DFS AND SHORTEST PATHS

Lecture 18
CS2110 – Spring 2014

1

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; change direction;
    else try {
        fly in direction dir;
    } catch (cliff collision e) {  //...).}
        if in last row, return the array;
        fly south; 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
Node \( u \) is visited means: \( \text{visited}[u] \) is true.

To visit \( u \) means to: set \( \text{visited}[u] \) to true.

Node \( v \) is reachable from node \( u \) if there is a path \((u, \ldots, v)\) 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.

/** 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(\( v \));
}

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

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

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

Dijkstra 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, Dijkstra says, “my solution is preferred to another one … “the amount of work to be done seems considerably less.”


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.

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.

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

The loop invariant

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


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

Case 1: v is in S.
Case 2: v is in F. Note that L[v] is 0; it has minimum L value

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

The algorithm

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 2:
When does loop stop? When is array L completely calculated?

The algorithm

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 3:
How is progress toward termination accomplished?

About implementation

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);
} else L[w]= Math.min(L[w], L[f] + weight(f,w));
}
For all \( w \), \( L[w] = \infty \); \( L[v] = 0 \);
\( F = \{ v \} \);
while \( F \neq \{ \} \) {
f:= node in \( F \) with min \( L \) value;
Remove \( f \) from \( F \);
for each edge \((f,w)\) {
if \((L[w] \equiv \text{Integer.MAX_VALUE})\) {
\( L[w] = L[f] + \text{weight}(f,w); \)
add \( w \) to \( F \);
} else \( L[w] = \text{Math.min}(L[w], L[f] + \text{weight}(f,w)); \)
}
}

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