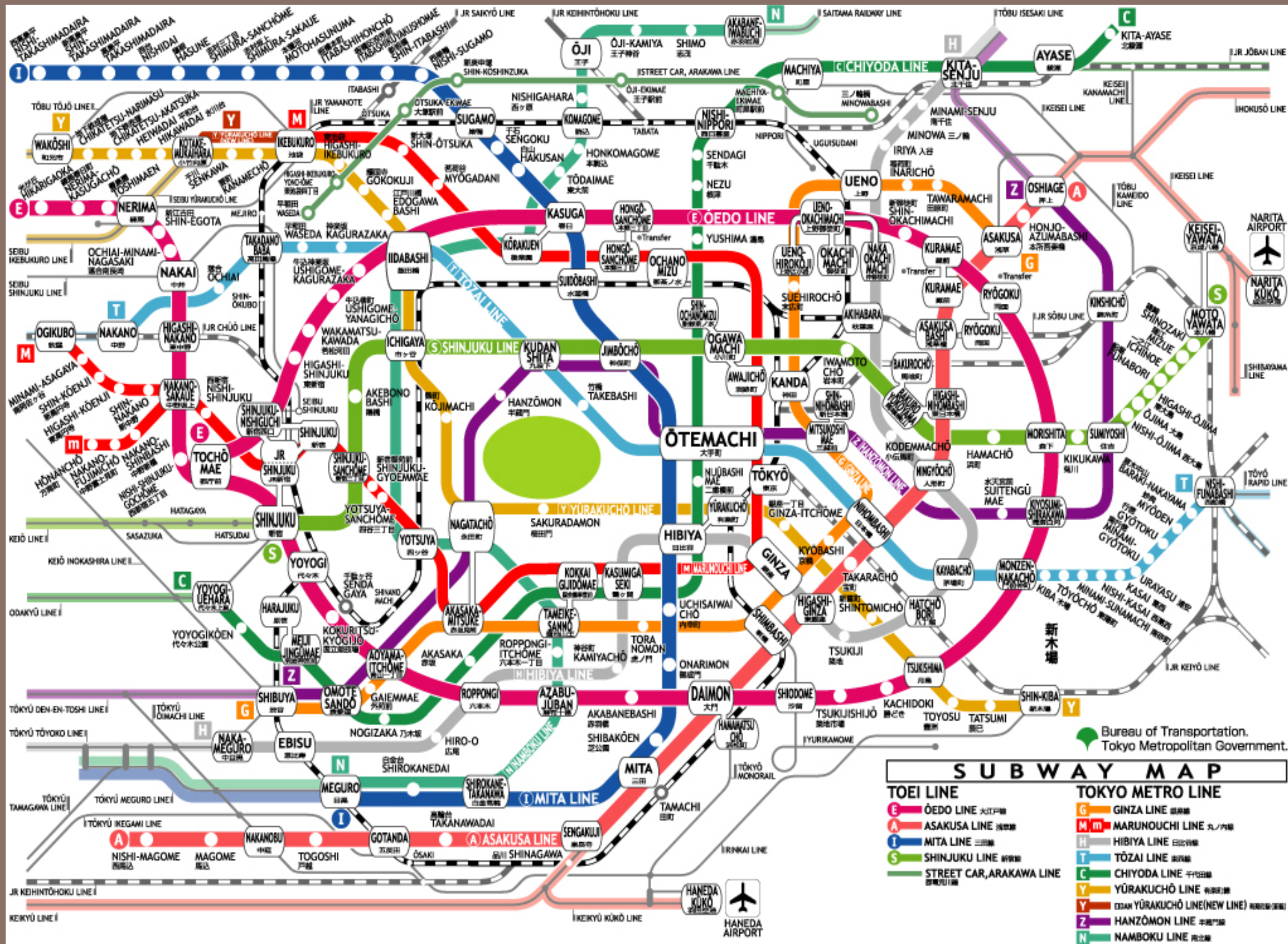


DFS AND SHORTEST PATHS

Lecture 18

CS2110 – Spring 2014

Readings?

- Read chapter 28

A3 "forgot a corner case"


```
while (true)
  try {
    if (in first column)
        if in last row, return StoredMap;
        fly south; refresh and save state, fly east
    if (in last column)
        if in last row, return StoredMap;
        fly south; refresh and save state, fly west
    if (row number is even)
        fly east;  refresh and save state;
    if (row number is odd)
        fly west;  refresh and save state;
    }
  catch (cliff exception e){
       if in last row, return StoredMap;
      fly south; refresh and save state
  }
```

It's not about "missing a corner case". The design is seriously flawed in that several horizontal fly(…) calls could cause the Bfly to fly past an edge, and there is no easy fix for this.

A3 “forgot a corner case”

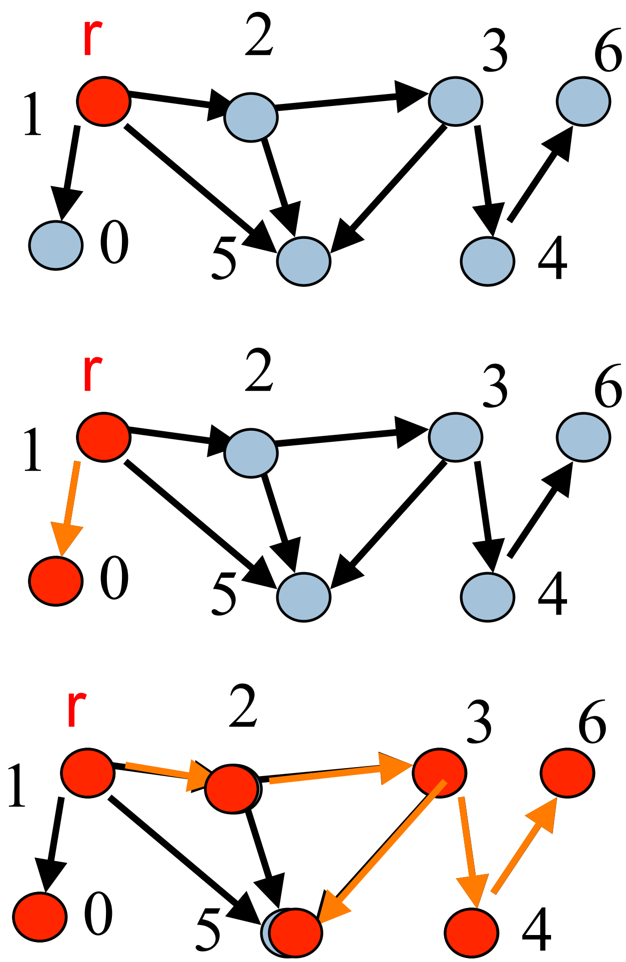
If you FIRST write the algorithm at a high level, ignoring Java details, you have a better chance of getting a good design

```
Direction dir= Direction.E;
while (true) {
    refresh and save the state;

    // Fly the Bfly ONE tile –return array if not possible
    if  in first col going west or last col going east
            if in last row, return the array;
            fly south and change direction;
      else try {
                fly in direction dir;
            } catch (cliff collision e) {
                 if in last row, return the array;
                 fly south and change direction;
            }
}
```

Depth-First Search (DFS)

Visit all nodes of a graph reachable from r.



Depth-first because:
Keep going down a path until
no longer possible

Depth-First Search

- Follow edges depth-first starting from an arbitrary vertex r, using a stack to remember where you came from
- When you encounter a vertex previously visited, or there are no outgoing edges, retreat and try another path
- Eventually visit all vertices reachable from r
- If there are still unvisited vertices, repeat
- O(m) time

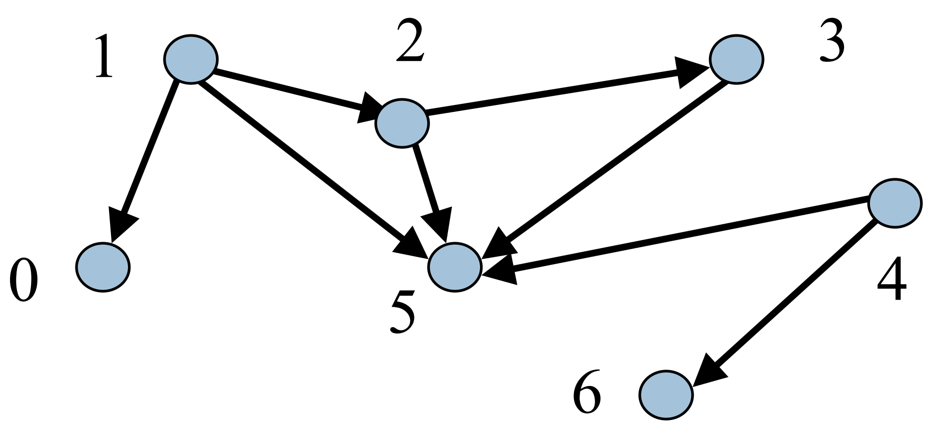
Difficult to understand!
Let's write a recursive procedure

Depth-First Search

boolean[] visited;

node u is visited means: visited[u] is true
To visit u means to: set visited[u] to true

Node v is REACHABLE from node u if there is a path (u, …, v) in which all nodes of the path are unvisited.



Suppose all nodes are unvisited.

The nodes that are REACHABLE from node 1 are 1, 0, 2, 3, 5

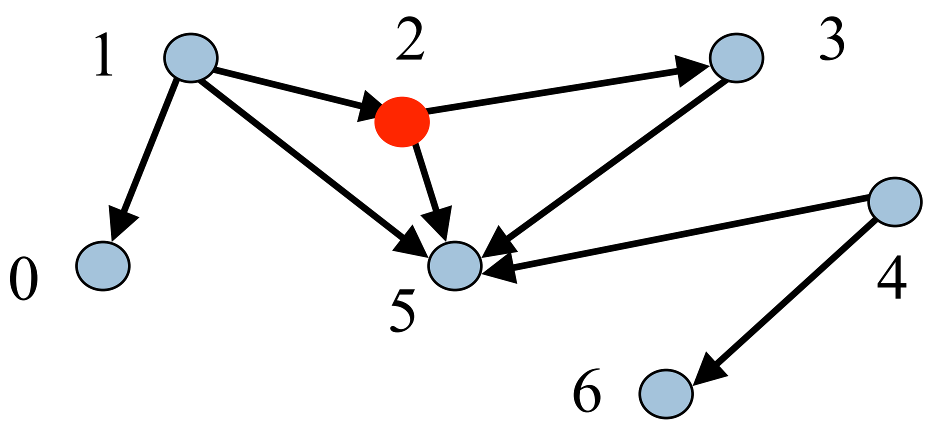
The nodes that are REACHABLE from 4 are 4, 5, 6.

Depth-First Search

boolean[] visited;

To "visit" a node u: set visited[u] to true.

Node u is REACHABLE from node v if there is a path (u, …, v) in which all nodes of the path are unvisited.



Suppose 2 is already visited, others unvisited.

The nodes that are REACHABLE from node 1 are 1, 0, 5

The nodes that are REACHABLE from 4 are 4, 5, 6.

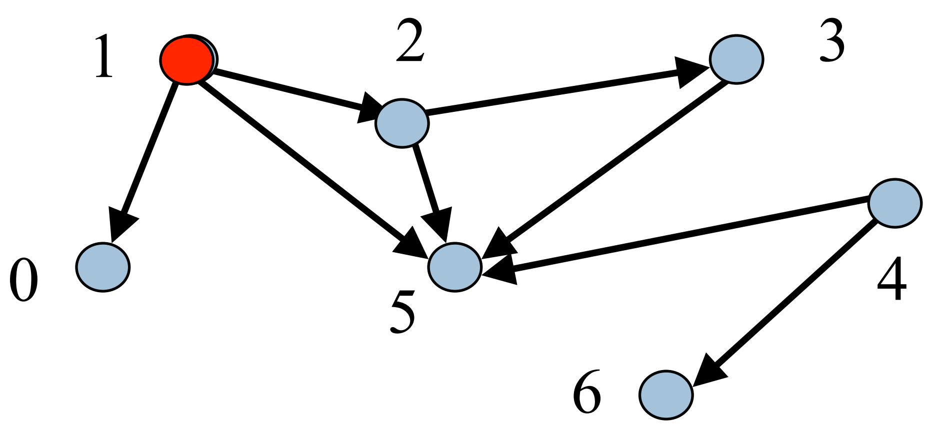
Depth-First Search


```
/** Node u is unvisited. Visit all nodes
    that are REACHABLE from u. */
public static void dfs(int u) {
      visited[u]= true;



}
```

Let u be 1
The nodes that are REACHABLE from node 1 are
1, 0, 2, 3, 5

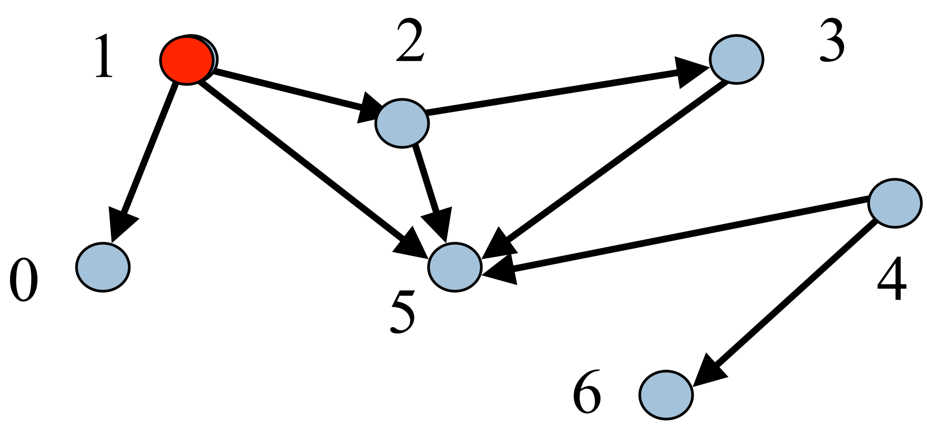


Depth-First Search


```
/** Node u is unvisited. Visit all nodes
    that are REACHABLE from u. */
public static void dfs(int u) {

     visited[u]= true;

     for each edge (u, v)
         if v is unvisited then dfs(v);
}
```



Let u be 1
The nodes to be visited are
0, 2, 3, 5

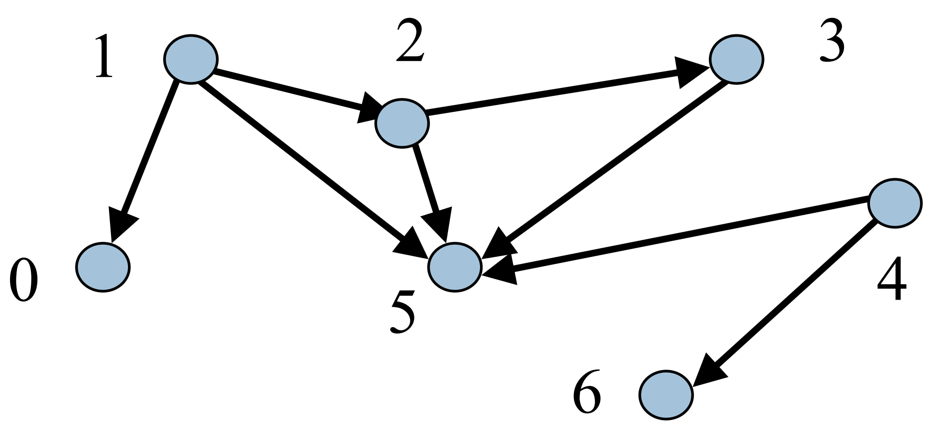
Have to do dfs on all unvisited neighbors of u

Depth-First Search


```
/** Node u is unvisited. Visit all nodes
    that are REACHABLE from u. */
public static void dfs(int u) {

    visited[u]= true;

    for each edge (u, v)
        if v is unvisited then dfs(v);
}
```



Let u be 1
The nodes to be visited are
0, 2, 3, 5

Suppose the for each loop visits neighbors in numerical order. Then dfs(1) visits the nodes in this order:
1, 0, 2, 3, 5

Depth-First Search


```
/** Node u is unvisited. Visit all nodes
    that are REACHABLE from u. */
public static void dfs(int u) {
     visited[u]= true;
     for each edge (u, v)
         if v is unvisited then dfs(v);
}
```

That's all there is to the basic dfs. You may have to change it to fit a particular situation.

Example: There may be a different way (other than array visited) to know whether a node has been visited

Example: Instead of using recursion, use a loop and maintain the stack yourself.

Shortest Paths in Graphs

Problem of finding shortest (min-cost) path in a graph occurs often

- Find shortest route between Ithaca and West Lafayette, IN
- Result depends on notion of cost
  - Least mileage… or least time… or cheapest
  - Perhaps, expends the least power in the butterfly while flying fastest
  - Many "costs" can be represented as edge weights

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, 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, Dijsktra 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).

Dijkstra's shortest path algorithm

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

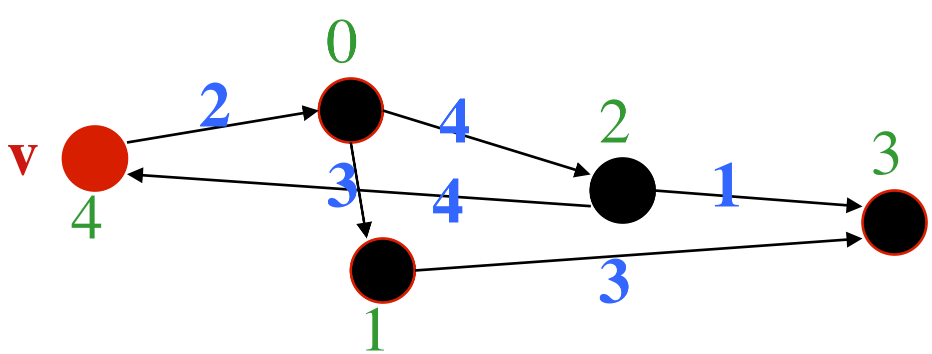
Each edge has a positive weight.

weight(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 L[0..n-1]: for **each** node w, store in L[w] the length of the shortest path from v to w.



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

Dijkstra's shortest path algorithm

Develop algorithm, not just present it.

Need to show you the state of affairs —the relation among all variables— just before each node i is given its final value L[i].

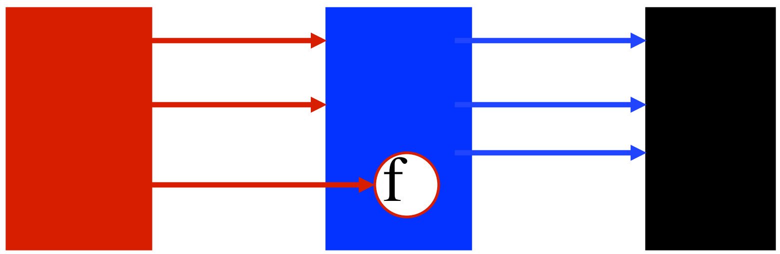
This relation among the variables is an *invariant*, because it is always true.

Because each node i (except the first) is given its final value L[i] during an iteration of a loop, the *invariant* is called a *loop invariant*.

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

Settled S Frontier F Far off

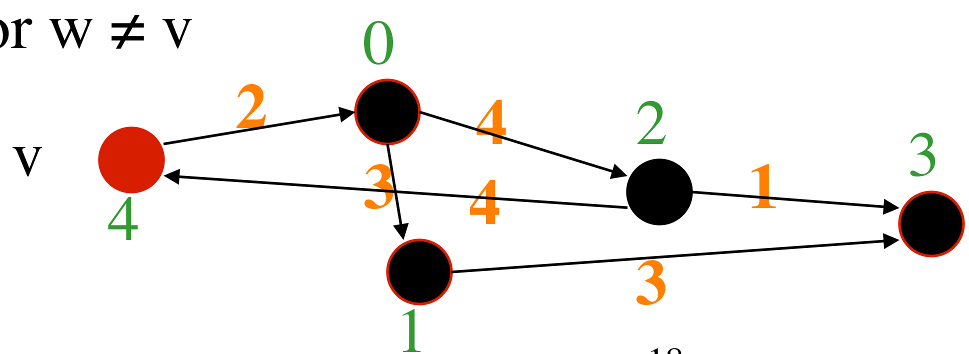
The loop invariant

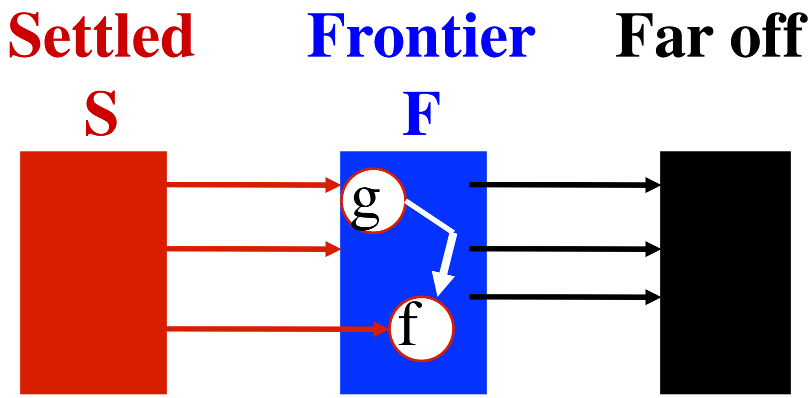


(edges leaving the black set and edges from the blue to the red set are not shown)

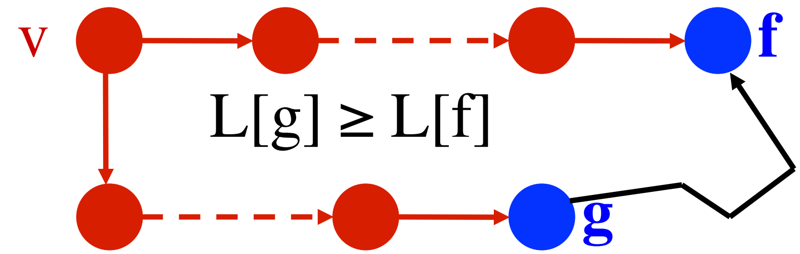
1. **For a Settled node s, L[s]** is length of shortest v → s path.
2. **All edges leaving S go to F.**
3. **For a Frontier node f, L[f] is length of shortest v → f path using only red nodes (except for f)**

4. **For a Far-off node b, L[b] = ∞**
5. L[v] = 0, L[w] > 0 for w ≠ v





Theorem about the invariant



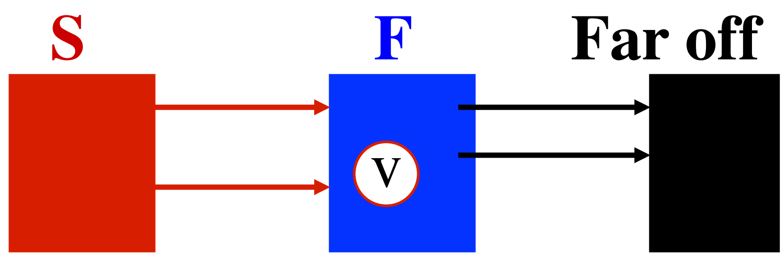
1. **For a Settled node s, L[s]** is length of shortest v → r path.
2. **All edges leaving S go to F.**
3. **For a Frontier node f, L[f]** is length of shortest v → f path using only Settled nodes (except for f).
4. **For a Far-off node b, L[b] = ∞. 5.** L[v] = 0, L[w] > 0 for w ≠ v

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

Case 1: **v** is in **S**.

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

The algorithm



1. **For s, L[s]** is length of shortest v→ s path.
2. **Edges leaving S go to F**.
3. **For f, L[f]** is length of shortest v → f path using red nodes (except for f).
4. **For b in Far off, L[b] = ∞**
5. L[v] = 0, L[w] > 0 for w ≠ v

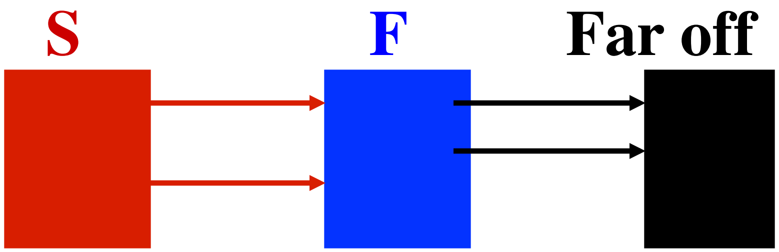
Theorem: For a node **f** in **F** with min L value, L[f] is shortest path length

For all w, L[w]= ∞;   L[v]= 0;
F= { v }; S= { };

Loopy question 1:

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

The algorithm



1. **For s, L[s]** is length of shortest v → s path.
2. **Edges leaving S go to F**.
3. **For f, L[f]** is length of shortest v → f path using red nodes (except for f).
4. **For b in Far off, L[b] = ∞**
5. L[v] = 0, L[w] > 0 for w ≠ v

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

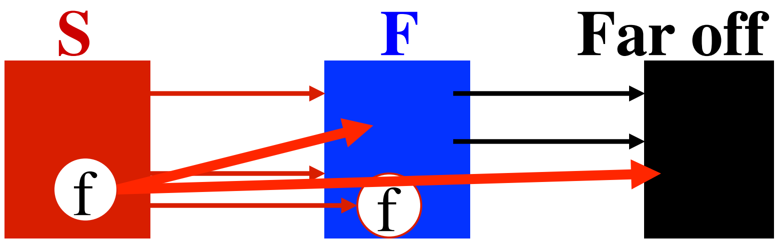
For all w, L[w]= ∞;   L[v]= 0;
F= { v }; S= { };
while   F ≠ {} {

}

Loopy question 2:

When does loop stop? When is array L completely calculated?

The algorithm



1. **For s**, **L[s]** is length of shortest v → s path.
2. **Edges leaving S go to F**.
3. **For f, L[f]** is length of shortest v → f path using red nodes (except for f).
4. **For b, L[b] = ∞**
5. L[v] = 0, L[w] > 0 for w ≠ v

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

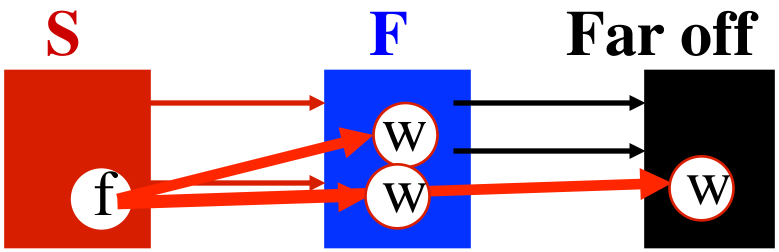
For all w, L[w]= ∞;   L[v]= 0;
F= { v }; S= { };
while   F ≠ {} {
  f= node in F with min L value;
  Remove f from F, add it to S;

}

Loopy question 3:

How is progress toward termination accomplished?

The algorithm



1. **For s, L[s]** is length of shortest v → s path.
2. **Edges leaving S go to F**.
3. **For f, L[f]** is length of shortest v → f path using red nodes (except for f).
4. **For b, L[b] = ∞**
5. **5.** L[v] = 0, L[w] > 0 for w ≠ v

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

```
For all w, L[w]= ∞;   L[v]= 0;
F= { v };  S= { };
while   F ≠ {} {
    f= node in F with min L value;
    Remove f from F, add it to S;
    for each edge (f,w) {
      if (L[w]  is ∞) add w to F;

      if (L[f] + weight (f,w) < L[w])
         L[w]= L[f] + weight(f,w);
    }
}
```

Algorithm is finished

Loopy question 4:

How is the invariant maintained?

About implementation

1. No need to implement **S**.
2. Implement **F** as a min-heap.
3. Instead of ∞, use
   Integer.MAX_VALUE.

For all w, L[w]= ∞; L[v]= 0;
F= { v }; ~~S= { };~~
while F ≠ {} {
  f= node in F with min L value;
  Remove f from F, ~~add it to S;~~
  for each edge (f,w) {
   ~~**if** (L[w] is ∞) add w to F;~~
   ~~**if** (L[f] + weight (f,w) < L[w])~~
    ~~L[w]= L[f] + weight(f,w);~~
  }
}

if (L[w] == Integer.MAX_VAL) {
  L[w]= L[f] + weight(f,w);
  add w to F;
} **else** L[w]= Math.min(L[w],
        L[f] + weight(f,w));

Execution time

n nodes, reachable from v. $e \geq n-1$ edges

$n-1 \leq e \leq n*n$

```
For all w, L[w]= ∞;  L[v]= 0;              O(n)
F= { v };                                  O(1)
while F ≠ {} {                             O(n)
  f= node in F with min L value;           O(n)
  Remove f from F;                         O(n log n)
  for each edge (f,w) {                    O(n + e)
    if (L[w] == Integer.MAX_VAL) {         O(e)
        L[w]= L[f] + weight(f,w);          O(n-1)
        add w to F;                        O(n log n)
    }
    else L[w]=                             O((e-(n-1)) log n)
        Math.min(L[w], L[f] + weight(f,w));
  }
}
```

outer loop:
n iterations.
Condition evaluated n+1 times.

inner loop:
e iterations.
Condition evaluated n + e times.

Complete graph: $O(n^2 \log n)$. Sparse graph: $O(n \log n)$