Introduction to Graphs and Their Data Structures: Section 3

Section 3: Finding the Best Path through a Graph

By **gladius** — *topcoder member*

…read Section 2

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 lang
};
```

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 bos
  // 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, giv
   // We initialize this value to the boss's health, as that is our default (with our K
   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 'be
   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.

More Resources

Member Tutorials
Read more than 40 data science tutorials written by topcoder members.

Problem Set Analysis
Read editorials explaining the problem and solution for each Single Round Match (SRM).

Data Science Guide
New to topcoder's data science track? Read this guide for an overview on how to get started in the arena and how competitions work.

Help Center
Need specifics about the process or the rules? Everything you need to know about competing at topcoder can be found in the Help Center.

Member Forums
Join your peers in our member forums and ask questions from the real experts - topcoder members!