Introduction
ANA* has proved to be extremely efficient in finding a solution through a very few sets of “improve solution” iterations. The real power of this algorithm is evident in higher difficulties: where a sufficiently optimal path has been found without exploring the entire workspace. Notice how there are still large white (unexplored) areas all over the hard and very hard maps.

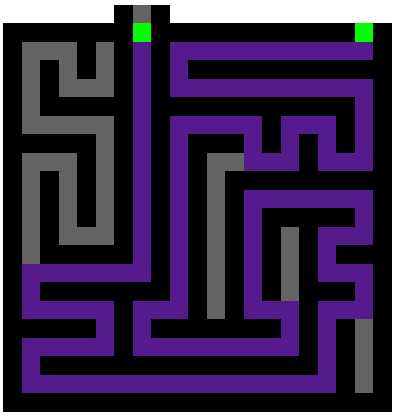
Algorithm
ANA* does guarantee and optimal and complete path (if one exists) and here is why: as we explore each node from the open list, this node is either part of the optimal solution (and its g cost is already optimal) or it is not. If it is, we keep it in the open list until we explore the rest of its neighbors and find the next neighbor that is also on the optimal path with optimal g-cost. If it is not then we either prune the node (if g+h> current actual cost of suboptimal path) or keep it in the open list until its optimal g score is calculated or all its neighbors are visited and none of them are along the optimal path either. We keep doing this until we reach the goal node: this ensures completeness. What ensures optimality is allowing the algorithm to run until the epsilon term (suboptimality bound) is sufficiently small or is not changing by much from iteration to iteration. Kind of similar to a PID controller where a "steady state error" (i.e. minimal or no change in error state) indicates that an equilibrium has been reached.

The following are some of the vital methods and functions used in ANA*:

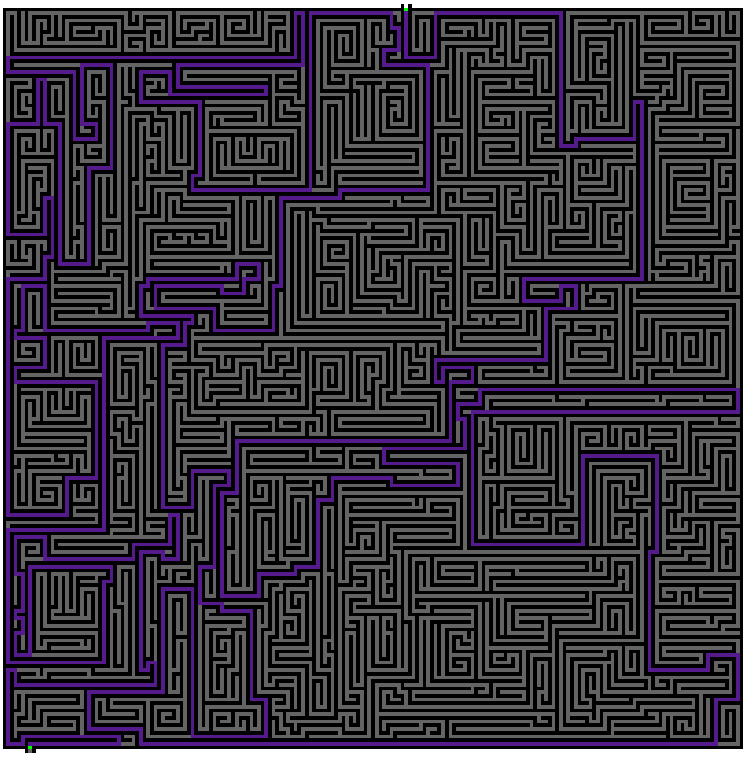
- A custom variant of the priority queue data structure is needed to “flip” the queue since we want to order it by the highest e_s value (that is to say: the smallest combination of cost so far and distance to go estimate)
- Heuristic cost function (can use Manhattan distance, I used eucledian)
- Traverse nodes function to tell the algorithm what are the dynamics and boundaries of its motion (can we move in all directions or only 4 connected, etc.)
- Prune function to remove nodes with a higher cost than any on the frontier and to remove nodes from the frontier that have been explored. This function can be paired with a place function but most lists and queues on python have built in append/put methods that can be used directly
- Visualize method which harbors all the Python Imaging Library methods to load an image, read pixels, convert colors, and display a window of the solution

Results

TRIVIAL

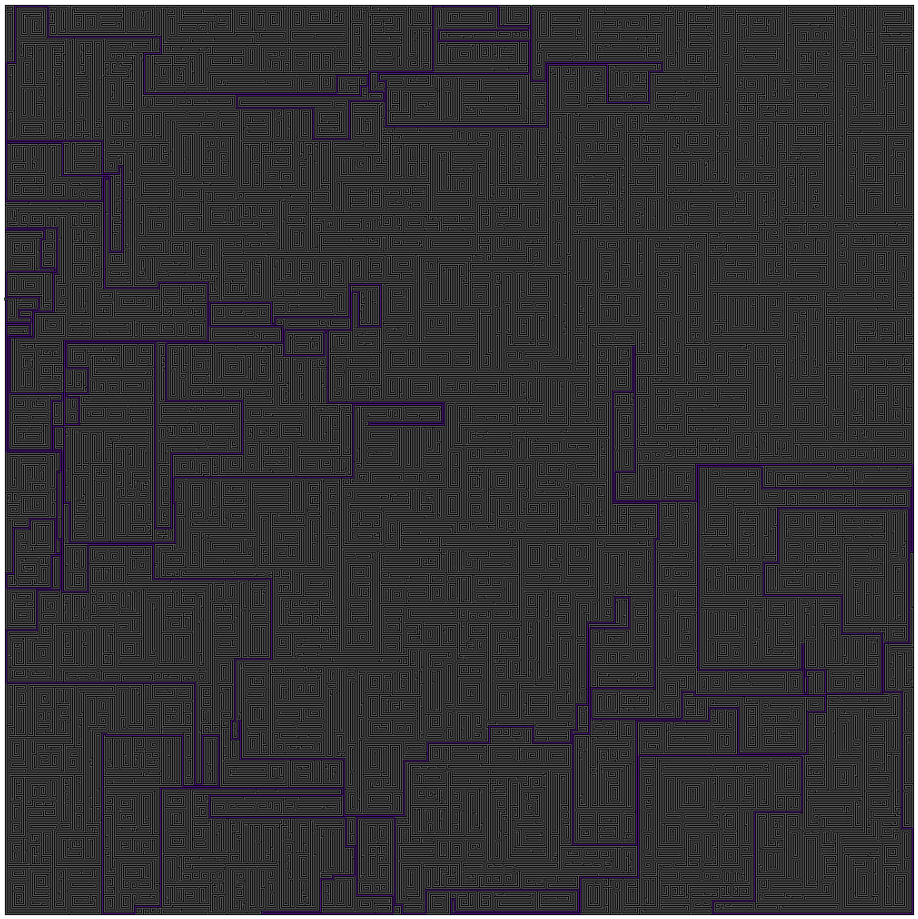


MEDIUM



HARD

VERY HARD



Finally, while all complexities took only two iterations of “improve solution”, the execution time grew exponentially as we went from low levels of complexity to higher levels: