CSEN 901: Introduction to Artificial Intelligence
Project 2: Simplified Water Sort Logic Agent

Project Report

Abdelrahman Aboelkehir, Hossain Ghoraba,
Mahmoud Aboelenein, Omar Hesham

21.11.2024

1 Introduction

This project is the implementation of logic agent to solve a simplified version of the water sort problem that we solved in Project 1. The logic agent is implemented in Prolog and reasons using a successor state axiom.

In this version of the problem, there are 3 bottles each with a capacity of 2 layers, and there are 2 possible colors. The goal is to have each bottle be either completely filled with one color, or completely empty.

2 Fluents

We consider there to be one fluent, called state(bottle1(C1, C2), bottle2(C3, C4), bottle3(C5, C6)) that captures the state of the bottles, and we write a successor state axiom state(bottle1(C1, C2), bottle2(C3, C4), bottle3(C5, C6), result(A, S)) to describe the state at the situation $S' = result(A, S)$

3 Successor State Axiom

The gist of the successor state axiom is that for every possible action A, we consider all possible pairs of bottles where a pour is possible and accordingly describe the resulting state from performing $A = \mathit{pour}(i, j)$. There are namely 4 possible configurations of two bottles i and j where pouring from bottle i to bottle j is possible. X and Y here represent a color layer and e represents an empty layer. Note that X and Y may or may not be the same color:

1. $\mathit{bottle}_i(X, Y)$, $\mathit{bottle}_j(e, X)$
2. $\mathit{bottle}_i(X, Y)$, $\mathit{bottle}_j(e, e)$
3. $\mathit{bottle}_i(e, X)$, $\mathit{bottle}_j(e, X)$
4. $\mathit{bottle}_i(e, X)$, $\mathit{bottle}_j(e, e)$

Accordingly, for every pour action in the successor state axiom, you will see 4 cases with disjunctions between them. These correspond to the 4 possible configurations we've described above (and in the same order that we've described above).

3.1 Example

Say we want to pour between $\mathit{bottle}_1(e, r)$ and $\mathit{bottle}_2(e, e)$. Our action is therefore $A = \mathit{pour}(1, 2)$, so we will match with $A = \mathit{pour}(1, 2)$ in the successor state axiom. After that, we will match with the fourth case because that is what matches the configuration of our bottles. The successor state axiom then says that after performing the pour, $T_1' = e$ (unchanged), $B_1' = e$ (because this layer has been poured out now), $T_2' = e$ (unchanged), and $B_2' = B_1$ (since we have poured the bottom layer from bottle_1 to bottle_2). The resulting state is then $\mathit{state}(\mathit{bottle}_1(e, e), \mathit{bottle}_2(e, r), \mathit{bottle}_3(\_, \_))$ in the situation $\mathit{result}(A, S)$. bottle_3 is represented here with underscores to signify that it remains unchanged in this case.

The Axiom

Predicates:

- $\text{Top}(\mathit{Bottle}_k, T_k, S)$: Bottle Bottle_k has top layer T_k in situation S.
- $\text{Bottom}(\mathit{Bottle}_k, B_k, S)$: Bottle Bottle_k has bottom layer B_k in situation S.
- $\text{Pour}(i, j)$: Action of pouring from bottle i to bottle j.
- $\text{Result}(A, S)$: The situation resulting from applying action A in situation S.

The full successor state axiom is as follows:

$$\Big(\text{Top}(\text{Bottle}_1, T_1, S) \wedge \text{Bottom}(\text{Bottle}_1, B_1, S) \wedge \text{Top}(\text{Bottle}_2, T_2, S) \wedge \text{Bottom}(\text{Bottle}_2, B_2, S) \wedge \text{Top}(\text{Bottle}_3, T_3, S) \wedge \text{Bottom}(\text{Bottle}_3, B_3, S)\Big)$$

$$\iff \tag{1}$$

$$
\begin{aligned}
&\Big[\Big(\mathrm{Top}(\mathrm{Bottle}_1, T_1', \mathrm{Result}(A,S)) \wedge \mathrm{Bottom}(\mathrm{Bottle}_1, B_1', \mathrm{Result}(A,S)) \wedge \\
&\mathrm{Top}(\mathrm{Bottle}_2, T_2', \mathrm{Result}(A,S)) \wedge \mathrm{Bottom}(\mathrm{Bottle}_2, B_2', \mathrm{Result}(A,S)) \wedge \\
&\mathrm{Top}(\mathrm{Bottle}_3, T_3', \mathrm{Result}(A,S)) \wedge \mathrm{Bottom}(\mathrm{Bottle}_3, B_3', \mathrm{Result}(A,S))\Big) \\
&\wedge
\end{aligned}
$$

$$
\begin{aligned}
\Big(A = \mathrm{Pour}(1,2) \Rightarrow \Big(\Big(T_1 \neq e \wedge B_1 \neq e \wedge T_2 = e \wedge T_1 = B_2\Big) \Rightarrow \Big(T_1 = e \wedge B_1' = B_1 \wedge T_2' = T_1 \wedge B_2' = B_2\Big)\Big) \vee \\
\Big(\Big(T_1 \neq e \wedge B_1 \neq e \wedge T_2 = e \wedge B_2 = e\Big) \Rightarrow \Big(T_1' = e \wedge B_1' = B_1 \wedge T_2' = e \wedge B_2' = T_1\Big)\Big) \vee \\
\Big(\Big(T_1 = e \wedge B_1 \neq e \wedge T_2 = e \wedge B_2 = B_1\Big) \Rightarrow \Big(T_1' = e \wedge B_1' = e \wedge T_2' = B_1 \wedge B_2' = B_2\Big)\Big) \vee \\
\Big(\Big(T_1 = e \wedge B_1 \neq e \wedge T_2 = e \wedge B_2 = e\Big) \Rightarrow \Big(T_1' = e \wedge B_1' = e \wedge T_2' = e \wedge B_2' = B_1\Big)\Big)\Big)\Big) \vee \\
\Big(A = \mathrm{Pour}(1,3) \Rightarrow \Big(\Big(T_1 \neq e \wedge B_1 \neq e \wedge T_3 = e \wedge B_3 = T_1\Big) \Rightarrow \Big(T_1 = e \wedge B_1' = B_1 \wedge T_3' = T_1 \wedge B_3' = B_3\Big)\Big) \vee \\
\Big(\Big(T_1 \neq e \wedge B_1 \neq e \wedge T_3 = e \wedge B_3 = e\Big) \Rightarrow \Big(T_1' = e \wedge B_1' = B_1 \wedge T_3' = e \wedge B_3' = T_1\Big)\Big) \vee \\
\Big(\Big(T_1 = e \wedge B_1 \neq e \wedge T_3 = e \wedge B_3 = B_1\Big) \Rightarrow \Big(T_1' = e \wedge B_1' = e \wedge T_3' = B_1 \wedge B_3' = B_3\Big)\Big) \vee \\
\Big(\Big(T_1 = e \wedge B_1 \neq e \wedge T_3 = e \wedge B_3 = e\Big) \Rightarrow \Big(T_1' = e \wedge B_1' = e \wedge T_3' = e \wedge B_3' = B_1\Big)\Big)\Big)\Big) \vee \\
\Big(A = \mathrm{Pour}(2,1) \Rightarrow \Big(\Big(T_2 \neq e \wedge B_2 \neq e \wedge T_1 = e \wedge B_1 = T_2\Big) \Rightarrow \Big(T_2 = e \wedge B_2' = B_2 \wedge T_1' = T_2 \wedge B_1' = B_1\Big)\Big) \vee \\
\Big(\Big(T_2 \neq e \wedge B_2 \neq e \wedge T_1 = e \wedge B_1 = e\Big) \Rightarrow \Big(T_2' = e \wedge B_2' = B_2 \wedge T_1' = e \wedge B_1' = T_2\Big)\Big) \vee \\
\Big(\Big(T_2 = e \wedge B_2 \neq e \wedge T_1 = e \wedge B_1 = B_2\Big) \Rightarrow \Big(T_2' = e \wedge B_2' = e \wedge T_1' = B_2 \wedge B_1' = B_1\Big)\Big) \vee \\
\Big(\Big(T_2 = e \wedge B_2 \neq e \wedge T_1 = e \wedge B_1 = e\Big) \Rightarrow \Big(T_2' = e \wedge B_2' = e \wedge T_1' = e \wedge B_1' = B_2\Big)\Big)\Big)\Big) \vee \\
\Big(A = \mathrm{Pour}(2,3) \Rightarrow \Big(\Big(T_2 \neq e \wedge B_2 \neq e \wedge T_3 = e \wedge B_3 = T_2\Big) \Rightarrow \Big(T_2 = e \wedge B_2' = B_2 \wedge T_3' = T_2 \wedge B_3' = B_3\Big)\Big) \vee \\
\Big(\Big(T_2 \neq e \wedge B_2 \neq e \wedge T_3 = e \wedge B_3 = e\Big) \Rightarrow \Big(T_2' = e \wedge B_2' = B_2 \wedge T_3' = e \wedge B_3' = T_2\Big)\Big) \vee \\
\Big(\Big(T_2 = e \wedge B_2 \neq e \wedge T_3 = e \wedge B_3 = B_2\Big) \Rightarrow \Big(T_2' = e \wedge B_2' = e \wedge T_3' = B_2 \wedge B_3' = B_3\Big)\Big) \vee \\
\Big(\Big(T_2 = e \wedge B_2 \neq e \wedge T_3 = e \wedge B_3 = e\Big) \Rightarrow \Big(T_2' = e \wedge B_2' = e \wedge T_3' = e \wedge B_3' = B_2\Big)\Big)\Big)\Big) \vee \\
\Big(A = \mathrm{Pour}(3,1) \Rightarrow \Big(\Big(T_3 \neq e \wedge B_3 \neq e \wedge T_1 = e \wedge B_1 = T_3\Big) \Rightarrow \Big(T_3 = e \wedge B_3' = B_3 \wedge T_1' = T_3 \wedge B_1' = B_1\Big)\Big) \vee \\
\Big(\Big(T_3 \neq e \wedge B_3 \neq e \wedge T_1 = e \wedge B_1 = e\Big) \Rightarrow \Big(T_3' = e \wedge B_3' = B_3 \wedge T_1' = e \wedge B_1' = T_3\Big)\Big) \vee \\
\Big(\Big(T_3 = e \wedge B_3 \neq e \wedge T_1 = e \wedge B_1 = B_3\Big) \Rightarrow \Big(T_3' = e \wedge B_3' = e \wedge T_1' = B_3 \wedge B_1' = B_1\Big)\Big) \vee \\
\Big(\Big(T_3 = e \wedge B_3 \neq e \wedge T_1 = e \wedge B_1 = e\Big) \Rightarrow \Big(T_3' = e \wedge B_3' = e \wedge T_1' = e \wedge B_1' = B_3\Big)\Big)\Big)\Big) \vee \\
\Big(A = \mathrm{Pour}(3,2) \Rightarrow \Big(\Big(T_3 \neq e \wedge B_3 \neq e \wedge T_2 = e \wedge B_2 = T_3\Big) \Rightarrow \Big(T_3' = e \wedge B_3' = B_3 \wedge T_2' = T_3 \wedge B_2' = B_2\Big)\Big) \vee \\
\Big(\Big(T_3 \neq e \wedge B_3 \neq e \wedge T_2 = e \wedge B_2 = e\Big) \Rightarrow \Big(T_3' = e \wedge B_3' = B_3 \wedge T_2' = e \wedge B_2' = T_3\Big)\Big) \vee \\
\Big(\Big(T_3 = e \wedge B_3 \neq e \wedge T_2 = e \wedge B_2 = B_3\Big) \Rightarrow \Big(T_3' = e \wedge B_3' = e \wedge T_2' = B_3 \wedge B_2' = B_2\Big)\Big) \vee \\
\Big(\Big(T_3 = e \wedge B_3 \neq e \wedge T_2 = e \wedge B_2 = e\Big) \Rightarrow \Big(T_3' = e \wedge B_3' = e \wedge T_2' = e \wedge B_2' = B_3\Big)\Big)\Big)\Big)\Big]
\end{aligned}
$$

$$\vee$$

$$\Big[\Big(A = \text{Pour}(1,2) \Rightarrow \Big(\Big(T_1 = e \wedge B_1 = e\Big) \Rightarrow \Big(T_1' = T_1 \wedge B_1' = B_1 \wedge T_2' = T_2 \wedge B_2' = B_2\Big)\Big) \vee$$
$$\Big(\Big(T_2 \neq e \wedge B_2 \neq e\Big) \Rightarrow \Big(T_1' = T_1 \wedge B_1' = B_1 \wedge T_2' = T_2 \wedge B_2' = B_2\Big)\Big) \vee$$
$$\Big(\Big(T_1 \neq e \wedge B_1 \neq e \wedge T_2 = e \wedge T_1 \neq B_2\Big) \Rightarrow \Big(T_1' = T_1 \wedge B_1' = B_1 \wedge T_2' = T_2 \wedge B_2' = B_2\Big)\Big) \vee$$
$$\Big(\Big(T_1 = e \wedge B_1 \neq e \wedge T_2 = e \wedge T_1 \neq B_2\Big) \Rightarrow \Big(T_1' = T_1 \wedge B_1' = B_1 \wedge T_2' = T_2 \wedge B_2' = B_2\Big)\Big)\Big)\Big) \vee$$
$$\Big(A = \text{Pour}(1,3) \Rightarrow \Big(\Big(T_1 = e \wedge B_1 = e\Big) \Rightarrow \Big(T_1' = T_1 \wedge B_1' = B_1 \wedge T_3' = T_3 \wedge B_3' = B_3\Big)\Big) \vee$$
$$\Big(\Big(T_3 \neq e \wedge B_3 \neq e\Big) \Rightarrow \Big(T_1' = T_1 \wedge B_1' = B_1 \wedge T_3' = T_3 \wedge B_3' = B_3\Big)\Big) \vee$$
$$\Big(\Big(T_1 \neq e \wedge B_1 \neq e \wedge T_3 = e \wedge T_1 \neq B_3\Big) \Rightarrow \Big(T_1' = T_1 \wedge B_1' = B_1 \wedge T_3' = T_3 \wedge B_3' = B_3\Big)\Big) \vee$$
$$\Big(\Big(T_1 = e \wedge B_1 \neq e \wedge T_3 = e \wedge T_1 \neq B_3\Big) \Rightarrow \Big(T_1' = T_1 \wedge B_1' = B_1 \wedge T_3' = T_3 \wedge B_3' = B_3\Big)\Big)\Big)\Big) \vee$$
$$\Big(A = \text{Pour}(2,1) \Rightarrow \Big(\Big(T_2 = e \wedge B_2 = e\Big) \Rightarrow \Big(T_2' = T_2 \wedge B_2' = B_2 \wedge T_1' = T_1 \wedge B_1' = B_1\Big)\Big) \vee$$
$$\Big(\Big(T_1 \neq e \wedge B_1 \neq e\Big) \Rightarrow \Big(T_2' = T_2 \wedge B_2' = B_2 \wedge T_1' = T_1 \wedge B_1' = B_1\Big)\Big) \vee$$
$$\Big(\Big(T_2 = e \wedge B_2 \neq e \wedge T_1 = e \wedge B_2 \neq B_1\Big) \Rightarrow \Big(T_2' = T_2 \wedge B_2' = B_2 \wedge T_1' = T_1 \wedge B_1' = B_1\Big)\Big) \vee$$
$$\Big(\Big(T_2 \neq e \wedge B_2 \neq e \wedge T_1 = e \wedge T_2 \neq B_1\Big) \Rightarrow \Big(T_2' = T_2 \wedge B_2' = B_2 \wedge T_1' = T_1 \wedge B_1' = B_1\Big)\Big)\Big)\Big) \vee$$
$$\Big(A = \text{Pour}(2,3) \Rightarrow \Big(\Big(T_2 = e \wedge B_2 = e\Big) \Rightarrow \Big(T_2' = T_2 \wedge B_2' = B_2 \wedge T_3' = T_3 \wedge B_3' = B_3\Big)\Big) \vee$$
$$\Big(\Big(T_3 \neq e \wedge B_3 \neq e\Big) \Rightarrow \Big(T_2' = T_2 \wedge B_2' = B_2 \wedge T_3' = T_3 \wedge B_3' = B_3\Big)\Big) \vee$$
$$\Big(\Big(T_2 = e \wedge B_2 \neq e \wedge T_3 = e \wedge B_2 \neq B_3\Big) \Rightarrow \Big(T_2' = T_2 \wedge B_2' = B_2 \wedge T_3' = T_3 \wedge B_3' = B_3\Big)\Big) \vee$$
$$\Big(\Big(T_2 \neq e \wedge B_2 \neq e \wedge T_3 = e \wedge T_2 \neq B_3\Big) \Rightarrow \Big(T_2' = T_2 \wedge B_2' = B_2 \wedge T_3' = T_3 \wedge B_3' = B_3\Big)\Big)\Big)\Big) \vee$$
$$\Big(A = \text{Pour}(3,1) \Rightarrow \Big(\Big(T_3 = e \wedge B_3 = e\Big) \Rightarrow \Big(T_3' = T_3 \wedge B_3' = B_3 \wedge T_1' = T_1 \wedge B_1' = B_1\Big)\Big) \vee$$
$$\Big(\Big(T_1 \neq e \wedge B_1 \neq e\Big) \Rightarrow \Big(T_3' = T_3 \wedge B_3' = B_3 \wedge T_1' = T_1 \wedge B_1' = B_1\Big)\Big) \vee$$
$$\Big(\Big(T_3 = e \wedge B_3 \neq e \wedge T_1 = e \wedge B_3 \neq B_1\Big) \Rightarrow \Big(T_3' = T_3 \wedge B_3' = B_3 \wedge T_1' = T_1 \wedge B_1' = B_1\Big)\Big) \vee$$
$$\Big(\Big(T_3 \neq e \wedge B_3 \neq e \wedge T_1 = e \wedge T_3 \neq B_1\Big) \Rightarrow \Big(T_3' = T_3 \wedge B_3' = B_3 \wedge T_1' = T_1 \wedge B_1' = B_1\Big)\Big)\Big)\Big) \vee$$
$$\Big(A = \text{Pour}(3,2) \Rightarrow \Big(\Big(T_3 = e \wedge B_3 = e\Big) \Rightarrow \Big(T_3' = T_3 \wedge B_3' = B_3 \wedge T_2' = T_2 \wedge B_2' = B_2\Big)\Big) \vee$$
$$\Big(\Big(T_2 \neq e \wedge B_2 \neq e\Big) \Rightarrow \Big(T_3' = T_3 \wedge B_3' = B_3 \wedge T_2' = T_2 \wedge B_2' = B_2\Big)\Big) \vee$$
$$\Big(\Big(T_3 = e \wedge B_3 \neq e \wedge T_2 = e \wedge B_3 \neq B_2\Big) \Rightarrow \Big(T_3' = T_3 \wedge B_3' = B_3 \wedge T_2' = T_2 \wedge B_2' = B_2\Big)\Big) \vee$$
$$\Big(\Big(T_3 \neq e \wedge B_3 \neq e \wedge T_2 = e \wedge T_3 \neq B_2\Big) \Rightarrow \Big(T_3' = T_3 \wedge B_3' = B_3 \wedge T_2' = T_2 \wedge B_2' = B_2\Big)\Big)\Big)\Big)\Big]$$

The first part of the axiom is the effect axiom that describes *how* the state changes when the corresponding action is performed. The second part of the axiom is the frame axiom that describes when the state does not change. We've discussed the logic of the effect axiom above. For the frame axiom, the state does not change when the pour action is unsuccessful. This happens in 4 cases:

1. $bottle_i(\_, \_,)$, $bottle_j(X, Y)$
2. $bottle_i(e, e)$, $bottle_j(\_, \_)$
3. $bottle_i(e, X)$, $bottle_j(e, Z_1)$
4. $bottle_i(X, Y)$, $bottle_j(e, Z_2)$

$(Z_1 \neq X, Z_2 \neq X)$

In other words: Either the source bottle is empty, the destination bottle is full, or the layer we are trying to pour from $bottle_i$ is a different color from the layer it is being poured onto in $bottle_j$. The 4 cases associated with every $pour(i, j)$ action in the frame axiom describe this similar to what we've done for the effect axiom above.

Note that we have omitted the frame axiom part of the successor state axiom from our Prolog program since it is not necessary to write.

3.1.1 A Small Note on Notation

In writing the successor state axiom, we've represented inequality with the $\neq$ symbol. However, this is not syntactically correct in first order logic. To represent inequality in first-order logic, we should write something of the form $\neg$(X = Y). We've chosen to write it this way (even though it is syntactically incorrect) to both make it easier to read and to save some space, since the successor state axiom we've written is quite large. We hope you won't find this objectionable.

In addition to that, we hope you forgive any possible typos we've made along the way, since there was a lot to write. Notwithstanding possible typos, we hope you've at least understood the logic we were trying to implement from reading the paragraph on the gist of the axiom above.

4 Searching with the Successor State Axiom

After we have written our successor state axiom, we can make our agent use it to search in Prolog. Here is a predicate in Prolog `goal(S)` that does just that:

```
goal(S) :-

      state(B1, B2, B3, S),
      is_goal_state(state(B1, B2, B3, S))
      ,!
      .
```

All this does is generate states and test whether they are goal states or not. If they are, the situation associated with this state is returned as the goal. The search runs in an iterative deepening manner, since the agent will first try all possible pour actions on the given situation, and if that fails, will backtrack and start searching from $s0$ again, and so on. This may be somewhat difficult to infer just from the written predicate, but if you run the Prolog program and use the debugger to analyze the search it should become more apparent.

Since we are essentially searching using IDS, we expect it to be complete. We also expect it to be optimal because the path cost is a monotonically increasing function of the depth, since the resulting situations only get longer as we perform more pour actions and go deeper in the search tree.

4.1 Addressing Infinite Searching

We still have one other problem to address: When does the search stop? When there is no solution, or when the agent is asked to verify if a given situation corresponds to a goal state, and it in fact does not, the search above will run forever.

4.1.1 Some Solutions That Were Tried

One idea was to implement a depth limit. We have calculated a suitable depth limit to be 126 (each bottle has 5 possible valid configurations, therefore the number of possible states for the problem is $5 \cdot 5 \cdot 5 = 125 + 1$).

With the way we have implemented our successor state axiom, this will work for finding solutions. However, this will not work for verifying given situations, because there can be a situation where many repeated states are visited in the middle yet the situation does correspond to a goal state (think of a simple example such as pouring between two bottles back and forth over 126 times, and then making one or two pours that land us in a goal state). If we use a search limit of 126 (or any other limit, for that matter), a situation long enough is bound to make the search fail, even though that situation may correspond to a goal state (this is actually only a problem when there are two layers and they are of the same color).

4.1.2 Our Implemented Solution

To counteract this, we've decided to simply stop the search from starting if the given initial state cannot possibly lead to a goal state. This also has the additional bonus of reducing the number of inferences our program makes since it will not be searching when the search is bound to fail.

Because our problem is relatively simple, this was not too hard to do. First, we made the observation that if the number of non-empty layers in the initial state is odd (regardless of what color(s) those layers are), then this could never lead to a goal state, since there must be at least 1 bottle that is neither completely empty nor completely full. Therefore, we immediately exclude any initial state where the number of non-empty layers is odd. (i.e. 1, 3, or 5 layers).

What about when the number of non-empty layers is even (i.e 0, 2, 4, or 6 layers)? If there is an even number of non-empty layers, then we can certainly have every bottle be either completely full or completely empty. But can we have it be full of the same color?

Since each bottle has a capacity of 2, we need the number of layers of every color we have to be a multiple of 2 (or to be 0) in order to fill some number of bottles. In other words, we need it to be an even number.

Therefore, in case the number of non-empty layers is even, we simply count the number of layers of each color. If the number of layers of every color is even, then this situation can lead to a goal state. Otherwise, it cannot.

One last remark is that this counting approach will not work if the number of non-empty layers is 6, i.e all the bottles are initially completely full, because even though the number of layers of each color may be even, it could be the case that some bottles are filled with one layer of each color, making this not a goal state. As a result of this, the first check we do is to check if there are 6 non-empty layers, and if so, if the state is a goal state.

Here are the predicates we've written in Prolog to do this:

```
can_lead_to_solution(state(bottle1(Top1, Bottom1), bottle2(Top2, Bottom2), bottle3(Top3, Bottom3), _)) :-
    is_goal_state(state(bottle1(Top1, Bottom1), bottle2(Top2, Bottom2), bottle3(Top3, Bottom3), _)).


can_lead_to_solution(state(bottle1(Top1, Bottom1), bottle2(Top2, Bottom2), bottle3(Top3, Bottom3), _)) :-
    Layers = [Top1, Bottom1, Top2, Bottom2, Top3, Bottom3],
    unique_colors(Layers, UniqueColors),
  (
    (
        length(UniqueColors, 1),
        [Color] = UniqueColors,
        nonvar(Color),
        count_occurrences(Color, Layers, Count),
        0 is Count mod 2
    )
    ;
    (
        length(UniqueColors, 2),
        [Color1, Color2] = UniqueColors,
        nonvar(Color1), nonvar(Color2),
        count_occurrences(Color1, Layers, Count1),
        count_occurrences(Color2, Layers, Count2),
        Count1 + Count2 < 6,
        0 is Count1 mod 2,
        0 is Count2 mod 2
    )
  ).
```

`count_occurunces/2` and `unique_colors/2` are two helper predicates that count the number of occurrences of a given element in a given list and extract unique elements from a given list to another list respectively:

```
count_occurrences(_, [], 0).
count_occurrences(X, [X|T], N) :- count_occurrences(X, T, N1), N is N1 + 1.
count_occurrences(X, [Y|T], N) :- X \= Y, count_occurrences(X, T, N).

unique_colors([], []).
unique_colors([e|T], T1) :- unique_colors(T, T1).
unique_colors([H|T], [H|T1]) :- H \= e, delete(T, H, T2), unique_colors(T2, T1).
```

We modify our `goal(S)` predicate from earlier to implement these changes:

```
goal(S) :-
    (
      state(B1I, B2I, B3I, s0),
      can_lead_to_solution(state(B1I, B2I, B3I, s0))
    )
    ->
    (
      state(B1, B2, B3, S),
      is_goal_state(state(B1, B2, B3, S))
    )
    ,!
    .
```

And finally, the only predicate we haven't mentioned yet, `is_goal_state`:

```
is_goal_state(state(bottle1(C1, C1), bottle2(C2, C2), bottle3(C3, C3), _)).
```

5 Test Cases

In this section we will look at some test cases and their outputs as well as some performance measures. We've used the predefined `time/1` predicate in SWI-Prolog (`https://www.swi-prolog.org/pldoc/man?predicate=time/1` to obtain the performance measures.

5.1 Test Case 1

```
bottle1(b,r).
bottle2(b,r).
bottle3(e,e).
```

Query: `goal(S).`
Result: `S = result(pour(1, 2), result(pour(2, 3), result(pour(1, 3), s0))) .`
`251 inferences, 0.000 CPU in 0.000 seconds`

Query: `goal(result(pour(1, 2), result(pour(1, 3), result(pour(2, 3), s0)))).`
Result: `true.`
`74 inferences, 0.000 CPU in 0.000 seconds`

Query: `goal(result(pour(2, 1), result(pour(1, 3), result(pour(2, 1), result(pour(1, 3), s0))))).`
Result: `false.`
`90 inferences, 0.000 CPU in 0.000 seconds`

5.2 Test Case 2

```
bottle1(e,r).
bottle2(b,r).
bottle3(e,b).
```

Query: `goal(S).`
Result: `S = result(pour(1, 2), result(pour(2, 3), s0)) .`
`132 inferences, 0.000 CPU in 0.000 seconds`

Query: `goal(result(pour(3, 2), result(pour(3, 2), result(pour(2, 1), result(pour(2, 3), s0))))).`
Result: `true.`
`86 inferences, 0.000 CPU in 0.000 seconds`

Query: `goal(result(pour(3, 1), result(pour(2, 3), s0))).`
Result: `false.`
`90 inferences, 0.000 CPU in 0.000 seconds`

5.3 Test Case 3

```
bottle1(e,g).
bottle2(e,g).
bottle3(e,e).
```

Query: `goal(S).`
Result: `S = result(pour(1, 2), s0) .`
`52 inferences, 0.000 CPU in 0.000 seconds`

Query: `goal(result(pour(1, 3), result(pour(2, 3), s0))).`
Result: `true.`
`55 inferences, 0.000 CPU in 0.000 seconds`

Query: `goal(s0).`
Result: `false.`
`53 inferences, 0.000 CPU in 0.000 seconds`

5.4 Test Case 4

```
bottle1(e,g).
bottle2(e,g).
bottle3(g,r).
```

Query: `goal(S).`
Result: `false.`
`64 inferences, 0.000 CPU in 0.000 seconds`

Query: `goal(result(pour(1, 2), result(pour(1, 2), result(pour(2, 1), s0)))).`
Result: `false.`
`64 inferences, 0.000 CPU in 0.000 seconds`

Query: `goal(s0).`
Result: `false.`
`64 inferences, 0.000 CPU in 0.000 seconds`

5.5 Test Case 5

```
bottle1(y,b).
bottle2(y,b).
bottle3(b,y).
```

Query: `goal(S).`
Result: `false.`
`69 inferences, 0.000 CPU in 0.000 seconds`

Query: `goal(result(pour(1, 2), result(pour(2, 3), result(pour(1, 2), result(pour(3, 2), s0))))).`
Result: `false.`
`69 inferences, 0.000 CPU in 0.000 seconds`

Query: `goal(result(pour(2, 3), result(pour(2, 3), result(pour(3, 1), result(pour(3, 1), result(pour(1, 2), result(pour(1, 3), result(pour(3, 2), s0)))))))).`
Result: `false.`
`69 inferences, 0.000 CPU in 0.000 seconds`

5.6 Test Case 6

```
bottle1(r,r).
bottle2(b,b).
bottle3(r,r).
```

Query: `goal(S).`
Result: `S = s0.`

`9 inferences, 0.000 CPU in 0.000 seconds`

Query: `goal(s0).`
Result: `true.`
`9 inferences, 0.000 CPU in 0.000 seconds`

Query: `goal(result(pour(2, 1), result(pour(3, 2), result(pour(1, 3), result(pour(3, 2), s0))))).`
Result: `false.`
`79 inferences, 0.000 CPU in 0.000 seconds`

5.7 Test Case 7

```
bottle1(e,e).
bottle2(e,e).
bottle3(e,e).
```

Query: `goal(S).`
Result: `S = s0.`

```
11 inferences, 0.000 CPU in 0.000 seconds
```

Query: `goal(s0).`
Result: `true.`
`10 inferences, 0.000 CPU in 0.000 seconds`

Query: `goal(result(pour(1, 3), result(pour(3, 2), result(pour(2, 1), result(pour(2, 3), s0))))).`
Result: `false.`
`72 inferences, 0.000 CPU in 0.000 seconds`

5.8 Test Case 8

```
bottle1(e,e).
bottle2(r,r).
bottle3(e,e).
```

Query: `goal(S).`
Result: `S = s0.`

```
41 inferences, 0.000 CPU in 0.000 seconds
```

Query: `goal(result(pour(2, 3), result(pour(2, 3), result(pour(1, 2), result(pour(2, 1), result(pour(3, 2), result(pour(1, 3), result(pour(2, 1), s0)))))))).`
Result: `true.`
`81 inferences, 0.000 CPU in 0.000 seconds`

Query: `goal(result(pour(1, 2), result(pour(3, 1), result(pour(1, 3), result(pour(1, 2), s0))))).`
Result: `false.`
`22 inferences, 0.000 CPU in 0.000 seconds`

6 Finding additional solutions

In our program we've added a cut (`!`) predicate to the `goal(S)` predicate in order to disallow Prolog from backtracking to find additional solutions. We did this because the first solution returned is the best one (as discussed earlier), and backtracking will only produce worse solutions. Nonetheless, if you would like to have the Prolog backtrack, you can remove the cut predicate and the agent will backtrack and find solutions, but with one additional problem:

Since we've defined initial states that are a goal state to be ones that can lead to solutions, with the way our goal predicate is currently written, this will allow Prolog to start a search from such initial states where no additional solutions can be found (this happens when all bottles are full). This search will obviously run forever since there is no other possible solution. There is one (at least without making changes to the logic of our existing program) easy solution for this, which is to simply add another `goal(S)` predicate on top of the existing `goal(S)` predicate to handle just this case, and disallow this predicate from backtracking:

```
goal(S) :-
    state(bottle1(Top1, Bottom1), bottle2(Top2, Bottom2), bottle3(Top3, Bottom3), S),
    Top1 \= e, Bottom1 \= e, Top2 \= e, Bottom2 \= e, Top3 \= e, Bottom3 \= e,
    is_goal_state(state(bottle1(Top1, Bottom1), bottle2(Top2, Bottom2), bottle3(Top3, Bottom3), S)),
    !.
```

This will slightly increase the number of inferences the program makes, so we've chosen to omit it since it is not necessary as long as we don't want to allow backtracking.