Exam 2
Dr. Fields
Christopher Seagraves
16th November 2022

The code used for the following problems is found in these two github repos.

https://github.com/nosv1/seagraves_unmanned_systems/tree/Exam2/SearchAlgorithms
This is the SearchAlgorithms module. SearchAlgorithms/main.py is currently a mess and is setup for the TSP stuff; it loads a Scenario (obstacles, gird, and such are defined in this), sets up the plot, defines the cost matrix, and then loops the permutations.

https://github.com/nosv1/seagraves_unmanned_systems_pkg
This is the ros2 package. The files of interest are seagraves_unmanned_systems_pkg/TagYoureIt/pursuer.py (self.on_lidar_callback() decides which object to chase), as well as support_module/TurtleNode.py (self._lidar_callback() calls support_module/DetectedObject.detect_objects() – where the magic happens)

1.

a. **Explain in what scenarios would RRT be the best choice (of the RRT, A*, Dijkstra's options)**

   I'm having a hard time coming up with a case when RRT is ever the best, but one idea is when the space is large and simple, there are multiple goals needed to be found, and the path doesn't need to be optimal. The key being a simple space, allowing RRT to take large steps compared to a dense defined grid. In any case, the slow part of an algorithm is checking to see if new points are not valid, so the less an algorithm needs to do that, the better – steps needed per goal is the metric I'm thinking of.

b. **If you have a single starting point and 20 stops on your delivery run (traveling delivery man). How many possible paths are there that visit each of the stops? Make sure to show your work**

   20! + 20
   There's *20!* ways to combine the 20 delivery points, but there's still 20 more ways to connect the starting point to a given delivery point.

c. **Describe how the genetic algorithm operates, and how it is applicable to the traveling salesman problem.**

   GA works by strategically, randomly changing attributes of many solutions for a given problem over many iterations.
   i. Determine which attributes make up a solution (which genes make up a chromosome)
   ii. Generate a population of chromosomes
   iii. Evaluate each chromosome in the population based on a fitness function
   iv. Select which chromosomes to keep
   v. Mutate those chromosomes
   vi. Loop until population converges

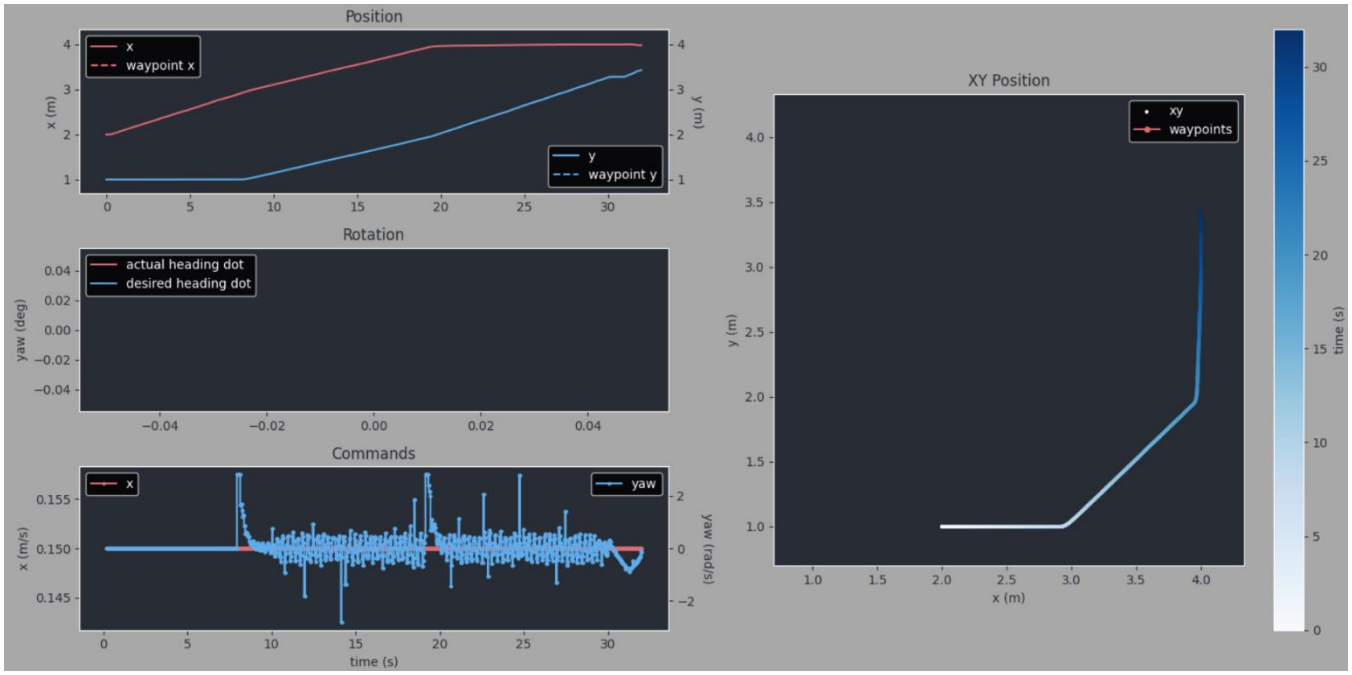
   GA is applicable to TSP as a chromosome is a specific order of genes (delivery points). We can determine the total cost of each connection between two delivery points, choose decent orders, mutate, or slightly change, the order of the genes in each chromosome, then do it all again until our computer starts smoking.

2. **You must provide a plot of each bots' path along with the desired trajectory generated for the evader, the code used for the problem, and provide any comments/discussions necessary to interpret your results and describe any modifications you had to make in order to be successful.**

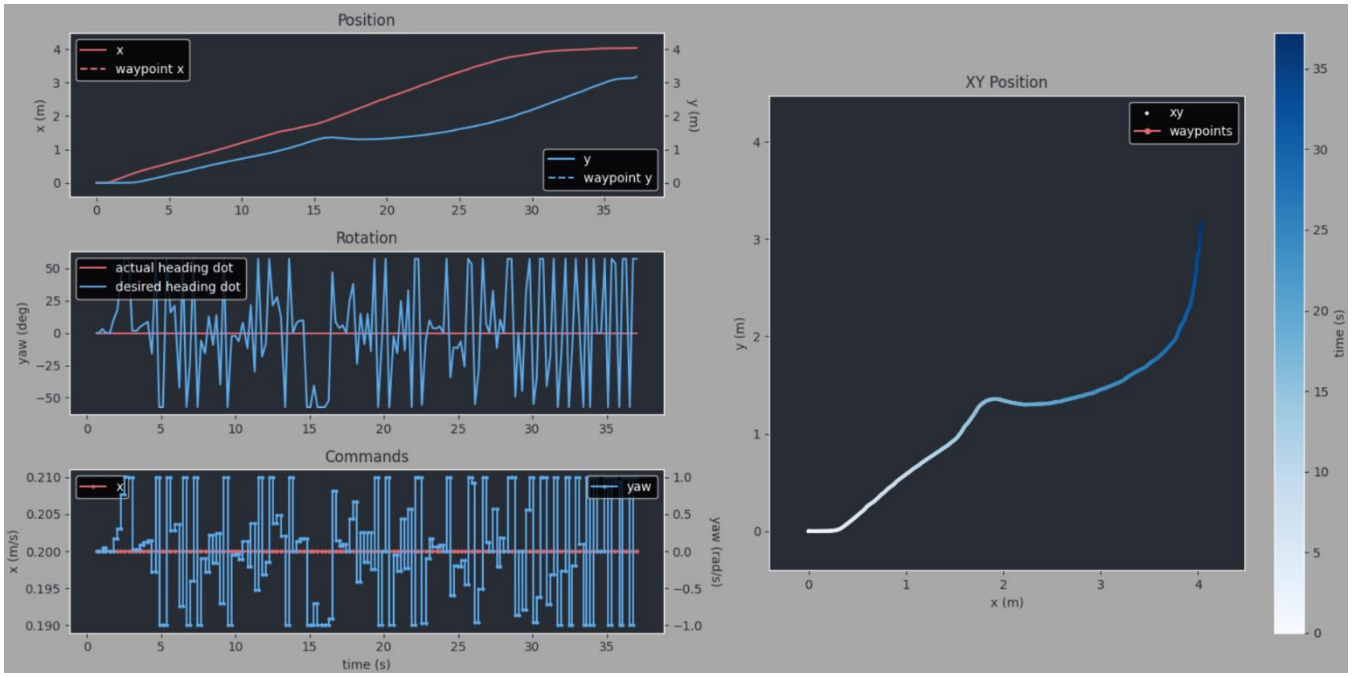
First successful run (the plots are this run)
https://1drv.ms/v/s!Aivmw7zUVvnRqoV-TFnOlMULec4NsA?e=77AouD

Additional example
https://1drv.ms/v/s!Aivmw7zUVvnRqoYCnEWZvAPQHcHqCw?e=0Gsyyq

Evader



Pursuer



I think the heading plot (2nd down on the left) is backwards, red line is the desired heading dot, and the blue line is the actual heading dot.

Given the time limitations and the surprise of having to come up with a way to detect > 1 object for the first time, I just needed something that appeared to work; in that sense some assumptions are being made during the pursuit. The main assumption is the pursuer should chase the closest object it sees; even this isn't trivial, but it is quite a limitation.

To be able to choose an object to chase, you must be able to detect objects, as in know their dimensions (width in this case). To detect objects, you can loop all points detected in a lidar scan and then based on difference of distance between two non-infinite distances, you can assign two significant points per object (left most and right most point). Then you can get the furthest point every object, and choose the smallest of those – object to chase = min(max(significant point distances per object)).

While this works when the lidar sees two massive surfaces and one small object, it can fail going around the first wall if the pursuer is closer to the short surface of the wall than to the evader; for this reason, in the second link, you'll see I don't start the evader until the pursuer is quite close, just to make sure it doesn't think the short surface is the evader.

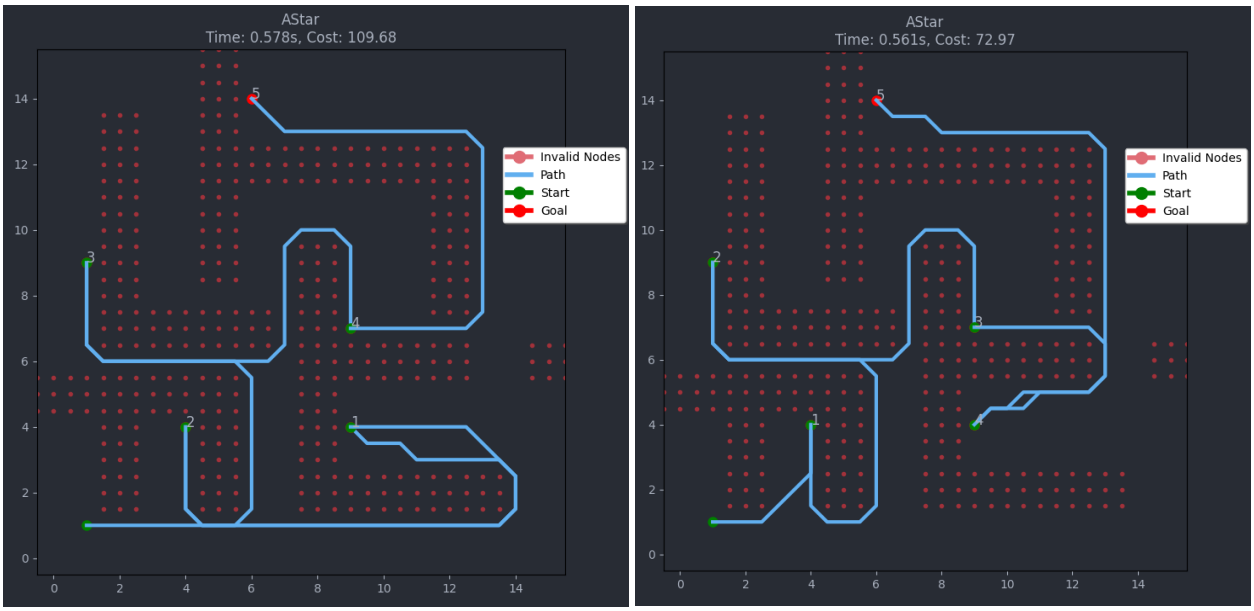
3. **Provide a plot showing the obstacles, A* paths, and delivery locations. Provide the total travel distance of your solution. How does this compare with the other possible paths (how good is the solution, are there other paths that have much higher travel costs)? Turn in a copy of your working code.**

Having a hard time understanding what's being asked in this question. I'm guessing you're looking for a brute force of the 5 delivery points, then comparing that to some order other than the optimal one?

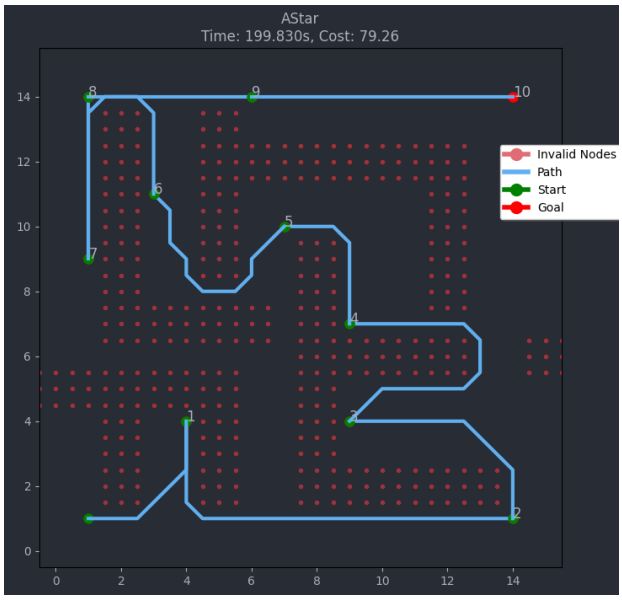
On the left is the A* solution if you traverse the delivery points in the order given; clearly this is suboptimal given the backtracking involved – specifically from delivery point 1 to 2.

On the right, the optimal path after brute force

Cost difference: (109.68 – 72.97) = 36.71 ~ 33% decrease



4. **Plot identical to Problem 3. Provide the computation time, and the minimal total travel cost.**
   Time: 199.830 seconds (> 10 min if printing per iteration)
   Cost: 79.26



Problem 5 on next page.

5. **Provide the computation time, and best path cost. Provide a plot of the obstacles with path (identical to previous problems).**

   https://github.com/nosv1/seagraves_unmanned_systems/blob/Exam2/SearchAlgorithms/modified_TSP_GenAlg_v2.py

   Didn't modify much, just needed to convert my inputs to the current code (needed to read my Scenario and set its attributes to the existing inputs). I also needed to import my A* code and adjust my cost matrix keys to the current one.

   Computation Time: 1057.27 seconds
   Travel Cost: 76.68

```
    population.mutate = mutate


def genetic_algorithm(
    cities,
    adjacency_mat,
    n_population=500,
    n_iter=2000,
    selectivity=0.75,
    p_cross=0.5,
    p_mut=0.3,
    print_interval=100,
    return_history=False,
    verbose=False,
):
    pop = init_population(cities
    best = pop.best
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMIN

Generation 1992: 76.6776695296637
Generation 1993: 76.6776695296637
Generation 1994: 76.6776695296637
Generation 1995: 76.6776695296637
Generation 1996: 76.6776695296637
Generation 1997: 76.6776695296637
Generation 1998: 76.6776695296637
Generation 1999: 76.6776695296637
Time: 1057.2702345
```