Trabalho 1: Searching Algorithms

Marc Clascà Ramírez Joaquim Ferrer Sagarra

March 2022

1 Introduction

This assignment is based on experiments regarding searching problems and different approaches to solve them. These problems are based in a solution that can be found in a set of different states, linked between them. Our goal in this kind of problems is search in all possible states the one that represents the solution. In order to do that, all states are represented in a tree structure and the goal is, in this tree, search for the optimal path from the root to the problem solution[1, Section 3].

A solution is an action sequence, so search algorithms work by considering various possible action sequences. The possible action sequences starting at the initial state form a search tree with the initial state at the root; the branches are actions and the nodes correspond to states in the state space of the problem[1, Section 3.3].

2 Problem: Jogo dos 15

The *game of 15* is a puzzle based on a 4x4 square matrix filled with numbers from 1 to 15 and one white space. This space have 4 movements into the matrix (left, right, up, down) that can be done swapping the blank space and the number located in the desired position. The goal is to, given an initial configuration, move the blank space until reach a final configuration, considered the solution. Not all initial configuration can be transformed to a given final configuration so this puzzle is not always solvable. Is possible to know if a configuration is solvable using the formula below (1) were Inv_i are the inversions of the initial configuration, Inv_f the inversions of the final configuration, $blankRow_i$ the row containing the white space in the initial configuration and $blankRow_f$ in the final.

$$(Inv_i\%2 == 0) == (blankRow_i\%2 == 1) == (Inv_f\%2 == 0) == (blankRow_f\%2 == 1) \quad (1)$$

To solve the *Game of 15*, once checked it is solvable, the strategy will be building a tree structure with all possible configurations starting from the initial. Each node in the tree will be a possible state of the matrix and it's leafs all the configurations resulting from the four possible movements of the blank space. This tree will be generated and explored using 4 different algorithms: DFS, BFS, IDFS, Greedy Search and A*.

3 Searching strategies

The searching strategies will be those general-purpose search algorithms that can be used to solve these problems. These will be described in the sections below.

3.1 Uninformed strategies

The uninformed strategies are those algorithms that are given no information about the problem other than its definition [1, p.64].

Depth First Search DFS by its acronym, is a strategy that always expands the deepest node in the current frontier of the search tree exploring first the deepest level of the tree. This algorithm is based on a LIFO data structure to explore the tree. This structure stores nodes and returns first the last one that was included. This data structure gives the possibility to explore always the deepest node stored in the stack. This algorithm is considered non-optimal. If the solution from a given configuration A is C and this can be reached through different paths, DFS will return the path found exploring the deepest nodes, not taking in account that might be a shortest path in one of the middle-nodes. The time complexity of depth-first graph search is bounded by the size of the state space so, is $O(b^m)$ where m is the maximum depth and b the number of branches. Is important to take in account that if m is infinite the algorithm will never end. The goal of this algorithm is the spacial complexity. With branching factor b and maximum depth m, depth-first search requires storage of only $O(bm)$ nodes [1, Section 3.4.3].

Breadth First Search BFS by its acronym, is a strategy where the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. This algorithm is based on a FIFO data structure to explore the tree. Thus structure stores the nodes and return first the last one that was stored in the queue. This data structure gives the possibility to explore always the shallower nodes. The solution given by BFS is always the optimal in terms of the path longitude. Despite that, time complexity is $O(b^d)$ where d is the solution depth and the space complexity is also $O(b^d)$ because it needs to visit all nodes in all depth levels, much more than DFS!! [1, Section 3.4.1]

Iterative depending Depth First Search IDFS by its acronym. This strategy aims to combine DFS and BFS to get a better time and spacial complexity. It does this by gradually increasing the depth limit (first 0, then 1, then 2, and so on) until a goal is found. This allows to find the optimal solutions with modest memory requirements, $O(bm)$ with a time complexity of $O(b^d)$, same as BFS [1, Section 3.4.5].

3.2 Informed Strategies

Informed Strategies are those that uses problem-specific knowledge beyond the definition of the problem itself. These strategies can find solutions more efficiently than can an uninformed strategy.

Greedy Search This strategy tries to expand the node closer to the goal using an heuristic function to determine the distance. The algorithm only takes in account the result of the heuristic, hence the solution may not be optimal. This happens because is not taking in account the depth of the solution so it has the same problem discussed when talking about DFS. The worst-case time and space complexity for the tree version is $O(bm)$, where m is the maximum depth of the search space. With a good heuristic function, however, the complexity can be reduced substantially [1, Section 3.5.1].

A* This strategy improves the Greedy Search by taking in account the depth of the nodes. It evaluates nodes, given a node n, by combining $g(n)$, the cost to reach the node, and $h(n)$ being the heuristic:

$$f(n) = g(n) + h(n) \tag{2}$$

The resulting $f(n)$ will be the considered cost. In order to find the best solution, A* treats first the nodes with the lower cost with a temporal and spacial complexity of $O(b^m)$ and is able top find the optimal solution. Its efficiency will depend directly to the heuristic [1, Section 3.5.2].

3.2.1 Heuristics

A heuristic function, denoted as $h(n)$ is a function that gives the estimated cost of the cheapest path from the state at node n to a goal state[1, p.92]. In this assignment we have been using two heuristics:

Pieces out of place This heuristic counts the number of pieces that are not in the correct place regarding the solution. So, it gives a number out of 16 wich will be the cost assigned to each possible configuration.

Manhattan distance This heuristic counts the number of movements needed for each element of the matrix to be in the correct place. The cost will be the sum of this distance for all the pieces in the configuration.

We used the pieces out of place heuristic in our tests because, experimentally, gave us better results.

4 Our Implementation

Programming language and runtime We chose C++ as the programming language for our program. We think it is a good choice because it is a compiled language, which gives better performance compared to interpreted languages (like Python) or languages that run over virtual machines (like Java). C++ allows compiling the source code to a binary that makes use of the features of the processor, like vector instructions, allowing fast computing of processes like large math operations. For Artificial Intelligence applications it is very important to be able to fine tune the program and the code, to use only the instructions needed and get a good time efficiency. In terms of spatial efficiency, C++ also allows to control better the memory usage and create data structures with the exact size needed. This is important when a large number of the same data structure is created and needed to work, like with nodes in this case. C++ allows fine tuning the allocation of dynamic memory, with the drawback of a more complicated programming, compared to other languages like Python.

Data structures The main data structures involved in problem solving are the following:

- *Tabuleiro* This class contains a specific state of the table. Because it will be created to represent the different states during the seach, its size has to be reduced. For this reason, it only contains an array of 16 bytes, to represent the number of the table (from 0 to 15) in the order they appear. This allows an instance of tabuleiro to only occupy 16 bytes in memory. This class contains methods to compare with another tabuleiro, consult the data, know where the blank space is, and print the table in different formats (list of numbers or formatted as a matrix). It also contains the methods to instantiate the new states that can be derived of moving the blank space.

- *Nó* This class represents a node of the tree, containing one specific state (a reference to an instance of the class `tabuleiro`. Each node contains a list of at most 4 children, that is, 4 references to other nodes, and a reference to a parent node. With 2 other integers, it finally adds up a total of 56 bytes only. As with the `tabuleiro` class, a lot of instances of node will be created. The node class contains methods to track the path back to the root node, and to generate the children.

Code structure Each game of the Jogo do 15 is represented with the class `jogo`. This class contains one initial configuration and one final configuration, therefore one game to solve. It also contains an algorithm to solve the game.

This class then starts the search for the given configuration pair and with the given algorithm.

The search method created a new instance of the `Algorithm` class. This class has polymorphism, meaning that we implemented a class for each algorithm that inherits from this general `Algorithm` class. This allows us to define abstract methods and then do custom implementations for each algorithm.

The `Jogo` class does the general search, calling the methods of the specific algorithm until a solution is found. When a solution is found, it prints out the path to that solution, with the statistics collected. To avoid endless running of the algorithms and a possible fill of the main memory of the computer, we implemented a timeout mechanism using the SIGALRM signal.

5 Results

In this section, we show and evaluate the results of solving one configuration with the 5 algorithms. For algorithm, we recorded the time, space used (in terms of number of nodes and in memory used), if could find a solution, and the depth of this solution.

Moreover, we executed the program with two different computer configurations:

1. **Personal laptop** Laptop with 4th Generation Intel Core i5 with 2 cores at 2.60GHz with 8GB of main memory. Running MacOS operating system. Compiled with Apple Clang v12.0.5.
2. **HPC Server** High Performance Computing server with 2 sockets Intel Xeon Platinum 8160 CPU with 24 cores each, at 2.10GHz, with 96GB of main memory. Running SuSE Linux Enterprise Server as operating system. Program compiled using Intel C++ Compiler version 17.0.4.

The timeout mechanism has been set to 30 seconds. The informed algorithms use the pieces out of place heuristic. The measurement of memory is computed for the nodes stored, as $\#nodes \times sizeof(nodeclass) \times sizeof(tabuleiroclass)$.

Configuration C2 The configuration that we executes is configuration C2:

```
Initial table: 9 12 13 7 0 14 5 2 6 1 4 8 10 15 3 11
Final table: 9 5 12 7 14 13 0 8 1 3 2 4 6 10 15 11
```

The results using our personal laptop can be seen in table 1 and the results of the execution in the HPC server can be seen in table 2.

Strategy	Time (ms)	Number of nodes	Memory (B)	Solution	Depth
DFS	Timeout	12006912	864497664	No	
BFS	351.287	973406	70085232	Yes	16
IDFS	218.284	6724	484128	Yes	16
Gulosa	Timeout	10227298	736365456	No	
A*	3.448	87	6264	Yes	16

Table 1: Execution of configuration C2 with Personal Laptop

Strategy	Time (ms)	Number of nodes	Memory (B)	Solution	Depth
DFS	Timeout	1053131	75825432	No	
BFS	559.077	973406	70085232	Yes	16
IDFS	238.264	6724	484128	Yes	16
Gulosa	Timeout	2534496	182483712	No	
A*	599.06	245	17640	Yes	16

Table 2: Execution of configuration C2 with HPC Server

We can see that the configuration could not be solved by the *Gulosa* algorithm. This hints us that maybe there is a problem with the implementation of the algorithm. The algorithm DFS also could not solve the configuration within the time. This has to do with the fact that with jogo do 15 most of the configurations have a solution not very deep.

6 Conclusions

This assignment gave us the chance to test and practice some of the strategies applied in searching problems. With it, we could reinforce our learning in theory lessons with an experimental lab.

Regarding the results, we could see DFS is indeed not the best option for this kind of problems. It only could solve a very poor number of configurations due to he huge depth of the tree. BFS instead, gave great results solving the game. It is more efficient to solve the *game of 15* using a BFS. We also realized the space used by BFS is much more than the other algorithms, thing solved, in most of the cases, by IDFS with out increasing the time used.

Informed strategies gave us different results. Greedy search is not performing always the same and don't give optimal solutions. It does contrast with A* that is mostly always giving the best time and always the optimal solution.

Also, working with C++ with this large amount of memory gave us a lot of struggles to solve. We had many problems managing memory and optimizing

code to make the algorithms finish. Small changes in the code affected completely the performance of the program and was also a learning the hours we spent profiling and analyzing the code performance to make it more efficient.

References

[1] Russell, S. J., Norvig, P. (1995). Artificial intelligence: A modern approach. Englewood Cliffs, N.J: Prentice Hall.

[2] Horatiuvlad.com. 2022. Solvability of the Tiles Game. [online] Available at: https://horatiuvlad.com/unitbv/inteligenta_artificiala/2015/TilesSolvability.html [Accessed 24 March 2022].