Physics 4250
University of Manitoba
April 30, 2017

EFFICIENT PATH OPTIMIZATION WITH QUADTREE PARTITIONING

GENE HARVEY, COURNEY BONNER, ADNAN HAFEEZ

Quadtree Data Structure

STRATEGY

We localize and isolate obstacles on a map and turn continuous empty space into a grid that can be treated as a graph. A data structure called a quadtree is used.

ADVANTAGES OF QUADTREE

Once the tree is constructed, the algorithm removes all lowest level children and converts each one into a node on a graph. This effectively discretizes an otherwise continuous distribution. A list for each quadrant containing its adjacent nodes is created.

The tree contains all the information we need to find the shortest path between any two points. The information of all the nodes is output to the Python file.

CHALLENGES

However, we had to find a tradeoff between memory efficiency and temporal efficiency. In our project, we choose CPU efficiency. In the end the memory usage was $O(4^{n^2})$ in the worst case. In reality it is likely better than this as this assumes that each node has an adjacency list with size on the order of the number of nodes, but it is unlikely that a node has that many nodes adjacent to it.

To try to offset the memory issues, we have some control variables in the code that can be set between runs. These can be found on lines 63 and 64 or can be set via command line arguments.

1. actual_to_max_children_ratio is a guess for what percentage of the maximum number of nodes will actually be created. By default it is 0.75. The threshold value has little effect on what this variable should be set to, it is better determined by the map.
2. max_num_adjacent is a guess for the maximum number of adjacent nodes any given node will have. It will have to be larger for larger threshold values. This value is 30 by default.

The program exits and notifies the user if either of these values is set too low. It was difficult to implement an efficient handler for when these values are set too high. This is because the result is NULL pointers being created when we try to allocate memory, which happens in many places throughout the program. What we needed to implement was a signal handler but as these must interface directly with the operating system we experienced problems when we tried to run on our Windows machines. If the C program crashes the likely cause is that these control variables are set too high.

BENEFITS OF QUADTREE

The complexity of the algorithm does not scale with the size of the map, but only with threshold value. Also, it was quick to create at a reasonable level of accuracy and contained all the information we needed to calculate any path.

Variation of Dijkstra's algorithm: A*

DIJKSTRA'S ALGORITHM

This portion of our code, written in Python, was to take the graph outputted by the C program and calculate the shortest path between any and all points on the graph. A variation of Dijkstra's algorithm, A* was used. A* evaluates the cost of travelling between two nodes bit differently than Dijkstra's but they are otherwise identical.

In Dijkstra's algorithm, the actual cost is used, we will refer to this as c(x,y) from now on (x and y being the two nodes we are evaluating the cost between)

A*

In the A* algorithm, a heuristic value is added to the actual cost to help speed up the search. Let's call this h(x,y). In our case, we used the Cartesian distance between the two points so that the algorithm would tend to travel in the direction of the end point. Essentially, this eliminates less optimal paths as possibilities sooner, speeding up the search.

CHALLENGES

When finding the shortest path between all points, the algorithm was still quite slow. So, we had to implement multithreading. We expected this change to speed up our program quite a bit but did not see results as good as we hoped for. After adding in multithreading to our python program we learned about the global interpreter lock which only allows one thread to execute at a time. If time had permitted we would have implemented the multithreading in C.

IMPROVEMENTS

The paths are currently found recursively, which is usually not as fast as an iterative algorithm. If time would have permitted it might have been worth our time to rewrite the algorithm iteratively so see how much of a speedup we could get. Also as mentioned before we would have translated our multithreaded Python program into C so we could see the effect of real multithreading.

RUNNING INSTRUCTIONS

We include a short demo file for a cursory view of our project. You can run this by double-clicking IAmADemo.bat. The demo will compile our quadtree, build the path optimization hashtable, and randomly find several paths in the distribution.

To use a custom distribution, create a map text file and place in PythonWorkspace/pathfinder/QTData. Then, run CreateQT.exe with options

>>> CreateQT.exe [mapfile].txt [qt_threshold] [actual_to_max_children_ratio] [max_num_adjacent]

Now start the python interpreter in the PythonWorkspace/pathfinder directory and run the command

>>> from loadscript import *

This loads all modules that we need. To generate the path optimization hashtable run the command

>>> pack,pdict = qt.findAllPaths()

This returns a graph representation of the quadtree, the figure (both in pack), and the path optimization hashtable. We then can use these results to find the fastest path between two points with

>>> line = qt.plotPathWithResults(pack, pdict, [starting point tuple], [ending point tuple])

Then in subsequent calls we can replace the line generated with

>>> line = qt.plotPathWithResults(pack, pdict, [starting point tuple], [ending point tuple], line)

Further instruction is available in module documentation.

Dynamic Programming

DIJKSTRA'S ALGORITHM

This portion of our code, written in Python, was to take the graph outputted by the C program and calculate the shortest path between any and all points on the graph. A variation of Dijkstra's algorithm, A* was used. A* evaluates the cost of travelling between two nodes bit differently than Dijkstra's but they are otherwise identical.

In Dijkstra's algorithm, the actual cost is used, we will refer to this as c(x,y) from now on (x and y being the two nodes we are evaluating the cost between)