# Problem Statement

Water Jugs problem also known as the three-glass riddle is a famous problem in Artificial Intelligence. The glass riddle dates back to 1484 (Bettilyon 2019) and a variant of this problem was featured in the plot of Die Hard 3 [2]. The problem gives us 3 bottles, 10L, 6L, and 5L respectively, with no marking on them. The objective is to fill one of the bottles with a certain amount of water by filling or emptying them without using any outside measuring tools.

# Representation of Problem

In Artificial Intelligence, problems such chess, Rubik’s cube, three-glass riddle, etc. are modeled using space state representation where all the possible states of the problem are considered and using a variety of graphs, trees and search techniques a path from the initial state to the goal state is searched for.

For our problem in hand, we have three bottles, given initial state of the bottles and goat state that we must reach. We can name the bottles b1, b2, and b3 respectively and represent their state with a list [b1, b2, b3] where b1 represents the amount of water in the 10-liter bottle, b2 represents the 6-liter bottle and b3 in the 5-liter bottle. The range of permissible water can be shown as 0 ≤ b1 ≤ b2 ≤ b3.

State representation = amount of water in each bottle

[b1, b2, b3] = [10, 0, 0]

For our problem we can represent our initial state and goal state as such-

**Scenario 1:**

Start: b1 = 10, b2 = 0, b3 = 0

End: b1 = 8, b2 = 0, b3 = 0

**Scenario 2:**

Start: b1 = 2, b2 = 0, b3 = 0

End: b1 = 4, b2 = 0, b3 = 0

**Scenario 3:**

Start: b1 = 3, b2 = 0, b3 = 0

End: b1 = 7, b2 = 0, b3 = 0

# Operators

As we cannot measure the amount of water in any bottle, we have to follow a set of rules (also known as production rules) to achieve our end goal. These are step toward the next state. We keep taking step to the next state until we reach a solution. Here is the list of variables we will need to express our problem in operation form:

b1, b2, b3:

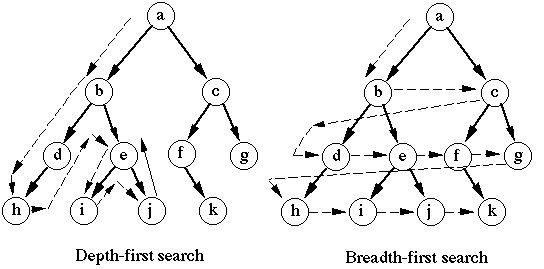
x, max capacity b1 = 10,  
y, max capacity b2 = 6,  
z, max capacity b3 = 5

Here is the list of all legal operations

|  |  |  |  |
| --- | --- | --- | --- |
| **R.No.** | **Condition** | **Final state** | **Description of action taken** |
| 1. | If b1 < x | (10, b2, b3) | Fill the 10-litre bottle completely |
| 2. | if b2 < y | (x, 6, z) | Fill the 6-litre bottle completely |
| 3. | if b3 < z | (x, y, 5) | Fill the 5-litre bottle completely |
| 4. | If b1 > 0 | (0, y, z) | Empty the 10-litre bottle completely |
| 5. | If b2 > 0 | (x, 0, z) | Empty the 6-litre bottle completely |
| 6. | If b3 > 0 | (x, y, z) | Empty the 5-litre bottle completely |
| 7 | If (b1+b2) ≥ x && b2>0 | (10, y-[10-x], z) | Pour some water from the 6-liter bottle to fill the 10-liter bottle (keeping leftover in the 6-litre bottle) |
| 8 | If (b2+b3) ≥ y && b3>0 | (x, 6, z-[6-y]) | Pour some water from the 5-liter bottle to fill the 6-liter bottle (keeping leftover in the 5-litre bottle) |
| 9 | If (b1+b3) ≥ x && b3>0 | (10, y, z-[10-x]) | Pour some water from the 5-liter bottle to fill the 10-liter bottle (keeping leftover in the 5-litre bottle) |
| 10 | If (b1+b2) ≥ y && z>0 | (x-[6-y], 6, z) | Pour some water from the 10-liter bottle to fill the 6-liter bottle (keeping leftover in the 10-litre bottle) |
| 11 | If (b1+b3) ≥ z && b1>0 | (x-[5-z], y, 5) | Pour some water from the 10-liter bottle to fill the 5-liter bottle (keeping leftover in the 10-litre bottle) |
| 12 | If (b2+b3) ≥ z && b2>0 | (x, y-[5-z], 5) | Pour some water from the 6-liter bottle to fill the 5-liter bottle (keeping leftover in the 6-litre bottle) |
| 13 | If (b1+b2) ≤ x && b2>0 | (10, 0, z) | Pour all water from the 6-liter bottle to fill the 10-liter bottle (no leftover) |
| 14 | If (b2+b3) ≤ y && b3>0 | (x, 6, 0) | Pour all water from the 5-liter bottle to fill the 6-liter (no leftover) |
| 15 | If (b1+b3) ≤ x && b3>0 | (10, y, 0) | Pour all water from the 5-liter bottle to fill the 10-liter bottle (no leftover) |
| 16 | If (b1+b2) ≤ y && b1>0 | (0, 6, z) | Pour all water from the 10-liter bottle to fill the 6-liter bottle (no leftover) |
| 17 | If (b1+b3) ≤ z && b1>0 | (0, y, 5) | Pour all water from the 10-liter bottle to fill the 5-liter bottle (no leftover) |
| 18 | If (b2+b3) ≤ z && b2>0 | (x, 0, 5) | Pour all water from the 6-liter bottle to fill the 5-liter bottle ((no leftover) |

# Search Strategies

As discussed earlier, a variety of search techniques are used to solve problems in artificial intelligence. The three-bottle riddle can be solved by modeling the problem as a non-weighted tree structure. From the initial state [10, 0, 0] we can have multiple possible next states using the production rules. By definition, we have reached a new state and we can check if those new states match with our end goal if not we can branch out from there. As each of the possible states is equally plausible that is why our tree is non-weighted.

There are multiple search algorithms that we can use to traverse through all the states(nodes) in the tree in a systematic way. Breadth-first search (BFS) and depth-first search (DFS) are two of the most used algorithms in non-weighted trees. Both algorithms fall under blind search or uninformed search algorithms, meaning the will traverse through all the nodes until they find a solution. [](https://miro.medium.com/max/1088/1*INwehwNaWrUmOvq_a5wMWg.gif)

BFS will traverse level by level until the goal state is reached, whereas DFS starts from the root node and 2ill follow each path to its greatest depth node before moving to the next path. Both algorithms guarantee a solution (if exist) but we can make a guess here that, as both of the algorithms traverse through all the nodes, space and time complexity is far from optimal. In the worst-case scenario, BFS will use a lot of space in memory to visit all the nodes (if there are many nodes in the tree) on the other hand, in DFS time complexity becomes the problem. In the worst-case scenario, it will start from a root node go to the deepest node, not find a solution and do this for other nodes as well. (Ng 2014)

The uninformed search will find the solution (if exist) but it is far from optimal. Therefore, in complex problems we use informed search algorithms, these algorithms can have an array of knowledge about the problem state such as how far we are from our goal, path cost, how to reach goal node, etc. these algorithms use the idea of heuristic, therefore these are also called Heuristic search.

The basic informed search strategies are Best-first Search also known as Greedy search and A\* Search. Greedy search expands the node which is closest to the goal node. The heuristic function for greedy search is f(n) = g(n). it is implemented using two lists open, and close. The closed node is not visited again. In the worst-case scenario, it can act like an unguided depth-first search. (javaTpoint, b)

A\* search utilizes a priority queue and based on the priority of the queued state next node to traverse is selected. Priority of the state is calculated using the heuristic f(n) = g(n) + h(n) where

g(n) = cost to reach node n from start state and h(n) = cost to reach from node n to goal node. (javaTpoint, b)

## Search 1 Method

For uninformed search implementation, the Breadth-first search is chosen to solve our problem of three bottles. As discussed, the BFS worst-case scenario is it uses more memory versus DFS whose worst-case scenario requires more time. As our case is relatively small and the number possible states are not exceedingly large, any of the algorithms would have performed well, but my preferred algorithm was BFS as I preferred to use more space than more time.

The time complexity of BFS is the number of nodes traversed until the shallowest node. Where   
d = depth of solution and b is the node at every state.

**T (b) = 1+b2+b3+.......+ bd= O (bd)**

And the space complexity is given by the Memory size which is **O (bd).** (javaTpoint, a).

## Search 2 Method

A\* algorithm was chosen as the informed search strategy. This algorithm does not always produce the shortest path as it is based on heuristic and approximation, but it is optimal and complete. The efficiency of the algorithm depends on the quality of heuristics.

The time complexity of A\* depends on the heuristic function and the number of nodes expanded is exponential to the depth of solution d.

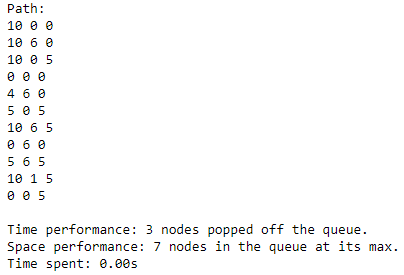
**T (b) = O(b^d),** where b is the branching factor.

And the space complexity is also **O(b^d)**

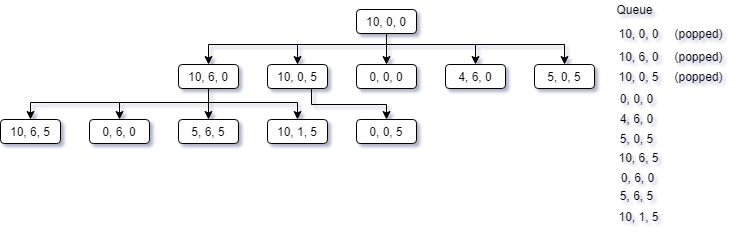
# Search Tree

## Breadth-First Search

To demonstrate the breadth first search tree, an initial position of 10,0,0 and goal position of 0, 0, 5 was set in the python script. Here is the result:



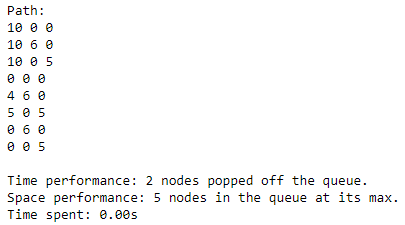
As we can see 3 nodes were popped off the queue and at its max 7 nodes were in the queue. To cross-check our result here is the tree diagram implementation.



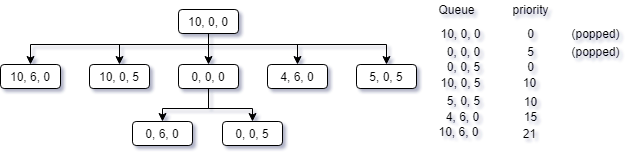
We start at the root [10,0,0] and that is currently initially at the queue. We pop off the first item and expand from left to write respectively and add all of the unvisited states to the queue. Following the order, we pop off the next state in queue [10, 6, 0] and expand all its child node and add them to the queue as well. As we didn’t arrive at our goal we continue popping off the next value [10,0,5] and expand its children where the first child is our solution and we stop our search here.

## A\* Search

Searching the goal state was faster, here is the result:



With A\* search we can see that it took 1 less node and the maximum number of nodes in queue was also less. Looking at the tree diagram can clearly explain why is so



A\* algorithm associates a priority number with each state. The priority is calculated using the heuristic function. For this case I used

**h(n) = |b1-D1| + |b2 – D2| + |b3-D3|**

Lower the h(n) value, the more priority that states get. The idea of the heuristic function is finding which state is at least distance from the goal with the constraint that when h(n) = 0 we have reached our goal.

Keeping the lowest value = highest priority in mind, at first we have [10, 0, 0] in our queue. We pop off the initial state and expand to other possible states. Getting priority of all the states, the most prioritized state is [0, 0, 0] with a priority value of 5. Therefore we pop that state off and expand it. Our result is the second child of the parent node and following the formula, at goal state h(n) is 0.

\*the heuristic function is a variation of the heuristic function discussed in stack overflow (Mok 2014).

# Code Implementation

## Uninformed Search: Breadth First

The basic structure of the code is:

1. Setting initial and goal state
2. Appending initial state to queue
3. Appending initial state to visited
4. Apply production rule and create a new state
   1. Check if the newly created state is visited
      1. If visited don’t append to queue and move to next production rule
   2. Check if the goal is reached
      1. If reached, print the path details and exit
5. Repeat step 3 until the goal is reached (4b)

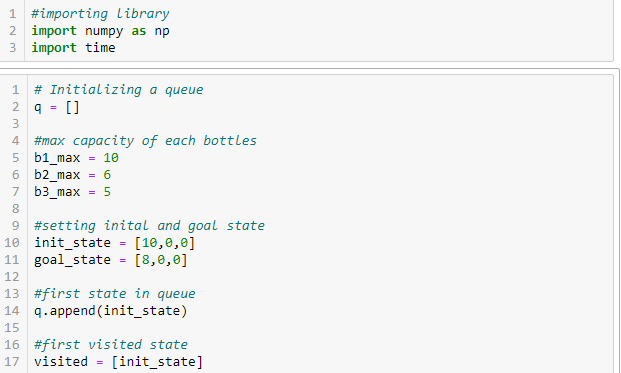


Figure : Step 1, 2, 3

Defining initial and goal state. Appending init\_state to visited list and queue list.

Next, pop the first value from the queue list and create new states. Here the “fill f1” is one of the 17 other production rules to create a new state. The whole process is enclosed in a while loop, basically the while loop checking if there are still items left to traverse through.

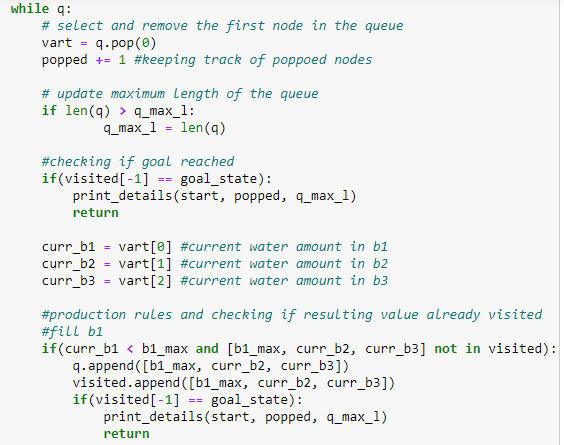


Figure : step 4, 5

Once we reach our goal, the time and space taken as well as the path from the initial state to the goal state are printed.

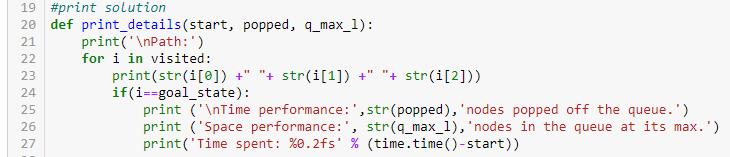


Figure : Printing result

## Informed Search: A\*

To reuse my BFS code (as much as I can), instead of importing the python queue library, the normal list is used as a priority list and using a lambda function the queue is ordered in ascending order.

As the whole code is the same, only the new part is discussed here.

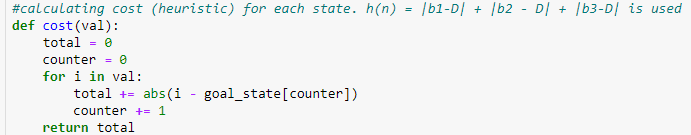
1. The list is created as a tuple, where the first value contains the state and the second value contains the total cost for that state.



2. All the available states in the pQ list is sorted in ascending order.

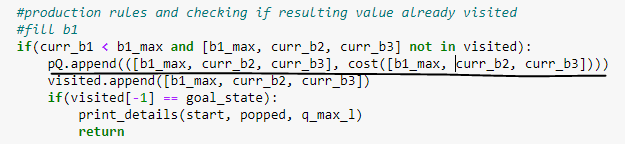


3. The cost is calculated using this function:



We are taking the absolute subtraction of b1/b2/b3 and their respective goal state value and summing their total.

4. lastly, all the production rules are the same as before, only difference is how we add new states to the priority que



Underlined code is the append() function. The basic structure is pQ.append(( [new state] , cost() ), here the cost() function is called to calculate the priority.

The rest of the code is similar to the implementation of BFS.

# Demonstration

## Uninformed Search: Breadth First

**Scenario 1:**

Start: b1 = 10, b2 = 0, b3 = 0

End: b1 = 8, b2 = 0, b3 = 0

Path:

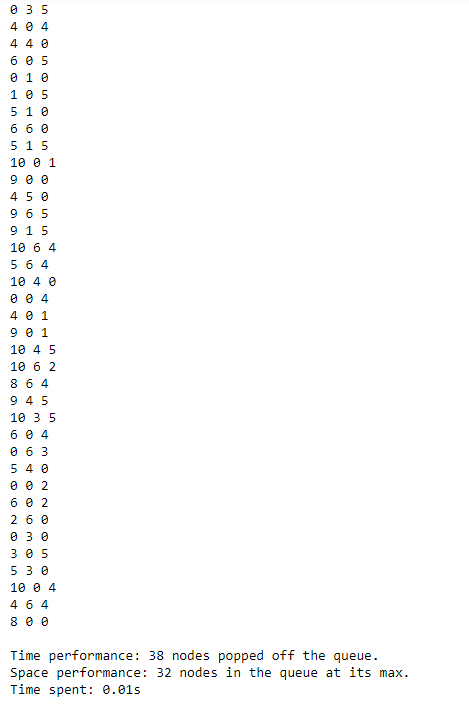
10 0 0

10 6 0

10 0 5

.. ..

.. ..

.. ..

**Scenario 3:**

Start: b1 = 3, b2 = 0, b3 = 0

End: b1 = 7, b2 = 0, b3 = 0

Path:

3 0 0

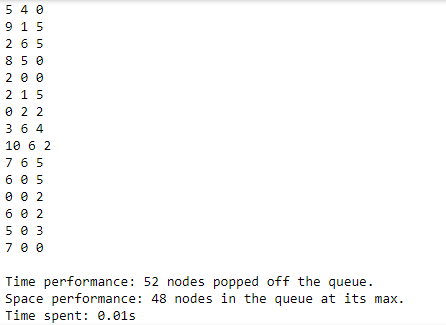
10 0 0

3 6 0

3 0 5

… .. …

.. .. .. .

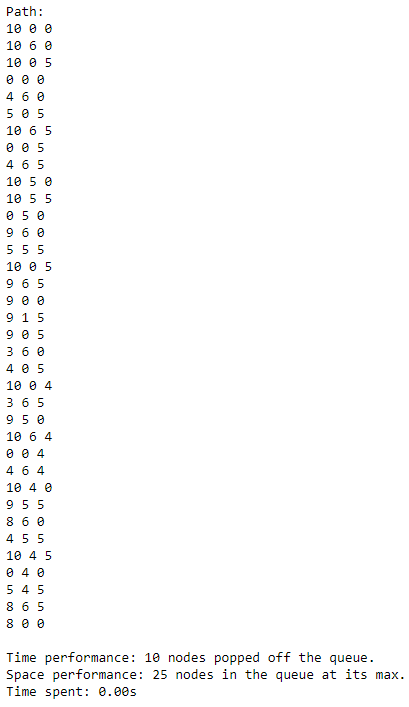
… .. ..

The results are not bad both of the scenarios were performed within 0.1 sec. But the number nodes they traverse through is nearly 3x than the informed search.

## Informed Search: A\*

**Scenario 1:**

Start: b1 = 10, b2 = 0, b3 = 0

End: b1 = 8, b2 = 0, b3 = 0

**Scenario 3:**

Start: b1 = 3, b2 = 0, b3 = 0

End: b1 = 7, b2 = 0, b3 = 0

Path:

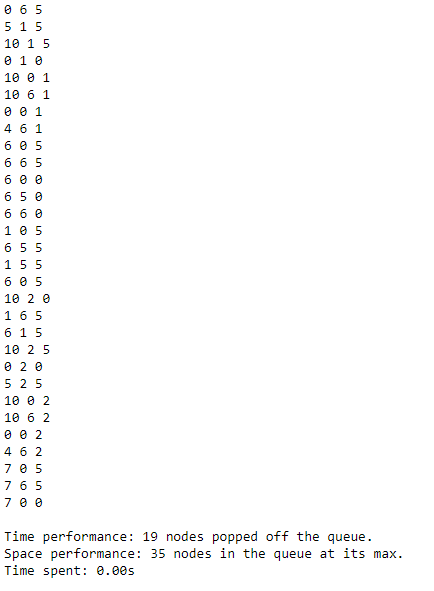
3 0 0

10 0 0

3 6 0

3 0 5

.. .. …

… … …

For both the scenarios the solution was found almost instantly and the number of nodes it had to traverse is significantly lowered. A clear improvement over uninformed search.

# Conclusion

Without a doubt, the informed search performed better than the uninformed search. From the performance matrix, we have seen the evolution of each search algorithms. As our problem’s space state was limited in number, therefore even the uninformed search was quite fast, but we can imagine a situation where the number of state space too big to traverse through all of them  
(e.g. Rubik’s cube) in such scenarios, the informed search is the way to go.

# References

1. Bettilyon, T. 2019, “Modeling Problems as Graphs”, retrieved from, <https://medium.com/tebs-lab/modeling-problems-as-graphs-6ab451f03868>, viewed on 01/05/2020
2. “Die Hard 3: Jugs Problem”, retrieved from, <https://www.youtube.com/watch?v=BVtQNK_ZUJg>, viewed on 28/04/2020
3. javaTpoint, a,“Uninromed Search Algorithms”, retrieved from, <https://www.javatpoint.com/ai-uninformed-search-algorithms>, viewed on 06/05/2020
4. NG, T. 2014, “BFS and DFS”, retrieved from, <https://medium.com/@tim_ng/bfs-and-dfs-52d3cb642a0e>, viewed on 06/05/2020
5. javaTpoint, b,“Informed Search Algorithms”, retrieved from, <https://www.javatpoint.com/ai-informed-search-algorithms>, viewed on, 06/05/2020
6. Mok, 2014, “Heuristic function for Water Jug”, retrieved from, <https://stackoverflow.com/questions/22770487/heuristic-function-for-water-jug>, viewed on 07/05/2020