Algorithm Tutorials

Introduction to graphs and their data structures: Section 1

Archive
Normal view
Discuss this article
Write for TopCoder

By gladius
TopCoder Member

Introduction

Graphs are a fundamental data structure in the world of programming, and this is no less so on TopCoder. Usually appearing as the hard problem in Division 2, or the medium or hard problem in Division 1, there are many different forms solving a graph problem can take. They can range in difficulty from finding a path on a 2D grid from a start location to an end location, to something as hard as finding the maximum amount of water that you can route through a set of pipes, each of which has a maximum capacity (also known as the maximum-flow minimum-cut problem - which we will discuss later). Knowing the correct data structures to use with graph problems is critical. A problem that appears intractable may prove to be a few lines with the proper data structure, and luckily for us the standard libraries of the languages used by TopCoder help us a great deal here!

Recognizing a graph problem

The first key to solving a graph related problem is recognizing that it is a graph problem. This can be more difficult than it sounds, because the problem writers don't usually spell it out for you. Nearly all graph problems will somehow use a grid or network in the problem, but sometimes these will be well disguised. Secondly, if you are required to find a path of any sort, it is usually a graph problem as well. Some common keywords associated with graph problems are: vertices, nodes, edges, connections, connectivity, paths, cycles and direction. An example of a description of a simple problem that exhibits some of these characteristics is:

"Bob has become lost in his neighborhood. He needs to get from his current position back to his home. Bob's neighborhood is a 2 dimensional grid, that starts at (0, 0) and (width - 1, height - 1). There are empty spaces upon which bob can walk with no difficulty, and houses, which Bob cannot pass through. Bob may only move horizontally or vertically by one square at a time.

Bob's initial position will be represented by a 'B' and the house location will be represented by an 'H'. Empty squares on the grid are represented by '.' and houses are represented by 'X'. Find the minimum number of steps it takes Bob to get back home, but if it is not possible for Bob to return home, return -1.

An example of a neighborhood of width 7 and height 5:

```
...X..B
.X.X.XX
.H.....
...X...
.....X."
```

Once you have recognized that the problem is a graph problem it is time to start building up your representation of the graph in memory.

Representing a graph and key concepts

Graphs can represent many different types of systems, from a two-dimensional grid (as in the problem above) to a map of the internet that shows how long it takes data to move from computer A to computer B. We first need to define what components a graph consists of. In fact there are only two, nodes and edges. A node (or vertex) is a discrete position in the graph. An edge (or connection) is a link between two vertices that can be either directed or undirected and may have a cost associated with it. An undirected edge means that there is no restriction on the direction you can travel along the edge. So for example, if there were an undirected edge from A to B you could move from A to B or from B to A. A directed edge only allows travel in one direction, so if there were a directed edge from A to B you could travel from A to B, but not from B to A. An easy way to think about edges and vertices is that edges are a function of two vertices that returns a cost. We will see an example of this methodology in a second.

For those that are used to the mathematical description of graphs, a graph G = {V, E} is defined as a set of vertices, V, and a collection of edges (which is not necessarily a set), E. An edge can then be defined as (u, v) where u and v are elements of V. There are a few technical terms that it would be useful to discuss at this point as well:

Order - The number of vertices in a graph Size - The number of edges in a graph

Singly linked lists

An example of one of the simplest types of graphs is a singly linked list! Now we can start to see the power of the graph data structure, as it can represent very complicated relationships, but also something as simple as a list.

A singly linked list has one "head" node, and each node has a link to the next node. So the structure looks like this:

```
structure node
   [node] link;
   [data]
end

node head;
```

A simple example would be:

```
node B, C;
head.next = B;
B.next = C;
C.next = null;
```

This would be represented graphically as head -> B -> C -> null. I've used null here to represent the end of a list.

Getting back to the concept of a cost function, our cost function would look as follows:

```
cost(X, Y) := if (X.link = Y) return 1;
          else if (X = Y) return 0;
          else "Not possible"
```

This cost function represents the fact that we can only move directly to the link node from our current node. Get used to seeing cost functions because anytime that you encounter a graph problem you will be dealing with them in some form or another! A question that you may be asking at this point is "Wait a second, the cost from A to C would return not possible, but I can get to C from A by stepping through B!" This is a very valid point, but the cost function simply encodes the *direct* cost from a node to another. We will cover how to find distances in generic graphs later on.

Now that we have seen an example of the one of the simplest types of graphs, we will move to a more complicated example.

Trees

There will be a whole section written on trees. We are going to cover them very briefly as a stepping-stone along the way to a full-fledged graph. In our list example above we are somewhat limited in the type of data we can represent. For example, if you wanted to start a family tree (a hierarchal organization of children to parents, starting from one child) you would not be able to store more than one parent per child. So we obviously need a new type of data structure. Our new node structure will look something like this:

```
structure node
   [node] mother, father;
   [string] name
end

node originalChild;
```

With a cost function of:

```
cost(X, Y) := if ((X.mother = Y) or (X.father = Y)) return 1;
           else if (X = Y) return 0;
           else "Not possible"
```

Here we can see that every node has a mother and father. And since node is a recursive structure definition, every mother has mother and father, and every father has a mother and father, and so on. One of the problems here is that it might be possible to form a loop if you actually represented this data structure on a computer. And a tree clearly cannot have a loop. A little mind exercise will make this clear: a father of a child is also the son of that child? It's starting to make my head hurt already. So you have to be very careful when constructing a tree to make sure that it is truly a tree structure, and not a more general graph. A more formal definition of a tree is that it is a connected acyclic graph. This simply means that there are no cycles in the graph and every node is connected to at least one other node in the graph.

Another thing to note is that we could imagine a situation easily where the tree requires more than two node references, for example in an organizational hierarchy, you can have a manager who manages many people then the CEO manages many managers. Our example above was what is known as a binary tree, since it only has two node references. Next we will move onto constructing a data structure that can represent a general graph!

Graphs

A tree only allows a node to have children, and there cannot be any loops in the tree, with a more general graph we can represent many different situations. A very common example used is flight paths between cities. If there is a flight between city A and city B there is an edge between the cities. The cost of the edge can be the length of time that it takes for the flight, or perhaps the amount of fuel used.

The way that we will represent this is to have a concept of a node (or vertex) that contains links to other nodes, and the data associated with that node. So for our flight path example we might have the name of the airport as the node data, and for every flight leaving that city we have an element in neighbors that points to the destination.

```
structure node
   [list of nodes] neighbors
   [data]
end

cost(X, Y) := if (X.neighbors contains Y) return X.neighbors[Y];
           else "Not possible"

list nodes;
```


This is a very general way to represent a graph. It allows us to have multiple edges from one node to another and it is a very compact representation of a graph as well. However the downside is that it is usually more difficult to work with than other representations (such as the array method discussed below).

Array representation

Representing a graph as a list of nodes is a very flexible method. But usually on TopCoder we have limits on the problems that attempt to make life easier for us. Normally our graphs are relatively small, with a small number of nodes and edges. When this is the case we can use a different type of data structure that is easier to work with.

The basic concept is to have a 2 dimensional array of integers, where the element in row i, at column j represents the edge cost from node i to j. If the connection from i to j is not possible, we use some sort of sentinel value (usually a very large or small value, like -1 or the maximum integer). Another nice thing about this type of structure is that we can represent directed or undirected edges very easily.

So for example, the following connection matrix:

```
    A  B  C
A   0  1  5
B  -1  0  1
C  -1 -1  0
```

Would mean that node A has a 0 weight connection to itself, a 1 weight connection to node B and 5 weight connection to node C. Node B on the other hand has no connection to node A, a 0 weight connection to itself, and a 1 weight connection to C. Node C is connected to nobody. This graph would look like this if you were to draw it:

This representation is very convenient for graphs that do not have multiple edges between each node, and allows us to simplify working with the graph.

...continue to Section 2

