INF-2600-1 24V: Artificial Intelligence, AI - Methods and applications

Assignment 1: Search

Amund H. Strøm
UiT id: ast283@uit.no

February 15, 2024

1 Introduction

This report describes the design and implementation of different search algorithms and their result on different problems. The search algorithms discussed in this report are designed to search through a tree data structure to find a solution the the given problem. The problem in this report revolves around solving a "Cube Tower" which consists of four cubes each with four sides with different colors, and the goal is to align the colors. Each algorithm in this report employ a different strategy and is therefore given different names, respectively Breadth-first search, Depth-first search, A* search, and Greedy Best-first search.

2 Design

This section will give a high-level view of some of the core concepts and ideas discussed in this report.

As described in the introduction the "Cube Tower" consists of four cubes that each have four different colors. The solution to the problem is to align the colors. There are 2 different moves that is allowed to preform on the tower. First move is to rotate a cube and all the other cubes above it. The second move is to hold two different cubes, and rotate the cubes you are holding and the cubes in-between. This makes for a total of 10 possible different moves. Interestingly, there is a move that is redundant which is to rotate the lowest cube and all the cubes above it. This move does not change the configuration of the tower, making it redundant. This means there are 9 moves that changes the configuration of the tower and should be considered.

As implied by the description, the problem will be represented by a tree like data structure. The tree is defined by nodes that contain general information about the state of the cube tower, most importantly the current configuration. The root of the tree is the initial configuration of the cube tower, and each node has 9 children, which is created by the 9 moves that changes the configuration. To find the solution to the problem, one must navigate the tree and find a state that has the colors aligned.

To navigate the tree one must apply a strategy to search the different nodes. A strategy in this case is called a search algorithm and there are many different search algorithms with varying level of advantages and disadvantages. This will give a high-level view of the strategy the different algorithms in this assignment uses to navigate the tree and find a solution.

- **Breadth-first search** preforms 5 tasks in a loop:
  1. Check if solution is found. If found, exit. If not, continue.
  2. Expand the current node (create all the possible children).
  3. Put the children in a global queue.
  4. Visit the next node in the queue.
  5. Check if the queue is empty. If not empty, repeat. If the queue is empty, no solution was found.

  This makes the algorithm explore the tree level-by-level, since it visits the nodes in a queue order (First-in First-out). It also always find the optimal solution since the first solution it encounters is the highest level solution. The disadvantage to this is that it usually uses a large amount of memory and time. It uses a lot of memory since it must save each node in every level, while the solution is commonly found some levels deep in the tree, resulting in a lot of nodes that needs to be saved. It uses a lot of time for the same reason since it must visit every node in each level which will be very time consuming.

- **Depth-first search** preforms 5 tasks in a loop:

  1. Check if solution is found. If found, exit. If not, continue.
  2. Expand the current node (create all the possible children).
  3. Put the children in a global stack.
  4. Visit the next node in the stack.
  5. Check if the stack is empty. If not empty, repeat. If the stack is empty, no solution was found.

  This algorithm behaves very similarly to Breadth-first search, but has some key differences. It explores the tree branch-by-branch, since it visits the nodes in a stack order (First-in Last-out). This makes the algorithm find the solution generally fast and spend little resources, because the solution is usually found deeper in the tree. The disadvantages is that it will most likely not find the most optimal solution.

- **A* search** is an informed search algorithm and preforms 6 tasks in a loop:

  1. Check if solution is found. If found, exit. If not, continue.
  2. Expand the current node (create all the possible children).
  3. Calculate the value of each node using an *evaluation function.*
  4. Put the children in a global priority queue, where the children with the lowest value is put at the front of the queue.
  5. Visit the next node in the priority queue.
  6. Check if the priority queue is empty. If not empty, repeat. If the priority queue is empty, no solution was found.

  The *evaluation function* of A* looks like this:

  $$f(n) = g(n) + h(n)$$

  $g(n)$ represents the cost it takes to get to this state from the initial state. In our case it could be interpreted as the depth of the node. $h(n)$ is the *heuristic* value, determined by a problem-specific function that calculates the cheapest path from the current state to the goal state.

  By enabling the evaluation function to prioritize which node to visit next, the algorithm essentially *prunes* the nodes that are not likely to give an solution, and it will effectively make ”smart” decisions to get faster to the goal state. This makes it an informed search algorithm.

  This has a lot of advantages compared to the previous search algorithms. It will be considerably faster and spend considerably less resources. But it may not find the optimal solution or no solution at all, depending on how the heuristic value is calculated.

- **Greedy Best-first search** Works very similarly to A* but with a small change in the evaluation function, now it simply looks like this:

  $$f(n) = h(n)$$

  This makes the algorithm choose the next node to visit solely depending on the heuristic value. This can cause the the algorithm to walk directly towards the goal and not check for other better options depending on the cost.

3 Implementation

The implementation of the tree and search algorithms matches with what is described in the design section. This section will go into detail of some key points about the implementation of the representation of the Cube Tower and how the search algorithms were implemented.

The **Cube Tower** is defined by a class which most importantly contains the tree data structure and the current node that is being visited. It also contains a method that lets it rotate the current node's configuration in every possible way and adds the new configurations as children for that node. The nodes them self are represented by another class which contain metadata about the node such as configuration, depth, heuristic value, parent and children.

The search algorithms are divided into 2 main methods, one method for the uninformed algorithms and one method for the informed algorithms. They both behave very similarly but have some key differences.

The **uninformed search algorithms** generally work just as described in the design section. A main loop that checks for completeness, expands the node, puts the children into a list, and visit next node. An important design decision is that this function uses a dictionary to keep track of nodes that it has already seen. If a node has already been seen it wont be visited again, this way it can avoid looping forever, since eventually it will run out of any new configurations.

The main method that the uninformed search algorithms uses takes two arguments, one that should be the Cube Tower which should be solved, and a second argument that should be a method for the order the new children should be added into the list for visiting. This can either be a *queue* or *stack* method.

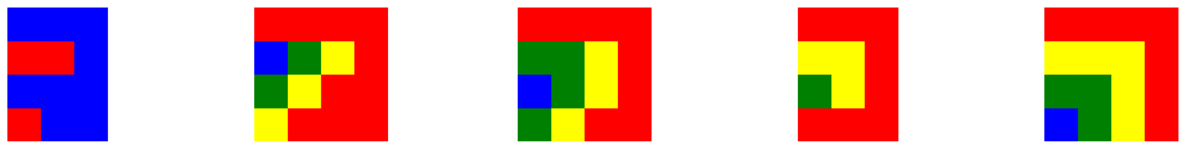
The main method for the **informed search algorithms** works very similarly as the main method for the uniformed algorithms. That is, it contains the logic for the evaluation function and priority queue. Each time a child is created its value is calculated and put into the priority queue, the queue then gets sorted before picking the next node to visit.

The main method for the informed search algorithms also takes two different arguments, one that should be the Cube Tower which should be solved, and a second argument that should be the method for how the evaluation function is calculated.

4 Results

To benchmark the different search algorithms they were compared against each other on 5 different Cube Tower configurations. The benchmarks recorded the time used, memory used, and the number of moves from the initial state to the solution state.

These are the different configurations the algorithms solved. These figures shows the initial configuration on the left and the most optimal path to the solution on the right.



(a) Problem 1 (b) Problem 2 (c) Problem 3 (d) Problem 4 (e) Problem 5

Figure 1: The most optimal path for each Cube Tower problem

The following illustrations shows the time used, memory used, and moves used to solve each problem. Each Cube Tower or problem is represented as a color and matches the order given above.

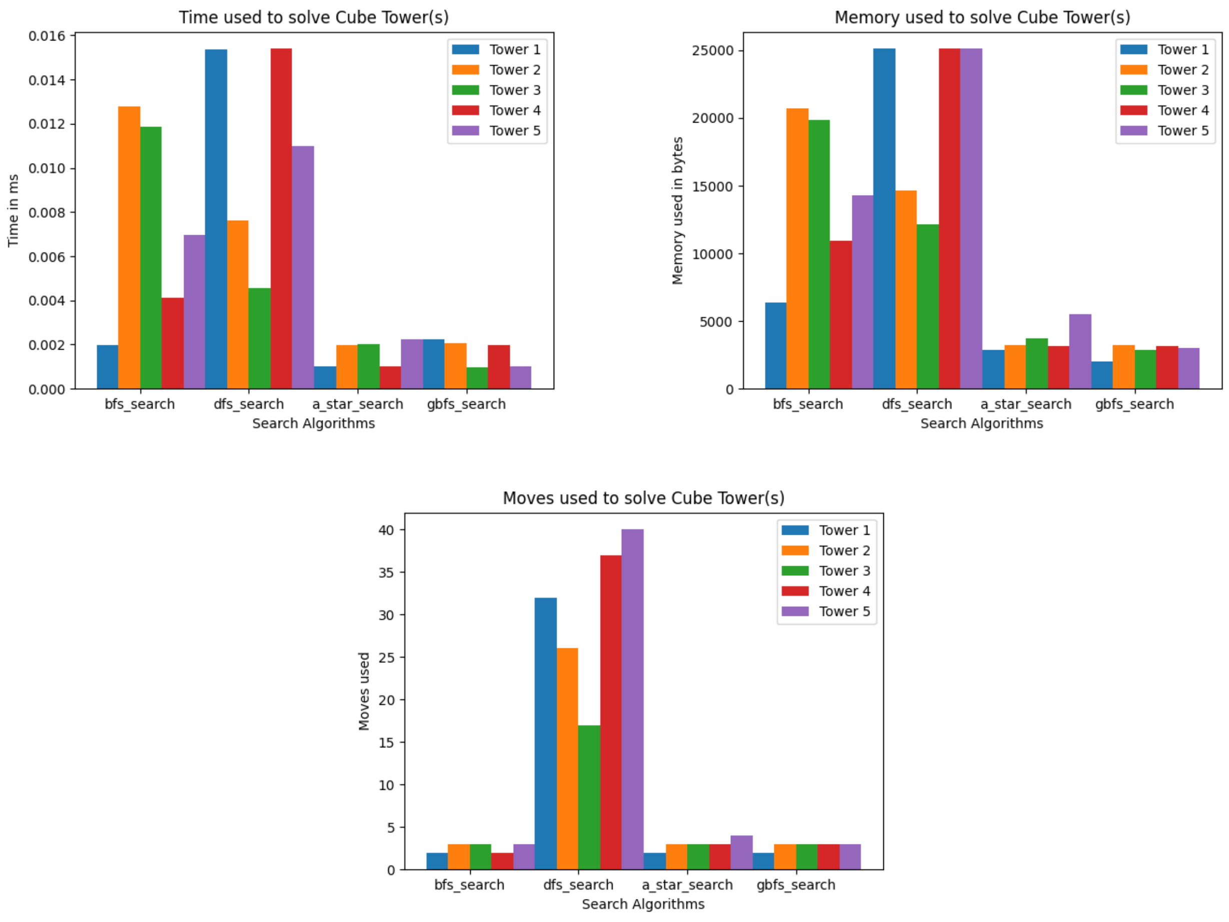


Figure 2: Bar charts that shows the time, memory, and moves used.

5 Discussion

As seen in the bar charts the informed search algorithms preforms much better in every benchmark, which matches with the theory. The only exception is a small loss in moves used to solve the Cube Tower for the informed searches. We also see that each search algorithm generally preforms as expected from their respective theory, as described in the design section.

If we compare the uniformed search algorithms internally because they were supposed to behave very similarly. We see that Breadth-first search (BFS) overall preforms better than Depth-first search (DFS). In theory, DFS should use less memory and be faster than BFS. There are many reasons why this is not the outcome, but the two main reasons in this case is the environment we are using the algorithms in, and the implementation. In this environment the solution is generally not very deep into the tree. This can be observed by looking at the number of moves used in figure 1. This means that it is unnecessary to go deep into the tree to look for a solution, resulting in more time spent looking for a goal state. The implementation of DFS could also be done better to spend less resources. If it was implemented recursively, it would not need to save the nodes that needs to be visited in a separate list, resulting in less memory used. The only thing that turned out according to theory is that BFS finds the optimal solution, while DFS does not find the optimal solution.

Comparing the informed search algorithms against each other we see that they preform very similarly in each benchmark. The reason for this is mainly the environment, because there isn't that many different paths to an optimal solution. Based on the heuristic alone, you will always find a solution and will never get stuck in a local minima. Further, the cost in each path depends solely on the depth and will therefore never give any real alternative path that minimizes the cost. For these reasons there wont be any huge differences in time and memory used. Therefore, the small variations we see in the bar charts is likely due to the inaccuracy of the measurement tools.

Interestingly we see that both A* search and Greedy best-first search does not give the optimal solution in some cases. The only reason for this is that the calculation of the heuristic value is implemented wrong. The following figure is a visualization of how to heuristic value is calculated.

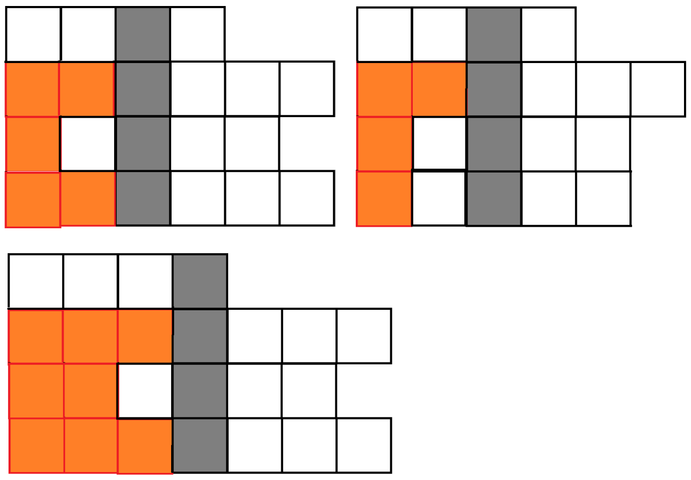
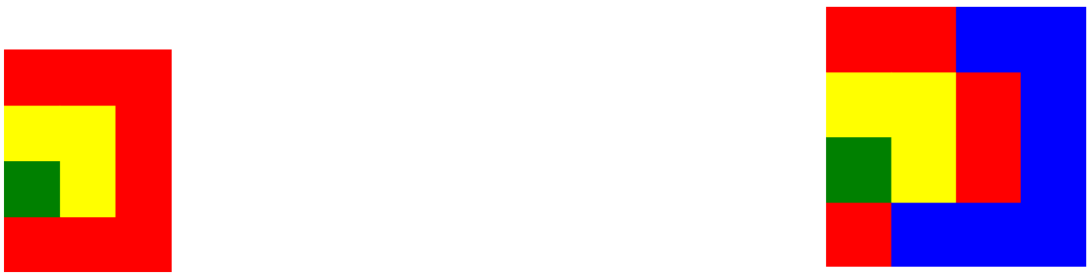


Figure 3: Visualization of the heuristic value.

Each <u>row</u> of the white/gray squares represent a single cube (four sides), and the gray squares represent the forward-facing color. The heuristic value is calculated by "counting" the orange squares, the less square the better. When there are no orange squares, the colors must be aligned. There may be some cases where an action decreases the most amount of orange squares on the first move, but will cause an extra move later on. This can be observed in this problem.



(a) Solved by BFS (b) Solved by A* and GBFS

Figure 4: A* search and Greedy best-first (GBFS) search does not choose the optimal path.

We can observe in the figure that the informed searches makes the wrong decision in the first move. The move that BFS made has a total of 4 orange squares, and the move that A* and GBFS has a total of 3 orange squares. To fix this you could maybe also look at how many of the same colors that are facing forward.

6 Conclusion

This report has given a high-level view of the design and implementation of a variety of search algorithms. It has shown the performance of each algorithm, and discussed how and why it turned out as it did. The report also discussed why the heuristic calculation may be wrong, and gave a solution that might solve the problem.

References

[1] GeeksforGeeks contributors. Difference between BFS and DFS. In GeeksforGeeks. Retrieved 15 Feb, 2024, form `https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/`

[2] Wikipedia contributors. A* search algorithm. In Wikipedia, The Free Encyclopedia. Retrieved 15 Feb, 2024, from `https://en.wikipedia.org/wiki/A*_search_algorithm`

[3] Code Academy contributors. Greedy Best-First Search. In Code Academy. Retrieved 15 Feb, 2024, from `https://www.codecademy.com/resources/docs/ai/search-algorithms/greedy-best-first-search`