Planner Summary

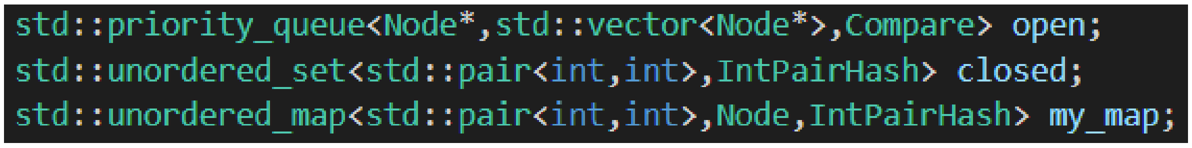
Algorithm Implemented

I Implemented the A* algorithm with no inflation. For my heuristic, I have used the euclidean distance from my chosen location target. The algorithm begins by computing a least-cost path to the final position of the target. At the same time, I have modified the A* algorithm to ensure that it does not search for paths that cannot be reached before the target disappears. I have done this by storing at every node a metric to measure the number of actions taken to reach that node. If this number exceeds the number of actions the target has left, then I simply stop expanding those nodes. This makes the algorithm faster by preventing the expansion of nodes that would be infeasible in any case.
Once a base path is computed to the final position of the target at each timestep after that I compute a path to the previous position of the target from the current robot position and check if this path is more efficient than the current best path that I have computed. I check this by storing the best g_value of the path I have found and the cost incurred. If the new cost is lower than these two combined, I change my path to the new one. Otherwise, I stick to my old path. In this alternative path-finding, I also keep a track of the time of computation that has elapsed and end the planning phase if it goes over 0.95 seconds. This ensures that I do not miss any chance to make a step.
To handle edge cases where the robot is just waiting at its final location to intercept the target, I keep outputting that location.

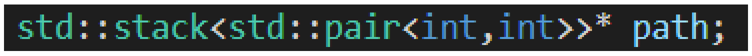
Data structures Used

I have used the following data structures in my algorithm:

Name	Data Structure Used
Open List	Priority Queue
Closed List	Unordered Set
Storage Container for nodes traversed	Unordered Map
Path	Global Stack

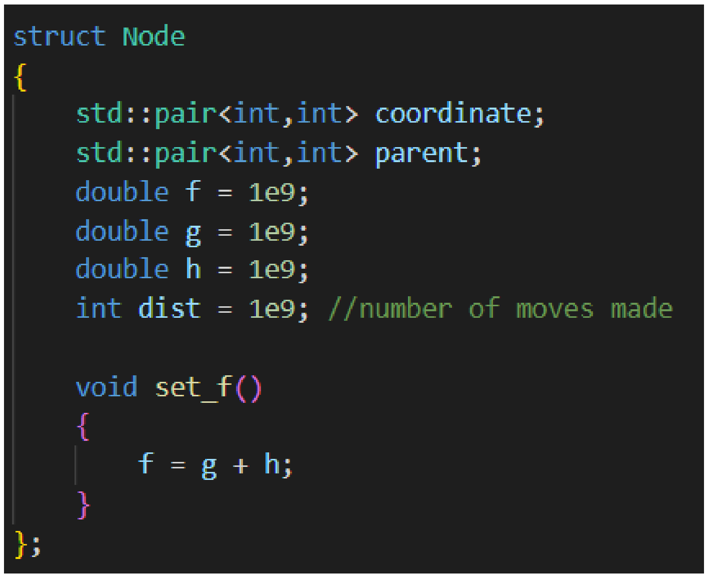


```
std::priority_queue<Node*,std::vector<Node*>,Compare> open;
std::unordered_set<std::pair<int,int>,IntPairHash> closed;
std::unordered_map<std::pair<int,int>,Node,IntPairHash> my_map;
```



```
std::stack<std::pair<int,int>>* path;
```

Node

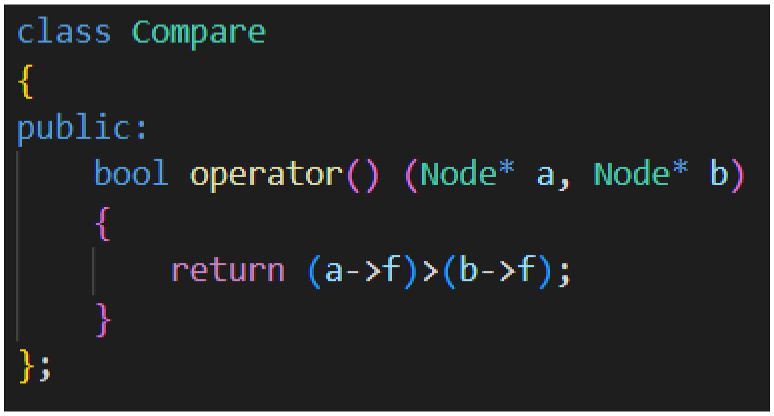


```
struct Node
{
    std::pair<int,int> coordinate;
    std::pair<int,int> parent;
    double f = 1e9;
    double g = 1e9;
    double h = 1e9;
    int dist = 1e9; //number of moves made

    void set_f()
    {
        f = g + h;
    }
};
```

My node is a struct to store the “f”, “g” and “h” values of a given coordinate. For backtracking purposes, I also store the coordinates of the parent of that node, and I have stored a copy of the node’s coordinates within the node. Finally, the dist is the number of moves required to reach said node from the starting position.

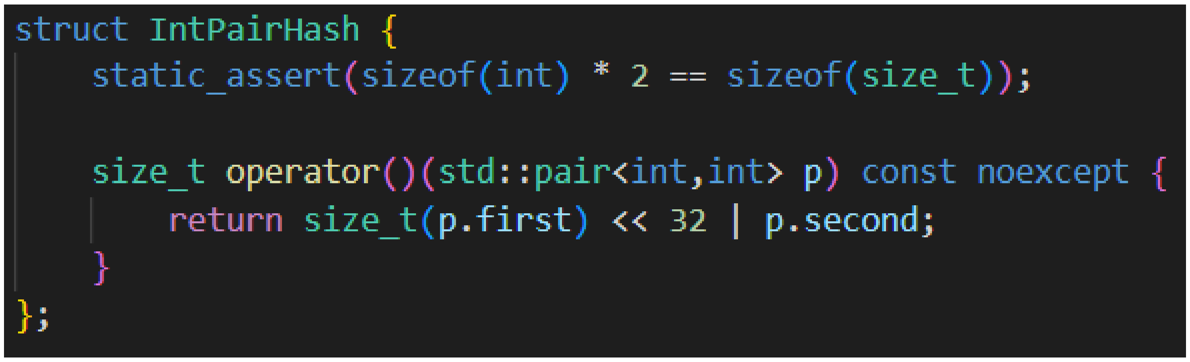
Custom Comparator



```
class Compare
{
public:
    bool operator() (Node* a, Node* b)
    {
        return (a->f)>(b->f);
    }
};
```

I wrote a custom comparator function to allow for sorting of my open list by the “f” values of the nodes.

Hashing Function



```cpp
struct IntPairHash {
    static_assert(sizeof(int) * 2 == sizeof(size_t));

    size_t operator()(std::pair<int,int> p) const noexcept {
        return size_t(p.first) << 32 | p.second;
    }
};
```

Since I am storing my coordinates as pairs, I had to write a custom hashing function for my maps. This is the hashing function that I am using, and it is giving me good performance. I tried using boost as well, but on the TA's advice, I switched it back to a regular function.

Open List

My open list contains pointers of type Node. Since I am using a custom struct, I have also defined a custom compare function for sorting my nodes by their F values. Since priority queues are heaps, insertion is O(log n), and retrieval is O(1) making the algorithm very efficient.

Closed List

My closed list is an unordered set since all I need to check is if a node has been closed before which can be done using its coordinates. Using an unordered set allows this operation to be O(1).

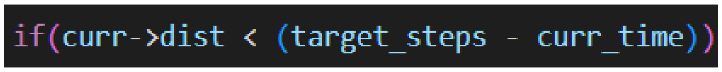
Nodes Traversed

I store the nodes traversed in an unordered map. This allows me to check if a node has been visited before and to only update its "f" and "g" values if they are lower i.e indicating that a better path has been found to the node. This reduces a lot of repeated nodes from being added to my open list, and since my open list is just a bunch of pointers, it also updates the values internally.

Efficiency

- At the end of every iteration all the stored data is deleted (as the robot has moved). This prevents any memory leak, and the algorithm is able to keep running efficiently. As all the nodes are generated on the stack deallocation happens automatically at the end of every iteration. Overall in all my tests, the algorithm needed no more than 1.5 GB of memory even for the largest maps.
- Since the problem specifies that the robot can take one "legal" step per second I have made the traversal cost of all directions equal to 1. This naturally makes diagonal movement much more preferable whenever available, and we can use this to gain on the target effectively.
- I artificially prevent A* from searching any nodes that would be too far to reach. This allows the algorithm to search for the optimal path while sticking to the constraint to

catch the target within a certain number of steps.



```
if(curr->dist < (target_steps - curr_time))
```

- To obtain an efficient path, I interleave planning and execution. I first plan a path to the last position of the goal as a backup, and then I keep checking for each previous position of the goal. This allows me to keep finding and moving on efficient paths and distributes my computational load over multiple time steps while ensuring that too much time is not spent on computation which would make it impossible to catch the target.

Guarantees

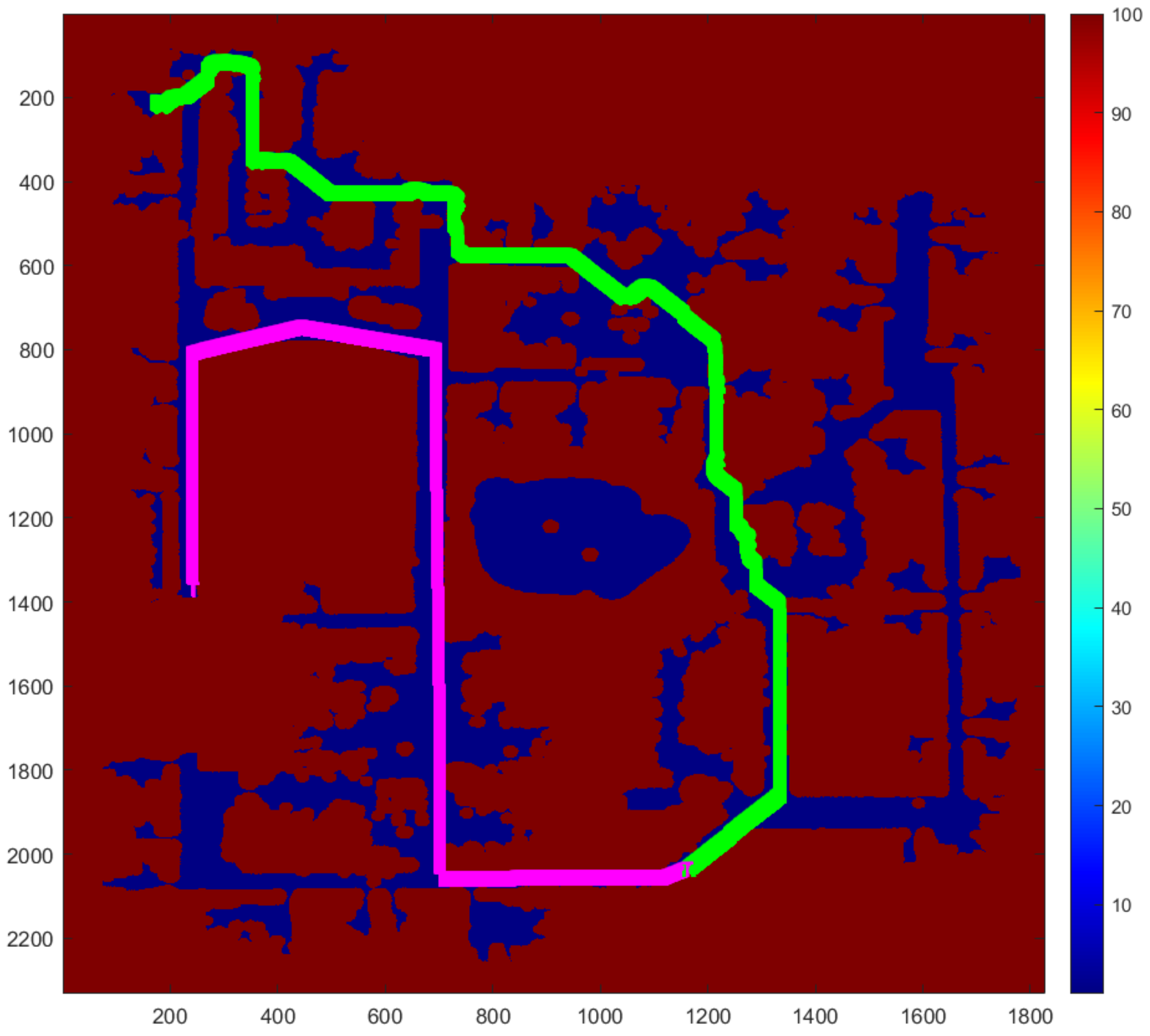
- As I first plan a path to the last position of the target, the algorithm is guaranteed to find a path if one exists.
- From the memory usage I saw and the fact that all path computations happened in well under a second even for map 1, I am confident that this algorithm can scale to larger environments.
- Any path the algorithm computes is guaranteed to be the optimal path to that coordinate because of the underlying guarantees of A*. What this does not guarantee however is that the global optimal trajectory to catch the target would be found. The algorithm might end up going to a point that is not the most optimal location to catch the target, but it is guaranteed to take the optimal path to that coordinate.

Compilation Instructions

Running the following command should compile all of the code.
“mex planner.cpp”

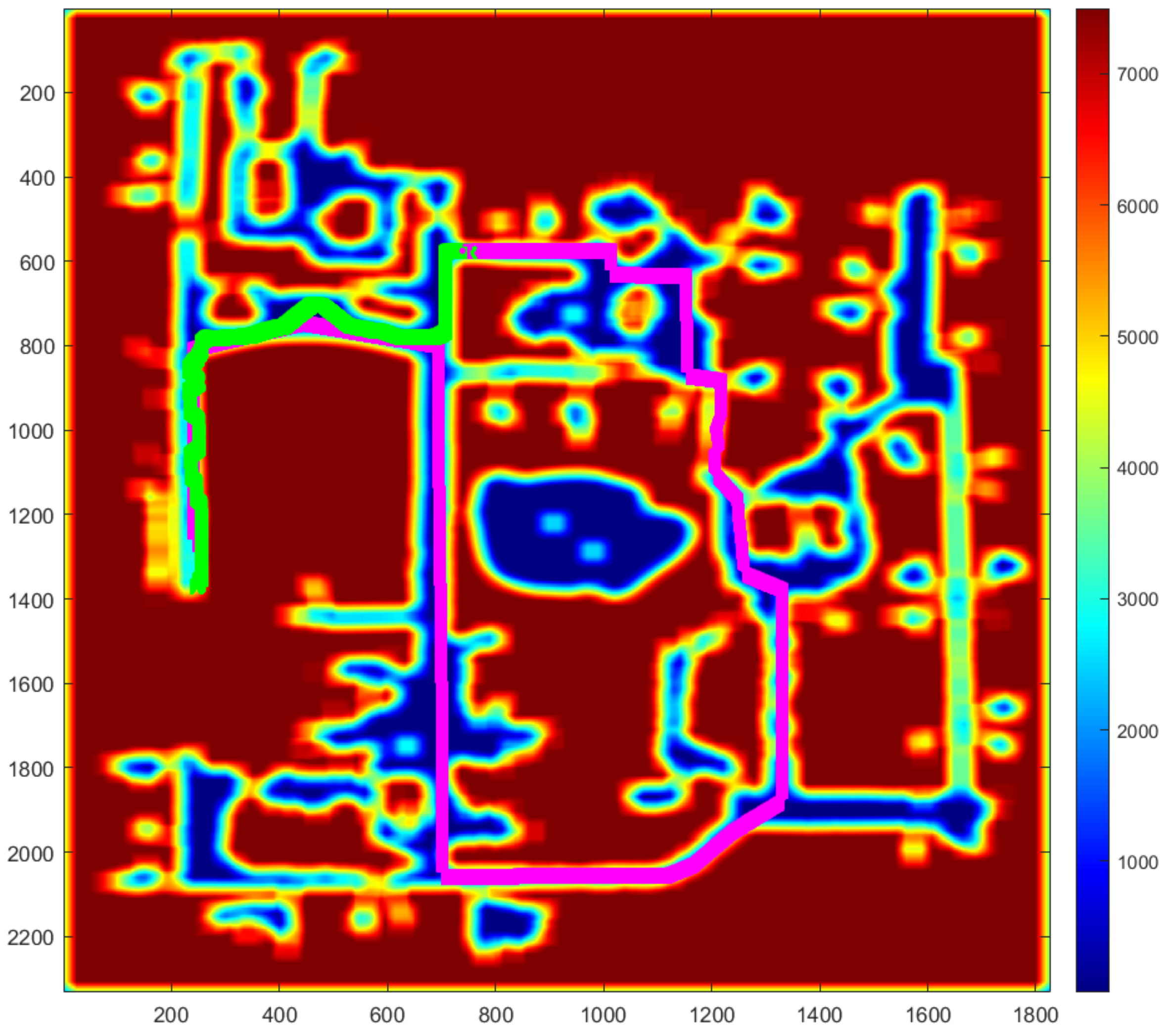
Results

Map 1



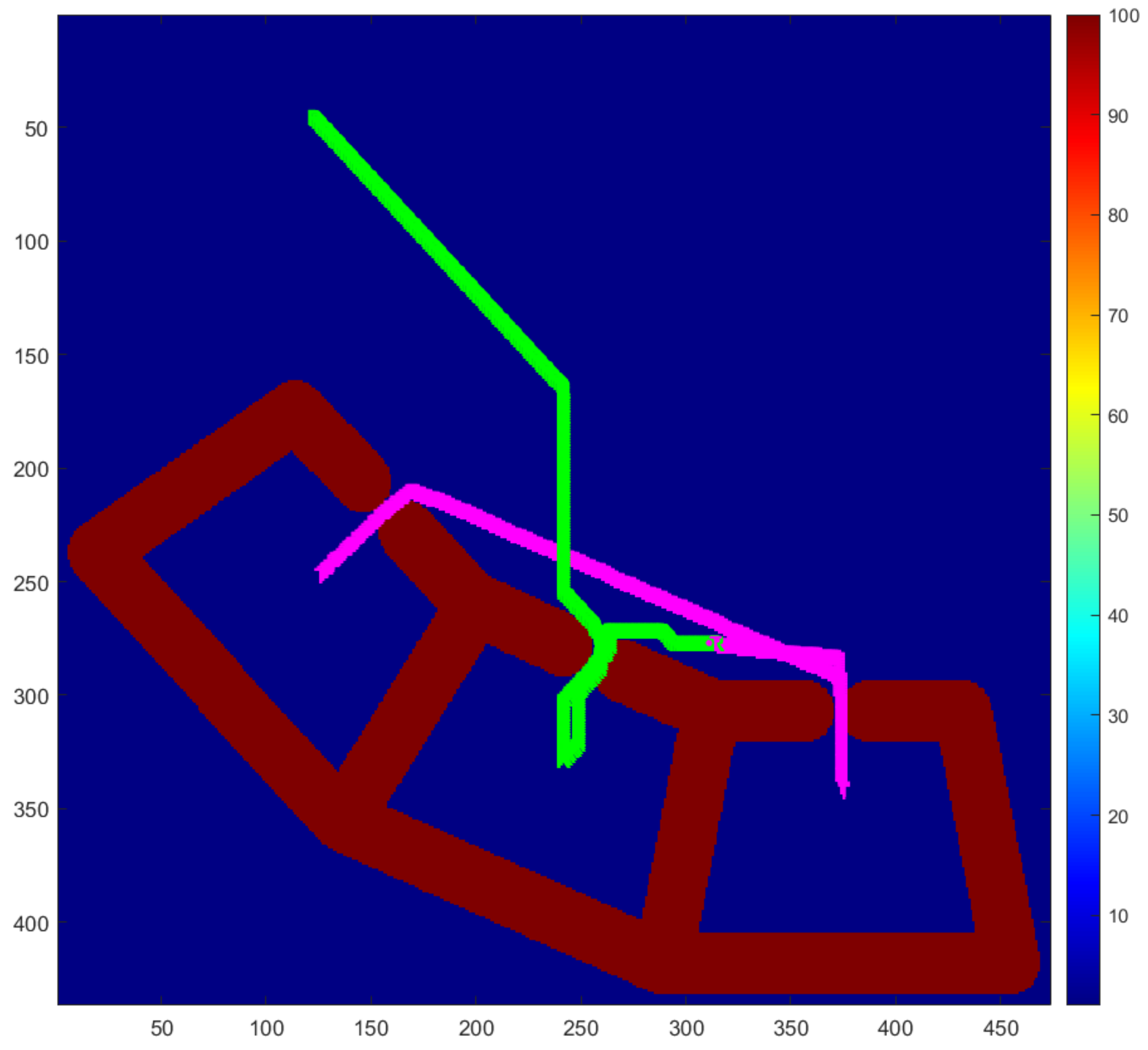
target caught = 1
time taken (s) = 2740
moves made = 2671
path cost = 2740

Map 2



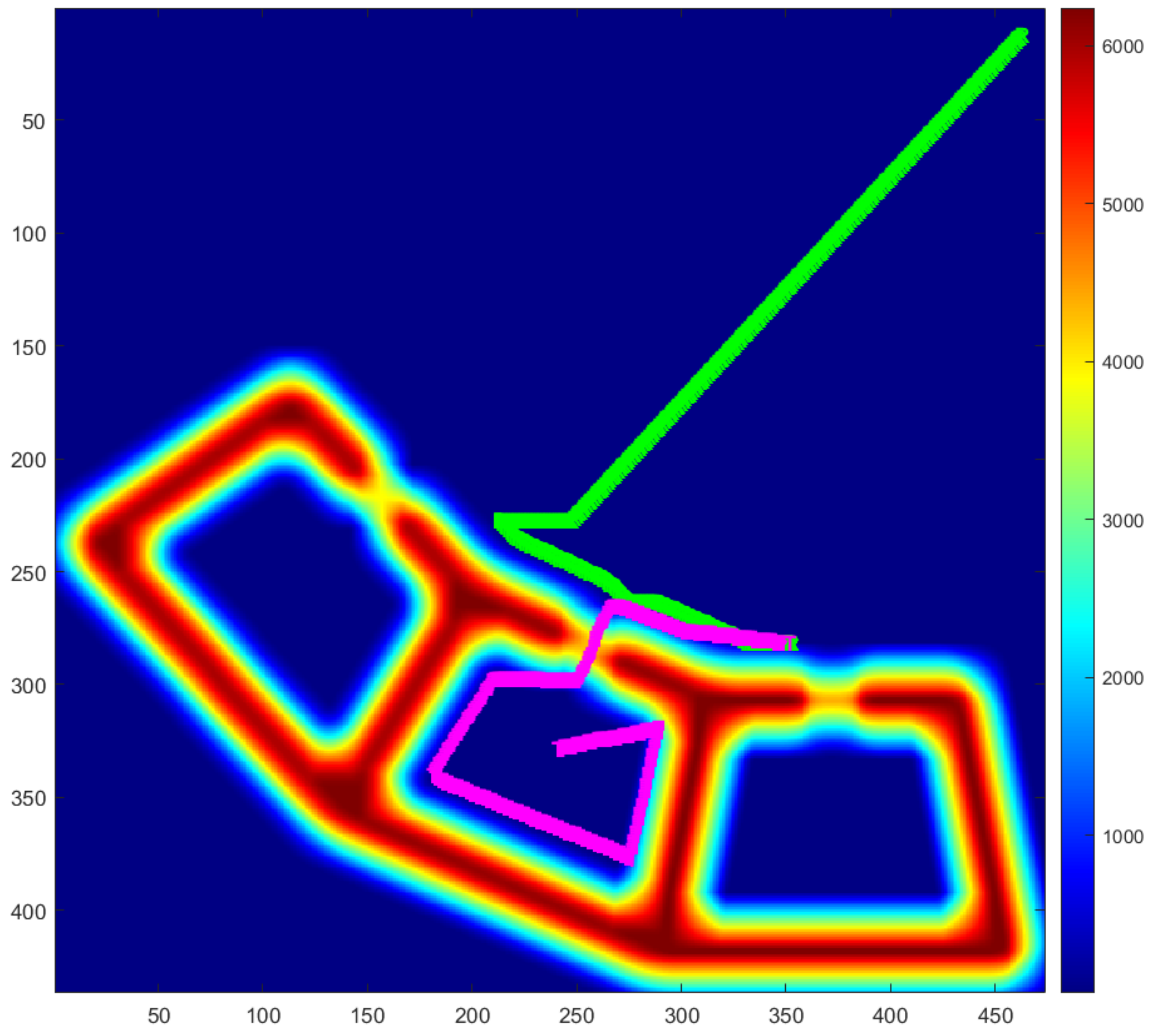
target caught = 1
time taken (s) = 4644
moves made = 1264
path cost = 11565801

Map 3



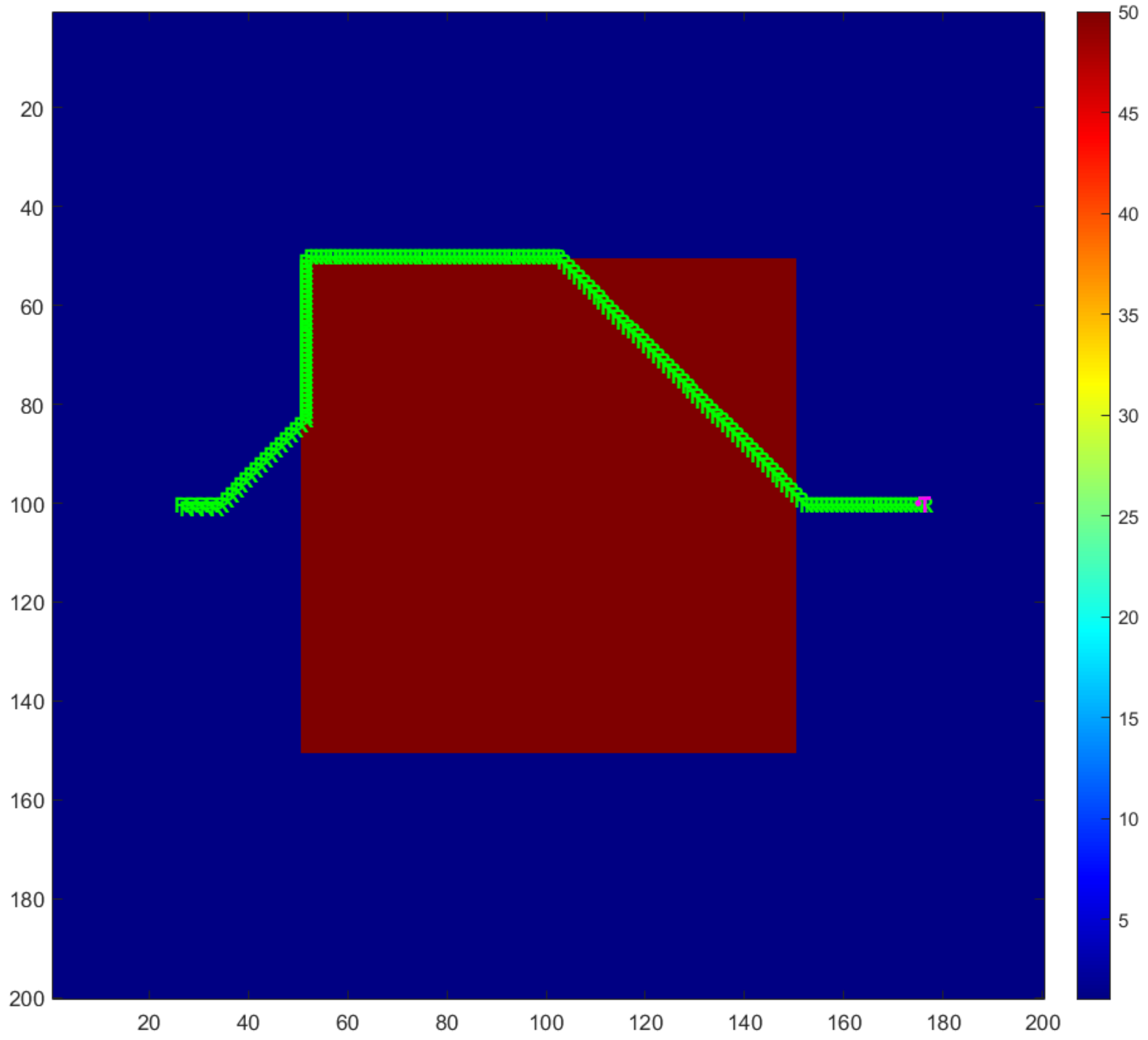
target caught = 1
time taken (s) = 423
moves made = 391
path cost = 423

Map 4



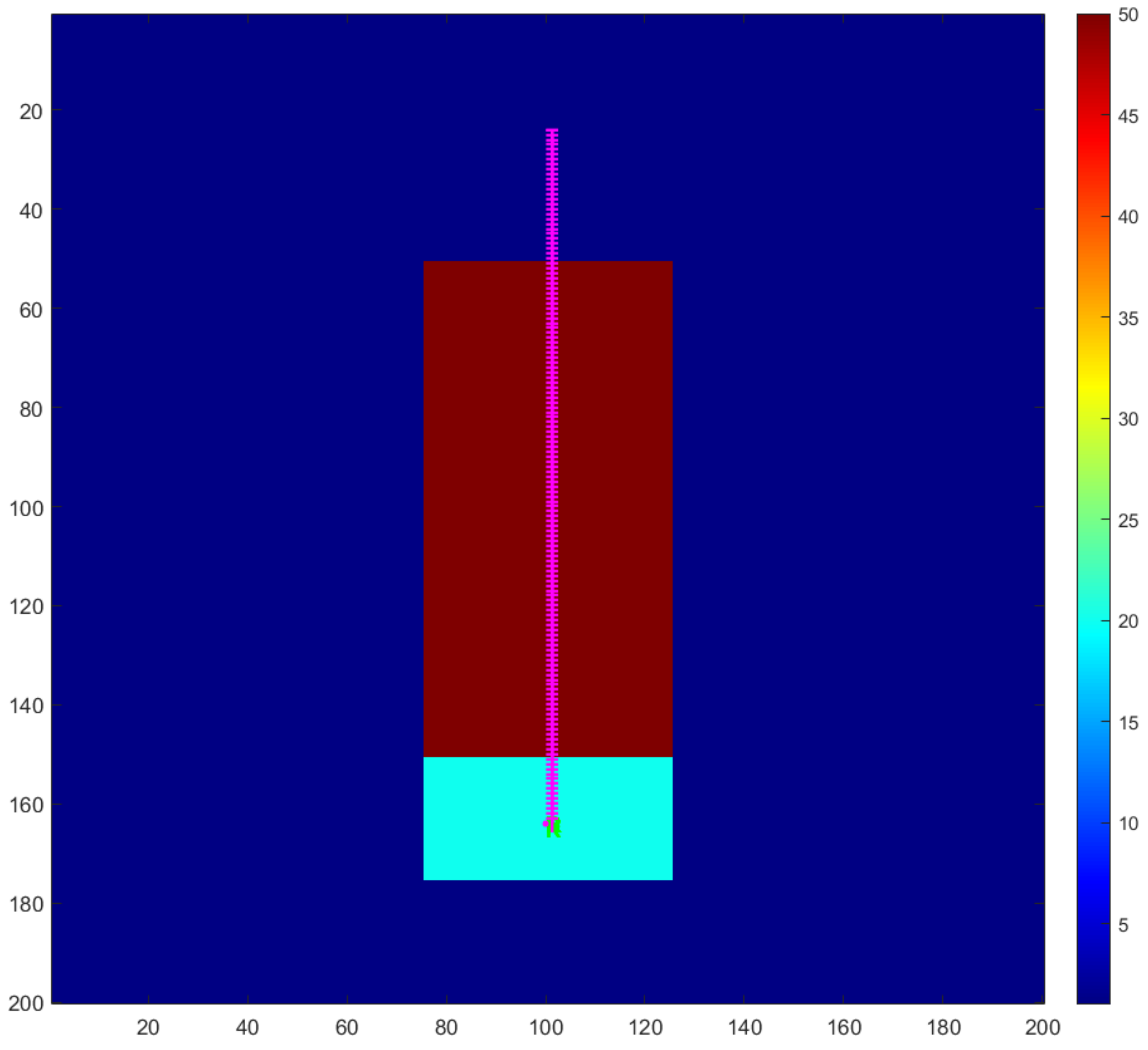
target caught = 1
time taken (s) = 408
moves made = 388
path cost = 408

Map 5



target caught = 1
time taken (s) = 182
moves made = 182
path cost = 2583

Map 6



target caught = 1
time taken (s) = 140
moves made = 1
path cost = 2800