VE593: Problem Solving with AI Techniques
Project I

Simona Emilova Doneva (J11803099007)

October 22, 2018

1 Introduction

The general objectives of the project are to get familiar with the workings of prominent search algorithms and apply them to solve real-world problems. The tasks are split into two parts according to which the paper is structured. First, seven search algorithms are shortly introduced. Their implemented solution is tested on the popular N-Puzzle game and the results are shown and analyzed. In the second part, the reader is guided through the solution of the Clickomania game using similar concepts.

2 Assignment I

The goal for this part was to implement seven search algorithms and test their performance on a state-space graph based game, i.e. N-puzzle, also called sliding game [1]. Note that such games are not a graph-based problem, but can be thought of as a state space tree [1]. Table 1 provides an overview and main details over the implemented algorithms.

Algorithm	**Algorithm Family**	**Implementation Details**
Breadth-First Search (BFS)	Uninformed Search	Implemented FIFO logic with queue principle.
Uniform-Cost Search (UCS)		Extended BFS to first sort the successors based on their cost.
Depth-First Search (DFS)		Implemented LIFO logic with stack principle.
Depth-Limited Search (DLS)		Implemented recursive logic.
Iterative-Deepening Search (DLS)		Based on iterative executing of DLS with increasing depth limit.
A*	Informed Search	Implemented two heuristics: - UCS for general valued graphs - Manhattan Distance for N-Puzzle
Monte-Carlo Tree Search (MCTS)	Stochastic Search based on Random Walks	Used inverse of the cost to compute the reward.

Table 1: Implemented Algorithms Overview

For all algorithms a feasible execution time was obtain only for an 8-Puzzle problem (eight numbered, movable tiles set in a 3x3 frame). The results for the algorithm are shown in Figure 1.

[1]Please refer to `https://en.wikipedia.org/wiki/15_puzzle` for details on the game rules.

The provided metric is a rounded average over five executions of the game with randomly chosen initial state.

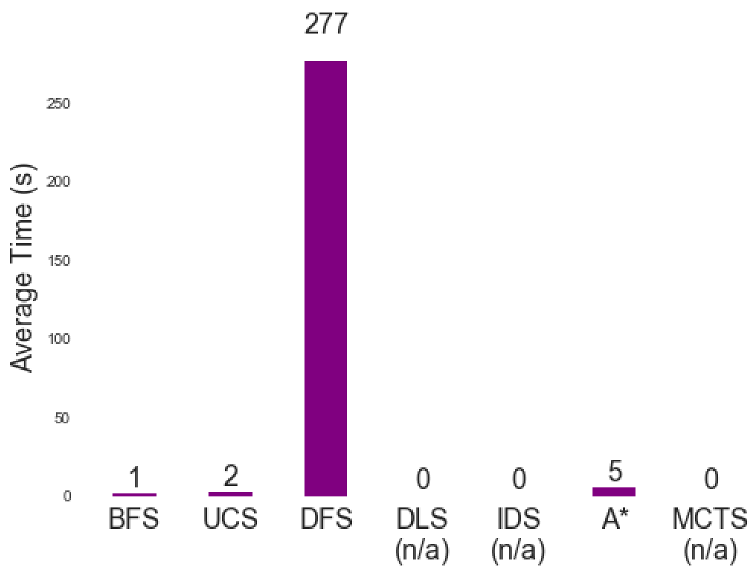

Figure 1: Average time performance for each algorithm across five executions.

It is evident that BFS and UCS have the best performance. This is due to their exploratory nature that allows to search through the tree without the risk of getting stuck going down the wrong path. The slightly higher time of UCS comes from the overhead of sorting the successor states. In addition to that, A* also computes the costs for reaching the given state, as well as the estimation of cost until the end goal is reached. The latter is performed by a corresponding Manhattan Heuristic. This explains why the algorithm is slightly slower.
The solutions which follow the depth of the search space show worse performance. Comparing the average length of the provided solution paths showed that DFS traverses many more states before reaching the goal[2]. This could indicate that exploring more alternative moves rather than exploiting all successors of a single move has a beneficial effects. Knowing that the average length (depth) of the provided solutions by the BFS-based algorithms did not exceed 25, the artificial depth cutoff threshold for DLS was set to this number. However, the algorithm was still not able to reach a valid solution in the given time budget. The behaviour of IDS is comparable to DLS with a slightly higher overhead in the number of generated nodes.
MCTS has several hyper-parameters that play a role for the performance of the algorithm. The general formula based on which successor nodes are chosen for visit is as follows:

$$argmax\frac{Q(v')}{n(v')} + \beta\sqrt{\frac{2\ln n(v)}{n(v')}}$$

Where n(v') and n(v) are the frequency of visiting the suggested state and its parent correspondingly. Q(v) computes the expected reward for exploiting node v'. The first component can be viewed as determinant of the quality of the state. The second component determines how likely less frequently visited states will be chosen for exploration. In the N-Puzzle setting, there is no clear concept for a reward. Therefore it was considered desirable to reward the shortest sequence

[2]Measured average path length for DFS is nearly 35,000. In comparison BFS returns a solution of average length 20.

leading to an end state the most. To achieve that the reward is computed as the reciprocal of the length from a random path found. By taking this approach the reward is scaled to a [0,1] range, which also allows to use $\frac{1}{\sqrt{2}}$ as an empirically derived value for the parameter β [2]. Unfortunately, the computational time for the algorithm was more than one hour even for a minimal number of simulations. Observations of the outputs showed that the algorithm incurred the highest costs while executing the roll-out, i.e. default policy for choosing random actions until an end state is reached. Increasing the number of N increased the complexity of the problem too much so that no further data could be gathered for evaluation.
There are several suggestions for improved computational time, which were out of scope for the project, but can be considered for future research. The most promising one is to divide the problem into independently solvable sub-goals and use distributed computing to solve them. This has shown beneficial to solve an N-Puzzle game up to a size of 399 tiles [3]. Furthermore Genetic Algorithms have been considered as suitable solution approach [1].

3 Assignment II

The second part deals with the implementation of the single-player game Clickomania, also known as SameGame. Relating the game to the state-space terminology [4], it can be described as follows:

- Initial state: $N \, x \, M$ grid filled with tiles of K different colors
- Goal state: grid with minimal number of tiles
- Path cost: instead of minimizing cost, the goal here is to maximize the final score. This includes maximizing the number of points achievable from a move, as well as minimizing the penalty received at the end for non-removable tiles. The reward from one move is dependent on the number of tiles removed.
- Successor: the grid state after the removal of one group (at least two) of adjacent same-colored blocks. All tiles above the removed group are shifted down. If there is an empty column, it is replaced by a non-empty column from the right.

The output from the game is the path from initial state to goal state along with the total score obtained. The first part of the solutions encompasses the encoding of the game according to the rules defined above. The grid of the game is encoded as an array filled with numbers from 1 to K. Table 2 and 3 provide an illustrative example.

2	3	2	2	3	1	1	2	3	3	3	3

Table 2: Initial state array view.

2	3	2	2
3	1	1	2
3	3	3	3

Table 3: Initial state grid view.

In Figure A.2 an overview of the logic to generate successor states is given. Which state to expand first is determined by the search strategy, which constitutes the second part of the solution.

```
Data: Initial grid state.
Result: List of successor states.
successors = [];
allColorGgroups = [];
foreach color k from K do
    indices < − getIndexPositions(k);
    kColorGroups < − groupAdjacentTiles(indices);
    allColorGgroups.append(kColorGroups)
end
foreach group in allColorGgroups do
    nextState, score < − computeNextState(initialState,
      group);
    successors.append((score, nextState));
end
return successors
```

Figure 2: Pseudocode for successor states generation.

Two approaches were considered for attacking the problem. First UCS as a greedy heuristic that chooses the maximum gain state sequentially. Secondly, MCTS as an alternative based on random walks.
The initial evaluation setup was to test the two algorithms in for a 3 x 4 board with 3 colours. The pre-set hyperparameters for MCTS were budget of 10 simulations, and exploration factor set to 0.5. The results over 1000 randomly generated states are shown in Table 4.

Algorithm	Min	Avg	Max
UCS	0	20.52	63
MCTS	0	18.7	63

Table 4: UCS vs. MCTS game score results over 1000 executions.

While the minimum and maximum scores achieved by the two approaches are the same, the average performance of MCTS is better. This can be explained by the fact that the short-term best solution might not lead to the globally optimal solution. One example is given by the previously introduced initial state (see Table 2). Applied to this setting, UCS would first select to remove the largest green group leading to the following sequence of rewards (16, 4, 1), i.e. 21 in total. MCTS on the other hand prefers to eliminate the yellow tiles first and thus be able to remove an even larger group of green blocks resulting in a final score of 30. This is due to the fact that MCTS enables the player to explore several alternative game sequences, that update the values of each move depending on the achieved end result. Based on those statistics it is more likely to choose successive moves that would lead to a globally optimal solution.

References

[1] Harsh Bhasin and Neha Singla. Genetic based algorithm for n-puzzle problem. *International Journal of Computer Applications*, 51(22), 2012.

[2] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton.

A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[3] Alex Drogoul and Christophe Dubreuil. A distributed approach to n-puzzle solving. In *Proceedings of the Distributed Artificial Intelligence Workshop*, 1993.

[4] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016.

A Readme

A short comment on the submitted code base.

A.1 Assignment I

Typing *python3.6 search.py* in a terminal will execute all seven algorithms with corresponding outputs on the example graph from the Project statement. The default settings for MCTS are $\beta = \frac{1}{\sqrt{2}}$, budget = 5.
Typing *python3.6 testtime.py* will execute the algorithms BFS, UCS, DFS and A* for the Npuzzle task, where n=3. The outputs from five iterations will be provided as well as final summary of the average performance. The other three algorithms have been commented out, but can also be tested. Please note that the execution will be interrupted automatically, if the algorithm runs for longer than 30 minutes. This time can be adjusted as desired. Also note that *testgraphs.py* has been extended with a valued version of the Npuzzle game, where the cost is set uniformly to one.

A.2 Assignment II

The initialization of a *Clikomania* object takes the parameters *M, N, K* (as described in the paper). The function *random_init_state()* from this object returns a filled array based on the configurations of the game. Those two variables can be used as input to the *clickomaniaPlayer* function. Example is given below:

```
from Clickomania import Clickomania;
game = Clickomania(3,4,3);
init_state = game.random_init _state();
import clickomaniaplayer as cp;
cp.clickomaniaPlayer(game,init_state);
```

MCTS will be applied to the game and the corresponding state sequence and final game results shown.