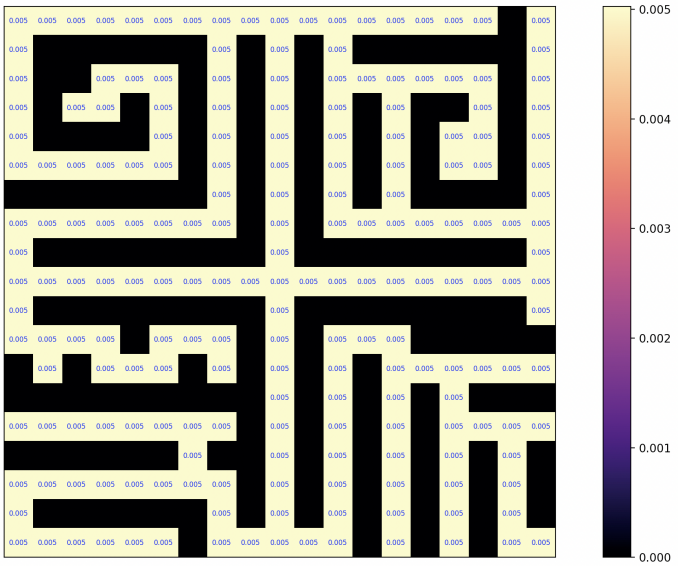
CS 520 Final Exam Q1

Tej P Shah, NetID: tps75

Question 1.1: A Priori Probabilities (5 Points)

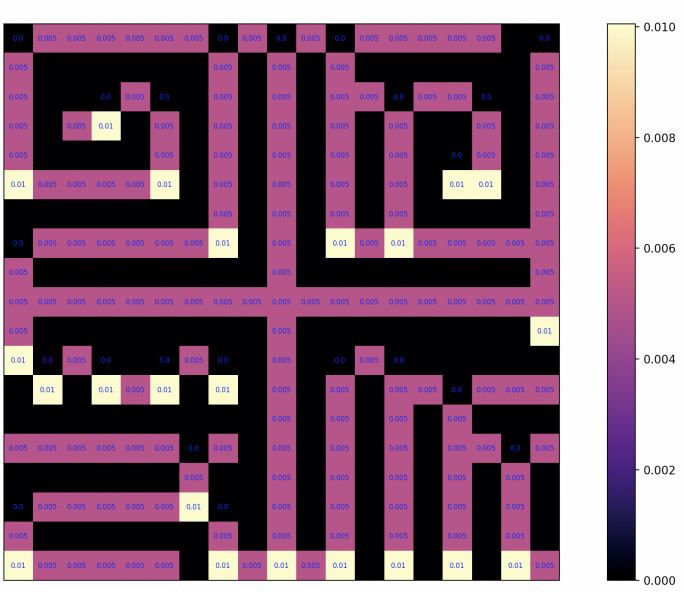


The submarine drone can move between any white cells, represented as _ in the text file but not to any black cells, defined as X in the text file. From this, we conclude 2 things: (1) it is not possible that the submarine drone exists in any of the black cells; (2) it is equally likely that the submarine drone exists in any of the white cells. Suppose we represent each cell `nuclear_reactor[i][j]` as 0 if the cell is white and 1 if the cell is black. Then, we have:

$$\texttt{p[i][j]} = \begin{cases} 0 & \text{if } \texttt{nuclear\_reactor[i][j]} = 1 \\ \frac{1}{\text{number of white cells}} & \text{if } \texttt{nuclear\_reactor[i][j]} = 0 \end{cases}$$

For the TH23-SA74-VERW Reactor Schematic, there are 199 white cells in the schematic. Since the top left corner of this particular schematic is white, the probability that the drone is in the top left corner is $\texttt{p[i][j]} = \frac{1}{199} = 0.00502512562$. The image above shows the Th23-SA74-VERW Reactor Initial Belief for the Submarine Drone Probability Distribution.

Question 1.2: Transition Probability Matrix (5 Points)



The image above shows the probabilities of the drone being at that particular location in the schematic after executing the command DOWN. We defined a move as invalid if the drone runs into a blocked cell or goes out of the bounds of the schematic. In other words, a move that is invalid will keep the drone in its current location.

Suppose (i, j) is any arbitrary node in the reactor and $(i-1, j)$ is a location directly above the node and $(i+1, j)$ is a location directly below the node. The drone is most likely to be at a location (i, j) where $(i+1, j)$ is invalid and $(i-1, j)$ is not invalid. The drone is least likely to be at a location (i, j) where $(i+1, j)$ is valid and $(i-1, j)$ is invalid. All other locations (i, j) have equal probability of occurring. Formally, the update rule for DOWN is:

- initialize `p'` with `p'[i][j]` $= 0$ with same size as original probabilities `p`. $\forall(i, j)$, do:
- if `invalid((i+1,j)) = T`, `p'[i][j] += p[i][j]` else `p'[i+1][j] += p[i][j]`

WLOG, similar update rules can be applied for commands UP, RIGHT, and LEFT.

Question 1.3: Localize Drone Optimally (25 Points)

Our goal is to localize the robot after a sequence of actions. Initially, our probability distribution over the unblocked cells is uniform. Formally, we want to reach a terminal state where the probability distribution has a 1.0 at one of the unblocked cells and a probability of 0.0 everywhere else. Formally, we want to move the belief state about the robot's location from the region of high entropy to the region of lowest entropy.

We can model this scenario as a Markov Decision Process that an agent solves as tuple of (s, a, r, s'). The states s are the belief states about where the agent is currently located. The actions $a \in \mathcal{A}(s)$ the commands the agent can take: UP, DOWN, LEFT, and RIGHT. There is no uncertainty in changing to a state since all state transitions are equally likely, so no probability term is denoted in the policy as would be in the classical formulation of the optimal policy for Bellman Equations. Since the state space is uncountably infinite, we cannot learn/estimate an optimal utility for a particular state with classical value/policy iteration methods without a deep RL approach. Hence, we hand-craft reward R and utility U.

The entropy of $H(x) = -\sum_x p(x) \cdot \log p(x), \forall x > 0$. The information gained from taking action a is $I(s, a) = H(a) - H(s)$. We define a cluster c_k in the probability grid s as a connected island of probability cells reachable from one another within the configuration of unblocked cells in the nuclear reactor schematic. Once we find all k clusters and all the coordinates in each c_k, we find the centroid c_k's location to be $\overline{c_k} = \frac{\sum_{i \in c_k} i}{|c_k|}$. Then, we find the centroid of the centroids of the k probability clusters $\overline{c_k}$ to be the location $\overline{\overline{c_k}} = \frac{\sum_{i \in \overline{c_k}} i}{|\overline{c_k}|}$. Let $d(p1, p2) = \sqrt{(p1_x - p2_x)^2 + (p1_y - p2_y)^2}, F = \frac{\sum_{j \in \overline{c_k}} \sum_{i \in \overline{c_k}} d(i,j)}{|\overline{c_k}|^2}$ and $G = \max_{i \in \overline{c_k}} \big[d(\overline{\overline{c_k}}, i)\big]$. $L(s) = \max_{i,j \in \overline{c_k}, i \neq j}[d(i, j)]$ is another metric that will let us bring clusters close to each other.

We can formulate this problem as a multi-objective reinforcement learning approach for (estimated) optimal control. Crucially, we have three objectives when selecting an action. Generally, we want our agent to select actions that: (1) decrease H, states with lower entropy; (2) decrease f, the average pairwise distance between each cluster's $\overline{c_k}$; (3) decrease g, the cluster k with maximal distance from $\overline{c_k}$ to $\overline{\overline{c_k}}$. Our goal state is when the probability mass of 1 is concentrated in one cell in s, when our objective metrics are $H = 0, F = 0$, and $G = 0$. Observe F or G can be 0 before $H = 0$, so we will have to consider that in our utilities. Also, since we are using heuristics as our utilities (i.e. greater uncertainty), we will discount future rewards by $\beta = 0.90$ and have a two-step forward lookahead to calculate future reward.

$$U(s) = H(s) \cdot \min(0.5, F(s)) \cdot \min(0.5, G(s))$$

$$\text{if } I(s, a) \neq 0, R(s, a) = I(s, a) = H(a) - H(s)$$

$$\text{else } I(s, a) = 0, R(s, a) = H(a) \text{ if } k = 1, \text{ else } L(a)$$

$$\pi_{agent} = \arg\min_{a \in \mathcal{A}(s)} \left[R(s, a) + \beta \sum_{s'' \in \mathcal{A}(s')} \sum_{s' \in \mathcal{A}(a)} U(s'') \right]$$

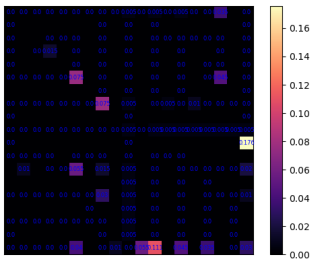


Figure 1: NR2D - 20 Moves

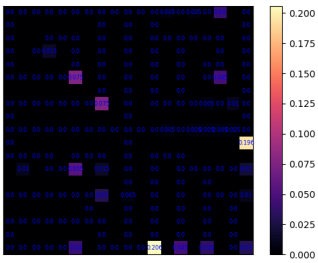


Figure 2: NR2D - 30 Moves

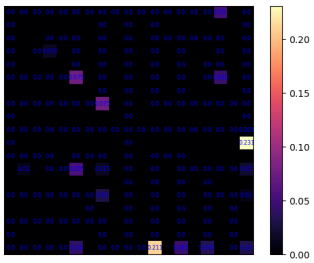


Figure 3: NR2D - 40 Moves

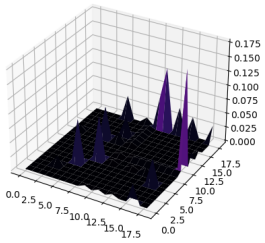


Figure 4: NR3D - 20 Moves

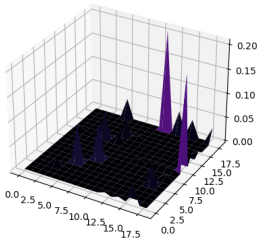


Figure 5: NR3D - 30 Moves

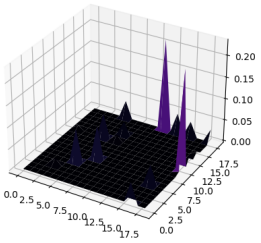


Figure 6: NR3D - 40 Moves

Nuclear Reactor TH23-SA74-VERW Schematic

The defined policy π_{agent} runs into a local optimum on this reactor schematic after a sequence of 40 commands that aims maximize the objective function defined in the previous discussion of methods. For this policy, the agent runs into a local optimum as seen in Figure 3 and Figure 6. The command sequence to get to this local optimum in Figure 3 is denoted below:

```
'D', 'D', 'D', 'D', 'D', 'R', 'R', 'R', 'R', 'R',
'D', 'D', 'D', 'R', 'R', 'D', 'D', 'D', 'D', 'R',
'R', 'D', 'D', 'D', 'D', 'D', 'D', 'R', 'R', 'D',
'R', 'R', 'D', 'R', 'D', 'D', 'R', 'D', 'R', 'R'
```

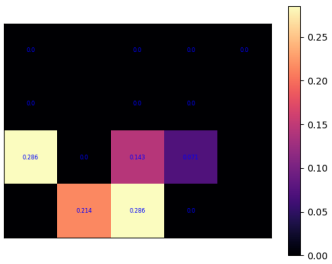


Figure 7: TOY2D - 3 Moves

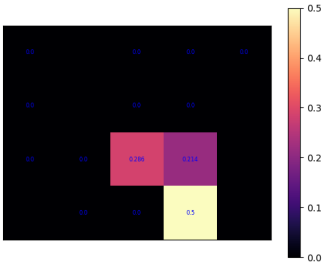


Figure 8: TOY2D - 5 Moves

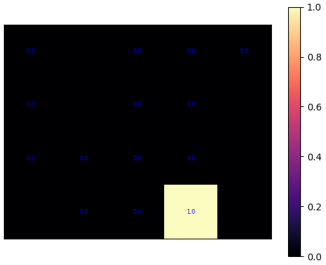


Figure 9: TOY2D - 7 Moves

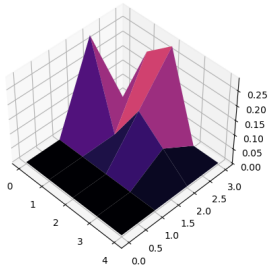


Figure 10: TOY3D - 3 Moves

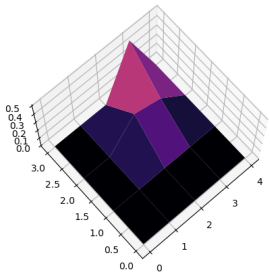


Figure 11: TOY3D - 5 Moves

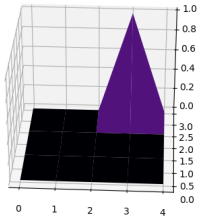


Figure 12: TOY3D - 7 Moves

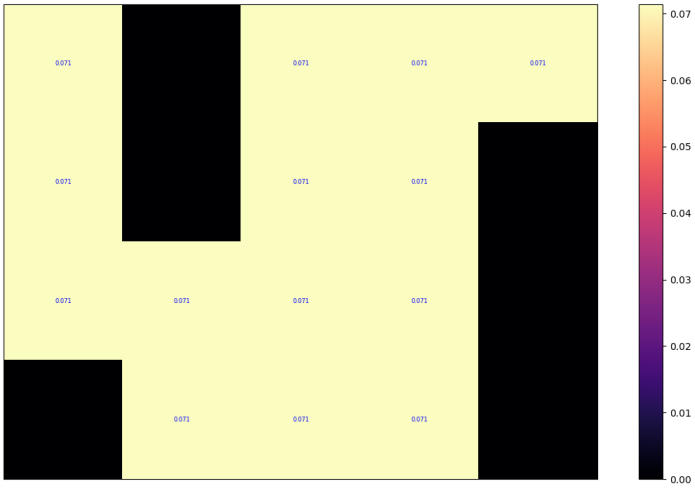


Figure 13: Toy Reactor Schematic

Toy Reactor Schematic

The defined policy π_{agent} reaches the global optimum in a sequence of 7 moves, defined below:

```
L,D,D,R,R,D,R
```

Discussion

We hypothesize that a Multi-Objective Reinforcement Learning (MORL) Agent gets stuck in local optima when the reactor size scales (note that for small reactors, this MORL agent achieves the global optima!) instead of the global optima due to a few reasons. First, we are greedy with respect to utility values that we did not "learn" with classical or deep reinforcement learning methods, so a greedy approach with respect to heuristics fails to consider possible necessary states needed to reach the goal state. To mitigate this issue, I would try a few more things if I had more time: (1) increase the number of moves lookahead for calculating the utilities so that a greedy policy is more likely not to miss important states, (2) use a deep reinforcement learning agent to learn from acting in the game and optimize the policy with some standard algorithm like Proximal Policy Optimization (PPO) or (3) use a strategy of ϵ-greedy to select some actions randomly to get out of local optima so that we don't miss relevant states. Observe that greedily picking actions with respect to minimal entropy will likely lead to local minima; hence, I augmented the reward and utility functions to also consider the distances between the two closest clusters's centroids, the average distance from each cluster to each other, and the maximal distance from the cluster farthest away from the centroid of centroids for all the clusters. Hence, I hoped that the MORL agent would act in the following phases: (1) decrease entropy until we reach local minima, (2) and then sequentially start combining disparate probability clusters together. But, since π_{agent} gets stuck in local minima on large grids, that suggests the reward R and U can be improved to better take into account all of the metrics that I have used as heuristics. If I had more time, I would spend more time better shaping the reward and utilities with respect to how the heuristics should score an action, and also the relative weightage of each heuristic within the reward and utility functions. Note that while this MORL agent doesn't always produce the shortest sequence as the reactor size scales, it generally produces short sequences to any optima (global or local) regardless because the multi-objective optimization framing of this problem necessitates that the agent moves towards regions of low entropy, low average distance between clusters, and low max distance from cluster to the centroid of centroids. Note that I initially had an $A*$ approach with a consistent heuristic $h(s) = H(s)$, guaranteeing optimality, and the cost $f(s_0, s) = I(s_0, s)$ where s_0 is the start state that worked for small configurations but was computationally infeasible as the reactor scaled. I modified the heuristic $h(s) = H(s) + \overline{\overline{c_k}} \cdot k$ and that scaled better but was still intractable as reactor configuration increased - though suboptimal, it produced short sequences still. Hence, I formulated the problem as a MDP instead of a search problem so that running the code would be as quick as possible.

Question 1.4: Longest Shortest Sequence (15 Points)

Note that since $Q1.4$ depends on $Q1.3$, while I have a solution for $Q1.3$, it is not necessarily optimal. Hence, I will assume that I have access to a function called `get_shortest_sequence`, which finds the shortest sequence of commands for any reactor to perfectly localize the robot. Since I did not have the implementation of the optimal sequence length, I have not implemented the algorithm for $Q1.4$ in code. However, the key ideas and algorithm to find the worst possible configuration for a reactor will be the same as detailed below. We will use genetic algorithms, an evolutionary approach, to find bad reactor schematics.

Initialization: Initialize a population of reactors P, where each reactor r_i is a randomly generated configuration of blocked and unblocked cells, ensuring the resulting reactors are valid (i.e., all unblocked cells are connected). Set the maximum number of generations G and the population size N so that we can decide how "bad" of configurations we want. Generally, a high number of configurations and bigger populations will result in reactor configurations that have the longest shortest sequences. The values of those hyperparameters represents the common tradeoff between optimality and efficiency, so those values will depend.

Evaluation: For each reactor r_i in the population, evaluate its fitness f_i by running the `get_shortest_sequence` function on it and returning the length of the shortest path. The fittest reactor is the function that has the highest value for `get_shortest_sequence`.

Selection: Select the fittest reactor from the population to be the parents for the next generation. This can be done using a selection operator such as tournament selection, where a random subset of reactor is chosen and the fittest one is selected as a parent.

Crossover: Generate offspring reactors from the selected parents using crossover. Crossover involves taking sections of the parent reactors and combining them to create new reactors. For example, we can use a one-point crossover operator where we choose a random index p and swap the sections of the parent reactors after that index to create the offspring reactors. Keep changing p until crossover results in a valid reactor (all unblocked cells are connected).

Mutation: Mutate the offspring reactors by flipping the blocked/unblocked status of cells in the reactor. This helps to introduce new solutions and prevent the algorithm from getting stuck in local optima. Only keep mutations that result in valid reactor configurations.

Evaluation: Evaluate the fitness of the offspring reactors and add them to the population.

Termination: Repeat steps 3-6 for a designated number of generations G or until a satisfactory solution is found (i.e. the fitness of the reactors hasn't changed in k generations). The reactor with the highest fitness in the final population is the solution.

Genetic Algorithm Pseudocode For Q1.4

```
def genetic_algorithm(G, N):

  # Initialize population
  P = initialize_population(N)

  for g in range(G):

    # Evaluate fitness of each reactor in population
    for r in P:
      r.fitness = evaluate_fitness(r)

    # Select fittest reactors as parents
    parents = select_parents(P)

    # Generate offspring reactors using crossover and mutation
    offspring = crossover(parents)
    offspring = mutate(offspring)

    # Evaluate fitness of offspring reactors and add to population
    for r in offspring:
      r.fitness = evaluate_fitness(r)
      P.append(r)

  # Return reactor with highest fitness in final population
  return max(P, key=lambda x: x.fitness)
```