Solving 8-puzzle problem with different Algorithms

May 17,2022

Artificial Intelligence course

by
Maria Sameh
Ester Elzek
Eslam Yahia
steven waheed

Supervised by:
Professor.Dr.Abdel-Rahman Hedar

Teacher Assistant :
Eng.Ibrahim Saad

Department of Bio-informatics Engineering
Faculty of Computer and Information Science
Assuit University, Egypt.
2021-2022

Contents

Abstract

Intelligent tracking methods and intelligent search mechanisms can be used to tackle problems in artificial intelligence. Understandably, the performance of a search algorithm is significantly influenced by the task being solved. We assess and compare the performance of two uninformed and informed search (breadth-first search, depth first search) methods to solve the eight-puzzle game issue and one heuristic function.Informed search utilizing heuristics also gives more stable performance than uninformed search, according to graph analysis. Researchers and game developers may be concerned about these issues, thus they might consider using the heuristically augmented search method to tackle similar problems more effectively.

1 Introduction

Artificial intelligence (AI) is an area of computer science that investigates machine intelligence, learning, and adaptation. AI's major goal is to imitate and copy human intelligence before applying it to machines. Artificial intelligence research focuses on automating machine processes that require intelligent behavior. Artificial intelligence is currently revolutionizing a wide number of applications and fields through promoting emerging technology interactions.

Intelligent tracking methods and an intelligent search mechanism can be used to solve artificial intelligence problems. Uninformed search and informed search with heuristics are two types of search algorithms. Uninformed search is a search mechanism that can only distinguish between goal and non-goal states and has no idea how far the goal state is from the current state. The informed search, on the other hand, can estimate the cost of obtaining the objective from a given condition using a heuristic (a function that calculates such cost estimates) .

A lot of research has been spent on studies about uninformed and informed search in the last five years. Some research focuses on discovering and inventing new search algorithms or refining existing ones, while others are more concerned with putting search algorithms into practise in diverse industries. The performance of a search algorithm, on the other hand, is largely dependent on the problem being addressed. Several algorithms perform differently in one particular scenario. As a result, a comparison of search algorithm performance for diverse case studies is required. The 8-puzzle problem is solved using a variety of search algorithms in this study.[3]

THE PROBLEM

The N Puzzle

The sliding puzzle is a straightforward yet difficult example of artificial intelligence. It entails having a fixed size of puzzle space (typically 3x3 for an 8-puzzle) and labelling the dimensions as N columns and M rows. A random configuration of cells/blocks exists in the problem area, with one empty space allowing adjacent cells/blocks to slide into it.

Only the top, bottom, left, and right blocks nearest to the vacant area may slide into its place because the pieces are square blocks. The cells can be numbered or printed with a portion of the complete image that the rearranged puzzle should display. The following illustration shows an example of a sliding puzzle.

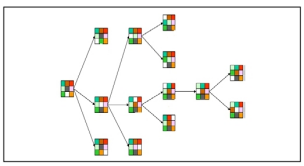


Figure. 1. Puzzle transition graph for a 3 x 3 puzzle

The Problem Formulation

As seen in Table 1, problem formulation is accomplished by analysing the puzzle's environment and extracting its features using PEAS.

Approaching the Solution

Distilling the alternative tile layouts as separate States would be most effective for solution seeking. As a result, each State depicts a conceivable tile position combination within the puzzle space. The State Space is the collection of all potential States. The size of the State Space will grow exponentially as N or M of the problem is increased.

TABLE I. ENVIRONMENT ANALYSIS

Sl No	Environment Characteristics of Puzzle	
		Description
1.	Performance	Arrangement of tiles/cells/blocks in the whole puzzle space. Main performance gauge is from the least number of moves to solve the puzzle.
2.	Environment	Puzzle space determined by N (columns) and M (rows), always with a single empty space for tiles to slide into. Numbers range from 1 to (N*M)-1. Initial state arrangements must be derived from Goal state arrangement or else there will not be possible solutions.
3.	Actuators	Tiles are moved into the empty space, either from Top, Bottom, Left or Right of the empty space.
4.	Sensors	Fully software, so the agent will have full view of the puzzle space.

The blank space location in each state defines which States can be transferred to. When the vacant area in the middle of a 3x3 puzzle, for example, tiles from the top, bottom, left, and right can move into it. Only the right or bottom tiles can slide into the vacant space if it is in the top left corner.

- **Initial Goal State:**

Any state can be initial and goal state as mentioned above.

- **Actions:**

Movement the blank square Left, Right, Up, Down

• **Transition Model:**
Given a state and action, it returns new state (if action is valid)
• **Path Cost:**
Each step is costs 1. So node depth equals to node cost for this problem.

Algorithm Review

The searches start at the root node of the search tree, which is determined by the beginning state. Three major things happen in order to access a node, among other bookkeeping details:

1)First, a node from the frontier set is removed.

2)Second, we compare the current state to the desired outcome to see if a solution has been identified.

3)Finally, if the check has a negative result, we enlarge the node. We build successor nodes nearby to the present node and add them to the frontier set to enlarge a given node.

It's worth noting that if these successor nodes are already in the border or have been visited, they shouldn't be added again.[4] The primary order of operations for search agents in this assignment is

(1) delete, (2) check, and (3) expand, which describes the life-cycle of a visit.

We will use three algorithms for solving this problem:

BFS

Breadth First Search: The Breadth First Explore (BFS) algorithm is used to traverse or search data structures such as trees and graphs. Before going on to the nodes at the next depth level, it explores all of the nodes at the current depth.

DFS

The Depth First Search (DFS) is a backtracking-based technique for traversing or exploring tree or graph data structures. It goes forward if possible or uses backtracking to explore all of the nodes.

A*

is a search algorithm that looks for the shortest path between the starting point and the destination. It's utilized in a variety of applications, including maps. The A* algorithm is used in maps to find the shortest distance between the source (starting point) and the destination (final state).

2 Methodology

8-puzzle is a basic game that consists of eight moveable tiles numbered 1 to 8 that are put on a three-tile "floor." One of the "floor" tiles is always empty, and any tiles adjacent to it (horizontally or vertically) can be moved into it. The goal of the game is to start with a given configuration and finish with the tiles arranged in numerical order.

The 8-puzzle problem is solved using a variety of search algorithms in this study. Using one heuristic function, we assess three search algorithms: breadth-first search, depth-first search, and A*.

The number of raised and investigated nodes is evaluated and analyzed for each algorithm. In terms of memory consumption and computing power requirements, the distribution and ratio of explored to raised nodes are also evaluated to determine the performance and efficiency of each algorithm.

2.1 Breadth-First Search (BFS)

Because the Breadth First Search algorithm is uninformed, it does not begin with complete knowledge of the State Space. Instead, it creates its own memory of the State Space by recalling all of the nodes it has visited. Furthermore, the BFS can only know when to halt if it finally reaches a node with the same state as the Goal Node. The BFS travels the Search Tree one level at a moment, discovering the frontier.

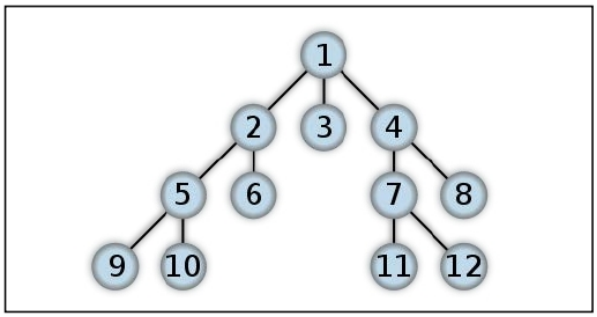


Figure.2. Breadth First Tree Traversal

The following steps will clearly describe the algorithm strategy:
Step 1: Make the Root Node the Active Node first.
Step 2: In the Solutions List, add the Active Node.
Step 3: Verify that the Goal Node is the Active Node.
Step 4: Proceed to the Final Step if it is the Goal Node.
Step 5: If this is not the case, derive all undiscovered successors and add them to the Queue.
Step 6: Proceed to the Final Step if the queue is empty.
Step 7: If not, set the next queue element as Active Node and repeat Step 2.
Final Step: Trim any Solution List nodes that aren't between the Goal Node and the Root Node in the last step.

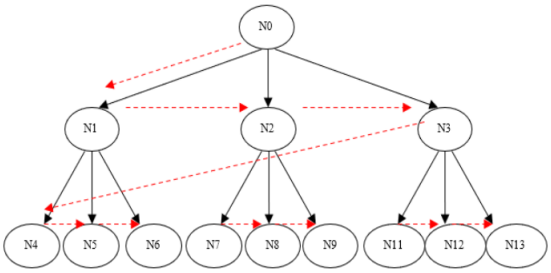


Figure.3. : Illustration of breadth-first search

To apply the Breadth-First Search method, the pseudo-code that is executed is as follows:
1. Give the starting node to the open list L
2. Loop: If the open list L is empty, then tracking is stopped
3. Put n at the beginning of the open list L
4. If n is a goal, then the tracking has been successful.
5. Remove n from the open list L.
6. Put n on the closed list C.
7. Expand n. Give the tail an open list L of all child nodes that have not appeared in open list L or closed list C and assign a pointer to n 8. Back to Loop

2.2 Depth First Search (DFS)

The Goal identification and Search Tree traversal modes are equivalent in the Depth First Search method, but not in the expansion motion. Rather than going horizontal first, the DFS approach selects a branch and continues to extend it until it encounters a dead end. If a goal cannot be found, the process will be repeated by returning to the next accessible alternate branch. Figure 4 illustrates this motion.[1]

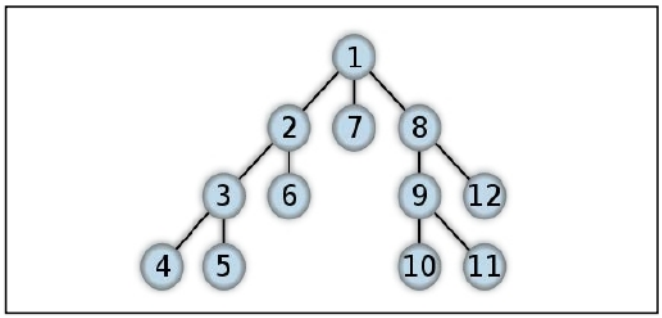


Figure. 4. Depth First Search Tree Travesal

As we know in the BFS strategy, the expansion flow is affected by the type of data structure used. Instead of using queue (First-In-First-out), DFS will use stack data structure (First-In, Last Out).The rest of the algorithm is identical to BFS. However, because pure DFS has a high probability of infinite depth, a depth restriction will be required to force the agent to backtrack. The DFS will now be called Depth-Limited Search.

Because the main distinction between DFS and BFS is the frontier data structure, DFS' programming flow chart is identical to BFS', with the exception that the Frontier is now a Stack instead of a queue. While the Depth Limited Search uses less memory than BFS when expanding 4743 but dropping 4832 nodes, it takes a different amount of time to search depending on whether it finds a branch that leads to the objective. Also, because it might get lost expanding a branch without reaching an end or objective, the DFS is not a comprehensive approach. With 4743 nodes expanded, it took substantially longer than BFS, and the solution is inefficient with 4730 movements.

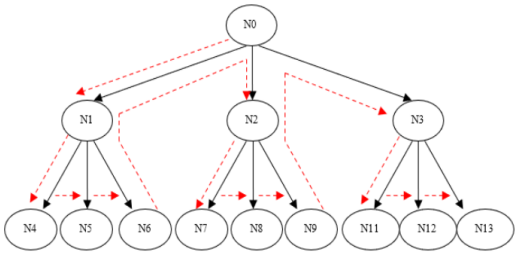


Figure. 5. Illustration of depth first search

The pseudo-code that is performed to apply the depth-first search strategy is as follows:

1. Give the open list L the starting node.
2. If the open list L is empty, the loop has failed.
3. Place n at the start of the open list L.
4. The tracking is successful if n is a target.
5. Take n out of the open list L.
6. Expand n, then add a pointer from the nth-child node to the open head.
7. Return to the Loop.

2.3 A*

The A* Search is an informed search algorithm based on heuristics. The employment of State Space awareness in the form of the Heuristics Function is referred to as "Informed." All informed search strategies rely on knowing how closely a given node resembles the Goal Node. Simply put, H(n) =Difference Heuristics (Goal State,Current State). The less ideal the option, the higher H(n).

A heuristics search method will essentially use the horizontal expansion motion of Breadth First Search, but will only expand leaves that are most 'ideal,' i.e. have the smallest H. (n). The Path Cost is also incorporated into the Heuristics function for each node in the A* algorithm. Nodes are now picked not only for their similarity to the Goal State, but also for their distance from the Root Node. Figure 6 is an example of the A* traversal.

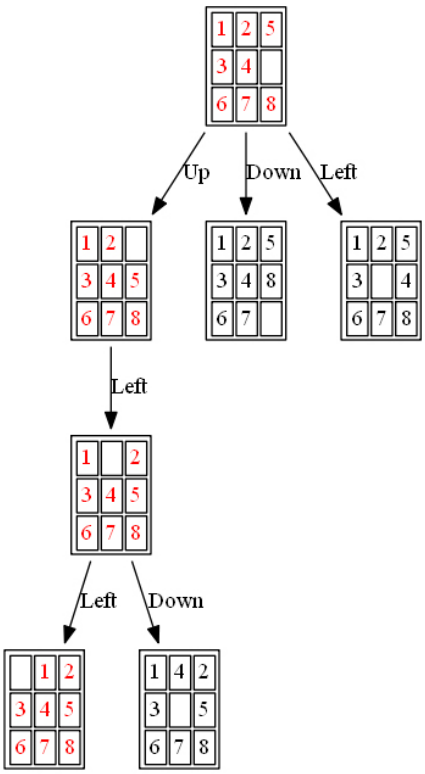


Figure. 6. A* traversal of the 8-Puzzle search tree

The BFS is the essential foundation of the A* search algorithm, although major changes have been made to successor generation and queue management, which involves heuristics and path cost calculations. Figure 7 shows a flowchart that reflects these details.

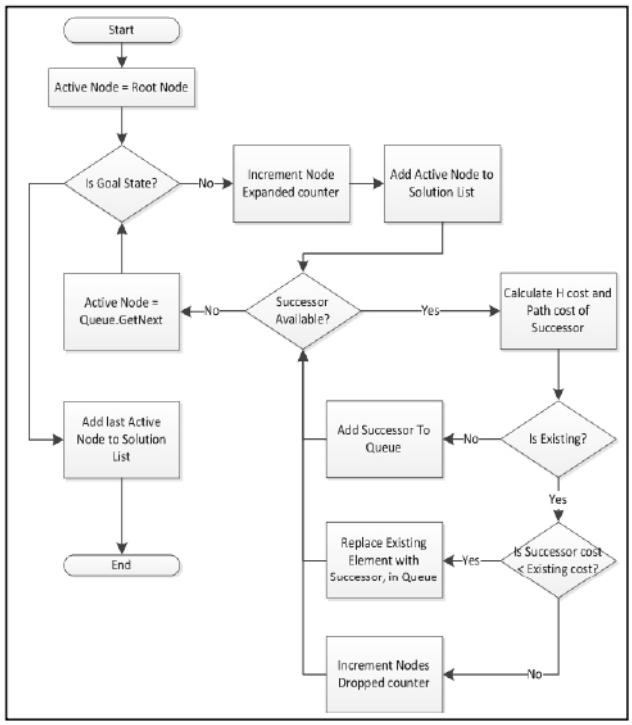


Figure. 7. A* Search Algorithm Implementation

It was observed that the A* algorithm, like BFS, was able to find the optimum answer in four stages, but with just four nodes enlarged. It also did so with just 9 nodes generated and 3 dropped, probably because they were too dissimilar from the Goal State. It uses less memory than DFS and is faster than BFS, but it takes more computing power because each node is evaluated not only for its ability to match the **Goal State**, but also for its ability to match the current state.how much difference between them and how far down the tree it has gone.[2]

Analyzing the Algorithms

BFS:

Completeness?
Yes, if solution exists, there is a guarantee to find it
Time complexity?
O(bd)
Space complexity?
O(bd)
Optimality?
yes

DFS:

Completeness?
Yes, assuming state space finite
Time complexity?
O(n), can do well if lots of goals
Space complexity?
O(n), n deepest point of search
Optimality?
No may find a solution with long path

A*:

Complete?
Yes (unless there are infinitely many)
Time/Space?
Exponential mostly b^d
Optimal? Yes
Optimally Efficient: Yes

3 Experimental Simulation

3.1 The programming languages and environments

we have used JavaScript; one of the most efficient programming language to implement our solution with multiple algorithms . It supports executing the search step by step, also visualizes the search tree.

It has a great run time and code custom code utilization we will discuss some major points of our solutions with portions of code snippets

Overview:

Programming Language: JavaScript and npm module
Environment(IDE) : Visual Studio Code v 1.67.2
Plugins and attachment: IntelliSense auto-completion, code glance and Copilot AI URL: `https://copilot.github.com/`

Code Style:

1- Focus on code readability
2- Standardize headers for different modules
3- Leave comments and prioritize documentation.
4- Try to formalize Exception Handling

3.2 The primary functions

```
// main search function :
function search(opt_options) {
    var options = _.assign({
        node: null,
        frontierList: [],
        expandedNodes: {},
        iteration: 0,
        iterationLimit: 1000,
        depthLimit: 0,
        expandCheckOptimization: false,
        callback: function() {},
        stepCallback: null,
        type: SearchType.BREADTH_FIRST,
        maxFrontierListLength: 0,
        maxExpandedNodesLength: 0,
        iterativeDeepeningIndex: 0
    }, opt_options || {});

    Board.draw(options.node.state);

    if (options.node.game.isFinished()) {
        return options.callback(null, options);
    }

    // Expand current node
    var expandedList = options.node.expand();
    options.expandedNodes[options.node.state] = options.node;
    options.maxExpandedNodesLength = Math.max(options.maxExpandedNodesLength, _.size(options
        .expandedNodes));

    // Filter just-expanded nodes
    var expandedUnexploredList = expandedList.filter(function(node) {
        // Check iterative deeping index
        if (options.type == SearchType.ITERATIVE_DEEPENING && node.depth > options.
            iterativeDeepeningIndex)
            return false;

        // Check depth
        if (options.depthLimit && node.depth > options.depthLimit)
            return false;

        // Check whether node is already expanded (with lower cost)
        var alreadyExpandedNode = options.expandedNodes[node.state];
        if (alreadyExpandedNode && alreadyExpandedNode.cost <= node.cost) return false;

        // Check whether there is a better alternative (lower-cost) in frontier list
        var alternativeNode = _.find(options.frontierList, {state: node.state});
        if (alternativeNode && alternativeNode.cost <= node.cost)
            return false;
        else if (alternativeNode && alternativeNode.cost > node.cost) {
            _.remove(options.frontierList, alternativeNode);
        }

        return true;
    });

    // Add filtered just-expanded nodes into frontier list
    options.frontierList = options.frontierList.concat(expandedUnexploredList);
    options.maxFrontierListLength = Math.max(options.maxFrontierListLength, options.
        frontierList.length);

    // Check whether desired state is in just-expanded list
    if (options.expandCheckOptimization) {
        var desiredNode = _.find(expandedUnexploredList, function(unexploredNode) {
            return unexploredNode.game.isFinished();
        });
```

```
        if (desiredNode) {
            return options.callback(null, _.assign({}, options, {node: desiredNode}));
        }
    }

    // Next call
    var nextNode = getNextNode(options);
    if (!nextNode) {
        return options.callback(new Error('Frontier list is empty'), options);
    }

    // Iteration check
    options.iteration++;
    if (options.iterationLimit && options.iteration > options.iterationLimit) {
        return options.callback(new Error('Iteration limit reached'), options);
    }

    if (window.searchStopped) {
        window.searchStopped = false;
        return options.callback(new Error('Search stopped'), options);
    }

    if (options.stepCallback) {
        options.stepCallback(_.assign(options, {node: nextNode}));
    } else {
        setTimeout(function() {
            search(_.assign(options, {node: nextNode}));
        }, 0);
    }
}
```

Listing 1: The main Search function

```
// function to go to the next node in track:
function getNextNode(options) {
    switch (options.type) {
        case SearchType.BREADTH_FIRST:
            return options.frontierList.shift();
        case SearchType.DEPTH_FIRST:
            return options.frontierList.pop();
        case SearchType.UNIFORM_COST:
            var bestNode = _.minBy(options.frontierList, function(node) {
                return node.cost;
            });

            _.remove(options.frontierList, bestNode);

            return bestNode;
        case SearchType.ITERATIVE_DEEPENING:
            var nextNode = options.frontierList.pop();

            // Start from top
            if (!nextNode) {
                options.iterativeDeepeningIndex++;

                if (options.depthLimit && options.iterativeDeepeningIndex > options.
                    depthLimit)
                    return;

                options.frontierList = [];
                options.expandedNodes = {};

                return new Node({state: game.state});
            }

            return nextNode;
        case SearchType.GREEDY_BEST:
            var bestNode = _.minBy(options.frontierList, function(node) {
                return node.game.getManhattanDistance();
            });

            _.remove(options.frontierList, bestNode);

            return bestNode;
        case SearchType.A_STAR:
            var bestNode = _.minBy(options.frontierList, function(node) {
                return node.game.getManhattanDistance() + node.cost;
            });

            _.remove(options.frontierList, bestNode);

            return bestNode;
        default:
            throw new Error('Unsupported search type');
    }
}
```

Listing 2: Javascript Example

3.3 Test cases

Test Cases:

Suppose that we have an Eight -puzzle board and we want to reach to the goal state like this:

1 2 3
8 0 4
7 6 5

Now we should enter our initial state for every test case and our aim is to solve the puzzle with all of the discussed algorithms

Note: 0 number represents the blank tile

Test case1: Initial state

1 3 4
8 0 5
7 2 6

This puzzle can be solved in six moves. First, the two is moved up, the six is shifted to the left, the five is swapped downward, the four is swapped downward, the three is shifted to the right, and the two is moved up. If you consider the movement of space, it moves down, right, up, up, up, left, down.

Movement flow:

1 3 4
8 0 5
7 2 6
"FIRST, from here, move DOWN."

→

1 3 4
8 0 5
7 0 6
"SECOND, from here, move RIGHT."

↓

1 3 4
8 2 0
7 6 5
"NEXT, from here, move UP

←

1 3 4
8 2 5
7 6 0
"THIRD, from here, move UP.".

↓

1 3 0
8 2 4
7 6 5
"THEN, from here, move LEFT.

→

1 0 3
8 2 4
7 6 5
"THEN, from here, move DOWN."

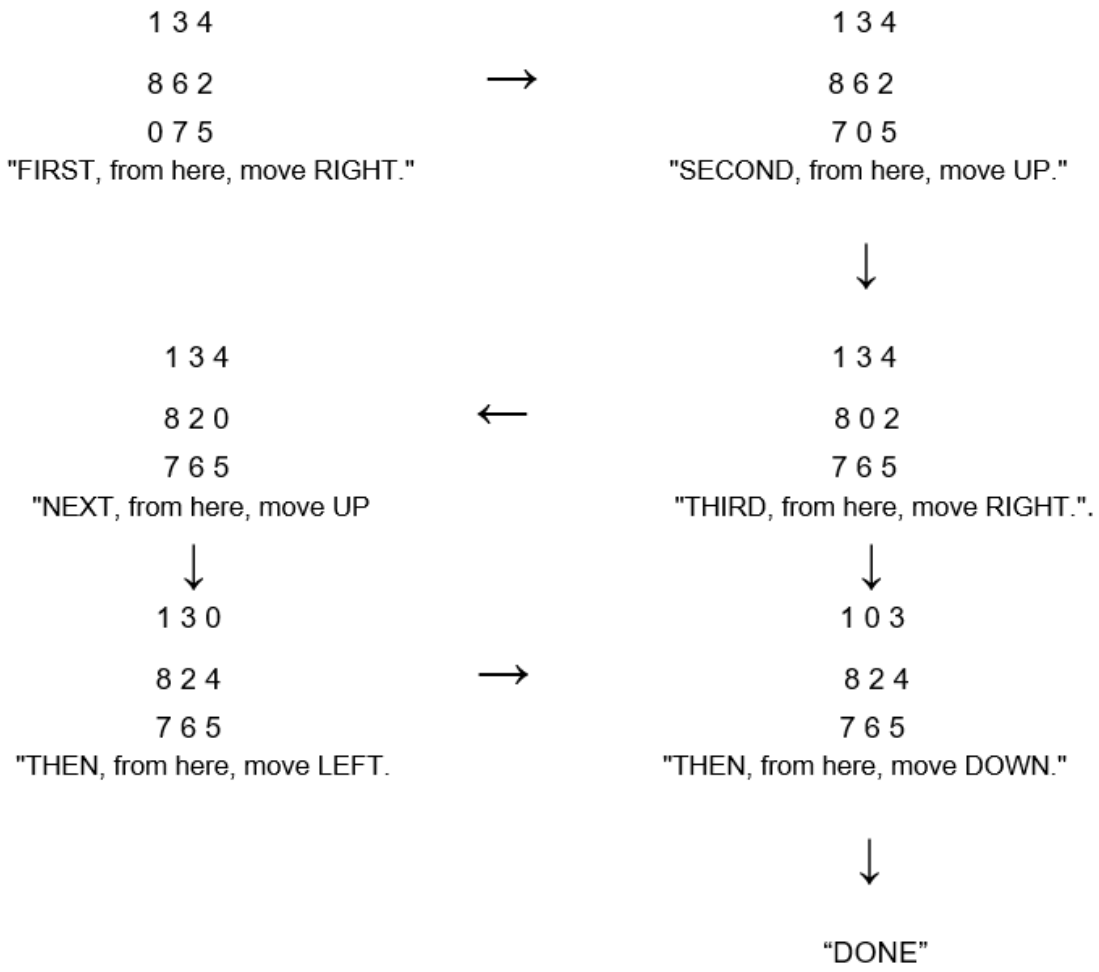
↓

"DONE"

Test case2: Initial state

1 3 4
8 6 2
0 7 5

This problem also takes six steps to complete. The seven is moved to the left, the six is moved down, the two is moved to the left, the four is moved down, the three is moved left, and the two is moved up. In terms of spatial movement, the solution is thus right, up, right, up, left, down. Movement flow:



4 Results and Technical Discussion

Experimental procedure

In this article and also in the program, all the states are represented by 9-length string. The blank square is 0. For example, the goal state is represented by 123456780.

To examine the algorithms in same manner, following states are chosen as input values:
- 5 states with optimal solution by depth 5
- 5 states with optimal solution by depth 10
- 5 states with optimal solution by depth 15
- 5 states with optimal solution by depth 20
- 5 states with optimal solution by depth 24
- 2 states with optimal solution by depth 26
- 2 states with optimal solution by depth 31

All the states are generated by program except with depth level 31. At first I tried to generate states in totally random order, but not all the states are valid and solvable. To overcome this issue, program starts the game from goal state and moves away randomly for provided steps without re-visiting any nodes, at some point A* algorithm runs and checks optimal solution depth. While researching, we encountered that the hardest 8-puzzle game can be solved in 31 moves, then added to data set for benchmark purposes. Uninformed search algorithms are not run for depth 26 and bigger, because they require too much time to find a solution.

For informed searches, Manhattan distance is used as heuristic function. All the algorithms will not add already expanded nodes to frontier list unless its cost is lower than already-expanded node. Also while adding frontier list, it will check current frontier list for same state and compare their cost, resulting in add to frontier list, ignore, replace in frontier list.

As output, following properties are examined:
- Maximum number of expanded nodes in memory
- Maximum number of frontier list in memory
- Total iteration count

Total iteration count is important. Because program can revisit nodes with lower cost, so that previous iteration becomes meaningless.

DFS algorithms are getting deeper and deeper blindly, for big trees like 8-puzzle problem, it never finds a solution. So itis better to set a maximum depth limit. In this experiment, 3 depth limit examined: m, m+1 and m+2 where m is optimal solution depth.

Analyzing the results

4.1 Breadth-First Search (BFS)

For all the BFS and UCS experiments, program finds the optimal solution. As seen in Table 1, program iterates 170k in average to solve depth 26.

BFS & UCS Average Results			
Optimal Solution Depth	***Iteration***	***Expanded Nodes***	***Frontier***
5	44	44	34.2
10	640	640	382.4
15	6467	6467	3667
20	49418	49418	17863.4
24	127959	127959	24238.6
26	170858	170858	24518.5

Fig. 8. BFS and UCS average results for 5 sample for each optimal solution depth level

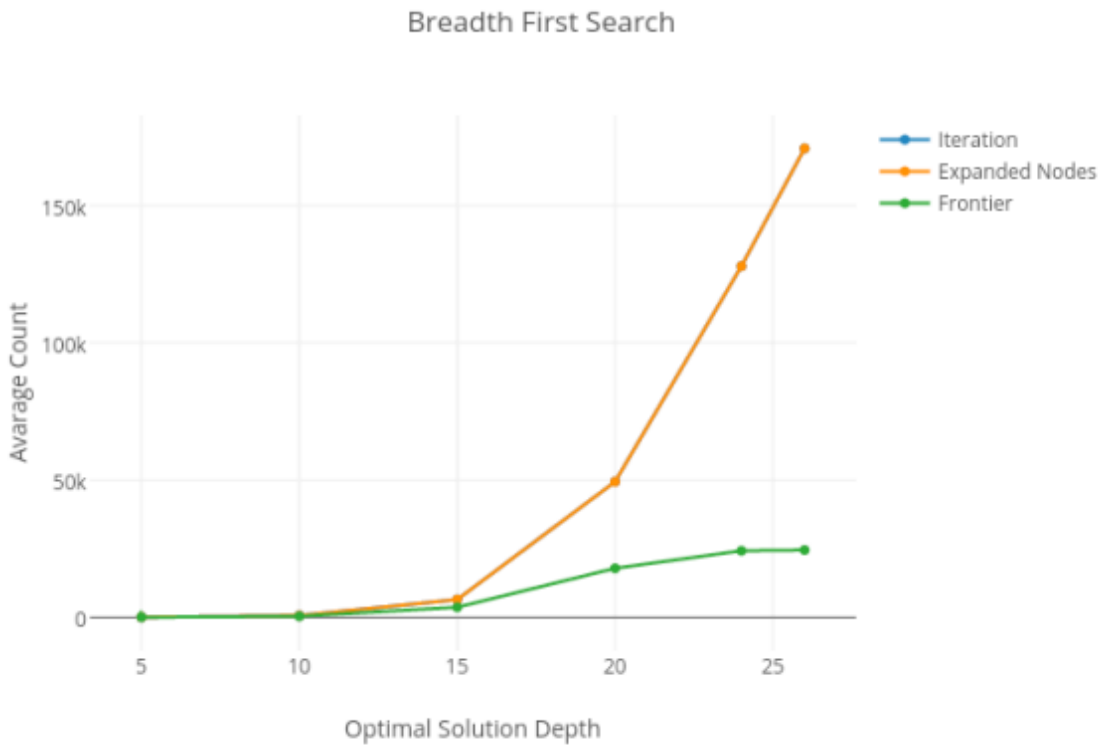


Fig.8.BFS results by optimal solution depths.Blue iteration line is under the yellow expanded nodes line.

4.2 Depth First Search (DFS)

For all the **DFS (with depth limit m)** where m is the optimal solution depth, program finds the optimal (and only) solution because it cannot go deeper from m. If compared with BFS,DFS (m) performs better at iteration count and expanded node count, but the frontier list memory usage drops huge. This is the best feature of IDS-variant algorithms. But in real world,the optimal solution of the state is generally unknown. It can be better alternative to BFS if solution depth is known or predictable.

DFS (with depth limit *m*) Average Results			
Optimal Solution Depth (m)	***Iteration***	***Expanded Nodes***	***Frontier***
5	36	36	7
10	300	289	10.2
15	4879	4255	15.4
20	37460	27060	18.8
24	122613	71872	22.2
26	202514	116634	24

Fig.9.DFS (with depth limit m) average results for 5 sample for each optimal solution depth level

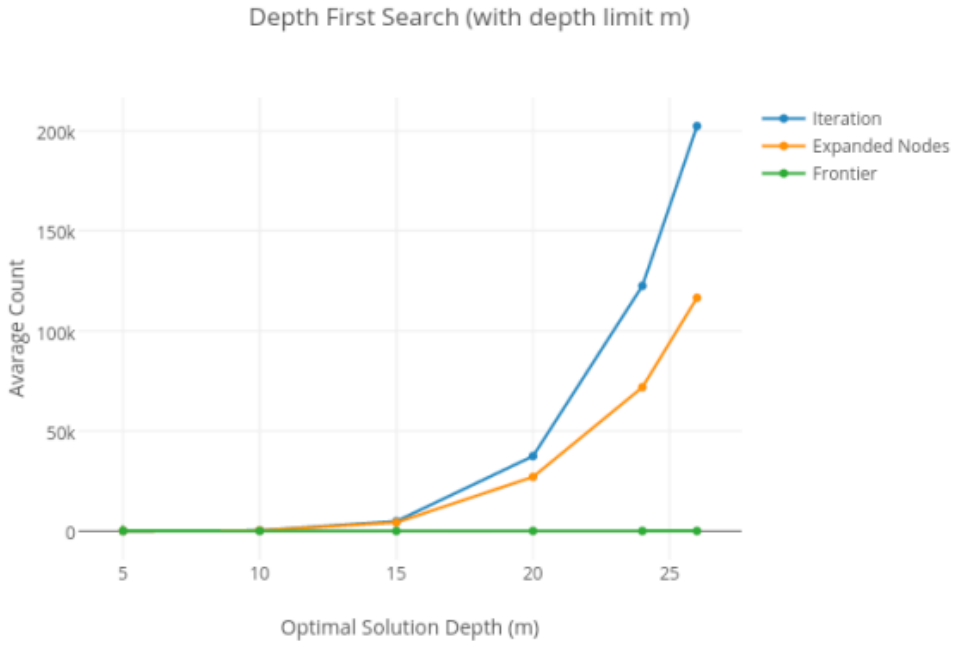


Fig. 10. DFS (with depth limit m) results by optimal solution depths.

Interestingly, for all **DFS (m+1)** experiments, program finds the optimal solution again where depth is m. We could expect to find a solution with m+1 depth, because depth m+1 is expanded before depth m. Also all the counts are increased if compared with DFS (m), because program firstly searches depth m+1

DFS (with depth limit *m+1*) Average Results			
Optimal Solution Depth (m)	***Iteration***	***Expanded Nodes***	***Frontier***
5	62	62	7.6
10	476	451	11.2
15	7630	6493	16
20	52919	36502	94.4
24	157150	86563	22.4
26	257176	135979	24.5

Fig.11.DFS (with depth limit m+1) average results for 5 sample for each optimal solution depth level

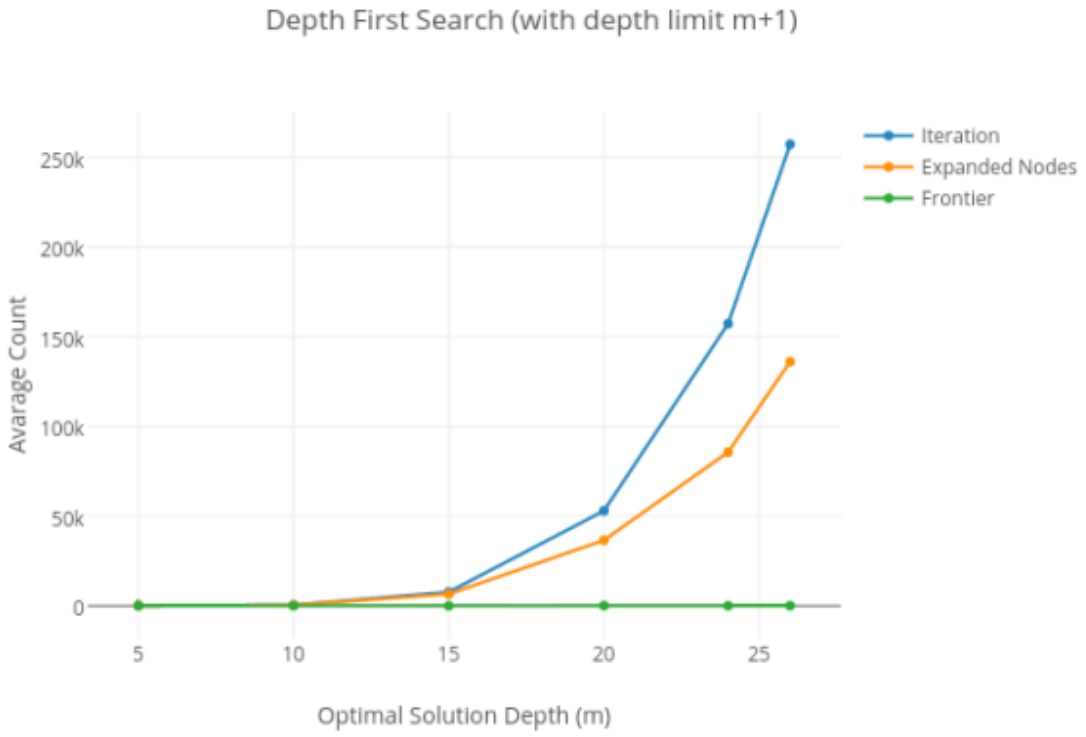


Fig.12. DFS (with depth limit m+1) results by optimal solution depths.

For all **DFS (m+2)** experiments, program generally finds the solution depth is m+2 as expected. But sometimes it can find the optimal solution with depth m

DFS (with depth limit *m+2*) Average Results			
Optimal Solution Depth (m+2)	***Iteration***	***Expanded Nodes***	***Frontier***
5	110	109	8.6
10	776	717	12.2
15	9853	8833	17
20	68389	45869	20.4
24	194845	97053	23.4
26	332467	156760	25.5

Fig.13.DFS (with depth limit m+2) average results for 5 sample for each optimal solution depth level

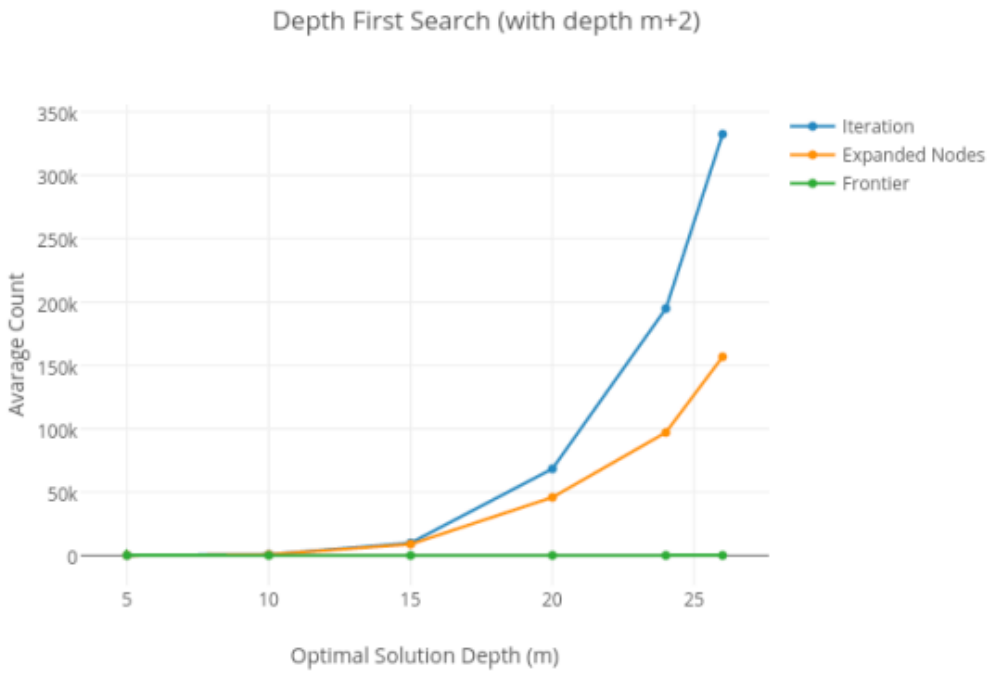


Fig.14. DFS (with depth limit m+2) results by optimal solution depths.

4.3 A*

In all of A* searches, program finds the optimal solution.Iteration counts and nodes in memory is increased if compared with GBS. It is impressing that A* algorithms finds the optimal solution for the hardest state with around 20k iterations.

A* Search Average Results			
Optimal Solution Depth	***Iteration***	***Expanded Nodes***	***Frontier***
5	5.2	5.2	6.6
10	22.4	22.4	19.2
15	83.4	83.4	56.8
20	395.6	395.6	238
24	2368	2368	1283.6
26	3904	3904	2018.8
31	20290	20290	8665

Fig.15. A* search average results for 5 sample for each optimal solution depth level, for depth 26 and 31 there are 2 samples.

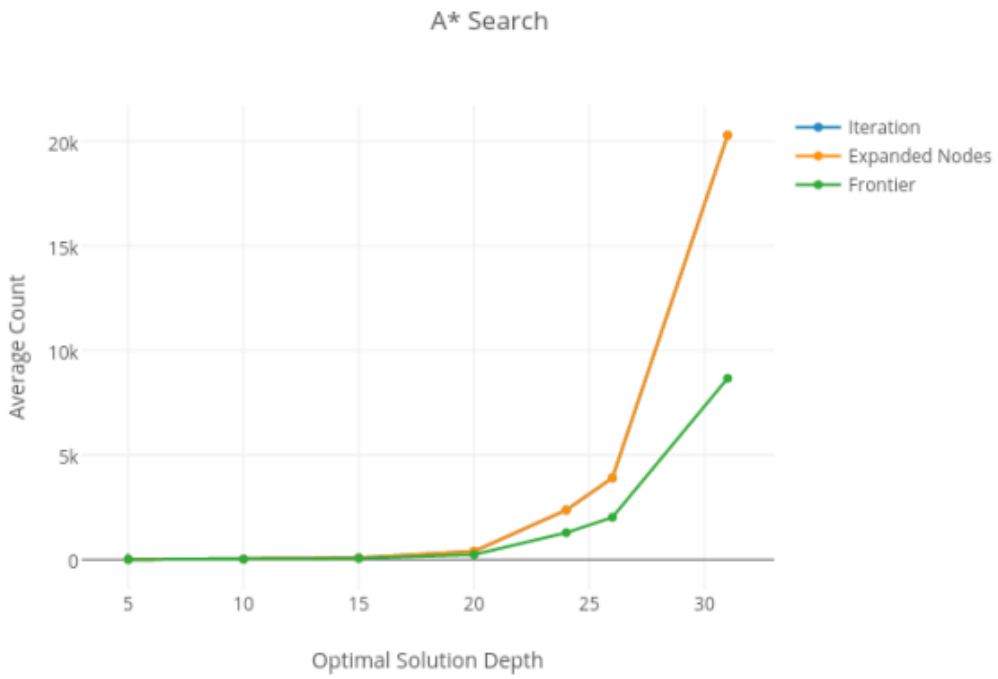


Fig.16. A* search results by optimal solution depths. Blue iteration line is under yellow expanded nodes line.

5 Conclusions

Breadth-First Search (BFS)

More comprehensive and compact ignorant method that finds the optimal solution with minimal compute but uses a lot of memory. BFS is best for small dimension sliding puzzles; however, for the expandable N*M type with exponentially increasing complexity, BFS is less efficient and can run out of memory before completing extended solutions.

Depth First Search (DFS)

Before going retracing, explore each branch separately. It's actually useful for long solutions, but only if it starts on a branch with a potential Goal State. Because the sliding puzzle has only one certain solution, DFS will be ineffective. A comprehensive search method is preferred.

A*

The A* search finds the shortest solutions with reasonable memory and time performance by combining heuristic values with path costs. If the puzzle is little, a lighter method will be preferable. This dynamically up-scaling challenge, on the other hand, will benefit more from the complete A*'s and heuristic approach.

After depth 20,(according to Fig.15) iterations and node counts are increasing exponentially. So an informed search algorithm should be chosen over uninformed ones. If solution cost is not important. A* algorithm will expand more nodes in breadth level to find better paths.

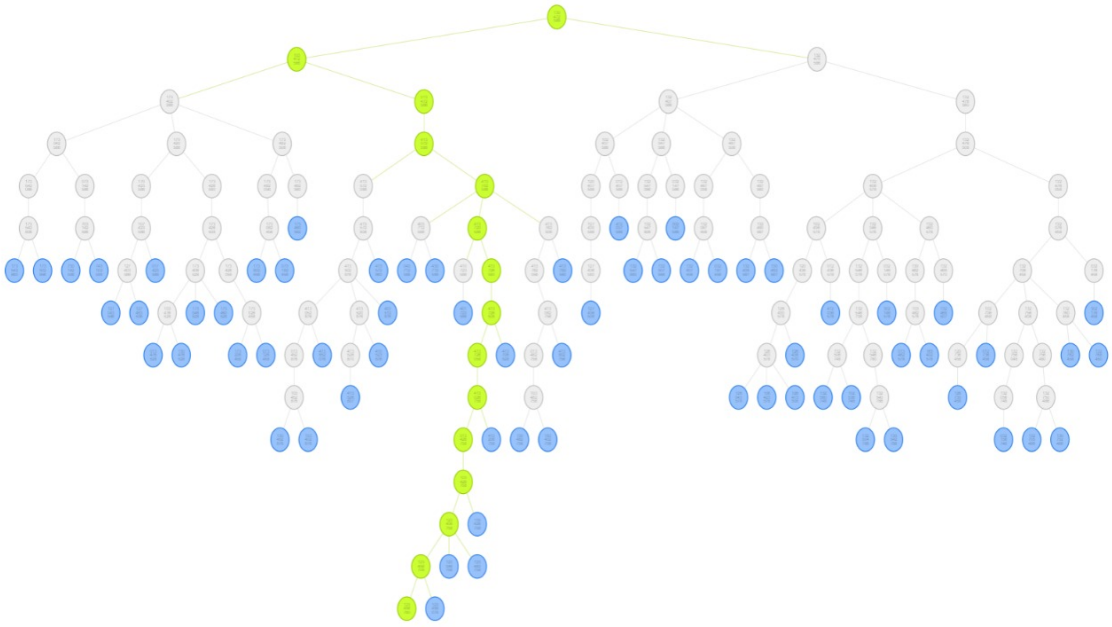


Fig. 17. Search tree visualization of A* search algorithm solves 012345678 state. (Solution depth: 14, expanded nodes: 96/96, frontier nodes: 66/67)

6 Appendix

Our source code now on Github

GitHub Repo:`https://github.com/eslamyounis1/8-Puzzle-FCI-AUN`

Website Host:`https://eslamyounis1.github.io/8-Puzzle-FCI-AUN/`

References

[1] Blai Bonet and Hector Geffner. "Learning depth-first search: A unified approach to heuristic search in deterministic and non-deterministic settings, and its application to MDPs." In: *ICAPS*. Vol. 6. 2006, pp. 142–151.

[2] Richard E Korf and Peter Schultze. "Large-scale parallel breadth-first search". In: *AAAI*. Vol. 5. 2005, pp. 1380–1385.

[3] Debasish Nayak. "Analysis and Implementation of Admissible Heuristics in 8 Puzzle Problem". PhD thesis. 2014.

[4] Judea Pearl and Richard E Korf. "Search techniques". In: *Annual Review of Computer Science* 2.1 (1987), pp. 451–467.