Indonesian Dot Puzzle - COMP472 Project 1 Report

Duy-Khoi Le

40026393 alvyn279@gmail.com

1 Abstract

Heuristic-based search algorithms are a powerful means to improve the speed and efficiency of a search, given that the chosen heuristic follows good admissibility, monotonicity, and informedness.

2 Introduction

This report describes the implementation process of the solutions to the Indonesian Dot Puzzle, and analyzes the results obtained.

2.1 Implemented solutions

In this project, three types of search algorithms were implemented in order to solve the Indonesian Dot Puzzle. Namely, *Depth-First Search (DFS)*, *Best-First Search (BeFS)*, and *Algorithm A* (AStar)* provided different ways to reach the same goal of attaining the perfect board (the goal state).

In this report, it will be seen how the difference in data structure usage and sequence of operations can impact the efficiency of the search algorithm as a whole.

2.2 Technical Details

The solutions were implemented in *Python3*. No additional frameworks were used, as all functions and modules were imported from the Python standard library. The exposed script can be ran on any machine that has *Python3* installed. Please refer to the repository's README.md for additional information.

2.3 Heuristics Search Concepts

In this sub-section, important concepts regarding heuristics will be reviewed.

$$f(n) = g(n) + h(n) \tag{1}$$

f(n) corresponds to the total cost to reach the goal state

g(n) corresponds to the actual cost from start to node n
h(n) corresponds to the estimated cost to reach the goal state from node n [1]

For Best-First search, the lower the value of h(n), the higher up in priority the node will get in terms of being visited. Equation (1) depicts the evaluation function for Algorithm A/A*, an evolution from Best-First Search, as the latter is solely based on computing priority with the heuristic function *h(n)*.

As a matter of fact, Algorithm A* takes into account the number of visits (cost) from the starting node in its calculation of f(n), whom ultimately determines its priority in the priority queue. These concepts were important for my implementation of heuristic search algorithms.

3 Heuristic

3.1 Disclaimer

I realized at an extremely late stage in the implementation and report-writing that my heuristic was actually inadmissible. I discuss in this section why the heuristic does not meet this criteria, and also in the *Results* section how the obtained results seem to contradict this fact.

3.2 The Checkered Board

I call my main heuristic ***The Checkered Board***. Its heuristic function calculates the number of cell-by-cell inconsistencies for a board in comparison to the *perfect checkered board*. The latter is illustrated in Figure 1.

The idea behind wanting a board as close to the perfect checkered board as possible is one that is based around the only possible action of the game: flipping a cross-shaped aggregation of tokens. As the *touch* action modifies the current and all adjacent tokens (left, right, up, down), **we want to set up the board in a state where each touch can turn a majority of the touched tokens into the same value tokens. In other words, the less the inconsistencies with the perfect checkered board, the more symmetric the resulting board will be following any *touch* action**. However, a misconception or a concern regarding the heuristic is the following:

Even though the perfect checkered board is achieved, what guarantees you that a solution will be found?

This heuristic is not guaranteeing a solution, but rather setting up the board in a state which promotes subsequent flips to make multiple same value changes. For example, touching a token where all four adjacents are *1*s will transform them all to *0*s, all four at once. This action therefore takes a step into the direction of the goal state, which is a board of all *0*s.

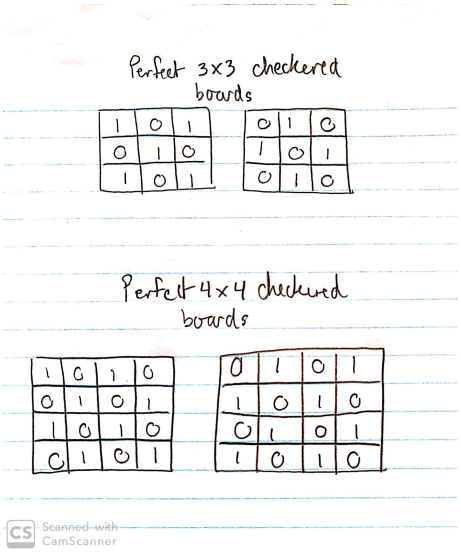

Fig. 1. Perfect checkered board states for 3x3 and 4x4 boards

3.3 The Checkered Board - Respecting Heuristic Design Criteria

To obtain satisfying results, it is important that this heuristic must meet predefined criterion.

As mentioned earlier in this report, I realized that my heuristic was not admissible. As seen in Figure 2, at board state (0), the puzzle can be solved with one move (touching the bottom right token). However, my heuristic would choose board (1) over board (0). We can then generalize that for all nodes, the estimated cost to the goal state (*h(n)*, being the number of inconsistencies from a goal state) overestimates the true cost to the goal state[1]. Hence, the statement that *for all nodes n visited, (h(n) < h*(n)[1]* does not hold. At first, I believed that the maximum value *(h(n))* being the number of tokens on the board would be a sufficient condition for admissibility. I was wrong.

The heuristic is monotonic as it uses the number of inconsistencies as the deciding factor for its priority. As a matter of fact, at every node, the children of the node are all inserted into the priority queue based on their state. If the number of inconsistencies for the node is low, it will most likely be retrieved earlier from the priority queue later on.

The heuristic is informed enough. As a matter of fact, an O(t) (t being the amount of tokens on the board) pass is necessary at every node that is visited in the priority queue. Basically, a snapshot of the board state is the means to make this heuristic more informed.

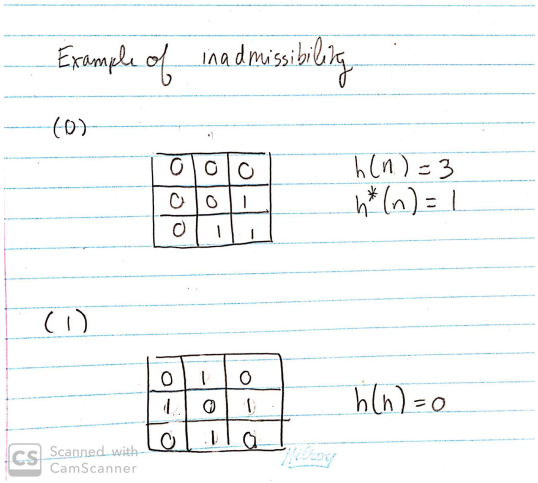

Fig. 2. 3x3 board state example of heuristic inadmissibility

3.4 Additional Disadvantages

I think one of the big disadvantages of this heuristic is that for small-sized boards, a few inconsistencies can still lead to a very asymmetric board state. Basically, a few flips to the area containing the inconsistencies will most likely propagate this concept of asymmetry and disorder throughout the board.

This being said, the fact that the boards are ordered in increasing value of inconsistencies in the open list ensures that uninteresting children nodes will not be visited as soon as possible.

Another disadvantage or flaw to this heuristic lies in the inherent nature of the checkered pattern itself. Even though we are trying to maximize the number of same value changes, touching anywhere **on the perfect checkered board** can possibly leave at least one value, the token that is touched, different from the others. Fortunately, the rapid computation and use of the priority queue hides from the user the extra moves that are needed to recover from this *dangling* token.

4 Difficulties Encountered

4.1 Design of a Robust Framework

Many implementation-level difficulties arose during the initial sketch of how the algorithms were going to be implemented.

One of the difficulties that quickly appeared was being able to hold the full representation of a board along with its various characteristics in-memory. A 2-dimension array (list in Python) of *1*s and *0*s was not sustainable, and would

result in extremely messy code. That's the reason why I started the implementation with establishing class models. With these, objects that were created in the algorithms would have to respect a contract. Furthermore, I would later on be able to make use of functional programming features in Python, such as sorting based on attributes through lambdas [2].

The design decision aforementioned also greatly helped in the establishment of backtracking in my algorithms (e.g.: DFS). As a matter of fact, maintaining data structures of custom type was rewarding as it allowed me to keep multiple states in-memory. Whenever it was necessary, I would pop out the custom object and be manipulating a past snapshot of the tree traversal.

In the context of this whole project, I noticed that we were implementing different algorithms that were destined to produce the same type of output/-solution. Hence, to make my code base scalable to future requirements, I made use of the Strategy design pattern to decouple the game runner with the type of algorithm it had to use. In other words, the game runner can dynamically change the algorithm by using another type of strategy object [3]. This suited well the use case, as multiple algorithms had to be tested on a single Game object, which encapsulated the game data given as input in the input files. I thought that this was good design.

5 Results Analysis

5.1 The Input/Output

Given the following input:

```
4 15 1000 1010010111001010
3 7 500 111001011
```

For analysis purpose, the upper limits of the max length of the search path were set to high numbers, such that a solution could be found.

```
Found solution!

0  1 0 1 0 0 1 0 1 1 1 0 0 1 0 1 0
B0 0 0 1 0 1 0 0 1 0 1 0 0 1 0 1 0
B2 0 0 0 0 1 1 1 0 0 1 1 0 1 0 1 0
C0 0 0 0 0 0 1 1 0 1 0 1 0 0 0 1 0
C1 0 0 0 0 0 0 1 0 0 1 0 0 0 1 1 0
C2 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0
D2 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1
D3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0


Time for DepthFirstSearchStrategy : 0.769700288772583 seconds

Found solution!
```

```
0  1 0 1 0 0 1 0 1 1 1 0 0 1 0 1 0
D1 1 0 1 0 0 1 0 1 1 0 0 0 0 1 0 0
C0 1 0 1 0 1 1 0 1 0 0 0 0 0 0 1 0
B3 1 0 1 1 0 1 1 0 1 1 1 1 1 1 0 1
A2 1 0 0 1 0 1 1 1 0 0 1 1 0 0 0 1
A3 1 1 0 0 1 1 0 0 1 0 0 0 1 1 0 1
B3 1 1 1 1 0 1 1 0 0 0 1 0 0 0 0 1
D2 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0
C0 1 1 1 0 1 1 0 1 1 1 1 1 1 0 0 1
D0 1 1 1 1 1 1 0 1 0 0 0 0 0 0 0 1
C3 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0
A1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Time for BestFirstSearchStrategy : 0.636239767074585 seconds

Found solution!

0  1 0 1 0 0 1 0 1 1 1 0 0 1 0 1 0
D1 1 0 1 0 0 1 0 1 1 0 0 0 0 1 0 0
A0 0 1 1 0 1 1 0 1 1 1 0 0 1 0 1 0
B0 0 0 1 0 1 0 1 1 0 0 1 1 1 0 0 0
A3 1 0 0 1 0 1 0 0 0 1 1 0 0 0 0 1
D1 0 1 0 0 1 1 1 0 0 1 0 0 0 0 0 0
B1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Time for AStarSearchStrategy : 0.37917399406433105 seconds

No solution found

Time for DepthFirstSearchStrategy : 0.016571998596191406 seconds

Found solution!

0  1 1 1 0 0 1 0 1 1
B1 1 0 1 1 1 0 0 0 1
C0 1 0 1 0 1 0 1 1 1
A1 0 1 0 1 0 0 0 0 1
B2 0 1 1 1 1 1 0 0 0
C1 1 1 1 0 1 0 0 0 0
A1 0 0 0 0 0 0 0 0 0

Time for BestFirstSearchStrategy : 0.022060632705688477 seconds

Found solution!

0  1 1 1 0 0 1 0 1 1
B1 1 0 1 1 1 0 0 0 1
C0 1 1 1 1 0 1 1 0 1
B2 0 0 1 0 0 0 0 1 0
```

```
C2 0 1 0 1 0 1 1 0 1
C1 0 1 0 1 1 1 0 1 0
B1 0 0 0 0 0 0 0 0 0

Time for AStarSearchStrategy : 0.014203310012817383 seconds
```

5.2 First Thoughts

From these results given the sample input, the mainly interesting point has to be the solving time progression that can be observed from *DFS* to *BeFS* to *AStar*. The improvement (deltas in the order of ˜100 ms for the 4x4 board) is actually huge in terms of computing time at each algorithm. After observing this behaviour across multiple other sample inputs, we can suggest that the heuristic chosen (perfect checkered board) can successfully improve the runtime complexity of the solver, at least in comparison to the classic DFS solutions.

Looking at these results, the advantages/disadvantages in usage of data structure from DFS to BeFS and AStar are deeply highlighted. In DFS, a stack was used as the underlying data structure of the open list. As all nodes are pushed onto the stack (except those from the closed list), the process of iteration will always be following the *LIFO (Last-in First-out)* concept. This way of storing nodes actually characterizes the *uninformedness* nature of this algorithm. The complete opposite is seen in BeFS and AStar searches, where nodes each have a visiting priority value determined by the heuristic function. To manage this priority, a min-heap (priority queue) is used based on the priority value: small priority values, or *h(n)*, lead to higher priority.

5.3 Uninformed Searching vs. Informed Searching

From the search files (that are not included in this report but can be generated with running the script), it is very noticeable that an uninformed search will explore many more nodes as opposed to heuristic search, which optimizes nodes that are visited. For example, in the 4x4 board of the sample input, DFS searches through 4129 nodes before getting to a solution. On the other hand, BeFS and AStar both go through only 389 and 314 nodes respectively. The algorithm can therefore have an even greater impact as the board increases in size.

5.4 The Flaw in my Heuristic

Surprisingly enough, AStar still yields results that make sense, even though my heuristic was not admissible. What I mean by this is that AStar produced the shortest input out of the three algorithms for several runs. From what I've observed, if a given sample input has really small maximum search path length (less than 50, for example), my algorithm would end up not being able to find a solution even to begin with. This makes sense, as my heuristic overestimates the

actual cost to find the solution, and hence is not visiting the objectively more promising tree nodes.

Within most my tests, I would say the AStar algorithm would produce the shortest path out of all three algorithms around 50% to 60% of the time. Of course, this is not the expected result. Ideally, my heuristic would be admissible, such that AStar would be applicable to the board. This would ALWAYS give me the shortest solution path.

5.5 Small Optimizations

I ran various sample tests with my different implementations of the algorithms. It was observed that the execution time was greatly improved if I kept a mechanism to check for an already visited node in constant time. This was achieved by hashing the visual representation of the board and storing it as a string in a set.

5.6 The Case of a More Promising Path

The implementation of the heuristic searches required the coverage of a particular case in the unraveling of the children nodes. If a node was analyzed to be a state of the board already encountered, but that had a higher priority than the latter, this higher priority node had to replace the existing node in the priority queue. This means that an update in the priority queue was necessary. To achieve this, I used a custom implementation of a priority queue that allowed the removal/update of an existing node anywhere in the min-heap[4].

5.7 Alternative Heuristics

The results obtained also enlighten the context in which my heuristic was preferred to other simplistic heuristics. If we take the *classic heuristic* that is the number of inconsistencies with the goal state, we can see that my heuristic takes into consideration a concept that the classic one doesn't: the flipping action will modify a cross-shaped grouping of tokens. Then again, the checkered board heuristic does not guarantee a solution. Similarly to the classic heuristic, there is no linear correlation between *h(n)* and *h*(n)*.

6 Work Distribution

As I worked on this project alone, I own all the design, implementation, and research involved in the completion of this project. Only the custom priority queue library was used from an existing reference (please refer to the repository's README).

References

1. Leila Kosseim, Russell S., Norvig P.: Annotated class slides - 472-2-search-Winter2020-annotated, [Class Moodle, robotics.stanford.edu/~latombe/cs121/2003/home.htm]. Last accessed 24 Feb 2020
2. Sorting HOW TO, https://docs.python.org/3/howto/sorting.html#key-functions. Last accessed 24 Feb 2020
3. Keep it Simple with the Strategy Design Pattern, https://blog.bitsrc.io/keep-it-simple-with-the-strategy-design-pattern-c36a14c985e9. Last accessed 24 Feb 2020
4. An object-oriented priority queue with updatable priorities. https://github.com/elplatt/python-priorityq. Last accessed 26 Feb 2020