UNDER THE SUPERVISION OF

PAWEŁ ZAWISTOWSKI, PhD

BRIDGES (HASHI)

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

earin miniproject 1  
OPTIMIZATION

ERNEST POKROPEK & KACPER WOJAKOWSKI

# introduction

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

# puzzle rules

Game is played on rectangular board of size *N*, with *N2* 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 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 each other, and can be placed only 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

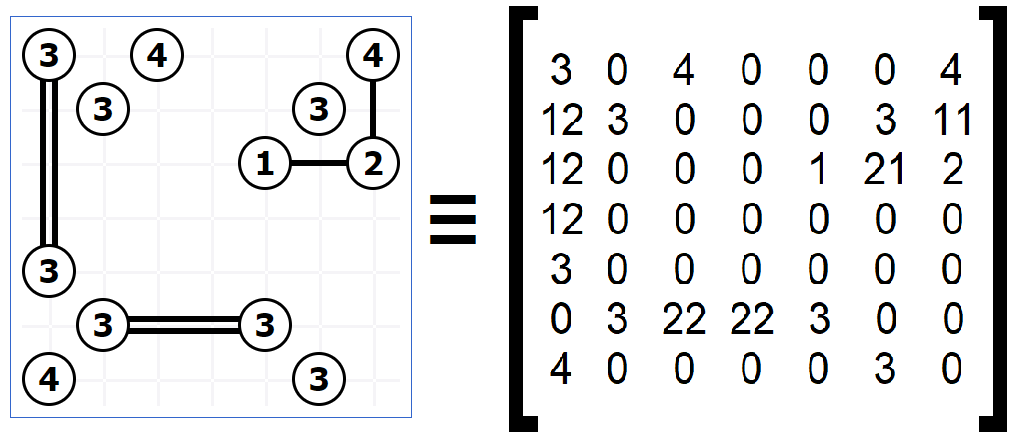
The example of representation is visible on Fig. 1.

Fig. 1) Hashi map represented as a matrix

### search space

The puzzle’s search space is implemented as 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.

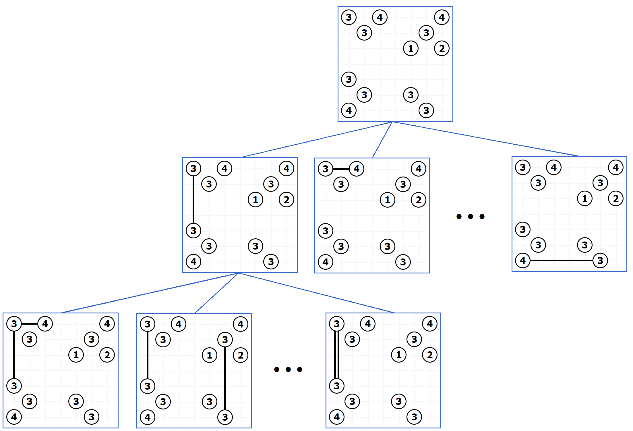


Fig. 2) Map’s board represented as tree search space

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:

|  |  |  |
| --- | --- | --- |
|  |  | (1) |

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

The maximal possible number of bridges *bm* can be evaluated as:

|  |  |  |
| --- | --- | --- |
|  |  | (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 dead end and unnecessary iterations. Therefore, parameter of ***board mass*** is introduced.

Defining bridge mass Mb as:

|  |  |  |
| --- | --- | --- |
|  |  | (3.1) |

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

|  |  |  |
| --- | --- | --- |
|  |  | (3.2) |

where bc stands for current number of bridges in the board, and is the currently considered bridge.

Therefore, the cost function can be described as:

|  |  |  |
| --- | --- | --- |
|  |  | (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 Cb as:

|  |  |  |
| --- | --- | --- |
|  |  | (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 complete heuristic function for the search space takes form:

|  |  |  |
| --- | --- | --- |
|  |  | (6) |

# performance

WYKRESY:

# conclusions