EARIN MINIPROJECT 1
OPTIMIZATION

Sotaro Suzuki

BRIDGES (HASHI)

Documentation file for EARIN project; searching algorithms for puzzle Hashi.

UNDER THE SUPERVISION OF

PAWEŁ ZAWISTOWSKI, PhD

INTRODUCTION

In the following documentation, implementation of two search algorithms for the game Hashi (Bridges) is described; **iterative deepening depth-first search** (**IDFS**) and **A***. The implementation was done in the Python language, and it works for game representation described in *Puzzle Space* section.

PUZZLE RULES

Game is played on square board of size *N*, with N^2 number of cells inside. The cells consisting of a number ranging from 1 to 8 (inclusive) are called **islands**. Those can be connected to each other with a single or double **bridge**, where each individual island has to finally have number of bridges connected to itself equal to its number value. Bridges cannot cross islands or each other and can only be placed in straight lines. The goal of the game is to connect all islands into a single group, maintaining the correct number of bridges for each island.

PUZZLE & SEARCH SPACE

PUZZLE SPACE

The *N* sized puzzle map is represented as *NxN* sized matrix (list of lists in Python implementation), where islands are implemented using their individual value, and bridges are shown as follows:

- 11 - one vertical bridge
- 12 - two vertical bridges
- 21 - one horizontal bridge
- 22 - two horizontal bridges

The example of representation is visible on Fig. 1.

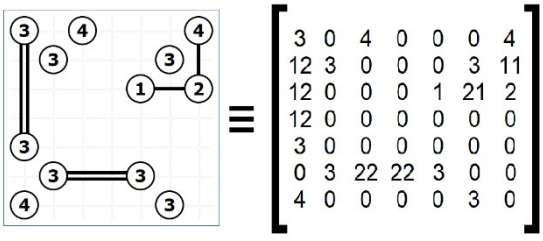

Fig. 1) Hashi map represented as a matrix

SEARCH SPACE

The puzzle's search space is implemented as a tree, where the root is the base map (without any bridges), and its children are every possible single bridge arrangement. This logic follows for next generations, where double bridge is counted as two individual bridges. The example of representation is visible on Fig. 2.

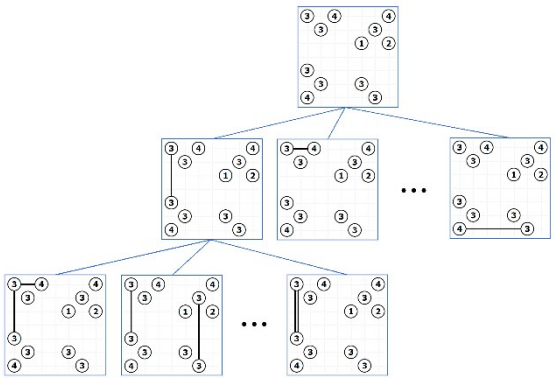

Fig. 2) Map's board represented as tree search space

Given the number of islands and its values, the maximal possible number of nodes n (for every *k* generation, each node has *k-1* children) in the tree can be evaluated as:

$$n = 1 + b_m + \sum_{m\ i=2}^{b_m} \frac{b_m!}{(b_m - i)!} \qquad (1)$$

where b_m is the maximal possible number of bridges in the map.

The maximal possible number of bridges b_m can be evaluated as:

$$b_m = \left(\sum_{j=1}^{k} i_j\right)/2 \qquad (2)$$

where i is the value of an island and *k* is the number of all islands on the board.

HEURISTICS

COST FUNCTION

The algorithm should rather connect islands with big values, rather than small ones, as, for example, connecting two ones will result in a dead end and unnecessary iterations. Therefore, parameter of ***board mass*** is introduced.

Defining bridge mass M_b as:

$$M_b = (i_p + i_q) * b \quad (3.1)$$

where i_p and i_q are the values of two islands, and b is number of bridges between them. Subsequently, the board mass B_m can be defined as:

$$B_m = \sum_{i=1}^{bc} M_{bi} \quad (3.2)$$

where b_c stands for current number of bridges in the board, and M_{bi} is the currently considered bridge.

Therefore, the cost function can be described as:

$$f_c = -B_m \quad (4)$$

HEURISTIC FUNCTION

Besides connecting the islands of biggest internal magnitude, the good tactics for the puzzle is to connect them into 'agglomerations', i.e. to seek such connections, that join islands which already have bridges, and form a longest possible path between those. Therefore, parameter of **board cohesion** is introduced.

Defining board cohesion C_b as:

$$C_b = k - x \quad (5)$$

Where k is the number of all islands on the board, and x is the number of islands forming the biggest path (agglomeration).

Combining the two, the objective function for the search space takes form:

$$f_c + 2 * C_b \rightarrow min \quad (6)$$

(Where board cohesion C_b is multiplied by 2 to take similar values to the cost function *fc*)

PERFORMANCE

The A* algorithm outperforms the IDFS drastically for bigger sizes of search space (more bridges). However, for trivial examples, the algorithm using IDFS search is more efficient.

The size of the puzzle does not influence the performance of the algorithms, thus only one case of this parameter has been considered (7 by 7 matrix). The results of the test are presented on Figures 3.1 and 3.2

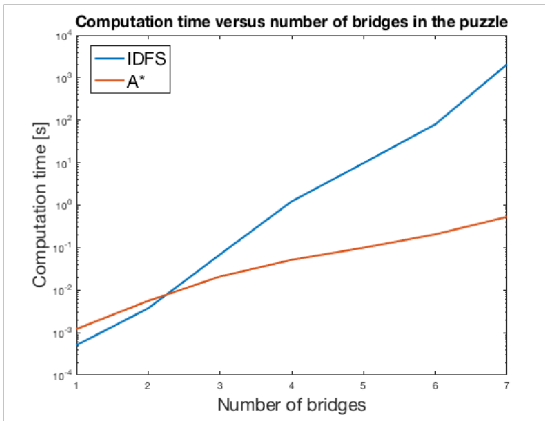

Fig. 3.1) Computation time to number of bridges relation on 7x7 sized map

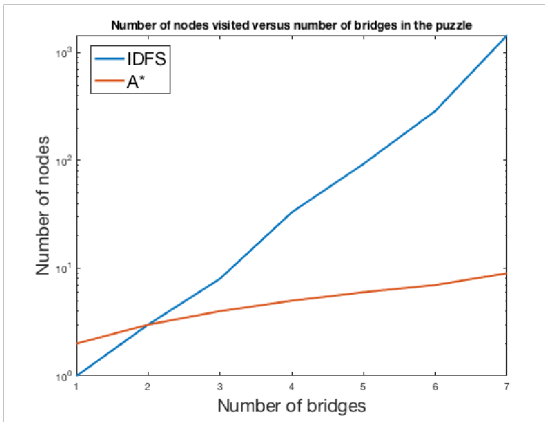

Fig. 3.2) Number of nodes visited by algorithm in the search space to number of bridges relation on 7x7 sized map

CONCLUSIONS

Both algorithms perform well for small search spaces, and the IDFS algorithm is more efficient for the most trivial puzzles, as there are no computations done for heuristics (the distance between the lines for small number of bridges is smaller on Fig. 3.1 than on 3.2). For bigger search spaces (depth of the tree bigger than 7), the IDFS algorithm becomes prohibitive, while the A* is still very efficient.

The implemented heuristics drastically improved the algorithm's performance for nontrivial search spaces, and without them the computation time would be too large to consider.

Detailed results of the simulations can be found in *results.txt* and *results_table.txt.* Full code for the algorithms can be found in *.py* files, and the MATLAB code for graphs in *plot_time.m*