CS 6901 (Applied Algorithms) – Lecture 3

Antonina Kolokolova

September 20, 2016

1 Basic graph algorithms

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

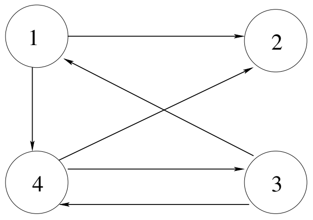
1.1 Graph definitions

- 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.
- A *degree* of a vertex v in an undirected graph, $deg(v)$, is the number of neighbours of v (that is, number of u such that $E(v, u)$). In a directed graph, in-degree and out-degree are the number of incoming (respectively, outgoing) edges from a given vertex. We will sometimes use $deg(v)$ for an out-degree in a directed graph.
- 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. These representations apply both to directed and undirected graphs.

The type of a representation is important for analysis of the running time of a graph algorithm, as different operations take different time on adjacency list vs. adjacency matrix. For example, checking whether there is an edge from u to v takes constant time in the adjacency matrix representation, whereas it can take time $O(m)$ in the adjacency list (more precisely, it takes time $O(deg(u))$.) However, adjacency matrix always takes space n^2, even if m is small relative to n, whereas adjacency list takes space $n + m$. Therefore, listing all edges can be much faster when a graph is given as an adjacency list.

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. These traversals work on both directed and undirected graphs.

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. There is also a wrapper function to initialize all vertices to be white, and store them in a list. For example, if an undirected graph is not connected, then the wrapper checks, after running $BFS(s)$, if there are any white vertices left, and if so, picks one and runs BFS from it. That allows it to explore vertices in all connected components.

```
BFS-WRAPPER($G$)
1  colour all vertices white and put them into a list
2  while there is a white vertex $u$
       do
3          BFS(u)
```

The following code traverses a connected component starting with a vertex s.

```
BFS($s$)
1 colour $s$ gray and remove it from the list of white vertices.
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 and remove it from the list of white vertices.
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 $O(n + m)$, when the input is given in adjacency list representation, as every edge is visited at most once, and the wrapper function visits once every vertex that may or may not have edges.

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 not 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, graph is not bipartite. Otherwise, if each vertex got assigned a side, the graph is bipartite.