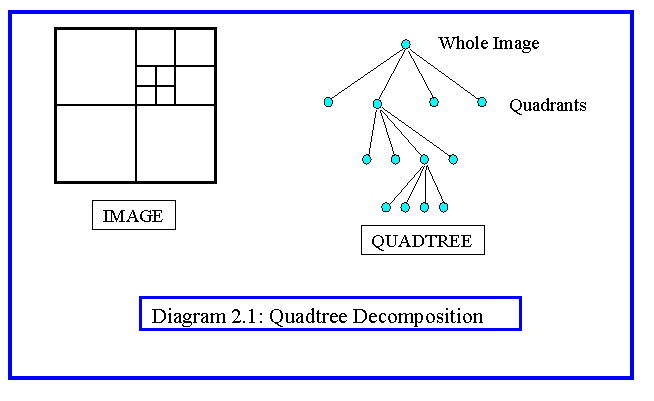
**Purpose**

To localize and isolate obstacles from a map and turn continuous empty space into a grid that can be treated as a graph.

**Method**

A data structure called a quadtree was used to accomplish this. A map is input to the program and it is repeatedly divided into 4 quadrants until either a specified number of divisions have occurred or the current quadrant does not contain any objects.



Source: <https://www.google.ca/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0ahUKEwjXlMjZzq_TAhVB2IMKHbAXDfsQjRwIBw&url=http%3A%2F%2Fwww.doc.ic.ac.uk%2F~dfg%2Fvision%2Fv02.html&psig=AFQjCNGQCrXVv0hpbaO_VGXjX-RU6qQ_hA&ust=1492660161748473>

The image on the left is a visual representation of what is being done to the map. All of the quadrants except the top right are not divided any further because they don’t contain objects. The image on the right is the tree (a type of data structure) that is formed by the program.

Once the tree is constructed the program removes all of the lowest level children of the tree (the above example would have 10) and constructs a list for each one containing its adjacent nodes (a node is a square on the tree). Now a graph has been constructed from what was originally a non-discrete map making pathfinding a computationally easier problem.

The advantage to a quadtree is that once it is constructed, we have all the information we need to find the shortest path between any two points on the graph.

In summary, we used a quadtree to turn a continuous space into a discrete graph. The information about the location and size off all of the nodes and each node’s adjacent nodes is output to a file that is used in Gene’s Python program to do the actual pathfinding.

**Challenges**

A trade-off had to be made between memory efficiency and temporal efficiency. This program uses memory on the order O(4n^2) in order to make the program run as quickly as possible. With my 16 GB of RAM this hasn’t been much of an issue but if memory were a larger concern than speed some modifications would have to be made. However, any modifications to conserve memory would almost certainly increase the time of execution drastically.

To try and offset the memory issues some control variables are implemented:

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. If they are too high (causing memory flow) was more difficult to handle, no handler for this has been implemented and this is the most likely cause for the program to crash.

**Successes**

1. The complexity of the algorithm does not scale with the size of the map but only with threshold value.
2. \*\*\* Will add more once more testing with more maps can be done.