**Problem Anlysis and Design:**

The Representation is as follows :

* **State :** A tuple (F, G, W C), in this each letter represents the position (side of the river) of Farmer, Goat, Wolf, and Cabbage respectively. Here 0’s and 1’s are used. For suppose 0 represents that it is on East side and 1 represents that it is on west side.
* **Initial State :** Here all the elements at the start are on the east side so will represent them as (0,0,0,0).
* **Goal State :** All of these need to get to other side of the river i.e., west side of the river. So it can be represented as (1,1,1,1).
* **Valid moves :** Define moves that change the state while adhering to the problem's constraints (e.g., while moving the wolf cannot be left alone with the goat, and the goat cannot be left alone with the cabbage).
* **Unsafe States or dead State :** Identify and avoid states where the entities eat each other, such as (1,0,0,0), (0,1,1,0), (1,0,1,0), (1, 0, 0, 1), (0, 1, 1, 0) etc,.

Solution states to reach the goal state is :

Farmer: 0, Goat: 0, Wolf: 0, Cabbage: 0

Farmer: 1, Goat: 1, Wolf: 0, Cabbage: 0

Farmer: 0, Goat: 1, Wolf: 0, Cabbage: 0

Farmer: 1, Goat: 1, Wolf: 0, Cabbage: 1

Farmer: 0, Goat: 0, Wolf: 0, Cabbage: 1

Farmer: 1, Goat: 0, Wolf: 1, Cabbage: 1

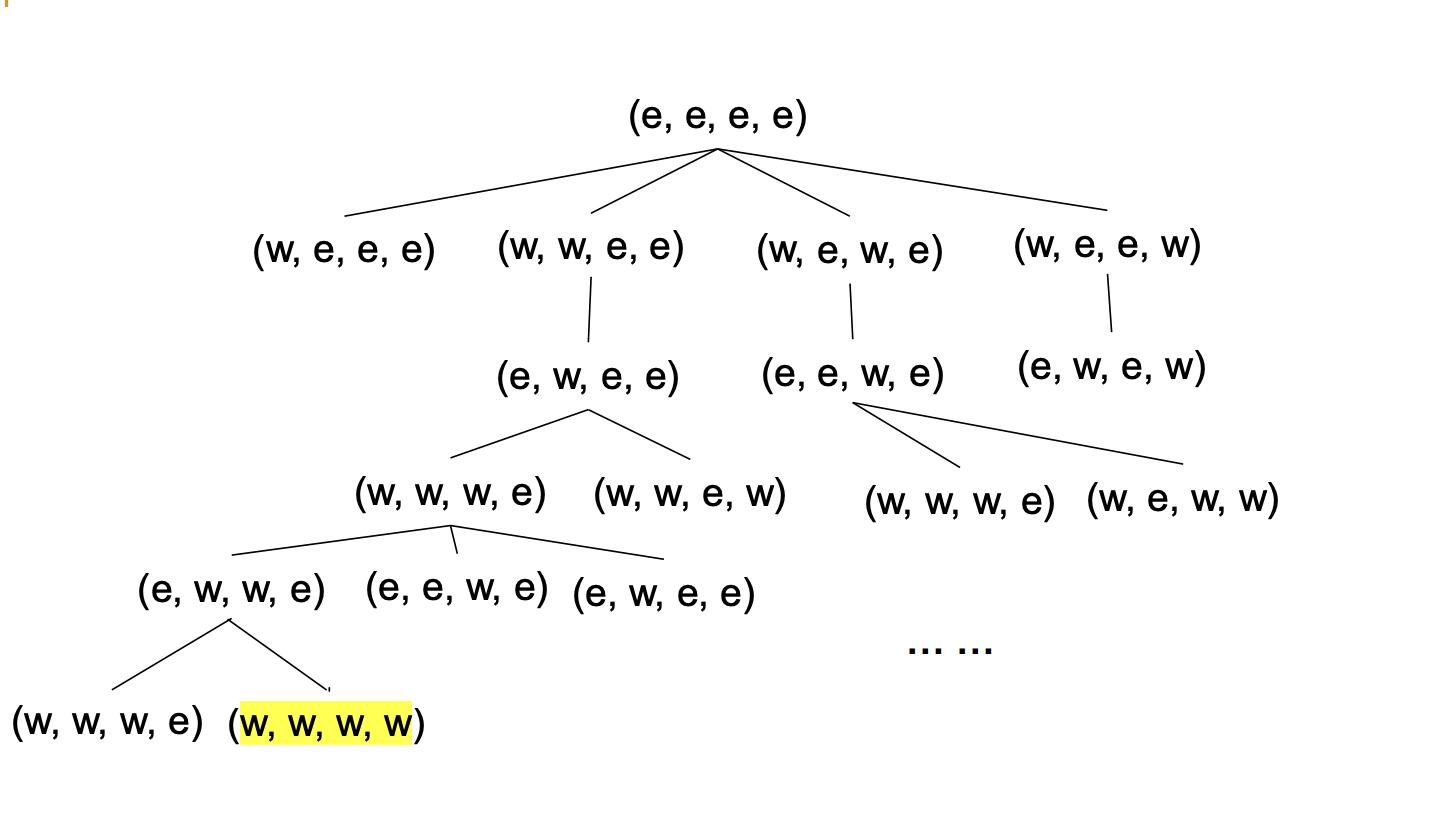
Farmer: 0, Goat: 0, Wolf: 1, Cabbage: 1

Farmer: 1, Goat: 1, Wolf: 1, Cabbage: 1

**Representation of the State Space (at least 8 levels)**:

1. The initial state is (0,0,0,0) – All elements on east side
2. Moving farmer and Goat on the west side : (1,1,0,0)
3. Farmer returns alone to the east side : (0,1,0,0)
4. Moving the farmer and cabbage to the west side : (1,1,0,1)
5. Moving farmer and Goat to the east side : (0,0,0,1)
6. Moving the farmer and wolf to west side : (1,0,1,1)
7. Farmer returns alone to the east side : (0,0,1,1)
8. Moving the farmer and Goat to west side : (1,1,1,1)

This steps are shown in the below state representation :



**Design of heuristic evaluation function in the A\* algorithm:**

* **State representation :** Here the ‘state’ class represents the state of problem, where each state includes the location of the farmer, goat, wolf and cabbage.
* **Heuristic function :** Here this heuristic function ‘heuristic(state)’ estimates the cost to reach the goal state. In the given code, it counts the number of items not on the east side.
* **A\* search(a\_star\_search)\*\* :** This process runs on A\* search. It initializes an open set and closed set, then updates the open set based on the total cost(g-score plus heuristic) while exploring states. The ACTIONS list outlines the potential course of action and verifies that it is valid(avoids dangerous states).

It's vital to note that A\* search often works better for resolving this issue because it uncovers the ideal answer.

**Selection of heuristic function :**

* The chosen heuristic function is pretty straightforward. The number of things (goat, wolf, and cabbage) that are now on the east side of the river is calculated, and it is assumed that transferring them to the west will increase the cost of getting to the desired state.
* Due to the fact that it never underestimates the actual cost, this heuristic is acceptable. In other words, because it doesn't account for any potential interdependencies or non-linear consequences of moving things, it consistently underestimates the number of steps necessary to achieve the goal.
* This heuristic was chosen because it offers a rapid estimation of the amount of labour still needed to move everything to the right bank, making it appropriate for this situation.

**Effectiveness of Heuristic function :**

* The heuristic function's efficiency is determined by how successfully it directs the search process towards the desired state while minimising pointless exploration.
* The heuristic is quite useful in this situation for directing the A\* search process. Assuming that these states are nearer the objective, it enables the algorithm to give higher priority to states where there are fewer items on the east side.
* The heuristic does not, however, account for the limitations of the issue, such as the requirement to prevent instances in which the goat or the wolf might consume the cabbage. As a result, the heuristic could occasionally undervalue the actual cost.
* However, A\* search is still likely to uncover an ideal solution because it uses this heuristic in conjunction with the real cost (g-score) to estimate the overall cost.

**Avoiding Unsafe States :**

Here the unsafe states can be avoided by the condition given in the problem description. The conditions include : while moving the wolf cannot be left alone with the goat, and the goat cannot be left alone with the cabbage. So, anytime in the execution of the program if this condition violates, then we can mark that as dead state and should not move forward with that state.

**Program Structure:**

**DFS :**

1. **Class ‘State’ :**

* This class, which includes the positions of the farmer, goat, wolf, and cabbage, represents a state in the problem.
* Attributes of this class are : farmer, goat, wolf, cabbage, parent(shows the parent state, which aids in tracing the route from the starting point to the final state.)

1. **Methods :**

* The methods is\_valid(), is\_goal(), \_\_eq\_\_(), \_\_hash\_\_(), \_\_str\_\_() has its own definitions.

1. **Actions :**

* The farmer can choose from the following list of potential actions, each of which is represented as a 4-tuple: (farmer\_action, goat\_action, wolf\_action, cabbage\_action).
* If the farmer crosses the river with the item, farmer\_action is 1; otherwise, it is 0.

Whether the goat, wolf, or cabbage is carried across the river is indicated by the variables goat\_action, wolf\_action, and cabbage\_action (1 if carried, 0 otherwise).

1. **dfs(initial\_state) :**

* To solve the issue, this function uses depth-first search technology.
* Initial\_state : The starting state for the search.
* Stack : A structured data to perform DFS.
* Visited : A set to keep track of visited states.
* It starts with the initial\_state on the stack and begins exploring states.For each state, it checks if it's the goal state, in which case it returns the goal state.
* It keeps track of visited states in the visited set. It generates new states by applying each possible action and checks if the new state is valid, whether it's already visited, or if it's a "dead" state.
* It prints information about the states to help debug and understand the DFS process.

**A\* Search Algorithm :**

* The a\_star\_search() function performs the A\* search algorithm to find a solution to the problem.

**Functionality**:

* It uses a priority queue (heap) to explore states with lower total costs first.
* It tracks the cost to reach each state (g\_score).
* It maintains a came\_from dictionary for backtracking to find the path to the solution.
* It terminates after a specified number of iterations or a time limit (60 seconds in the provided code).
* Returns the goal state or None if no solution is found.

The programme displays the results of the A\* search, which are a series of states that show how to relocate every object to the west bank. It prints a message stating that if no solution is discovered.