Motion Planning

Nathan Cusson-Nadeau

I. Introduction

In the field of robotics, it is often a valuable and desirable trait for a robot agent to devise a control policy that describes a sequence of valid state configurations which generates a trajectory between its current position and some goal. This goal could be something static like a box a warehouse robot needs to pick up to complete its task, or something dynamic, like an explosive projectile which an autonomous drone needs to intercept and disarm before it is able to detonate and cause harm. This type of problem is called *motion planning* or the piano mover's problem.

To allow a robot to motion plan, there are several algorithms that can employed. These algorithms can be categorized based on the ways in which the control policy for the robot is generated. For instance, there are *search-based* planning algorithms such as Dijkstra's, A* and Jump Point Search (JPS) which explore surrounding nodes until a goal node is found then construct best path available through the explored space. And then there are *sampling-based* algorithms such as, Rapidly Exploring Random Trees (RRT) and Probabilistic Road Maps (PRM) which take progressive samples from the environment and construct a graph of interconnected nodes which can produce a path from the goal to the starting location.

In this paper we investigate the efficacy of a particular search-based algorithm and discuss it's efficacy and shortcomings when applied to a pursuit-evasion game.

II. Problem Formulation

Consider two robotic agents playing a *pursuit-evasion game*, in a closed 2-D grid-world environment containing obstacles. One robot is a pursuer P and the other is an evader E. The finite state space $\mathcal{X}$ of both agents is identical. The agents are able to move in their allowable directions independent of their orientation. Because of this their current state can be expressed as simply the coordinates in the environment, which can be thought of as a discretized representation of $\mathbb{R}^2$. Thus, an arbitrary state $\mathbf{x_i} \in \mathcal{X} \in \mathbb{R}^2$ is represented as:

$$\mathbf{x_i} := \begin{cases} x_1 = x - coordinate \\ x_2 = y - coordinate \end{cases} \tag{1}$$

The environment, or configuration space $C(\mathbf{x_i})$, can be viewed as an occupancy grid where:

$$C(\mathbf{x_i}) := \begin{cases} 0, & \text{free} \\ 1, & \text{obstacle} \end{cases} \tag{2}$$

see Figure 1 for an example of a configuration space.

The goal of P is to catch E. To be considered *caught*, P must occupy the same grid coordinate as E, we will call this the goal state $\mathbf{x}_\tau$. P is able to move to any adjacent square and E can only move to squares on the cardinals. This gives P an inherent advantage over E and ensures in closed environments E can be captured with the proper control sequence applied to P.

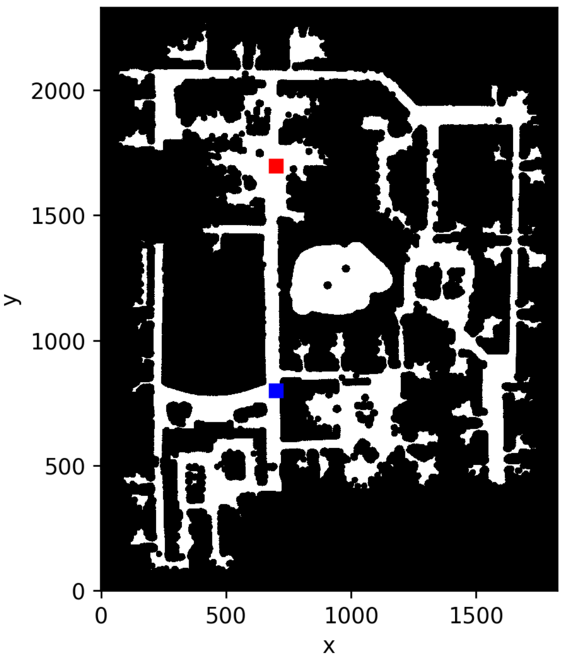

Fig. 1: Example configuration space. Blue Square: Pursuer, Red Square: Evader, Black squares: Obstacles, White squares: Free Space

Control Definitions		
u	Direction	Effect
U	UP	$y+1$
D	DOWN	$y-1$
L	LEFT	$x-1$
R	RIGHT	$x+1$
UL	UP & LEFT	$x-1,\ y+1$
UR	UP & RIGHT	$x+1,\ y+1$
DL	DOWN & LEFT	$x-1,\ y-1$
DR	DOWN & RIGHT	$x+1,\ y-1$

TABLE I: Control Space Dictionary

The control space of agent P is defined as:

$$\mathcal{U}_P := \{\text{U, D, L, R, UL, UR, DL, DR}\} \tag{3}$$

where the pursuer controls $u_P \in \mathcal{U}_P$ represent and map as described in Table I. The control space of E is a subset of

$\mathcal{U}_P$, as it can only move horizontally and vertically. It can be expressed as:

$$\mathcal{U}_E := \{\text{U, D, L, R}\} \subseteq \mathcal{U}_P \tag{4}$$

The deterministic motion models $f_P(x, u)$ and $f_E(x, u)$ of agents P and E are defined as:

$$f_P(\mathbf{x_i}, u_P) := \begin{cases} \mathbf{x_i}, \ \mathbf{x_j} \text{ would be an obstacle} \\ \mathbf{x_j}, \ \mathbf{x_j} \text{ is free space} \end{cases} \tag{5}$$

$$f_E(\mathbf{x_i}, u_E) := \begin{cases} \mathbf{x_i}, \ \mathbf{x_j} \text{ would be an obstacle} \\ \mathbf{x_j}, \ \mathbf{x_j} \text{ is free space} \end{cases} \tag{6}$$

where $\mathbf{x_j}$ is the next state achieved when applying the effect of a control u_P or u_E on prior state $\mathbf{x_i}$.

P must plan its next move in t seconds. If $t > 2$ seconds elapses before a new plan is generated, E is allowed to make additional moves. Specifically, E is able to make N moves for every move of P, where $N = t/2$ *rounded to the nearest integer value*.

E can use a variety of strategies to attempt evade P. However, for the purposes of this paper E will only be trialed using a *minimax* decision rule strategy.

A. DSP Formulation

We can reformulate this problem as a *Deterministic Shortest Path* (DSP) problem by defining several terms. Firstly, we can convert the state-space $\mathcal{X}$ to graph of nodes i which compose the vertex set V:

$$i \in \mathcal{V} \tag{7}$$

with an associated undirected edge set $\mathcal{E}$:

$$\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V} \tag{8}$$

and edge weights:

$$\mathcal{C} := \{c_{ij} \in \mathbb{R} \cup \{\infty\} \mid (i, j) \in \mathcal{E}\} \tag{9}$$

where c_{ij} represents the cost from traveling on the edge from node x_i to node x_j, defined as:

$$c_{ij} := \begin{cases} 1, i \to j \text{ is diagonal transition} \\ \sqrt{2}, i \to j \text{ is cardinal transition} \end{cases} \tag{10}$$

where a diagonal transition can be described as moving according to $u_p \in \{\text{UL, UR, DL, DR}\}$ and a cardinal transition as $u_p \in \{\text{L, R, U, D}\}$.

We then can define a path $i_{1:q}$ through the graph as:

$$i_{s:\tau} := (i_1, i_2, \dots, i_q), \ i_k \in \mathcal{V} \tag{11}$$

The set of all paths $\mathcal{P}$ from the start node $s \in \mathcal{V}$ to goal node $\tau \in \mathcal{V}$ is defined as:

$$\mathcal{P}_{s,\tau} := \{i_{1:q} | i_k \in \mathcal{V}, i_1 = s, i_q = \tau\} \tag{12}$$

The length of a path $i_{1:q}$ can be described by:

$$J^{i_{1:q}} = \sum_{k=1}^{q-1} c_{i_k, i_{k+1}} \tag{13}$$

By defining the problem in this way, the objective of P can now be described as taking moves along the best path $i^*_{s:\tau}$, which minimizes the path length J:

$$i^*_{1:q} = \arg\min_{u \in \mathcal{U}} J^{i_{1:q}} \tag{14}$$

III. Technical Approach

By formulating the problem as a DSP, several label-correcting search-based algorithms can be applied to determine the best path to τ. By computing the best path repeatedly and taking a move(s) along the path, P will eventually catch E due to its diagonal movement advantage. Thus solving the problem.

For this paper we elected to only assess a label-correcting algorithm called *Weighted A**, a modified version of the Dijikstra's algorithm, which implements a heuristic function h_i. This was chosen because it serves as the baseline for many other more advanced search-based algorithms (ARA*, LPA*, LRTA*, etc.) so the potential pitfalls of the algorithm can highlight the strengths of using alternative algorithms.

Additionally, to improve tractability of the weighted A* algorithm (which is known to have difficulty in larger environments) a modified version of A* we coin the *Delayed Weighted A** algorithm was implemented.

Several configuration spaces were used to assess the efficacy of both algorithms. These can be seen in Figures 16 through 26 in the Appendix.

A. Weighted A*

The Weighted A* algorithm can be found in algorithm 1. To execute the algorithm we first initialize an OPEN and CLOSED list. The OPEN list contains only node s and is treated as a priority queue. The CLOSED list is initially empty. We also define the heuristic weight $\epsilon \geq 1$. Additionally, the node labels of $\mathcal{V}$ were initialized as $g_s = 0$ and $g_i = \infty$. These represent the cost of going from node s to node i.

We then begin the main loop of the algorithm, which will continue until the goal node τ has been placed in the CLOSED list. At the start of each loop we remove a node i with the smallest priority label f_i, defined as:

$$f_i := g_i + \epsilon h_i \tag{15}$$

where h_i is the heuristic function defined as as the Euclidean distance between a node i and the goal τ:

$$h_i := \|x_\tau - x_i\|_2 \leq \text{dist}(i, \tau), \ \forall i \in \mathcal{V} \tag{16}$$

Because the environment is a grid-world, this guarantees the heuristic function to be admissible (a requirement for A* to return the best path).

We then insert node i into the CLOSED list and find all of its children/successors that are not in the CLOSED list. For each child node j we determine if its current label g_j is greater than the sum of the value of the parent's label g_i and the edge cost c_{ij}. If it is, we assign g_j this computed cost and assign i as the parent of j. If j was in OPEN then the priority label f_j is updated to reflect this new lower label g_j. Otherwise, the child j is added to OPEN. This completes the main loop.

When τ is added to CLOSED and the algorithm terminates, the best path $i^*_{s:\tau}$ is computed from algorithm 2. This algorithm simply follows the parents of each node starting with the goal τ and ending at s, then returns the path taken.

ALGORITHM 1
Weighted A* Algorithm

```
1:  OPEN ← {$s$}, CLOSED ← {}, $\epsilon \geq 1$
2:  $g_s = 0$, $g_i = \infty$ for all $i \in \mathcal{V} \backslash \{s\}$
3:  while $\tau \notin$ CLOSED do
4:      Remove $i$ with smallest $f_i := g_i + \epsilon h_i$ from OPEN
5:      Insert $i$ into CLOSED
6:      for $j \in$ Children($i$) and $j \notin$ CLOSED do
7:          if $g_j > (g_i + c_{ij})$ then
8:              $g_j \leftarrow (g_i + c_{ij})$
9:              Parent($j$) ← $i$
10:             if $j \in$ OPEN then
11:                 Update priority of j
12:             else
13:                 OPEN ← OPEN ∪ {$j$}
14:             end if
15:         end if
16:     end for
17: end while
18: $i^*_{s:\tau}$ ← recoverPath($s$, $\tau$, $\mathcal{V}$)
19: return $i^*_{s:\tau}$
```

ALGORITHM 2
Recover Path

```
1: Given: $s$, $\tau$, $\mathcal{V}$
2: Initialize: path as empty list, $i \leftarrow \tau$
3: while $i \neq s$ do
4:     $i^*_{1:\tau} \leftarrow i^*_{1:\tau} \cup$ Parent($i$)
5:     $i^*_{s:\tau}$ ← Parent($i$)
6: end while
7: return $i^*_{s:\tau}$
```

*B. Delayed Weighted A**

As aforementioned, Weighted A* proves intractable in large graphs/state-spaces. To circumvent this issue we propose an augmented Weighted A* algorithm called Delayed Weighted A*. This algorithm takes advantage of the high likelihood that the first moves along the best path returned from weighted A* will not change between iterations and saves computation time by only computing the path every N iterations until the target agent E is within a radius r of P. In this radius, the path will be recomputed at every iteration using algorithm 1. This ensures that the final paths are not chasing the ghost positions of E.

Seen in algorithm 3, this new algorithm is quite simple. A quick rundown goes as such: given the current iteration of moves I, the start s, goal τ, vertex set of the environment $\mathcal{V}$, and ϵ_1, ϵ_2, begin by computing the Euclidean distance from s to τ, R. If I is perfectly divisible by N and R is larger than r, we pre-compute the next N moves using Weighted A* once. Otherwise, we are within the radius r and compute the next move every iteration using only the first entry from the returned Weighted A* path.

ALGORITHM 3
Delayed Weighted A* Algorithm

```
1:  Given: $I$, $s$, $\tau$, $\mathcal{V}$, $\epsilon_1 \geq 1$, $\epsilon_2 \geq 1$
2:  $R \leftarrow \|s - \tau\|_2$
3:  if modulo($I/N$) = 0 and $R > r$ then
4:      $i_{s:\tau}$ ← WeightedA*$(s, \tau, \mathcal{V}, \epsilon_1)$
5:      moves ← $i_{2:N}$
6:  else
7:      $i_{s:\tau}$ ← WeightedA*$(s, \tau, \mathcal{V}, \epsilon_2)$
8:      moves ← $i_2$
9:  end if
10: return moves
```

IV. RESULTS

After testing both A* algorithms on a variety of configuration spaces, it was found that it performed very well on the majority of presented spaces but quite poorly in certain starting configurations (notably maps 1b and 7). P consistently caught E in the majority of cases for Weighted A*, and nearly all cases for Delayed Weighted A* within a reasonable amount of time.

The Weighted A* algorithm was assessed based on 3 metrics: *total moves*, *time to compute first move*, *initial path cost*. Each of these metrics was tested for 3 different heuristic weights $\epsilon \in \{1, 2, 5\}$.

Similarly, the Delayed Weighted A* algorithm was assessed by only the total move metric (the time and cost of the initial path will be identical to the unaltered algorithm), also using the 3 different ϵ weights ($\epsilon_1 = \epsilon_2$). $N = 100$ and a radius $r = 30$ were chosen for testing the Delayed Weighted A* algorithm, as these seemed like suitable values for the tested maps.

Tables II through VI provide the results from all test runs. Figures 3 through 15 illustrate some trajectories in green of P throughout the various maps. In the following sections, each algorithms' strengths and weaknesses will be discussed, as well as future direction.

A. Weighted A Results*

As seen in Tables II through Table IV maps 1b, 3b, 3c, and 7 proved difficult for the Weighted A* algorithm. For these

Total Moves: Weighted A*			
Maps	ϵ = 1	$\epsilon = 2$	$\epsilon = 5$
Map 0	5	4	4
Map 1	*	1279	1279
Map 2	13	12	12
Map 3	*	223	223
Map 4	9	9	8
Map 5	86	121	87
Map 6	39	40	40
Map 7	*	*	*
Map 1b	*	*	*
Map 3b	*	*	*
Map 3c	*	*	762

TABLE II: Total moves required to catch the evader for each map using Weighted A* algorithm. Stars (*) mean results could not be generated due to intractability.

Cost of Initial Path: Weighted A*			
Maps	ϵ = 1	$\epsilon = 2$	$\epsilon = 5$
Map 0	6.243	6.243	6.243
Map 1	900	900	900
Map 2	15.071	15.071	15.071
Map 3	253.137	253.137	253.137
Map 4	10.414	10.414	13.48
Map 5	86	121	87
Map 6	84.468	87.397	86.225
Map 7	*	*	*
Map 1b	*	*	*
Map 3b	457.960	467.073	494.914
Map 3c	732.997	802.907	802.907

TABLE III: Cost of initial path from start to goal using Weighted A* algorithm. Stars (*) mean results could not be generated due to intractability.

maps, the time to compute the next move typically greatly exceeded 2 seconds making it a poor choice as a motion planning algorithm for most practical implementations.

For maps 1b and 7, the clear pitfall was the sheer size of the configuration space of the environment as well as the agents' relative positions from one another. To compute a singular move, Weighted A* required a very large number of nodes which would take a considerable amount of time. Even with an $\epsilon = 5$ the time to compute the first move on Map 1b exceeded 3 minutes.

Time of Initial Plan (seconds): Weighted A*			
Maps	ϵ = 1	$\epsilon = 2$	$\epsilon = 5$
Map 0	0.001	0.001	0.001
Map 1	0.226	0.200	0.258
Map 2	0.002	0.003	0.003
Map 3	15.253	0.047	0.044
Map 4	0.002	0.002	0.002
Map 5	0.332	0.050	0.027
Map 6	0.153	0.023	0.021
Map 7	*	*	*
Map 1b	*	*	227.613
Map 3b	117.140	5.929	2.981
Map 3c	748.576	78.233	3.887

TABLE IV: Time to compute initial path using Weighted A* algorithm. Stars (*) mean results could not be generated due to intractability.

Some notable behaviors of Weighted A* are the increased cost of paths (i.e. longer paths) generated from larger ϵ values in larger maps. Map 3b specifically illustrates a trend that as ϵ increased the length of the path also did. This behavior can be attributed to the presence of obstacles and the nature of what the heuristic measures. With a large weight given to the heuristic nodes that are closer to the goal are given preference in expansion. Without the presence of many obstacles between the start and goal this does not change the path cost, however with obstacles the inaccuracy of labeling this nodes is apparent. Clearly, a larger ϵ will increase performance speed, but at the cost of decreased performance overall (more likely to return lengthier paths).

B. Delayed Weighted A* Results

Most results from utilizing Delayed Weighted A* proved uninteresting, as they mirrored behavior in the normal Weighted A* due to the maps being within the radius. However, the tractability over Weighted A* was greatly apparent. In all maps except Map 7 and Map 1b, the algorithm proved tractable.

This shows that even though the algorithm did increase tractability, the intractability of A* is still a limiting factor. With lower epsilon values, the first path computation took exorbitant amount of times to compute in larger maps, making maps 1b and 7 infeasible.

However, the increased tractability was not without a cost. Assuming that the path would not change for N moves means that in some situations the total moves to capture E was lengthened. Table VI illustrates that in Map 5 and 3c, additional moves were needed to capture E than normal. Additionally, because of the strategy employed by E, this strategy belies this pitfall more. E commonly was stuck in some region of the map due to its minimax strategy. If this were not the case, P could easily arrive at a location that was long departed by E, adding many more moves, if not making it outright impossible to catch E! Lowering N or increased r could mitigate this issue, but eventually this approaches the regular Weighted A* algorithm - rendering the approach pointless.

Another avenue that could be further explored which could improve tractability is using different ϵ values for ϵ_1 and ϵ_2. Using a large value for ϵ_1 and a small value for ϵ_2 could shorten the path length of reaching the goal while remaining tractable. This is because large movements would be computed quickly because of a larger heuristic weight, while finer paths when close to the target could be handled by a lower weight (when computation times are well below 2 seconds). Figure 2 illustrates an example of this where $\epsilon_1 = 5$ and $\epsilon_2 = 1.5$.

C. Correcting the Algorithm and Alternatives

As seen from these results, while A* does guarantee a path from s to τ, it can quickly become computationally intractable and infeasible to use on larger configuration spaces where the

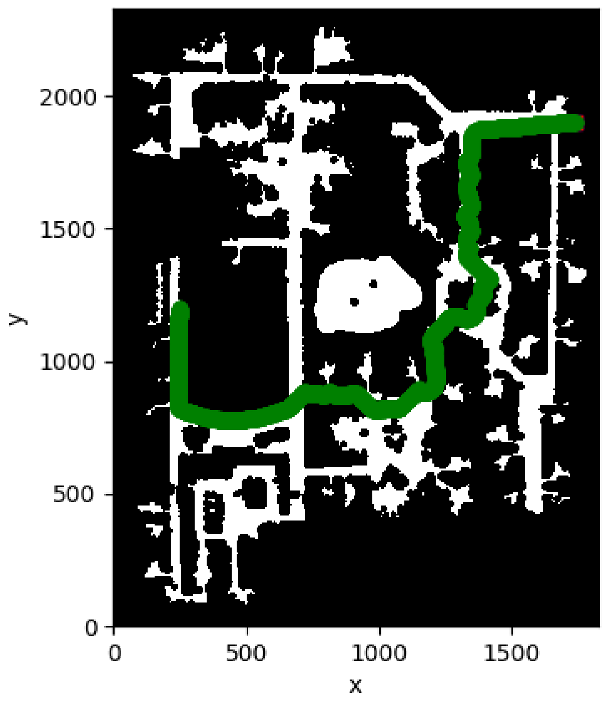

Fig. 2: Map 1b: Delay Weighted A*, Trajectory of P, $\epsilon_1 = 5$, $\epsilon_2 = 1.5$, N=500, r=30

Total Moves: Delayed Weighted A*			
Maps	$\epsilon = 1$	$\epsilon = 2$	$\epsilon = 5$
Map 0	5	4	4
Map 1	1280	1279	1279
Map 2	13	12	12
Map 3	351	223	223
Map 4	9	9	8
Map 5	127	125	89
Map 6	39	40	40
Map 7	*	*	*
Map 1b	*	*	2776
Map 3b	398	524	413
Map 3c	637	776	777

TABLE V: Total moves required to catch the evader for each map using Delayed Weighted A* algorithm with $r = 30$ and $N = 100$. Stars (*) mean results could not be generated due to intractability.

goal is far away the starting position with large amounts of free space. By slightly augmenting the algorithm this drawback can be mitigated but still proves to struggle in larger environments.

To improve on these algorithms further, we could modify the configuration space resolution to scale with the distance from the goal. For instance, if the agent was further away, the grid resolution would be coarser - allowing for a much smaller amount of computations far from the target. As the agent approached the grid resolution could then become finer with an upper limit approaching the original resolution in some radius around the goal.

Alternatively, a different *search-based* algorithm entirely

Difference in Total Moves			
Maps	$\epsilon = 1$	$\epsilon = 2$	$\epsilon = 5$
Map 0	0	0	0
Map 1	*	0	0
Map 2	0	0	0
Map 3	*	0	0
Map 4	0	0	0
Map 5	41	4	2
Map 6	0	0	0
Map 7	*	*	*
Map 1b	*	*	*
Map 3b	*	*	*
Map 3c	*	*	15

TABLE VI: Difference in total moves needed to catch evader (Delayed Weighted A* - Weighted A*). Stars (*) mean results could not be generated due to intractability in either algorithm. Red indicates move moves were necessary in Delayed Weighted A*.

could have been used. Because of the time-constraint on the problem, it would be an ideal situation to employ an *anytime* search algorithm. As the name implies, setting an upper-bound of 2 seconds on any computations would ensure some sort of path, be it possibly sub-optimal, will be generated at every time step.

Additionally, incorporating a *sampling-based* method like Rapidly Exploring Random Tree (RRT*) or a Probabilistic Road Map (PRM) would have been an interesting comparison, as these are often used to overcome the intractability search-based methods are faced with in high-dimensional spaces.

V. Appendix

Figures depicting trajectories and configuration spaces, found below.

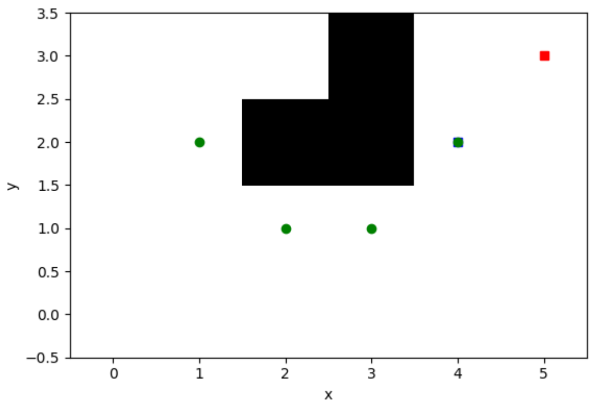

Fig. 3: Map 0: Weighted A*, Trajectory of P, $\epsilon = 1$

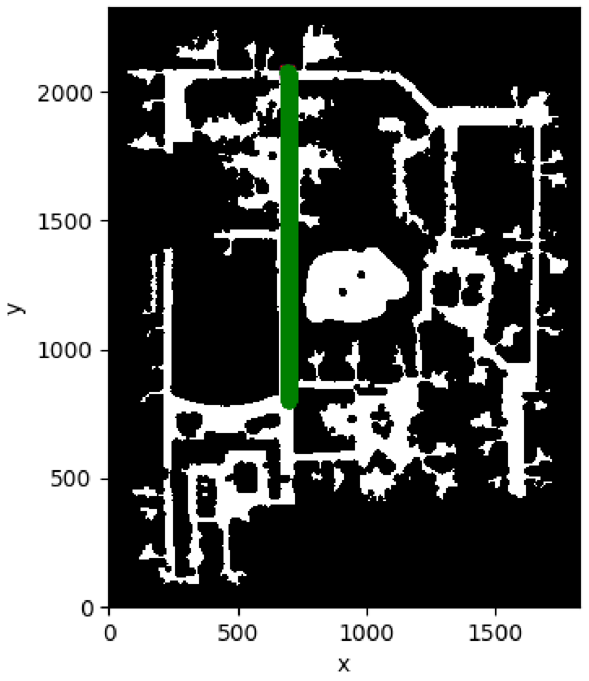

Fig. 4: Map 1: Weighted A*, Trajectory of P, $\epsilon = 5$

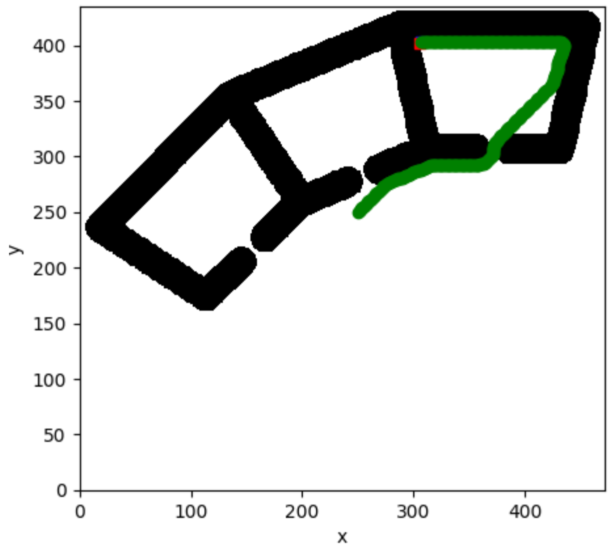

Fig. 6: Map 3: Weighted A*, Trajectory of P, $\epsilon = 1$

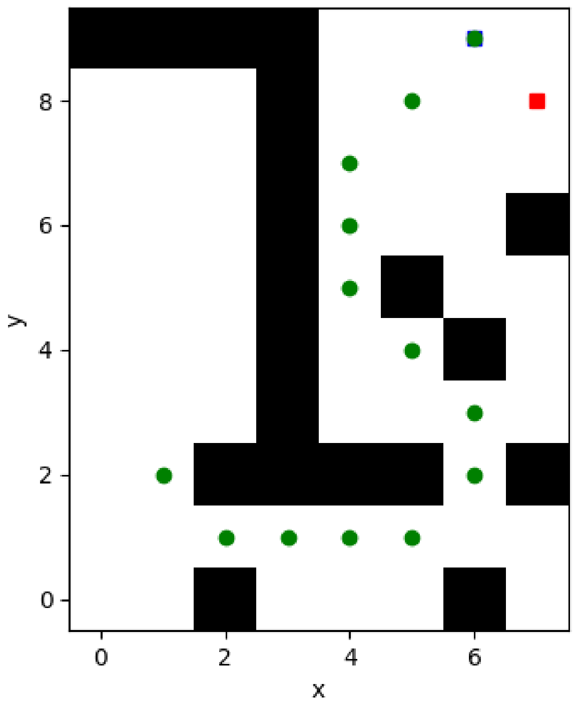

Fig. 5: Map 2: Weighted A*, Trajectory of P, $\epsilon = 1$

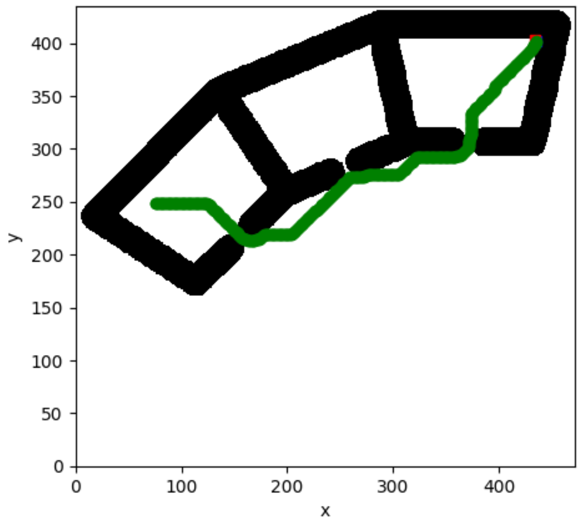

Fig. 7: Map 3b: Delay Weighted A*, Trajectory of P, $\epsilon = 1$

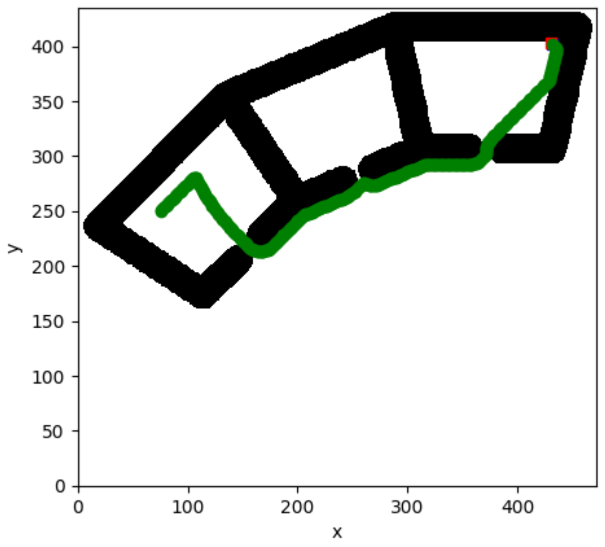

Fig. 8: Map 3b: Delay Weighted A*, Trajectory of P, $\epsilon = 5$

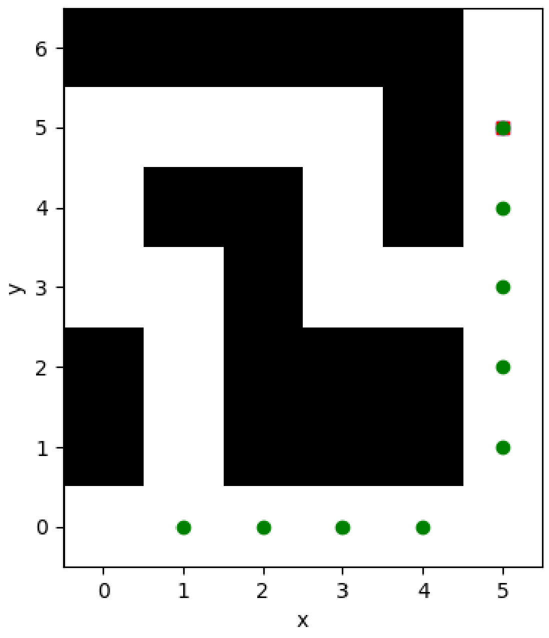

Fig. 10: Map 4: Weighted A*, Trajectory of P, $\epsilon = 1$

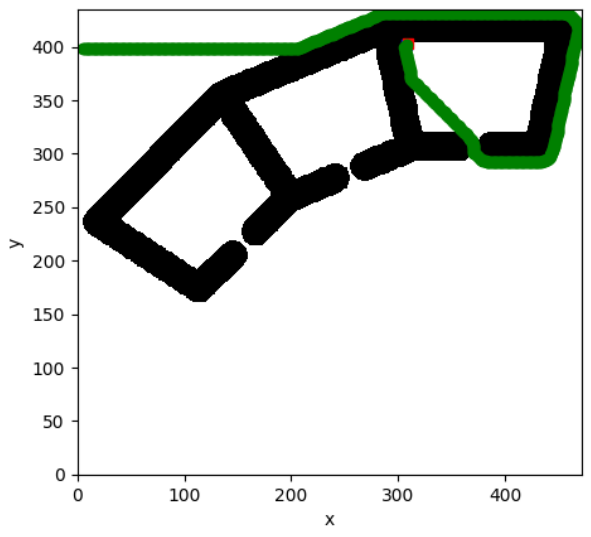

Fig. 9: Map 3c: Weighted A*, Trajectory of P, $\epsilon = 5$

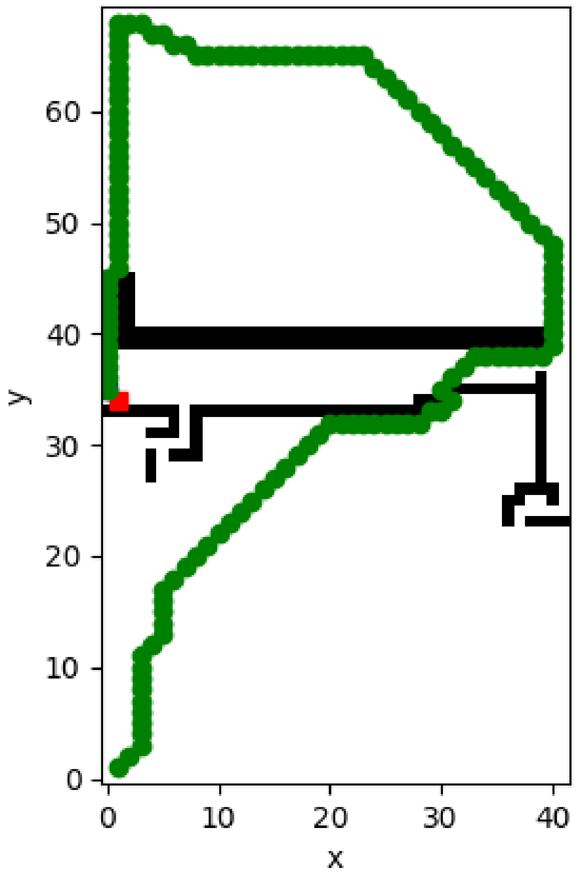

Fig. 11: Map 5: Weighted A*, Trajectory of P, $\epsilon = 1$

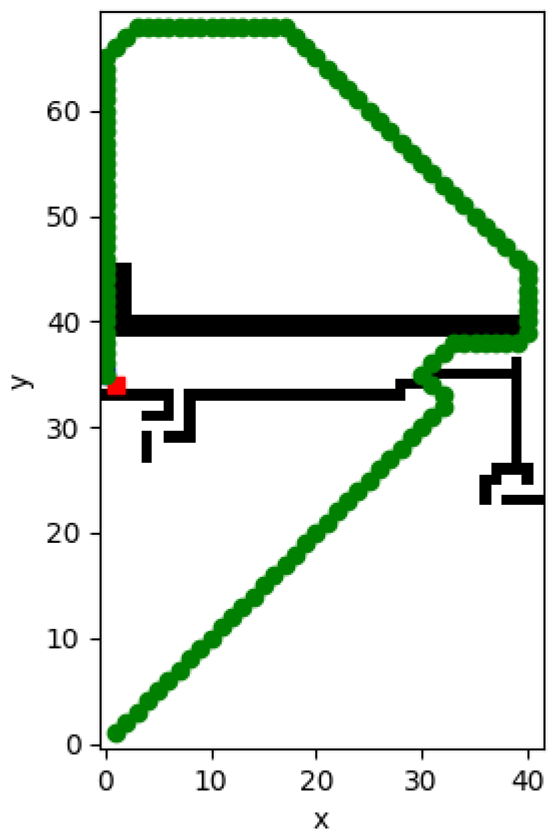

Fig. 12: Map 5: Weighted A*, Trajectory of P, $\epsilon = 2$

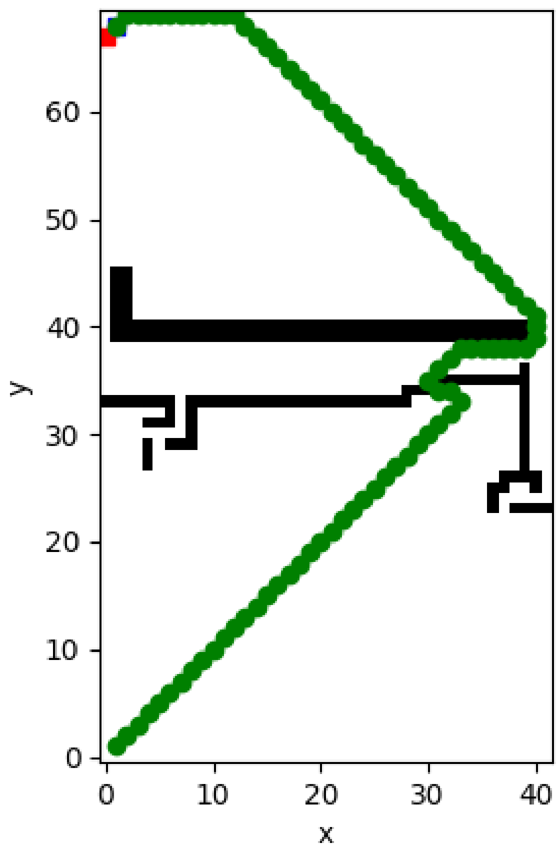

Fig. 13: Map 5: Weighted A*, Trajectory of P, $\epsilon = 5$

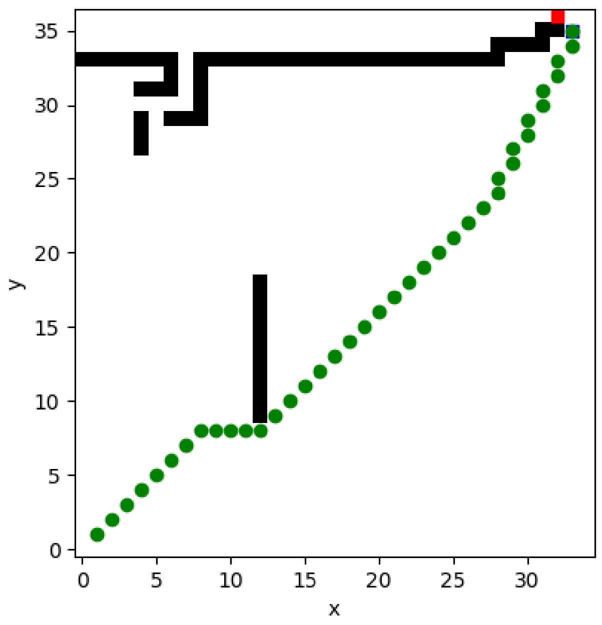

Fig. 14: Map 6: Weighted A*, Trajectory of P, $\epsilon = 1$

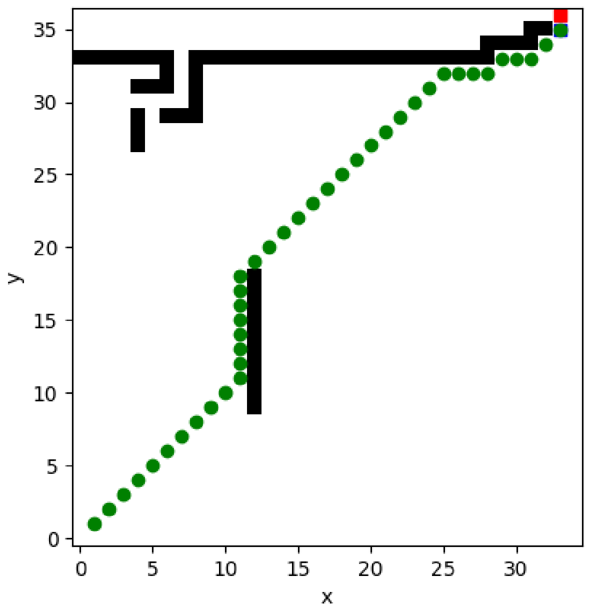

Fig. 15: Map 6: Weighted A*, Trajectory of P, $\epsilon = 2$

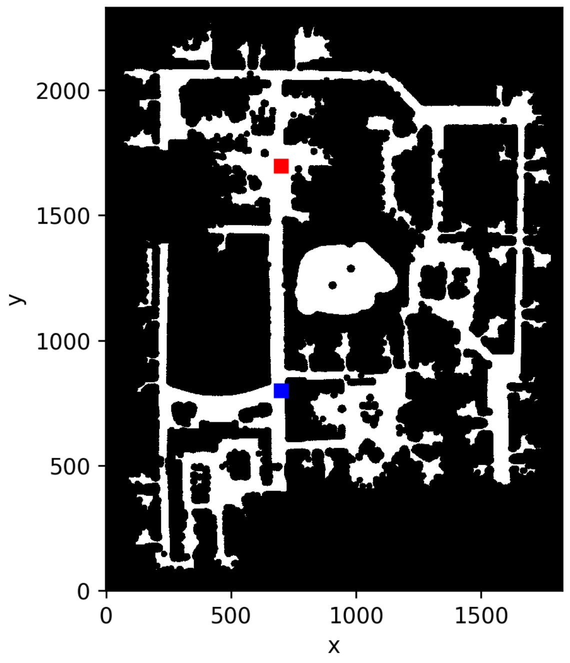

Fig. 17: Map 1 Configuration Space

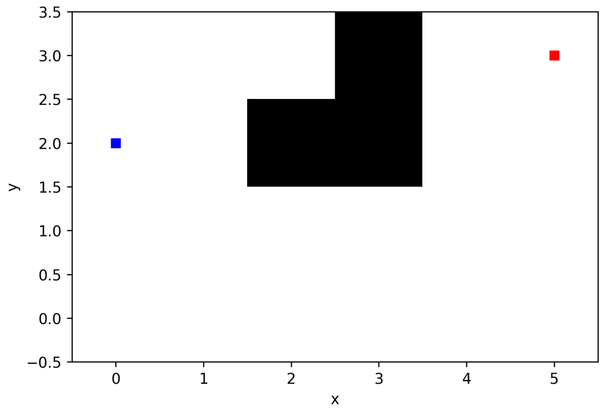

Fig. 16: Map 0 Configuration Space

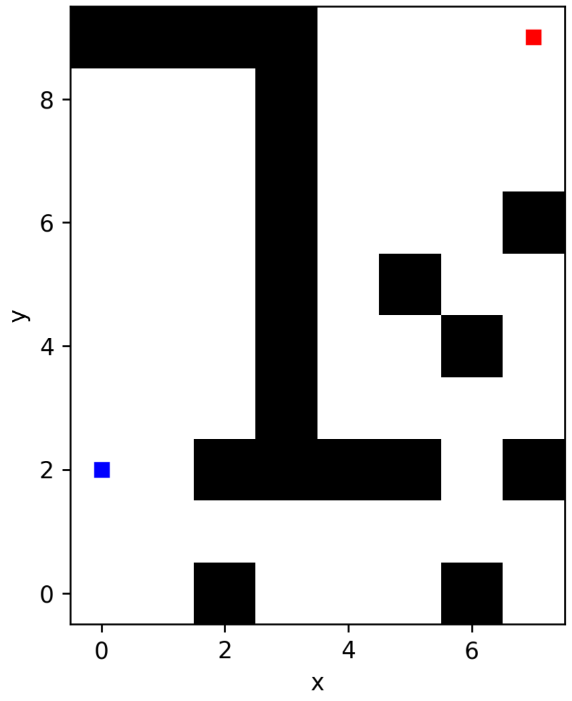

Fig. 18: Map 2 Configuration Space

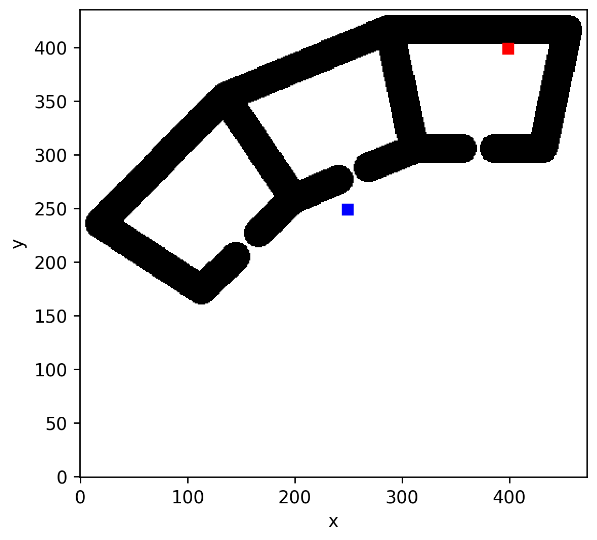

Fig. 19: Map 3 Configuration Space

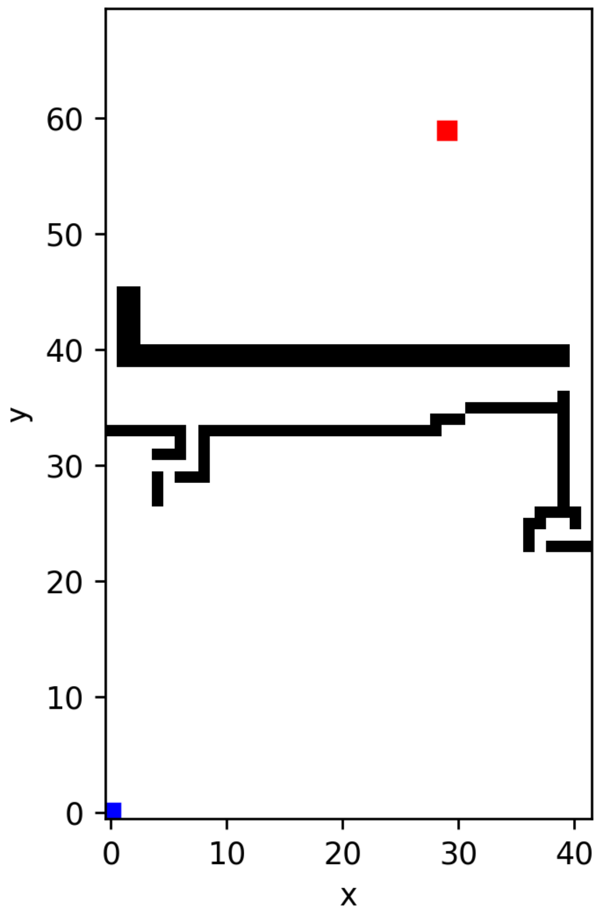

Fig. 21: Map 5 Configuration Space

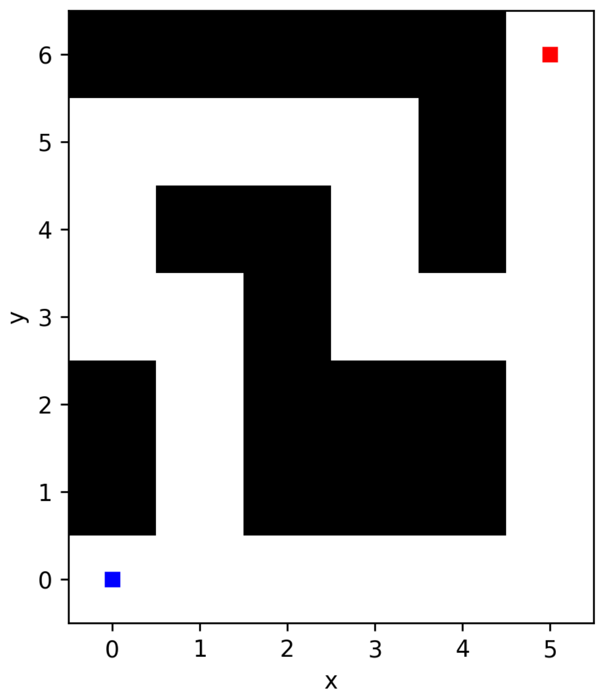

Fig. 20: Map 4 Configuration Space

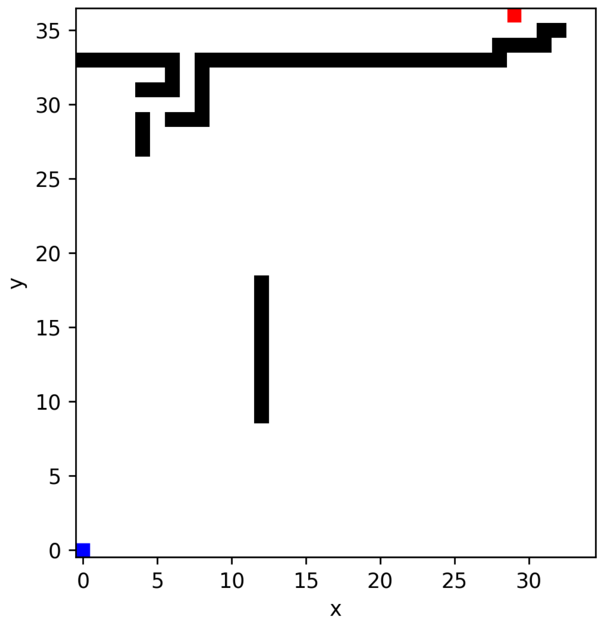

Fig. 22: Map 6 Configuration Space

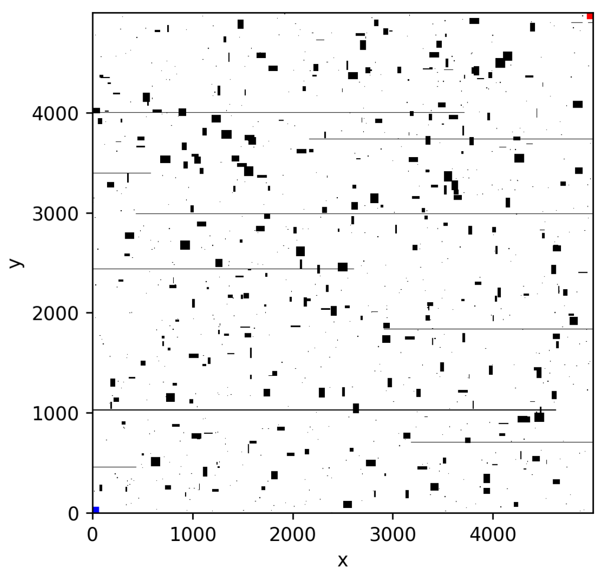

Fig. 23: Map 7 Configuration Space

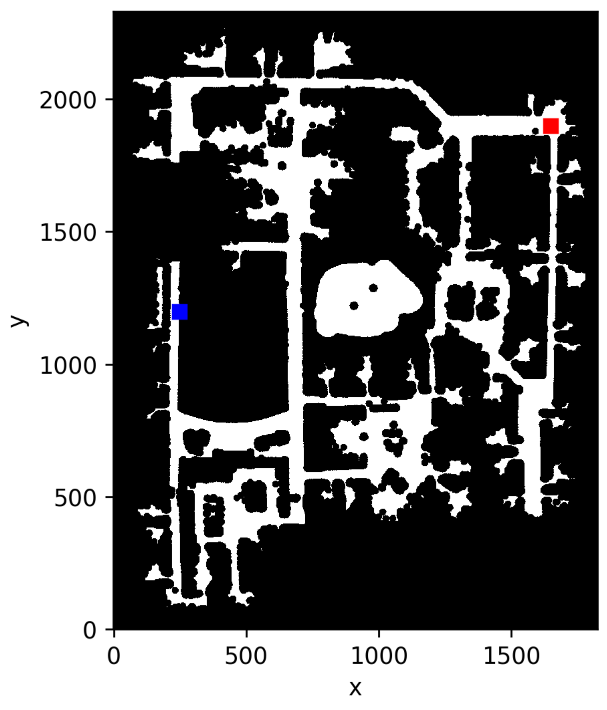

Fig. 24: Map 1b Configuration Space

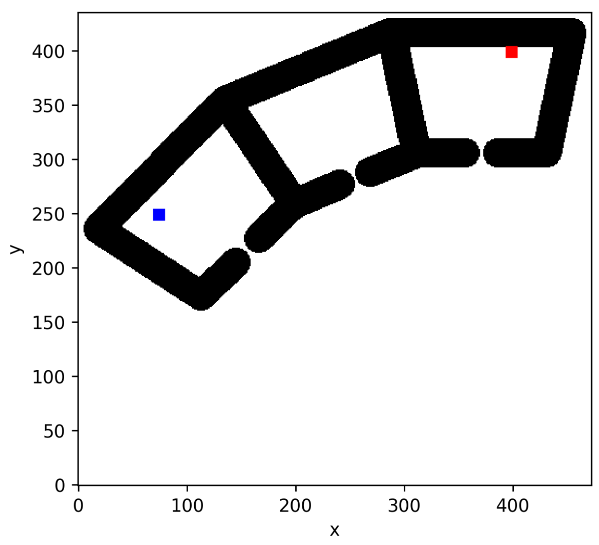

Fig. 25: Map 3b Configuration Space

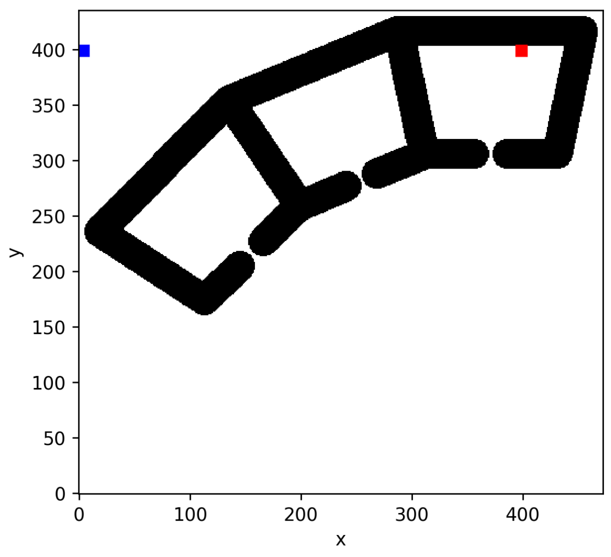

Fig. 26: Map 3c Configuration Space

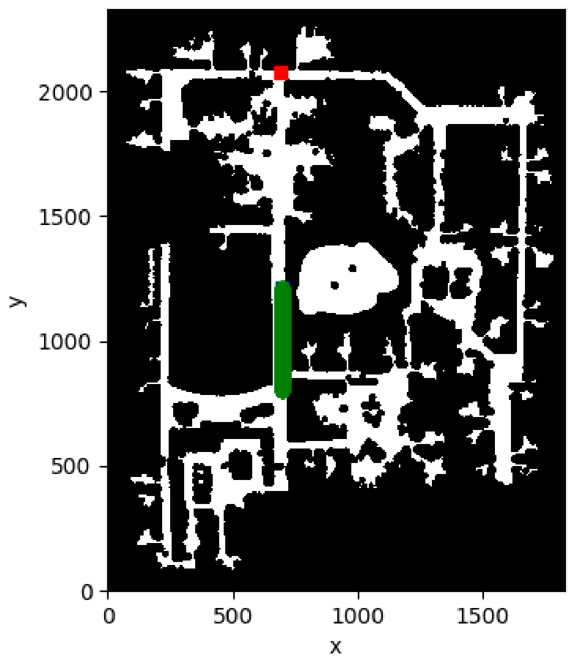

Fig. 27: Map 1: Weighted A*, Trajectory of P, $\epsilon = 1$. Agent Stuck.