Unblock Me - Group 19

Guilherme Silva (201603647)
IART
FEUP
Porto, Portugal
up201603647@fe.up.pt

Miguel Duarte (201606298)
IART
FEUP
Porto, Portugal
up201606298@fe.up.pt

Rui Alves (201606746)
IART
FEUP
Porto, Portugal
up201606746@fe.up.pt

***Abstract*—The aim of this paper is to implement and compare Artificial Intelligence algorithms for solving the game *Unblock Me*, in which rectangular pieces block the path of a red special piece. The objective of the game is for the special piece to reach the exit of the level, which is achieved by moving it and the other pieces. The problem will be described and then formulated as a *Search Problem*, together with some discussion on the reasoning behind some of the decisions taken during this process. In the pursuit of the optimal *Search Algorithm*, several algorithms were implemented and tested, which resulted in a study of their performance, being that different metrics were taken into account. This study was applied to puzzles of varying complexity and difficulty.**

***Index Terms*—Artificial Intelligence, Search Problems, Path-finding algorithms, Decision-making Heuristics, Graph Algorithms, Depth-First Search, Breadth-First Search, Iterative Deepening, Greedy Search, A*, PyPy, Python**

I. Introduction

The game that is the subject of study in this paper (*Unblock Me*) will be approached with several different *Search Algorithms*, implemented in *Python*, in order to conclude what is the best approach for solving it using *Artificial Intelligence Algorithms*.

Firstly, the game will be described in further detail, ensuring that all non-formal aspects of it are well documented.

Secondly, it will be formulated as a *Search Problem*, with the approach taken in doing so being illustrated in increased depth, providing reasoning for the decisions taken.

Thirdly, the game implementation will be touched upon, being some of the main aspects of it described, namely the representation of the initial game state in the text files that can be loaded by the program.

Moreover, the *Search Algorithms* that were used to approach the problem being studied will be succintly described, also focusing on the description of the implemented *Heuristics*.

Additionally, the performed experiments and their results will be presented and some conclusions extrapolated from them.

Finally, a section will cover the conclusions and future work, by expanding on the already developed tasks and elaborating on what can be done to further develop work related to this topic.

II. Problem Description

Unblock Me is a puzzle game that was released in 17-06-2009 by *Kiragames*. It is an adapted version of *Rush Hour*, a puzzle game invented by Nobuyuki Yoshigahara in the 1970s. Each level consists of a 6x6 board, surrounded with walls (except for the puzzle's exit) [1].

Fig. 1. Example of an *Unblock Me* level

The game's objective is to move a special piece to the level's exit, by moving that and the other pieces with the least number of movements possible. Pieces are rectangles with a given orientation (vertical or horizontal) and constant length. Pieces can only move in the direction of their orientation into empty cells (they may not overlap).

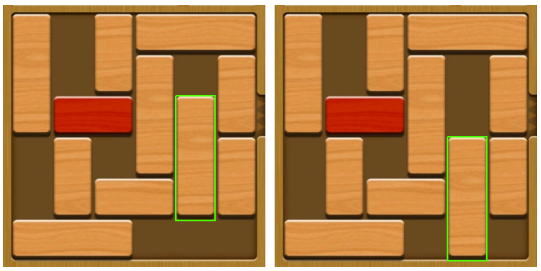

Fig. 2. Piece moving down one cell

Levels are fully surrounded by walls, except for the level's exit door that is alligned with the special piece and by which only the special piece can go through. The level is completed once the piece goes through the exit door.

Fig. 3. Beating a level by going through the exit

Some levels may even contain fixed blocks that can not be moved, representing obstacles.

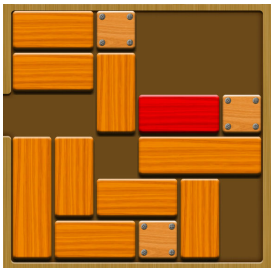

Fig. 4. Example of an *Unblock Me* level containg fixed blocks

III. Problem Formulation

The game's solving process can be formulated as a search problem, in which the goal is to find the sequence of moves (state transitions) that take the special piece to the level's exit door - the problem's goal state.

A. Game State Representation

- List of pieces, where each piece contains the following information:
  - Origin Square (top left piece corner, e.g. (0,0))
  - Length (e.g. 4)
  - Direction (H or V)
- Reference to the special piece
- Matrix of booleans, where True means the cell is empty

This representation of the game state makes the generation of valid movements easier, due to the fact that to determine possible movements of a single piece, the other pieces do not need to be taken into account (assuming that all empty cells are known). Thus, to generate all possible branching options, the list of pieces needs to be iterated over, while checking if there are empty cells adjacent to the piece's extremities (these change with the piece's direction). This representation facilitates the execution of these tasks, making it less costly (important fact due to high number of operations of this kind):

- Using a list of pieces makes it trivial to determine which of them can be moved in a given state;
- Using a matrix (of booleans, that state if a cell is occupied or not) makes it trivial to check if adjacent cells to the pieces extremities are empty;
- Finnaly, using a reference to the special piece allows accessing it in constant time, which will be useful to heuristic related operations and determine if the goal state (subsection III-C) is achieved.

B. Initial State

The initial state depends of the level in question, being represented by a game state as described in subsection III-A.

C. Goal State

The goal state consists of a piece configuration where the special piece reaches the level exit (Figure 3).

D. Operators

- Move piece to the left:
  - Pre-conditions:
    - Piece with horizontal orientation
    - At least the cell that is adjacent to the piece's left extremity must be empty
  - Results: The piece is moved a number of cells to the left (at most the number of empty cells, at least one cell)
  - Cost: 1 movement
- Move piece to the right:
  - Pre-conditions:
    - Piece with horizontal orientation
    - At least the cell that is adjacent to the piece's right extremity must be empty
  - Results: The piece is moved a number of cells to the right (at most the number of empty cells, at least one cell)
  - Cost: 1 movement
- Move piece up:
  - Pre-conditions:
    - Piece with vertical orientation
    - At least the cell that is adjacent to the piece's top extremity must be empty
  - Results: The piece is moved a number of cells upwards (at most the number of empty cells, at least one cell)
  - Cost: 1 movement
- Move piece down:
  - Pre-conditions:
    - Piece with vertical orientation
    - At least the cell that is adjacent to the piece's bottom extremity must be empty
  - Results: The piece is moved a number of cells downwards (at most the number of empty cells, at least one cell)
  - Cost: 1 movement

E. Path cost

The solution's path cost that is to be minimized is equal to the number of movements made (number of state transitions).

IV. Related Work

Unblock Me is also widely known as *Rush Hour*. In June 2018, Michael Fogleman (a Software Engineer at Formlabs) developed an artificial intelligence that solves the puzzle and created a database with over two million and a half different puzzles [2]. His solution is based on the usage of bitboards [3] (data structure that is often used by board game computer systems due to its reduced space usage and the efficiency of operating upon it) for representing the game states.

This game is also studied in the Computer Science MSc from the University of Princeton, namely in the course "Computer Science 402 - Artificial Intelligence". In this course, students implement multiple heuristics to solve the problem using the A* algorithm [4]. Rainhard Findling, a researcher in Aalto University in Finland, proposes a solution to this problem statement, by using a heuristic based on a vote system, in which each piece that is blocking the way of the special piece to the level's exit votes on how they want to free the way (how many moves they have to perform) for the special piece [5].

Even though that the problem is not complex, increasing the board size quickly leads to a combinatorial explosion, being that a 6x6 board has over 27 billion possible states. Thus, it is not efficient to find a solution by exploring all the possible states, being a good heuristic for state exploration mandatory for complex game levels.

V. Game Implementation

In the implementation of the game there are two classes: **GameState** and **Piece**. An instance of **Piece** contains the position of the top left corner, the length and the direction of the piece. Each of these is part of an instance of GameState, which besides a list of the pieces also contains a reference to the special piece and a boolean matrix, where each element of it represents the state of a given cell (empty or occupied). Using this implementation, the objective test is made by checking if the special piece intersects with the exit.

To determine all adjacent states, the list of pieces is iterated and for each one the following steps are taken:

1) Determine the position of the piece's extremities
2) For each extremity determine the direction of the directly adjacent cell (i.e. for the left extremity of an horizontal piece the adjacent cell it will be to the left)
3) Then create N new states with an updated matrix and an updated piece, where N is number of empty cells adjacent to the piece in direction determined in the previous step

Each of the states created using this method has the cost of the original state plus one.

A. File representation

Each puzzle can represented by a single string, where 'o' represents an empty space, 'x' represents an immovable space and each piece is identified by an upper case letter, 'A' being reserved for the representation of the special piece.

VI. Search Algorithms

As determined beforehand, the following algorithms were implemented:

- BFS (Breadth-First Search)
- DFS (Depth-First Search)
- Iterative Deepening Depth-First Search
- Greedy Search
- A* Search

The **Uniform Cost Search** algorithm was not implemented, as all of the **Operators** (described in subsection III-D) have the same cost - this algorithm would simply be a **Breadth-First Search (BFS)**.

Furthermore, it was decided that, unlike initially planned, **Bi-Directional Search Algorithms** would not be implemented. This is due to the fact that despite the **Goal State** of *Unblock Me* always featuring the special piece over the level exit, all of the other pieces might be in very different configurations. Thus it is not possible to start searching from the final state of the level, making these algorithms not applicable to the problem under consideration.

A. Breadth-First Search

Breadth-First Search (BFS) analyzes all of the states with a depth lower than N prior to analyzing states with a depth of N. To accomplish this, all of the nodes that need to be analyzed are kept in a *Queue*. The next state to be analyzed is at the front of the *Queue*. When analyzing a state, the next possible states are generated and appended to the end of this structure.

The time and space complexity of this approach is exponential, making the algorithm consume copius amounts of memory. This makes it hard to run for puzzles that have a solution size bigger than 10. However, by using a *Set* of searched states to avoid the expansion of visited nodes, the algorithm can be run in a reasonable amount of time, due to the small search space and high frequency of repeated states.

BFS guarantees an optimal solution due to expanding each level of the state search tree sequentially, always choosing the smallest possible solution.

B. Depth-First Search

Depth-First Search (DFS) analyzes all of the states for a certain branching path before backtracking to another branch. To accomplish this, all of the nodes that need to be analyzed are kept in a *Stack*. The next state to be analyzed is on the top of the *Stack*. When analyzing a state, the next possible states are generated and placed on top of this structure.

However, this approach does not always reach a solution, due to the possibility of loops in the state search tree, which leads to infinite exploration of the same looping states. In order to avoid this, an auxiliary *Set* was used to keep track of visited nodes, thus excluding the possibility of loops.

DFS does not guarantee and optimal solution due to the order in which it expands the search tree - a deeper level can be expanded before a more shallow level and thus a larger

(and therefore non-optimal) solution reached before a smaller one.

C. Iterative-Deepening Depth-First Search

Iterative-Deepening Depth-First Search (IDDFS) executes a **DFS** with a certain limited depth, repeating this process with a larger value for this limit each time until it reaches a solution.

Unlike **BFS**, this algorithm does not need to store all of the states to explore in memory simultaneously and unlike **DFS** it cannot enter an infinite loop.

IDDFS guarantees an optimal solution due to expanding each node of a given level before moving to the next level of the search tree, gurantying the smallest possible solution.

D. Greedy Search

Greedy Search expands the next node based on the value given to it by the chosen *Heuristic*, always expanding the node with the lower value. To accomplish this, all of the nodes that need to be analyzed are kept in a *Priority Queue*. The next state to be analyzed is on top of the *Priority Queue*. When analyzing a state, the next possible states are generated and inserted into this structure, which will reorder them as needed.

This algorithm's efficiency depends on the chosen *Heuristic*. Even when using an *Admissible Heuristic*, this algorithm does not guarantee an optimal solution due to not necessarily expandsing the nodes in a more shallow level before expanding nodes at a deeper level.

E. A* Search

A* Search expands the next node based based on the value given to it by the sum of the current state's weight and the chosen heuristic. To accomplish this, all of the nodes that need to be analyzed are kept in a *Priority Queue*. The next state to be analyzed is on top of the *Priority Queue*. When analyzing a state, the next possible states are generated and inserted into this structure, which will reorder them as needed.

Both the algorithm's efficiency and solution length depend on the chosen *Heuristic*. In order for this algorithm to achieve the optimal solution, the chosen heuristic must be an *Admissible Heuristic*.

F. Heuristics

Heuristics are problem specific functions that estimate the cost to reach the objective state.

For an heuristic to be **Admissible** it cannot, at any point, overestimate the number of steps it takes to reach the goal state. Thus, when evaluating a state, the calculated value must be lower than or equal to the state's real value, never overshooting the solution.

1) Heuristic 1: The first heuristic consists of counting the number of pieces between the special piece and the exit. These pieces need to move at least once so the piece can have a clear path to the exit, making this heuristic **Admissible**.

2) Heuristic 2: The second heuristic consists of attributing a value to every piece between the special piece and the exit. This value is 1 for pieces that can move and 2 for pieces that are blocked. This heuristic is not **Admissible**, because moving one piece can free 2 pieces at the same time, therefore the heuristic can overestimate a state's value when multiple pieces can be freed by one movement. Despite this, this heuristic can reach the optimal solution for most puzzles when used in the **A* Search Algorithm**.

VII. Experiments and Results

Note: All of the below described tests were ran using *PyPy* version 3.6.

Several tests were ran, using all of the implemented algorithms and heuristics (except some cases where the runtime or memory overhead of the algorithm was not manageable - such was the case for Iterative Deepening for Long Puzzles) for different *Categories of Puzzles*.

A. Puzzle Categorization

A selection of some interesting levels of the game was grouped into four categories - Easy, Medium, Hard and Long. This grouping was done based on the number of possible states since the initial configuration and the length of the solution, as these are the main determinants on how difficult it is to find a solution, and furthermore the optimal one. Eight Easy and Hard and ten Medium and Long levels were selected for evaluating the algorithms' performance, these presenting some of the most interesting initial piece configurations.

Easy and **Medium** levels have simillar lengths for their optimal solutions (around 10 moves), differing in the number of possible states - **Easy** levels have several hundreds of possible states while **Medium** levels have tens of thousands.

Hard levels have a much higher number of possible states (hundreds of thousands) and a larger solution length (around 20 moves).

Finally, **Long** levels do not necessarily have a large amount of possible states (ranging between thousands and tens of thousands), but instead have a much larger solution length (around 50 moves).

B. Perfomance analysis

After running all of the implemented algorithms described in section VI for each of the selected puzzles, the following data was collected:

- Answer Length
- Execution Time
- Number of Expanded Nodes

The full experimental data obtained in all the different experiments, using the above described metrics, can be be found in section IX.

By analyzing the **Answer Length**, it is possible to directly measure how good the algorithm's solution is, by checking against the optimal solution of the puzzle (that has the minimum amount of moves required to reach the **Goal State**).

The solution lengths achieved with all algorithms in the easy puzzles were the following:

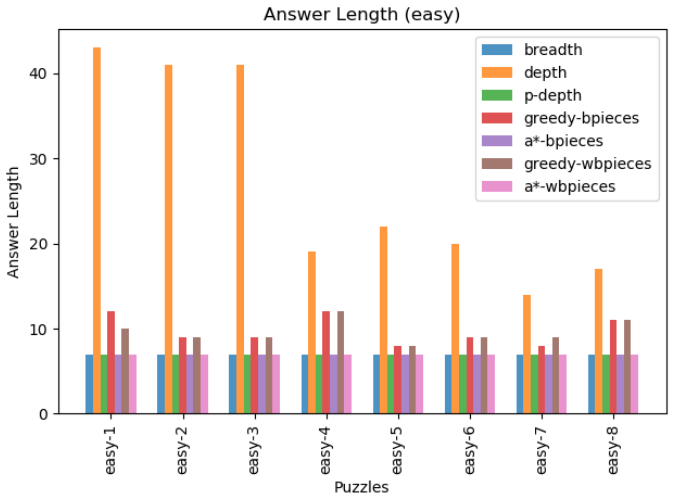

Fig. 5. Solution length in easy puzzles

As expected, the **BFS**, **Iterative Deepening** and **A*** algorithms always reached an optimal solution, being that **DFS** almost never reaches a short solution. This difference is even more visible in the long puzzles:

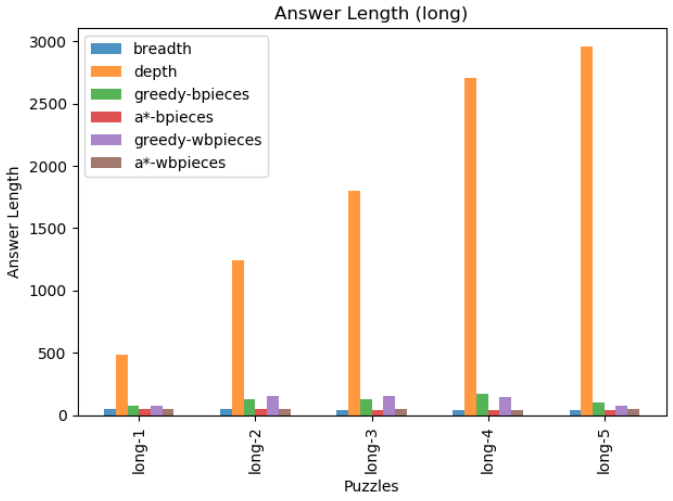

Fig. 6. Solution length in long puzzles

Additionally, by inspecting the **Execution Time**, some insight is gained on the algorithm's **Time Complexity**, which differs for each different algorithm. The solution execution times achieved with all algorithms in the hard puzzles were the following:

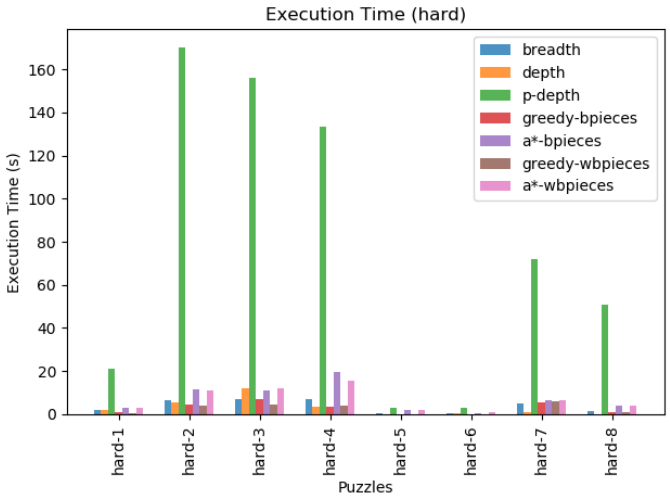

Fig. 7. Solution execution time in hard puzzles

In most of the puzzles, **DFS** quickly reaches a solution, although its length is much larger than the other achieved solutions. The same applies to the **Greedy** algorithms, that also provide a solution in a short time, although with a much better quality (not optimal, but with a much shorter length that DFS).**BFS** and **A***(using heuristic 1) quickly achieve an optimal solution. The optimization made to **BFS** resulted in a drastic increase of performance of the algorithm's execution time, it consisting of using a *Set* to save all of the visited nodes, avoiding the expansion of visited nodes in the search tree. Due to the fact that the considered problem has a very large amount of repeated states (it is possible to reverse any movement done at any given moment), this increase in performance allows **BFS** to achieve the overall best performance when it comes to execution time.

Finally, by studying the **Number of Expanded Nodes**, both the algorithm's **Space Complexity** and how fast it approaches the optimal solution (or any solution in some cases) can be inspected. The number of expanded nodes in the solutions achieved by all of the algorithms in the medium puzzles were the following:

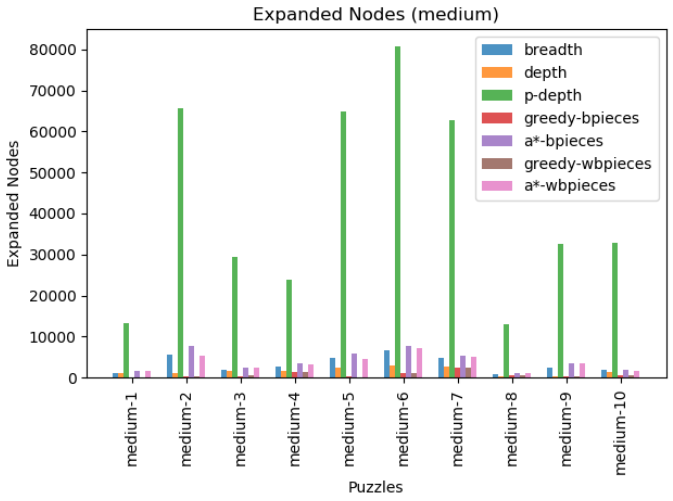

Fig. 8. Solution number of expanded nodes in medium puzzles

In most of the puzzles, **Iterative Deepening** expanded the most states, due to the fact that in each iteration a full DFS is performed (up to a given depth). The **DFS** and **Greedy** algorithms also achieved solutions without expanding a large number of nodes, although not achieving optimal solutions. The **A***(with heuristic 1) and **BFS** algorithms achieved optimal solutions without expanding an excessively large amount of nodes.

The algorithms with the best overall performance for the problems were, therefore, **BFS** and **A*** algorithms, which are able to achieve optimal solutions in a very short period of time (with low time complexity) and without needing to expand a large number of states (with low space complexity).

VIII. Conclusions and Future Work

In this project, all of the applicable algorithms that were proposed were implemented and tested (except for the Uniform-Cost search, which explores the states in the exact same order that Breadth-First search does for this specific problem and was, therefore, not necessary for the best solution study, and for the Bi-Directional algorithms, which were not possible to apply since the final objective state is not known *a priori*).

A graphical user interface was developed to visualize the different puzzles and the solutions produced by each of the algorithms.

In order to study the upsides and downsides of each of the algorithms, a thorough examination of their performance using different metrics (solution length, execution time and number of expanded states) was made. When analysing the produced solutions, it was concluded that they matched with the theoretical expectations:

- Considering the answer length, the **BFS**, **Iterative Deepening** and **A***(using heuristic 1) algorithms achieved optimal solutions and the **DFS** algorithm achieved the longest solutions.
- Considering the execution time required to reach a solution, the **DFS** and **Greedy** algorithms achieved solutions in short execution times (although not optimal). The **BFS** and **A*** algorithms were both able to reach optimal solutions in also very small execution times. The **Iterative Deepening** algorithm originated the longest execution times for most of the studied puzzles.
- Finally, considering the number of expanded states, the **A*** and **BFS** algorithms achieved the solutions with the fewest states expanded. On the other hand, the **Iterative Deepening** algorithm originated, by a large margin, solutions with the highest number of expanded states, due to the fact that, in this algorithm, multiple iterations (for different depths) of the **DFS** algorithm are applied.

In conclusion, the algorithms that achieved the best solutions according to the previously described methods where the **A*** and **BFS** algorithms, which were able to achieve optimal solutions in short execution times, without requiring the expansion of many states.

All of the objectives were successfully completed and future improvements could include the study of the different heuristics in this problem and the exploration of the studied algorithms in puzzles of bigger sizes.

References

[1] ”Unblock me FREE.”. Google Play. February 28, 2019.https://play.google.com/store/apps/details?id=com.kiragames.unblockmefree.

[2] Fogleman, Michael. ”Solving Rush Hour, the Puzzle.”. July, 2018. https://www.michaelfogleman.com/rush/.

[3] ”Bitboard.”. Wikipedia - The free Encyclopedia. December 6, 2018. https://en.wikipedia.org/wiki/Bitboard.

[4] Littman, Michael. ”Programming Assignment P1 - What A* Rush”. Priceton. 2012. https://www.cs.princeton.edu/courses/archive/fall12/cos402/assignments/programs/rushhour/.

[5] Findling, Rainhard. ”The RushHour Puzzle an Artificial Intelligence Toy Problem.”. April 4, 2012. http://geekoverdose2.rssing.com/browser.php?indx=39804402&item=1.

IX. Annex

A. Answer Length in Easy puzzles

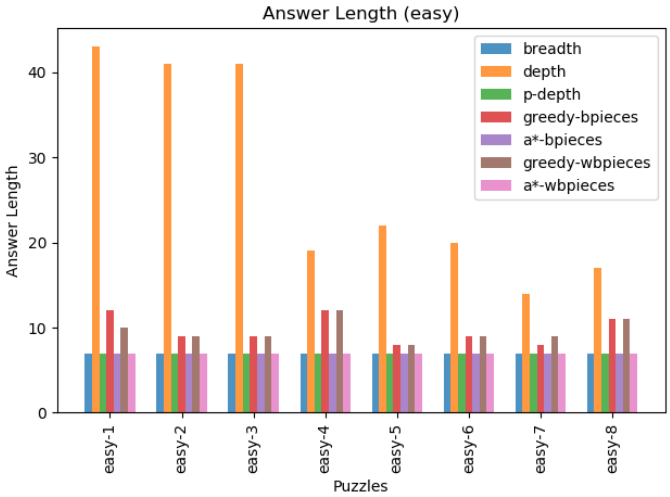

Fig. 9. Answer Length in Easy puzzles

B. Answer Length in Medium puzzles

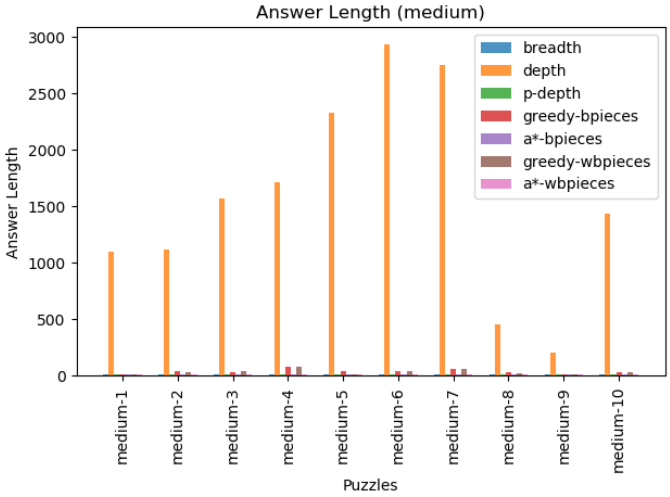

Fig. 10. Answer Length in Medium puzzles

C. Answer Length in Hard puzzles

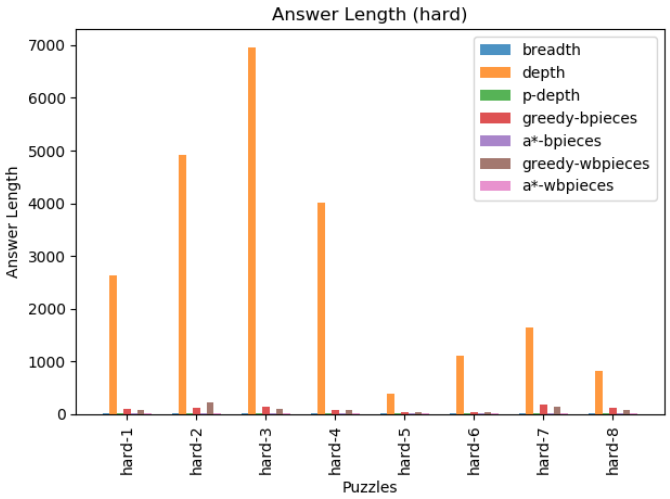

Fig. 11. Answer Length in Hard puzzles

D. Answer Length in Long puzzles

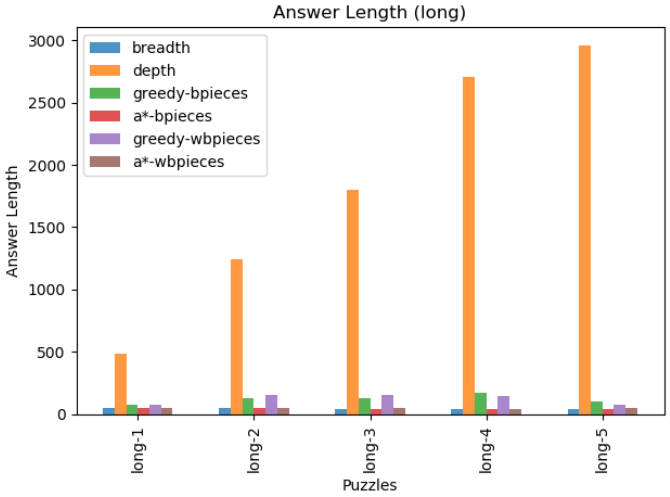

Fig. 12. Answer Length in Long puzzles

E. Execution time in Easy puzzles

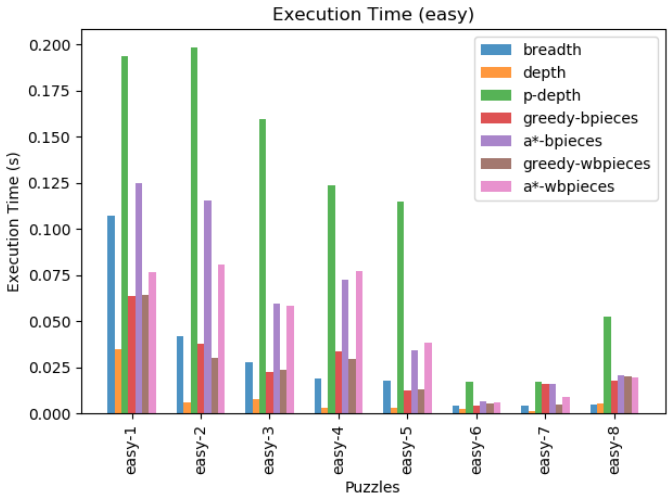

Fig. 13. Execution time in Easy puzzles

F. Execution time in Medium puzzles

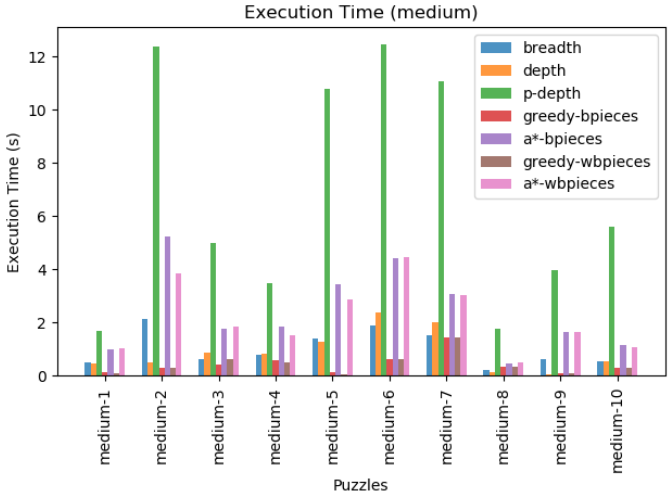

Fig. 14. Execution time in Medium puzzles

G. Execution time in Hard puzzles

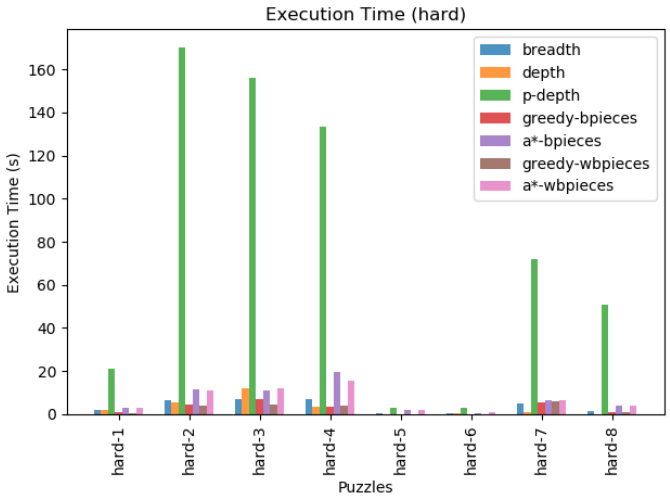

Fig. 15. Execution time in Hard puzzles

H. Execution time in Long puzzles

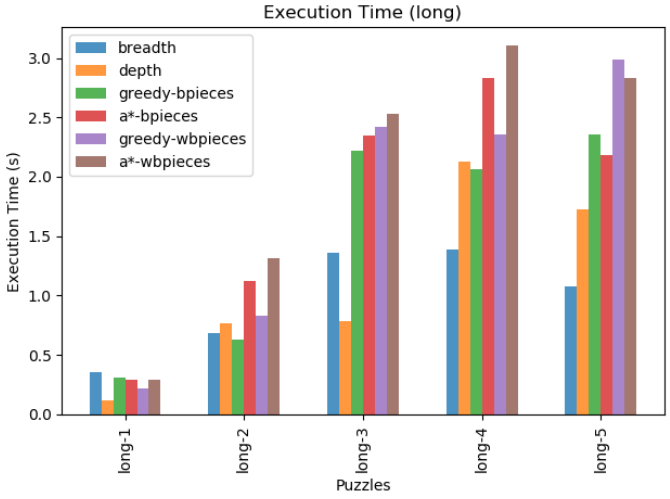

Fig. 16. Execution time in Long puzzles

I. Number of expanded states in Easy puzzles

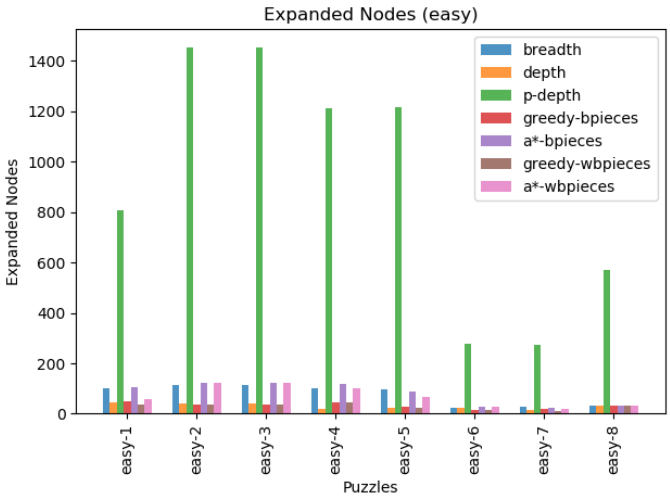

Fig. 17. Number of expanded states in Easy puzzles

J. Number of expanded states in Medium puzzles

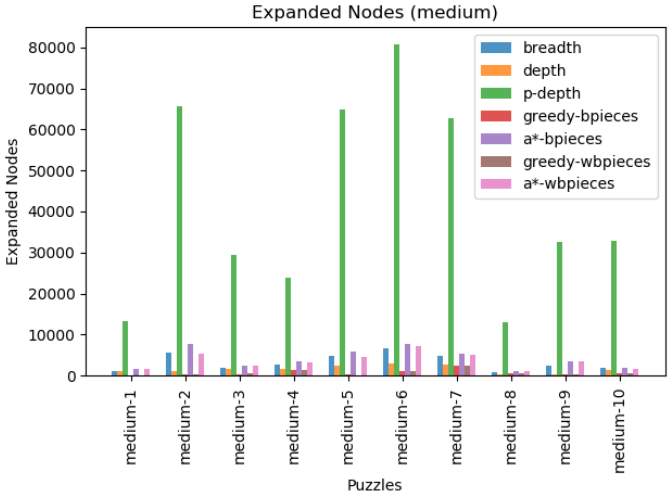

Fig. 18. Number of expanded states in Medium puzzles

K. Number of expanded states in Hard puzzles

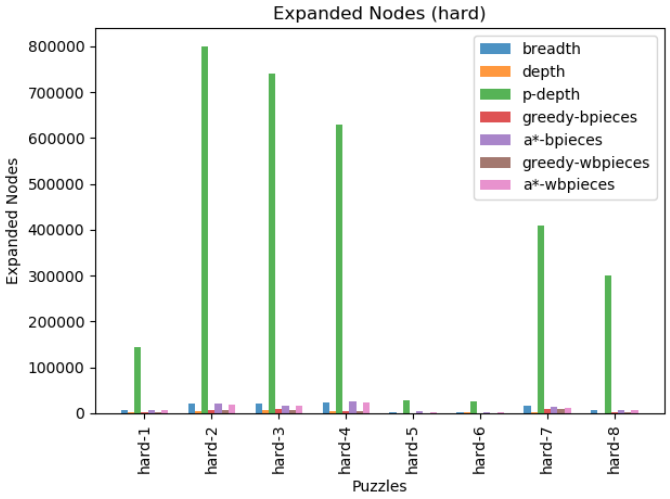

Fig. 19. Number of expanded states in Hard puzzles

L. Number of expanded states in Long puzzles

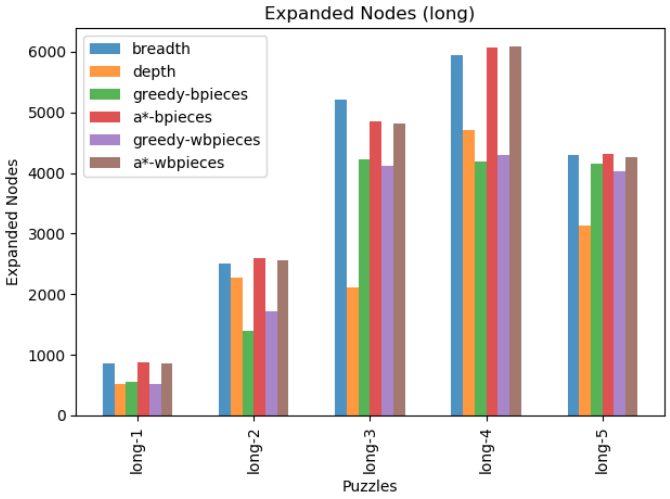

Fig. 20. Number of expanded states in Long puzzles