American University of Armenia

College of Science and Engineering

Artificial Intelligence Project

Search Methods for Water Sort Puzzle

Team: Ani Aloyan, Seda Bayadyan, Elina Ohanjanyan, Alexander Shahramanyan

Fall, 2023

Abstract

Water Sort Puzzle is a relatively new puzzle, which came out at the beginning of 2020s. The game consists of a set of tubes, some of which are empty, while the rest are filled with layers of different-colored liquids and the goal is to sort the liquids by color. The game holds a similarity to a different game - the Ball Sort puzzle, which has fewer restrictions. In this paper, we will explore and compare a number of uninformed, informed, and local search algorithms to find an optimal solution to the puzzle.

Keywords: water sort puzzle, uninformed search, informed search, local search, BFS, DFS, A* search, random walk, hill climbing, heuristic function

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem Setting and Description

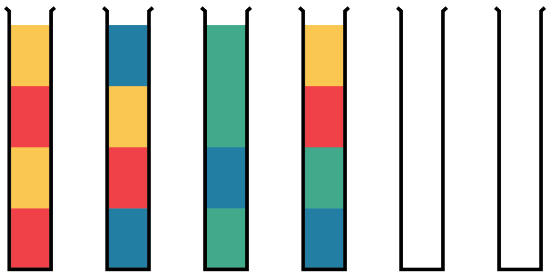


Figure 1.1: Example of a water sort puzzle.

The Water Sort Puzzle has recently gained popularity through smartphone games. The game starts with a random configuration of n tubes, each with a capacity of l, filled with c different-colored liquids, that is, a total of $n \cdot l$ units of liquids of c colors and e empty tubes. For example, the state in *Figure 1.1* $n = 4$, $l = 4$, $c = 4$, and $e = 2$. The goal for a given initial state is to sort the liquids so that each tube is either empty or filled with liquid of the same color. The player can pour out only the topmost liquid of a color from one tube into another only if the target tube's topmost liquid matches that color or if it is empty. The amount of colored liquid moved depends on the states of the tubes. If the target tube has the capacity, then all the liquids of the same color pour there. On the other hand, if the target tube doesn't have enough room, only the allowed volume of the same-colored water pours, filling up to the top of the target tube. The problem settings that will be discussed in methodologies make the game easily adaptive to various informed, uninformed

as well as local search algorithms. Our research explores different algorithms for solving this puzzle.

1.1.1 The Structure of the Project Paper

This paper will consist of the following sections:

- Literature review
- Methodology
- Description of the Algorithms
- Results

The ”Literature review” section will explore the different publications and research that has been done on this and similar games. The ”Methodology” section will focus on the different methods we will use to solve the puzzle. The ”Description of the Algorithms” will contain detailed explanations of the algorithms that have been implemented for this problem. Finally, the ”Results” section will analyze and compare the results of the algorithms implemented to try and find the optimal algorithm for solving this puzzle.

Chapter 2

Literature Review

As was mentioned in the introduction, Water Sort Puzzle is a new game, that only became popular at the beginning of the 2020s. Due to the novelty of the game, there are only a few papers on this topic. Because of this, our literature review also includes not entirely academic articles and papers, such as Medium articles and GitHub repositories.

2.1 Reinforcement Learning Solution

”Reinforcement Learning for the Water sort game”, published in September 2022, solves the Water Sort Puzzle using a machine learning technique, namely reinforcement learning: the agent is comprised of two components: a policy and a learning algorithm. The parameters of the policy, which is a mapping from states to actions that defines the agent’s behavior, are continuously updated by the learning algorithm based on the actions, observations, and rewards. The author used Deep Q-learning (DQN) and Double DQN models for the learning algorithms and Random, Greedy, and Epsilon Greedy policies in the project. The author proposes three methods for handling invalid actions: not handling them at all, assigning a negative reward for such actions, and masking out illegal actions. Hyperparameter tuning (implemented via grid search) was also utilized in the training phase. Discussing the results, the author states that a separate model was developed and trained for game variations with different numbers of tubes. Their results state that the policy with masked actions performs well (always finds the optimal solution) for the game variant with five tubes. Additionally, the same policy cannot be used to train a game variant with a different number of tubes[Ben22].

This does not coincide with the area of our study [Ben22].

2.2 BFS vs DFS

A different project on the same topic was undertaken by Ollie Haymen in 2021. He used breadth-first search (BFS) and depth-first search (DFS) to solve the puzzle. According to his findings and the pros and cons of these algorithms, the author concludes that BFS always finds the shortest path to a solution, sacrificing the time to find a solution in favor of optimality, whereas DFS generally finds a solution quicker, although the solution it finds is not guaranteed to be the optimal one [Hay21]. However, we should take into account that one of the downsides of DFS is the possibility of loops in the case of tree search, which leads to the unstoppable marching of time that is slowly guiding us all toward an inescapable cycle.

2.3 A* Search

Another article, titled ”Solving hoop games with A-Star”, aims to solve the Hoop Sort Puzzle, which is identical to the Water Sort Puzzle. The game consists of poles, which have an identical function to the tubes in the Water Sort Puzzle, and hoops, which are identical to the liquids. Another thing that makes the two games identical is that, like in the Water Sort Puzzle, in the Hoop Sort Puzzle played by the author, the hoops of the same color move simultaneously when we try to move the topmost hoop. The goal of this game is to collect all the same-colored hoops on the same pole. To solve the puzzle, the author makes use of the A* search algorithm along with a heuristic function developed by himself. The heuristic, which the author prefers to call a score, is calculated as `moves + (5 * trapped hoops)`, where `moves` is the number of moves already made and `trapped hoops` is the number of non-topmost hoops whose color does not match with that of the topmost hoop. As mentioned in the article by the author himself, this algorithm was written as a passion project, so it is not backed by any further research or any checks for optimality [Spe22].

Another attempt of an A* Search on our Water Sort Puzzle was done,

where the author takes the action of moving a single unit of water to be of cost one. He takes the heuristic to be the `number of tubes + the number of consecutive pairs of different colors - the number of sorted tubes - the number of empty tubes`, which essentially becomes the `number of unsorted tubes + the number of consecutive pairs having different colors`. The author, however, admits the heuristic to be inadmissible. This outcome was expected since the author takes the heuristic mentioned previously and adds the number of unsorted tubes, which is unnecessary [Per22].

Another Water Sort Puzzle solver has also been developed using the A* search algorithm and a heuristic by Florian Forster in 2022 [For22]. The action cost of moving a single unit of liquid is set to be a constant. The heuristic used by the author is quite similar to the one we have come up with. Basically, our heuristic is calculated as follows:

$$h = \sum_{\text{tubes}} (\#\text{ of distinct water units - 1}),$$

where the water units are groups of adjacent water cells of the same color.

The logic behind this heuristic is that for the tube to be complete, we need to remove at least all the water units in the tube except the one at the bottom. Hence we can conclude that this is an admissible heuristic. The heuristic developed by the author of this solution additionally takes into account the distribution of the bottom-most colors. For example, if a color is at the bottom of k bottles, at least $k-1$ moves are required. The author also mentions that other solutions to this problem use depth-first search (DFS); however, A* is advantageous because it guarantees the optimality of the solution, and the search space is dramatically reduced [For22].

2.4 General Analysis of the Problem

There is another game quite similar to Water Sort Puzzle yet less restrictive called Ball Sort Puzzle. A paper, published in Japan on February 19, 2022, shows that both puzzles are the same from the viewpoint of solvability, and then compares the computational complexity of the games. The authors provide an extensive representation of the game state, using which they prove that both the Ball Sort Puzzle and the Water Sort

Puzzle belong to NP problems and that both can be solved in O(l^n) time, where l is the number of levels per tube and n is the number of filled bins [Ito22].

A different paper, again published in Japan in August 2023, proposes a physical zero-knowledge proof protocol for the ball sort puzzle using a deck of playing cards, which enables a prover to physically show that the puzzle can be solved with some t moves without revealing it [Rua23]. However, this does not overlap with our area of study.

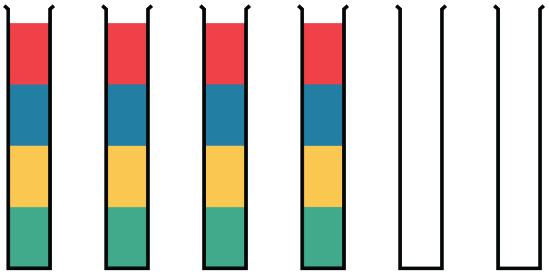


Figure 2.1: Example of a non-solvable water sort puzzle.

Now, let us briefly touch on the solvability of the puzzle. We did not find any literature that provides a method of determining whether the game state is solvable or not. However, not all states of the puzzle are solvable. An example of a non-solvable state is provided in *Figure 2.1*. [ant20]

Chapter 3

Method

3.1 Background

As mentioned previously, we start with a random configuration of n tubes filled with l units of liquids of c different colors, meaning a total of nl waters, and e empty tubes. If we wanted to count the number of all possible states, it would be approximately:

$$|S| = \frac{\binom{(n+e)l}{nl} \cdot (nl)!}{c \cdot l!}.$$

Now, let us try to explain this formula. Firstly, $\binom{(n+e)l}{nl}$ calculates the number of all possible combinations of empty/full water cells. This can overestimate the number of cases if it considers a lower cell empty but an upper cell full. Multiplying by $(nl)!$ gives us the number of combinations of the water units in the non-empty cells. Lastly, since there are c distinct colors each with l water units, we divide everything by $c \cdot l!$ so that we do not consider the permutations of the same color. This can overestimate the number of cases when $c! = n$, i.e., there are fewer colors than filled tubes.

The game has the sole action of pouring the liquid fully or partially from the top of one tube into another tube. This is valid when the initial tube is non-empty, the target tube is either empty or has the capacity to hold the moved amount of liquid, and most importantly the topmost liquid colors match in the case when both tubes are non-empty. Each tube essentially behaves as a stack since the actions deal exclusively with the topmost liquids, following the "last in - first out" (or LIFO) principle. For this reason, we will be representing the state of a tube as a stack reading the distribution of the colors from bottom to top. For instance, a state [

“r”, “r”, “b”, “g”] would mean that a tube contains two red units of liquid at the bottom, followed by single units of blue and green. Returning to the actions, in the scope of our formulation, we would consider the cost of a single action of pouring a liquid regardless of the amount moved to be constant, let us take 1. Finally, for a given initial configuration, the goal of this puzzle is to make each tube either empty or full of liquid of the same color.

Having the puzzle formulated as search problem, we can easily apply the known, uninformed (BFS, DFS), informed (mainly A* approach), and local (Random Walk, Hill Climbing) search strategies. But in order to be able to apply these strategies, let us lastly observe the properties of the puzzle environment. To begin with, the environment is single-agent, i.e., there is one agent playing the game. It is also fully observable as the agent has full knowledge regarding the states of all the tubes and liquids within them. It is discrete and deterministic; by applying the action pour (b_1,b_2), we know that it will pour the liquid of b_1 into b_2 if it satisfies the requirements (in this scope, if it does not satisfy them, we will simply ignore those branches), and the number of such actions is discrete. Since we initially assume no time factor is included, the environment is considered to be static. As the actions follow each other, the puzzle environment is also sequential. Finally, the environment is known because the agent is aware of the possible outcomes of applying an action.

Now, having determined the environment properties and verified that they satisfy the requirements of the search strategies, we can have them implemented. The children of a given state are the states generated after applying the pour action, with the liquid poured from one tube into another if and only if the move is legal, i.e., it satisfies the constraints. If the action is not legal, it is simply omitted from the generated states. An upper bound for the branching factor will be $(n+e)(n+e-1)$, where we simply take any tube from the available $n+e$ tubes and try to pour its top liquid into one of the remaining tubes.

3.2 Solving the Puzzle

We have tried multiple search algorithms to solve the puzzle. In this section, we will discuss the properties and implementation details of each

of them.

We will be writing the algorithms in Python. Now let us briefly introduce the representation of the problem in OOP. There are three main classes:

- Game_State - represents a game state in terms of bins with liquid;
- Bin - represents a tube or bin that holds liquid;
- Water_Unit - represents a water unit - certain volume of colored liquid.

Now, we will discuss the search algorithms we used to tackle this puzzle.

3.2.1 Uninformed Search Strategies

Let us first consider the uninformed search strategies. We will start the discussion with Breadth-First Search algorithm. At each step, the BFS algorithm expands all the child states of the current state generated by applying one of the legal pour actions, which already hints that it will converge to the solution slowly. Nevertheless, despite how slow the algorithm might be, due to its completeness, BFS will still lead to the shallowest solution, which is also the optimal solution in our case as the cost of each action is considered to be 1.

The Depth-First Search algorithm is another strategy that will pave the way towards the solution. Since DFS is depth-oriented, it will build upon the previous states and will quickly develop. If the sequence of actions does indeed lead to a dead end, it will ”backtrack” up to some actions and develop another path. DFS seems quite favorable if the solution lies in the leftmost branches. But if it lies on the rightmost branches, then the algorithm would still have to traverse the whole search tree. There is another concern with this algorithm; consider the state in Figure 2. The tree search version of DFS might end up in a loop trying to pour one unit of the red liquid back and forth from the first to the second tube. To avoid this issue we can implement the graph search version of the algorithm when we forbid revisiting a state that has already been observed throughout the path. Having this, DFS can serve as a good solver.

3.2.2 Informed Search Strategies

Now let us discuss how the informed search strategies can tackle this puzzle. We will consider only the A* search strategy, which has been already attempted before by other users on different platforms. We implement both the tree and graph search versions of the algorithm. As discussed, the action cost of pouring a liquid is constant. Now let us develop a heuristic by relaxing the problem. Assume we can sort each tube in one move. Then we'll get a heuristic, which is the sum of unsorted tubes:

$$H_1 = \text{number of unsorted tubes.}$$

This heuristic appears to be admissible, since it is not generally possible to sort an unsorted tube in just one move.

Now, assume all the constraints are omitted except that we only move the topmost water units. Then we would be able to take any liquid and place it in any tube without worrying about the available space. In that case, we are concerned with essentially how many of the liquids have to be removed from each tube. At least all but one water unit should be removed from each tube. Then we could take a heuristic to be

$$H_2 = \sum_{\text{tubes}} (\#\text{ of distinct water units} - 1).$$

This heuristic also appears to be admissible, since if there are n distinct water units, then at least $n-1$ of them should be removed, so that the tube gets sorted.

Let's build another heuristic based on the second one. For this one, we will also take into account the distribution of the bottommost liquids. For each color, we should have as many bottommost water units as many filled tubes of that color there are.

$$H_3 = H_2 + \sum_{\text{color}} (\#\text{ of bottommost units} - \#\text{ of filled tubes}).$$

This heuristic seems to be admissible, since on top of the logic for the H_2, the distribution of the bottommost liquids should also be fixed.

3.2.3 Local Search Strategies

Finally, let us consider the local search strategies for solving the puzzle. We will use Random Walk and Hill Climbing, and will discuss other strategies, but not necessarily implement them as we believe these two are sufficient in the scope of the game.

Let us first discuss the Random Walk for solving the puzzle. We give the initial state of a puzzle instance and randomly select a successor of that state, then randomly select from the successors of the successor, and so on. This approach seems quite short-sighted if we consider it to run once only. That is why we will also incorporate restarts, that is, the algorithm will run again from the initial state in case there are no legal moves available.

Hill Climbing can also be implemented to solve the puzzle. Let us discuss its advantages and disadvantages. Again, we will give the initial state of a particular problem to the algorithm; then, it will generate all the possible children from applying all the legal pour actions and take the best successor according to the given evaluation function. The goodness of the successor is determined by the evaluation function. The evaluation functions are the same as the heuristics defined in the A* search strategy. The goal states have the value of the evaluation function to be 0. The Hill Climbing takes a successor that minimizes the value of the evaluation function.

It is possible for the algorithm to stuck at a point from which no other action is legal, or if there is, the newly generated states are not better than the current state in terms of the evaluation function value. Here the algorithm stops. To address this issue, we also incorporated the sideways options, which, if set true, allows accepting states that equally good as the current state. However, this comes with its disadvantages, the biggest of which is the possibility that the algorithm will get stuck in a plateau.

Additionally, the randomness of the Simulated Annealing algorithm can be useful in solving this puzzle and not getting stuck in cycles.

Chapter 4

Results

In this chapter, we will discuss the results of each of the algorithms and then compare them with each other based on factors such as runtime, number of expanded nodes (where applicable), and solution length. This will help us find the search algorithm that best fits our problem of concern.

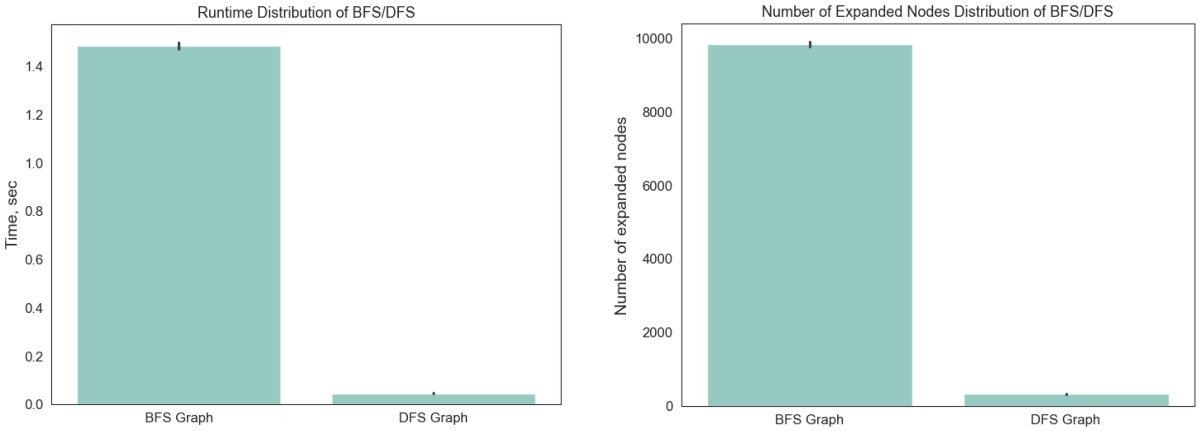
4.1 Analysis of the Algorithms

For the analysis, of the algorithms we generated 1000 random initial states with $n = 4$, $l = 4$, $c = 4$, and $e = 2$. The limitation of expanded nodes is set to be 10000; the maximum solution length (maximum depth for BFS and DFS) is 100 for every algorithm, except Random Walk, for which it is 200; the maximum number of restarts for Random Walk is 20. The ties are resolved by considering the tubes from left to right. Finally, runtime was measured by running the search algorithms on the same problem multiple times and taking the average.

4.1.1 BFS and DFS

As we can see from *Figure 4.1*, BFS exceeded DFS both in runtime and in the number of expanded nodes. BFS takes about 30 times longer on average than DFS. The same comparison holds for the number of nodes expanded by the algorithm during the search. On the other hand, as *Figure 4.2* suggests, the average solution length found by BFS is about 10 times shorter than that of DFS. This is because of the limitation on the number of expanded nodes. BFS is only able to solve the instances

where the solution is close to the initial state. In any case, the solutions found by DFS are not optimal, since it does a lot of unnecessary steps. If we were to give BFS more runtime by increasing the limit of expanded nodes, it would have been able to solve more cases optimally.



(a) Runtime distribution

(b) Expanded nodes distribution

Figure 4.1: Comparison of distributions for BFS and DFS.

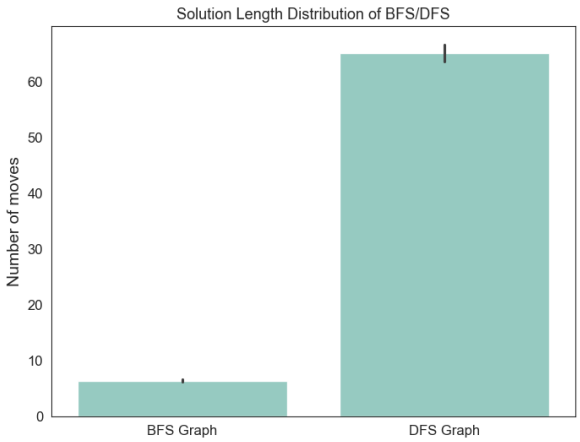
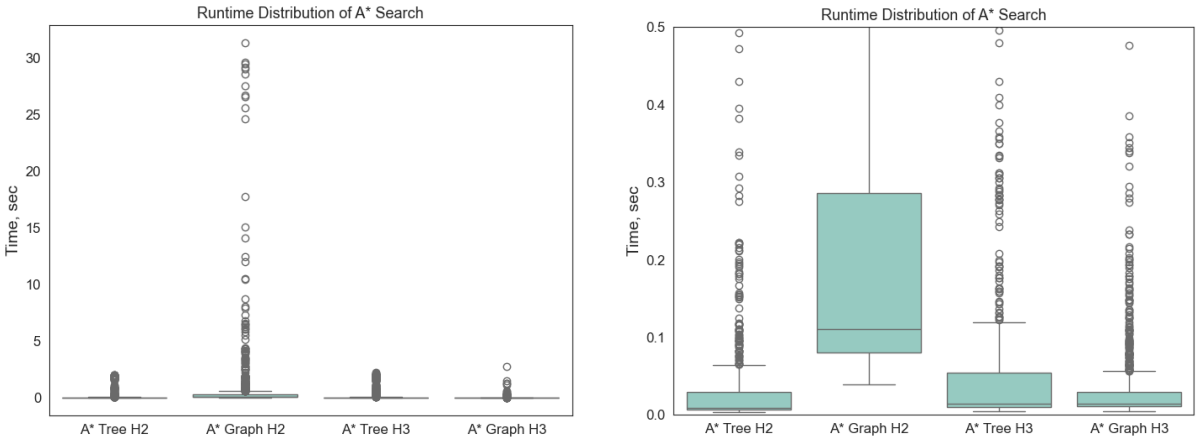


Figure 4.2: Solution length distribution of DFS and BFS algorithms.

4.1.2 A* Search

The Tree and Graph versions of A* search was performed using two different heuristics mentioned earlier in this paper. The first heuristic, representing the number of unsorted tubes, was excluded due to its negligible effect on finding the favorable path. *Figure 4.3 (b)* shows the clipped runtime distribution for better visual interpretations. The most evident anomaly is perhaps observed in the runtime of the Graph version

with H2. This is because the algorithm had to keep a larger set of the expanded nodes, which slowed down the code. For H3, this is not the case since it was a better heuristic that succeeded in finding the solution easier. *Figure 4.4* shows that the solution length is around 10, which is acceptable for the given game settings. Medians of the solution lengths coincide for all versions. However there is a range difference when examining the Tree and Graph versions with H3. While we could justify the admissibility for the heuristics, we could not say anything regarding the consistency of the heuristic, which is causing the Graph search version to deviate from the optimal solution. As can be seen in *Figure 4.5*, the Tree versions of the algorithm expand many more nodes as compared to the Graph versions.



(a) Runtime distribution

(b) Clipped runtime distribution

Figure 4.3: Comparison of runtime distribution for A* search algorithm.

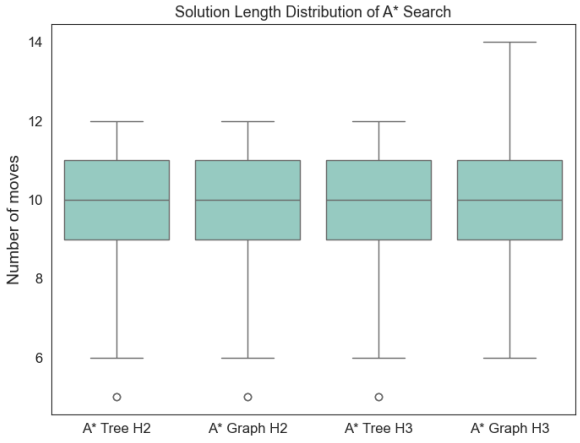
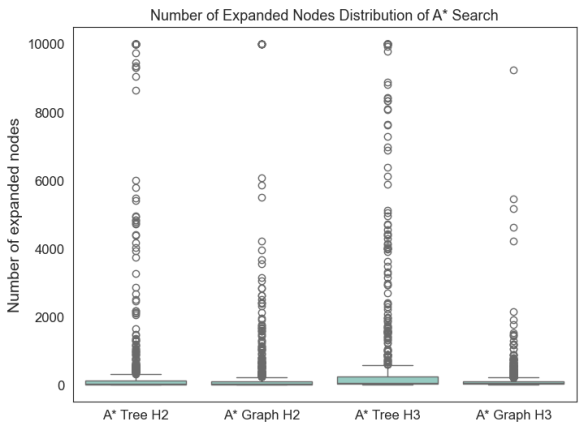
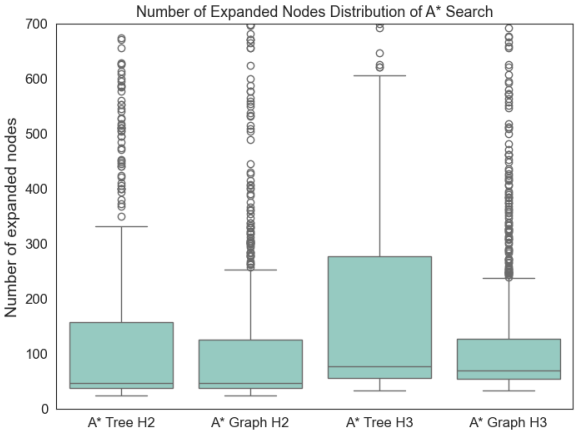


Figure 4.4: Solution length distribution of A* search algorithm.



(a) Expanded nodes distribution

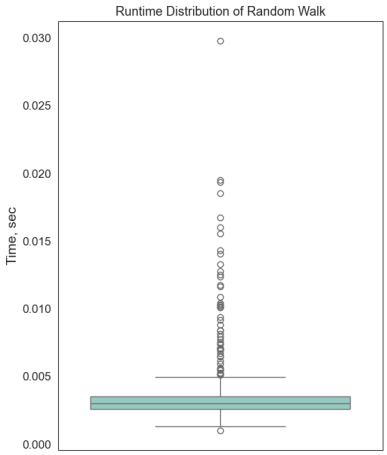


(b) Clipped expanded nodes distribution

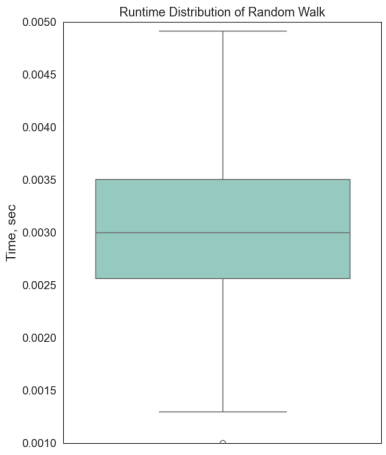
Figure 4.5: Comparison of expanded nodes distribution for A* search algorithm.

4.1.3 Random Walk

Surprisingly enough, the Random Walk algorithm almost takes no time to find the solution, as *Figure 4.6* shows. Looking further into the performance of the algorithm, it is observed that the number of restarts are 0 in the vast majority of cases (*Figure 4.7(b)*). The solution length falls into reasonable bounds for the given game instances as *Figure 4.7(a)* suggests; however, the optimality of the solution is not guaranteed.

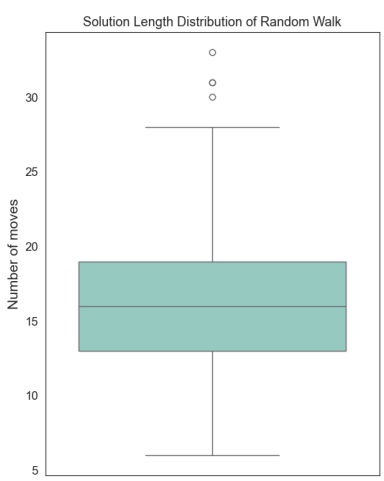


(a) Runtime distribution.

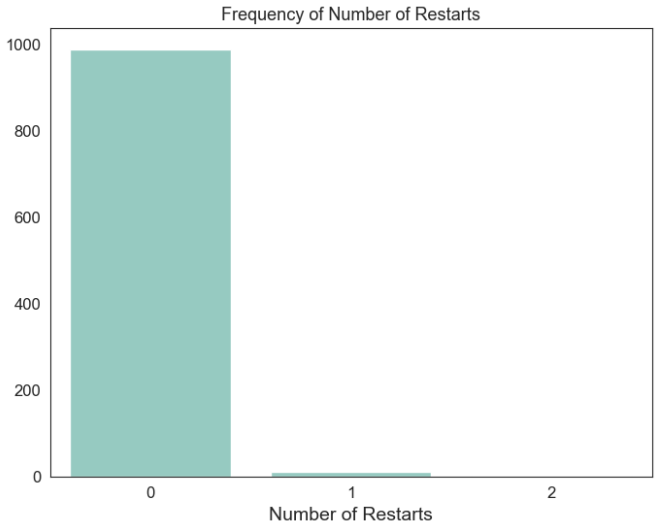


(b) Clipped runtime distribution.

Figure 4.6: Runtime distribution of Random Walk search algorithm.



(a) Solution length distribution.



(b) Frequency of restarts.

Figure 4.7: Solution length distribution and frequency of random restarts of Random Walk search algorithm.

4.1.4 Hill climbing

Finally, let us discuss the performance of the Hill Climbing Algorithm in the Steepest Descent version with three different above-mentioned evaluation functions. The algorithm was run with and without allowing sideways moves. As we can see on *Figure 4.8* and *Figure 4.9*, the terrible performance of the algorithm with heuristic H1 immediately stands out. Both in runtime and solution length, it demonstrates poor results. Pay attention also to the fact that without allowing of sideways moves, the heuristic returned the average solution length of zero, meaning it failed to solve any of the given instances of the game. This is because, with the unsorted tubes heuristic, the first move is almost always considered a sideways move, i.e., moving liquid does not reduce the number of unsorted tubes unless the tube from which the liquid was moved was sorted at the bottom in the initial state. The other two heuristics, on the other hand, run in significantly less time and show a reasonable number of solution lengths. Naturally, it's observed that the runtime gets longer as the sideways moves are allowed with any of the evaluation functions. Essentially, there is no significant difference observed between the two heuristics when considering these metrics.

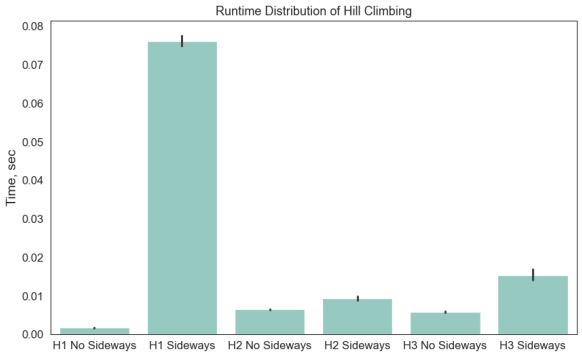


Figure 4.8: Runtime distribution of Hill climbing algorithm.

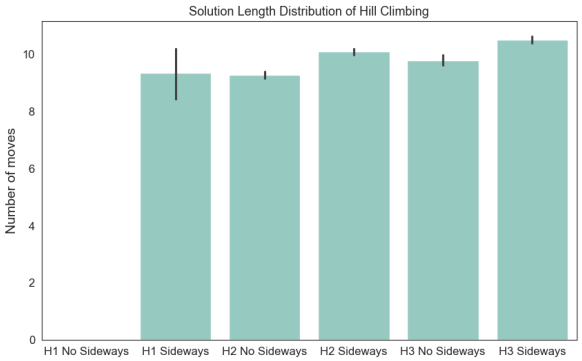
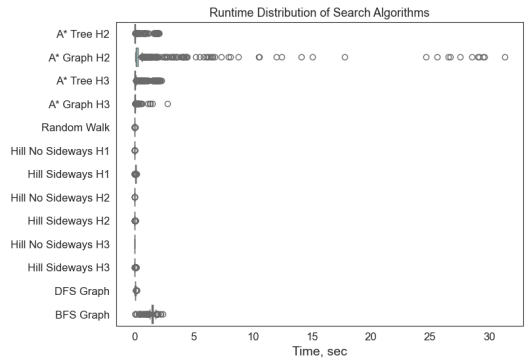


Figure 4.9: Solution length distribution of Hill climbing algorithm.

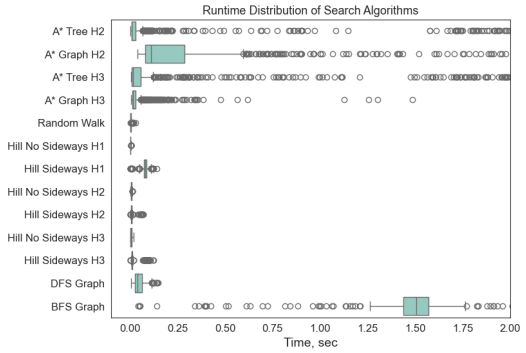
4.2 Comparison of the Algorithms

Wrapping up, let us compare all of the search strategies by their runtime, solution length, the number of solved cases, and the number of expanded nodes. Again, it is important to take into account that this is not the full potential of the performance of the informed and uninformed search strategies due to the limitation on the expanded node number. As can be seen in *Figure 4.10(a)* and *Figure 4.10(b)*, the worst runtime was demonstrated with BFS, the reasons for which were discussed earlier in this paper. All other algorithms did pretty well, even the cute Hill Climbing with H1, who gave up fast and said that it was too weak to find the solution. Regarding the solution length, it is seen that DFS did the worst, adding too many redundant moves in its solutions. Let us finally answer the most important question: are these algorithms actually able to find the solutions? The *Figure 4.11(b)* demonstrates the number of solved cases by each algorithm. BFS, and Hill Climbing with H1 failed to

solve most of the 1000 game instances passed to them, the reasons for which have already been discussed. Surprisingly enough, Random-Walk, Graph version of DFS, and Graph version of A* with H3 solved all of the cases given to them, even with the limitations set on them. The other algorithms are not falling too far behind, succeeding in solving more than 90 % of the given game instances.

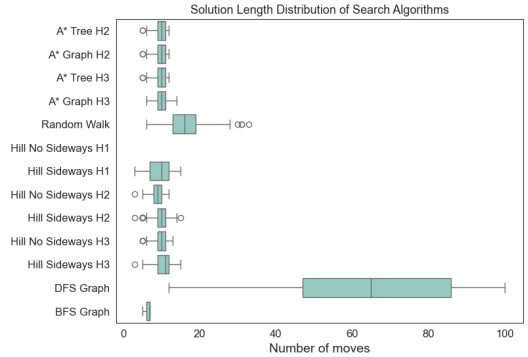


(a) Runtime distribution.

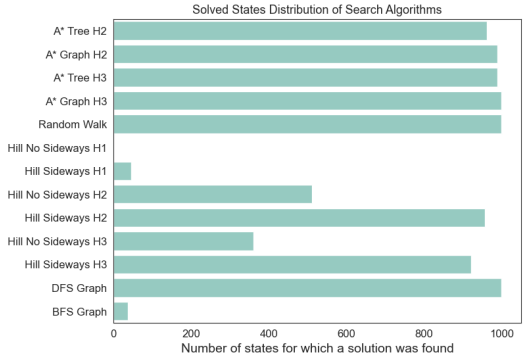


(b) Runtime distribution of all algorithms (clipped).

Figure 4.10: Comparison of runtime distribution for all search algorithms.



(a) Solution length distribution



(b) Solved states distribution

Figure 4.11: Comparison of solution length and solved states distribution for all search algorithms.

The algorithms were rerun several times with other game settings to observe the metrics. 1000 random initial states with $n = 8$, $l = 5$, $c = 8$, and $e = 2$ were generated. The metrics stayed proportional for each strategy due to the more complex nature of the configuration, and solutions were again found in vast majority of cases. Individually, extreme cases were generated to assess the performance of the algorithms.

The ones that showed great performance previously again successfully passed these tests.

Overall, we are more than satisfied with the performances of the algorithms. We will simply eliminate the strategies with the worst performances to solve this game, which were BFS and Hill Climbing with H1.

Chapter 5

Further Work

5.1 Unknown Colors

For further work, we could try to develop search strategies for the variation of the game where the bottom colors are unknown, as shown in *Figure 5.1*. However, solving this puzzle setting will be more challenging since we now deal with a partially observable environment. Moreover, it will be impossible to use the current heuristics in this case, as the agent will not be able to assess the states.

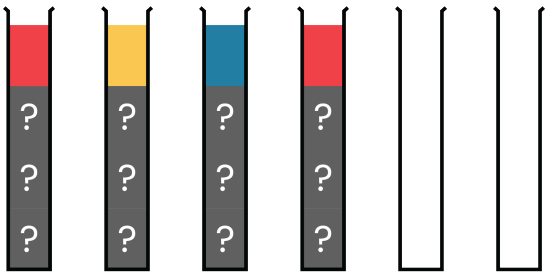


Figure 5.1: Example of a water sort puzzle with unknown colors.

Bibliography

[ant20] antkam. *Does Ball Sort Puzzle always have a solution?* June 2020.

[Hay21] Ollie Hayman. *GitHub - discorev/colour-puzzle-solver: A python solver for colour sorting puzzle games implementing both breadth-first and depth-first algorithms*. Feb. 2021.

[Ben22] Selma Benouadah. “Reinforcement Learning for the Water sort game”. In: *ResearchGate* (Sept. 2022).

[For22] Florian Forster. *GitHub - octo/watersort: Solver for Water Sort Puzzle levels.* June 2022.

[Ito22] Takehiro Ito. *Sorting Balls and water: Equivalence and computational complexity*. Feb. 2022.

[Per22] Persimmon-Persimmon. 広告でよく見る試験管パズルゲームをプログラムで解いてみた. Apr. 2022.

[Spe22] Ted Spence. “Solving hoop games with A-Star - CodeX - Medium”. In: (Nov. 2022).

[Rua23] Suthee Ruangwises. *Physical Zero-Knowledge proof for ball sort puzzle*. Jan. 2023, pp. 246–257.