COMP6231: Basic Search Methods

Julio Cesar Aguilar Jimenez[1,2,3]

[1] *School of Electronics and Computer Science, University of Southampton*
[2] *MSc. Artificial Intelligence, ID. 29312175*
[3] *Author email: jcaj1n17@soton.ac.uk*

January 9, 2018

The results of applying different search algorithms (DFS, BFS, IDS, A*) for the resolution of a problem "Blocks World Tile Puzzle") are presented. The first approach is a "Tree Search" type being very unfavourable for this problem and failing in the majority of cases even without increasing the difficulty of the same. The second approach is an adaptation of the algorithms in order to storage the visited nodes, which converts them into a "Graph Search" type. This greatly improves the performance and allows to find optimal solutions to the problem in such a way that this scale.

Nikola Tesla: 'I have not failed. I've just found 10,000 ways that won't work.'

1. INTRODUCTION

AI research is defined as the study of *Intelligent Agents*. Any device that perceives its *environment* and takes *actions* that maximise its chance of *success* at some *goal*.

In this paper we present 4 different *search methods* that solve a defined problem, they start in a defined *initial state* and perform a search until reaching the desired *final state*.

Search algorithms allow us to find solutions to problems, allowing *machines* the ability to compare *states* and analyse possible situations to reach the desired state.

2. APPROACH

One of the crucial points to correctly perform the *algorithms* is to have **data structures** that fit the needs of each *algorithm*.

The diversity of **data structures** helps the *developer* to focus on the flow and *strategy* of the problem and not on the *data management* by the *compiler*.

2.1. Language

The languages with more *time* of development and resources have this type of *structures* and that is why the **Java** language was chosen for this work.

2.2. Problem

We will examine the game **Blocks World Tile Puzzle**, the *board* is a matrix of tiles of dimension *NxN* where there will be four special tiles, called blocks (**A, B, C, Agent**).

The *objective* of the game is to group these three blocks in a *tower* so that **A** is above **B** and this one of **C**. The **Agent** block will move through the board changing its position with the tile or block in the direction that it moves, it *respects* the *edges* of the board.

The **initial** and **final** *states* are given at the beginning of the game, the blocks (**A, B, C**) always **start** in the *lower left corner* and the **Agent** in the *lower right corner*. The **final** state is a *tower*, where the last block **C** is at the edge of the board and a place moved to the *right* of the *lower left corner*, the other two blocks are *above* it, where **A** is at top and the **Agent** block should be in the *lower right corner*. The following *matrices* illustrate the foregoing.

$$\underset{}{\overset{\boldsymbol{4x4 - Initial}}{\begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \boldsymbol{A} & \boldsymbol{B} & \boldsymbol{C} & \boldsymbol{Ag} \end{bmatrix}}} \rightarrow \overset{\boldsymbol{4x4 - Goal}}{\begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \boldsymbol{A} & \bullet & \bullet \\ \bullet & \boldsymbol{B} & \bullet & \bullet \\ \bullet & \boldsymbol{C} & \bullet & \boldsymbol{Ag} \end{bmatrix}}$$

$$\overset{\textbf{\textit{5x5 – Initial}}}{\begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ A & B & C & \bullet & Ag \end{bmatrix}} \rightarrow \overset{\textbf{\textit{5x5 – Goal}}}{\begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & A & \bullet & \bullet & \bullet \\ \bullet & B & \bullet & \bullet & \bullet \\ \bullet & C & \bullet & \bullet & Ag \end{bmatrix}}$$

2.3. Breadth-First Search (BFS)

For this method a **FIFO** type buffer (*First In, First Out*) is necessary. *Java* has the **LinkedList** subclass that emulates this behaviour and will contain the *nodes* to analyse. The **HashSet** class provides a *structure* that stores *unique* elements, making it ideal for keeping track of *visited nodes*.

2.3.1. Operation

The first step is to **add** the initial *node* to the *lists* or *structures*, then a **loop** will be carried out until the list is *empty*, we will obtain the *node* at the *top* and **validate** if its state is equal to the goal, if it is, the console prints the *result*. *Otherwise* we create a list of nodes with the **possible movements** from the current position of the agent and again validate if this node was already visited, if it was, it's ignored and otherwise it will be *added* to both structures.

It will continue in the *loop* until it finds a result or otherwise it will return a *null* indicating an *error* in the search.

2.4. Depth-First Search (DFS)

For this method, a **LIFO** type buffer (*Last In, First Out*) is necessary. *Java* has the **Stack** class which provides this behaviour in our array or list. The *operation* is the same as that shown in Section 2.3.1.

2.5. Heuristic Search A*

For the *heuristic search* it is necessary to *add* another parameter to our nodes, this is the **cost** from that *node* to the *final node*.

The **cost** is calculated by adding the *Manhattan distances* of the blocks (**A, B, C**) to the final *coordinates* of the respective block. It is important to mention that since the **Agent** must move freely on the board this should not be considered in the *calculation* of the cost.

The *operation* is almost the same as previously described in Section 2.3.1, with the difference that when validating the *possible movements* of a *node*, if this *node* has not been visited, its **cost** must be calculated *before* being added to both structures.

2.6. Iterative Deepening Search

This method is *born* from **DFS** and it is necessary to *restrict* its search to a maximum **depth**, therefore it is necessary to *save* the **depth**. The **HashMap** class will allow us to *record* both the *node* and its **depth**.

2.6.1. Operation

We make use of the *recursion* by means of a *secondary method* that will do the *searches* and it will be constantly *calling*. Once it I the result it will call the *main* method to notify it and *print* results.

In the first instance, we initialise our **depth** to 1 and create a *null node* that will be the *flag* that will indicate when the *algorithm* find the result.

We will *loop* the execution of the search method increasing the **depth** in each *iteration* and giving it as parameters the *nodes* with the **initial** and **final** state.

The search works in a similar way to Section 2.3.1, only now we will use the **HashMap** structure to *store* the *node* and the **depth** in which it was visited. When analysing the *possible movements*, we must validate the **depth** and in case of not having been visited that *node*, it is stored with its respective **depth**.

3. STRUCTURE OF THE APPLICATION

The *application* was developed in *IntelliJ Idea* environment. It has 5 classes and *one* enumeration file. Due to the nature of *Java*, it is necessary to create the *enums* in separate files.

3.1. State and Node Classes

The **State Class** is a *representation* of a *physical* configuration. It contains methods to create a *temporary board*, set the *positions* of the blocks and print them in console, *compare* the current *state* with someone else (the goal state) and *validations* to move the agent in the *different* directions.

The **Node Class** is a *data structure* and contains the properties of *parent*, *state*, *path cost*, *depth* and *direction*. It has the method to calculate the **cost** in the *heuristic* search by *Manhattan distance*, 2 types of *constructors* and the function that calculates the *possible movements*.

This design was chosen taking as reference the previously seen in class, each *node* stores its own *state*. [1]

3.2. Block and Board Class

The **Block Class** will allow us to create both the tiles and blocks **A, B, C** and **Agent**. It has the *name* and the *coordinates* in X and Y as properties. It is important to mention that the tiles or *empty* blocks will have the **title** of *tile* while the blocks will be called *A, B, C, Ag* respectively.

The **Board Class** has as properties the **initial** and **final** *nodes*, it has the functions that are responsible for executing the *search* methods.

3.3. Main Class

The **Main Class** that executes the others classes, contains **flags** for the *search methods* that will be changed to *false* if some *error* occurs such as *running out of memory*, in this way the *application* will continue with the process.

Its functions are to create boards of size $4x4$ up to $20x20$, set the **initial** and **final** *states*, and finally execute the *search methods* on each board.

4. EVIDENCE

Below are the impressions in console of the different tests that were made. Due to the size of DFS method, the complete output for the *Graph Search* is appended in Section **??**.

```
TREE SEARCH APPROACH

*****  SEARCHING IN 4x4 BOARD  *****
DFS search failed for 4x4 board. Out of memory error
BFS search failed for 4x4 board. Out of memory error

- IDS
Moves: 17726954
Time: 45473.0
Depth: 16
Path: [←, ←, ←, ↑, →, ↓, →, →, ↑, ←, ↑, ←, ↓, ↓, →, →]


- A*
Moves: 110717.0
Time: 1358.0
Depth: 16
Path: [←, ←, ←, ↑, →, ↓, →, →, ↑, ↑, ←, ←, ↓, ↓, →, →]

*****  SEARCHING IN 5x5 BOARD  *****
HEUR search failed for 5x5 board. Out of memory error

-----------------------------------------------------------

EXTRA-GRAPH SEARCH APPROACH

*****  SEARCHING IN 4x4 BOARD  *****
- DFS
Moves: 9062
Time: 179.0
Depth: 4758
Path: [←, ←, ←, ↑, →, →, →, ↓, ←, ←, ←, ↑, ↑, →, →, →...]

- BFS
Moves: 4969
Time: 61.0
Depth: 16
Path: [↑, ←, ↓, ←, ←, ↑, →, ↓, →, ↑, ↑, ←, ↓, ↓, →, →]

- IDS
Moves: 10569
Time: 163.0
Depth: 16
Path: [←, ←, ←, ↑, →, ↓, →, →, ↑, ←, ↑, ←, ↓, ↓, →, →]

- A*
Moves: 561
Time: 31.0
Depth: 16
Path: [←, ←, ←, ↑, →, ↓, →, →, ↑, ↑, ←, ←, ↓, ↓, →, →]

*****  SEARCHING IN 5x5 BOARD  *****

- DFS
Moves: 202616
Time: 3650.0
Depth: 93892
Path:
[←, ←, ←, ←, ↑, →, →, →, →, ↓, ←, ←, ←, ←, ↑, ↑, →, →...]


- BFS
Moves: 10648
Time: 54.0
Depth: 18
Path:
[↑, ←, ←, ↓, ←, ←, ↑, →, ↓, →, ↑, ↑, ←, ↓, ↓, →, →, →]


- IDS
Moves: 22645
Time: 328.0
Depth: 18
Path:
[←, ←, ←, ←, ↑, →, ↓, →, →, ↑, ←, ↑, ←, ↓, ↓, →, →, →]


- A*
Moves: 1254
Time: 19.0
Depth: 18
Path:
[←, ←, ←, ←, ↑, →, ↓, →, →, ↑, ↑, ←, ←, ↓, ↓, →, →, →]
```

5. RESULTS

The application was tested on a *MacBook Pro* (13-inch, 2017), with a *2.3* GHz Intel Core *i5* processor, 8 GB 2133 MHz LPDDR3 memory and *solid state hard drive.*

5.1. Tree Search

The use of **DFS** and **BFS** proved to be *unable* to solve the problem for the *minimum* size ($4x4$), these algorithms **expand** *nodes* very *fast* in a *short* amount of **time** causing *system resources* to run out and an *error* occurs.

For the case of **A***, the algorithm is *able* to solve the problem for a size of $4x4$ but not $5x5$, it ends up *exhausting* the *resources* of the *system.*

The **IDS** algorithm proved to *solve* the problem for sizes $4x4$ and $5x5$, however the **time** it takes is **long**. The Table 1 and the Figures 5.1, 5.2 show the results obtained.

Table 1. *Results obtained in different methods*

Method	Nodes	Size	Success
DFS	$1,367,918$	4	No
BFS	$1,034,903$	4	No
IDS	$17,726,954$	4	Yes
	$403,168,687$	5	Yes
$A*$	$110,717$	4	Yes
	$680,759$	5	No

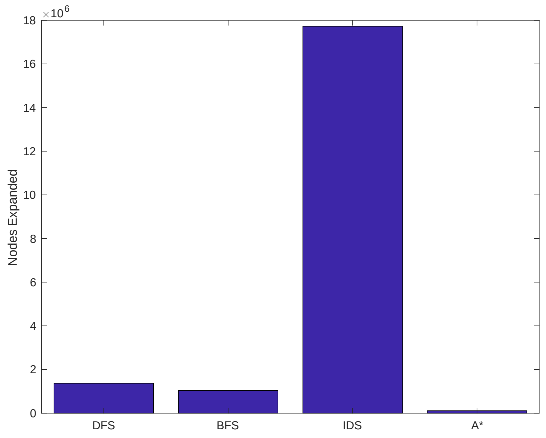

Fig. 5.1. Nodes expanded in 4x4 board

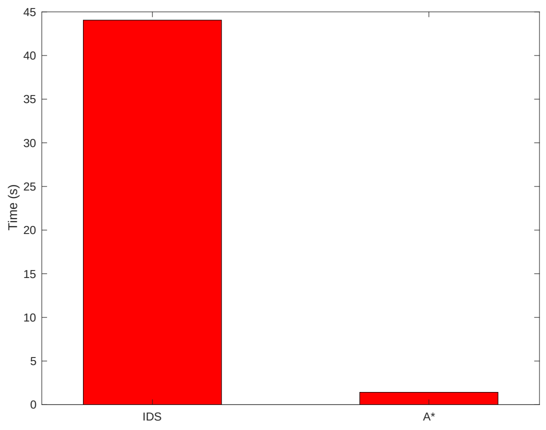

Fig. 5.2. Time to solve the problem in 4x4 board

5.2. Extras (Graph Search)

The algorithms were *adapted*. A record of the **visited nodes** was *added*, which greatly **improvised** the *performance* of these. It is an approach oriented to the *Graph Search* unlike the previously seen *Tree*, where a *state* can be visited an *infinity* of *times*.

The code was made in a scalable way creating boards of dimensions $4x4$ to $20x20$, the results printed on the console were collected and processed in MATLAB. The Figures 5.3, 5.4 show the results obtained.

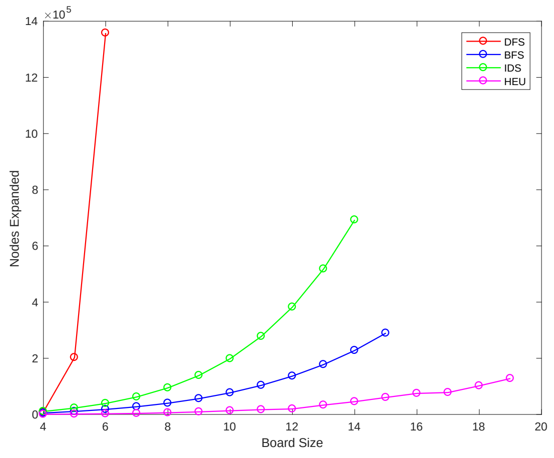

Fig. 5.3. Nodes expanded in different sizes

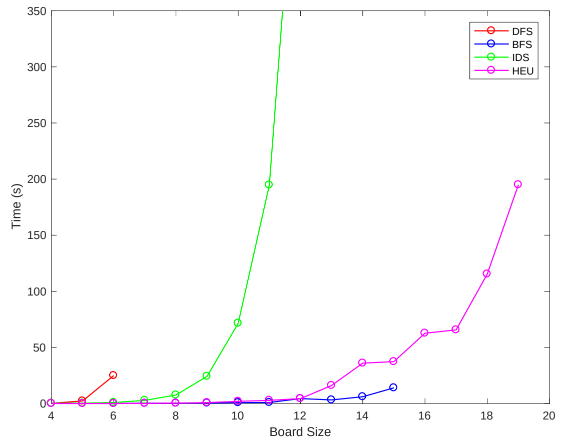

Fig. 5.4. Time to solve the problem in different sizes

5.3. DFS

This method had the **poorest** *performance* compared to the others, the *complexity* of the *time* and *space* **increase sharply** as the difficulty of the problem *increases*.

DFS is able to find the solution to the problem for *small* dimensions of the board $(4x4 - 6x6)$, however the solution is **far** from **optimal**. Given the amount of resources it needs and its poor performance, this search method is *inefficient* for this problem.

5.4. BFS

This method has generally shown *good* performance, the *complexity* of *time* and *space* is *good*, and the solution is **optimal**. Even so, this method took too many resources and failed when the board reached a size of $16x16$.

5.5. IDS

No doubt **IDS** is better than **DFS**, the solution in this method is **optimal**. However, the complexity of *space* and *time* continue to be *very large*, causing it to fail once the board reaches the dimensions of $15x15$. In addition, the *time* it takes is much **longer** than **BFS** and **A***.

5.6. A*

The use of *heuristics* is the one that showed **great** *performance* in both *time* and *space* complexity and the response is **optimal** (at least for this problem). This method **depends** to a large extent on the *heuristic used*, therefore it can *increase* or *decrease* its *performance* depending on the **heuristic estimate** used.

6. SCALABILITY

The code was made so that the problem could *easily* scale, with the creation of $4x4$ boards up to $20x20$ and test each of the *methods* on each *board*. When the methods *finish* their search in the current board, a new board whose dimension will be greater in 1 will be *created*.

The code also has the option of being able to *adjust* the *coordinates* of **initial** and **final** *states* of the board so that tests can be done with *different* positions in the blocks.

7. CONCLUSIONS

Finding an **optimal solution** is the *goal* of a search method, however the amount of *computational resources* and the *time* this method entails are *important*.

Given the nature of the problem, the *Tree Search* is **not adequate** because it can be *infinitely cycled*, it allows the **Agent** to visit *previous nodes* an infinity of *times* and exhaust *system resources*.

The *Graph Search* is able to solve this problem due to its feature of *saving* and *validating* past *states* in order not to *repeat* them.

Taking into account the results obtained, we can affirm that the most adequate method for the solution of this problem is **A***, as long as it is provided with a **good estimate** in the *heuristic*.

BFS is another feasible method to use and does not need *additional* information such as **A***.

IDS is an *inefficient* method, it produces an **optimal** solution but the *time* and *resources* required for this are **very high**.

DFS is the *last resort* to use, it can only be used in *small dimensions* and the *resources* you need are *very big*.

8. LIMITATIONS AND WEAKNESSES

The use of languages such as **C** and **C++**, given their *nature* and *processing control* can produce *better performance*, however the solution will remain **equally optimal** and the *complexity* of handling and release *memory* can be very *high*.

The use of **structures** such as *HashMap* and *HashSet* in *Java* affect the *complexity* of the *space* due to the detection of *loops* and *visited nodes*.

Heuristic estimates are another factor to consider, if a **better** *estimate* is provided it will lead to *more efficient* results.

Algorithms can be improved, reducing the number of lines and *calls* to functions. In turn, a *graphical interface* can be implemented where the *user* can easily choose a *board size* and establish the *coordinates* or **initial** and **final** *states* for the blocks.

Although it is possible to change the **initial** and **final** *states* of the *blocks* and the *agent*, the code *does not allow* the *creation* of new *blocks*, *obstacles* or a second *agent*.

REFERENCES

1. R. Watson. (2010, Oct.) Search1-blind.pptx. [Online]. Available: https://secure.ecs.soton.ac.uk/notes/comp6231/lectures/07%20-%20Search1-Blind.pptx
2. SJ. (2015, May) Breadth-first search/traversal in a binary tree. [Online]. Available: hhttp://algorithms.tutorialhorizon.com/breadth-first-searchtraversal-in-a-binary-tree/
3. R. Watson. (2010, Oct.) Search2-heuristic.pptx. [Online]. Available: https://secure.ecs.soton.ac.uk/notes/comp6231/lectures/08%20-%20Search2-Heuristic.pptx
4. S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Pearson Education, 2003.