Assignment 3

Priyanka Dhulkhed
Liam O'Brien

November 27, 2017

1 Question 8: Deliverables:

2 Part A

In order to visualize the map, we used matplotlib
A particular map along with the outputs of different heuristic functions can be seen below:

Figure 1: A map generated from running the code. The very dark blue lines are the rivers. The lighter blue cells are blocked. Light blue cells are hard to traverse cells. Normal cells are yellow.

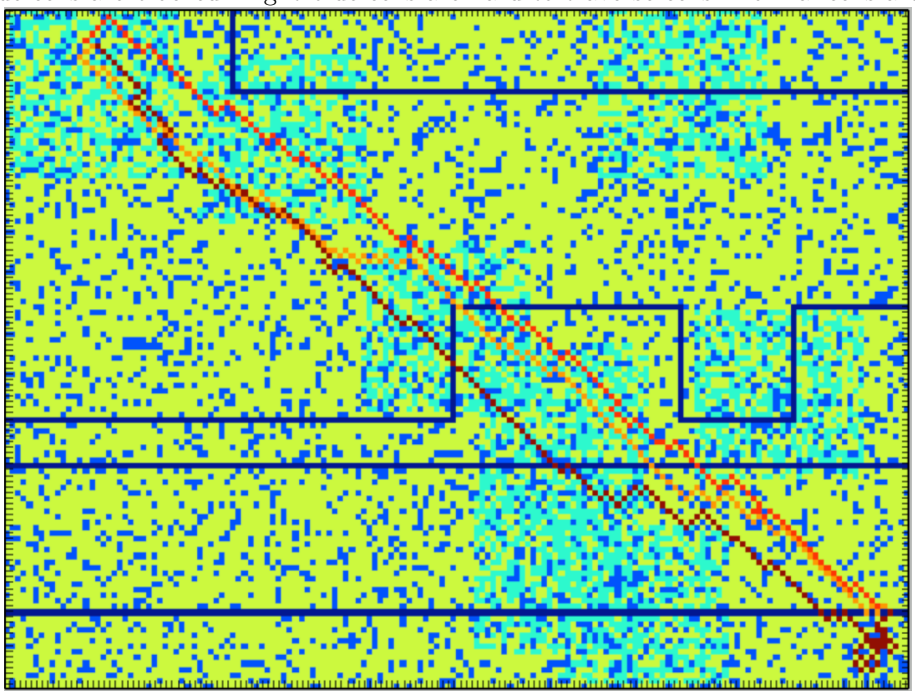


Figure 2: Maroon: A*, Red: Uniform-Cost, Gold: A* with 2.5 weight

3 Part B

The function heuristicPath(mapp, algo, rows, columns, w) takes in:

- a mapp object (which has the map, the start list, and the goal list)
- the algo–
  - 'U' for uniform cost search with w = 1

  - 'A' for A* with weight = 1
  - 'W' for weighted A* with any weight specified in w
- rows = 120
- columns = 160
- w, the weight used for weighted A*

A particular map is shown below:

Figure 3: A map generated from running the code. The very dark blue lines are the rivers. The lighter blue cells are blocked. Light blue cells are hard to traverse cells. Normal cells are yellow.

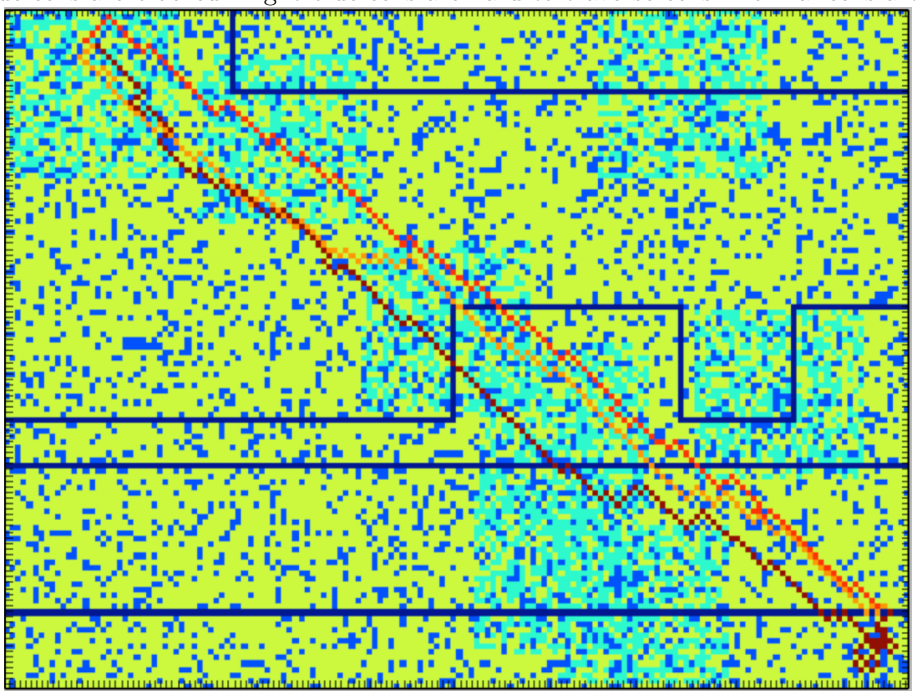


Figure 4: Maroon: A*, Red: Uniform-Cost, Gold: A* with 2.5 weight

4 Part C

Some things we have done to improve the speed of the algorithm:

1. Using Binary Heap instead of List or Set: improves running time from O(n) to $O(logn)$

2. Instead of using a closed list or set, we have set a flag on each cell that is set to True once the cell has been touched by the A* algorithm. Using a flag reduces the operation of checjking whether an element is in the closed list from O(n) to $O(1)$. We had even considered using a heap to constrcuct the closed set. However, using a flag reduces the running time of checking in the closed list to $O(1)$.

5 Part D

A good heuristic gives a good estimate of the true cost of the search process. The purpose of the heuristic value os to guide the search process and a seach that recieves appropriate, accurate

guidance will terminate faster than a search that is poorly guided. Keeping this in mind, some of the heuristic functions that we have considered are as follows:

1. One consideration was to use the actual distance between a given node and the goal node. The maximum diagonal cost to travel between two nodes is $\sqrt{2}$. However, not all traversals are diagonal. If the given node were above the goal node, then $\sqrt{2}$ would be too much of a cost. Thus we must factor in the cost of 1 (the cost to traverse between two regular cells) into the heuristic function. This heuristic is admissable because it underestimates the actual cost, which would be greater because of the increased costs from hard to traverse cells. Thus we used the Euclidean distance as shown below:

```
1●
2  def heuristicVal3(s, start, goal):
3    sx = s[0]
4    sy = s[1]
5    startx = start[0]
6    goalx = goal[0]
7    starty = start[1]
8    goaly = goal[1]
9    dx = Decimal(abs(sx - goalx))
10   dy = Decimal(abs(sy - goaly))
11   return Decimal(math.sqrt(dx*dx + dy*dy))
```

Listing 1: Euclidean Distance

2. Another consideration was to use a metric that would guide the heuristic is such a way as to choose the direction to move based on whether the traversal between rows or the traversal between column was less expensive. Thus, we used the Chebyshev Distance, which was the diagonal distance optimized so that the search would go in the path (row or column) that was less costly. It is given below:

```
1●
2  def heuristicVal2(s, start, goal):
3    sx = s[0]
4    sy = s[1]
5    startx = start[0]
6    goalx = goal[0]
7    starty = start[1]
8    goaly = goal[1]
9    dx = Decimal(abs(sx - goalx))
10   dy = Decimal(abs(sy - goaly))
11   return Decimal((dx + dy) - min(dx, dy))
```

Listing 2: Chebyshev Distance

3. Another consideration was to use a heuristic that takes into consideration the average number of blacked and hard to traverse terrain there is between the start and the goal. We know from the description of the problem that around 18.75 percent of all the cells in the grid are hard to traverse and the minimum distance to traverse between hard to traverse cells is $\sqrt{8}$. The minimum length between the start and end goal is 100. So, if 19 of the 100 cells are hard to traverse, the worst case placement would be all 19 of them placed after the regular cells. Thus, one heuristic we used is as follows:

```
1●
2  def heuristicVal1(s, start, goal):
3    sx = s[0]
4    sy = s[1]
5    startx = start[0]
```

```
  goalx = goal[0]
  starty = start[1]
  goaly = goal[1]
  dx = Decimal(abs(sx - goalx))
  dy = Decimal(abs(sy - goaly))
  dxx = Decimal(abs(startx - goalx))
  dxy = Decimal(abs(starty - goaly))
  if random.random() < 0.1875:
    return Decimal(math.sqrt(dx*dx + dy*dy)*math.sqrt(2) + math.
        sqrt(8)*math.sqrt(dxx*dxx +dyy*dyy))
  return Decimal(math.sqrt(dx*dx + dy*dy)*math.sqrt(2))
```

Listing 3: Heuristic Based on average number of hard to traverse cells

4. Another consideration for us was to use a heuristic function that takes into account the percentage of blocked cells and emphasized moving horizontally or vertically based on that. We know that that 20 percent of all the cells are blocked. Thus, if the minimum length between the start and the end is 100, then on average 20 of the cells between the start and end are blocked. Thus, 20 percent of the time we will have to place more emphasis on moving horizontal or vertical. We will combine the Euclidean heuristic function with the Manhattan one to obtain a new heuristic function as follows:

```

def heuristicVal4(s, start, goal):
  sx = s[0]
  sy = s[1]
  startx = start[0]
  goalx = goal[0]
  starty = start[1]
  goaly = goal[1]
  dx = Decimal(abs(sx - goalx))
  dy = Decimal(abs(sy - goaly))
  dxx = Decimal(abs(startx - goalx))
  dxy = Decimal(abs(starty - goaly))
  if random.random() < 0.2:
    return Decimal(dx + dy)
  if random.random() < 0.1875:
    return Decimal(math.sqrt(dx*dx + dy*dy)*math.sqrt(2) + math.
        sqrt(8)*math.sqrt(dxx*dxx +dyy*dyy))
  return Decimal(math.sqrt(dx*dx + dy*dy)*math.sqrt(2))
```

Listing 4: Heuristic Based on average number of hard to traverse cells as well as blocked cells

5. In order to attempt to make the heuristic function as fast as possible, we will consider the heuristic that makes the search emphasized along the straight line path between the start and goal cells. Even though it will explore on certain blocked cells, it will remain on the path of the straight line distance. The heuristic is as follows:

```

def heuristicVal5(s, start, goal):
  sx = s[0]
  sy = s[1]
  startx = start[0]
  goalx = goal[0]
  starty = start[1]
  goaly = goal[1]
  dx = Decimal(abs(sx - goalx))
  dy = Decimal(abs(sy - goaly))
  dxx = Decimal(abs(startx - goalx))
  dxy = Decimal(abs(starty - goaly))
  cross = Decimal(abs(dx*dyy - dxx*dy))
```

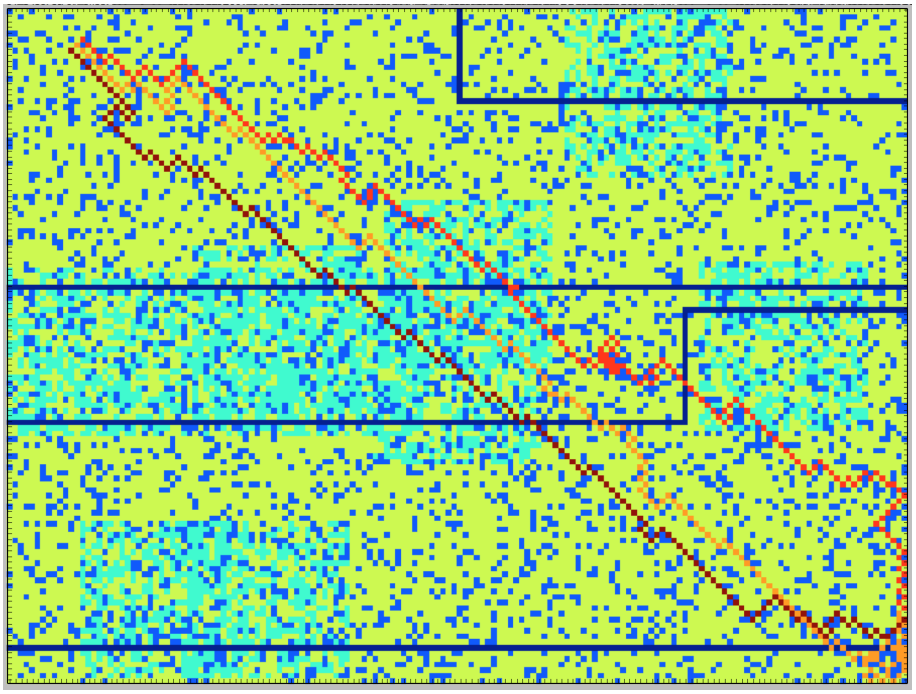
```
14      return cross
```

Listing 5: Heuristic Based on average number of hard to traverse cells as well as blocked cells

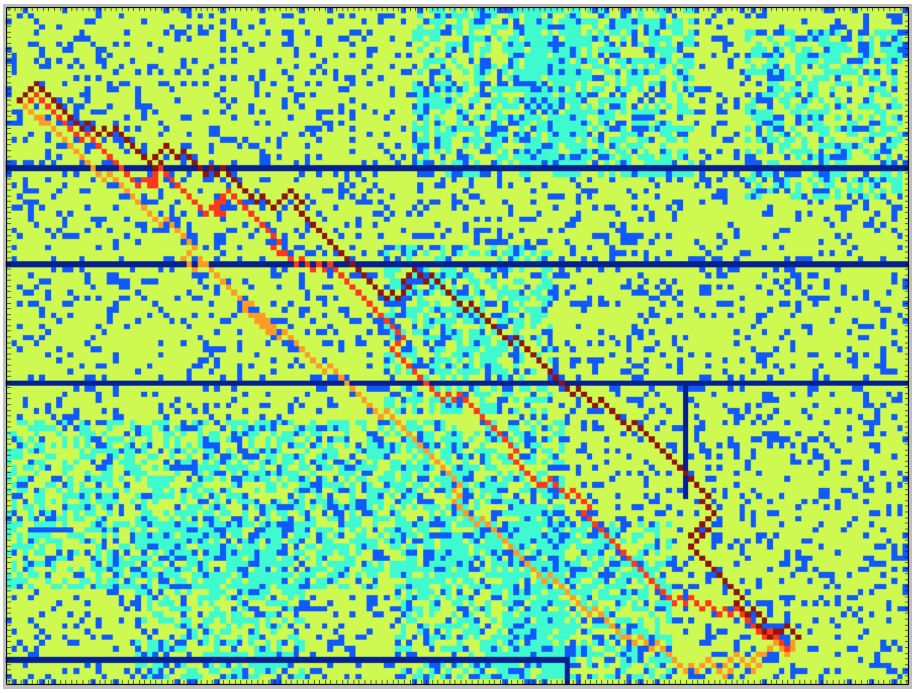
6 Part E

6.1 Maps Generated From Euclidean Heuristic (HeuristicVal3)

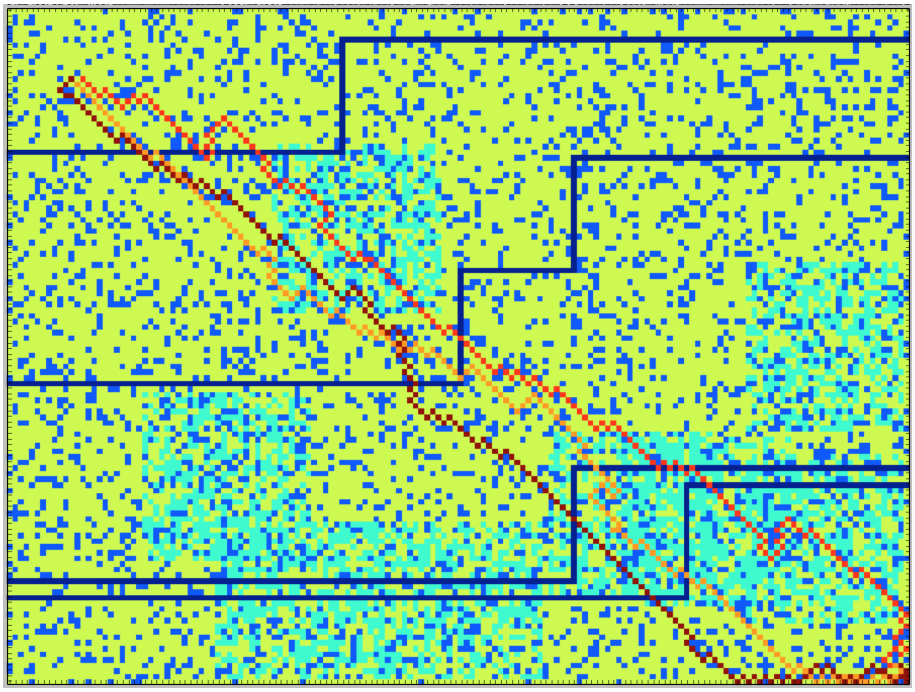
6.1.1 Maps Generated from weight 1.75 for Weighted Heuristic



6.1.2 Maps Generated from weight 2.0 for Weighted Heuristic



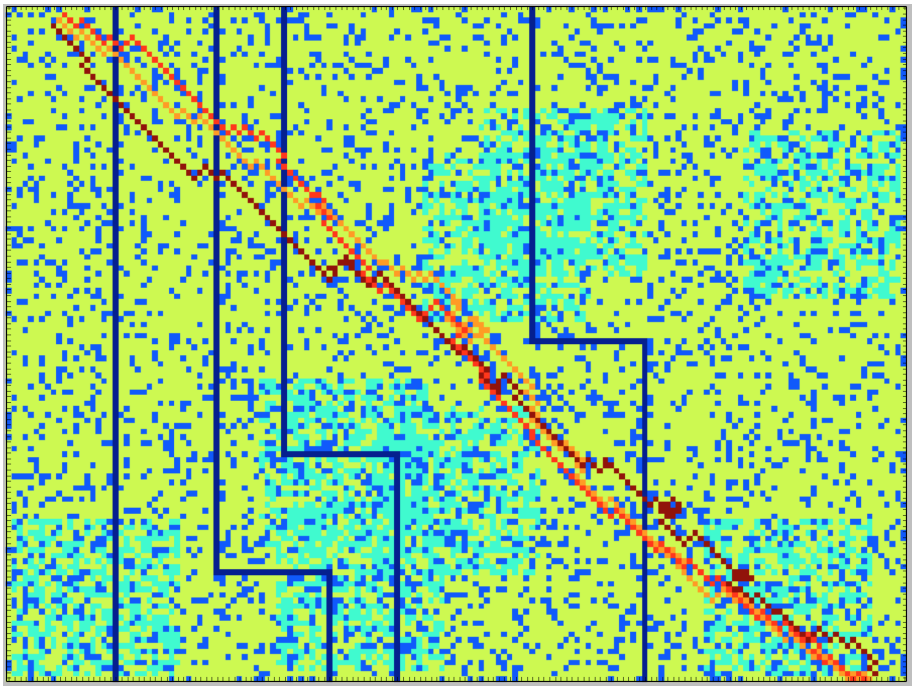
6.1.3 Maps Generated from weight 2.5 for Weighted Heuristic



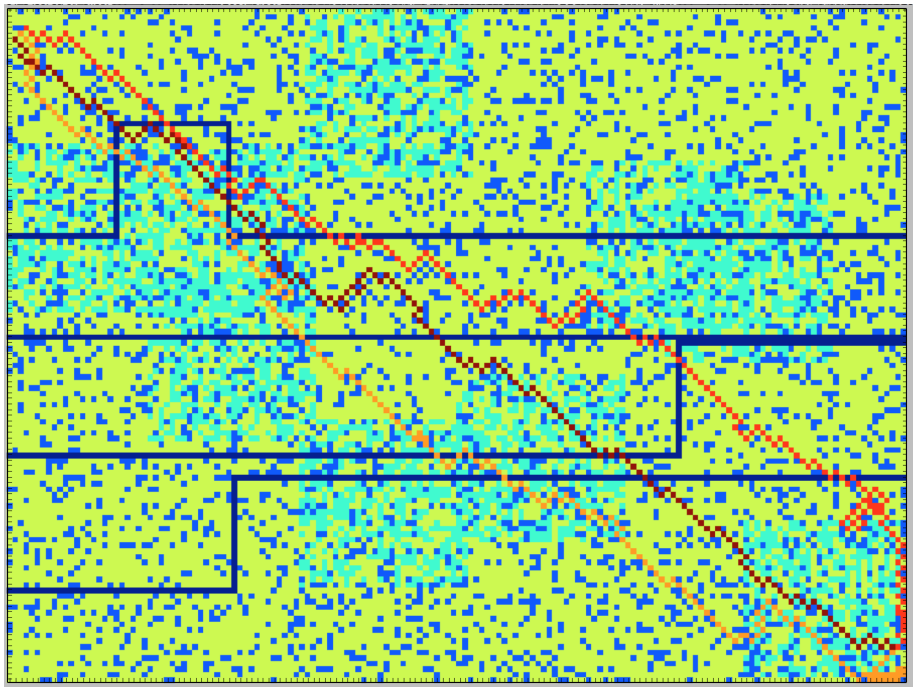
From these three graphs the one with a weight of 2 performed the best (6.1.2). With a lesser weight such as 1.75 it took longer for each search to eventually find the goal, and for a weight of 2.5 (6.1.3) it t

6.2 Maps Generated from Chebyshev Distance (HeuristicVal2)

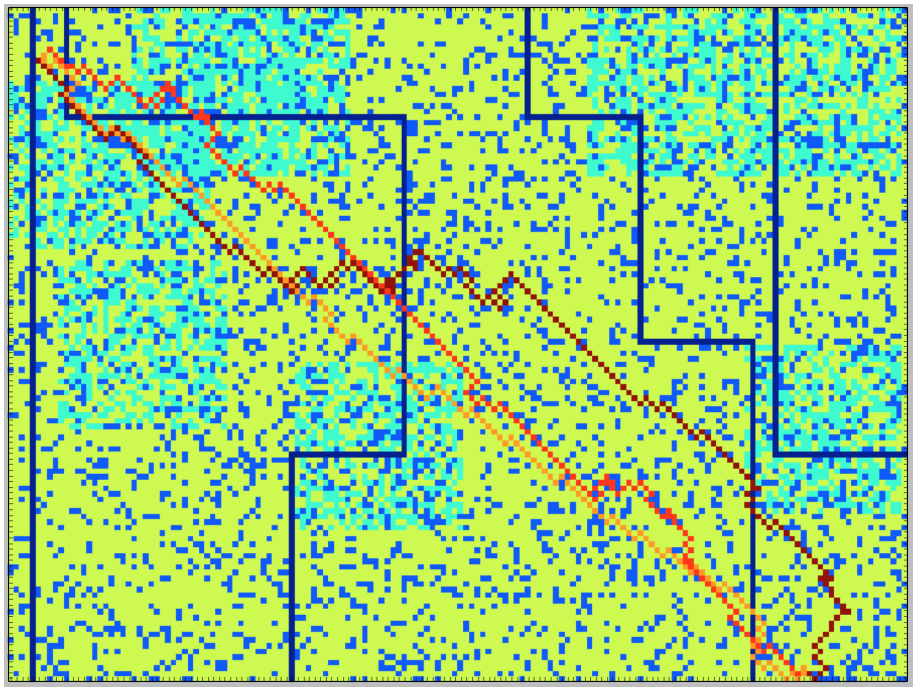
6.2.1 Maps Generated from weight 1.75 for Weighted Heuristic



6.2.2 Maps Generated from weight 2.0 for Weighted Heuristic

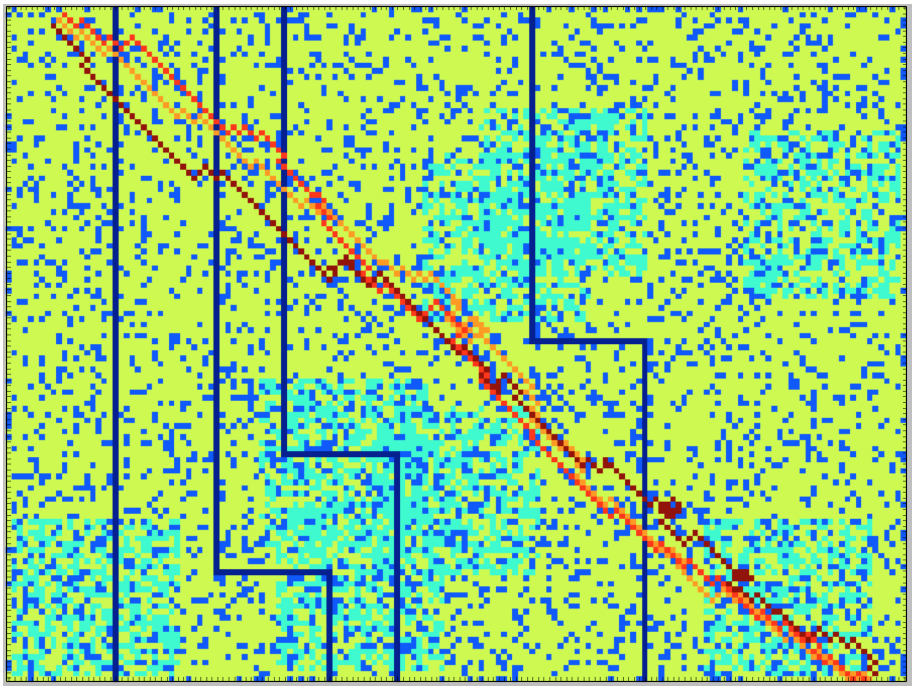


6.2.3 Maps Generated from weight 2.5 for Weighted Heuristic

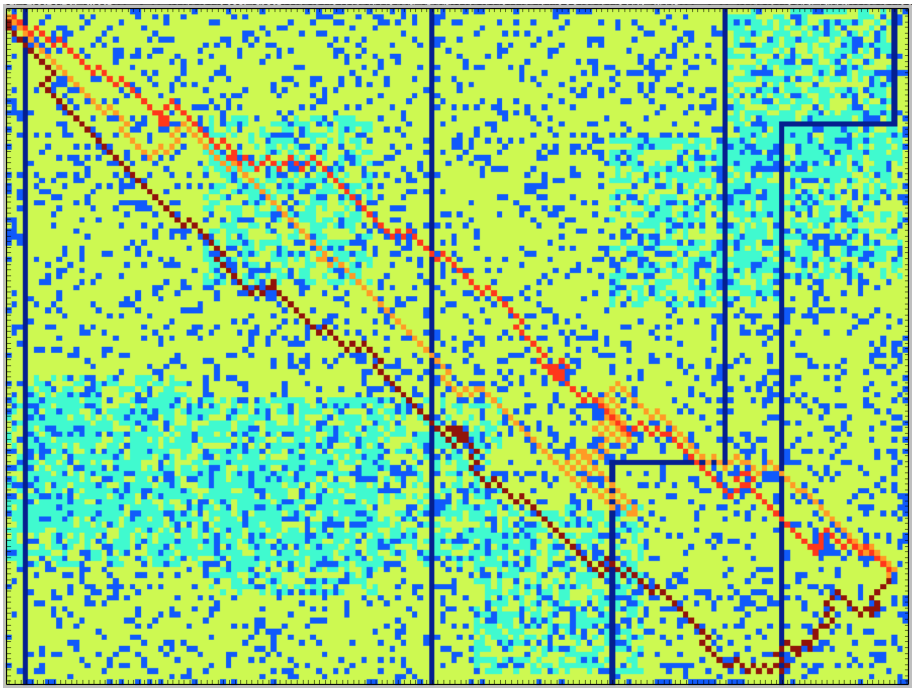


6.3 Maps Generated from the 3rd Heuristic described (HeuristicVal1)

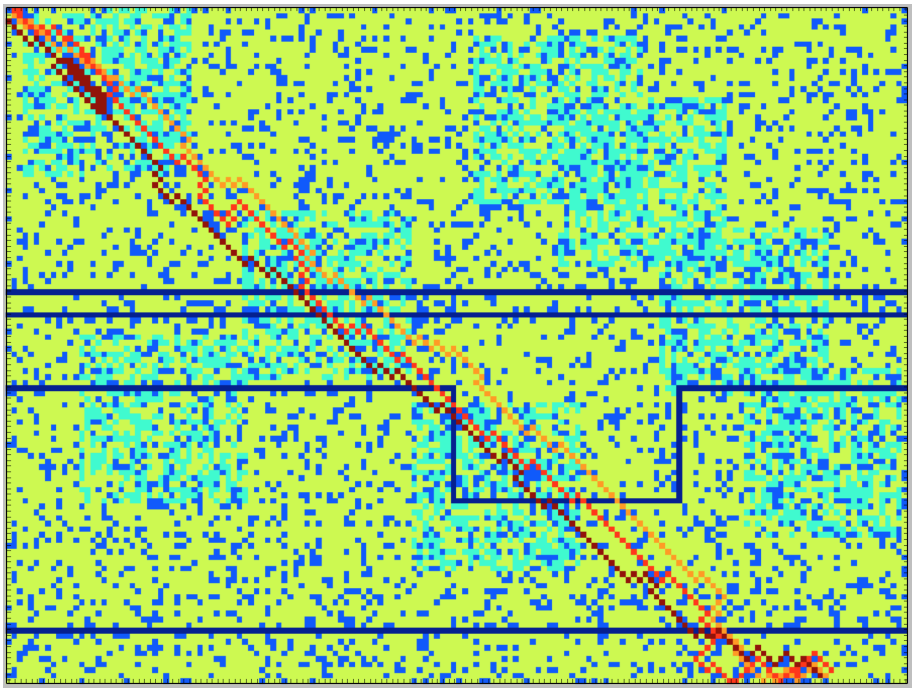
6.3.1 Maps Generated from weight 1.75 for Weighted Heuristic



6.3.2 Maps Generated from weight 2.0 for Weighted Heuristic

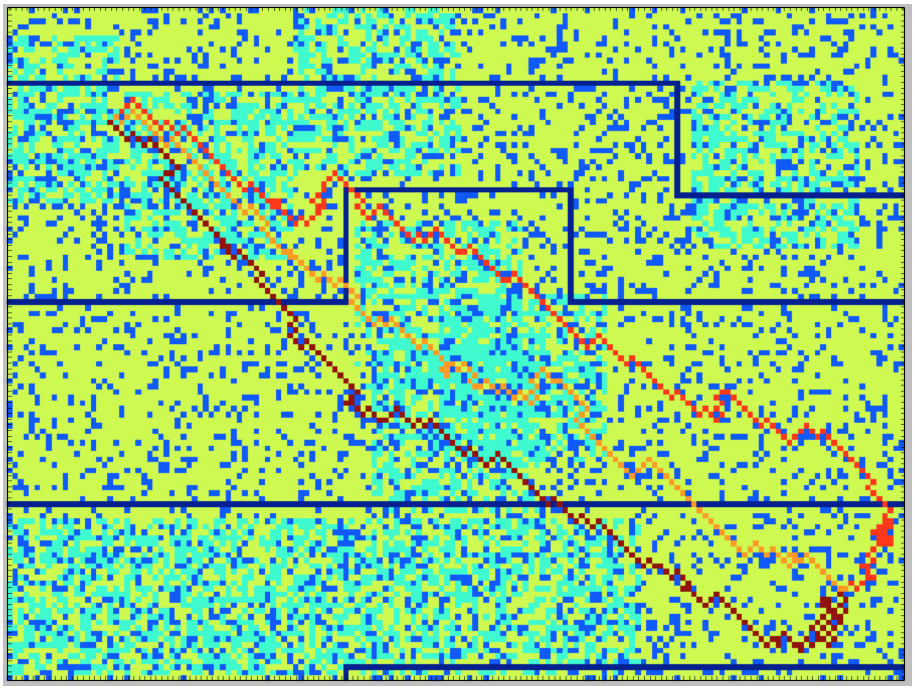


6.3.3 Maps Generated from weight 2.5 for Weighted Heuristic

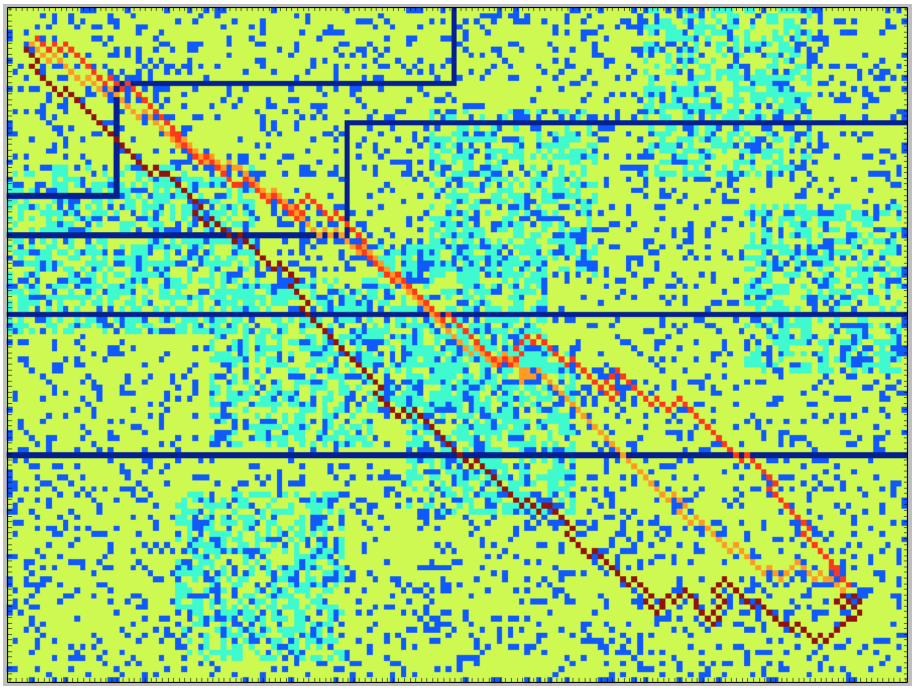


6.4 Maps Generated from 4th Heuristic function described(HeuristicVal4)

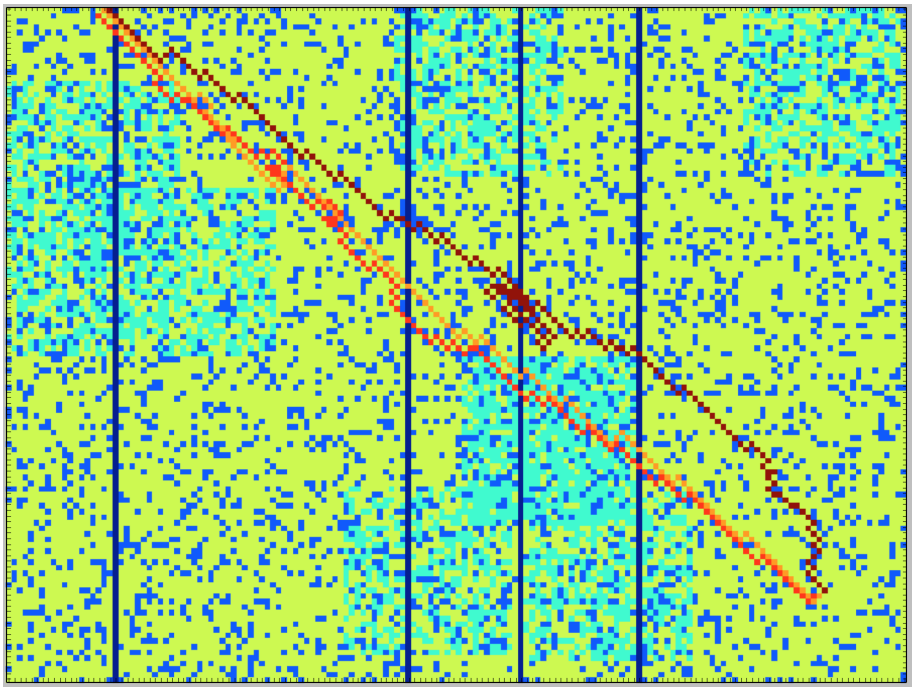
6.4.1 Maps Generated with weight 1.75



6.4.2 Maps Generated with weight 2



6.4.3 Maps Generated with weight 2.5



7 Part F

From the maps and times we gathered, it is clear that each heuristic performed differently accoring to a) the weight and b) the actual heuristic implemented. We ran each of the heuristics until we got 5 successful searches and took the average over the 5 tests. These were our times for each heuristic:

Heuristic 1: w = 1.75: Time to sovle: 2.01

Heuristic 1: w = 2.0: Time to sovle: 2.44

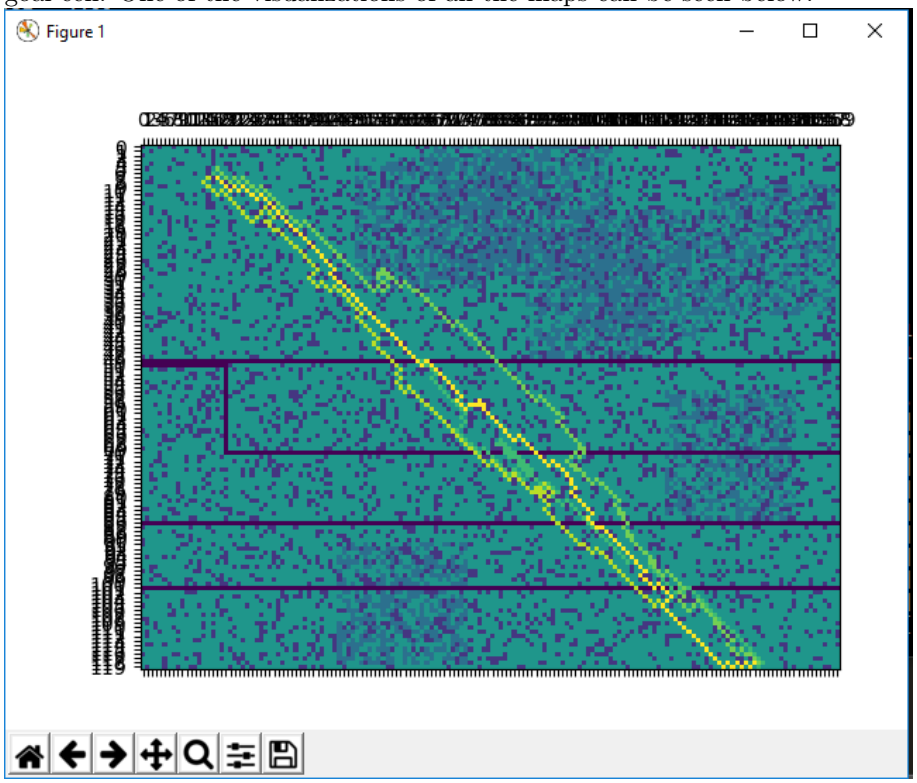
Heuristic 1: w = 2.5: Time to sovle: 2.57

Heuristic 2: w = 1.75: Time to sovle: 2.71
Heuristic 2: w = 2.0: Time to sovle: 2.81
Heuristic 2: w = 2.5: Time to sovle: 2.97
Heuristic 3: w = 1.75: Time to sovle: 1.75
Heuristic 3: w = 2.0: Time to sovle: 1.81
Heuristic 3: w = 2.5: Time to sovle: 1.89
Heuristic 4: w = 1.75: Time to sovle: 2.80
Heuristic 4: w = 2.0: Time to sovle: 2.92
Heuristic 4: w = 2.5: Time to sovle: 3.03
Heuristic 5: w = 1.75: Time to sovle: 4.11
Heuristic 5: w = 2.0: Time to sovle: 4.34
Heuristic 5: w = 2.5: Time to sovle: 4.73

From the above data we can see that our third heuristic performed the best with respect to time. Our slowest performance was heuristic 5 which ran almost 40% slower than our fastest. An overall trend over all the heuristics is that as the weight increased for each heuristic, it took longer computing time to find a solution.

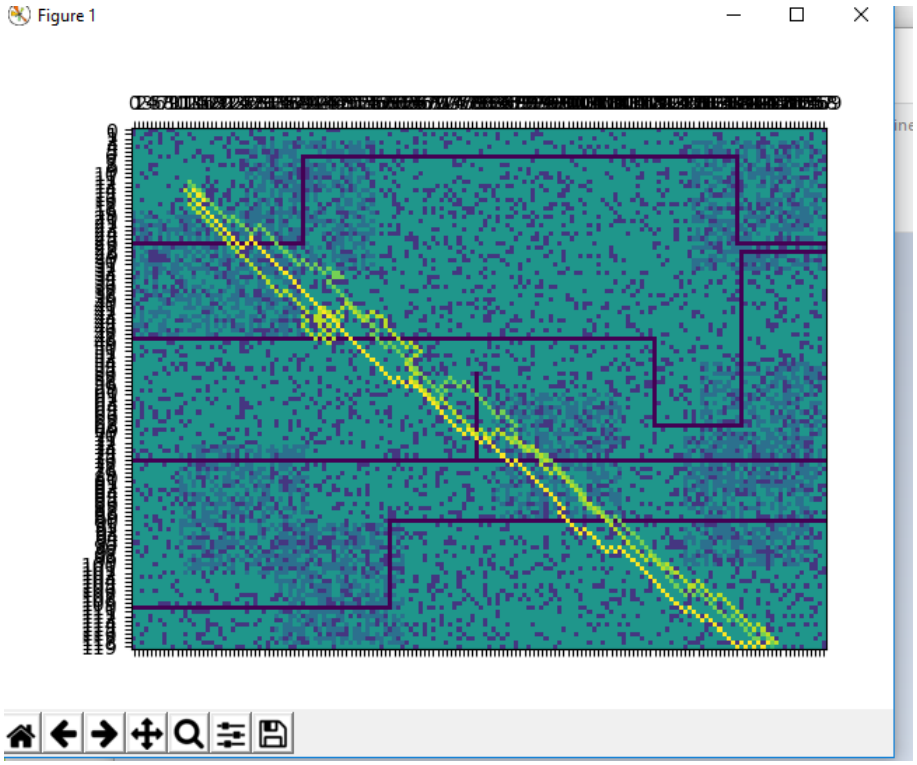
8 Part G

We have implemented the algorithm in node.py. What we noticed was that the smaller w1 is compared to w2, the faster the convergence was. The cells visited became closer and closer to the goal cell. One of the visualizations of all the maps can be seen below:

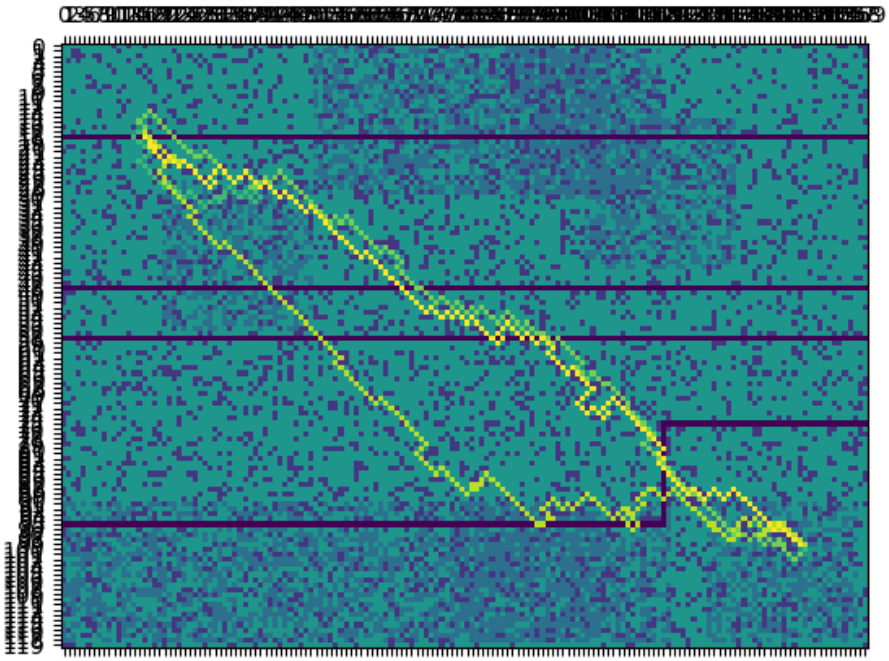


The darkest yellow line is the sequential path. w1 = 1.25 and w2 = 2.

With w1 = 3 and w2 = 2, time = 3.15200018883 seconds. As we can see from the visualization below, when w1 is higher, the path resembles a straight line distance because that was our anchor heuristic. The visualization is below:



Another visualization with time = 15.28500858 seconds wall time is below. The w1 = 1.05 and w2 = 3. This might be an anomoly but is consistent with our deduction that the higher w1 is, the faster the path will be.



The average run time was time =13.5568 seconds over all 50 grids.

9 Part H

I attempted to optimize the space complexity and time complexity of running these algorithms by avoiding having to store a closed list. Instead, I made an array of flags for each of the 5 heuristics so that checking if something is in the closed list takes O(1) time. Also, the anchor that we used

was powerful in that it was a good estimate (And admissable) of the path. We used the Chebyshec Distance first as the anchor and then proceeded to use heuristicVal5, which was the straight line distance between the start and goal nodes. We also attempted using heuristicVal3 as our anchor since it was the fastest of all the heurisitcs as per our findings in part e. The straight line distance was the most accurate as the anchor, especially when we used a high w1 compared to w2. This is because the algorithm came back to the straight line path even after obstacles. The algorithm that does the best to minimize horizontal/vertical traversals and focuses on diagonal, straight-line traversals did better at finding the path in terms of time and accuracy.

10 Part I

For this proof, we are going to assume that the provided holds are true:

1. Key(s,0) ≤ Key(u,0) ∀u ∈ $OPEN_0$
2. Key(s,0) = $g_0(s) + w_1 * h_0(s) > w_1 * g * (s_{goal})$

From the hint provided we know that the least cost path P = (s_0 = start, ... s_n = goal) exists because s_0 is put into $OPEN_0$ when the queue is initialized.

We need to find the first s_i that has not been expanded in the search so we need to look for the base case. If I = 0, then:

$g_0(s_i) = g_0(s_{start}) = 0$

$0 \leq w_1 * g^*(s_i)$

Next, if I ≠ 0, then there exists a s_{i-1} that has already been expanded, so now we have the equation from the Sequential A* Search:

$g_0(s_i) \leq g_0(s_{i-1}) + c(s_{i-1}, s_i)$

$\leq w_1 * (g^*(s_{i-1}) + c(s_{i-1}, s_i))$

Since (s_{i-1}, s_i) is an optimal path:

$Key(s_i, 0) = g_0(s_i) + w_1 * h_0(s_i)$

$\leq w_1 * g * (s_i) + w_1 * h_0(s_i)$

$\leq w_1 * g * (s_i) + w_1 * g * (s_i, s_{goal})$

$= w_1 * g * (s_{goal})$

Since s_i exists in $OPEN_0$ and $Key(s_i, 0) \leq w_1 * g * (s_i, s_{goal}) <$ key(s,0), there is a contradiction, thus proved.

For the second proof we again need to observe the Sequential A* Search. From the hint, we needed to find all cases that the algorithm can terminate.
The first case that can cause it to terminate is on line 19. If the algorithm terminates here it means that a finite cost solution does not exist.

Next, the algorithm can terminate on lines 21 through 22:

$g_i(s_{goal}) \leq w_1 * OPEN_0$.Minkey()

From the above proof we know that the following holds true:

$\leq w_2 * w_1 * g * (s_{goal})$

The next termination can occur on line 30. We know that

$g_0(s_{goal}) \leq w_1 * g * (s_{goal})$

$g_0(s_{goal}) \leq w_2 * w_1 * g * (s_{goal})$

As long as $w_2 \geq 1$

In the previous two cases, there exists a solution cost in between $w_2 * w_1$. The first case means that the cost is infinite and thus it is not complete.