Motion Planner: Discrete Planners
===========================

This document serves as an analysis on the performances of a random planner and an optimal planner. The code used to generate figures and results is present in `analyze_planners.py`

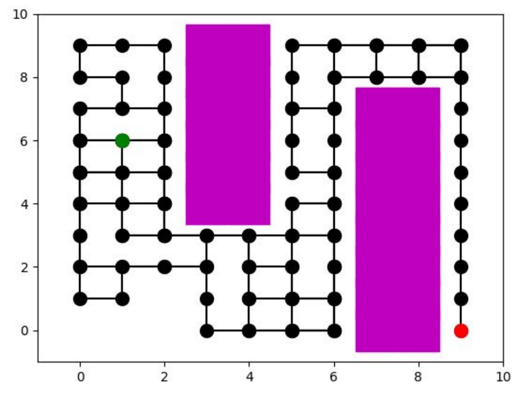
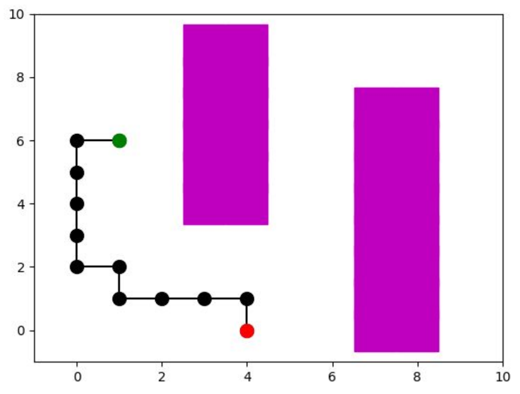
It is important to note right off the bat that a *complete analysis* would require computing all the relevant metrics over a *distribution of workspaces*. Due to time constraints, analysis on only one workspace could be performed, therefore all statistics shown here are at best an approximation.

Sample Executions

Two examples of each planner along with a brief description of the parameters used are provided to showcase their working. Identical workspaces are used for both planners. The workspace is a uniform square grid of 10 units each side. As per problem statement requirements, the grid permits only orthogonal movements, with uniform costs attached to all directions.

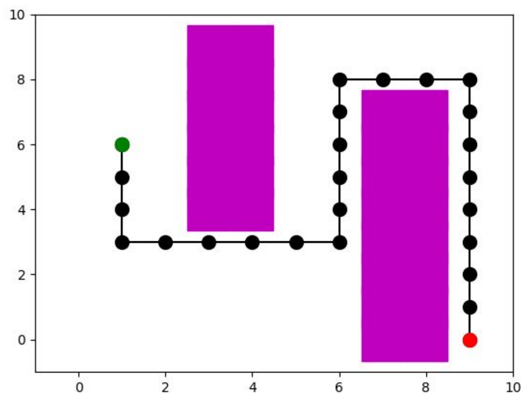
In all example plots, the green node is the start query point or “robot_pose”, the red node is the end query point or the “goal_pose”, and the black nodes represent the path generated. The magenta blocks represent the obstacles.

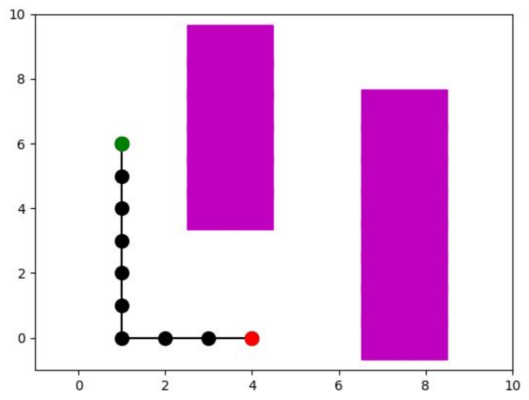
Random Planner:

`max_step_number` = 200 was chosen, with the memory queue limit consequently set as `sqrt(max_step_number)`. The two sets of query points capture the stochasticity in this planner fairly well. Example 1 shows that the planner had to visit nearly every valid node in the workspace (and revisit nodes when the memory bank allowed it to) to reach the goal. Example 2 shows a relatively “clean” path generation, where the planner did not have to visit a huge number of nodes. However, this is not replicated every time the planner runs, due to the random sampler -- a second run of the same configuration may result in a path which explores a far greater number of nodes. The analysis presented in the section goes into greater detail on how this affects performance.

Optimal Planner:





The configurations used in the random planner are reused here. A standard implementation of A* is used to generate the shortest non-colliding path. It is clear that the paths shown here are optimal. The admissible heuristic function used here is the manhattan distance, and analysis shows that it indeed is close to the optimal heuristic.

Analysis

First, we attempt to characterize the complexity of A*. A*'s complexity depends largely on its heuristic, and good heuristics are hard to find, especially since they also depend on the kind of environment they are in. However, time complexity of A* as a rule of thumb is generally in the order of its *branching factor b* - the average potential branches explored while getting to the optimal path, as function of the depth of the search *d*. That is, $O(b^d)$. However, a good heuristic can significantly reduce this by "pruning away" unnecessary branches and speeding up the process. Therefore, the complexity can be rewritten as $O((b^*)^d)$, where *b** is the *effective* branching factor caused by the heuristic function. In the absolute optimal case, *b** tends to 1. If N is the number of nodes explored by the search, then solving the following equation can get the value of the effective branching factor [1].

$$N + 1 = 1 + b^* + (b^*)^2 + (b^*)^3 + ... + (b^*)^d$$

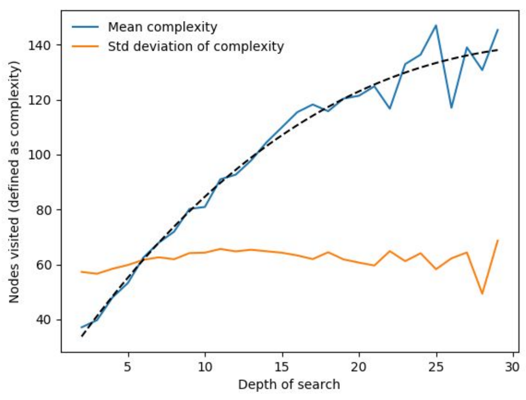
A test set of 100 start-end configurations were generated and using the following code snippet in analyze_planners.py, the mean effective branching factor was calculated. This function essentially tries to find the roots of the *d*-order polynomial shown above.

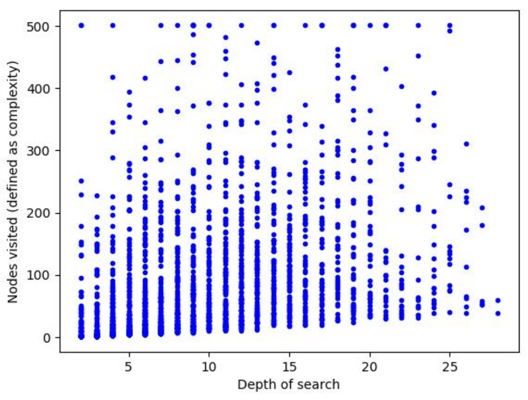
[1] Russell, Stuart; Norvig, Peter (2009) [1995]. *Artificial Intelligence: A Modern Approach* (3rd ed.). Prentice Hall. p. 103

```
def branching_factor(n, d):
    coeff = np.zeros((1, d+2))[0]
    coeff[0] = 1
    coeff[-2] = -n+1
    coeff[-1] = -n
    b_roots = np.roots(coeff)
    return [x for x in list(b_roots) if np.isreal(x) and x > 0][0]
```

The returned value of *b** was - **an average of 1.53 with standard deviation 0.22.**
Thus, this particular A* implementation on an average branches out 1.53 times per node explored, averaged over the entire workspace. This is close to the optimal branching factor of 1, thereby validating our selection of the heuristic function. As a matter of practice, multiple heuristic functions should be evaluated to figure out which one best fits the problem at hand.

Calculating a similar metric for the random planner is a little trickier. Because of its stochastic nature with a high degree of variance, the chances that it will generate the same path every time between two query points is very unlikely.





As mentioned in the beginning, coming up with an accurate estimate for complexity for a planning algorithm given just one fixed environment is spurious, and may lead to false conclusions. The plots shown above attempt to plot the complexity as a function of the depth of search, similar to A*. Here, the depth of search is actually the length of the optimal path generated by A* for the same configuration, while the complexity is defined as the nodes visited including repetition by the random planner.

A test set of 10,000 configurations was generated - this large number (compared to the relatively small grid size of 10 x 10) was chosen deliberately to ensure that there is plenty of repeated configurations, as the the random planner will generate different paths for the same configuration run multiple times. The right figure illustrates this, where each dot represents an instance of the random planner being run. The nodes visited were averaged across all instances for each depth of search value, and is plotted as function of the depth in the figure on the left.

The black-dotted line shows a polynomial fit carried out on the mean complexity. While it indicates the complexity has an almost asymptotic relationship with the depth of search, it really means that as the query points get placed in the extreme ends of the workspace, the random planner tends to explore pretty much the entire workspace every time and thus, the difference between the average number of nodes explored reduces with increasing depth of search. Therefore, evaluating the random planner on a boundless space may give a better indication of the complexity relationship.

It is pretty clear that A* beats the random planner in speed and path optimality handily. However, an area where the random planner actually beats A* is *space optimality.* That is, the memory consumed by storing all the nodes explored to get the optimal path. Since A* does not permit revisiting nodes, this creates an ever expanding set of nodes marked as “explored” - in large graphs, this may be a problem. On the other hand, the random planner designed here has a fixed memory, therefore capping it at a reasonable size would make it less memory intensive.