Belgian Olympiad in Informatics

Easter Training — Day 2

Graphs: Basics and Common Algorithms

Trainer:
Victor Lecomte

April 15, 2014

Contents

There are intentionally no implementations in these notes, in order to let the students figure it out by themselves.

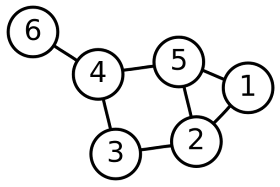
1 Basics of graphs

In this section we'll define what a graph is, learn about the associated vocabulary and figure out ways to represent them in memory.

1.1 Definition

Basically, a graph is a set of vertices and a set of edges linking them by pairs.

Mathematically, a graph is a pair $G = (V, E)$ where V is the set of vertices and E is a set of pairs of vertices. The definition may vary according to the type of graph and the author.



In the above example,

$$\begin{aligned} V &= \{1,2,3,4,5,6\} \\ E &= \big\{\{1,2\},\{1,5\},\{2,3\},\{2,5\},\{3,4\},\{4,5\},\{4,6\}\big\} \end{aligned}$$

1.2 Terminology

1.2.1 Basics

- The points are called *vertices* or *nodes.*
- The links are called *edges* or *lines.*
- Two vertices connected by an edge are *adjacent*, or *neighbours.*
- Vertices belonging to an edge are called the *ends* or *endpoints.*
- A *loop* is an edge whose ends are the same vertex.

1.2.2 Properties

- The *order* of a graph is $|V|$ (the number of vertices).
- The *size* of a graph is $|E|$ (the number of edges).
- The *degree* of a vertex is the number of edges that connect to it.

1.2.3 Classes of graphs

- A *simple* graph has no loop and no more than one edge between any two vertices. The opposite is a *multigraph.*
- In a *directed* graph, the edges point in a specific direction. They are represented by *ordered* pairs of vertices.
- In a *weighted* graph, a number (the weight) is assigned to each edge. It can represent cost, length, etc.

1.2.4 Paths and cycles

- A *path* is a sequence of adjacent vertices, all different. In a directed graph, the arrows have to point in the right direction.
- Two vertices are *connected* if there is a path between them.
- A graph is *connected* if all pairs of vertices are connected.

- A *cycle* is a path that goes back to beginning.
- An *acyclic* graph has no cycle.

1.2.5 Special graphs

Those are all simple, undirected graphs.

- A *tree* is a connected, acyclic graph.
- In a *complete* graph, each pair of vertices is joined by an edge; it contains all possible edges.
- In a *sparse* graph, only very few pairs of vertices are joined by an edge; it is very incomplete. It is useful when algorithms depend on the number of edges.

1.3 Representation

Let us now discover two ways to represent graphs in memory, and find out their respective advantages.

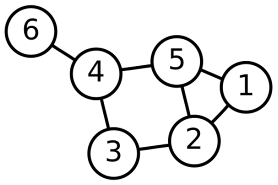
1.3.1 Adjacency matrix

The adjacency matrix is a two-dimensional array `a`, such that the element

$$\texttt{a[i][j]} = \begin{cases} 1 & \text{if there is an edge from } i \text{ to } j, \\ 0 & \text{otherwise.} \end{cases}$$

note that for an undirected graph, `a` is symmetric.

Let us take our first example again:



The corresponding adjacency matrix is:

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

For non-simple graphs, 1 is replaced by the number of edges joining the vertices (loops count twice).

1.3.2 Adjacency list

The adjacency list contains for each vertice a list of all its neighbours.

For the same example, the adjacency list is:

$$\begin{cases} 1: & 2,5 \\ 2: & 1,3,5 \\ 3: & 2,4 \\ 4: & 3,5,6 \\ 5: & 1,2,4 \\ 6: & 4 \end{cases}$$

The order of the neighbours does not matter.

1.3.3 Comparison

In terms of memory usage, adjacency lists will generally be more compact for sparse graphs, since it will only store data for existing edges, while adjacency matrices will store a bit for every pair of vertices. But for near-complete graphs, adjacency matrices will be more space-efficient.

As for time, it depends on the kind of operation you have to do. In many cases, you have to visit all neighbours of a node, so adjacency lists are faster, but if you often check whether two nodes are adjacent, the adjacency matrix is more appropriate.

2 Traversal: DFS and BFS

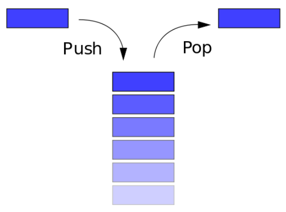
Traversing a graph means visiting its nodes in some order. To do that, we need to store the nodes we discover in a container so that we remember to explore them later. In this section, we will introduce the different containers we will use for that, then what kind of behavior they induce in the traversal.

2.1 Stacks and queues

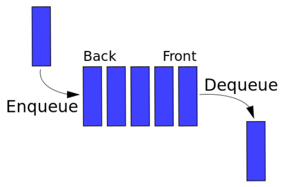
Stacks and queues are the containers we will use here. They are very similar in nature, and they are both special cases of *deques*, double-ended queues, but they will make a huge difference in the order of exploration.

Deques are lists that allow insertion, access and deletion of elements on both ends, in constant time. We will not worry about the implementation here.

A *stack* only allows insertion, acess and deletion at the back of the list. We can view it as a stack of pancakes: you can only add or remove a pancake at the top of the pile. It follows the *LIFO* (last-in-first-out) principle: the last element that is inserted is the first to be processed.



A *queue*, on the other hand allows insertion only at the back, while acess and deletion only happen at the front. We can view it as a queue at the checkout in a store. It follows the *FIFO* (first-in-first-out) principle: the elements are processed in the same order that they were inserted.



The corresponding function members are `.push()` for insertion, `.pop()` for deletion and for acess, stacks have `.top()` and queues have `.front()`.

2.2 Depth-first search

When traversing a graph, the same procedure happens over and over again:

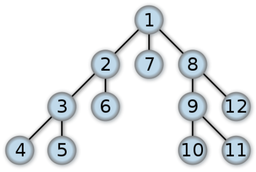
- choose a node;
- check that it has not been visited before;
- process the node;
- add its neighbours to the waiting list.

At the beginning, we add the starting point to the waiting list, then we execute that procedure until the waiting list is empty.

If the graph is connected, it means we have traversed it entirely. Otherwise, it means we have traversed one *connected component* of the graph, and we have to choose a new starting point.

Depth-first search, or *DFS*, uses a stack as waiting list. The consequence is that as soon as a new node is discovered, it will be explored, so depth-first search will progress quickly through the graph, going in *depth* instead of exploring its most direct neighbourhood first.

The diagram below shows the order in which nodes are visited, taking the example of a tree:



DFS can also be implemented with a recursive function: instead of adding the neighbours to the waiting list, just calling the visit procedure on them will have the same effect. This may sound more simple, but internally it still uses the a stack, the *call stack*, that stores information about which functions the program is in.

An interesting property is that the search never jumps from one side of the graph to the other. After it has visited the first neighbour, the search only has to go back to the original node through one edge, then continue with the next neighbour. In the above example, the path taken would be $1, 2, 3, 4, 3, 5, 3, 2, 6, 2, 1, 7, 1, 8, \ldots$ It goes through every edge exactly two times.

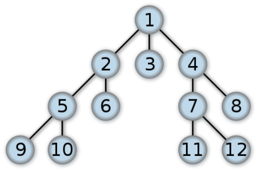
This makes it a viable option to physically traverse a graph (for example, finding the center of a maze with loops, or exploring a cave system in Minecraft) without having to remember anything, at the cost of a few markers.

2.3 Breadth-first search

Breadth-first search, or *BFS* uses a queue as a waiting list, and that is literally the only difference in the implementation.

What changes is the order the nodes will be processed in. Since the nodes are taken from the queue in the same order as they were inserted, the algorithm will first process all the neighbours of the starting point, then when it has done that it will process the neighbours of the neighbours, etc. It will work its way *by layers*.

The diagram below shows the order in which nodes are visited for the same example:



The advantage of BFS is that if we keep track of which edges it took to get to the nodes, it gives us the shortest path from the starting point to every node. That is because of the order in which BFS visits the nodes. It first makes sure all the nodes at distance 1 have been processed before moving on to the nodes at distance 2, so we can be sure there was no path shorter than 2. The same applies for distance 3, then 4, and so on.

Note that it only gives the shortest path in terms of number of edges. If the edges have different costs (that is, if the graph is weighted), then BFS will not work.

2.4 Comparison

Let n be the number of nodes and m the number of edges. The complexity of both algorithms is $O(n + m)$, assuming an adjacency list structure, so there is no time difference. But this shows they are particularly efficient for sparse graphs.

As a consequence, the decision depends on their advantages. DFS is simpler to implement if you use recursive functions, and it is perfect if you cannot acess any node instantly. BFS will mostly be useful when you have to find a shortest path in a non-weighted graph.

3 Shortest paths

3.1 Dijkstra's algorithm

Dijkstra's algorithm is the counterpart of BFS for weighted graphs, and like BFS, it gives the shortest path from one node to all other nodes.

It will process all the nodes in increasing order of their distance to the starting point. In order to determine that distance, it will keep an array, `dist[i]`, that will keep the shortest distance found *so far* from the starting point to vertice i.

At each step it will follow this procedure:

- find the smallest node in `dist[]` not yet visited;
- process the node;
- update `dist[]` for its neighbours.

This approach is correct but not optimal: for each node you have to go through all the elements of `dist[]` and find the minimum, so the complexity is $O(n^2)$. As before, n is the number of nodes and m is the number of edges.

We can do better by using a heap (priority queue) that keeps the smallest element on top. The procedure becomes:

- choose the node at the top of the heap;
- check that is has not been visited before;
- process the node;
- add the neighbours to the heap with their updated distance.

This second version has complexity $O(m + n \log n)$. It is more efficient on sparse graphs, but on graphs with a lot of edges you should use the first one, as it is simpler and more space efficient.

3.2 Floyd-Warshall algorithm

The Floyd-Warshall algorithm is interesting when you have to know the shortest path from any node to any other node, as opposed to one node to all other nodes.

Floyd uses dynamic programming by progressively filling an array `dist[][]` with the distance from a node to another. `dist[i][j]` is initialised with:

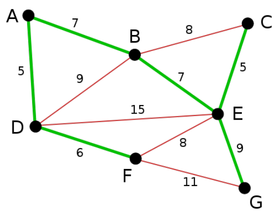
- 0 if $i = j$;
- the weight of the edge from i to j, if it exists;
- ∞ otherwise.

Then the algorithm is simply:

```
for(int k=0; k<n; k++)
    for(int i=0; i<n; i++)
        for(int j=0; j<n; j++)
            dist[i][j] = min(dist[i][j] + dist[j][k], dist[i][j]);
```

4 Minimum spanning trees

In a connected graph, a spanning tree is a tree that contains all nodes and a subset of the edges. A minimum spanning tree is a spanning tree with minimum total length. An example is given below:



Finding the minimum spanning tree means finding the shortest edges in total that keep the graph connected.

4.1 Kruskal's algorithm

In order to construct the minimum spanning tree, Kruskal's algorithm just picks the smallest edges first, unless adding them forms a cycle (it would not be a tree anymore). This is a greedy approach, and surpisingly it works.

To check if adding an edge, we can check if they belong to the same connected component. Since those connected components are merged together as we add edges, they will best be represented with a union-find data structure.

The algorithm goes through the edges in increasing order of weight. For each edge (u, v), if u and v are in different components, add (u, v) to the tree and unite their components; otherwise, discard the edge.

The complexity is $O(m \log m)$.

4.2 Prim's algorithm

The approach of Prim's algorithm is slightly different from Kruskal's. Instead of looking at the shortest edges in the whole graph, it begins with a starting point and picks edges to add around it. That way it will slowly spread to the whole graph, each time choosing the shortest edge that is in contact with the current zone.

It traverses the graph in nearly the same way as Dijkstra's algorithm, the only difference being that Dijstra picks vertices according to their total distance from the starting point, while for Prim only the last edge matters. Usually, only one line needs to be changed.

When implemented with a heap, Prim's complexity is also $O(m \log m)$.

5 Sources of the figures

- `http://en.wikipedia.org/wiki/File:6n-graf.svg`
- `http://en.wikipedia.org/wiki/File:Data_stack.svg`
- `http://en.wikipedia.org/wiki/File:Data_Queue.svg`
- `http://en.wikipedia.org/wiki/File:Depth-first-tree.svg`
- `http://en.wikipedia.org/wiki/File:Breadth-first-tree.svg`
- `http://en.wikipedia.org/wiki/File:Kruskal_Algorithm_6.svg`