Anytime and Incremental Algorithms for Unknown Environments

Jeremy Kilbride
jkilbrid@andrew.cmu.edu

Trevor Stack
tstack@andrew.cmu.edu

Evan Wassmann
ewassman@andrew.cmu.edu

December 12, 2024

Abstract

In this work, compare the popular anytime and incremental algorithms for robot planning. Specifically, we explore the use of these algorithms for the case of a robot navigating in an unknown environment. The algorithms we explore include: A*, Anytime Repairing A*[1], D* lite[2], and Anytime D* [3]. Our code can be found at `https://github.com/JeremyKilbride/Anytime_Dstar`.

1 Motivation

As the number of autonomous system use cases increases, it is important to understand which path-planning algorithm is best suited to achieve a goal given the restrictions of the situation. It is unrealistic to assume that a robot always have full knowledge of its environment and there will be no changes in the operational environment during movement. In a majority of cases, an autonomous robotic system is required to react to changes its environment within a certain time constraint using information from its on-board sensors. In this work, we explore path-planning algorithms to enable a robot to navigate a completely unknown environment using its onboard sensors, which have a finite sensing range. This scenario is analogous to the application of autonomous robots in exploration or surveillance tasks. These robots need to re-plan their paths quickly as they collect new information and detect changes in the environment. Examples of where exploration and surveillance robots are needed are: military surveillance, inspection of uncontrolled environments (such as construction sites or warehouses), autonomous driving in open spaces, and search and rescue operations after disasters. In this work, we implemented and tested three algorithms that are commonly used in autonomous robots to enable quick re-planning. The three algorithms are: Anytime Repairing A* (ARA*), D*lite, and Anytime Dynamic A* (AD*). As a baseline, we compared the performance

of these algorithms to the performance of basic A*. We used several simulated scenarios to test and compare these algorithms.

2 Planning Representation

In this work, we implement Anytime Repairing A* (ARA*), D*lite, and Anytime D* (AD*) to solve a 2-dimensional planning problem. In the problem, a robot must navigate an unknown environment filled with obstacles to a goal location. The goal location and all states are represented by x and y coordinates. The robot moves on an 8-connected grid using a sensor, which detects obstacles within a finite number of cells from the robot in all four directions. The cost function is the number of steps or moves taken by the robot. A diagonal move has the same cost as a horizontal or vertical cost. In D*lite and Anytime D*, we calculated the heuristic using the equation below:

$$h(s) = MAX(|x_s - x_{start}|, |y_s - y_{start}|)$$

Here, (x_s,y_s) is the location of the state and (x_{start},y_{start}) is the location of the start. This is an admissible heuristic as it never overestimates the true cost of reaching the goal. In ARA*, we used an inadmissible heuristic that is calculated using the equation below:

$$h(s) = (\sqrt{2}-1)*MIN(|x_s-x_{start}|, |y_s-y_{start}|+MAX(|x_s-x_{start}|, |y_s-y_{start}|)$$

This heuristic produced more direct paths from the ARA* algorithm because it is equivalent to the minimum distance between a state and the start on an 8-connected grid.

3 Algorithms

We will now give a brief overview of each of the three algorithms we implemented for the task described above, along with details about our specific implementations of each algorithm.

3.1 Anytime Repairing A*

When there is a limit on how long a robot has to compute a path, it may be suitable to quickly compute a suboptimal path and improve upon that path in the time remaining. Anytime Repairing A* accomplishes this by computing an initial path with a large weight on the heuristic (ε), then decreasing (ε) and improving the path by re-using states. To mitigate the computational cost of re-planning from scratch, Anytime repairing A* maintains the open list across all searches and adds inconsistent states to the open list before every search.

In the code, every state has a g-value, v-value, and f-value. The g-value is the cost to get to the state using the least-cost path available. The v-value is

the cost of the state at the time it was expanded. The f-value is calculated using the equation below:

$$f(s) = g(s) + \varepsilon * h(s)$$

When a state is visited during the search, the state's g-value is updated based off of its parent and the state is added to the open list. Its v-value remains infinity until the state is expanded. When a state has a v-value that is greater than its g-value, it is considered overconsistent.

Then, when a state is expanded, its v-value is updated to equal its g-value and it becomes consistent. It is added to the closed list and is not expanded a second time during the current search iteration. However, if during the current search iteration, a better path is found to the state, it is inserted into the inconsistent list. When the next search iteration begins with a smaller ε, this state is removed from the inconsistent list and added to the open list. Every search iteration continues until the f-value of the start state is no longer greater than the f-value of the state at the top of the open list.

Every time ε is decreased, the f-values of the states in the open list are recalculated using the new epsilon. Then, the open list is re-constructed so the states with the smallest f-value are at the top. In our code, this process is computationally expensive because it completely reconstructs `std::priority_queue` open list to re-order the states based off of their new f-values. We decreased epsilon in the order shown below:

$$\varepsilon = 50.0, 5.0, 2.0, 1.5, 1.2, 1.0$$

We decided on these values after experimenting with our algorithm. We found that an ε of 50 generated a suboptimal solution very quickly. The subsequent searches, with decreasing epsilons, gradually expanded more states and improved the path.

Our implementation of ARA* is based on the pseudocode shown in figure 3 of [1].

3.2 D* lite

D* lite is an incremental graph search algorithm, which means that it is meant to quickly solve a series of similar searches by reusing information. More specifically, D* lite recomputes a plan when edge costs in the graph change. It does this by leveraging the idea of local consistency. Local consistency checks answer the following question: "does the current g value of a state match what should be based on the g values of its neighbors?". D* lite uses the rhs value of a state to check local consistency. The rhs value of a state is defined as follows:

$$rhs(s) = \begin{cases} 0 & \text{if } s = s_{goal} \\ \min_{s' \in Pred(s)}(g(s') + c(s', s)) & \text{otherwise} \end{cases}$$

During the actual search, the algorithm searches backward from the s_{goal} to s_{start} . For a robot moving on a grid, s_{start} is set to the robot's current position. The algorithm is designed to only expand inconsistent states. A state is

defined as inconsistent if $g(s) \neq rhs(s)$, further an inconsistent state is defined as over-consistent if $g(s) > rhs(s)$, and under consistent if $g(s) < rhs(s)$. In the case that the search finds an over-consistent state, it sets $g(s) = rhs(s)$, and updates the rhs values of its neighbors. For an under-consistent state the search sets $g(s) = \infty$ and updates the rhs values of that state and its neighbors. D* lite is able to efficiently recompute plans because it only expands inconsistent states that are relevant to the search. Our implementation of the algorithm is all in c++ and is modeled after the psuedocode shown in figure 3 of [2]. Our implementation makes heavy use of data structures in the c++ standard library in order to avoid issues with dynamic memory allocation as much as possible. For the open list, we do not use `std::priority_queue`, but instead we use t`std::set`. This allows us to not only remove states from the top of the queue, but also to remove items from the middle of the queue, which allows us to strictly follow the original implementation of the algorithm. Our implementation could be optimized in a few ways. Firstly, we could implement our own version of a min heap that allows for removal from the top and the middle of the heap. This would allow for faster insertion and searching. Currently, we use a simple linear search when we search the open list for a given state. Apart from implementing our own data structure, we could also implement a more efficient search algorithm for searching the open list. Lastly, there are several optimizations shown in figure 4 of [2], implementing these optimizations would also most make the algorithm more efficient.

3.3 Anytime D*

Our AD* algorithm implements the D* lite algorithm in an anytime fashion and is based off the pseudocode in figures 4 and 5 of [3]. This algorithm performs the search backward and calculates the heuristic values g, and rhs in the same way as D*lite. However, when calculating the key or priority for a state, it includes the weight ε on the heuristic for overconsistent states. The key is computed using the equation below [2]:

$$key(s) = \begin{cases} [rhs(s) + \varepsilon * h(s); rhs(s)] & \text{if } g(s) > rhs(s) \\ [g(s) + h(s); g(s)] & \text{otherwise} \end{cases}$$

Every key is a pair of values $k_1(s)$ and $k_2(s)$. A state's key value is compared using the equation below.

$$\text{key}(s) < \text{key}(s') \iff \begin{cases} k_1(s) < k_1(s') \\ \text{or } \big(k_1(s) = k_1(s') \text{ and } k_2(s) < k_2(s')\big) \end{cases}$$

At $time = 0$, the algorithm creates the start and goal states. For the start state, both the rhs and g values are set to infinity. For the goal state, the rhs value is set to 0 and the g value is set to infinity, making it overconsistent. The goal state is added to the open list and the function $ComputeorImprovePath()$

is called with the first inflated epsilon value (we used the same sequence of epsilons in AD* as we used in ARA* 3.1). Inside the planning loop of $ComputeorImprovePath()$, every overconsistent state taken from the top of the open list is made consistent and the neighboring states are updated using $UpdateState()$. Also, every underconsistent state is made overconsistent. Then, the state and its neighbors are updated using $UpdateState()$. The loop runs while: key_{start} is less than the key of the top element in the open list, or s_{start} is inconsistent.

$UpdateState(s)$ updates the rhs value of the state using the rhs equation shown in the D*lite section 3.2. Then, if the state is inconsistent, the function checks if the state is in the closed list. If it is not in the closed list, it is added to the open list to possibly be expanded in the current search iteration. If it is in the closed list, it is added to the inconsistent list, to be expanded in the next search iteration using a decreased epsilon (similar to ARA*). When the planning loop terminates, the algorithm publishes a current ε-suboptimal path by tracking from the start state to the goal state using the function below: Here s_{curr} is the

while $s_{start} \neq s_{goal}$ **do**
  $s_{\text{next}} = \arg\min_{s \in \text{neighbors}(s_{\text{curr}})} \big(c(s_{\text{curr}}, s) + g(s)\big)$;
  move agent to s_{start};
end

previous state added to the path and s_{next} is the next state to be added to the path. Since the robot moves on an 8-connected grid, the code uses neighbors, or the 8-cells directly next to a cell, instead of predecessors and successors. When constructing the path, the code ensures that the same location is not visited twice by the published path. If there is time remaining, the algorithm conducts subsequent searches with decreasing epsilons to try to improve the current path. As the robot moves, it may detect changes in its environment. In this case, the algorithm identifies the states whose edge cost changed, updates the edge cost, and updates the neighboring states using $UpdateState()$. Then $ComputeorImprovePath()$ is called with the initial inflated ε to propagate the changes to the rest of the cells and construct a plan that avoids obstacles. If the robot does not detect changes in the environment at a step in the robot's movement, and the previous search successfully computed an optimal path using ε=1, the robot simply follows the optimal path. If there are no changes in the environment but the planner did not have enough time to compute an optimal path using ε=1 in the previous step, $ComputeorImprovePath()$ is called using the epsilon that the planner was using in the previous step when time elapsed. This ensures that the planner is continuously working to produce an optimal path at every time step.

For the open list, the code uses a `static std::priority_queue` of `OpenListEntry` structures. These `OpenListEntry` structures store a pointer to a state as well as the state's key at the time it was added to the open list. During the execution of the code, $UpdateState()$ will add an `OpenListEntry` to the open list that may point to the same state as one of the other `OpenListEntries`.

But, the *key* of the recently added `OpenListEntry` will be more up-to-date. In the $ComputeorImprovePath()$ planning loop, the code checks if the top `OpenListEntry` in the open list has an up-to-date *key*. If the *key* is out-of-date, the `OpenListEntry` is removed from the open list and the next `OpenListEntry` in the open list is considered. In $line08$ within the pseudocode's $UpdateState()$, it removes state s from open prior to updating it. The procedure outlined above accomplishes the same task without the computational cost of searching the open list for a specific entry.

An area of improvement in this algorithm's code is the way the open list is re-ordered when ε is changed or the robot moves to a new position. Currently, the code completely reconstructs the open list using `OpenListEntries` with updated *keys*. The new *keys* are calculated using the new ε and heuristic values. This slows down the algorithm's execution because it is very computationally expensive.

4 Experiments

4.1 Experimental setup

We conducted experiments in multiple simulated environments in order to test the performance of our algorithm. In each environment, the robot starts out knowing its current state and the goal state and nothing about the environment. The environment is a 2D eight-connected grid map. Each cell in the map has a binary value showing if that cell is occupied or not. The cost for moving to any neighboring cell s' (diagonally or axis-aligned) is 1 if s' is unoccupied. If a neighboring cell s' is occupied, then the cost to move to that cell is ∞. The robot also has a sensor range of c. This means that the robot "senses" the values of the cells within $\pm c$ of its current position in the x and y directions. Initially, the robot assumes all cells are unoccupied. After initializing the environment, the robot generates an initial plan. After the initial plan is generated, the robot follows the currently published plan until changes in the map are detected. When map changes are detected, the robot recomputes the plan and then continues moving toward the goal. The simulation terminates when the robot reaches the goal. An example of the maps used can be seen in figure 1. Other maps used for the simulated scenarios can be seen in appendix B. For each environment, we run the simulation with the planners described in section 3 along with A* using a forward search as a baseline.

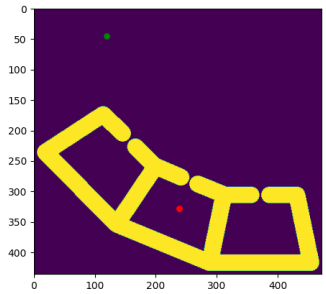

Figure 1: Visualzation of `new_map3` used for robot navigation scenarios. Map size is 473×436. The green dot is the robot start, the red dot is the goal, yellow cells are occupied, purple cells are free.

4.2 Results and Discussion

Figures 2 and 3 shows examples of the paths generated by our algorithms in the simulated environments. It is quite evident from these figures that the robot's overall path tends to be suboptimal since it does not have full knowledge of obstacles.

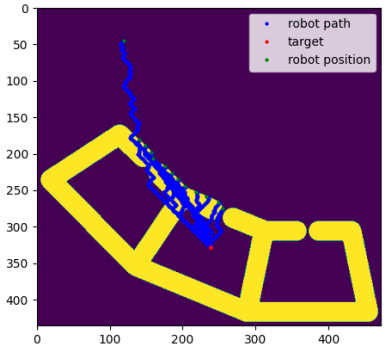

Figure 2: Paths generated as the robot moves in an unknown environment using the A* planner.

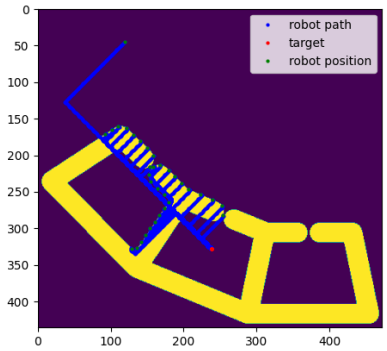

Figure 3: Paths generated as the robot moves in an unknown environment using the D* Lite planner.

Table 1: Average number of states expanded per search

map	Sensor range	A*	ARA*	D* Lite	AD*
`new_map1/map1`	1	10,434.4	3539.2	118.0	630.8
	5	12,836.5	4104.9	104.4	132.0
`new_map3/map3`	1	1562.9	421.6	141.8	85.3
	5	1609.6	1401.7	136.5	91.1
`new_map5/map5`	1	1172.5	199.0	103.0	50.1
	5	1356.7	273.4	174.2	76.1
`new_map8/map8`	1	6864.2	235.6	290.8	76.0
	5	4027.0	554.6	223.8	91.0
`new_map9/map9`	1	3683.3	405.3	207.1	94.1
	5	4055.4	403.3	177.1	89.2
`new_map13/map13`	1	226.5	119.5	35.2	57.4
	5	213.2	171.08	44.5	66.59

Table 2: Number of steps taken by each planner, the optimal column shows the length of the path computed by A* with full knowledge of the map

map	sensor range	A*	ARA*	D* Lite	AD*	optimal
new_map1/map1	1	1313	1357	1100	1355	421
	5	1337	1307	1347	1067	
new_map3/map3	1	316	1067	608	608	283
	5	554	633	600	604	
new_map5/map5	1	214	224	249	199	199
	5	230	223	199	199	
new_map8/map8	1	596	419	410	410	410
	5	591	415	410	410	
new_map9/map9	1	376	599	551	549	350
	5	375	396	551	549	
new_map13/map13	1	209	249	205	207	106
	5	180	176	182	183	

Table 3: Average planning time (ms) per step by each planner

map	sensor range	A*	ARA*	D* Lite	AD*
new_map1/map1	1	7.25	8.75	5.10	63.13
	5	7.61	10.54	5.84	61.63
new_map3/map3	1	0.80	0.82	3.02	5.40
	5	0.58	2.94	3.42	9.64
new_map5/map5	1	0.54	0.40	2.19	1.99
	5	0.64	0.528	3.09	2.32
new_map8/map8	1	3.26	0.60	7.62	4.48
	5	1.26	1.20	5.93	4.55
new_map9/map9	1	1.79	0.897	7.41	5.18
	5	1.26	0.864	6.98	5.54
new_map13/map13	1	0.54	0.41	1.12	1.61
	5	0.55	0.61	1.57	1.44

Tables 1-3 show the results of running our planners in 5 different environments. For these experiments we set the sensor range to 1 and 5. Additionally, for all simulations we set the max planning time for ARA* to 10ms. By this, we mean that each time we call ARA* the algorithm will decrease ε as much as possible within 10 ms.

The maximum planning time for AD* fluctuated based on the complexity of the map. For example, AD* can navigate to the goal in map5.txt with a sensor range of 5 and a max planning time of 2 ms. However, on map1.txt (which is much larger and more complex than map5.txt) and a sensor range of 5, AD* requires a max planning time of around 150 ms or higher to successfully navigate to the goal. When AD* executes, the entire maximum planning time is only used

at step 0 (when computing an initial plan) or when the robot detects significant edge cost changes. As you can see from 3, the average planning time per step for map1.txt was 61.63 ms, much less than 125 ms. The AD* algorithm can achieve better performance if, when significant edge cost changes are detected, it increases ε or replans from scratch. This is an area of improvement for future work. The paths the robot took on each map using the AD* algorithm are shown in A.

One result that is consistent across all algorithms is that as the sensor range is increased, the number of states expanded in a single planning episode increases. This is because the robot is receiving more information about the environment at each step, so at any given time in the simulation with a range of 5 the map that the robot has will have as many or more obstacles as would be the case with a range of 1. Thus, for any given time in the simulation, the robot within the greater sensor range will expand as many or more states as the robot with the lower range, since the robot with the greater range has discovered more obstacles.

It can be seen from Table 1 that D* lite and AD* expand significantly fewer states each planning step compared to A* or ARA*. However, in our case, re-planning from scratch with A* or ARA* is often faster because of the computational cost of the D* lite and AD* algorithms. In our A* and ARA* implementations, the open list is a `std::priority_queue`, whereas the open list in D* lite is a `std::set`. As discussed in section 3.2, using a `std::set` for the open list is quite inefficient and could be improved with a more custom implementation of a min heap. We believe that more efficient data structures would allow D* lite to beat A* not only in number of states expanded but also in computation time.

The AD* algorithm not only generates states, but also `OpenListEntries` that each store a pointer to a state and a *key* value. The addition of a second structure makes the execution of the AD* algorithm significantly more computationally expensive. This, in conjunction with how frequent the open list is reconstructed (as discussed in 3.3), makes AD* slower than A* and ARA*.

Further, we see that sometimes the robot makes more moves when using the D* lite and AD* planner when compared to A*. This stems from the fact that the two algorithms reconstruct the path in slightly different ways. A* uses back pointers to the best predecessor, whereas D* lite and AD* use a backward search and calculate the best successor (or neighbor) as $\text{argmin}_{s' \in Succ(s)}(c(s, s') + g(s'))$. In the case where two or more neighbors of a state have equal g, the two algorithms will most likely break the tie in different ways. This seems to cause the algorithms to take different paths around obstacles, which can clearly lead to highly suboptimal behavior when the robot only has partial knowledge of the environment. For example, in 8, the robot takes the long way around the obstacle when using AD*.

Another possible cause of this sub-optimality is our cost function. If diagonal moves had an increased cost of $\sqrt{2}$ instead of 1, and we modified the heuristic function to the heuristic used by ARA* (see 2), our D* lite and AD* algorithms

may be able to generate more direct, lower cost paths.

5 Conclusion

In this work, we compared three different algorithms that can be used for efficiently re-planning in unknown environments: Anytime Repairing A* (ARA*), D* lite, and Anytime Dynamic A* (AD*). Due to the condition that the robot has a finite sensor range and only partially knows the environment around it, these algorithms produce suboptimal paths. This suboptimality can be seen in table 2. However, all algorithms successfully navigated the robot to the goal in all scenarios. Overall, we find that the best algorithm is dependent on the environment and the constraints on planning. Given an environment with many complex obstacles and no constraint on planning time, the D* lite is the best algorithm since it is able to reuse a vast majority of previously computed states. When looking at the statistics from Tables 1 and 3 for the `new_map1` environment, D* lite expands $\sim$ 100 times fewer states than A*, resulting in faster computation, even with the inefficiencies of our specific D* lite implementation. If there is a constraint on planning time in a complex environment, AD* would be the best algorithm to use. Looking at the statistics in Tables 1 and 3 for the `map1`, AD* expands $\sim$ 16 times fewer states than A*. However, its average planning time per step is $\sim$ 8 times higher. A focus for future work will be to optimize the AD* algorithm to decrease planning times.

6 How to Compile and Run our Code using CMAKE

Navigate to the `Anytime_Dstar` folder in your Developer Command Prompt and follow these steps:

1. Enter `mkdir build`. Navigate to the `build` folder.
2. Enter `cmake ..`
3. Enter `cmake --build .  --config release`
4. Enter `cd release`

To run our algorithms, use the following command:

```
planner [map file] [planner number] [sensing range]
```

- `[map file]`: The map you want to run the algorithm on. **Note:**
  - Use `new_map#.txt` for A* and D* Lite algorithms.
  - Use `map#.txt` for ARA* and AD* algorithms.
- `[planner number]`: The algorithm you want to run.

  - Enter `0` for A*.
  - Enter `1` for D* Lite.
  - Enter `2` for ARA*.
  - Enter `3` for AD*.
- `[sensing range]`: The integer sensing range that you want the robot to have.

To visualize the robot's path:

- Navigate to the `scripts` folder.
- For ARA* and AD*:
  - Enter: `python visualizer_v1.py ../maps/map#.txt`
  - This plots the path the robot takes to reach the goal. The entire map is shown.
- For D* Lite and A*:
  - Enter: `python visualizer_v2.py ../maps/new_map#.txt [sensorRange] [Speedup]`
  - A speedup number of 2 to 5 will work.
  - This plots every path the planner computes as the robot moves towards the goal. Only the portions of the map that the robot knows are shown.

Example Entries:

- To run AD* on `map3.txt` with a sensing range of 5:
  - In the `release` folder, enter: `planner map3.txt 3 5`
  - In the `scripts` folder, enter: `python visualizer_v1.py ../maps/map3.txt`
- To run D* Lite on `new_map3.txt` with a sensing range of 5:
  - In the `release` folder, enter: `planner new_map3.txt 1 5`
  - In the `scripts` folder, enter: `python visualizer_v2.py ../maps/new_map3.txt 5 5`

References

[1] M. Likhachev, G. J. Gordon, and S. Thrun, "Ara∗ : Anytime a∗ with provable bounds on sub-optimality," in *Advances in Neural Information Processing Systems*, S. Thrun, L. Saul, and B. Schölkopf, Eds., vol. 16, MIT Press, 2003. [Online]. Available: `https://proceedings.neurips.cc/paper_files/paper/2003/file/ee8fe9093fbbb687bef15a38facc44d2-Paper.pdf`.

[2] S. Koenig and M. Likhachev, "D∗ lite," in *Eighteenth National Conference on Artificial Intelligence*, Edmonton, Alberta, Canada: American Association for Artificial Intelligence, 2002, pp. 476–483, ISBN: 0262511290.

[3] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun, "Anytime dynamic a∗: An anytime, replanning algorithm.," in *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, Jan. 2005, pp. 262–271.

A AD* Performance

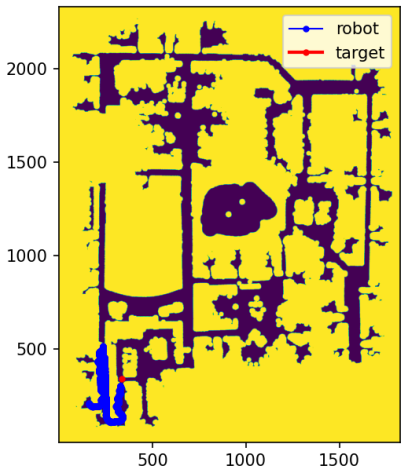

Figure 4: Performance of AD* on `map1.txt`, Planning Time Constraint: 750ms, Sensor Range: 5

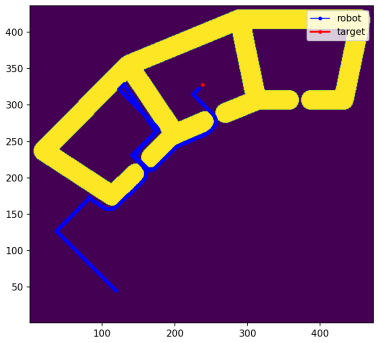

Figure 5: Performance of AD* on `map3.txt`, Planning Time Constraint: 25 ms, Sensor Range: 5

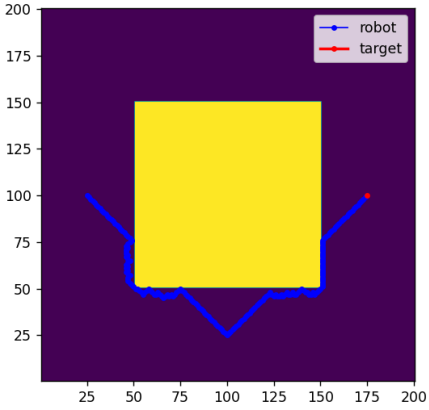

Figure 6: Performance of AD* on `map5.txt`, Planning Time Constraint: 2 ms, Sensor Range: 5

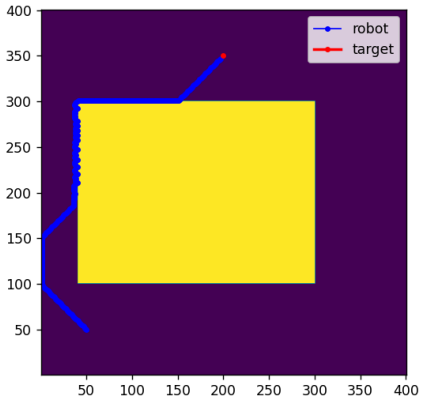

Figure 7: Performance of AD* on `map8.txt`, Planning Time Constraint: 5 ms

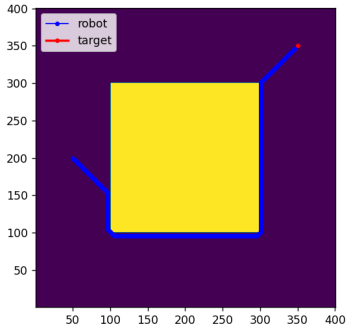

Figure 8: Performance of AD* on `map9.txt`, Planning Time Constraint: 10 ms, Sensor Range: 5

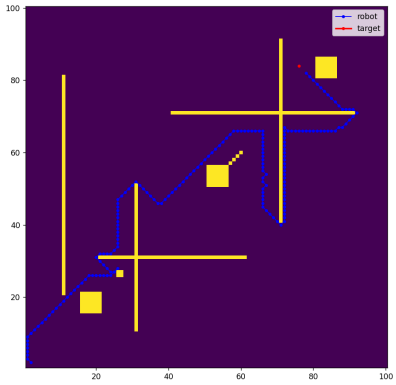

Figure 9: Performance of AD* on `map13.txt`, Planning Time Constraint: 3 ms, Sensor Range: 5

B Maps

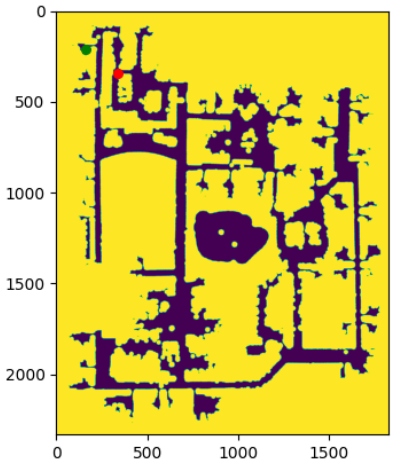

Figure 10: visualization of `new_map1`, map size is 1825×2332

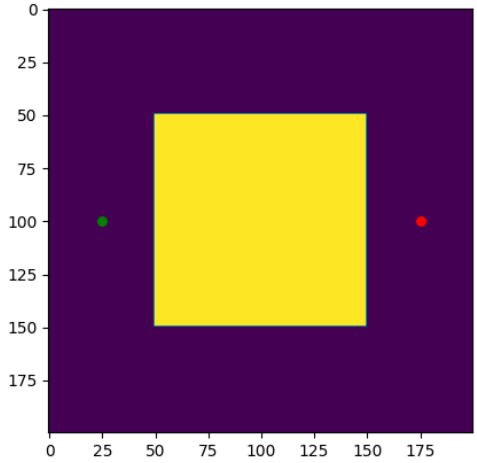

Figure 11: visualization of `new_map5`, map size is 200×200

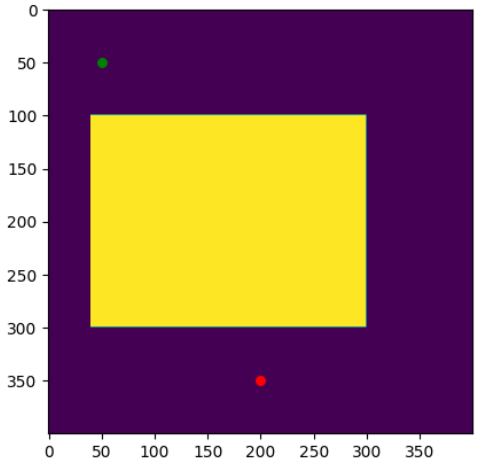

Figure 12: visualization of `new_map8`, map size is 400×400

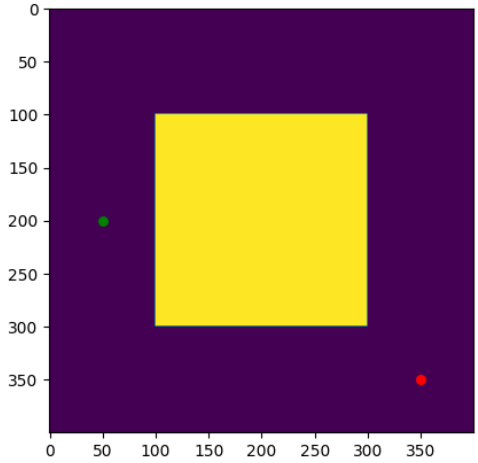

Figure 13: visualization of `new_map9`, map size is 400×400