**Intro To AI, HW2:**

Part A – Improved-Greedy:

1. The definition of the game is such that:

In our case, it was given that . Therefore:

1. In order to define a smart heuristic that leads to a good outcome, we’re going to take into account the following variables of the environment into consideration:
2. Env.packages
3. Env.robots
4. Env.charge\_stations

We’ll find the distance between our agent and the nearest charge station and denote it as “closest\_station”.

If the agent is not holding a package, we have to walk at least through tiles to pick up a package, therefore we’ll define to be:

Else our agent is already holding a package and we have to we walk at least through tiles to deliver it to its destination, therefore we’ll define to be:

If we don’t have enough battery to deliver / pick up the package therefore, we should head towards a charging station to charge up. Therefore,

Return

Otherwise, we have enough battery to deliver / pick up the package therefore, we should head towards the package itself or towards its destination. Therefore,

Return

In the first return value, we want to encourage recharging the agent's battery, which happens when we reach a charging station and exchange the agent's credits for battery points. This means we prefer a state in which we have exchanged our credit points for battery points. We also prefer a state in which we are close to a charging station when we can't reach the destination.

In the second return value, we want to encourage picking up a package. Therefore, when picking up a package, we give a bonus of 100, and we also prefer having more credit points. Therefore, we add the agent's credit points to the return value. However, we want to minimize the distance to the destination, so we subtract the distance from the return value.

1. The main disadvantage of using Greedy as opposed to Minimax is, in Greedy the agent takes the locally optimal decision at every step of the game. Whereas in Minimax we explore the game tree (to a certain depth or until a final state has been reached) such that at each step we try maximizing the utility of the agent and minimizing the utility of the adversary. Thus, a possible outcome is that while using Greedy we might take a step that maximizes the utility of our agent locally but in doing so, we reach a state that results in our loss. Whereas in minimax the algorithm would’ve looked a few steps ahead and decided to take step in a different direction that will pay off in the future and eventually win the game.

Part B – RB-Minimax:

1. Using an easy-to-calculate heuristic in RB-Minimax is advantageous because it doesn’t waste much time on computing the heuristic value. Therefore, we would be able to delve deeper into the game tree and obtain better results than those computed at a lower depth which we would most likely have stopped at if we had used a heuristic that is difficult-to-calculate. However, it could also be disadvantageous in some cases. For instance, when the heuristic is very misleading, it can waste precious time on taking bad steps that might result in a draw or a loss. In contrast, a more informed but difficult-to-calculate heuristic can provide actual good steps and lead to a win.
2. The behavior in Dana’s algorithm doesn’t necessarily mean she has a bug. For example, in case there are 2 goals such that one is 1 step away and the second goal is 2 steps away such that the utility that’s gained from the second goal is a lot bigger than that of the first goal. In such a case the algorithm will choose to take the path to the second goal even though it was possible to reach one of the goals in one step and end the game. The reason for which is to maximize the agent’s utility by reaching the “better” goal.
3. Minimax algorithm with K players in a zero-sum game:
4. In this case, each agent wants to win. Therefore, each agent will play as a “MAX” player in order to maximize his utility. While assuming that all other agents want to minimize his utility. In order to account for the fact that there is other player trying to minimize his utility and not just one, in the “MIN” loop we will calculate the minimum value after taking 1 step for each of the adversary agents and calculate the min-value after wards. Such that the min-value that is calculated takes into account all possible moves by the other agents and picks the one with the minimum-value, such that our agent takes into account the worst-case scenario. In short, the min-value that we calculated corresponds to a tuple of moves by other adversaries. To clarify, for each agent, his adversaries are the other agents which differ from agent to agent.
5. In this case, all other agents want us to lose. Therefore, we will consider all of the other

agents as a collective to be our adversary such that whenever one of the agents win, we lose. Our agent is trying to win therefore he will act as a “MAX” player and our adversary is also trying to win therefore they will act as a “MIN” player such that our adversary includes agents such that a move by our adversary represents an individual move by each of the agents that make up our adversary. To clarify, in this section the adversary that includes agents is constant and doesn’t change based on the agent in opposition to the previous section.

1. In this case, each agent wants the agent next in line to win. Therefore, each agent will try to maximize the utility of the agent next in line while also trying to minimize the utility of all other agents. In order to do that, in the heuristic-utility function we will return the value according to the heuristic-value of the agent next in line instead of our own agent. Moreover, in the “MIN” loop we will take a step for each one of the other agents and pick the value that minimizes the utility of our agent that corresponds to a tuple of moves made by the other agents (which means the utility that is most likely to lead to the loss of the agent next in line). And in the “MAX” loop we will act as we did previously, picking the value that has the highest value which corresponds to one move made by our agent. To clarify, for each agent the other agents differ.

Part C – Alpha-Beta:

1. Yes, the Alpha-Beta agent might behave differently from the minimax agent that we implemented in the previous section. The reason for the difference in behavior is due to the fact that the alpha-beta agent doesn’t explore all sub trees and prunes those that it knows will not affect the result of its decision. Therefore, within the same time limit the alpha-beta agent will reach a greater depth than that reached by the minimax agent. Which could result in different choices because a greater depth means that the agent is more informed and is able to make better choices. In addition to the potential difference in choices, there’s a difference in the speed for which solutions are computed for a given depth that are in favor of the alpha-beta agent.

Part D – Expectimax:

1. In such a case where we are playing against a completely random agent, I would use a uniform probability to explore all sub-trees equally because there is an equal chance to pick each one of the legal operators at every step and therefore, we shouldn’t prioritize one over the other.
2. In case of a bounded heuristic-value function like the one we have, , we can calculate the lower bound and upper bound for chance nodes which we will denote as: with the help of which we can calculate upper and lower bounds for the parent node up till the root. If we reach a lower bound equal to 1, we know that we’re not going to get a higher value because the heuristic-value function in bounded and therefore, we can prune all other sub-trees that haven’t been explored yet. And similarly, if we reach an upper bound that’s equal to we know that we’re not going to find a lower value and therefore we can prune all other sub-trees that we haven’t explored yet.

Part E – Games with large branching factor

1. Possible changes to Bina warehouses:
2. Increasing the board size and adding barriers will not add any new operations that we couldn’t perform prior to the change. The only difference being is that now is that there are slots that the agent can’t pass through and therefore, in slots near a barrier the number of legal operators will be less depending on how many barriers surround the agent. However, the upper bound on the branching factor still stands as it was before when the agent could perform up to 7 operations. In conclusion, .
3. Adding the ability to place a block will add multiple operations and therefore, increases the branching factor. For each empty slot on the board there is a new operation that is added such that the number of empty slots is dependent on whether the robots are holding packages or not, and which slot the robot is standing on. There are 2 charging stations on the board therefore, there will always be 2 slots that are taken. In addition to that, if the robots aren’t holding packages there is at least 1 slot that is taken by the packages, The destination of a package could be the same as its point of origin or on top of a charging station. The worst-case scenario for the branching factor is when one of the agents is holding a package and standing on top of the delivery location which also houses a charging station and the other package. While the other agent is standing on top of the other charging station package-less, in which case there are only 2 slots that are taken on the board. Therefore, the first agent can place a block on top of  other slots but he can also perform the 7 original operations {

. In summary, the new branching factor stands at:

. If then .

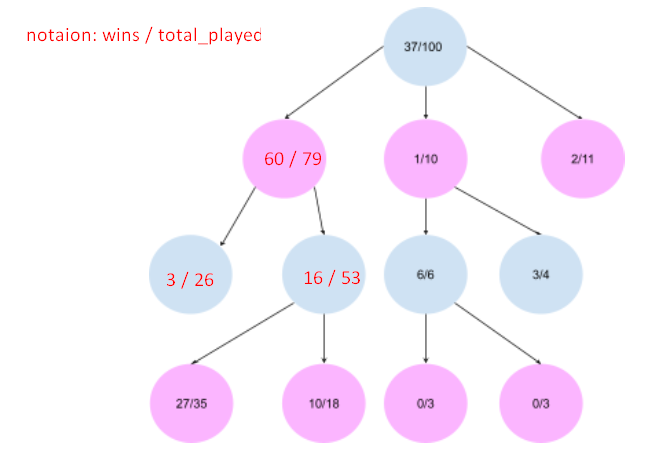
1. Implemented the second change for the environment:
2. Yes, we can use the Improved-Greedy algorithm on the game after adding the new changes which could have a running-time that is similar to the running-time we had before adding in the new changes. Because the running time it takes the Improved-Greedy algorithm to compute a step is proportional to the branching factor and . before adding the changes, it was and we can see that the difference isn’t very large. Whereas for other algorithms such as minimax the running-time it takes to compute a step is that has become approximately after adding in the changes which is much larger compared to the complexity before adding in the changes. For example, for

.

1. I would suggest using the MCTS algorithm as it’s a great, general algorithm for dealing with games with large branching factors like we have after adding the option to place blocks on empty slots (), it can take advantage of knowledge when it’s present and make better decisions. Moreover, it’s an algorithm that can be run anytime which is great for us because we’re time bounded and therefore, we don’t want to take any risk on algorithms that may use up all of our time without returning a result. And most importantly MCTS has a much lower time-complexity compared to minimax and therefore, can perform many more iterations.

Part F – MCTS

1. Here’s the completed graph:



1. The next node to choose in the selection phase:

First action:

Pick:

Second action:

Pick:

Third action:

Therefore, the next node in the selection phase is the one with the values because it has the highest value as we can see in the calculations above.

1. The formula for calculating a UCB value of a node is as follows: .

Since every simulation ends with Tal’s victory, for every pink node it will have the following ratio: (n / n). Whereas for every blue node it will have the following ratio: (0/ played\_trials).

We’ll denote the number of trials for the first most ancient ancestor: and similarly we’ll denote the number of trials for the second most ancient ancestor: , and we’ll denote the number of trials by their parent to be . Similarly, we’ll define the values of the first and second most ancient ancestor to be . But since Tal wins every game, we know that the values of pink nodes are equal to 1. Therefore,

At first the UCB value of the first ancient ancestor is the greatest and in order to pick a descendant of the second ancient ancestor we want the UCB value of the second ancient ancestor to be greater than that of the first ancient ancestor.

Therefore, we’ll want: .

A sufficient condition to pick a descendant of the second most ancient ancestor is: .

1. In order to prefer exploration over exploitation, we’ll want the ratio between exploration to exploitation to be greater than 1:

Therefore, in order to prefer exploration, we’ll adjust N(s) to be:

.