Introduction to graphs and their data structures: Section 1

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:

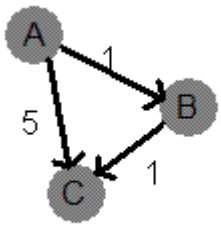
```
    A  B  C

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.

Basic methods for searching graphs

Introduction

So far we have learned how to represent our graph in memory, but now we need to start doing something with this information. There are two methods for searching graphs that are extremely prevalent, and will form the foundations for more advanced algorithms later on. These two methods are the Depth First Search and the Breadth First Search.

We will begin with the depth first search method, which will utilize a stack. This stack can either by represented explicitly (by a stack

data-type in our language) or implicitly when using recursive functions.

Stack

A stack is one of the simplest data structures available. There are four main operations on a stack:

1. Push - Adds an element to the top of the stack
2. Pop - Removes the top element from the stack
3. Top - Returns the top element on the stack
4. Empty - Tests if the stack is empty or not

In C++, this is done with the STL class stack:

```
#include

stack myStack;
```

In Java, we use the Stack class:

```
import java.util.*;

Stack stack = new Stack();
```

In C#, we use Stack class:

```
using System.Collections;

Stack stack = new Stack();
```

Depth First Search

Now to solve an actual problem using our search! The depth first search is well geared towards problems where we want to find any solution to the problem (not necessarily the shortest path), or to visit all of the nodes in the graph. A recent TopCoder problem was a classic application of the depth first search, the flood-fill. The flood-fill operation will be familiar to anyone who has used a graphic painting application. The concept is to fill a bounded region with a single color, without leaking outside the boundaries.

This concept maps extremely well to a Depth First search. The basic concept is to visit a node, then push all of the nodes to be visited onto the stack. To find the next node to visit we simply pop a node of the stack, and then push all the nodes connected to that one onto the stack as well and we continue doing this until all nodes are visited. It is a key property of the Depth First search that we not visit the same node more than once, otherwise it is quite possible that we will recurse infinitely. We do this by marking the node as we visit it, then unmarking it after we have finished our recursions. This action allows us to visit all the paths that exist in a graph; however for large graphs this is mostly infeasible so we sometimes omit the marking the node as not visited step to just find one valid path through the graph (which is good enough most of the time).

So the basic structure will look something like this:

```
dfs(node start) {

 stack s;

 s.push(start);

 while (s.empty() == false) {

  top = s.top();

  s.pop();

  mark top as visited;
```

```
  check for termination condition

  add all of top's unvisited neighbors to the stack.
  mark top as not visited;
 }
}
```

Alternatively we can define the function recursively as follows:

```
dfs(node current) {
 mark current as visited;
 visit all of current's unvisited neighbors by calling dfs(neighbor)
 mark current as not visited;
}
```

The problem we will be discussing is grafixMask, a Division 1 500 point problem from SRM 211. This problem essentially asks us to find the number of discrete regions in a grid that has been filled in with some values already. Dealing with grids as graphs is a very powerful technique, and in this case makes the problem quite easy.

We will define a graph where each node has 4 connections, one each to the node above, left, right and below. However, we can represent these connections implicitly within the grid, we need not build out any new data structures. The structure we will use to represent the grid in grafixMask is a two dimensional array of booleans, where regions that we have already determined to be filled in will be set to true, and regions that are unfilled are set to false.

To set up this array given the data from the problem is very simple, and looks something like this:

```
bool fill[600][400];
initialize fills to false;

foreach rectangle in Rectangles
    set from (rectangle.left, rectangle.top) to (rectangle.right, retangle.bottom) to true
```

Now we have an initialized connectivity grid. When we want to move from grid position (x, y) we can either move up, down, left or right. When we want to move up for example, we simply check the grid position in (x, y-1) to see if it is true or false. If the grid position is false, we can move there, if it is true, we cannot.

Now we need to determine the area of each region that is left. We don't want to count regions twice, or pixels twice either, so what we will do is set fill[x][y] to true when we visit the node at (x, y). This will allow us to perform a Depth-First search to visit all of the nodes in a connected region and never visit any node twice, which is exactly what the problem wants us to do! So our loop after setting everything up will be:

```
int[] result;
```

```
for x = 0 to 599
 for y = 0 to 399
  if (fill[x][y] == false)
   result.addToBack(doFill(x,y));
```

All this code does is check if we have not already filled in the position at (x, y) and then calls doFill() to fill in that region. At this point we have a choice, we can define doFill recursively (which is usually the quickest and easiest way to do a depth first search), or we can define it explicitly using the built in stack classes. I will cover the recursive method first, but we will soon see for this problem there are some serious issues with the recursive method.

We will now define doFill to return the size of the connected area and the start position of the area:

```
int doFill(int x, int y) {
// Check to ensure that we are within the bounds of the grid, if not, return 0
 if (x < 0 || x >= 600) return 0;
// Similar check for y
 if (y < 0 || y >= 400) return 0;
// Check that we haven't already visited this position, as we don't want to count it twice
 if (fill[x][y]) return 0;

// Record that we have visited this node
 fill[x][y] = true;

 // Now we know that we have at least one empty square, then we will recursively attempt to
 // visit every node adjacent to this node, and add those results together to return.
 return 1 + doFill(x - 1, y) + doFill(x + 1, y) + doFill(x, y + 1) + doFill(x, y - 1);
}
```

This solution should work fine, however there is a limitation due to the architecture of computer programs. Unfortunately, the memory for the implicit stack, which is what we are using for the recursion above is more limited than the general heap memory. In this instance, we will probably overflow the maximum size of our stack due to the way the recursion works, so we will next discuss the explicit method of solving this problem.

Sidenote:
Stack memory is used whenever you call a function; the variables to the function are pushed onto the stack by the compiler for you. When using a recursive function, the variables keep getting pushed on until the function returns. Also any variables the compiler needs to save between function calls must be pushed onto the stack as well. This makes it somewhat difficult to predict if you will run into stack difficulties. I recommend using the explicit Depth First search for every situation you are at least somewhat concerned about recursion depth.

In this problem we may recurse a maximum of 600 * 400 times (consider the empty grid initially, and what the depth first search will do, it will first visit 0,0 then 1,0, then 2,0, then 3,0 ... until 599, 0. Then it will go to 599, 1 then 598, 1, then 597, 1, etc. until it reaches 599, 399. This will push 600 * 400 * 2 integers onto the stack in the best case, but depending on what your compiler does it may in fact be more information. Since an integer takes up 4 bytes we will be pushing 1,920,000 bytes of memory onto the stack, which is a good sign we may run into trouble.

We can use the same function definition, and the structure of the function will be quite similar, just we won't use any recursion any more:

```
class node { int x, y; }



int doFill(int x, int y) {

 int result = 0;



 // Declare our stack of nodes, and push our starting node onto the stack

 stack s;

 s.push(node(x, y));



 while (s.empty() == false) {

  node top = s.top();

  s.pop();



// Check to ensure that we are within the bounds of the grid, if not, continue

  if (top.x < 0 || top.x >= 600) continue;

// Similar check for y

  if (top.y < 0 || top.y >= 400) continue;

// Check that we haven't already visited this position, as we don't want to count it twice

  if (fill[top.x][top.y]) continue;



  fill[top.x][top.y] = true; // Record that we have visited this node



  // We have found this node to be empty, and part

  // of this connected area, so add 1 to the result

  result++;



  // Now we know that we have at least one empty square, then we will attempt to

  // visit every node adjacent to this node.

  s.push(node(top.x + 1, top.y));

  s.push(node(top.x - 1, top.y));
```

```
  s.push(node(top.x, top.y + 1));

  s.push(node(top.x, top.y - 1));

 }



 return result;

}
```

As you can see, this function has a bit more overhead to manage the stack structure explicitly, but the advantage is that we can use the entire memory space available to our program and in this case, it is necessary to use that much information. However, the structure is quite similar and if you compare the two implementations they are almost exactly equivalent.

Congratulations, we have solved our first question using a depth first search! Now we will move onto the depth-first searches close cousin the Breadth First search.

If you want to practice some DFS based problems, some good ones to look at are:

TCCC 03 Quarterfinals - Marketing - Div 1 500
TCCC 03 Semifinals Room 4 - Circuits - Div 1 275

Queue

A queue is a simple extension of the stack data type. Whereas the stack is a FILO (first-in last-out) data structure the queue is a FIFO (first-in first-out) data structure. What this means is the first thing that you add to a queue will be the first thing that you get when you perform a pop().

There are four main operations on a queue:

1. Push - Adds an element to the back of the queue
2. Pop - Removes the front element from the queue
3. Front - Returns the front element on the queue
4. Empty - Tests if the queue is empty or not

In C++, this is done with the STL class queue:

```
#include

queue myQueue;
```

In Java, we unfortunately don't have a Queue class, so we will approximate it with the LinkedList class. The operations on a linked list map well to a queue (and in fact, sometimes queues are implemented as linked lists), so this will not be too difficult.

The operations map to the LinkedList class as follows:

1. Push - boolean LinkedList.add(Object o)
2. Pop - Object LinkedList.removeFirst()
3. Front - Object LinkedList.getFirst()
4. Empty - int LinkedList.size()

```
import java.util.*;

LinkedList myQueue = new LinkedList();
```

In C#, we use Queue class:

The operations map to the Queue class as follows:

1. Push - void Queue.Enqueue(Object o)
2. Pop - Object Queue.Dequeue()
3. Front - Object Queue.Peek()
4. Empty - int Queue.Count

```
using System.Collections;

Queue myQueue = new Queue();
```

Breadth First Search

The Breadth First search is an extremely useful searching technique. It differs from the depth-first search in that it uses a queue to perform the search, so the order in which the nodes are visited is quite different. It has the extremely useful property that if all of the edges in a graph are unweighted (or the same weight) then the first time a node is visited is the shortest path to that node from the source node. You can verify this by thinking about what using a queue means to the search order. When we visit a node and add all the neighbors into the queue, then pop the next thing off of the queue, we will get the neighbors of the first node as the first elements in the queue. This comes about naturally from the FIFO property of the queue and ends up being an extremely useful property. One thing that we have to be careful about in a Breadth First search is that we do not want to visit the same node twice, or we will lose the property that when a node is visited it is the quickest path to that node from the source.

The basic structure of a breadth first search will look this:

```
void bfs(node start) {

 queue s;

 s.push(start);

 while (s.empty() == false) {

  top = s.front();

  s.pop();

  mark top as visited;
```

check for termination condition (have we reached the node we want to?) add all of top's unvisited neighbors to the stack.

```
 }

}
```

Notice the similarities between this and a depth-first search, we only differ in the data structure used and we don't mark top as unvisited again.

The problem we will be discussing in relation to the Breadth First search is a bit harder than the previous example, as we are dealing with a slightly more complicated search space. The problem is the 1000 from Division 1 in SRM 156, Pathfinding. Once again we will be dealing in a grid-based problem, so we can represent the graph structure implicitly within the grid.

A quick summary of the problem is that we want to exchange the positions of two players on a grid. There are impassable spaces represented by 'X' and spaces that we can walk in represented by ' '. Since we have two players our node structure becomes a bit more complicated, we have to represent the positions of person A and person B. Also, we won't be able to simply use our array to represent visited positions any more, we will have an auxiliary data structure to do that. Also, we are allowed to make diagonal movements in this problem, so we now have 9 choices, we can move in one of 8 directions or simply stay in the same position. Another little trick that we have to watch for is that the players can not just swap positions in a single turn, so we have to do a little bit of validity checking on the resulting state.

First, we set up the node structure and visited array:

```
class node {
```

```
 int player1X, player1Y, player2X, player2Y;

 int steps; // The current number of steps we have taken to reach this step

}



bool visited[20][20][20][20];
```

Here a node is represented as the (x,y) positions of player 1 and player 2. It also has the current steps that we have taken to reach the current state, we need this because the problem asks us what the minimum number of steps to switch the two players will be. We are guaranteed by the properties of the Breadth First search that the first time we visit the end node, it will be as quickly as possible (as all of our edge costs are 1).

The visited array is simply a direct representation of our node in array form, with the first dimension being player1X, second player1Y, etc. Note that we don't need to keep track of steps in the visited array.

Now that we have our basic structure set up, we can solve the problem (note that this code is not compilable):

```
int minTurns(String[] board) {

 int width = board[0].length;

 int height = board.length;



 node start;

 // Find the initial position of A and B, and save them in start.



 queue q;

 q.push(start);

 while (q.empty() == false) {

  node top = q.front();

  q.pop();



  // Check if player 1 or player 2 is out of bounds, or on an X square, if so continue

  // Check if player 1 or player 2 is on top of each other, if so continue



  // Make sure we haven't already visited this state before

  if (visited[top.player1X][top.player1Y][top.player2X][top.player2Y]) continue;

  // Mark this state as visited

  visited[top.player1X][top.player1Y][top.player2X][top.player2Y] = true;
```

```
  // Check if the current positions of A and B are the opposite of what they were in start.

  // If they are we have exchanged positions and are finished!

  if (top.player1X == start.player2X && top.player1Y == start.player2Y &&

      top.player2X == start.player1X && top.player2Y == start.player1Y)

      return top.steps;



  // Now we need to generate all of the transitions between nodes, we can do this quite easily
using some

  // nested for loops, one for each direction that it is possible for one player to move.  Since we
need

  // to generate the following deltas: (-1,-1), (-1,0), (-1,1), (0,-1), (0,0), (0,1), (1,-1),
(1,0), (1,1)

  // we can use a for loop from -1 to 1 to do exactly that.

  for (int player1XDelta = -1; player1XDelta <= -1; player1XDelta++) {

   for (int player1YDelta = -1; player1YDelta <= -1; player1YDelta++) {

    for (int player2XDelta = -1; player2XDelta <= -1; player2XDelta++) {

     for (int player2YDelta = -1; player2YDelta <= -1; player2YDelta++) {

     // Careful though!  We have to make sure that player 1 and 2 did not swap positions on this
turn

      if (top.player1X == top.player2X + player2XDelta && top.player1Y == top.player2Y +
player2YDelta &&

         top.player2X == top.player1X + player1XDelta && top.player2Y == top.player1Y +
player1YDelta)

        continue;



     // Add the new node into the queue

      q.push(node(top.player1X + player1XDelta, top.player1Y + player1YDelta,

                top.player2X + player2XDelta, top.player2Y + player2YDelta,

                top.steps + 1));

     }

    }

   }

  }

 }
```

```
 // It is not possible to exchange positions, so

 // we return -1.  This is because we have explored

 // all the states possible from the starting state,

 // and haven't returned an answer yet.

 return -1;

}
```

This ended up being quite a bit more complicated than the basic Breadth First search implementation, but you can still see all of the basic elements in the code. Now, if you want to practice more problems where Breadth First search is applicable, try these:

Inviational 02 Semifinal Room 2 - Div 1 500 - Escape

Finding the best path through a graph

An extremely common problem on TopCoder is to find the shortest path from one position to another. There are a few different ways for going about this, each of which has different uses. We will be discussing two different methods, Dijkstra using a Heap and the Floyd Warshall method.

Dijkstra (Heap method)

Dijkstra using a Heap is one of the most powerful techniques to add to your TopCoder arsenal. It essentially allows you to write a Breadth First search, and instead of using a Queue you use a Priority Queue and define a sorting function on the nodes such that the node with the lowest cost is at the top of the Priority Queue. This allows us to find the best path through a graph in O(m * log(n)) time where n is the number of vertices and m is the number of edges in the graph.

Sidenote:
If you haven't seen big-O notation before then I recommend reading this.

First however, an introduction to the Priority Queue/Heap structure is in order. The Heap is a fundamental data structure and is extremely useful for a variety of tasks. The property we are most interested in though is that it is a semi-ordered data structure. What I mean by semi-ordered is that we define some ordering on elements that are inserted into the structure, then the structure keeps the smallest (or largest) element at the top. The Heap has the very nice property that inserting an element or removing the top element takes O(log n) time, where n is the number of elements in the heap. Simply getting the top value is an O(1) operation as well, so the Heap is perfectly suited for our needs.

The fundamental operations on a Heap are:

1. Add - Inserts an element into the heap, putting the element into the correct ordered location.
2. Pop - Pops the top element from the heap, the top element will either be the highest or lowest element, depending on implementation.
3. Top - Returns the top element on the heap.
4. Empty - Tests if the heap is empty or not.

Pretty close to the Queue or Stack, so it's only natural that we apply the same type of searching principle that we have used before, except substitute the Heap in place of the Queue or Stack. Our basic search routine (remember this one well!) will look something like this:

```
void dijkstra(node start) {

 priorityQueue s;

 s.add(start);

 while (s.empty() == false) {
```

```
  top = s.top();

  s.pop();

  mark top as visited;
```

check for termination condition (have we reached the target node?)
add all of top's unvisited neighbors to the stack.

```
 }

}
```

Unfortunately, not all of the default language libraries used in TopCoder have an easy to use priority queue structure.

C++ users are lucky to have an actual priority_queue<> structure in the STL, which is used as follows:

```
#include

using namespace std;

priority_queue pq;

1. Add - void pq.push(type)

2. Pop - void pq.pop()

3. Top - type pq.top()

4. Empty - bool pq.empty()
```

However, you have to be careful as the C++ priority_queue<> returns the *highest* element first, not the lowest. This has been the cause of many solutions that should be O(m * log(n)) instead ballooning in complexity, or just not working.

To define the ordering on a type, there are a few different methods. The way I find most useful is the following though:

```
Define your structure:

struct node {

 int cost;

 int at;

};
```

And we want to order by cost, so we define the less than operator for this structure as follows:

```
bool operator<(const node &leftNode, const node &rightNode) {

 if (leftNode.cost != rightNode.cost) return leftNode.cost < rightNode.cost;

 if (leftNode.at != rightNode.at) return leftNode.at < rightNode.at;

 return false;
```

```
}
```

Even though we don't need to order by the 'at' member of the structure, we still do otherwise elements with the same cost but different 'at' values may be coalesced into one value. The return false at the end is to ensure that if two duplicate elements are compared the less than operator will return false.

Java users unfortunately have to do a bit of makeshift work, as there is not a direct implementation of the Heap structure. We can approximate it with the TreeSet structure which will do full ordering of our dataset. It is less space efficient, but will serve our purposes fine.

```
import java.util.*;

TreeSet pq = new TreeSet();



1. Add - boolean add(Object o)

2. Pop - boolean remove(Object o)
```

In this case, we can remove anything we want, but pop should remove the first element, so we will always call it like

```
this: pq.remove(pq.first());

3. Top - Object first()

4. Empty - int size()
```

To define the ordering we do something quite similar to what we use in C++:

```
class Node implements Comparable {

 public int cost, at;



 public int CompareTo(Object o) {

  Node right = (Node)o;

  if (cost < right.cost) return -1;

  if (cost > right.cost) return 1;

  if (at < right.at) return -1;

  if (at > right.at) return 1;

  return 0;

 }

}
```

C# users also have the same problem, so they need to approximate as well, unfortunately the closest thing to what we want that is currently available is the SortedList class, and it does not have the necessary speed (insertions and deletions are O(n) instead of O(log n)). Unfortunately there is no suitable built-in class for implementing heap based algorithms in C#, as the HashTable is not suitable

either.

Getting back to the actual algorithm now, the beautiful part is that it applies as well to graphs with weighted edges as the Breadth First search does to graphs with un-weighted edges. So we can now solve much more difficult problems (and more common on TopCoder) than is possible with just the Breadth First search.

There are some extremely nice properties as well, since we are picking the node with the least total cost so far to explore first, the first time we visit a node is the best path to that node (unless there are negative weight edges in the graph). So we only have to visit each node once, and the really nice part is if we ever hit the target node, we know that we are done.

For the example here we will be using KiloManX, from SRM 181, the Div 1 1000. This is an excellent example of the application of the Heap Dijkstra problem to what appears to be a Dynamic Programming question initially. In this problem the edge weight between nodes changes based on what weapons we have picked up. So in our node we at least need to keep track of what weapons we have picked up, and the current amount of shots we have taken (which will be our cost). The really nice part is that the weapons that we have picked up corresponds to the bosses that we have defeated as well, so we can use that as a basis for our visited structure. If we represent each weapon as a bit in an integer, we will have to store a maximum of 32,768 values (2^15, as there is a maximum of 15 weapons). So we can make our visited array simply be an array of 32,768 booleans. Defining the ordering for our nodes is very easy in this case, we want to explore nodes that have lower amounts of shots taken first, so given this information we can define our basic structure to be as follows:

```
boolean visited[32768];


class node {
 int weapons;
 int shots;
 // Define a comparator that puts nodes with less shots on top appropriate to your language
};
```

Now we will apply the familiar structure to solve these types of problems.

```
int leastShots(String[] damageChart, int[] bossHealth) {
 priorityQueue pq;

 pq.push(node(0, 0));

 while (pq.empty() == false) {
  node top = pq.top();
  pq.pop();

  // Make sure we don't visit the same configuration twice
  if (visited[top.weapons]) continue;
  visited[top.weapons] = true;
```

```
  // A quick trick to check if we have all the weapons, meaning we defeated all the bosses.
  // We use the fact that (2^numWeapons - 1) will have all the numWeapons bits set to 1.
  if (top.weapons == (1 << numWeapons) - 1)
    return top.shots;

  for (int i = 0; i < damageChart.length; i++) {
    // Check if we've already visited this boss, then don't bother trying him again
    if ((top.weapons >> i) & 1) continue;

    // Now figure out what the best amount of time that we can destroy this boss is, given the weapons we have.
    // We initialize this value to the boss's health, as that is our default (with our KiloBuster).
    int best = bossHealth[i];
    for (int j = 0; j < damageChart.length; j++) {
     if (i == j) continue;
     if (((top.weapons >> j) & 1) && damageChart[j][i] != '0') {
      // We have this weapon, so try using it to defeat this boss
      int shotsNeeded = bossHealth[i] / (damageChart[j][i] - '0');
      if (bossHealth[i] % (damageChart[j][i] - '0') != 0) shotsNeeded++;
      best = min(best, shotsNeeded);
     }
    }

    // Add the new node to be searched, showing that we defeated boss i, and we used 'best' shots to defeat him.
    pq.add(node(top.weapons | (1 << i), top.shots + best));
  }
 }
}
```

There are a huge number of these types of problems on TopCoder; here are some excellent ones to try out:

SRM 150 - Div 1 1000 - RoboCourier
SRM 194 - Div 1 1000 - IslandFerries
SRM 198 - Div 1 500 - DungeonEscape
TCCC '04 Round 4 - 500 - Bombman

Floyd-Warshall

Floyd-Warshall is a very powerful technique when the graph is represented by an adjacency matrix. It runs in O(n^3) time, where n is the number of vertices in the graph. However, in comparison to Dijkstra, which only gives us the shortest path from one source to the targets, Floyd-Warshall gives us the shortest paths from all source to all target nodes. There are other uses for Floyd-Warshall as well; it can be used to find connectivity in a graph (known as the Transitive Closure of a graph).

First, however we will discuss the Floyd Warshall All-Pairs Shortest Path algorithm, which is the most similar to Dijkstra. After running the algorithm on the adjacency matrix the element at adj[i][j] represents the length of the shortest path from node i to node j. The pseudo-code for the algorithm is given below:

```
for (k = 1 to n)

 for (i = 1 to n)

  for (j = 1 to n)

   adj[i][j] = min(adj[i][j], adj[i][k] + adj[k][j]);
```

As you can see, this is extremely simple to remember and type. If the graph is small (less than 100 nodes) then this technique can be used to great effect for a quick submission.

An excellent problem to test this out on is the Division 2 1000 from SRM 184, TeamBuilder.