# PUI Probabilistic Planning report

Prokop Černý

May 2020

## 1 Agent designs

Our task is to solve a navigation problem in a grid maze, with the desired actions failing some percentage of the time - agent is moving in a stochastic environment. As the probabilities are fixed, and the probability of the selected action succeeding is over 50%, we have decided that planning with a determinized version of the world is a suitable option. To that end, we have created two agent designs.

### 1.1 A* agent

Our first agent is an agent based on the FF-Replan principles. This agent creates its plan on a determinized version of the maze, where the actions don't fail. Planning a path in such a grid is a classical graph problem with many ways to solve it.

As our determinized maze is basically a uniform cost graph, we could use BFS, as that is the algorithm with optimal time complexity for finding shortest paths in a uniform cost graph. But we can do better, as our maze is a grid, which has additional structure compared to a general graph, namely that each grid cell has some coordinates.

We can exploit that and employ an informed search algorithm. We selected A* with the following heuristics.

- None - with no heuristic, A* degenerates into Dijkstra's algorithm, which on a uniform cost graph behaves like BFS.
- Euclidean distance - straight air distance, when movement in any direction is permitted, defined as $\sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2}$ for two points[1] $a, b \in \mathbb{R}^2$.
- Manhattan distance - distance when only movement in the direction of main axes is allowed, defined as $|a_1 - b_1| + |a_2 - b_2|$ for two points[1] $a, b \in \mathbb{R}^2$.

After the agent finds an optimal path using A* in the determinized maze, it then tries to execute the steps it has found. If an action succeeded, it shortens its saved path by one, removing the action it has just performed. If the task failed, there are two options. The action failed in such a way that the agent has not moved, in which case it doesn't modify its saved path, and tries to repeat it next time. If an action failed that the agent moved to an unexpected location, it discards it's plan and runs A* from it's current location to the exit on the determinized maze.

Using this strategy, the agent takes the optimal path to reach the goal, and only computes it's path on demand.

However, there can be an even more efficient solution for our maze problem.

[1] Note that in the maze, all points $a, b \in \mathbb{N}_0 \times \mathbb{N}_0$

### 1.2 Navmesh agent

Our A* agent is good in the way that it only finds plans on demand, but it can often happen that when an action fails and takes it to an unplanned spot, it then redoes much of the search again, as large parts of the optimal paths to the exit are probably usually shared, so it ends up expanding a lot of nodes several times when being forced to replan.

One solution would be to save for each full plan to the exit the correct action for each cell on the path of the plan. That way, when replanning the algorithm would try to expand a node which it already expanded, it could stop the search there, and use most of the path it has already found before. But this can lead to using suboptimal plans, as a previously saved node might not actually lie on the real optimal path from the current location to the exit when the algorithm tries to expand it.

But here comes an opportunity to exploit our maze environment even further. As the environment does not change, and can easily fit into memory, we can run a single BFS flood fill from the exit, and assign to each cell its distance from the exit. Due to the way BFS explores the grid, we are guaranteed that the assigned distances to each cell are actually the optimal distances from the cell to the exit. After performing this, we obtain a maze filled with numbers representing the BFS distance to the exit. We shall refer to this maze as the navmesh.

The agent computes the navmesh once, before it starts to move in the maze. Once it computes the navmesh, navigation is very simple. All it has to do is find the neighboring cell to its current location which has the lowest value in the navmesh, and try to move to it. Whether the desired action fails or not does not matter, as the agent has the navmesh for all reachable cells in the maze. The agent repeats the same procedure until it reaches a cell with navmesh value 0, which only the exit location has.

## 2 Theoretical analysis of selected agent designs

We believe that our Navmesh agent design is one of the best solutions for our maze environment, where the maze is fixed, e.g. the environment does not change, the probabilities of action failure are lower than 50%, and that the maze fits reasonably in memory. It will always take the most optimal path from the location it finds itself in, and only the failures of action can prolong it's journey. It has an advantage in only exploring as many nodes as there are reachable cells in the maze, and it does not need to ever replan. It knows the optimal decision for each position it finds itself in, as it precomputes the navmesh.

But our Navmesh agent is not suitable when the maze itself does not fit completely into memory. If that would be the case, but the problem is otherwise the same, then the A* based agent becomes a better option (assuming it would still be able to plan the optimal journey, e.g. the maze would still be somehow accessible even despite it's size, either being available in chunks, or some query on whether a cell is a wall or not).

The A* agent also follows optimal paths, same as the Navmesh agent, but does not explore the entire maze during a single search, unlike navmesh which explores the entire maze. But the downside that Navmesh agent explores only once, but A* agent makes potentially a lot of searches, so the total number of explored nodes is usually higher.

Both of our agent designs become insufficient when the environment becomes adversarial, e.g. the rate of action failure would be over 50%, or some other inconvenient probability distribution over the actual performed action. In this case, the classic MDP techniques such as Value Iteration would be much better, as they would handle this adversarial environment without any modification.

Another case in which A* and Navmesh agents would be insufficient is where there are some failure locations in the maze. In our problem, we only have one goal location, which can be considered

a positive reward, and otherwise all locations don't give a reward. If our maze had some failure locations, such as bottomless pits, lava, or something else, where the agent unsuccessfully ends (gets a negative reward), then our agents wouldn't be able to avoid them, as they only look for shortest paths to the goals.

If there would be failure locations in the maze, MDP techniques (Value/Policy Iteration) would create policies which take these locations into account (as negative rewards), and will try to avoid reaching them, e.g. try to choose action to not fall into them, even if it means not selecting the action on the shortest path. Reinforcement Learning techniques such as Q-Learning or SARSA could also be employed in this case successfully.

But as our mazes don't have failure states and mazes do fit in memory, our Navmesh and A* approaches are computationally much better suited than VI or Reinforcement Learning techniques to this specific problem.

# 3 Experiments

As all our designs follow the most optimal path they can to the exit location, we decided to measure the amount of how many times the agent had to (re)plan, how many nodes it has expanded in total, and how many steps it has taken to reach the exit location.

We have run our experiments for four agents, to which we'll refer by names in table 1.

| | |
|---|---|
| **Dijkstra** | A* agent without a heuristic, behaves like BFS |
| **A*Euclid** | A* agent using the euclidean distance heuristic |
| **A*Manhattan** | A* agent using the manhattan distance heuristic |
| **Navmesh** | The Navmesh agent |

Table 1: Agents used in experiments

Our testing data consists of mazes of five different radii, with each radius having five mazes, for a total of 25 mazes. As the mazes are squares, each has a side of length $2 \cdot radius + 1$.

We have then ran all of the agents 100 times on each maze and recorded how many nodes in total they've expanded, how many steps they've taken and how many times they had had to (re)plan.

We set the same seed before each of the 100 runs, so that the each agent would experience the same maze in the same conditions, while still solving each maze 100 times with each agent with a differently behaving environment.

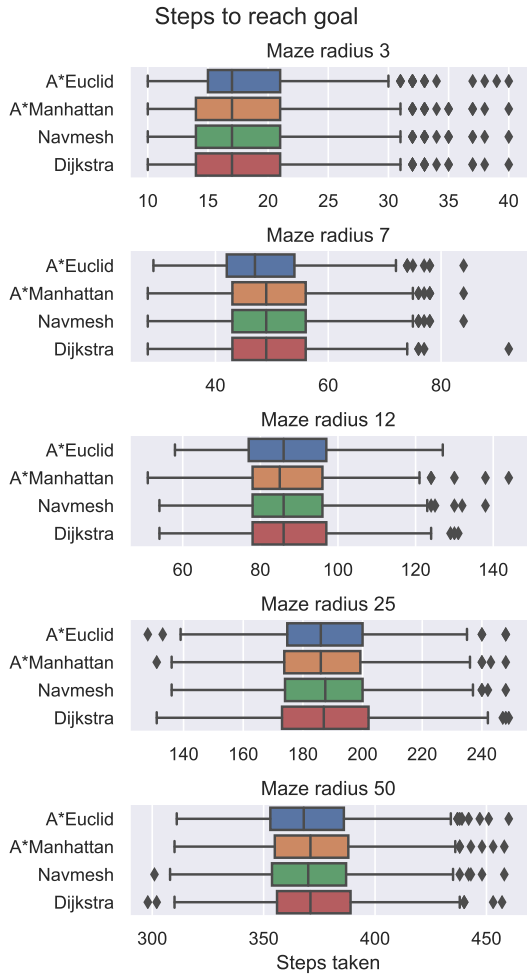


Figure 1: Comparison of the amounts of steps taken to reach the exit.

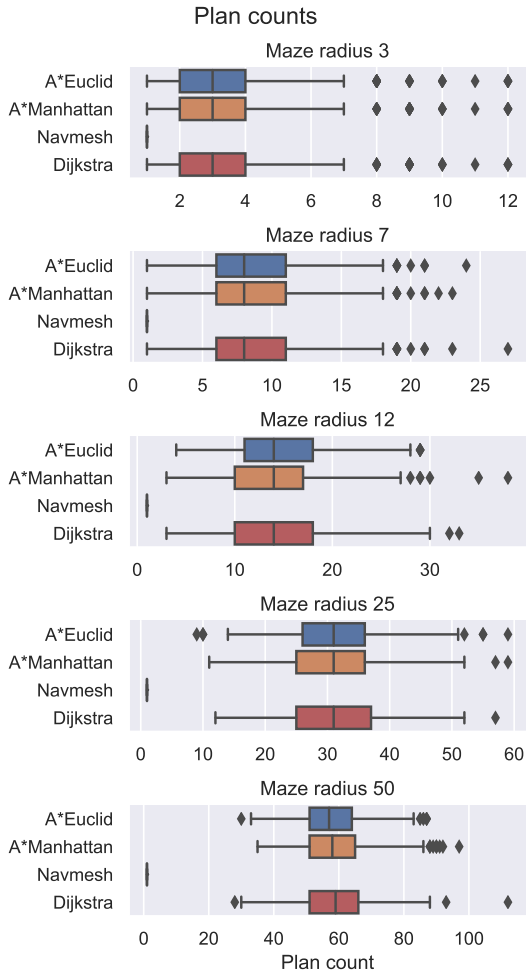


Figure 2: Comparison of amounts of plans that were created by each agent.

As we can see in figures 1, 2 and 3, we grouped the experiments by the radius of the maze.

Figure 1 shows us how many steps it took each agent to reach the exit. As we can see, all agents behave quite similarly. This is because all of them always try to follow the optimal path to the exit from their current location, and therefore take quite similar amount of steps.

Figure 2 tells a similar story as fig. 1. With the exception of the Navmesh agent, which creates one plan, all A* based agents had to usually create the same amount of plans, although there is some variation.

Actually, variation in both fig. 1 and fig. 2 can be a bit unexpected, as we have said that we reset the random seed to the same value before the 100 repetitions for each agent/maze combination, so one could expect that since all algorithms follow optimal paths, that they would perform exactly identically using these measures.

But the variation can be explained. Although the initial conditions are always reset so they are the same, there are many optimal paths from a location to the exit in these mazes, and the algorithms find different optimal paths. In the A* agents this is given by how the heuristic orders the nodes in the

priority queue, and in the Navmesh agent, this is affected by the order the agent checks the current location's neighbors in the navmesh when searching for the smallest neighbor value.

But now, as we get to fig. 3, we observe differences between the algorithms. Figure 3 shows how many nodes each agent expanded. The navmesh agent always expanded precisely the amount of reachable cells in the maze. We can see that on mazes of smaller radius, the A* based agents could sometimes expand less nodes.

That is because they didn't encounter any failures which would make them need to replan, or the amount of replans was low and the heuristic well informed the algorithm that it didn't expand many nodes when searching for the correct path.

But such cases are rare, as most of the time, the A* agents expand dramatically more nodes in total compared to the Navmesh agent.

Another thing fig. 3 tells us is that the Manhattan heuristic is very good for our mazes, is it results in the fewest expanded nodes in general for all maze radii. It is followed by the Euclidean distance heuristic, with the gap increasing on larger and larges radii.

This is most likely because the Manhattan distance accurately describes the way in which the agent can move, while using Euclidean distance is not accurate for our agent.

The Dijkstra agent expands the most nodes, with it being worse and worse as the maze size increases.

The main takeaway of fig. 3 is that the Navmesh agent is the best choice for performance reasons, especially as the size of the maze increases. The fact that it does not need to replan and expands only a constant amount of nodes for a fixed maze size is a huge boon compared to the A* agents.

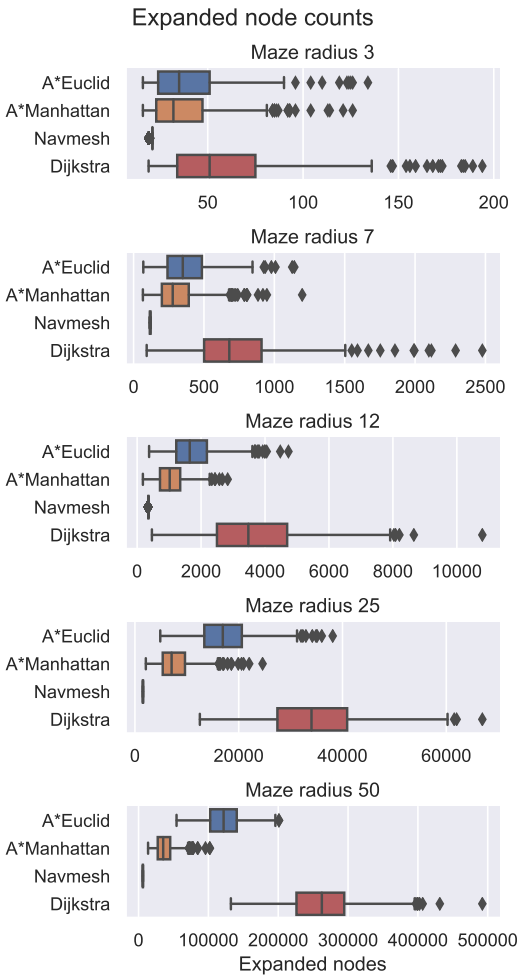


Figure 3: Comparison of the total number of expanded nodes by each kind of agent when planning the paths to reach its goal

# 4 Conclusion

We have created several agents designed specifically for our maze enviroments, and empirically shown that for this particular problem domain, the Navmesh agent is undisputedly the best option compared to an A* based approach.

While very powerful, we argued that classical MDP approaches and Reinforcement Learning are overly complex tools, which would excel if the problem was modified in some ways, for which we've provided examples, but are unnecessary for this version of maze traversal.