Atividade 2a

João Pedro Viveiros Franco
Mestrado Integrado em Engenharia Informática e Computação
Faculdade de Engenharia da Universidade do Porto
Porto, Portugal
up201604828@fe.up.pt

***Index Terms*—n-puzzle, artificial, intelligence, search, problems, a, star, heuristic**

I. Introduction

This document aims to explain the resolution of the ”n-puzzle” game using well-known search algorithms such as breadth-first search, depth-first search, iterative deepening depth-limited search, uniform-cost search, greedy search and the A* algorithm using the C++ programming language.

II. Problem description

The n-puzzle (more commonly known as 15-puzzle) problem is a well known puzzle consisting of a square grid with numbers from 1 to n and an empty space (also known as the number 0). The objective of the puzzle is to order these numbers by moving a piece to the empty space, effectively swapping the position of the empty space with the number.

III. Problem Formulation

For the reasons mentioned before the problem can be summarized in moving the empty space in 4 different directions: up, right, down and left, each costing 1 move. The chosen state of each node is a vector of vectors in C++, identical to a square matrix. As this is a search problem, one can represent the problem as traversing a graph in which each nod e consists of a state (the board matrix), a cost of operations, a heuristic value, a string representing the previous operation performed, and a parent node, to traverse back to the root node after finding a solution. The coords vector, representing the coordinates of the empty space, was saved in order to make operations more efficient, saving the trouble of finding the empty space before every operation.

Let x and y be the x and y coordinates of the empty space (the coords vector), accordingly, the problem can now be summarized with the following 4 operators:

Direction	Precondition	Effect
up	$y > 0$	swap(state[y][x], state[y-1][x])
right	$x < state.size() - 1$	swap(state[y][x], state[y][x+1])
down	$y < state.size() - 1$	swap(state[y][x], state[y+1][x])
left	$x > 0$	swap(state[y][x], state[y][x-1])

Each of the operations cost the same. In this implementation the cost chosen was 1.

```
class Node
{
public:
    vector<vector<int>> state;

    vector<int> coords;

    int cost = 0;
    double h = -1;
    double f;

    Node* parent = NULL;
    string operationName = "";
```

Fig. 1. The Node class.

```
typedef Node* (*Operator)(Node*);

extern std::vector<Operator> operations;
extern std::vector<std::string> operationNames;
```

Fig. 2. The operator typedef and the operators vector.

IV. Code

By using lambda functions for the operators, the programmer benefits from a simple and highly modular code, allowing it to be easily modified for different search problems. To simulate a different problem all that needs to be changed is the Node class, the operators and the mapLoader.

```
Operator up = [](Node* node) { Node* new
Operator right = [](Node* node) { Node*
Operator down = [](Node* node) { Node* n
Operator left = [](Node* node) { Node* n
```

Fig. 3. The Operators.

The Operator typedef allows a simplified type for a lambda function receiving a Node pointer as input and returning a Node pointer that represents the resulting state after the specific operation.

V. Results

The following data aims to summarise and compare the different results taken from executing the program. In order to remove irregularities in execution, these results were averaged from 5 consecutive runs. The depth limited algorithms were all limitid by maximum depth of 12.

Breadth	map	map2	map3	map4
Cost	4	7	10	10
Execution time	0.0002 s	0.0006 s	0.0024 s	0.0196 s
Memory spent	12.74 KB	41.48 KB	248.98 KB	2.39 MB

Depth	map	map2	map3	map4
Cost	10	7	12	10
Execution time	0.0092 s	0.0018 s	0.0016 s	0.0866 s
Memory spent	1.02 MB	186.24 KB	214.05 KB	11.11 MB

IDDFS	map	map2	map3	map4
Cost	4	7	10	10
Execution time	0.0002 s	0.0010 s	0.0055 s	0.0364 s
Memory spent	27.27 KB	101.16 KB	647.10 KB	4.93 MB

Uniform	map	map2	map3	map4
Cost	4	7	10	10
Execution time	0.0002 s	0.0007 s	0.0022 s	0.0203 s
Memory spent	13.37 KB	42.10 KB	221.48 KB	2.20 MB

Greedy (H1)	map	map2	map3	map4
Cost	4	7	12	Failed
Execution time	0.0001 s	0.0001 s	0.0003 s	0.0847 s
Memory spent	2.13 KB	3.99 KB	22.74 KB	4.69 MB

Greedy (H2)	map	map2	map3	map4
Cost	4	Failed	12	Failed
Execution time	0.0030 s	0.0058 s	0.0038 s	0.0850 s
Memory spent	199.00 KB	359.30 KB	257.11 KB	4.80 MB

A* (H1)	map	map2	map3	map4
Cost	4	7	10	10
Execution time	0.0002 s	0.0008 s	0.0026 s	0.0142 s
Memory spent	6.80 KB	27.43 KB	114.93 KB	496.80 KB

A* (H2)	map	map2	map3	map4
Cost	4	7	10	10
Execution time	0.0006 s	0.0016 s	0.0090 s	0.1241 s
Memory spent	22.44 KB	70.55 KB	397.43 KB	4.46 MB

Considering the previous data, the *tiles out of place* (H1) heuristic provides a much faster solution than the *manhattan distance* (H2) heuristic, both for Greedy search as well as the A* algorithm. Using H1 as the heuristic makes greedy search find a solution faster than A*, but has the opposite effect when using H2, which is to be excepted as greedy search has to backtrack when reaching the limit, so it can be concluded that using a bad heuristic has very adverse effects on this non-optimal algorithm.