COMP 472 Project 1
Heuristic Search: Indonesian Dot Puzzle

Matteo Esposito[1], Matthew Liu[2], and Kabir Soni[3]

[1] 40024121 matteoesposito97@gmail.com
[2] 40029238 matthew.jx.liu@gmail.com
[3] 40033019 kabirsoni524@gmail.com

1 Introduction & Technical Details

This project was developed using python 3.7.4 64-bit (conda).

1.1 Files

The file structure of our project is as follows:

Table 1. Files in project 1

Directory	Filename	Usage
`out_dfs/`	*	Search and solution files for dfs
`out_bfs/`	*	Search and solution files for bfs
`out_a_star/`	*	Search and solution files for A^*
`src/`	`input_parser.py`	Input parsing functions, used to cast and convert input data into readable format
	`board.py`	Board class, used to represent the current puzzle
	`node.py`	Node class, used in all search algorithm and traversals
	`dfs.py`	Recursive Limited Depth-First Search algorithm implementation script
	`bfs.py`	Recursive Best-First Search algorithm implementation script
	`astar.py`	Recursive and iterative Algorithm A^* implementation script

1.2 Packages

We used a total of 7 packages in our project, 5 existing, along with our 2 internal packages (board and node).

1. Existing
   - `shutil` and `os`: Folder and file management in the creation and deletion of output folders for our search and solution files.

- `time`: Runtime output.
- `math`: Updating and calculation of h values (heuristic).
- `copy`: Used to create deep copies of parent node when creating child nodes in the `Board.touch()` method.

2. Internal
   - `board`: Class used to represent the puzzle
   - `node`: Class that is used to represent the nodes in the tree traversal of all puzzle states involved in the search algorithms.

1.3 Board and Node Classes

The board class will take as input the puzzle in the form of a nested list, where each sublist is a row in the dot puzzle provided. The puzzle is converted into a nested list with the help of the `parse()` method of the `input_parser.py` script. It will then allow the search algorithms to call the goal test function to allow the search to terminate, as well as the print grid and touch methods to allow for well formatted printing of the puzzle and child node generation.

The node class will store the current move associated with the resulting puzzle, the puzzle itself (attribute 'state'), the node's parent puzzle. In the non-dfs cases, attributes g, h and f will also be used (cost, heuristic and evaluation function values) to help with the informed searches.

2 Heuristic

Matthew and Kabir

3 Difficulties

Limited DFS Since this was the first algorithm to be implemented, typical difficulties that come with the start of a new project arose (such as project organisation & strong theoretical understanding of algorithms to be used). These were mainly design issues. Since object-oriented design was a priority, we had to spend some time figuring out how exactly to structure the classes that would be involved in the project. We eventually chose to create a class for the puzzle (Board) and for the nodes which would be later used in the tree-traversal that we end up implementing here.

Best-First Search blabla

Algorithm A^* blabla

4 Result & Experiment Analysis

Note: To ensure consistency, all 3 search algorithms were tested on the same 3 sets of inputs as follows:

```
2 4 100 1001
3 7 100 111001011
4 15 10 1010010111001010
```

4.1 Limited Depth-First Search

Table 2. Limited dfs runtimes as a function of the puzzle characteristics

Puzzle size	Max depth	Runtime	Average memory used
2x2	4	$\sim$ 0.5 seconds	36.5MiB
3x3	7	$\sim$ 31 seconds	36.5MiB
4x4	15	$\sim$ 52 seconds	36.5MiB

Compared to the other, informed search algorithms, we expect limited dfs to be the worst performer from a memory usage and time standpoint due to its uninformed nature. This search is also not complete, in that it is not guaranteed to find a solution. Since it is an exhaustive, recursive and stack based search, given branching factor b and depth limit l, we have a time complexity of $O(b^l)$ and a space complexity of $O(bl)$. [2]

Using the `memory-profiler` module, which monitors memory consumption of a process [1], we were able to generate memory usage charts of each of our serach algorithms. We notice that after the initial linear increase in memory usage (due to the I/O and parsing methods prior to the recursive dfs call), we have an average memory usage of 36.5MiB per call of our recursive serach algorithm. (See Appendix A)

4.2 Best-First Search

Matthew and Kabir

4.3 Algorithm A^*

Matthew and Kabir

5 Team Responsibilities

Due to there being three search algorithms that needed to be implemented and we are a group of three, we decided to take on an algorithm each. The breakdown of tasks were as follows:

1. Matteo Esposito
   - Input parsing functionality
   - Node & board class
   - Limited Depth-First Search
2. Kabir Soni
   - Best-First Search
3. Matthew Liu
   - Board class
   - Algorithm A^*

References

1. memory-profiler 0.57.0, https://pypi.org/project/memory-profiler/. Last accessed 28 Feb 2020
2. S.J. Russel, P. Norvig: Artificial Intelligence: A Modern Approach. 3rd edn. Pearson, Harlow (1994)

6 Appendices

6.1 Appendix A

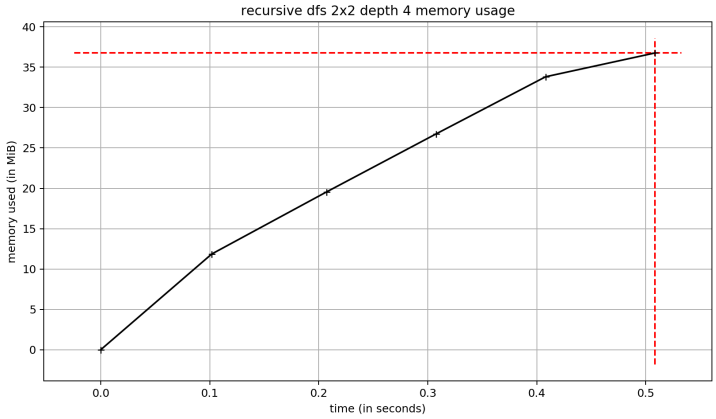

Fig. 1. Memory usage of limited dfs on a 2x2 puzzle

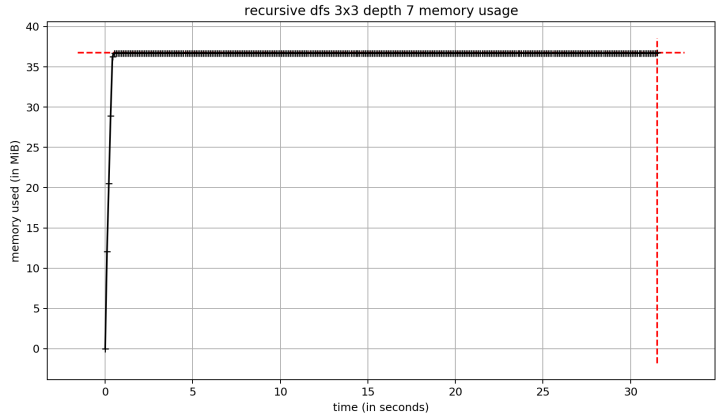

Fig. 2. Memory usage of limited dfs on a 3x3 puzzle

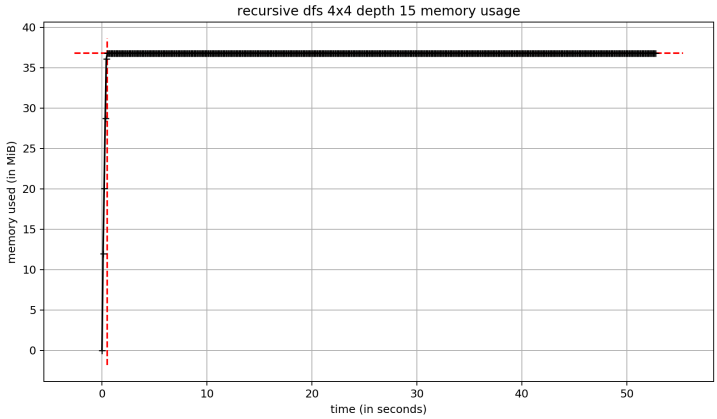

Fig. 3. Memory usage of limited dfs on a 4x4 puzzle