Artificial Intelligence Practical Work

Heuristic Search Methods for One Player Solitaire Games

Cohesion

Jorge Vieira (up202204385)

***Abstract*— The present study compares the performance of different search algorithms in the context of puzzle solving, more specifically on the puzzles of the solitaire game: Cohesion. First, formulating the problem as a search problem and implementing the game, only after having these steps done can we apply the algorithms (both uninformed and informed) and compare their performances based on quality of solution, number of operations performed, maximum memory used, time spent and nodes explored. With this study, it was possible to understand how these algorithms work and how well they perform on puzzle solving.**

I. INTRODUCTION

Solitaire games are characterized by usually being about solving a specific puzzle or challenge, this challenge can be from moving pieces on a board in order to achieve a certain final state, to ordering cards in a specific manner to reach a goal, there are a lot of different solitaire games with various ways of playing them.

The specific solitaire game this project focuses on, is the game of Cohesion. This game is played on a 4 by 4 board (though can be scaled to other sizes) with pieces of different colours occupying positions on the board (keep in mind the board is made out of square positions, so a piece that only occupies one position is also a square). In this game if two pieces of the same colour become adjacent, they ”glue” together forming a bigger piece, for example if a piece that occupies three positions joins another that occupies 2, they become one piece that occupies 5. The goal of the game is quite simple, only have one piece of each colour on the board.

In this report, I dive into the details of the implementation of the game in *Python*, the way this can be formulated as a Search Problem and, of course, the implementation and comparison of the search algorithms used to solve the puzzles of Cohesion.

II. PROBLEM FORMULATION

In order to formulate a given problem as a Search Problem, one must know what type of problem they are going to solve. This particular one can be seen as a Single State Problem (accessible and deterministic) which means the agent that is solving it, always has access to all the information about the environment and the next state of the environment is only determined by the previous state and the actions of the agent. The following steps are the components needed to effectively formulate this problem as a search one.

A. State Representation

The first step when formulating a problem as a search one is, how is the state or the environment going to be represented. In the case of Cohesion, a given state is composed of the board, the positions of the pieces in it, the blank positions and optionally, the size of the board. So after knowing the information needed to represent a state, I had to create a way to represent it in *Python* which was the language used to implement the game. The way I came up with to represent a state of Cohesion was a *Python* dictionary. Dictionaries store information in a very organized way, which make accessing the value of a given key take, on average, constant time (O(1)), they are also easily readable for humans which was important for debugging purposes. Here is the dictionary relative to the board of the first puzzle of the game:

```
{"blank" :
    [(1,0),(2,0),(0,2),(3,2),(1,3),(2,3)],
     "red" : {
            1 : [(0,0)],
            2 : [(3,0)],
            3 : [(1,1),(2,1)]
           },
     "blue" : {
            1 : [(0,1)],
            2 : [(3,1)],
            3 : [(1,2),(2,2)]
           },
     "yellow" : {
            1 : [(0,3)],
            2 : [(3,3)]
           },
     "size" : (4,4)
    }
```

On this board dictionary, the key ”blank” has a list of positions (x,y) that represent the squares that are blank. The colours ”red”,”blue” and ”yellow” have a dictionary of pieces where each key is the id of a piece and the value respective to that key is a list of the positions that piece occupies. As for the ”size” key it simply has a tuple (width,height) representing the size of the board (added to be able to have boards of different sizes).

This representation enables one to search directly for a given piece. For example, if someone wants to know where piece ”Red 1” is, they just have to look for the piece with ID 1 on the dictionary of the colour Red and retrieve its list of positions.

B. Initial State

The initial state of a puzzle of Cohesion is the starting point for the player and can determine the difficulty of that same puzzle. A puzzle is easier if there are less moves to make in order to solve it, so here the initial state has a lot of influence. On Cohesion, the initial state is determined by the game and there is only one per puzzle.

In the case of my project, all the initial boards are stored on a file called `boards.py` and are simply dictionaries in the form explained on the previous section.

C. Objective test

In order for an agent to know if it has solved the problem proposed to it, they have to test if the state they are on is one that corresponds to a solved state.

In a puzzle of Cohesion, as mentioned before, a puzzle is complete if there is only one piece of each colour, so for the agent to know it has reached a state like this, I created a function called `isComplete()` which determines if the board of a state has been solved or not, returning True if yes and False otherwise. With this, by testing every time it arrives at a new state, the agent can know if it has solved a puzzle or not.

D. Operators

The way an agent interacts with the environment and changes it to reach their goal, is through the use of operators, which are essentially the actions the agent can make in that environment.

In the game of Cohesion the only possible actions the player can make are moving a piece up, down, left or right. The pre-conditions to these actions are:

1) A piece cannot be moved to a position outside of the board
2) A piece can't occupy the same position as another piece. So if a piece wants to move in a given direction, it must have space for it to move (space that is not occupied by another piece).

The way this was implemented in *Python* was with the use of a function for each movement (up,down,left,right), that checks if that piece can or not move in that direction, if it can, it updates all the positions in that pieces' list of positions and also updates the board, checking if that piece became adjacent to any other piece of the same colour, making them become one whole piece if true. It is important to mention that, when a piece glues to another, the list of positions of the piece that moved is emptied and the positions of that piece go to the list of the piece it became adjacent to.

E. Solution Cost

This component of the formulation was not implemented since it wasn't necessary to get good results. But something like the amount of moves made to solve the puzzle, could be used as the cost of a given solution, since the objective is to try to solve the puzzle with the least amount of moves.

F. Puzzle Solvability

After formulating the problem and testing some algorithms, I realized they were getting stuck and not giving a solution to certain puzzles, this made me re-evaluate my approach to the problem formulation and realize this could be due to the algorithms being stuck in impossible states.

In order to solve this problem, I decided to create a function that, given a board, tells if that board is solvable or not and make the algorithms only expand nodes that have solvable boards, based on this function. This turned out to be quite a challenge since in this game it is very hard to tell the computer if a given board is impossible or not, there are many cases where the board is impossible and there is no formula to it.

Even though it was very hard to tell the computer if a board was impossible, I still managed to implement a function that got a few cases right, which are:

- If a board has pieces of different colours separated by another piece that occupies a full row or column.
- If between two pieces of the same colour there aren't as many blank spaces as the amount of positions occupied by the smallest piece of the pair.

The second case wasn't always true, but even then, this function showed to work most of the time, as it never considers a board is impossible when it isn't, at most considers the opposite of that (saying a board is possible when it isn't).

In the end, after also changing the algorithms a bit, this function didn't seem to affect the results of the algorithms too much since they would find the solution to the puzzles regardless of the use of this solvability check.

III. UNINFORMED SEARCH

Entering now the territory of the actual search algorithms, more specifically the ones that use Uninformed Search, there were 4 different algorithms of this nature that could've been implemented to solve the Cohesion puzzles:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)
- Uniform Cost Search (UCS)
- Iterative Deepening Depth-First Search (IDDFS)

Out of these 4 algorithms, the ones that were implemented were, Breadth-First Search and Iterative Deepening Depth-First Search. The reason why the other 2 were not used is because DFS got easily stuck in infinite search and was not very fit for this kind of problem and UCS was also not implemented because I was not using a solution cost, as mentioned before.

NOTE: IDDFS uses a version of DFS that only does the search until a given depth limit, so in a way DFS was implemented, just with a little twist to it.

A. Breadth-First Search

Breadth-First Search is an algorithm that is complete, which means it always finds a solution, so it was a good algorithm to implement since it always gave a result, even

though it could not be the best one and could take a lot of time, it was good to use for comparison with other algorithms.

The way I implemented this algorithm was with a queue that would store the children of the nodes of the current iteration. It starts with the initial state and then expands all nodes in that queue, stopping when it finds a solution.

B. Iterative Deepening Depth-First Search

This algorithm is fit for this type of problem because it is also complete and usually gives good solutions. It can sometimes be better and sometimes worse than BFS.

The way this algorithm works is by performing limited DFS every iteration, always increasing the maximum depth, until a certain depth limit defined by the user. I implemented this algorithm by creating a function that does limited DFS and then another that would call the limited DFS one iteratively until a certain depth limit.

IV. INFORMED SEARCH

Algorithms that use Informed Search are usually much better at finding solutions, in terms of its quality and also time used. This is because they have some information about how to find the solution, hence the name Informed Search. There are 3 different general approaches to this type of algorithms which are:

- Greedy Search
- A* Search
- Weighted A* Search

These algorithms heavily rely on something called Heuristics, which is exactly what gives the algorithms the information they need in order to find the solution. In the case of this project, all the 3 algorithms were implemented using 2 different heuristics.

A. Heuristics

As just mentioned, Heuristics are an essential part of Informed Search, without them the algorithms do not have any information about how to get to the solution, defeating the purpose of this type of search. Heuristics can give better or worse information about how to get to the solution, depending on their implementation.

In the case of this project, I implemented two admissible heuristics (which means they never overestimate the true cost) that give almost the same amount of information about how to get to the solution and both use the solution of the board in order to give information to the algorithms.

1) ***Heuristic 1****:* This heuristic determines the number of incorrect placed pieces on a board by comparing the positions of the pieces of that board with the ones from the solution board. Following the logic of this heuristic, a piece is incorrectly placed if:

- It should have an empty list of positions but doesn't
- It doesn't have the right positions in its list

The main objective of this heuristic is to give the algorithms information about the number of incorrectly placed pieces, so that they try to minimize this number and get to the solution by putting all the pieces in their correct place.

2) ***Heuristic 2****:* As for this heuristic, it also uses the solution board but, instead of determining the incorrectly placed pieces, it determines the Manhattan Distance of each piece of the current board to the final piece of the same colour in the solution board and returns the sum of all the distances.

The goal of this heuristic is to give information about how far each piece of the current board is to the final piece of the same colour in the solution board in order for the algorithms to try to minimize this distance and reach the solution.

B. Greedy Search

Greedy Search is an heuristic search algorithm that aims to find the optimal solution to a problem by making the locally optimal choice at each step. At each step, the algorithm chooses the best next step based on the heuristic function, without considering the future consequences of that choice, hence the name Greedy.

The way this algorithm was implemented in this project, was with the use of a function that, before choosing which node to expand next, determines the heuristic value of all the child nodes and then chooses the one with the lowest heuristic score. If all the child nodes have a higher score than the one the function is on, it returns that node as the solution. This implementation works for both heuristics since, in both of them, the objective is to minimize the score.

C. A* Search

A* (pronounced "A-star") is an heuristic search algorithm that uses both the cost to reach a node and an estimate of the cost to get from that node to the goal state. At each step, A* evaluates the cost of each possible next state, which is the sum of the actual cost to reach that state and the estimated cost to reach the goal from that state.

This algorithm was implemented using the greedy search function but adding the cost to get to the node, which is the length of the move history, which simply means the number of moves done to reach the node. This way, the A* function tries to get to the solution in the least amount of moves possible, being a bit more intelligent than the Greedy Search, which didn't care about this factor.

D. Weighted A* Search

Weighted A* is a modification of the A* algorithm that introduces a weight factor to control the trade-off between the actual cost to reach a node and the estimated cost to reach the goal state. It aims to find a better balance between the completeness and optimality of the A* algorithm and the speed of the greedy search algorithm.

The weight factor w is a positive real number that determines the importance of the estimated cost relative to the actual cost. Higher values of w make the algorithm focus more on the estimated cost, while lower values make it focus more on the actual cost.

On the context of my project, this weight w has a value of 3, this is because with values above that the algorithm struggled to find a solution in a reasonable amount of time. The implementation of this algorithm uses again the greedy search function, but multiplies the weight value to the number of moves needed, so the higher the weight value the more importance we are giving to the number of moves cost.

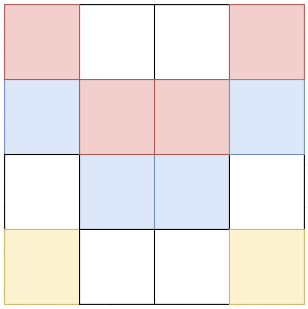
V. EVALUATION OF RESULTS

In this chapter, it is done an evaluation and discussion of results from all the algorithms implemented and also comparison taking into account the following aspects of the performance of each algorithm:

- Quality of the solution (number of moves used)
- Number of nodes explored
- Time taken to reach the solution (in seconds)
- Maximum memory used during the search (in KB)

After testing each algorithm in 3 different puzzles of the game, these were the results I got for each of the algorithms:

A. *Puzzle 1*



Algorithm	Moves	Nodes	Time	Memory
BFS	4	12311	29.52	36712
IDDFS (50)	4	4671	2.17	16772
Greedy H1	4	6	0.018	8167424
Greedy H2	4	5	0.015	8212480
A* H1	4	6	0.018	8110080
A* H2	4	5	0.014	8196096
Weighted A* H1	4	21	0.067	8343552
Weighted A* H2	4	19	0.062	8486912

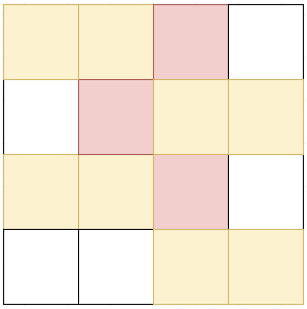
By observing the values on the table we can see that the best performing algorithm was A* with heuristic 2. The result was expected since this algorithm usually has the best performance and this heuristic was also better than the other, since it provided more information about how to get to the solution.

Something worth noting is that, the weighted A* algorithm performed worse than all the other informed search algorithms, this is because weighted A* is not very fit for this kind of problem because, by giving more weight to the number of moves, it explores nodes it really doesn't have to.

The values of the memory used for the informed search are unusually high, one reason for this might be because these algorithms have to keep the final board in memory which was not the case for the uninformed search algorithms.

Overall, informed search performed way better than the uninformed one as expected. All of the algorithms got to the solution in 4 moves and the least number of nodes explored by an algorithm to get to the solution was 5.

B. *Puzzle 2*



Algorithm	Moves	Nodes	Time	Memory
BFS	8	11620	86.86	29928
IDDFS (50)	9	98788	69.38	146036
Greedy H1	12	2191	5.28	13361152
Greedy H2	9	11	0.027	8138752
A* H1	10	126	0.28	9179136
A* H2	8	13	0.034	8126464
Weighted A* H1	8	6272	13.52	48984064
Weighted A* H2	8	1154	2.94	16347136

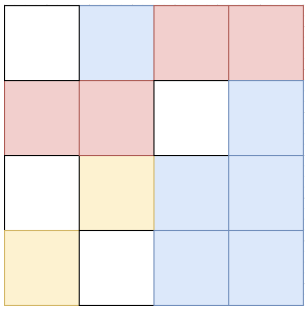
By analyzing the results above, it is clear to see that A* with heuristic 2 was the best performing algorithm in all aspects. But because this puzzle is a bit more complex and has various ways of getting to a solution we see algorithms reaching solutions with different numbers of moves, the lowest being 8.

In terms of speed, the fastest algorithm was Greedy with heuristic 2, but did not reach an optimal solution (since there are solutions with less moves), this was also the solution which explored the least amount of nodes.

Something that was a bit unexpected was the fact that A* with heuristic 1 gave a non-optimal solution, explored a lot of nodes and also took a while longer than the heuristic 2 result. With this information we start to see that heuristic 1 might not be very efficient, because even with Greedy Search it didn't perform very well.

On this puzzle, Informed Search performed better than the Uninformed one, as expected. Although the solutions from the Uninformed Search were better than some of the ones from the Informed One, due to the heuristics used. But when looking at the time taken and nodes explored, the Informed Search clearly performs better.

C. Puzzle 3



Algorithm	Moves	Nodes	Time	Memory
BFS	6	707	1.09	9412
IDDFS (50)	6	3065	1.92	12460
Greedy H1	6	9	0.016	8171520
Greedy H2	6	7	0.014	8187904
A* H1	6	27	0.057	8151040
A* H2	6	28	0.058	8220672
Weighted A* H1	6	212	0.41	9347072
Weighted A* H2	6	305	0.66	9936896

Surprisingly enough, on this puzzle the algorithm that performed the best was Greedy Search with heuristic 2 and the same algorithm with heuristic 1 being also very close. This result tells us one thing, this map favors a greedy strategy rather than a more thoughtful one. This result was interesting also to show that different algorithms work best depending on the problem they're used on, each of them serves a specific purpose and can sometimes be better than the others depending on the problem they're trying to solve.

Overall, all the algorithms got to the solution in only 6 moves which is the lowest amount of moves possible. And the Informed Search got the best results which is again expected. Proving that, in all these cases, Informed Search is a better approach than the Uninformed one in order to solve the puzzles of Cohesion.

VI. CONCLUSION

After implementing all the search algorithms , I decided to also implement a way to engage with the game, in other words, create an interface. This is a step of the this project that didn't need a chapter of its own, but deserves to be mentioned since it adds to the experience. This interface is quite simple and lets the user play the game himself or let the algorithms solve the puzzles for him, allowing the user to see the algorithm solve the puzzle, step by step. This was also implemented in *Python* using the *Pygame* library which makes it quite simple to make user interfaces and graphical content.

In summary, the results I got in the end were quite good and expected for this study. All the Informed Search algorithms proved to be more efficient than the Uninformed ones. In the context of the Informed Search algorithms, some can be better than others depending on the problem in question, but overall an algorithm that usually works well for every case is the A*.

To conclude, this project was quite challenging, from implementing the game and formulating the problem to using all the Search algorithms to solve the puzzles. In my opinion, this is what made it so interesting and enriching. After concluding this practical work, my knowledge about how AI works and how to implement intelligent agents to solve problems has definitely increased, opening new opportunities for me to apply this knowledge in the future.

REFERENCES

[1] https://chat.openai.com/chat
[2] https://www.pygame.org/docs/
[3] https://stackoverflow.com/