Graphs and BFS

We will devote two lectures of the Algorithms and Data Structures thread to an introduction to graph algorithms. As with many other topics we could spend the entire course on this area.

9.1 Directed and Undirected Graphs

A *graph* is a mathematical structure consisting of a set of *vertices* and a set of *edges* connecting the vertices. Formally, we view the edges as pairs of vertices; an edge (v, w) connects vertex v with vertex w. Formally:

- V is a set, and
- $E \subseteq V \times V$.

We write $G = (V, E)$ to denote that G is a graph with vertex set V and edge set E. A graph $G = (V, E)$ is *undirected* if for all vertices $v, w \in V$, we have $(v, w) \in E$ if and only if $(w, v) \in E$, that is, if all edges go both ways. This definition makes it clear that undirected graphs are just special directed graphs. When studying undirected graphs we often represent a complementary pair of directed edges (u, v) and (v, u) as just one 'undirected' edge using some special notation. In this introduction we will not go to this extent but state in each case if a graph is directed or undirected. A useful convention is that in drawing diagrams of directed graphs we indicate the direction of an edge with an arrow. In the case of undirected graphs we do not draw pairs of directed edges (one from u to v and one from v to u) but just one edge without an arrow[1]. If we want to emphasise that the edges have a direction, we say that a graph is *directed*.

Note that in principle V and hence E can be infinite sets. Such graphs are very useful in many areas (including computing) but for this introduction we will assume always that V is finite. Since $E \subseteq V \times V$ is follows that E is also finite.

Example 9.1. Figure 9.2 shows a drawing of the (directed) graph $G = (V, E)$ with vertex and edge sets given by:

$$V = \{0, 1, 2, 3, 4, 5, 6\}$$
$$E = \{(0, 2), (0, 4), (0, 5), (1, 0), (2, 1), (2, 5), (3, 1), (3, 6), (4, 0), (4, 5), (6, 3), (6, 5)\}.$$

We state here a definition that will be needed subsequently.

Definition 9.3. Let $v \in V$ be a vertex in a directed graph $G = (V, E)$.

- The *in-degree* $\text{in}(v)$ of v is the number of incoming edges to v, i.e., the number of edges of form (u, v). The set of in-edges to u is written as $\text{In}(v)$.
- The *out-degree* $\text{out}(v)$ of v is the number of outgoing edges from v, i.e., the number of edges of form (v, u). The set of out-edges from v is written as $\text{Out}(v)$.

[1] Do not confuse a diagram representing a graph with the graph itself. The graph is the abstract mathematical structure and can be represented by many different diagrams.

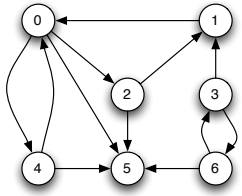

Figure 9.2. A directed graph

In an undirected graph, the *degree* of v is the number of edges $e \in E$ for which one endpoint is v. Two vertices are *adjacent* if they are joined by an edge. You should also know definitions of graph theory concepts such as *paths*, *cycles*, *connectedness* and *connected components* from your *Maths-for-Informatics* courses. You will find them in any textbook on graph theory or discrete mathematics and also in most books on algorithms. Graphs are a useful mathematical model for numerous problems and structures. We give just a few examples.

Example 9.4. *Airline route maps.*
Vertices represent airports, and there is an edge from vertex A to vertex B if there is a direct flight from the airport represented by A to the airport represented by B.

Example 9.5. *Electrical Circuits.*
Vertices represent diodes, transistors, capacitors, switches, etc., and edges represent wires connecting them.

Example 9.6. *Computer Networks.*
Vertices represent computers and edges represent network connections (e.g., cables) between them.

Example 9.7. *The World Wide Web.*
Vertices represent webpages, and edges represent hyperlinks.

Example 9.8. *Flowcharts.*
A flowchart illustrates the flow of control in a procedure. Essentially, a flowchart consists of boxes (vertices) containing statements of the procedure and arrows (directed edges) connecting the boxes to describe the flow of control.

Example 9.9. *Molecules.*
Vertices are atoms, edges are bonds between them.

The graphs in Examples 9.4, 9.7 and 9.8 are directed. The graphs in Examples 9.5, 9.6 and 9.9 are undirected.

9.2 Data structures for graphs

Let $G = (V, E)$ be a graph with n vertices. We assume that the vertices of G are numbered $0, \ldots, n-1$ in some arbitrary manner.

The adjacency matrix data structure

The *adjacency matrix* of G is the $n \times n$ matrix $A = (a_{ij})_{0 \le i,j \le n-1}$ with

$$a_{ij} = \begin{cases} 1, & \text{if there is an edge from vertex number } i \\ & \text{to vertex number } j; \\ 0, & \text{otherwise.} \end{cases}$$

For example, the adjacency matrix for the graph in Figure 9.2 is

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Note that the adjacency matrix depends on the particular numbering of the vertices.

The adjacency matrix data structure stores the adjacency matrix of the graph as a 2-dimensional Boolean array, where TRUE represents 1 (i.e., there is an edge) and FALSE represents 0 (i.e., there is no edge). It is worth noting here that for some applications it is preferable to use actual numbers rather than Boolean values. The main advantage of the adjacency matrix representation is that for all vertices v and w we can check in constant time whether or not there is an edge from vertex v to vertex w.

However we pay a price for this fast access. Let m be the number of edges of the graph. Note that m can be at most n^2. If m is close to n^2, we call the graph *dense*, and if m is much smaller than n^2 we call it *sparse*. Storing a graph with n vertices and m edges will usually require space at least $\Omega(n+m)$. However, the adjacency matrix uses space $\Theta(n^2)$, and this is much more than $\Theta(n+m)$ for sparse graphs (when a large fraction of entries in the adjacency matrix consists of zero). Moreover, many algorithms have to inspect all edges of the graph at least once, and to do this for a graph given in adjacency matrix representation, such an algorithm will have to inspect every matrix entry at least once to make sure that it has seen all edges. Thus it will require time $\Omega(n^2)$; we will see some important algorithms that run in time $\Theta(n+m)$ with an appropriate data structure.

The adjacency list data structure

The *adjacency list* representation of a graph G with n vertices consists of an array *vertices* with n entries, one for each vertex. The entry for vertex v is a list of all vertices w such there is an edge from v to w. We make no assumptions on the order in which the vertices adjacent to a vertex v appear in the adjacency list, and our algorithms should work for any order.

Figure 9.10 shows an adjacency list representation of the graph in Figure 9.2.

For sparse graphs an adjacency list is more space efficient than an adjacency matrix. For a graph with n vertices and m edges it requires space $\Theta(n+m)$, which might be much less than $\Theta(n^2)$. Moreover, if a graph is given in adjacency list

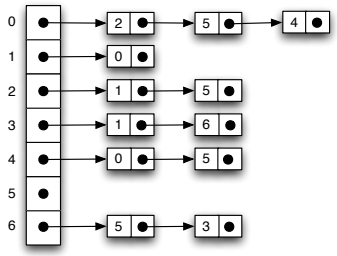

Figure 9.10. Adjacency list representation of the graph in Figure 9.2

representation one can visit efficiently all neighbours of a vertex v; this just requires time

$$\Theta\big(1 + \text{out}(v)\big)$$

and not time $\Theta(n)$ as for the adjacency matrix representation[2]. Therefore, visiting all edges of the graph only requires time $\Theta(n+m)$ and not time $\Theta(n^2)$ as for the adjacency matrix representation. On the other hand, finding out whether there is an edge from vertex v to vertex w requires (in the worst case) stepping through the whole adjacency list of v, which could have up to n entries. Thus a *simple adjacency test* takes time $\Theta(n)$ in the worst case, compared to $\Theta(1)$ for adjacency matrices.

Extensions

We have only described the basic data structures representing graphs. Vertices are just represented by the numbers they get in some numbering, and edges by the numbers of their endpoints. Often, we want to store additional information. For example, in Example 9.4 we might want to store the names of the airports represented by the vertices, or in Example 9.7 the URLs of the webpages. To do this, we create separate vertex objects that store the number of a vertex and an object that contains the additional data we want to store at the vertex. In the adjacency list representation, we include the adjacency list of a vertex in the vertex object. Then the graph is represented by an array (possibly a dynamic array) of vertex objects.

Similarly, we might want to store additional information on the edges of a graph; in Example 9.4 we may want to store flight numbers. We can do this by setting up separate edge objects which will store references to the two endpoints of an edge and the additional information. Then in the adjacency list representations, the lists would be lists of such edge objects.

A frequent situation is that edges of a graph carry *weights*, which are real numbers providing information such as the cost of a flight in Example 9.4 or the capacity of a wire or network connection in Examples 9.5 and 9.6. Graphs whose edges carry weights are called *weighted graphs*.

[2] It might be tempting to replace $\Theta\big(1 + \text{out}(v)\big)$ with $\Theta\big(\text{out}(v)\big)$ but this is wrong if $\text{out}(v) = 0$.

9.3 Traversing Graphs

Most algorithms for solving problems on graphs examine or process each vertex and each edge of the graph in some particular order. The skeleton of such an algorithm will be a *traversal* of the graph, that is, a strategy for visiting the vertices and edges in a suitable order.

Breadth-first search (BFS) and depth-first search (DFS) are two traversals that are particularly useful. Both start at some vertex v and then visit all vertices reachable from v (that is, all vertices w such that there is a path from v to w). If there are vertices that remain unvisited, that is, if there are vertices that are not reachable from v, then the only way they can be listed is if the search chooses a new unvisited vertex v' and visits all vertices reachable from v'. This process would have to be repeated until all vertices are visited.

We can present the general graph searching strategy as algorithms 9.11 and 9.12. In these algorithms we assume that we start with every vertex unmarked and we

Algorithm search(G)

1. ensure that each vertex of G is not marked
2. initialise schedule S
3. **for all** $v \in V$ **do**
4. **if** v is not marked **then**
5. searchFromVertex(G, v)

Algorithm 9.11

Algorithm searchFromVertex(G, v)

1. mark v
2. put v onto schedule S
3. **while** schedule S is not empty **do**
4. remove a vertex v from S
5. **for all** w adjacent to v **do**
6. **if** w is not marked **then**
7. mark w
8. put w onto schedule S

Algorithm 9.12

have some efficient way of marking it. Note that once a vertex is marked it stays as such. The schedule S is some (efficient) data structure such that we can put vertices on it and take them off one at a time. For BFS we use a *Queue* while for DFS we use a *Stack* as our schedule S. As we will see, because recursive procedures use an inherent stack we will be able to re-express DFS rather neatly without

an explicit schedule S (all the same it is there but given to us by the underlying system for handling recursion). Algorithm 9.12 should be viewed as a general plan which we can to amend slightly for various forms of search. Indeed, for DFS we will change the method of marking vertices by replacing lines 1 and 7 of Algorithm 9.12 with a single new line after (what is now) line 5. This simplifies the pseudocode and still ensures that we do not end up in a cycle visiting the same vertices repeatedly.

In our presentation we think of vertices as being in one of two states: unmarked and marked. There is some advantage to thinking of them as being in one of three states which we represent by colours:

- White: not yet seen.
- Grey: put on schedule.
- Black: taken off schedule.

Each vertex starts off as white then becomes grey and finally black. We can think of these states as corresponding to "not yet investigated", "under investigation", "completed". We will use the three colour scheme later on when studying an algorithm for topological sorting of graphs.

Breadth-First Search

A BFS starting at a vertex v first visits v, then it visits all neighbours of v (i.e., all vertices w such that there is an edge from v to w), then all neighbours of the neighbours that have not been visited before, then all neighbours of the neighbours of the neighbours that have not been visited before, etc. For example, one BFS of the graph in Figure 9.2 starting at vertex 0 would visit the vertices in the following order:

$$0, 2, 5, 4, 1$$

It first visits 0, then the neighbours $2, 5, 4$ of 0. Next are the neighbours of 2, which are 1 and 5. Since 5 has been visited before, only 1 is added to the list. All neighbours of 5, 4, and 1 have already been visited, so we have found all vertices that are reachable from 0. Note that there are other orders in which a BFS starting at 0 could visit the vertices of the graph, because the neighbours of 0 might be visited in a different order. If the neighbour vertices are visited in numerical order, then the BFS from 0 would be $0, 2, 4, 5, 1$. Vertices 3 and 6 are not reachable from 0, so to visit them we must to start another BFS, say at 3.

The traversal heavily depends on the vertex we start at. If we start a BFS at vertex 6, for example, all vertices are reachable, and the vertices are visited in one sweep, for example:

$$6, 5, 3, 1, 0, 2, 4.$$

Other possible orders are $6, 3, 5, 1, 0, 2, 4$ and $6, 5, 3, 1, 0, 4, 2$ and $6, 3, 5, 1, 0, 4, 2$.

In an *undirected graph*, however, the number of different BFS searches (or DFS searches) that need to be made to visit all vertices is independent of the choice of start vertices.

During a BFS we have to store vertices that have been visited so far and also the vertices that have been completely processed (all their neighbours have been visited). We maintain a Boolean array *visited* with one entry for each vertex, which

is set to TRUE when the vertex is visited. The vertices that have been visited, but have not been completely processed, are stored in a *Queue*. This guarantees that vertices are visited in the right order—vertices that are discovered first will be processed first. Algorithms 9.13 and 9.14 show a BFS implementation in pseudocode. The main algorithm bfs first initialises the *visited* array and the queue and then loops through all vertices, starting a bfsFromVertex for all vertices that have not been marked 'visited' in previous invocations of bfsFromVertex. The subroutine bfsFromVertex visits all vertices reachable from the start vertex in the way described above. The inner loop in lines 5–8 can be implemented using an iterator over the adjacency list of the vertex v.

Algorithm bfs(G)

1. Initialise Boolean array *visited* by setting all entries to FALSE
2. Initialise *Queue* Q
3. **for all** $v \in V$ **do**
4. **if** *visited*$[v]$ = FALSE **then**
5. bfsFromVertex(G, v)

Algorithm 9.13

Algorithm bfsFromVertex(G, v)

1. *visited*$[v]$ = TRUE
2. Q.enqueue(v)
3. **while not** Q.isEmpty() **do**
4. $v \leftarrow Q$.dequeue()
5. **for all** w adjacent to v **do**
6. **if** *visited*$[w]$ = FALSE **then**
7. *visited*$[w]$ = TRUE
8. Q.enqueue(w)

Algorithm 9.14

To see the progress of bfs(G) we can put a **print** v statement after each *visited*$[v]$ = TRUE. In practice, BFS (or DFS) is often applied to applications such as all-pairs shortest paths for a graph. For applications like this, the "current vertex" obtained by BFS is used in the top-level algorithm for the particular application.

Exercises

1. Give an adjacency matrix and an adjacency list representation for the graph displayed in Figure 9.15. Give orders in which a BFS starting at vertex n may

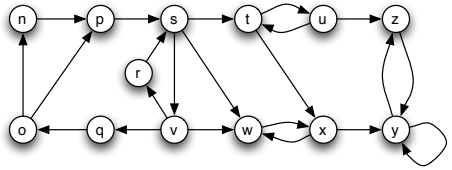

Figure 9.15.

traverse the graph.