COMP 472 Project 1: Indonesian Dot Puzzle Solving

Kevin Lin[1] and Sean Howard[2] and Yefei Xue[3]

[1] 40002383 sean.how0@gmail.com
[2] 26346685 Lin_Kevin_1995@hotmail.com
[3] 26433979 felixyf0124@gmail.com

1 Introduction

In this research project, we've come across on solving the DotPuzzle game which is a game with fairly simple rules. This game consists of solving a puzzle with various board sizes (e.g. 3x3) by touching a dot on the board. By touching a dot, the dot along with its adjacent dots will toggle colors between black and white. The goal is to turn every dot white or black depending on the player's preference (we chose white). In order to solve the game, we have developed many ways in order to find a solution as fast as possible through various computing search algorithms.

1.1 Technical Details

To solve the DotPuzzle, we first had to create a simple game engine using a Python 3.7 environment by writing the code using Visual Studio Code as our IDE. The algorithms used to solve our DotPuzzle were: depth-first search, best-first search, A* algorithm search. All three searching algorithms were written in Python with the help of the NumPy library.

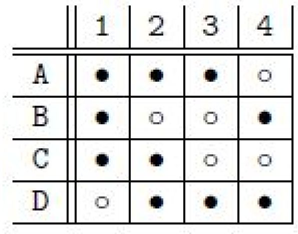
2 Heuristics

Due to time constraints, we decided to use the simplest heuristic.

2.1 How it works

Our heuristic counts the number of ●s (“1”s in code), and lower is better, with a final goal state $h_{goal}(n) = 0$.

For example, the heuristic of the state for **Fig. 1** is 10.



	1	2	3	4
A	●	●	●	○
B	●	○	○	●
C	●	●	○	○
D	○	●	●	●

Fig. 1. Example of simplest heuristic with h(n) = 10 for state “1110100111000111”

2.2 Strength

Since our heuristic algorithm is only counting the number of “1”s. the performance of calculating heuristic is at $O(n)$ while n is the size of the n dot puzzle (n * n).

Based on the requirement the max size of the dot puzzle is 10. Therefore, the max performance operation is $O_{max}(n) = 100$.

2.3 Weakness

The weakness of this heuristic algorithm is when it hits some edge cases, such a state only contains one “1”, e.g. “1000000000000000”.

The game rule of the dot puzzle is that it flips the touched dot as well as its top, left, right, and bottom ones into its opposite color, which flips 5 in total. Its edge case with a minimum number of total flip would happen at the corner of the board. E.g. touching A1 will flip A1, A2, and B1 with a total of 3 flipped tiles.

Therefore, once the search algorithm (Best-First Search) hits the state of $h(n) = 1$ or $h(n) = 2$, it will have no choice but go one step back to a state with higher heuristic value.

3 Difficulties

The difficulties encountered were mostly involved with the programming of the code and the analysis of the code runtime to trace the walkthrough of the traversal of the search paths of the search algorithms implemented. In all three search algorithms implemented, storing states was the highest of our concern and having to make it readable was an initial challenge on how to display the results and tracing the pathing of the algorithm in a readable fashion such as the close list being very huge.

One challenge was the optimization of the operation time in the Depth-first search as well as properly handling the backtrack for the state's depth level as it was initially not handled properly. This problem was solved by replacing the data type of the closed list (**list**) by a dictionary (**dict)** and storing the depth level as value to the associated state key.

Another issue that took some time to solve was the implementation of tracking the explored path of the tree since our initial game data structure is not using a connected tree. A refactoring was required in order to implement the best-first search and a-star search algorithms. The latter was solved by expanding the inherited traversed parent's "root-to-current" path with the child state. This method added one more sub-element inside the solution path of the elements in the open list. The aftermath of this solution helped the best-first search and A* search algorithm to backtrack accordingly so that the solution path could be computed without having the algorithm act up.

4 Analysis

4.1 Search in List, Set, and Dict

List is an ordered data type while the **set** and **dict** are unordered. The performance time complexity for querying for **list** is $O(n)$ while the performance time complexity for querying for **set** and **dict** is $O(1)$ [1] [2].

While developing searching algorithms, we have tried both ordered and unordered data types. During deliverable 1, we used **list** for both the closed list and the open list for the Depth-first search. It resulted a runtime of about 10 seconds for a 4x4 dot puzzle given sample initial state, and about 4 seconds for a 3x3 dot puzzle given sample with a CPU i7-9700k running at 4.9G Hz.

Since the closed list can expand extremely wide, we created a new Depth-First search with a copied-element **set** for the closed list. Also, in order to guarantee the solution output, we need to revisit some closed list if there exists a less deep level. Therefore, we replaced the **set** with **dict** (key for the state, and value for minimum depth).

The results are dramatically faster. With the new Depth-First search by using **dict**, it resulted in a runtime of about 0.7 sec for the 4x4 dot puzzle given sample initial state, and 0.07 sec for 3x3 dot puzzle with the given sample initial state with the same CPU.

In conclusion, we should always be aware of using **set** or **dict** for frequent searching on specific data from a large data pool.

4.2 DFS vs BFS vs A* Algorithm

Depth-First Search (DFS) with limited depth

Depth-First Search is not guaranteed for a solution, especially with an edge case shown in **Fig. 2**. Although we have filtered duplicated states from the open list (e.g. E in **Fig. 2**), there might be some state (e.g. F in **Fig. 2**) that gets added to the closed list before being added to the open list, which could cause a result of no solution at the end of the runtime.

In order to guarantee a solution being output, as mentioned in section 4.1, we used **dict** to hold both the state and its depth in a closed list as a solution. Thus, we could revisit the same closed state if and only if the newly expanded child nodes whose depth is less than the depth of those in the closed list(**dict** data type) and replace their minimum depth. By using this method, we guarantee a solution to be found if there is an actual solution within the provided limited depth.

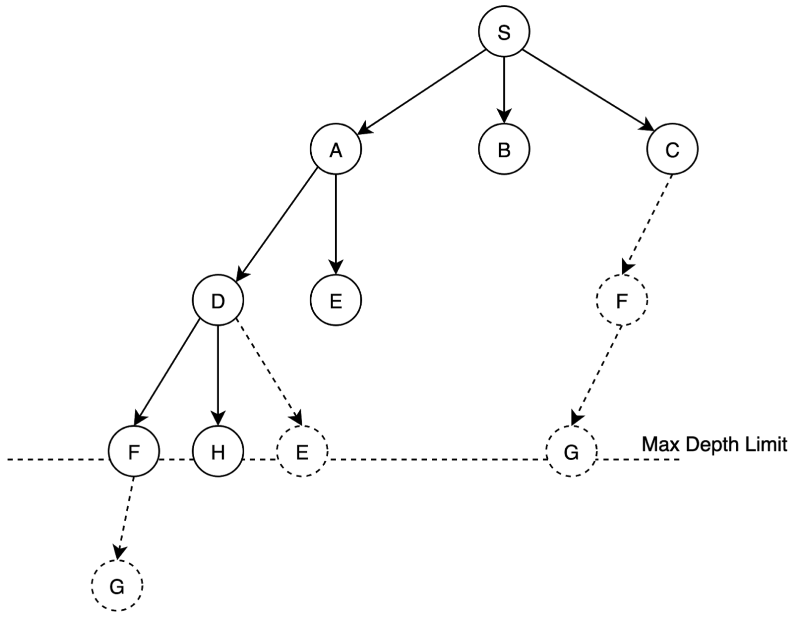


Fig. 2. Edge case for Depth-First Search

The operation of DFS could get terrible if the only approach path contains every last node in every depth.

The number of search path nodes is about 4k for the 4x4 dot puzzle given sample initial state resulted in about 675ms and about 700 for the 3x3 dot puzzle given sample initial state resulted in about 73.8ms.

Best-First Search (BFS)

Best-First Search uses only heuristic value $h(n)$. As mentioned a bit in section 2.3, it might hit wrong paths with edge cases like “100000000000000”, which slows down its operation speed. Thus, it does not guarantee the shortest solution path, and eventually, it will hit the final goal state with very high efficiency.

Overall, it has a very high efficient operating performance. It results in a runtime of about 5.98ms with 10 nodes searched for a 4x4 dot puzzle given sample initial state with a solution path of 10 nodes, and about 6.98 ms with 19 nodes searched for a 3x3 dot puzzle given sample initial state with a solution path of 18 nodes. Moreover, the higher percentage of the edge cases in the 3x3 dot puzzle is the reason which caused a longer operating time than the 4x4 dot puzzle runtime trial.

A* Algorithm Search

Similar to the Best-First Search, A* algorithm uses the heuristic value $h(n)$ but it also takes the cost $g(n)$ of root-to-node into account. It uses the total cost of $f(n) = h(n) + g(n)$ to sort the iteration priority of the nodes in the open list.

Unlike the Best-First Search using only the heuristic value, $f(n)$ can help A* to avoid continuing down a wrong path, maintain a certain efficiency while guaranteeing the shortest solution path being found as fast as possible.

A* algorithm search ends in about 16.95 ms with 29 nodes searched for the 4x4 dot puzzle given sample initial state with a solution path of 6 nodes, and about 17.02 ms with 63 nodes searched for the 3x3 dot puzzle given sample initial state with a solution path of 6 nodes

Conclusion

The depth-first search does not guarantee both efficiency and solution path; best-first search has overall very high efficiency with guaranteed solution path but it is possible to hit into a wrong path; A* algorithm maintains certain efficiency while guarantees the shortest path.

Table 1. Search Results from Given Dot Puzzle Sample Initial States

4x4 3x3	No. of Searched nodes	Length of Solution Path
DFS	3718 706	14 6
BFS	10 19	10 18
A*	29 63	6 6

5 Team Responsibilities

Kevin Lin	Code: - Worked on DFS with @Yefei Xue - Worked on BFS with @Yefei Xue - Worked on result output with @Sean Howard - Worked on main Doc: - Introduction - Difficulties with @Yefei Xue
Sean Howard	Code: - Worked on dot puzzle game logic with @Yefei Xue - Worked on input & game state import with @Yefei Xue - Worked on result output with @Kevin Lin Doc:
Yefei Xue	Code: - Worked on dot puzzle game logic with @Sean Howard - Worked on input & game state import with @Sean Howard - Worked on DFS with @Kevin Lin - Worked on BFS with @Kevin Lin - Worked on A* search Doc: - Heuristic - Difficulties with @Kevin Lin - Analysis Github repo management

References

1. Python: List vs Dictionary vs Set,
   https://stackoverflow.com/questions/52023758/python-list-vs-dictionary-vs-set
2. In Python, when to use a Dictionary, List or Set?
   https://stackoverflow.com/questions/3489071/in-python-when-to-use-a-dictionary-list
   -or-set/13003500