Lab 2

S2

Graph algorithms - graph traversal

Problem

- Given a starting vertex *s*, find all vertices that are accessible from it;
- Additionally, find a path, or a minimum length path, from the starting vertex to a given destination vertex;

Breadth-first traversal algorithm

The algorith below visits all the vertices that are accessible from the start vertex. They are visited in the order of increasing distances from the starting vertex. A *previous* vector or map is computed, allowing us to compute the minimum length path from the starting vertex to any choosen accessible vertex.

```
Input:
    G : graph
    s : a vertex
Output:
    accessible : the set of vertices that are accessible from s
    prev : a map that maps each accessible vertex to its predecessor on a path from s to it
Algorithm:
    Queue q
    Set visited
    q.add(s)
    visited.enqueue(s)
    while not q.isEmpty() do
        x = q.dequeue()
        for y in Nout(x) do
            if y not in visited then
                q.enqueue(y)
                visited.add(y)
                prev[y] = x
            end if
        end for
    end while
    accessible = visited
```

Accessibility - proof of correctness

The proof comes in 3 parts:

1. All returned vertices are accessible,
2. The algorithm finishes,
3. All accessible vertices are returned.

1. When a vertex is put into the queue, it is also put into the visited set, and no vertex is ever removed from the visited set, so any vertex in the queue is also in the visited set.

Next, we claim that, at each iteration, all vertices in the visited set are accessible from the start. At the beginning, this is true, because only the starting vertex is in the visited set. Next, a vertex is put in the visited set only if it is the outbound neighbour of a vertex in the queue; that vertex is therefore in the visited set and so it is accessible from start, so the added vertex is also accessible from start.

2. A vertex added to the queue only if not already in the visited set; it is also added in the visited set and never removed. So, any vertex gets at most once in the queue. So, the algorithm finishes in at most n iterations of the main loop. At each iteration, the inner loop executes outdeg(x) times, which sums up, on all iterations, to the total number of edges. So, the algorithm runs in $O(n+m)$.

3. We claim that, at each iteration, for any vertex x accessible from start, either x is in the visited set, or there is a walk going from start to x that has a vertex in the queue and that vertex is followed only by vertices not in the visited set.

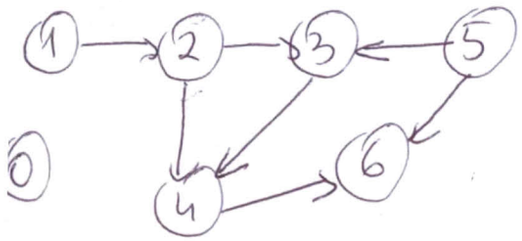
The above condition is true in the beginning. When going from one iteration to the next, there are two changes: the top vertex is extracted from the queue, and its neighbours are inserted into the queue and into the visited set.

If $(s=x_0,\ldots,x_j,x_{j+1},\ldots,x_k=t)$ is a path from the starting vertex, x_j is the top of the queue and $x_{j+1},\ldots,x_k$ are not visited, then x_{j+1} is added to the queue and is followed only by non-visited vertices.

If, though, a vertex x_l, with $l>j$, is added to the visited set, it is also added to the queue and is followed only by non-visited vertices.

Radu-Lucian LUPSA
2015-02-08

[Lab2] Lowest length path between s and t by using breadth-first search from s



Nout-dictionary

key	value
0	[ ]
1	[2]
2	[3,4]
3	[4]
4	[6]
5	[3,6]
6	[ ]

s=1, t=6	x	y	queue Q:	visited (set)	dist-dictionary	prev-dictionary
initialization			←|1|←	{1}	key: 0 1 2 3 4 5 6 / val: 0	0 1 2 3 4 5 6
iteration 1	1		=			
iteration 1.1		2	←|2|←	{1,2}	0 1 2 3 4 5 6 / 0 1	0 1 2 3 4 5 6 / 1
iteration 2	2		=			
iteration 2.1		3	←|3|←	{1,2,3}	0 1 2 3 4 5 6 / 0 1 2	0 1 2 3 4 5 6 / 1 2
iteration 2.2		4	←|3|4|←	{1,2,3,4}	0 1 2 2	1 2 2
iteration 3	3		←|4|←			
iteration 3.1		4				
iteration 4	4		=			
iteration 4.1		6		{1,2,3,4,6}	0 1 2 2 3	0 1 2 3 4 5 6 / 1 2 2 4

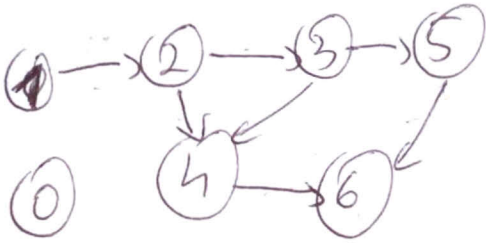
y = 6 = t ⇒ stop.

The reverse path is built from prev: 0 1 2 3 4 5 6 / 1 2 2 4 beginning with t = 6.

prev[6] = 4, prev[4] = 2, prev[2] = 1 = s ⇒ reverse-path: [6, 4, 2, 1].

path: [1, 2, 4, 6], length = dist[6] = 3.

Lab 2 | Lowest length path between s and t by using backward breadth-first search from t.



Nim-dictionary

key	value
0	[ ]
1	[ ]
2	[1]
3	[2]
4	[2,3]
5	[3]
6	[4,5]

s=1, t=6	x	y	queue: q	visited	dist-dictionary	next-dictionary
initialization			⟸ |6| ⟸	{6}	key: 0 1 2 3 4 5 6 / val: ∞ ∞ ∞ ∞ ∞ ∞ 0	0 1 2 3 4 5 6 / ∅ ∅ ∅ ∅ ∅ ∅ ∅
iteration 1	6					
iteration 1.1		4	⟸ |4| ⟸	{4,6}	0 1 2 3 4 5 6 / ∞ ∞ ∞ ∞ 1 ∞ 0	0 1 2 3 4 5 6 / ∅ ∅ ∅ ∅ 6 ∅ ∅
iteration 1.2		5	⟸ |4|5| ⟸	{4,5,6}	0 1 2 3 4 5 6 / ∞ ∞ ∞ 1 1 ∞ 0	0 1 2 3 4 5 6 / ∅ ∅ ∅ 6 6 ∅ ∅
iteration 2	4		⟸ |5| ⟸			
iteration 2.1		2	⟸ |5|2| ⟸	{2,4,5,6}	0 1 2 3 4 5 6 / ∞ ∞ 2 ∞ 1 1 0	0 1 2 3 4 5 6 / ∅ ∅ 4 ∅ 6 6 ∅
iteration 2.2		3	⟸ |5|2|3| ⟸	{2,3,4,5,6}	0 1 2 3 4 5 6 / ∞ ∞ 2 2 1 1 0	0 1 2 3 4 5 6 / ∅ ∅ 4 4 6 6 ∅
iteration 3	5		⟸ |2|3| ⟸			
iteration 3.1		3				
iteration 4	2		⟸ |3| ⟸			
iteration 4.1		1	⟸ |3|1| ⟸	{1,2,3,4,5,6}	0 1 2 3 4 5 6 / ∞ 3 2 2 1 1 0	0 1 2 3 4 5 6 / ∅ 2 4 4 6 6 ∅

y = s = 1 ⇒ stop.

The path is built from next: 0 1 2 3 4 5 6 / ∅ 2 4 4 6 6 ∅ ; beginning with s = 1.

s = 1, next[1] = 2, next[2] = 4, next[4] = 6 = t

path = [1, 2, 4, 6], length = dist[s] = dist[1] = 3.