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

Project Report

Abdelrahman Aboelkehir, Hossain Ghoraba,
Mahmoud Aboelenein, Omar Hesham

26.11.2024

## 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. Additionally, only one layer can be poured at a time.

## 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 `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 $i$ and $j$ 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 $\mathit{bottle}_1$ to $\mathit{bottle}_2$). The resulting state is then $\mathit{state}(\mathit{bottle}_1(e, e), \mathit{bottle}_2(e, r), \mathit{bottle}_3(\_, \_))$ in the situation $\mathit{result}(A, S)$. $\mathit{bottle}_3$ is represented here with underscores to signify that it remains unchanged in this case.

### 3.2 The Axiom

**Predicates:**

- $\text{Top}(\mathit{Bottle}_k, T_k, S)$: Bottle $\mathit{Bottle}_k$ has top layer $T_k$ in situation $S$.
- $\text{Bottom}(\mathit{Bottle}_k, B_k, S)$: Bottle $\mathit{Bottle}_k$ 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:

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

$$\iff \tag{1}$$

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

$$\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}$$

$$\vee$$

$$
\begin{aligned}
\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]
\end{aligned}
$$

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.**

### 3.3 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 ¬(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 paragraphs explaining the logic of the successor state 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/1` 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 only action is $pour(i,j)$, which means that all actions have the same cost.

## 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 One Attempted Solution: Setting a Depth Limit

One idea was to implement a depth limit. We have calculated a suitable depth limit to be **4**. This is because the ”hardest” initial configuration of the problem is one with 4 layers. (If there are 2 layers, we can reach a goal configuration in at most 1 pour. If there are 6 layers, there are no pours to do, and either the initial state is a goal state, or it is not). If there are 1, 3, or 5 layers, we can never reach a goal state no matter what (we explain why this is the case in section 4.1.3). Notice also that for a configuration of 4 layers to lead to a solution, the 4 layers must be either all of the same color, or 2 of one color and 2 of another (again, we also explain why this must be the case in section 4.1.3.)

The case for 4 layers when they are all of the same color is easy: at most one pour is needed to reach a goal state. With 2 layers of each color, we can show that we need at most 4 pours to reach a goal state.

Without going into any calculations, there is an easy way to show this: For there to be 4 layers, the layers can either be placed in a 2 + 2 configuration (2 layers in one bottle, 2 layers in another bottle, and the remaining bottle empty) or a 2 + 1 + 1 configuration (2 layers in one bottle, 1 layer in each of the remaining bottles). In the first configuration, since there is an empty bottle, any of the top layers of the 2 filled bottles can be poured into it. In the second configuration, if the bottle that is completely filled is filled with a layer of the same color, then the two remaining half full bottles must be of the same color too, and one of them can be poured on the other. Otherwise, if the bottle that is completely full is filled with one layer of each color, then one of the two remaining half full bottles has the other layer matching the color of the top layer of the full bottle, in which case it can be poured onto it. After we do the first pour, we essentially have the same problem again: 4 layers in some configuration, and by the same reasoning above, there is still a valid pour to do.

Since there are 4 layers, we only need to do this at most 4 times to arrive at a goal state. This is because there is no reason to pour any one layer more than once. If we cannot do this in 4 moves, it means that there cannot be a valid solution.

### 4.1.2 Shortcoming of a Depth Limit

Setting a depth limit will work for finding a solution when there is one, and returning false when there isn’t one. It does, however, have one shortcoming: verifying solutions. This is because the agent can be asked to verify a situation where there are more than 4 pours (or more generally, a situation where there are more pours than our search limit allows) that does eventually lead to a goal state (think of a configuration such as $bottle_1(e, r)$, $bottle_2(e, r)$, $bottle_3(e, e)$ where one of the red layers is poured back and forth any number of times before eventually settling on top of the other red layer, landing us in a goal state). For such situations, the search can fail not because the given situation does not lead to a goal state, but because it has reached the depth limit.

Of course, such situations are very bad solutions and serve very little purpose, but they are valid nonetheless, and our agent must accept them.

### 4.1.3 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, and any search that does start, starts without any depth limit. Since the initial state can lead to a solution, this search will never run forever and will always find some solution(s), and because there is now no depth limit, it can verify any given situation of any length.

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 (because each bottle has a capacity of 2, and odd numbers are by definition not a multiple of 2). 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 a color to be a multiple of 2 (or to be 0) in order to completely fill some number of bottles with that color. 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. count_occurunces/2 counts the number of occurrences of a given element in a given list, and unique_colors/2 extracts the unique elements from a given list to another list:

```
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/1 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.`
```
85 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.`
```
84 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)) .`
```
133 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.`
```
87 inferences, 0.000 CPU in 0.000 seconds
```

Query: `goal(result(pour(3, 1), result(pour(2, 3), s0))).`
Result: `false.`
```
78 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) .`
```
53 inferences, 0.000 CPU in 0.000 seconds
```

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

Query: `goal(s0).`
Result: `false.`
```
44 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.`

`10 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(2, 1), result(pour(3, 2), result(pour(1, 3), result(pour(3, 2), s0))))).`
Result: `false.`
`18 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.`

```
10 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.`
```
22 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.`

```
10 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.`
```
50 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/1` 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 solutions that are either as good as the first solution we find, or worse. Nonetheless, if you would like to have Prolog backtrack, you can remove the cut predicate and the agent will be able to backtrack and find additional 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 backtrack (after returning `S = s0.`) and start a search from such initial states where no additional solutions can be found (this happens when all bottles are completely full or all the bottles are completely empty). 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 modify the `goal/1` predicate to handle such cases:

```
goal(S) :-
    (
      state(B1I, B2I, B3I, s0),
      (
        all_layers_empty(state(B1I, B2I, B3I, s0)) -> S = s0
      ;
        all_layers_full(state(B1I, B2I, B3I, s0)) -> S = s0
      ;
        (
          can_lead_to_solution(state(B1I, B2I, B3I, s0)),
          \+ all_layers_empty(state(B1I, B2I, B3I, s0)),
          \+ all_layers_full(state(B1I, B2I, B3I, s0))
        )
        ->
        (
          state(B1, B2, B3, S),
          is_goal_state(state(B1, B2, B3, S))
        )
      )
    ).
```

And add the helper predicates used in the new `goal/1` predicate:

```
all_layers_empty(state(bottle1(e, e), bottle2(e, e), bottle3(e, e), _)).
all_layers_full(state(bottle1(C1, C2), bottle2(C3, C4), bottle3(C5, C6), _)) :-
    C1 \= e, C2 \= e, C3 \= e, C4 \= e, C5 \= e, C6 \= e.
```

This does not have a noticeable effect on the performance of the agent. It increases the number of inferences made by a very small number (around 10 on average), and in the case where all the layers are empty or all the layers are full, appears to decrease the number of inferences from an average of 10 to an average of around 4 or 5.

We have only chosen not to include this version of the `goal/1` predicate because it is somewhat larger and more difficult to understand than the more straightforward one from earlier that we include in our program, and as mentioned earlier, we do not need it if we are not interested in allowing backtracking (which we aren't). Nonetheless, in case you are interested in finding additional solutions, here it is.