# Project 3: Robot Guiddance

Anirban Chakraborty

May 1, 2024

## No Bot

- $$T_{\text{no bot}}(\text{cell}) = 1 + \sum_{\text{neighbor}} P_{\text{cell, neighbor}} \cdot T_{\text{no bot}}(\text{neighbor})$$

  is the equation to calculate the expected steps. For every open cell, $T_{\text{no bot}}(\text{cell})$ is the sum of transitional probabilities of each cell and its neighbors, plus the added step of the crew member going to the neighboring cell. This is based off on Markov's Decision Process, in which when there is no influence, the crew member moving randomly will have decreasing expected steps the closer to the center it gets.

- **Bonus**: Assuming $\beta$ is the discount factor in MDP, $\beta$ is typically used to diminish values of future rewards in cases where initial expected values might be infinite. However, the fixed grid configuration allows me to always have a guarantee that the crew member will eventually hit the center if randomness is uniform, simulation has no limit on time, and the grid configuration is valid i.e crew member can never be trapped by random blocked cells.

- No cells can show infinite expected times as every accessible cell has a pathway to the teleport pad, as stated from the bonus above. The only values allowed be zeroed out are the center and the blocked obstacles.

- I can compute the mean of expected times for all accessible cells to the center. This average would suffice as the expected time for a crew member to reach the teleport pad. Below lists the abbv. view of $T_{\text{no bot}}(\text{cell})$ table of expected values.

  $$\text{Average Expected Time} = \frac{\sum_{\text{open cell}} T_{\text{no bot}}(\text{cell})}{\text{open cells}}$$

```
236  235 233  ...  232  232 232
225 ..  ...     0  ....  ... 215
233  232  ..    ..  .... 222   0
```

## Optimal Bot

- $$T_{\text{bot, crew}}(cell) = \min\{1 + P(cell'|cell) \cdot T_{\text{bot, crew}}(cell')\}$$

  where *cell* and *cell'* represent the current and next states, and $P(cell'|cell)$ are the transition probabilities, adjusted based on the bot's adjacency and direction to guide the crew towards the center. By initializing our guess to 300 (values derived from when there is no bot), I iterate this equation over each cell in the probability matrix and minimize the result. This allows the bot to zero in on the crew member's location and activate the fleeing function of the crew member to herd them towards the center. One idea to note is that it doesn't intuitively make sense that the expected times for each cell is less than 1, meaning the true expected steps may either be computed incorrectly, or I went too far in minimizing steps towards the center, even if the probability logic functions correctly in creating optimal actions. Below is the converged expected time matrix, which averages around 7 iterations:

  ```
  0.3245  0.2980  0.2980  ...  0.2980  0.2980  0.3245
  0.2980  .......  ......  ...  .....  ......  0.2980
  ...     ......  ......  ... ....... ....... ......
  0.3245  0.2980  0.2980  ...  0.2980  0.3245  300.0
  ```

- It took 17 steps for the crew to reach the teleport pad, starting from position (2, 7) and ending at position (5, 5). Path : (3, 4), (2, 5), (1, 6), (1, 7), (1, 8), (1, 9), (2, 9), (2, 8), (1, 8), (2, 8), (2, 7), (2, 6), (2, 5), (2, 6), (2, 5), (2, 6), (2, 5), (3, 5). The Bot seems to always stay in motion in herding the crew member to the center by taking advantage of the crew member's flee. To me, the optimal action to take when the bot needs to stay still is when the minimum value is where they were to begin with e.g when the bot is between the crew member and the teleport pad. In all cases when the crew member is closer to the center than the bot, then all optimal actions should ideally be in motion to keep following the crew member until it gets to the center.

- The bot's optimal position at (7, 8) is the result of running simulations at each bot's position over every valid open cell in the grid with the same crew spawn at (2, 7) with the expected time of 1. While I can't get much from this optimal position, it seems to me that the optimal position for a bot to take would be aligned with one of the corners. That way, we can always account for the crew's distance to the center to be less than the bot, and for the bot to immediately start moving towards the crew. This assumes that the bot does not know where the crew member would spawn. If the bot does know, running the optimal simulation will give the best position for the bot to universally minimize the expected time, compared to any other valid position.

- The time to escape with the bot drastically lowers the time steps compared to having no bot at all. Without the bot existing, the crew member takes 135 steps to reach the center, compared to 17 from before.

## 1 Learned Bot

- The neural network is a CNN that has an input layer of 4 nodes corresponding to the bot and crew positions, batch normalization to speed up convergence, dropout layers to prevent overfitting, Rectified Linear Unit for my activation function, and softmax to predict the optimal action for the bot from nine possible outcomes (the bot actions and whether the simulation is a success or not. I duplicated the inner structure before the prediction as well for more computation. The inputs are normalized for lower complexity and the output from (action, success) is reduced to a binary value for the same reason (The success parameter is more for generalizing, it's not really factored in for just the Learned Bot). The model is trained with back-propagation with the standard Negative Log Likelyhood Loss function and an Adam optimizer with a learning rate of .1%. I wanted to make sure my model was small enough to reduce the amount of time for each epoch and to prevent over fitting, so it took some trial and error to stop at this structure. Duplicating my initial structure and converging on an ideal number of nodes in each layer (128) proved to be the most effective in terms of computation done and time taken.
- I achieve around 80% accuracy with my model. To prevent overfitting, I use dropout layers to randomize neurons to ignore to make sure my model isn't simply trying to replicate the states in my training data. I also made sure to reduce the amount of neurons around 128 for each layer from trial and error until I nailed at least above 80%. I tried to implement L1/L2 Regularization, but my loss function increased, while my accuracy decreased. Having my training data to be very minimal, however, makes it hard to determine if I am over fitting, even if I tried to reduce complexity by normalizing my input data, including dropout layers, and simplifying my output data to only match a binary value each time step. Utilizing the optimal simulation with my training method would be the ideal way to test different configurations, but increases complexity with how my model trains itself. Thus, I added a success parameter to indicate every state that leads to a success, which is utilized more effectively in the generalized data.

## Generalizing

- I generated training data across at most, 10000 valid configurations, minus some configuration that exceed the 100000 time step due to bad random generation of blocked obstacles and crew/bot spawn. I modified my model to adjust the success parameter within the output value, along with action from the previous model, to indicate the different steps it takes to reach the center, and to restart. This means the training data does technically separate data from every action that leads to a success state, though this may not be the best way to handle generalized data. However, I lead with a higher accuracy of around 88% from approx. 1000 to 10000 configurations, leading me to believe that it's unfeasible to test around 30 billion valid configurations minus configurations that lead to infinite expected steps from bad grid spawns. The mode remains the same, but every epoch takes 2 minutes for 10000 inputs. Without the success parameter, the accuracy shoots down to less than 10%, leading me to believe that segmenting each successful run is necessary for optimal accuracy.

## Alien

While I didn't code for Alien, I want to discuss the implementation for fun since I had an extra day. It seems obvious to me that the Alien is effectively a crew member that the bot acts as a buffer between the other correct crew member. Excluding improbable situations where the Alien and Crew spawn next to each other and collide, the Bot would ideally place itself between the crew member and Alien, in addition to "herding" the crew member in by taking advantage of the fleeing function of the crew member and the fleeing function of the Alien. Scenarios where the Alien, then the Bot, and then the Crew member are respectively that far from the center would probably need the Bot to simply act as the buffer and stay in motion/still until the order is switched from Crew, Bot, Alien, or the simulation just ends successfully. For training the neural network and generalizing, probably just including only successful scenarios and keeping the same training data from before would be the best since Alien coordinates aren't necessary. The Learned Bot is simply replicating successful steps the Bot took.