CS 6901 (Applied Algorithms) – Lecture 4

Antonina Kolokolova

September 18, 2014

1 Basic graph algorithms

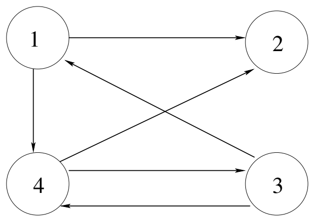
In this lecture, we will briefly review the main graph definitions and basic graph traversal algorithms: BFS and DFS.

1.1 Graph definitions

- A *graph* $G = (V, E)$ is defined by a set of vertices (nodes) V and a set of pairs of these vertices (edges) E. We will consider graphs without self-loops (that is, edges of the form (u, u)) or multiple edges. We usually use n to denote the number of vertices $|V|$ and m for the number of edges $|E|$.
- A graph is *undirected* if whenever $(u, v) \in E$, then $(v, u) \in E$. That is, every edge can be traversed in both directions. Otherwise, the graph is *directed*.
- A *path* in a graph is a sequence of vertices $v_1, \ldots, v_k$ such that $\forall i < k, (v_i, v_{i+1}) \in E$. That is, for every pair of subsequent vertices on the path, there is an edge. A path is *simple* if no vertex on the path repeats. A *cycle* is a path with the same first and last vertex.
- An undirected graph with no cycles is a *tree*. A directed graph with no cycles is a DAG ("directed acyclic graph").
- An indirected graph is *connected* if there is a path from any vertex to any other vertex. Otherwise, it can have several *connected components*. If each connected component of a graph which is not connected is a tree, then the graph is called a *forest*.
- A directed graph is *strongly connected* if there is a path from any vertex to any other vertex. It is *weakly connected* if there is such path in the underlying undirected graph (where we add (v, u) for any (u, v) in the original graph).
- An undirected graph is *bipartite* if the vertices can be split into two groups such that there are no edges between vertices in the same group. For example, graphs we used to illustrate the Stable Marriage problem were bipartite.

There are two main ways of representing a graph as an input to algorithms: *adjacency list* and *adjacency matrix*. In the adjacency list representation, for every vertex$u$ a list of its neighbours (nodes v such that there is an edge (u, v)) is given. In the adjacency matrix representation, a $n \times n$ matrix M is given, where $M(i, j) = 0$ if there is no edge from i to j, and $M(i, j) = 1$ otherwise.

Example 1 *The following is a directed graph, in which vertices* $(1, 4, 3)$ *form a directed cycle. This graph is weakly, but not strongly, connected.*



Adjacency list:

1: 2,4
2:
3: 1,4
4: 2,3

Adjacency matrix:

	1	*2*	*3*	*4*
1	*0*	*1*	*0*	*1*
2	*0*	*0*	*0*	*0*
3	*1*	*0*	*0*	*1*
4	*0*	*1*	*1*	*0*

1.2 BFS

A traversal algorithm visits nodes of a graph in some order. The two most well-known traversal algorithms are BFS (breadth first search) and DFS (depth first search). BFS lists vertices layer by layer, starting from a given vertex. DFS follows a path from a given vertex as far as possible, then backtracks to the nearest vertex that has unexplored edges and continues from there.

The following is the code for the BFS, starting from a given vertex s. We use the following convention (from CLRS book): an unexplored vertex is coloured white, a vertex being explored is gray, and once it is finished, it is coloured black. Usually you would need a wrapper function to initialize all vertices to be white, and sometime to continue exploring vertices in other connected components.

```
BFS($s$)
1  colour $s$ gray
2  insert $s$ into (initially empty) queue $Q$
3  while $Q$ is not empty
       do
4          take next vertex $u$ off $Q$
5          for each edge $(u, v) \in E$
               do
6                  if $v$ is white        ▷ $(u, v)$ becomes a BFS tree edge
                     then
7                        colour $v$ gray
8                        insert $v$ into $Q$
9          colour $u$ black
```

In the example graph above, starting from vertex $s = 1$, the algorithm would first process vertex 1 putting 2 and 4 on the queue (in some order, say first 4 and then 2), then process 4 putting 3 on the queue, then process 2, and finally 3. The three layers are $\{1\}$, $\{2, 4\}$ and $\{3\}$, and the tree edges are $(1, 2), (1, 4), (4, 3)$.

The running time of this algorithm is $O(m)$, as every edge is visited at most once. If there is a wrapper which makes sure that all the vertices are traversed in a graph which is not necessarily connected, then the running time can be better described as $O(n + m)$.

This is the basic BFS traversal. Now, it can be modified to do a number of things. For example, the only way to see an edge to a gray or black vertex while running BFS on an undirected graph is when the graph has a cycle, so the algorithm can be used to detect a cycle. Also, this algorithm always finds the shortest paths from s to any other reachable vertex in the graph; one common modification to keep a distance parameter say $v.dist$ and set $s.dist = 0$ at the beginning, then updating $v.dist = u.dist + 1$ inside the "if" statement.

Another interesting application of BFS is to test if a graph is bipartite. You can convince yourself that a graph is bipartite if and only if it has a cycle of odd length. Now, modify the algorithm above by setting

$s.side = 1$, and then, inside the "if", setting $v.side = -u.side$. In that way, vertices are assigned to either side "1" or side "-1". Also, add a check for gray vertices "else if v is gray, check if $v.side = u.side$; if so, output "graph is not bipartite. Otherwise, if each vertex got assigned a side, the graph is bipartite.

1.3 DFS

A different traversal algorithm DFS (depth first search) explores the recursively. Here, we again use the colour convention of white/gray/black, and also, following CLRS book, keep track of the time a vertex was first discovered ($u.d$) and a time a vertex was coloured black (finishing time, $u.f$). Assume that time is a global variable. Here, we will also use a wrapper to explore all vertices.

```
DFS-WRAPPER(G)
1  colour all vertices white and put them into a list
2  $time \leftarrow 0$
3  **while** there is a white vertex $u$
         **do**
4            DFS(u)
```

The following DFS algorithm explores the graph starting from a given vertex.

```
DFS($s$)
1  $time \leftarrow time + 1$
2  $u.d \leftarrow time$; $u.colour \leftarrow gray$
3  **for** each $v$ such that $(u, v) \in E$
         **do**
4            **if** $v$ is white, DFS(v)
5  $u.colour \leftarrow black$
6  $time \leftarrow time + 1$
7  $u.f \leftarrow time$
```

In the example graph above, if we start traversing from vertex 1, one of the possible executions of DFS proceeds like this. Set the discovery time for vertex 1 (let us call it v_1 here to avoid confusion) to "1." Then, for example, go to vertex v_4, and set its discovery time $v_4.d = 2$. From there, say we explore v_2 next, with discovery time $v_2.d = 3$. As there is nowhere to go from v_2, we are finished with this vertex; set $v_2.f = 4$ and backtrack to v_4. From there, go to v_3, resulting in $v_3.d = 5$ and $v_3.f = 6$. Back to v_4, which is now done so it gets $v_4.f = 7$. Finally, set $v_1.f = 8$. Sometimes we will use the notation "2/7" to mean the discovery time is 2 and the finish time is 7 for a specific vertex (especially on the pictures). The edges of the DFS tree for this run of DFS are $(1, 4), (4, 2), (4, 3)$.

Note one property of these discovery and finish times of vertices: if a vertex v is a descendant of a vertex u (that is, v is in the tree underneath the u), then $u.d < v.d < v.f < u.f$. This property is quite useful for proving correctness of algorithms based on DFS.