorem about the invariant**

S **F**



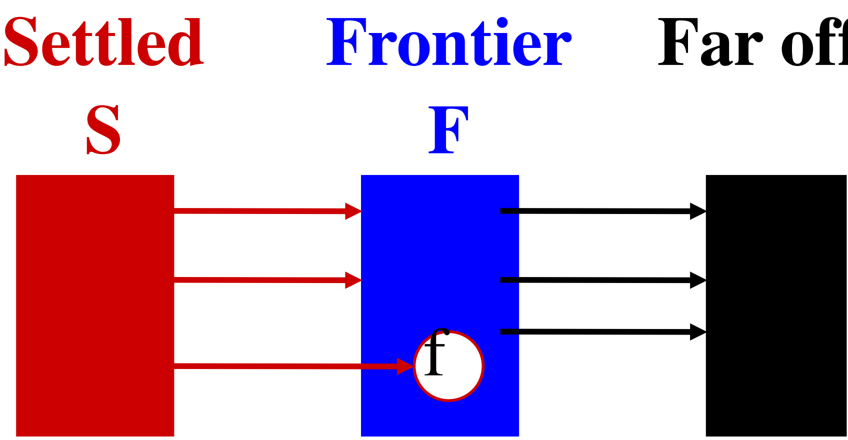


1. For a Settled node s, d[s] is length of shortest v → s path.
2. For a Frontier node f, d[f] is length of shortest v → f path using only Settled nodes (except for f).
3. All edges leaving S go to F.

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



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?

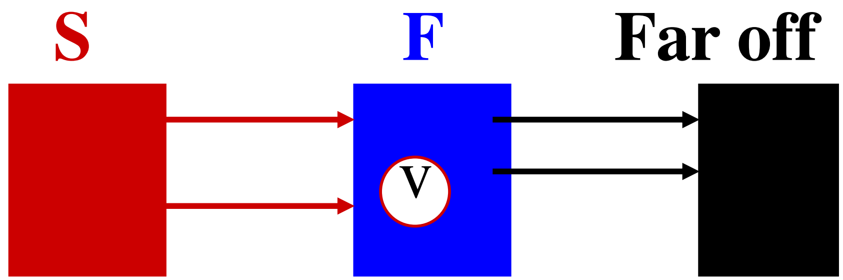
(Cortland, 20 miles) **(Dryden, 11 miles)**
(Enfield, 10 miles) **(Tburg, 15 miles)**

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.

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**.

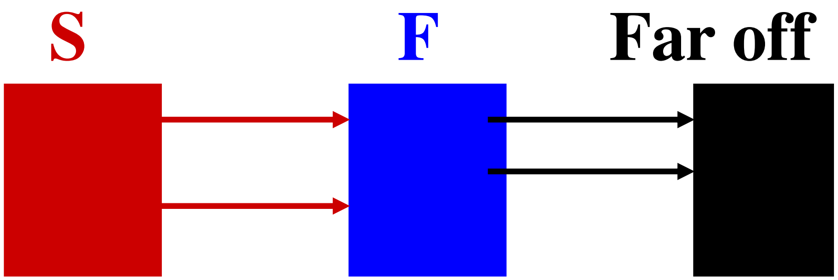
Theorem: For a node **f** in **F** with min d value, d[f] is shortest path length

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

Loopy question 1:

How does the loop start? What is done to truthify the invariant?

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.**

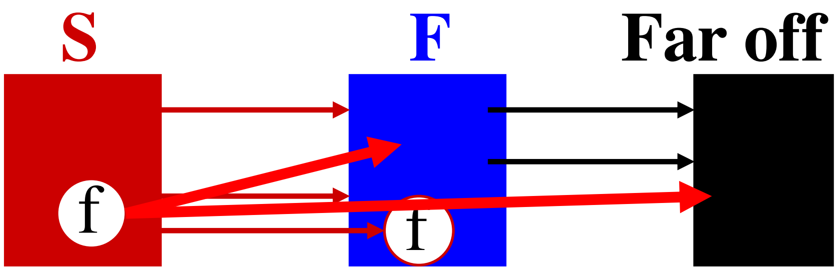
Theorem: For a node **f** in **F** with min d value, d[f] is shortest path length

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

}

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

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

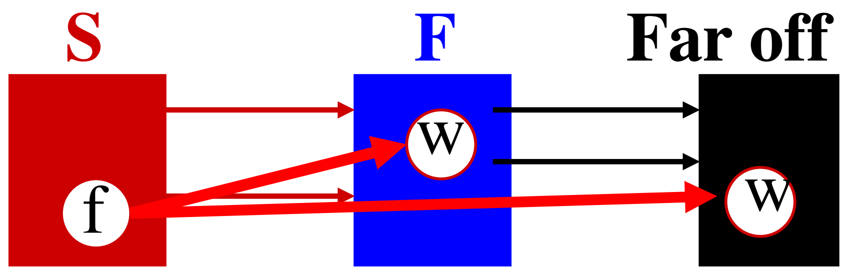
S= { }; F= { v }; d[v]= 0;
while (F ≠ {}) {
    f= node in F with min d value;
    Remove f from F, add it to S;

Note: this is the earliest time that the shortest distance to this f is known!

}

Loopy question 3: Progress toward termination?

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

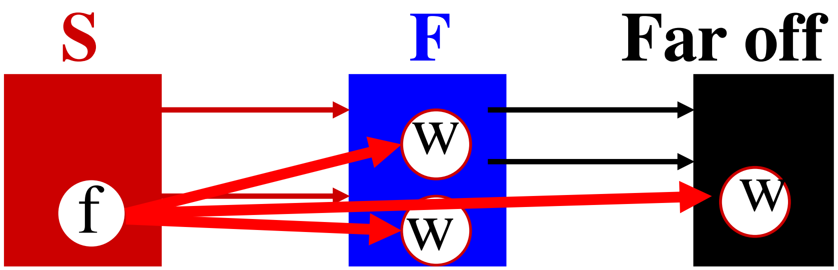
```
S= { }; F= { v }; d[v]= 0;
while ( F ≠ {} ) {
    f= node in F with min d value;
    Remove f from F, add it to S;
    for each neighbor w of f {
        if (w not in S or F) {

        } else {

        }
    }
}
```

Loopy question 4: Maintain invariant?

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**.

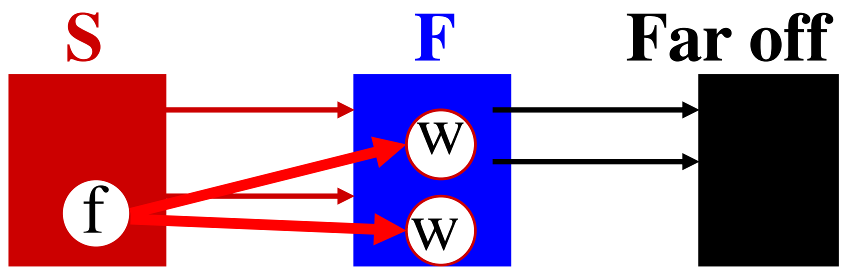
Theorem: For a node **f** in **F** with min d value, d[f] is shortest path length

```
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?

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.

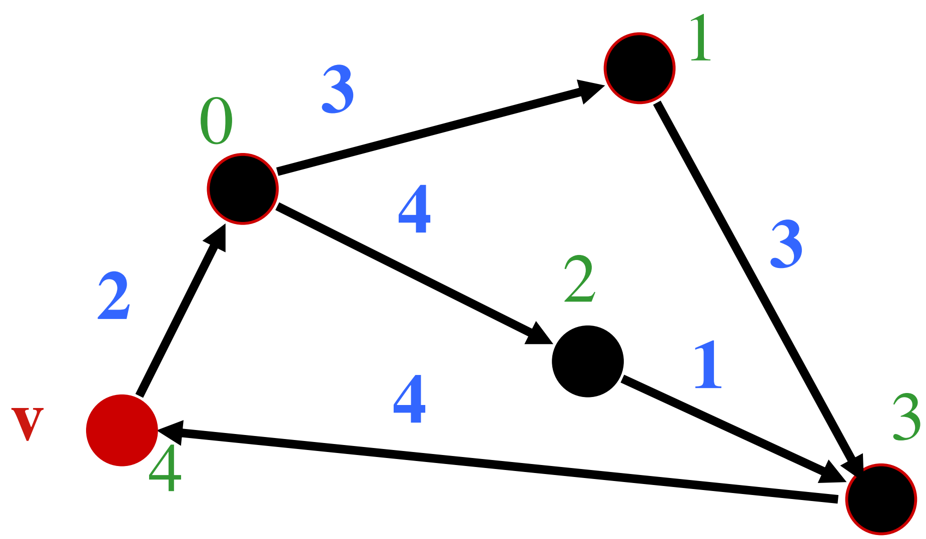
Theorem: For a node **f** in **F** with min d value, d[f] is its shortest path length

```
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!

Extend algorithm to include the shortest path

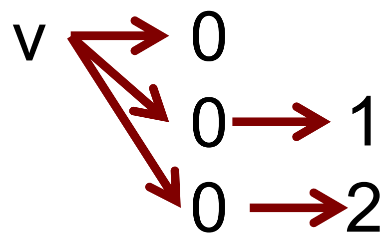
Let's extend the algorithm to calculate not only the length of the shortest path but the path itself.



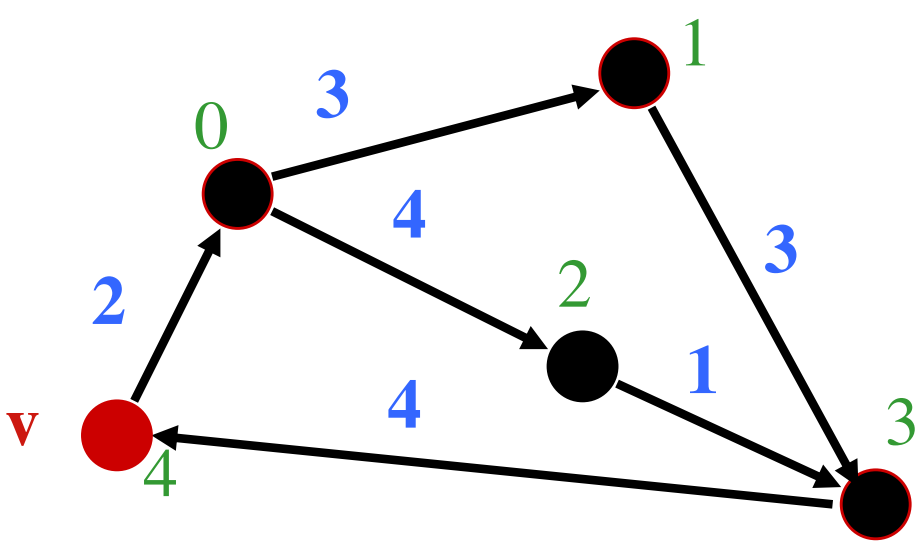
d[0] = 2
d[1] = 5
d[2] = 6
d[3] = 7
d[4] = 0

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

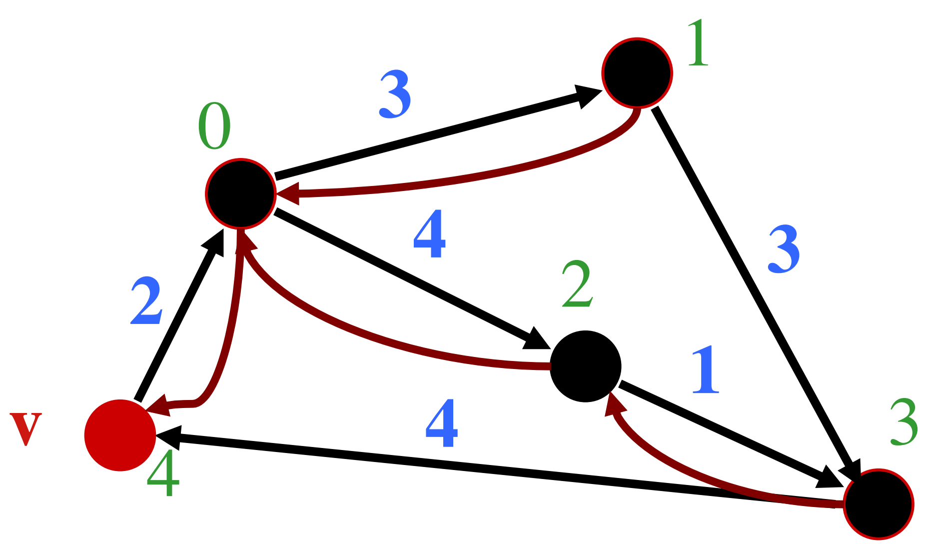
For each node, maintain the *backpointer* on the shortest path to that node.

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.

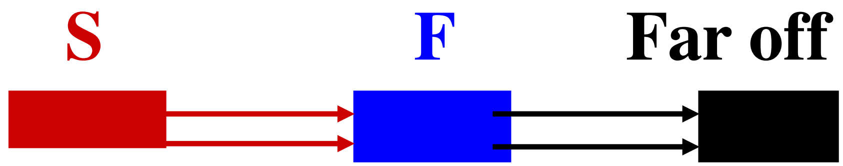
Shortest path to 2 is v -> 0 -> 2. Node 2 backpointer is 0.

Shortest path to 3 is v -> 0 -> 2 -> 3. 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)

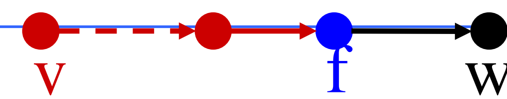



```
S= { }; F= {v}; d[v]= 0;
while (F ≠ {}) {
  f= node in F with min d value;
  Remove f from F, add it to S;
  for each neighbor w of f {
    if (w not in S or F) {
       d[w]= d[f] + wgt(f, w);
       add w to F;  bk[w]= f;
    } else if (d[f] + wgt (f,w) < d[w]) {
          d[w]= d[f] + wgt(f, w);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

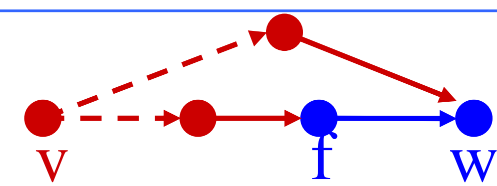
Maintain backpointers

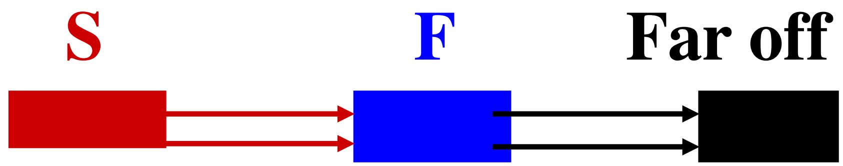
Wow! It's so easy to maintain backpointers!

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:

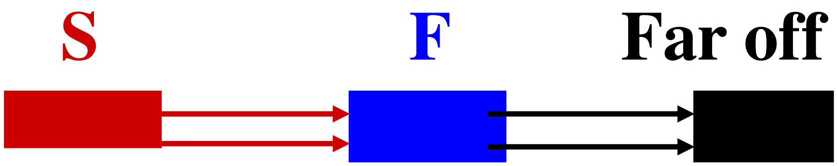





```
S= { }; F= {v}; d[v]= 0;
while (F ≠ {}) {
   f= node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         d[w]= d[f] + wgt(f, w);
         add w to F; bk[w]= f;
      } else if (d[f]+wgt (f,w) < d[w]) {
         d[w]= d[f] + wgt(f,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

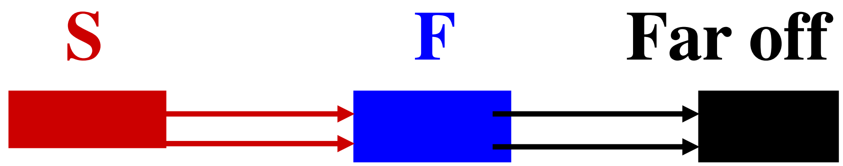


1. How do we implement F?

```
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;
     }
}}
```

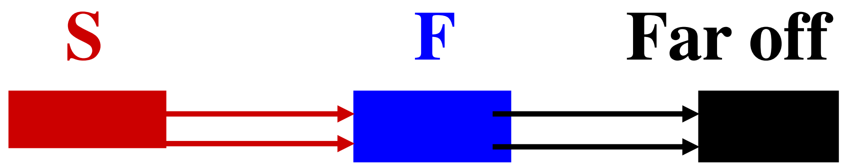
Use a min-heap, with the priorities being the distances!

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



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

```
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;
      }
}}
```

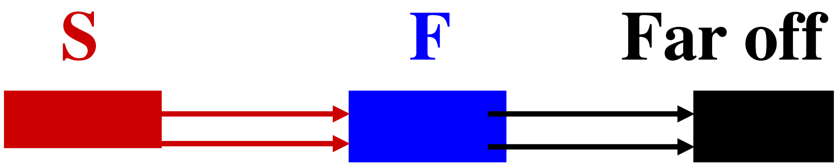



```
S= { }; F= {v}; d[v]= 0;
while (F ≠ {}) {
   f= node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         d[w]= d[f] + wgt(f, w);
         add w to F; bk[w]= f;
      } else if (d[f]+wgt (f,w) < d[w]) {
         d[w]= d[f] + wgt(f,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

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

For every node in S and every node in 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?

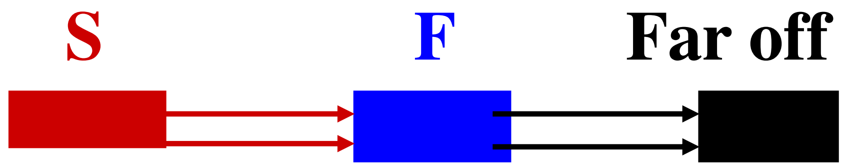



```
S= { }; F= {v}; d[v]= 0;
while (F ≠ {}) {
   f= node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         d[w]= d[f] + wgt(f, w);
         add w to F; bk[w]= f;
      } else if (d[f]+wgt (f,w) < d[w]) {
         d[w]= d[f] + wgt(f,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

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

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

```
public class DistBack {
    private int distance;
    private node backPtr;
    …
}
```

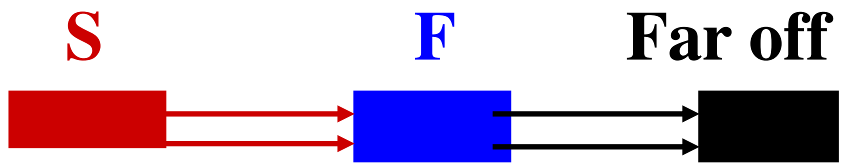



```
S= { }; F= {v}; d[v]= 0;
while (F ≠ {}) {
   f= node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
           d[w]= d[f] + wgt(f, w);
           add w to F; bk[w]= f;
      } else if (d[f]+wgt (f,w) < d[w]) {
           d[w]= d[f]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

F implemented as a heap of Nodes.
What data structure to use to maintain a DistBack 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 DistBack {
      private int distance;
      private node backPtr;
    …
  }
```




```
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;
    }
}}
```

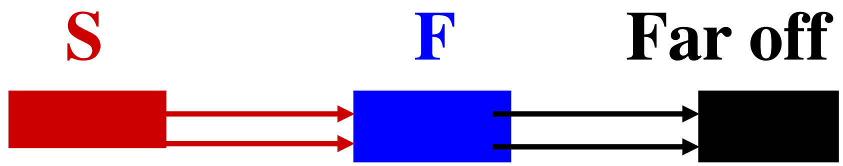
Given a node in S or F, we need to gets its DistBack object quickly.
What data structure to use?

HashMap<Node, DistBack > info

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

```
public class DistBack {
    private int distance;
    private node backPtr;
    …
}
```

Final abstract algorithm



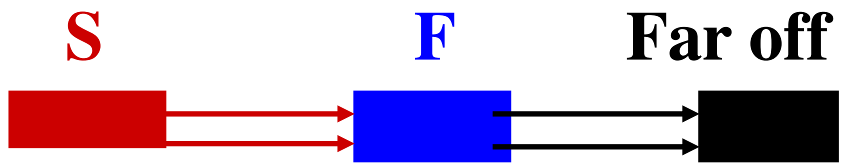

```
S= { }; F= {v}; d[v]= 0;
while (F ≠ {}) {
   f= node in F with min d value;
   Remove f from F, add it to S;
   for each neighbor w of f {
      if (w not in S or F) {
         d[w]= d[f] + wgt(f, w);
         add w to F; bk[w]= f;
      } else if (d[f]+wgt (f,w) < d[w]) {
         d[w]= d[f] + wgt(f,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

```
HashMap<Node, DistBack> info
```

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 DistBack {
   private int distance;
   private node backPtr;
}
```



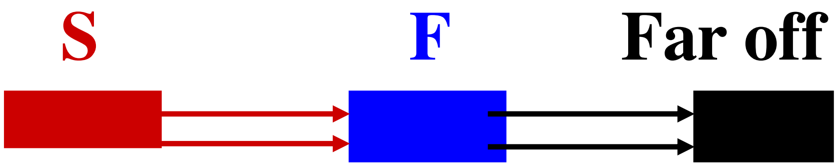

```
S= { }; F= {v}; d[v]= 0;                          1 x
while (F ≠ {}) {                                  true n x
   f= node in F with min d value;                 n x
   Remove f from F, add it to S;                  n x
   for each neighbor w of f {
      if (w not in S or F) {
         d[w]= d[f] + 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 the n nodes

Question. How many times does F ≠ {} evaluate to true?
To false?

```
HashMap<Node, DistBack> info
```

```
public class DistBack {
    private int distance;
    private Node backptr;
}
```




```
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 {
      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

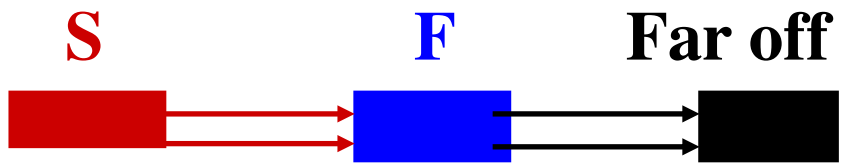
Harder: In total, how many times does the loop

for each neighbor w of f

find a neighbor and execute the repetend?

HashMap<Node, DistBack> info

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




```
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 {
      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;
      }
}}
```

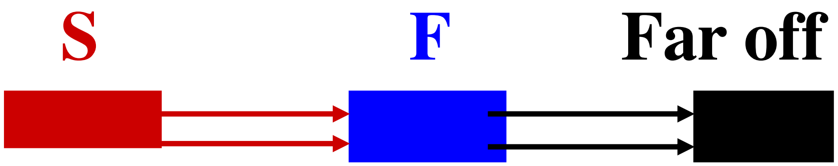
Directed graph
n nodes reachable from v
e edges leaving the n nodes

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.



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

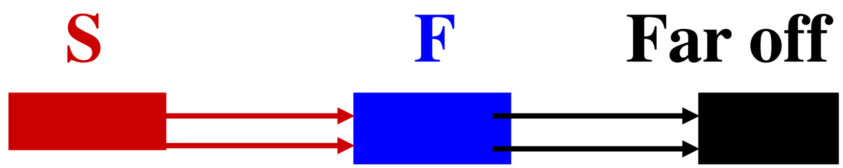
```
S= { }; F= {v}; d[v]= 0;                    1 x
while (F ≠ {}) {                           true n x
   f= node in F with min d value;            n x
   Remove f from F, add it to S;             n x
   for each neighbor w of f {              true e x
      if (w not in S or F) {                 e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

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.

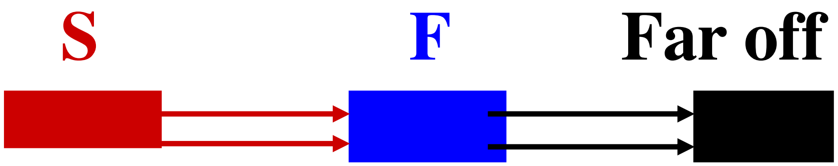


Directed graph
n nodes reachable from v
e edges leaving the n nodes

```
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 e x
      if (w not in S or F) {                e 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]) {
         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.

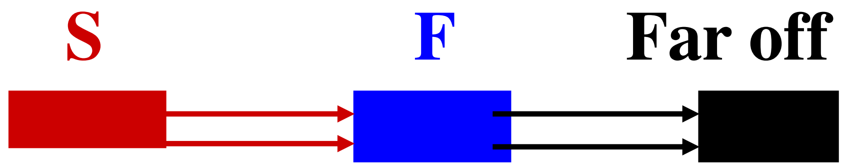


Directed graph
n nodes reachable from v
e edges leaving the n nodes

```
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 e x
      if (w not in S or F) {                      e 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]) {         e+1-n x
         d[w]= d[f] + wgt(f, w);
         bk[w]= f;
      }
}}
```

How many times is the if-condition true and d[w] changed?

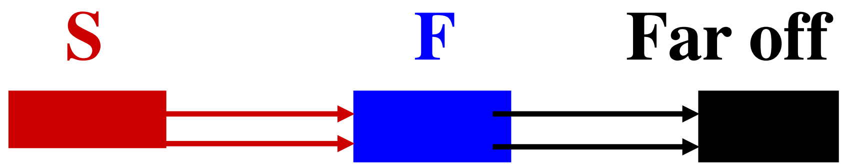
Answer: We don't know. Varies.
expected case: e+1-x times.




```
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 e x
      if (w not in S or F) {                        e 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]) {           e+1-n x
         d[w]= d[f] + wgt(f, w);                    e+1-n x
         bk[w]= f;                                  e+1-n x
      }
}}
```

Directed graph
n nodes reachable from v
e edges leaving the n nodes
Expected-case analysis

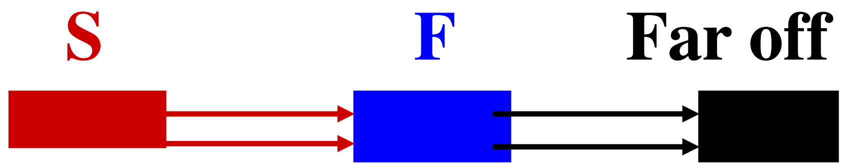
We know how often each statement is executed. Multiply by its O(…) time




```
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 e x  O(e)
      if (w not in S or F) {                      e x       O(e)
         d[w]= d[f] + wgt(f, w);                  n-1 x     O(n)
         add w to F; bk[w]= f;                    n-1 x     O(n log n)
      } else if (d[f]+wgt (f,w) < d[w]) {         e+1-n x   O(e–n)
         d[w]= d[f] + wgt(f, w);                  e+1-n x   O((e–n) log n)
         bk[w]= f;                                e+1-n x   O(e–n)
      }
}}
```

Directed graph
n nodes reach-able from v
e edges leaving the n nodes
Expected-case analysis

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




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

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)