Grid Developer Challenge

Callum Pearce

November 19, 2017

1 Running the Solution

The solution was written in Java, therefore ensure that you have an up to date version of JDK (Java SE development kit). Whilst developing I was running javac 1.8.0_131 on OSX, you can easily check your version with:
$javac - version$.

1.1 Compilation

Run the command below in the directory where the EventsFinder.java file is located in order to compile the solution:
$javac\ EventsFinder.java$

1.2 Run the solution

To run the solution the following command with the listed parameters is used:
$java\ EventsFinder$ [$DIMENSIONS$] [$EVENTS\_COUNT$]

- [$DIMENSIONS$]: The dimensions of the grid. Set to 21 for challenge to produce a 21x21 grid of nodes ranging from x,y: -10 to $+10$
- [$EVENTS\_COUNT$]: The number of closest events to found and printed. Set to 5 for challenge.

Use the following for the challenges direct solution: $java\ EventsFinder$ 21 5

Next you can enter in the x and y coordinates you wish to search from.

2 Approach to Challenge

With the task of building a world represented by a grid which has multiple properties, I decided to use Java (an object oriented programming language) as these properties can be broken down into objects allowing me to write intuitive code. A lower level language such as C would have resulted in many more lines of code to achieve the same quality program, making it harder to grasp what the program does as a single entity. Whilst using a higher level scripting based language like Python would not express my solution in as much detail as I was aiming for (also the program would likely be slower at runtime).

3 Assumptions

- The world is always square i.e. dimensions $x = y$. In order to create a simpler program which directly meets the challenges requirements.
- Events are randomly distributed across the world, where a given node on the grid has a probability of $p(E) = 0.15$ for containing an event.
- Every event has 500 tickets which have prices ranging from \$0.1 to \$100.
- The world has a even scale on both the negative and positive axis, meaning the dimensions of the grid have to be odd. For example to make a -10 to $+10$ grid a dimension of 21 must be given as there are 21 points from -10 to $+10$.

- If two events are of equal distance from the input coordinates there is no preference given to which event is printed first (or printed at all). The first one discovered by $findClosestEvents$ will be the first one to be printed.

4 Supporting Multiple events at the same location

The solution was written in such a way supporting multiple events would be a relatively simple change. Instead of a $Node$ object containing a single $Event$ object a list of $Event$ objects could be stored. Then whenever a nodes $Event$ is referenced instead iterate over all of the $Node$s $Event$s e.g. in the $findClosestEvents$ method for a $Node$ containing multiple $Event$s it will iterate through all of that $Node$s $Event$s.

5 Working with a much larger world

Note the solution supports changing the size of the world so different size world can be experimented with is you wish. The algorithm to find the closest 5 events is an implementation of Breadth First Search which is an efficient graph exploration algorithm with a complexity of $O(|V| + |E|)$, where $|V|$ is the number of vertices in the grid and $|E|$ is the number edges in the grid. Therefore the points below should instead be focused on in order to improve the programs performance on a very large world:

- **Concurrently generate grid world and its events** by using threading in Java. By using an design pattern where a farmer and many worker threads the board can split into sections for each thread to generate, then sync all threads once each one has generated its required section of the world.
- **Use a profiler such as JProfiler** to analyse whether the code is memory bandwidth performance bound or computational performance bound. Then if we find the program is memory bandwidth bound (most common with modern architectures) we can manipulate data types (e.g. change Integer to short where viable) so less data needs to be loaded from lower down the memory hierarchy. Whilst for computationally bound the program could be modified to store calculation results and reuse them across loop iteration such as that in generating random ticket prices.
- **Store the grid world in binary form** where 1 represents a $Node$ an event and 0 represents an empty $Node$. Then at when searching for closest events when 1 is found only then generate the event and its data and tickets. This will reduce the runtime by a large amount as far less memory is required due to a lower amount of events generated, which in turn reduces the memory demand for the program as far less events will be stored in memory.