1

Friday is Halloween.
Why did I receive a

Christmas card on
Halloween?

SHORTEST PATHS

Lecture 19
CS2110 – Fall2014

Readings?

2

- Read chapter 28

Shortest Paths in Graphs

3

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

Every time you use googlemaps to find directions you are using a shortest-path algorithm

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.

4

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).

5

1968 NATO Conference on Software Engineering, Garmisch, Germany

Term "software engineering" coined for this conference

6

1968 NATO Conference on
Software Engineering, Garmisch, Germany

Marktoberdorf
Summer School,
Germany, 1998

(Each year,~100
PhD students
from around the
world would
get two weeks
of lectures by
CS faculty.

Dijkstra's shortest path algorithm

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

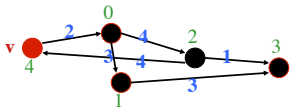
Each edge has a positive weight.

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

9

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.

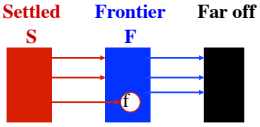
Because each node i (except the first) is given its final value L[i] during an iteration of a loop, the *invariant* is called a *loop invariant*.

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

10

Settled S | Frontier F | Far off

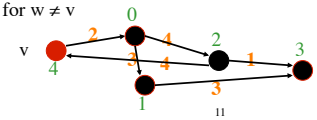
The loop invariant



(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)**

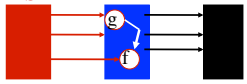
4. **For a Far-off node b, L[b] = ∞**
5. L[v] = 0, L[w] > 0 for w ≠ v

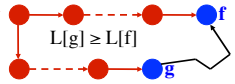


11

Settled S | Frontier F | Far off

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

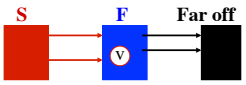
Theorem. For a node **f** in **F** with minimum L value (over nodes in **F**), **L[f]** is the length of the shortest path from **v** to **f**.

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

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

12

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

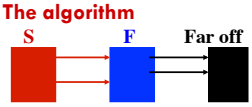
Theorem: For a node **f** in **F** with min L value, L[f] is shortest path length

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

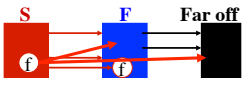
For all w, L[w]= ∞; L[v]= 0;
F= { v }; S= { };
while F ≠ {} {

}

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

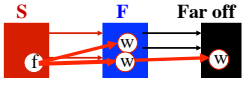
For all w, L[w]= ∞; L[v]= 0;
F= { v }; S= { };
while F ≠ {} {
f= node in F with min L value;
Remove f from F, add it to S;

}

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. 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?

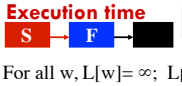
About implementation

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 ≥ n-1 edges
n–1 ≤ e ≤ n*n

For all w, L[w]= ∞; L[v]= υ; **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(n² log n). Sparse graph: O(n log n)