Intro To AI, HW2:

Part A – Improved-Greedy:

1) The definition of the game is (S, O, I, G) such that:
   Each state in S is represented by a 5x5 Matrix which show the locations of charging stations, robots, whether they carry packages or not, locations of packages, credit and battery for each robot, and number of remaining steps to the game.

$$O = \{\text{Move-north, Move-south, Move-east, Move-west,Pick up, Charge, Drop off}\}$$

$$I = initial\ state, in\ which\ robots\ start\ with\ battery = 20, and\ zero\ credit\ ..$$

$$G = \{all\ states\ in\ which\ the\ battery\ of\ both\ robots\ have\ been\ exhausted\ completely\ or\ that\ the\ remaining\ number\ of\ steps\ reached\ zero\}$$

2) 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:
   i) Env.packages
   ii) Env.robots
   iii) 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 $dist$ tiles to pick up a package, therefore we'll define $dist$ to be:

$$dist = Manhattan\_Distance(\ agent.position, nearest\_package\_on\_board)$$

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

$$dist = Manhattan\_Distance(\ agent.position, agent.package.destination)$$

   If $agent.battery < dist$: 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 $(-(\ agent.credit\ +\ closest\_station)\ )$

   Using this 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.

Otherwise, if we're not short on battery that we have enough battery to deliver / pick up the package, we should head towards the package itself or towards its destination. Therefore,

$$\text{Return } (100 \cdot 1_{\{agent.package\ is\ not\ None\}}) + (100 \cdot agent.credit) - dist$$

In this value, we want to encourage picking up a package, thus when picking up a package, we give a bonus of 100, in the same time we want to encourage the agent to drop the package when it finally reaches the destination which is reflected in the (100*agent.credit) value. Finally, what drives the agent towards the closest package/ towards the shortest path to its package destination is the fact that we want to minimize that distance through the $'(-1) * dist'$ value.

4) 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, and unlike the Minimax Agent, it doesn't consider the future implications of its decisions throughout the game, thus it can easily fall into a sub-optimal solution. As opposed to the Minimax one which is guaranteed to be a win in most cases.

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. For example, we could be in a MAX state that has two children: - one that's a final state with utility value = 10, and the other is a MIN state with two children that have values: 20 and 30. in this case the algorithm would prefer to delay finishing the game by choosing the first child with value 10 and go deeper in the game.

4) Minimax algorithm with K players in a zero-sum game:
   a) In this case, each agent wants to maximize its own Utility function that's independent of other players, for example each agent would choose the action which gets him to more credit points. So, for each agent we return the node which gets him the maximum utility/heuristic value.

   b) 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. So assume we have N players in the game, we'll define the Utility function from each final state for each player to be Minus the sum of utility function values of all other players, meaning:- $\forall i \leq N, s \in G : U^i(s) = -\left(\sum_{j \neq i}^{1 \leq j \leq N} U^j(s)\right)$ and we try to use the same pattern in the heuristic function as well, so when going down in the recursion we simply return the max value from each state. This would ensure that a state good for one player would probably be bad for his adversaries and vice versa.

   c) 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 $K - 1$ agents. We define a new utility function based on the previous one (in part b) as following: $\forall i \leq N, s \in G : U^i_{NEW}(s) = U^{i+1}(s)$.

Part C – Alpha-Beta:

2) 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) For probabilistic games, that mostly involve a cube: we consider the cube to be also a player, and we call its nodes: chance nodes, such that these nodes are supposed to return the expected value from all its children's Minimax Values. In order to combine pruning with Expectimax, we use the standard alpha-beta technique on nodes that are not child nodes, and for each child node we try to find an upper and lower bound on its returned expected value, for example if one child node has 3 children: one with p = 0.5 and value = 1, the 2nd with p = 0.4 and value 0.8 thus the third node would be with p = 0.1, then we know that the total expected value would be between: $0.5 * 1 + 0.4 * 0.8 - 0.1 * 1 = 0.72 \quad and \quad 0.5 * 1 + 0.4 * 0.8 + 0.1 * 1 = 0.92$,

   We propagate this value back to our parent node so it can update its alpha-beta values and maybe prune some of the subtrees if needed.

Part E – Games with large branching factor

1) Possible changes to Bina warehouses:
   a) 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 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, $b \leq 7 \rightarrow b_{worst-case} = 7.$

   b) 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 $(boardsize^2 - 2)$ other slots but he can also perform the 7 original operations $\{drop-off, left, right, up, down, pick-up, recharge\}$. In summary, the new branching factor stands at: $b = 7 + (boardsize^2 - 2) = boardsize^2 + 5$. If $boardsize = 5$ then $b_{worst-case} = 30.$

2) Implemented the second change for the environment:
   a) 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 linear to the branching factor $O(b')$, thus the effect won't be that significant.

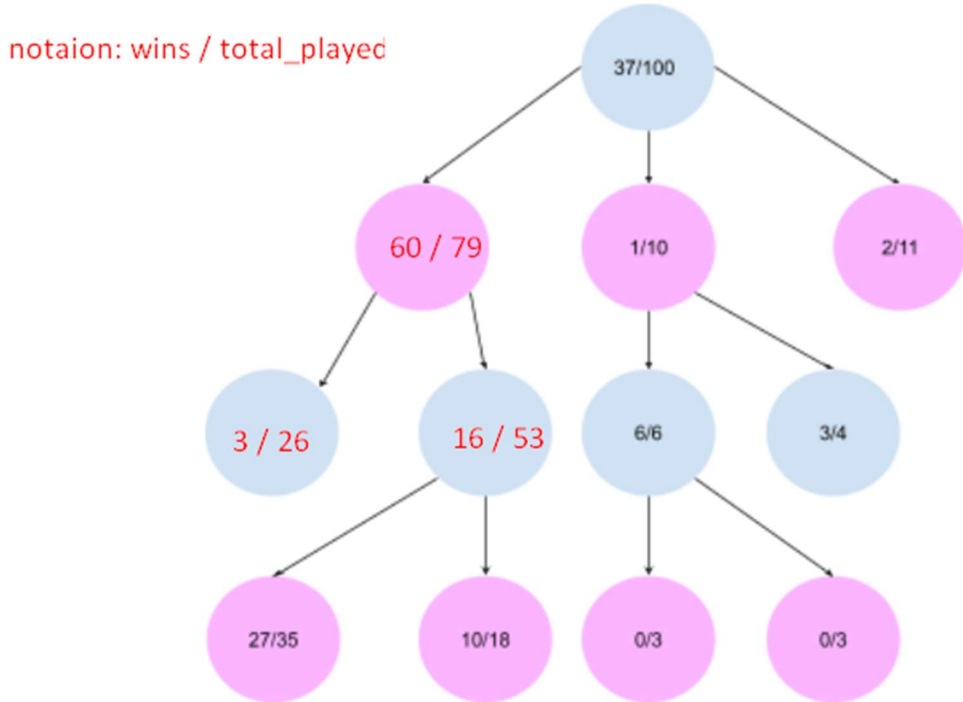
      Whereas for other algorithms such as minimax the running-time it takes to compute a step is $O(b^d)$ that has become approximately $O((b+23)^d)$ after adding in the changes, and we know that: $b'^d = O((b+23)^d)$
      thus, the change would affect runtime exponentially, for example
      $d = 5, O(b^d) = O(7^5 = 16807), O(b'^d) = O(30^5 = 24300000).$

   b) 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 ($b = 30$), 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:



2) The next node to choose in the selection phase:

First action:

$$node_{left} = \frac{60}{79} + \sqrt{2} * (\sqrt{\frac{\ln(100)}{79}}) = 1.1$$

$$node_{middle} = \frac{1}{10} + \sqrt{2} * \left(\sqrt{\frac{\ln(100)}{10}}\right) = 1.06$$

$$node_{right} = \frac{2}{11} + \sqrt{2} * \left(\sqrt{\frac{\ln(100)}{11}}\right) = 1.09$$

Pick: $node_{left}$

Second action:

$$node_{left} = \frac{3}{26} + \sqrt{2} * (\sqrt{\frac{\ln(79)}{26}}) = 0.69$$

$$node_{right} = \frac{16}{53} + \sqrt{2} * \left(\sqrt{\frac{\ln(79)}{53}}\right) = 0.71$$

Pick: $node_{right}$

Third action:

$$node_{left} = \frac{27}{35} + \sqrt{2} * (\sqrt{\frac{\ln(53)}{35}}) = 1.24$$

$$node_{right} = \frac{10}{18} + \sqrt{2} * \left(\sqrt{\frac{\ln(53)}{18}}\right) = 1.21$$

Therefore, the next node in the selection phase is the one with the values $27/35$ because it has the highest value as we can see in the calculations above.

3) The formula for calculating a UCB value of a node is as follows: $UCB_{node} = v_i + C * \sqrt{\frac{\ln(N)}{n_i}}$.

   Now if we run a winning simulation from the rightmost state (currently: 2/11), we get new UCB values:

$$\frac{60}{79} + \sqrt{2} * \sqrt{\frac{\ln(100+x)}{79}} < \frac{2+x}{11+x} + \sqrt{2} * \sqrt{\frac{\ln(100+x)}{11+x}}$$
$$\leftrightarrow x \geq 1$$

   In which case the rightmost state (currently 2/11) becomes the one with the maximum UCB value out of the three states in depth = 1, and thus it will be chosen in selection for expansion.

4) In order to focus more on exploration rather than exploitation, we want to modify the UCB value: $UCB_{(s)} = \frac{U(s)}{N(s)} + C * \sqrt{\frac{\ln(N(s.parent))}{N(s)}}$ where N(s) is the total number of simulations that went through s, in a way equivalent to multiplying the 'C' constant with some value greater than 1, if we multiply N(s) by a constant b > 1, then we get a new UCB value:

$$UCB_{new} = \frac{U(s)}{b * N(s)} + C * \sqrt{\frac{\ln(N(s.parent))}{b * N(s)}} = \frac{1}{b} * \frac{U(s)}{N(s)} + \frac{1}{\sqrt{b}} * C * \sqrt{\frac{\ln(N(s.parent))}{N(s)}}$$
$$= \frac{1}{b} * \left(\frac{U(s)}{N(s)} + (\sqrt{b} * C) * \sqrt{\frac{\ln(N(s.parent))}{N(s)}}\right)$$

$$which\ is\ equivalent\ to\ using:\ UCB_{new} = \frac{U(s)}{N(s)} + (\sqrt{b} * C) * \sqrt{\frac{\ln(N(s.parent))}{N(s)}}$$

   This way we give more weight to the exploration factor and focus more on exploration.