CDS 234a Final Project

Neil Janwani, Krishna Pochana, Shiva Sreeram

March 2023

1 Overview

For a path planning system, one must consider the myriad of trade-offs when selecting the algorithmic approach to navigating your environment. Your approach is also heavily dependent on your prior knowledge of the map and the ability to detect any unexpected discrepancies in the map. As such, in this report, we will explore these very trade-offs with the following algorithms: Dijkstra, A*, and D*. This exploration will be conducted in partially known environments where obstacles are discovered as the robot executes the planned path; consequently replanning needs to occur in order to navigate the robot to completely to the goal.

2 Approach

2.1 A*/Dijkstra Replanning

For this algorithm, we utilized a generalized A* implementation to accept a graph populated with nodes and their neighbors to plan from a start to goal node. The deck is populated by nodes sorted by the sum of the minimum cost to reach said node and the cost estimate to the goal node. Note that for Dijkstra, the cost to the goal node is multiplied by zero since Dijkstra's algorithm is not directed. The costs are computed by the Euclidean distance metric–from one node to another.

With this implementation, we determine the path from start to the goal node under the assumption that there are no obstacle nodes present in the system. Then, we introduce at least one obstacle (represented as a node switched from a traversable node to an obstacle) right in front of the robot's traversal along said path. The graph is reconfigured with this obstacle, causing this obstacle node to lose its neighbors, and then the A* planner is run again with the new start position at the location the robot was at before it detected an obstacle. Throughout this process, we track a variety of statistics regarding the efficiency of the algorithm which we will report in the Results and Analysis section.

2.2 D*

2.2.1 D* Concept Overview

To implement D*, we follow the original publication by Anthony Stentz (1994). In this method, a priority queue is used to store the list of nodes to potentially expand to their connected neighbors in trying to reach the start node from a rooted goal node. To sort the queue, we use *key values* $k(x)$, defined by the minimum path cost from the rooted goal node to a given graph node over all modifications made since the node was added to the queue.

The robot starts off with an initial awareness of its environment, with an associated graph and traversal cost estimates between nodes of this graph; in our case, the robot knows no obstacles in the graph. From the start position, the robot executes an initial planning phase to generate a path to the goal. In this phase, there are no modifications to the path cost estimates once they are added to the priority queue, and thus $h(x) = k(x)$, where $h(x)$ is the *path cost function* (an estimate of the arc path costs from x to goal), for all states x in the graph. Thus, in this phase the path planner is equivalent to Dijkstra's algorithm.

When the robot observes an obstacle or other environment change that alters the traversal cost of a graph segment, this traversal cost is updated and the graph nodes that are affected are re-inserted into the priority queue. Within the priority queue, when a node is popped to be processed, we can check its descendant connections. If the stored path cost no longer reflects the actual path cost (due to the updated traversal costs), we can update the stored path cost $h(x)$. This yields $k(x) < h(x)$ and defines a `RAISE` state, in which a node propagates a cost *increase* throughout the graph nodes. This `RAISE` state is then added onto the queue, as are all neighbors for similar examination.

Similarly, if the stored path cost does equal the actual path cost, then $k(x) = h(x)$, which defines a `LOWER` state. Here, the planner tries to find a new optimal path from the rooted goal node to the robot's current position.

These `RAISE` and `LOWER` nodes continue to propagate from the obstacle node and from the 'wavefront' of `RAISE` nodes, respectively. We could allow these wavefronts to propagate throughout the entire graph, however, once our priority queue processes a node with k_{min} such that $h(y) \leq k_{min}$, we know we have done 'enough work' to generate our optimal path using backpointers (relating nodes to parents). The reason for this is because the rest of the priority queue only relates to nodes that are further away–have a higher cost path–than our current robot position to our goal.

We can repeat this cycle of modifying costs and updating our graph by use of the functions `modify_cost` and `process_state` respectively. Overall, the combined use of these functions allows for a highly optimized replanning algorithm, especially when compared to Dijkstra's algorithm detailed earlier.

2.2.2 D* Comparison to Dijkstra and A*

To make a comparison between D* and the Dijsktra and A* planners, we initially plan on an environment with no obstacles. Then, a phase of obstacles is introduced, as described in the next section. The introduction of obstacles forces a replan for the Dijkstra and A* planners, while `modify_cost` is used for D* to re-plan by modifying the existing search tree.

2.3 Environment and Obstacles

The first environment these algorithms must navigate is a randomly seeded set of 500 nodes as well as the start and goal nodes with the domain and range being $x \in [-5, 5]$ and $y \in [-5, 5]$. We then connect nodes to all neighbors that are within a Euclidean distance of 1. During this process, we will induce a total of four replans due to obstacles introduced. This environment setup can be seen as follows:

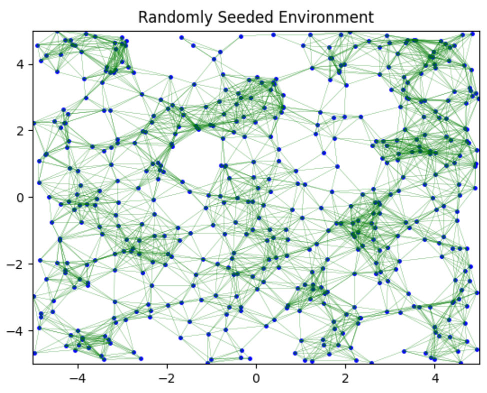

Figure 1: Nodes are shown in blue, connections in green.

The next type of environment is a rectangle. This rectangle is made in the same axes and involves connections to the adjacent points in the rectangle. Here, the start node is $(-4, -3)$ which is the bottom left and the goal node is $(4, 3)$ which is the top right. In this setup, we will introduce an obstacle, force a replan, introduce another obstacle while removing the previous one, and force another replan. The environment setup can be seen as follows:

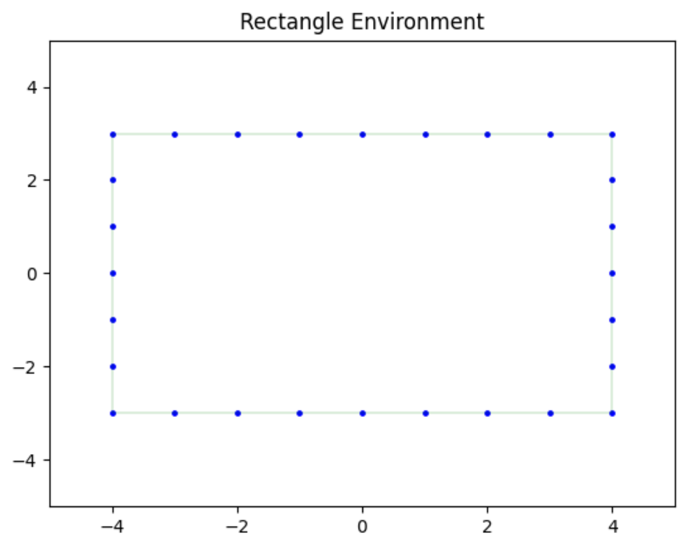

Figure 2: The rectangle environment the robot will navigate through. Nodes are shown in blue, connections in green.

Finally, we have the contour environment, in which we model the cost function as a 3D contour. This, we thought, might be representative of a more realistic robot environment, one with hills and/or different terrains etc. Again, we maintain that the start node is at $(-4, -4)$ as in the other environments, as well as the goal being at $(4, 4)$. However, we of course do not have a straight-line path as we did in the randomly seeded environment. Furthermore, the contour polynomial is passed into an exponential function to bound it below from 0 (such that we do not have negative costs) while maintaining the order of costs in our priority queue.

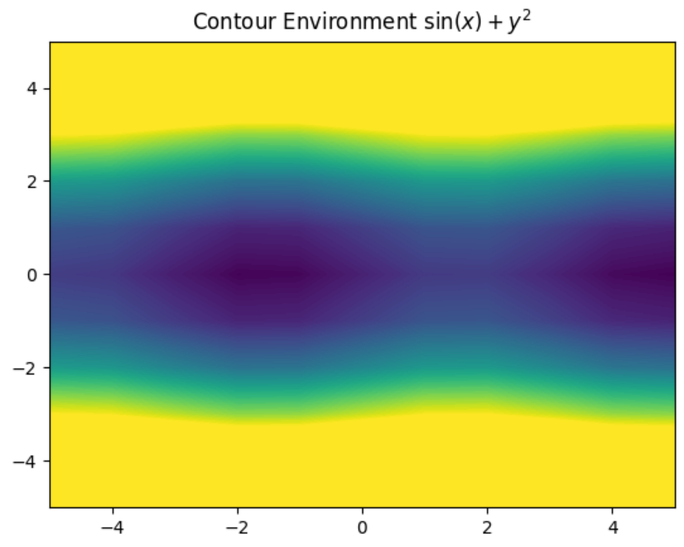

Figure 3: Sample contour environment with polynomial $z = sin(x) + y^2$ representative of the cost function.

2.4 Metrics for Analysis

To compare D* to Dijkstra's algorithm and A*, we track three main metrics: the number of nodes added to the `ONDECK` priority queues, the number of nodes that have been processed, and the number of iterations made by each algorithm.

The number of nodes added to the queue for each algorithm is representative of the number of times a node is added to the queue for processing. This means the D* algorithm, for example, counts every time a node is re-inserted onto the on-deck for operations such as delaying cost-propagations until optimality is reached.

The number of nodes processed is representative of the number of nodes popped from the queues during path solving. This is representative of the number of nodes popped after meaningful computation. For example, due to using Python's built-in priority queue for our D* implementation, removing non-surface queue elements was not possible, and thus duplicate queue insertions were made. Thus, there were cases where a node would be marked `DONE` only to re-appear on the queue due to redundancy. Such nodes were thus not counted.

Lastly, the number of iterations is representative of the number of node expansions performed by each algorithm, as this is the main action done by the algorithms in each loop pass.

3 Results and Analysis

3.1 Random Seeded Environment

We begin with our random seeded environment. The planners initially assume no obstacles, then obstacles are introduced at nodes 25%, 50%, and 75% along the way of the latest path, defined $O1, O2, O3$, respectively. One node before the robot reaches $O1$, it detects all three obstacles and begins a replan. For Dijkstra's algorithm and A*, this is a complete replan as defined in section 2.1, while for D* this is a graph modification as described in section 2.2.1. We see the results of our algorithms below:

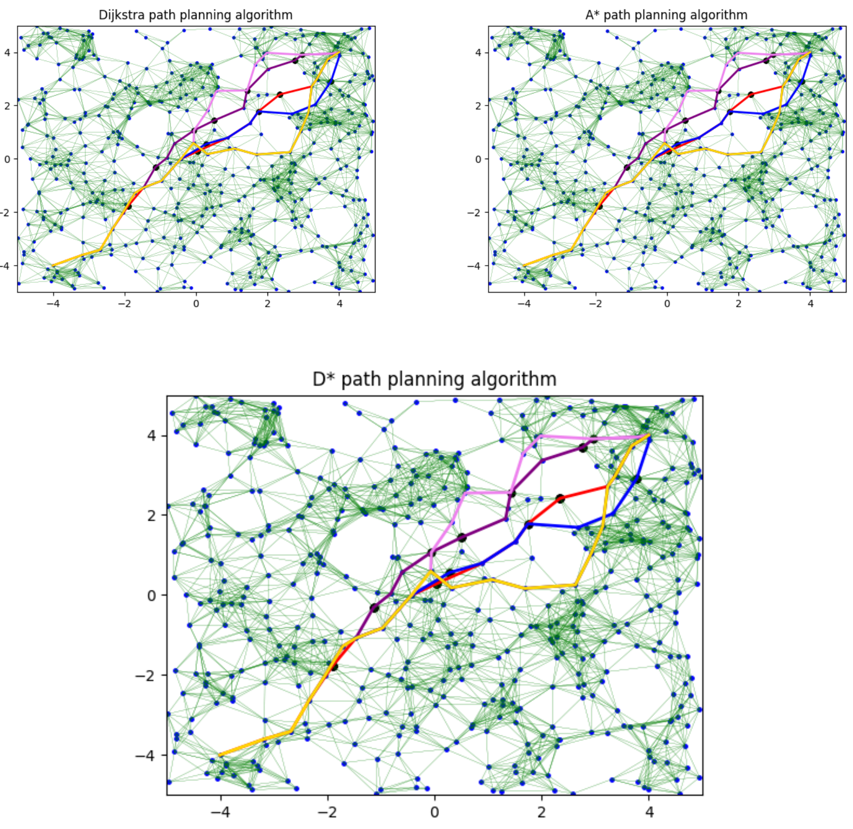

Figure 4: Each algorithm initially generates the path in red from start to goal, it then encounters the first set of obstacles (represented as black dots) and replans to form the path in purple. This process of replanning repeats to form the blue, violet, and indigo paths. We then show the final path that would form in gold.

Note that all of these paths are the same and in fact all have the same final cost of 13.4. This is due to the fact that all of these algorithms have a consistent cost heuristic and guarantee optimality. The differences will arise when we instead consider the computational cost of these algorithms.

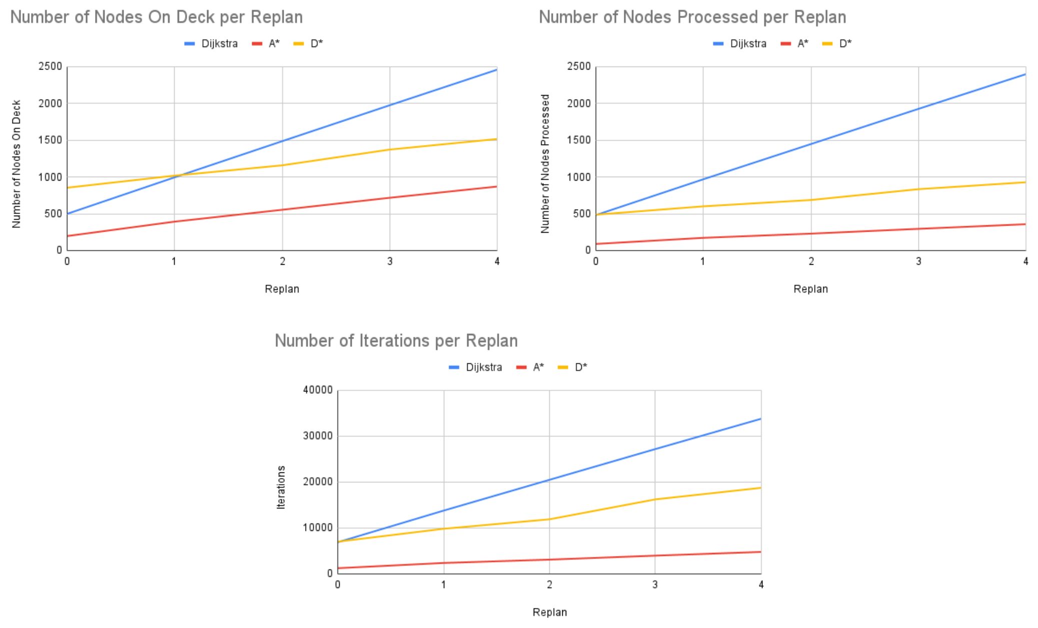

Figure 5: Tracking the number of nodes placed on the deck, the number of nodes that have been processed, and the number of iterations (runs of a loop) for each algorithm as replans occurred due to new obstacles.

Observing the results from these plots, we see the number of nodes processed as well as iterations before any replanning for D* and Dijkstra are quite similar. This is to be expected as the cost heuristic is the same for both and the modification of cost doesn't arise until later. We do see that for the nodes put on deck, D* is higher, and this is due to the `LOWER` state being run and capturing more neighbors to put on the queue. Eventually, with the replans, D* is significantly more efficient than Dijkstra thanks to the information it saves. Regardless, we see that A* is better in these categories, despite not saving information. The directed nature of this algorithm works very well in this environment as no pockets or bug traps exist. As such, we see that A* would be the ideal implementation here, although combining directed planning with the advantages of D* would yield the best algorithm, such as with focused D* or D* lite.

3.2 Rectangle Environment

Next, we have the rectangle environment. As discussed, we will introduce an obstacle, force a replan, introduce another obstacle while removing the previous one, and force another replan. Let us explore this environment with both A* and D*.

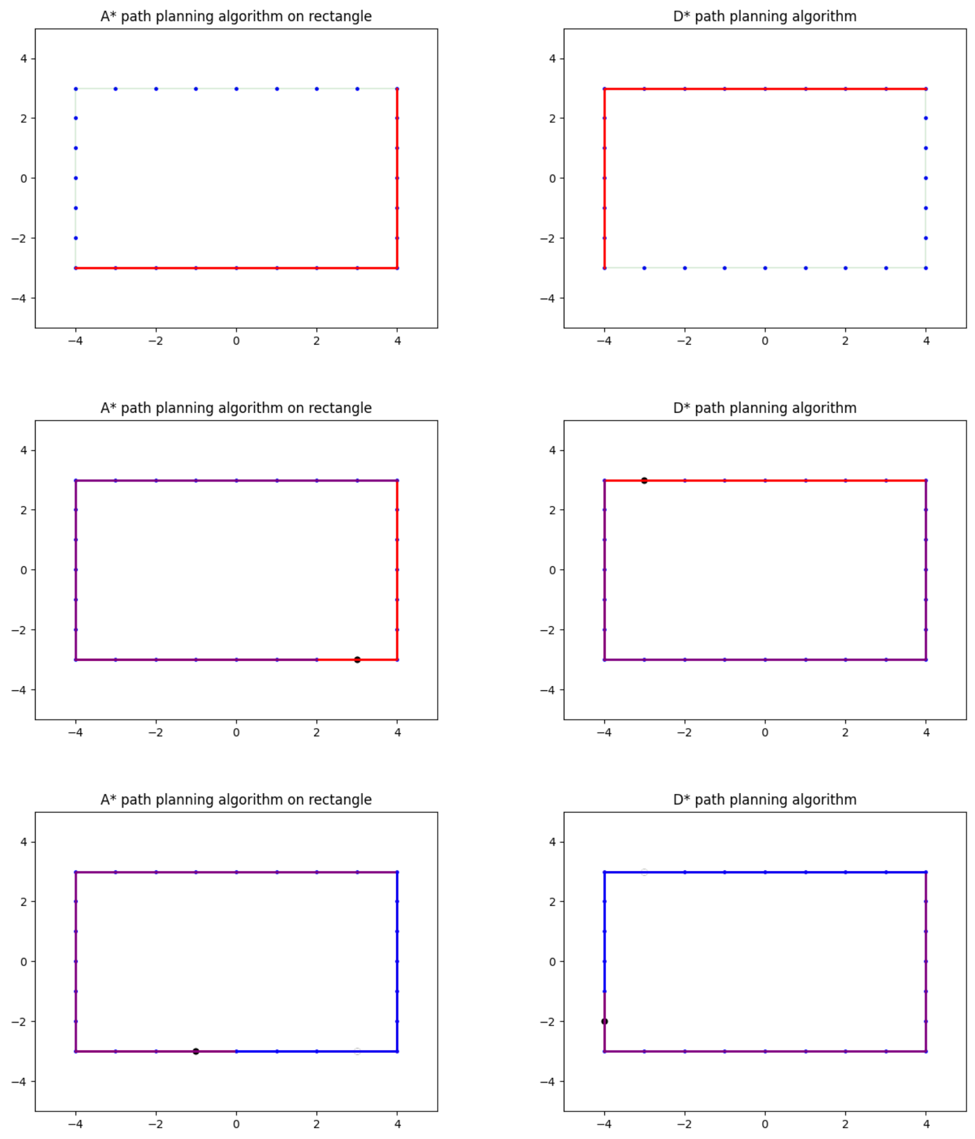

Figure 6: Traversing around the rectangle with an obstacle being introduced along the path. The first row defines the initial plan before the robot knows of any obstacles with the path in red. Then, it encounters an obstacle (shown as the black dot) and replans a path from there shown in purple. However, it encounters a new obstacle while the previous one is freed up. As such, it replans once more from here, reaching the goal in the blue path.

For both of these, the final cost of the path is 18. An interesting point of note, unlike the Random Seed Environment, D* performs better than A* in the metrics we discussed.

	A* Nodes On Deck	D* Nodes On Deck
Initial	27	27
Replan 1	49	41
Replan 2	61	48

	A* Nodes Processed	D* Nodes Processed
Initial	27	28
Replan 1	48	41
Replan 2	59	46

	A* Iterations	D* Iterations
Initial	54	54
Replan 1	96	94
Replan 2	118	104

We see the initial values are about the same as the direction from A*'s cost to goal heuristic isn't particularly helpful since both directions to move from the start have the same result. However, the saved information in D* was helpful in reducing computation. As such, in this environment, D* is the ideal implementation of the two.

3.3 Contour environment

One of the benefits of D* is the ability to alter the cost function continuously as the robot traverses the graph. For instance, imagine the robot is attempting to traverse a windy environment in which large amounts of sand cover the rocky surface. Assume the robot's wheels perform better on a rocky surface, and thus it defines such surfaces as safer–having lower cost. Unfortunately, due to the wind, the sand frequently changes positions. The resulting partially known (at least initially) graph can be searched efficiently with D*. We show an example below:

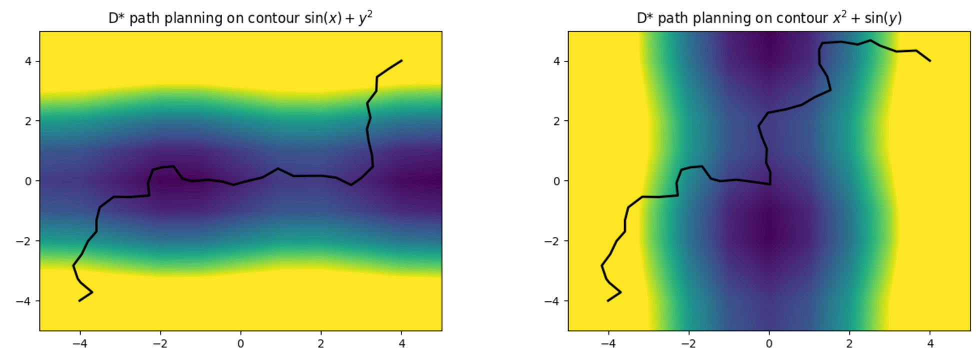

Figure 7: On the left is the initially discovered environment. Midway through the path, the robot determines that the environment has changed. We model the 2D cost function using a contour plot.

We now plan with Dijkstra's algorithm in the same environment, with the same resultant path being achieved once the contour change is enacted and replanning is executed. Furthermore, since the *entire* contour cost surface changed, D*'s previous knowledge of the graph is essentially overloaded by its wavefronts. Thus, Dijkstra and D* perform identically in terms of computational speed, nodes processed, and nodes ondeck. This is because path-optimality is maintained in both the D* and Dijkstra implementations, and thus paths of the same cost, and in thus the same segments, are achieved.

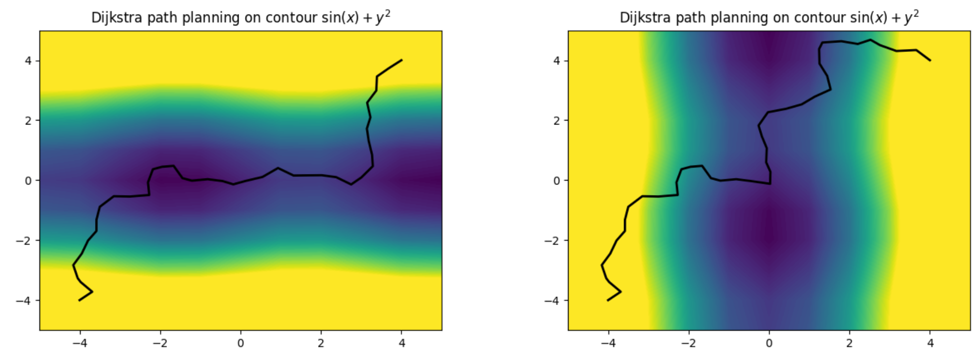

Figure 8: We repeat the same procedure with Dijkstra's algorithm. On the left is the initially discovered environment. Midway through the path, the robot determines that the environment has changed. We model the 2D cost function using a contour plot.

4 Conclusion

In this project, we have shown implementations of Dijsktra's algorithm, A*, and D* that can be used for robot path planning in dynamic environments. Through simulation in environments representative of traveling through well-connected space, constricted space, and a continuous-changing-cost space, we have demonstrated how D* can be used to navigate in a wide breadth of terrains that might be found in the real world. Lastly, we have made comparisons between the three planning algorithms in these environments, showing that in environments where there are changes relatively close to the robot, D* enables a large reduction in computational time and memory requirement, with the difference increasing as the number of replans increases. However, large propagations of environmental changes negates this advantage, yielding performance similar to other optimal planners such as Dijkstra's algorithm.

This project lays down a code base that can be used for future comparisons including more detailed testing of computation and memory requirements versus the number of replans using a larger variety of environments with more interesting obstacles to reach edge cases. In addition, more variants of the D* family could be implemented, such as focused D* or D* lite to make a more meaningful comparison with A*. Lastly, the code base could be expanded to parse sensor inputs and potentially be used in ROS simulation or a physical robot, such as Flippert Jr.

Figure 9: Flippert (Jr.)

5 Appendix

We include our (public) Github repository here. We also include our planner code below:

```python
import numpy as np
import bisect
from queue import PriorityQueue
from lib.node import *

class Planner:

    def __init__(self, init_onDeck = 0, init_processed = 0, init_iterations=0):
        self.nodes_onDeck = init_onDeck
        self.nodes_processed = init_processed
        self.iterations = init_iterations

    def plan():
        raise NotImplemented

    def getCounts(self):
        return (self.nodes_onDeck, self.nodes_processed, self.iterations)

    def addOnDeck(self):
        self.nodes_onDeck += 1

    def removeOnDeck(self):
        self.nodes_onDeck -= 1

    def addProcessed(self):
        self.nodes_processed += 1

    def addIterations(self):
        self.iterations += 1


class Astar(Planner):

    def __init__(self, cost_multiplier=1, init_onDeck = 0, init_processed = 0, init_iterations = 0):
        super().__init__(init_onDeck, init_processed, init_iterations)

        self.path = []
        self.cost_multiplier = cost_multiplier

    def plan(self, graph, start, goal):
        onDeck = []

        start.state = Node.ONDECK
        start.creach = 0.0
        start.cost = self.cost_multiplier * start.cost_to(goal)
        start.parent = None
        bisect.insort(onDeck, start)

        while True:
            # Grab the next node (first on the sorted on-deck list)
            node = onDeck.pop(0)

            # Check the neighbors
            for neighbor in graph[node]:
                self.addIterations()
                # Skip if already processed
                if neighbor.state == Node.DONE:
                    continue

                # Pre-compute the new cost to reach the neighbor
                cdelta = node.cost_to(neighbor)
                creach = node.creach + cdelta

                # If already on deck: skip if lower/same cost, else remove
                if neighbor.state == Node.ONDECK:
                    if neighbor.creach <= creach:
                        continue
                    onDeck.remove(neighbor)
                    self.removeOnDeck()

                # Add to the on-deck queue (in the right order)
                neighbor.state = Node.ONDECK
                self.addOnDeck()
                neighbor.creach = creach
                neighbor.cost = creach + self.cost_multiplier * neighbor.cost_to(goal)
                neighbor.parent = node
                bisect.insort(onDeck, neighbor)

            # Mark this node as processed
            node.state = Node.DONE

            # Check whether the goal is thereby done
            if goal.state == Node.DONE:
                break

            # Make sure we have something pending on the on-deck queue
            if not (len(onDeck) > 0):
                print("FAILED")
                return None, None

        # Create the path to the goal (backwards)
        node = goal
        path = []
        while node is not None:
            path.insert(0, node)

            # Move to the parent
            node = node.parent

        for node in graph.keys():
            if node.state == Node.DONE:
                self.addProcessed()
        # total_cost = 0
        # for i in range(len(path) - 1):
        #     total_cost += path[i].cost_to(path[i+1])
        total_cost = goal.creach

        return path, total_cost


class Dstar(Planner):
    NEW = 0
    OPEN = 1
    CLOSED = 2

    def __init__(self, g, graph, init_onDeck = 0, init_processed = 0,
                 init_iterations = 0):
        """
        Initialization function (many details are left out and can be seen in
        the D* paper) that declares variables and initializes dictionaries.

        g       is the goal
        graph   is the environment graph dictionary
        """
        super().__init__(init_onDeck, init_processed, init_iterations)

        self.t = {}                         # the types for a state X (i.e. NEW,
                                            # OPEN, or CLOSED)

        self.h = {}                         # the current ESTIMATES (i.e. robot
                                            # may find it is wrong) of the arc
                                            # costs from goal to X

        self.k = {}                         # decides whether we are a RAISE,
                                            # LOWER, or Dijkstra "mode". these
                                            # are the priorities used to order
                                            # the pqueue. This is the
                                            # simplification made by Gunters
                                            # approach
        self.b = {}                         # backpointers dictionary to later
                                            # reconstruct the path. Very
                                            # important for D*...

        self.open_list = PriorityQueue()    # OPEN list of states

        self.path = []                      # will eventually hold the path

        self.graph = graph                  # graph with neighbor relations

        self.total_cost = 0

        # initialization steps
        for x in graph.keys():
            self.t[x] = Dstar.NEW

        self.goal = g
        self.h[g] = 0
        self.open_list.put((self.h[g], self.goal))
        self.t[g] = Dstar.OPEN
        self.b[self.goal] = None
        self.fullpath = []

    def min_state(self):
        """
        Returns the top of the OPEN priority queue
        """
        return self.open_list.get(block=False)


    def get_kmin(self):
        """
        Peeks at the top of the OPEN priority queue
        """
        kmin, temp = self.open_list.get(block=False)
        self.open_list.put((kmin, temp))
        return kmin


    def insert(self, x, h_new):
        """
        Inserts states into the open list based on their RAISE, LOWER, or
        'Dijkstra' state
        """
        if self.t[x] == Dstar.NEW:
            self.k[x] = h_new
        elif self.t[x] == Dstar.OPEN:
            self.k[x] = min(self.k[x], h_new)
        elif self.t[x] == Dstar.CLOSED:
            self.k[x] = min(self.h[x], h_new)

        self.h[x] = h_new
        self.t[x] = Dstar.OPEN
        self.open_list.put((self.k[x], x))
        self.addOnDeck()


    def process_state(self, start):
        """
        Compute one iteration of D*

        start is the start state (note the goal state is fixed)
        """
        k_old, x = self.min_state()
        # print(k_old, x)
        if x == None:
            Exception('Priority queue is empty')
        # print(x.x, x.y)

        if self.t[x] == Dstar.CLOSED:
            return 0
        self.t[x] = Dstar.CLOSED
        self.addProcessed()
        if start != None and self.t[start] == Dstar.CLOSED:
            return 0

        # RAISE state
        if k_old < self.h[x]:
            for y in self.graph[x]:
                self.addIterations()
                if self.h[y] <= k_old and self.h[x] > self.h[y] + x.cost_to(y):
                    self.b[x] = y
                    self.h[x] = self.h[y] + x.cost_to(y)

        # LOWER state
        if k_old == self.h[x]:
            for y in self.graph[x]:
                self.addIterations()
                if self.t[y] == Dstar.NEW \
                or (self.b[y] == x and self.h[y] != self.h[x] + x.cost_to(y)) \
                or (self.b[y] != x and self.h[y] > self.h[x] + x.cost_to(y)):
                    self.b[y] = x
                    self.insert(y, self.h[x] + x.cost_to(y))

        # "DIJKSTRA" state (effectively a dijkstra implementation)
        else:
            for y in self.graph[x]:
                self.addIterations()
                if self.t[y] == Dstar.NEW \
                or (self.b[y] == x and self.h[y] != self.h[x] + x.cost_to(y)):
                    self.b[y] = x
                    self.insert(y, self.h[x] + x.cost_to(y))
                else:
                    if self.b[y] != x and self.h[y] > self.h[x] + x.cost_to(y):
                        self.insert(x, self.h[x])
                    else:
                        if self.b[y] != x \
                        and self.h[x] > self.h[y] + y.cost_to(x) \
                        and self.t[y] == Dstar.CLOSED \
                        and self.h[y] > k_old:
                            self.insert(y, self.h[y])

        return self.get_kmin()


    def modify_cost(self, x, y, cval):
        if cval == float('inf'): x.make_obstacle()
        elif cval == 0: x.make_free()
        else:
            x.set_cost(cval)
        # assert(x.cost_to(y) == cval)
        if self.t[x] == Dstar.CLOSED:
            self.insert(x, self.h[x])
        return None #self.get_kmin()

    def plan(self, start, y=None):
        curr_k = 0
        def get_condition(curr_k):
            if y == None:
                return self.t[start] != Dstar.CLOSED
            else:
                return curr_k < self.h[y]

        init = None
        if y == None:
            init = start


        while get_condition(curr_k) and curr_k > -1:
            curr_k = self.process_state(init)

        self.path = []
        if y == None:
            node = start
            self.total_cost = 0
```

```
else:
    node = y

while True:
    self.path.append(node)
    if node == self.goal:
        break
    node = self.b[node]

if y == None:
    self.fullpath = self.path
else:
    self.fullpath = self.fullpath[:self.fullpath.index(y)] + self.path

print('COST:', np.sum([self.fullpath[i].cost_to(self.fullpath[i + 1]) for i in range(len(self.fullpath) - 1)]))

return self.path
```