Colab: https://colab.research.google.com/drive/1SnnJelgKGs811hTUe3Idchw9r0CvmU05?usp=sharing

GitHub:
https://github.com/johnsonlarryl/csce_5210_pre_release/tree/project_3/robot_navigation

Design Approach:
The software implementation I used was the same as outlined in my design. The lower-level implementation used an Object-Oriented Design (OOD) and Object-Oriented Programming (OOP) approach from a design pattern perspective. This resulted in a cohesive software architecture where classes and methods performed very specific functions where objects collaborated with one another for the maze configuration, maze generator, policy configuration, and maze policy. This provided a clean separation between objects and clearly defined contracts or interfaces between objects.

This project utilizes a model-free, utility-based agent. The agent does not have a model like some of the traditional search informed heuristics algorithms. But rather it will be implemented using a Machine Learning (ML) sub-problem of the Reinforcement Learning (RL) archetype. In particular, the agent will implement a Markov Decision Processes (MDP) known as value iteration thereby implementing a Value Iteration Policy (VIP) algorithm. The VIP for this algorithm is a fully observable stochastic process. MDP, in general, is a decision problem where a series of sequential decisions utilize a Markovian transition model and additive rewards system to reach some goal. As a factored representation, MDP consists of "a set of states (with an initial state s0); and a is ∈ A(s) or set of actions in each state; a probabilistic transition model and a reward function.

In order to maze the policy and maze configurations fed into maze generator to create an m x n multi-dimensional array where m is the number of rows and n is the number of columns. Then utilizing a dynamic programming (DP) the VIP algorithm will converge to an optimum navigation policy through the maze.

T2.

The number of paths from the starting position is 20 not counting the destination and obstacle. The shortest path is the number of paths that have the shortest path to the destination. In this case the shortest path is grid position (0, 2) that moves to the destination at (0, 3). It has a total reward of 11.99. Also, 1 out of 20 paths or approximately 5% of the paths have the shortest path to the destination.

T3:

1. R1=50, R2=-50 and r=-5 => P2. Produce the new policy P2 and compare it with the policy you generated in T1 above. For the comparison make use of the two criteria mentioned in T2 above.

   The shortest path is grid position (3, 0) that moves to the destination at (0, 3).

   position (3, 0), follows the sequence of positions [(3, 0), (2, 0), (1, 0), (1, 1), (1, 2), (0, 2), (0, 3)] for a total reward 136.83.

2. R1=100, R2=-500, r=-5 => P3. Compare it with the two policies you generated in T1 and T3 part 1 above.

The path is still the same as the previous two policies. However, the cumulative reward is now 478.86.

3. R1=100, R2=-50 and r=-1 => P4.a

   The path is still the same as the previous three policies. But the total cumulative reward for this path is approximately 457.86.

   R1=100, R2=-50 and r=-5 => P4.b

   The path is still the same as the previous three policies. But the total cumulative reward for this path is approximately 372.04.

   The living reward of -1 seems to have the most beneficial impact to the policy as it maximizes the cumulative reward.

T4. Only applicable to those enrolled in 5210.

We will need to implement an additional probability for a collision object (ie. other dynamically moving objects). In addition, we will need to include the value of this cost multiplied against this probability. Then lastly, we will need to subtract this product from the iterative reward of the other possible probabilities.

```
function optimize_policy(s: State, a: A(s), t: P(s_prime |, s, a), r: R(s, a,
s_prime), C:C(s, a) d: ɣ, r:int, l: int, c: int) -> U:
  U = 0 , U_prime = 0, vectors of utilities for states in S
  ε  = 0.01

  U <- U_prime,

  while i  < 500:
      for s in S:
        u_prime[s] = l + (bellman_equation - C*c) # D and H as the positive
and negative reward respectively will be factored in based on the states (ie.
maze passed in).  Note l is the living reward.
C = Collision Probability
C = cost of collision.

        if |u_prime[s] - U[s]| < ε :
          return U

  return U
```