Graphs

The remaining lectures of the Algorithms and Data Structures thread will be devoted to graph algorithms.

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 \(w\). 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. If we want to emphasise that the edges have a direction, we call a graph directed. For brevity, a directed graph is often called a digraph, and an undirected graph is simply called a graph. We shall not use this convention here; for us ‘graph’ always means ‘directed or undirected graph’.

When drawing graphs, we represent a vertex by a point or circle containing the name of the vertex, and an edge by an arrow connecting two vertices. When drawing undirected graphs, instead of drawing two arrows (one in each direction) between all vertices, we just draw one line connecting the vertices.

Example 9.1. Figure 9.2 shows a drawing of the (directed) graph \(G = (V, E)\) with vertex set

\[
V = \{0, 1, 2, 3, 4, 5, 6\}
\]

and edge set

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

I will assume that you are familiar with basic notions from graph theory from your maths classes. You should know notions such as adjacent vertices, degree of a vertex in an undirected graph and in-degree and out-degree of a vertex in a directed graph, paths, cycles, connectedness, connected components. If you don’t, go to the library and look these notions up. 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 "real life" problems and structures. Here are a few examples.

Example 9.3. 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 9.4. Road Maps.
Edges represent streets and vertices represent crossings.

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

Vertices represent computers and edges represent network connections (cables) between them.

Vertices represent webpages, and edges represent hyperlinks.

A flowchart illustrates the flow of control in a procedure. Essentially, a flowchart consists of boxes containing statements of the procedure and arrows connecting the boxes to describe the flow of control. In a graph representing a flowchart, the vertices represent the boxes and the edges represent the arrows.

Suppose we want assign time slots to exams. Of course we must not assign overlapping time slots to exams taken by the same students. We can model the situation by a graph whose vertices represent the exams, with an edge between two vertices if there is a student who has to take both exams.

Example 9.10. Molecules.
Vertices are atoms, edges are bonds between them.

For example, consider the ‘divides’ relation on the integers. The vertices of a graph representing this relation are the integers, and there is an edge from \( a \) to \( b \) if \( a \) divides \( b \).
This is an example of an infinite graph. In the following, we will only talk about finite graphs. For example, the ‘divides’ relation on the integers \(\{1, \ldots, 10\}\) yields a finite graph.

The graphs in Examples 9.3, 9.7, 9.8, and 9.11 are directed. The graphs in Examples 9.5, 9.6, 9.9, and 9.10 are undirected. For Example 9.4 it depends on whether we want to take one-way streets into account or not.

### 9.2 Data structures for graphs

We have seen that graphs are a useful mathematical model. We could now define an abstract data type for graphs by listing methods we would like our graphs to support. (Recall from LN3 that an ADT is a mathematical model equipped with appropriate methods for accessing and modifying it.) We skip this step and go directly to the data structure level, where we discuss only informally how the most basic methods can be implemented.

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 \(A = (a_{ij})_{0 \leq i, j \leq n-1}\) with

\[
a_{ij} = \begin{cases} 
1 & \text{if there is an edge from vertex number } i \\
0 & \text{otherwise.}
\end{cases}
\]

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 \texttt{TRUE} represents 1 (i.e., there is an edge) and \texttt{FALSE} represents 0 (i.e., there is no edge). 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 \(w\).

But we pay a price for this fast access to the adjacencies. 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 may be much more than \( \Theta(n + m) \) for sparse graphs (for which most entries in the adjacency matrix will be 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 \( \Omega(n^2) \).

The adjacency list data structure

The adjacency list representation of a graph \( G \) with \( n \) vertices consists of an array \( \text{vertices} \) 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 9.12 shows an adjacency list representation of the graph in Figure 9.2.

Figure 9.12. Adjacency list representation of the graph in Figure 9.2

Note that for sparse graphs the adjacency list representation is more space efficient than the adjacency matrix representation. For a graph with \( n \) vertices and \( m \) edges it requires space \( \Theta(n + m) \), which may be much less than \( \Theta(n^2) \). Moreover, if a graph is given in adjacency list representation one can efficiently visit all neighbours of a vertex \( v \); this just requires time 

\[
\Theta(1 + \text{out-degree}(v))
\]

and not time \( \Theta(n) \) as for the adjacency matrix representation. 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 stepping through the whole adjacency list of \( v \), which may 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 9.3 we may 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 which store the number of a vertex and an object that contains the additional data we may 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. (You can find a JAVA implementation of this enhanced adjacency list data structure for directed and undirected graphs on the CS2 lecture notes web page.) Similarly, we may want to store additional information on the edges of a graph. For example, in Example 9.3 we may want to store flight numbers, or in Example 9.4 road names. 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 9.3, the distance 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.

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 BFS and DFS pick a new vertex \( v' \) and visit all vertices reachable from \( v' \). They repeat this process until they have finally visited all vertices.

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, et cetera. For example, a 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 may visit the vertices of the graph, because the neighbours of 0 may be visited in a different order. An example is 0, 5, 4, 2, 1. The vertices 3 and 6 are not reachable from 0, so we have to start another BFS, say at 3. It first visits 3 and then 6.

It is important to realise that the traversal heavily depends on the vertex we start at. For example, if we start a BFS at vertex 6 it will visit all vertices in one sweep, maybe in the following order:

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.

During a breadth-first search we have to keep track of which vertices have been visited so far and which vertices have been completely processed in the sense that all their neighbours have also been visited. To keep track of which vertices have been visited so far, we set up a Boolean array visited with one entry for each vertex, which is set to TRUE when the vertex is visited. To keep track of which vertices have been visited, but not fully processed yet, we store them in a Queue. This guarantees that vertices are visited in the right order, because the 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 loop can easily be implemented using an iterator over all vertices of the graph. 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** \( \text{bfs}(G) \)

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

Algorithm 9.13

Of course if we start \( \text{bfs}(G) \) we will see nothing happening. To get something useful out of it, we have to add additional instructions. As a simple example, we may just put a print \( v \) statement after each \( \text{visited}[v] = \text{TRUE} \). Then the algorithm will print the vertices in the order in which they are visited. More interesting applications of BFS include algorithms for finding the shortest path between any two vertices in a graph.
Algorithm \texttt{bfsFromVertex}(G, v)

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

Algorithm 9.14

9.4 Depth-first search

A DFS starting at a vertex \( v \) first visits \( v \), then some neighbour \( w \) of \( v \), then some neighbour \( x \) of \( w \) that has not been visited before, et cetera. Once it gets stuck, the DFS backtracks until it finds the first vertex that still has a neighbour that has not been visited before. It continues with this neighbour until it has to backtrack again. Eventually, it will visit all vertices reachable from \( v \). Then a new DFS is started at some vertex that is not reachable from \( v \), until all vertices have been visited.

For example, a DFS in the graph of Figure 9.2 starting at 0 may visit the vertices in the order

\[ 0, 2, 1, 5, 4. \]

After it has visited 0, 2, 1 the DFS backtracks to 2, visits 5, then backtracks to 0, and visits 4. A DFS starting at 0 might also visit the vertices in the order 0, 4, 5, 2, 1 or 0, 5, 4, 2, 1 or 0, 1, 5, 4, 2. As for BFS, this depends on the order in which the neighbours of a vertex are processed.

DFS can be implemented in a similar way as BFS using a \texttt{Stack} instead of a \texttt{Queue}, see Algorithms 9.15 and 9.16. There is also an elegant recursive implementation of DFS which will be discussed in the next lecture.

Algorithm \texttt{dfs}(G)

1. Initialise Boolean array \texttt{visited} by setting all entries to \texttt{FALSE}
2. Initialise \texttt{Stack} \texttt{S}
3. \texttt{for all} \texttt{v} \in \texttt{V} \texttt{do}
   4. \texttt{if} \texttt{visited}[\texttt{v}] = \texttt{FALSE} \texttt{then}
   5. \texttt{dfsFromVertex}(G, \texttt{v})

Algorithm 9.15
**Algorithm** dfsFromVertex\((G, v)\)

1. \(S\).push\((v)\)
2. **while not** \(S\).isEmpty() **do**
3. \(v \leftarrow S\).pop()
4. **if** visited\([v]\) = FALSE **then**
5. \(\text{visited}[v] = \text{TRUE}\)
6. **for all** \(w\) adjacent to \(v\) **do**
7. \(S\).push\((w)\)

**Algorithm 9.16**

**Exercises**

1. Give an adjacency matrix and an adjacency list representation for the graph displayed in Figure 9.17. Give orders in which a BFS and DFS starting at vertex \(n\) may traverse the graph.

![Figure 9.17.](image)

2. Let \(G\) be a graph with \(n\) vertices and \(m\) edges. Explain why \(O(\log n)\) is \(O(\log m)\).

3. Suppose you want to explore the web graph (starting from a particular web page) by using BFS. Clearly the Web graph is too large to store in your computer’s main memory, and you’ll probably only want to explore a small part of it (like all web pages that can be reached via at most ten hyperlinks).

What is the problem with using the BFS version described in Algorithms 9.13 and 9.14 for this purpose? How can you fix it?

*Don Sannella*