# Maxime Kotur CSCI_6511 - Extra Credit Homeworks

## Fruit Sort Problem:

The problem is to sort 10 apples, 10 bananas, and 10 oranges of different sizes in a 3x10 array such that the fruits go from top to bottom in ascending order of size. We can use the A* algorithm to minimize the number of moves required to achieve the sorted state.

The A* algorithm uses a priority queue to store and prioritize the states of the problem. At each step, it selects the state with the lowest estimated cost (which is the sum of the actual cost of reaching the state from the initial state and the heuristic cost of reaching the goal state from the state) and generates its successors.

To apply the A* algorithm to the fruit sorting problem, we first define an initial state of the fruits, which is a 3x10 array of apples, bananas, and oranges of different sizes, randomly shuffled. We also define a goal state of the fruits, which is the sorted 3x10 array of apples, bananas, and oranges.

Then, we define a heuristic function that estimates the cost of reaching the goal state from a given state. For this problem, we can use the Manhattan distance as the heuristic, which calculates the sum of the absolute differences in the row and column positions between each fruit in the given state and its corresponding fruit in the goal state.

Next, we define a generate_successors function, which generates all possible successors of a given state by swapping two adjacent fruits horizontally or vertically. We iterate through all possible positions of adjacent fruits in the given state, and for each position, we create a new state by swapping the fruits and adding it to a list of successors.

Finally, we apply the A* algorithm to the problem by initializing an open list with the initial state and its estimated cost, and a closed list with no states. We then repeat the following steps until the open list is empty or the goal state is reached:

Select the state with the lowest estimated cost from the open list.
If the selected state is the goal state, return the solution.
Generate the successors of the selected state using the generate_successors function.
For each successor, calculate its actual cost and estimated cost using the cost of the selected state and the heuristic function, and add it to the open list if it has not been visited before.
Add the selected state to the closed list.
When a solution is found, the algorithm returns the number of states from the initial state to the goal state, which represents the optimal moves required to sort the fruits.

**Snake Grid Sorting Problem**

The variables in this problem are the letters that we need to assign values to in order to solve the puzzle. Each variable represents a unique letter. We used a list of variables to represent the puzzle, with each variable corresponding to a letter in the puzzle.

The domain of each variable is a set of possible values that the variable can take. In our case, each variable has a domain of [A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z]. This means that each variable can take on any one of the 26 letters of the alphabet.

The constraints are rules that limit the possible values that each variable can take. In our case, we have several constraints. Each variable must be assigned a unique value, so we need to ensure that each letter is assigned to a different value. Each letter must be assigned a value that satisfies the rules of the puzzle. These rules include making sure that no two letters have the same value, that adjacent letters in the puzzle have values that differ by exactly 1, and that placing a letter does not make any adjacent letters invalid.

To solve the CSP, I used a backtracking search algorithm. The algorithm starts by assigning a value to one of the variables, and then checks if the assignment satisfies all of the constraints using the is_valid function. If the assignment is valid, the algorithm moves on to the next variable and repeats the process. If the assignment is not valid, the algorithm backtracks to the previous variable and tries a different value. The algorithm continues this process until a valid solution is found or all possible assignments have been tried.

Overall, the CSP approach allowed me to represent the alphametics puzzle in a way that made it easy to define the constraints and to search for a solution. The backtracking search algorithm provided an efficient way to explore the solution space and to find a valid solution.