Algorithms and Heuristics to Solve Free-Cell Solitaire Games

CSCI 4511W: Final Project Report

Leo Cui and Ryan Diaz

December 14, 2022

1 Abstract

In this paper, the use of different search algorithms in the game of free-cell solitaire was explored. Involved in this process was the evaluation of the performance of A iterative deepening search, A* search, depth-first search, and breadth-first search on a range of pre-set solitaire configurations. The experimental results showed that A* iterative deepening search with the "difference-from-top" heuristic performs the best overall. These findings provide insight into the strengths and limitations of different search algorithms in free-cell and have implications for future work in similar areas.*

2 The Problem

Free-Cell Solitaire (FCS) is a variation on the classic 52-card single-player game that presents an initial state of all 52 cards in the deck face-up in a configuration. Unlike traditional solitaire, FCS introduces four "free cells" that allow players to move cards around the board more easily. Although the goal of orderly stacking each suit in their respective "foundation" piles is the same, the drastic change in the initial state has allowed the vast majority of random deals to be solvable, making FCS more akin to a puzzle rather than a game partly governed by chance. Having manually played FCS before, it would be interesting to implement an automatic solver, since the vast majority of initial states have a solution and the game itself can be modeled as a search problem.

There are several rules outlined in FCS that make it a more complex game than other single player games (IE: minesweeper). First, it is important to describe the different areas of the field. The field of play is divided into three parts: columns, foundation, and free-cells. Columns are eight unsorted piles that hold the starting place of every card. Here, each card is visible but cannot be moved unless it is "connected" to the top most card in the column. A card is "connected" to the top if it is a sequence of decreasing order and alternating color. For example, a black 6 is "connected" if the cards on top of it follow the sequence of red 5, black 4, red three, black 2, red ace. Additionally, cards can only be moved between columns if they yield to this decrement order and alternating color sequence. Foundation is a special stack that allows a player to pile cards from Ace to King based off of their suits. As mentioned before, if each suit has been filled from Ace to King, the game is complete. Free-cells are four single-card spaces that allow for any card to be stored to reveal room for other cards below on the columns. The goal of the game is to complete each suit in the foundation. In text form, these rules may be difficult to comprehend. Therefore, it is recommended that the reader should watch a video or play a game of FCS to fully understand the game.

As a single player game, FCS provides many interesting variables in regards to developing search algorithms. As there are a total of 16 different cells (8 columns, 4 foundation, 4 free-cell), each state may have many different possible moves. This variability provides both a constraint and challenge to developing a good algorithm that can track these moves without consuming too much space. Given that each of these three areas have different functions, there exists many different variables to use as inspiration to generate better searching algorithms. Finally, given that FCS is designed to almost always have an existing solution, search algorithms can be designed to always work. The numerous constraints and challenges of FCS make it a non-trivial problem that is particularly well-suited for testing the effectiveness of different search algorithms.

3 Background and Related Works

3.1 Introduction

This section is intended to provide a comprehensive overview of the previous work done in the development algorithms used to search game states in general as well as the use of certain techniques to efficiently solve Free-Cell Solitaire (FCS) puzzles. As the project entails developing an agent with A* search techniques across multiple

different heuristics for FCS, relevant sources will be analyzed to give context to the work outlined in this paper. This review of related works will provide an overview of research done on the development and comparison of informed and uninformed search algorithms, as well as previous work done in solving various FCS game configurations.

3.2 Searching Game States

First, observe the prior work done in the area of developing techniques to search game states. As computer games are a popular application of artificial intelligence, many observations have been made on the challenges of such research. Complex decision spaces, knowledge engineering, unanticipated situations, and rhetorical objectives each pose different and important restrictions on artificial intelligence research on games [10]. Complex decision spaces refers to the inherent complexity of a computer game; in solitaire, the complexity of moves often leads to a handpicked strategy for the algorithm to follow. Therein lies an issue, as the artificial intelligence agent will only be as good as the specified strategy and not find an optimized strategy for itself. Knowledge engineering follows a similar issue, as the domain knowledge that is given to the agent must be complete for proper results. This is often a rigorous and difficult task to correctly convey domain specifications to the behavioral language for agents. Additionally, there still is the possibility of unanticipated situations that are not handled by the agent, due to space or time constraints of execution time. Finally, even if all proper work is done, it is possible that the end result of an agent does not quite match the rhetorical objective that was required. Stating a goal such as, "win in as few moves as possible," may be difficult to achieve with specified parameters for the agent. As these are some relevant difficulties that prior research has revealed in computer game artificial intelligence research, close attention must be payed to particular cases of free-cell solitaire. Accounting complex decision spaces, engineering domain knowledge, handling unexpected situations, and framing rhetorical objectives in the right form will be critical factors to consider.

3.3 Comparison of Search Algorithms

Since the game of FCS can be modeled as a search problem, it is necessary to compare the performances of both uninformed and informed search algorithms and determine which algorithm provides the most efficient time and space cost considering the optimality of the solution found. The breadth-first search algorithm was shown to guarantee that a solution with optimal length would be found [8]. However, the memory requirement of this algorithm has proven to be too costly, as breadth-

first search will exhaust its memory while searching the space of a game that is even slightly extensive. Indeed, searching just twelve moves ahead in a game of FCS will require exploring upwards of 29 billion nodes [2]. Depth-first search is an alternative strategy that somewhat relieves the memory problems of breadth-first search, but it comes at the cost of potentially running infinitely or finding a suboptimal solution [8] [2]. One way to address the memory problems of breadth-first search and the optimality issues of depth-first search is with an iterative-deepening depth-first search, which performs multiple depth-first searches at incrementally increasing depth cutoffs. For FCS, an algorithm known as Heineman's Staged Deepening (HSD) uses certain rules of the game, such as the fact that some moves are irreversible, to construct an optimized version of iterative-deepening search that proved to be the most effective FCS solver for a period of time [2].

3.4 Iterative-Deepening A* and its Optimizations

A further improvement to the algorithms involved in searching the game states comes in the form of informed search algorithms such as the A* algorithm. The advantages of this algorithm are evidenced in an experiment comparing depth-first search, breadth-first search, and A* search performance on a simulated maze, where A* search achieved the cost optimality of breadth-first search while also obtaining a run time comparable to that of depth-first search. In other words, A* search was able to combine the strengths of the other two search algorithms [4]. However, just like breadth-first search, the A* algorithm is also memory-intensive, although there are ways to improve upon this algorithm [12] [3]. An enhancement to the A* search algorithm involves an application of the iterative-deepening approach, which produces an algorithm known as Iterative-Deepening A* (IDA*). IDA* acts much in the same way as A*, though it performs successive A* searches with increasing depth limit cutoffs in the form of total path cost limits. IDA* has been shown to be optimal given an admissible heuristic, and also expands the same asymptotic number of nodes as A*, so IDA* presents no definitive drawbacks compared to its original counterpart [8]. The main shortcoming of IDA* is the overwhelming amount of redundant exploration of nodes, so some optimizations have been implemented to try and reduce this redundancy. One such optimization is the use of transposition tables: a hash table that essentially stores states already explored. The added lookup of this table during the search results in a slight slowing of performance over A*, but IDA* with the use of transposition tables has been found to succeed in trials where A* failed due to memory limitations [7]. Another technique to improve the performance of IDA* has been the use of parallel processing to search through several branches

of the state space at once, which reduces the time complexity of the search based on the number of threads used [13] [11]. IDA* provides many advantages over other search algorithms while also having many potential opportunities for optimization, so it will be one of the main focuses of the following work in solving FCS puzzles.

3.5 Heuristics and Strategies Used for Search

Since A* and IDA* are the algorithms of choice to search for solutions to FCS game states, it is necessary to compare the performances of different heuristics that evaluate the goodness of a given state of FCS. Much prior work has already been done in formulating heuristics for FCS as well as developing strategies for using these heuristics. One such strategy known as the "rollout method" has been used on a variant of solitaire very similar to FCS known as "thoughtful solitaire", where all of the cards start face-up in a configuration mirroring the classic Klondike solitaire. The rollout strategy involves improving a single initial strategy based on the best action found as the search progresses, with the strategy gradually upgrading across multiple iterations of the rollout process. This algorithm was able to solve up to 70% of thoughtful solitaire test configurations, though higher iterations of the rollout process lead to a much longer computation time [14]. Later work on the rollout method involved utilizing different heuristics depending on the stage of the game, which led to an 82% solution rate [1]. For FCS, multiple heuristics have been devised, the most optimistic of which is the total number of cards not in a foundation pile. A heuristic involving the number of moves needed to resolve all "deadlocks", which can be represented as a cycle when the FCS configuration is translated into a directed graph, has resulted in optimal solutions being found for all tested FCS puzzles out of the first 5000 in the Microsoft Free-Cell dataset [6]. Another strategy involved the use of genetic programming and evolutionary algorithms with multiple different heuristics, including the heuristic used by Heineman's Staged Deepening (HSD) algorithm mentioned previously. Fitness of a heuristic was determined by how much the heuristic reduced the search space compared to the HSD heuristic. Hillis-Style Coevolution was also used, which involves evolving a population of solutions alongside a population of problems; this allows the evolved solutions to adapt to a wide variety of problems. The coevolution technique of evolving heuristics was able to solve 98% of all presented FCS problems, a major improvement over using just the HSD heuristic [5]. Although these advanced techniques will not necessarily be used in this project, they represent the complexity of the FCS puzzle and the possible necessity for us to simplify the problem a bit by reducing the deck size.

4 The Approach

In order to have a working version of FCS to build an automatic solver for, an open-source repository that implemented FCS in Python was used [9]. The code base contained functions that allowed for manual shuffling of the deck as well as applying specific moves on the board via a string of characters, and these features would be used in various aspects of the search algorithms. A structure similar to that of the AIMA code for search methods was adopted when writing the search algorithms: a search algorithm would operate on nodes whose behaviors are dictated by a 'SolitaireBoardProblem' class. These nodes encoded states of the solitaire board at a given point in time, which were represented by a dictionary that separated cards in the free-cells, the foundation, and each of the 8 columns. The cost of an action that led to any of these nodes was set to be constant; most of the variation when evaluating nodes would be present in the heuristics used during informed search. The 'SolitaireBoardProblem' class defined methods for finding all possible moves from a given state for node expansion via a brute force algorithm, as well as applying actions to states and checking if a given state is a goal state. As a note, although heuristics would be considered for the informed search algorithms, some preliminary pruning of actions during move generation was done to speed up the run time and reduce the memory usage of all the searches. In particular, legal actions that swapped cards around empty columns or free-cells were omitted to prevent the expansion of redundant moves.

Solving instances of FCS puzzles involved using both uninformed and informed search algorithms, with a special focus on the informed search algorithms and the various heuristics that could be applied. For the uninformed search algorithms, the breadth-first (BFS) and depth-first (DFS) searches were analyzed to a certain extent, while the A* and iterative-deepening A* (IDA*) algorithms made up the informed searches that were used. Most search algorithms used a data structure to track all the previously visited states in order to prevent infinite loops and redundant expansion of nodes. The IDA* algorithm was implemented in two ways: as a series of normal A* searches with gradually increasing cost cutoffs, and as a modification of the iterative deepening DFS algorithm where nodes were cutoff by cost and node exploration was ordered by the heuristic. Six separate heuristics were utilized when measuring the performance of A* and IDA* [5]. The most basic of these heuristics (the **foundation heuristic**) counted the total number of cards not in the foundation piles. The **well-placed heuristic** found all cards in the columns that were part of a pile of descending rank and alternating colors. The **free heuristic** counted the

number of empty free-cells and columns that were on the board. The **difference-from-top heuristic** calculated the difference in the averages between the top cards in the columns and the top cards in the foundation piles. Finally, the **lowest-home-card heuristic** and the **highest-home-card heuristic** looked at the lowest and highest rank cards respectively in the foundation piles.

Due to the massive state space that a game of FCS using the full deck of 52 cards presented, the likelihood of an informed search algorithm with a relatively simple heuristic and no significant run time or memory optimizations to solve an FCS puzzle would be incredibly low. Therefore, most of the simulations of FCS games using the implemented search algorithms were done on decks with far less than 52 cards. This allows for a more substantial comparison of search algorithms and heuristics, since they will be able to terminate and a reasonable measurement of their memory and run time performance metrics can be taken. Search algorithms such as A* and BFS are known for consuming large amounts of memory in order to store nodes in their priority queues, so a smaller deck would allow them a reasonable chance to solve an FCS puzzle with the resources they are given. Although there are certainly modifications that can be made to these algorithms to allow them to handle larger games, part of the comparison of these algorithms will involve determining the largest deck size where they can still perform well.

5 Experiment Design and Results

To test the validity and functionality of the search algorithms, all searches were first tested on small deck sizes, guaranteeing that the searches would find a solution in a short amount of time. This also allows for the actual solutions generated by the algorithms to be short enough for manual inspection. After the solutions found by each of the searches were verified to be accurate, the algorithms were tested on larger deck sizes. The process for running and comparing these algorithms involved running each search on 20 FCS games using a given deck size (this size was determined by the highest "rank" that would be present in the deck; for example, a deck with a highest rank of 6 would have 24 cards). For each iteration, a new deck is generated; the sequence of generated decks was set up to be completely deterministic as dictated by a seed value in order to be able to ensure that all algorithms were operating on the same games. The use of a seed also allowed for the same sequence of decks to be used if a new algorithm or heuristic were to be tested. In order to introduce variability and ensure that the search algorithms were not being hindered by a bad seed, a different seed number was used for each round of simulations across different deck sizes.

The performance of each algorithm across the 20 iterations was evaluated on a number of different criteria. These metrics included the percentage of given games that were solved, the average solution length (i.e. the number of moves in the solution), the average number of states explored in a successful run, and the average process time for both successful and unsuccessful runs. An "unsuccessful" search fails to find a solution before a predefined cutoff for the number of nodes explored. Although all search algorithms could theoretically find a solution given enough time, the cutoff defines a notion of a reasonable time to terminate.

Algorithm (heuristic)	**Success Rate**	**Average Solution Length**	**Average States Explored**	**Average Time Taken**
Depth-first search	75%	184.2	187.2	11.4
A* Foundation	85%	16.6	45.1	2.7
A* Well-placed	85%	17.1	69.9	5.9
A* Free	100%	18.2	47.7	1.8
A* Difference-from-top	100%	16.9	23.1	0.9
A* Highest-home-card	95%	16.9	32.5	1.8
IDA* Foundation	55%	16.5	42.4	1.6
IDA* Well-placed	60%	16.8	45.8	1.7
IDA* Free	95%	19.1	74.7	2.0
IDA* Difference-from-top	100%	17.7	40.9	1.5
IDA* Highest-home-card	100%	17.4	42.1	1.4

Figure 1: Performance metrics for search algorithms across 20 games using a deck with a highest rank of 4 and a search cutoff of 400 states.

Algorithm (heuristic)	**Success Rate**	**Average Solution Length**	**Average States Explored**	**Average Time Taken**
Depth-first search	30%	208.7	211.7	18.0
A* Foundation	35%	21.0	63.7	5.9
A* Well-placed	35%	21.6	130.3	28.1
A* Free	65%	22.8	128.3	14.3
A* Difference-from-top	100%	23.1	87.4	9.4
A* Highest-home-card	80%	22.3	114.9	16.5
IDA* Foundation	25%	21.0	119.6	6.0
IDA* Well-placed	20%	21.5	152.5	7.8
IDA* Free	55%	23.7	156.9	5.7
IDA* Difference-from-top	90%	24.4	74.2	3.4
IDA* Highest-home-card	80%	24.0	62.6	2.6

Figure 2: Performance metrics for search algorithms across 20 games using a deck with a highest rank of 5 and a search cutoff of 500 states.

It is worth mentioning a few aspects of the reported performance metrics shown in Figures 1 and 2. First, the data reported for the IDA* algorithms corresponds only to the iterative-deepening DFS search, since the repeated A* search approach was found to perform in a very similar manner to its original A* counterparts in terms of success rate. In addition, the BFS algorithm and the lowest-home-search heuristic for A* and IDA* were both not tested for decks with a maximum card rank of 4 or higher. This is due to the fact that all three of these algorithms had a success rate of 0% on decks of an even smaller size; it is reasonable to assume that they would perform similarly on larger decks. In general, if an algorithm demonstrated sufficiently bad performance on a given deck size, they would not be tested during simulations of FCS games with a higher maximum card rank.

With this principle in mind, only the difference-from-top, highest-home-card, and free heuristics for A* and IDA* were tested on decks with a maximum card rank of 6. While the difference-from-top and highest-home-card heuristics were still able to achieve success rates of 70% and 60% respectively, the free heuristic was not able to solve more than 20% of games. On decks with a maximum card rank of 7, both the difference-from-top and highest-home-card heuristics for IDA* dropped to success rates of 10% and 25% respectively. No more testing beyond this was done, since it was assumed that these algorithms would perform no better on larger deck sizes.

6 Analysis of Results

Overall, there is a great disparity that can be seen between the uninformed search and the informed searches in both simulations. Although DFS (surprisingly) achieved a somewhat similar success rate as the informed searches, the number of states explored and the length of the solution found were far from optimal compared to the other searches. On a related note, the search time for the DFS algorithm was almost ten times longer than some of the informed searches. Despite the worse performance of DFS, by raw numbers it performed way better than was expected in terms of success rate. It was assumed that like BFS, DFS would also begin to fail on larger deck sizes, but it somehow managed to keep up with the informed algorithms. This might be attributed to the ordering of nodes explored due to the move ordering generated by the 'SolitaireBoardProblem' class; the ordering of moves might have been excessively beneficial for DFS even with a search cutoff. The benefits of move ordering would not apply to BFS, which has to explore an entire level of the tree (and thus all possible actions) before moving on to a deeper level anyways.

Observing the table, the strongest performing algorithms were the two informed searches A* and IDA*. At first glance, Figure 1 reveals little difference between A* and IDA* in terms of average time taken. Inspecting a bit deeper, it is noticeable from Figure 2 that IDA* search performs searches significantly faster than its A* counterpart. The largest difference can be seen with Highest-home-card heuristic, where IDA* search performs a factor of 6.346 faster than its counterpart. Given these results, IDA* search can be seen as more time-efficient in comparison to A* search. This may be due to the ability of IDA* to stop at an early depth level compared to A* search, as it searches level by level rather than the entirety of a branch. Noting the significant change between Figure 1 and Figure 2, it is hypothesized that the inclusion of more ranks would highlight this difference even more.

In comparing the performances of the heuristics used in both A* and IDA*, there didn't seem to be any heuristic that especially stood out for any of the search metrics for a deck with a highest card rank of 4. One heuristic that seemed promising was the Difference-from-top heuristic, which managed to solve all 20 of the presented configurations while also taking the least time and exploring the least number of nodes on average across all of the heuristics for A*. This stellar performance was magnified in the simulation involving a deck with a highest rank of 5, where this heuristic was the only one to solve all the deals for the A* searches, and also achieved the highest success rate across the IDA* search algorithms. On larger deck sizes, the Difference-

from-top heuristic was still able to achieve a respectable success rate, although it dropped drastically (along with the other heuristics) on decks with a highest card rank of 7 or greater. One aspect of the performances of each of the heuristics was that the average solution lengths differed; in other words, A* and IDA* were not optimal in some cases. Theoretically, since A* is optimal given an admissible heuristic, the lengths of all solutions found should be the same, but this was not the case. This discrepancy may be attributed to how the heuristics were implemented: modifications were done to some of the heuristics to improve their performance, but this may have made them inadmissible in some cases.

Finally, there are a few interesting miscellaneous findings during the experiment that should be discussed. One important issue is the existence of memory consumption during each algorithms. Due to the generation of a node, a data structure that holds the current state of the board persists in memory as it is still being used for later for backtracking. As a consequence of this limitation, BFS failed to pass any of the tests presented in Figure 1 and Figure 2. Memory was not the only limitation found during execution; programming language was another issue. In both figures, the average states explored never exceeds 300, yet the average time taken is often longer than a second. In previous works, thousands of states could be explored in a fraction of the time consumed by the program. These works often used more powerful programming languages like Java or C++ to perform their work, an advantage these algorithms lack. If this research was done again using a different program, results may have been able to extend well beyond the bottleneck of rank 5. Lastly, a reader may notice that although IDA* performed significantly faster than its A* counterpart, the success rate is not always as high. This may due to IDA* search being implemented in a method that has dynamic node cutoffs. Since this paper's program of IDA* search was based off of DFS, it may potentially have hit the cutoff earlier than its A* counterpart. This issue may be resolved by re-implementing IDA* to have a different node cutoff that doesn't interfere with its success rate performance. Another possible fix is to not immediately end when finding a solution state, allowing it to continue exploring for a more optimal solution.

7 Conclusions and Future Work

As a result of the findings stated above, it can be concluded that IDA* search is the most effective algorithm in finding solutions for FCS. The strong results of this search are due to its ability to halt at an early search depth, compared to A* search. Due to this trait, A* iterative deepening can explore less nodes and use less

memory during runtime, and it finds these solutions much faster than normal A* though it sacrifices a bit of accuracy and optimality. In addition, this paper has determined that the best-performing heuristic is the difference-from-top heuristic, achieving superior results in success rate, the number of states explored, and the average time taken on a successful search. The heuristic was also one of only two that was still able to find solutions to decks with a maximum rank of up to 7. Alternative heuristic methods that also proved valuable include giving higher weight to the foundation stack, empty cells, and how close a foundation pile was to being completed. These are the findings for the performance of various search algorithms for FCS under reduced deck sizes.

As for future work, there are several areas of potential interest that have emerged from this study. The most critical issue encountered during the study was the limitation of time to test the algorithms against a complete deck of FCS. In this aspect, assets like more time, stronger computer processors, and multiple machines may help expand the scope of experiments to the unaltered form of FCS. Alternatively, experimental design decisions could also be modified to expand this work. Python, the program used to process game states and algorithms, is inherently a slower programming language than its counterparts. Using a faster language for compilers, such as C, could provide significant improvements to the range of these search algorithms. Future work could also focus on finding optimizations within the code written for this project to handle this drawback. Another area of interest in relation to this work is the possibility of adding more diverse and complex heuristics. Many of the heuristics used in this study focus on a particular element of the game state. In retrospect, well-built algorithms that encompass more variables is a large opportunity for more data that this paper has not explored. This future work could give insights into the design of more complex heuristics and respective performance. FCS is also a single form of solitaire that has rules that test the effectiveness of a search algorithm. Each card being revealed and the existence of free-cells provides a state that is more controllable to test than other solitaire variants. Therefore, it would be interesting to apply some of the heuristic methods discussed in this paper to other solitaire games with different rule sets.

8 Contributions

This project represents the cumulative work of both Ryan Diaz and Leo Cui. Ryan contributed to the coding portion of the project by implementing the 'SolitaireBoardProblem' class that allowed the search algorithms to operate on the exist-

ing open-source code base, as well as implementing the search algorithms themselves and the general mechanisms for testing these algorithms. Ryan contributed to this project report through parts of the Background and Related Works section as well as the Approach and Experiment Design and Results sections. Leo contributed to the coding portion of the project by modifying the existing open-source code base to accept decks of smaller sizes. He also implemented all of the heuristics that were used when testing the A* and IDA* algorithms, and also assisted in implementing various parts of the 'SolitaireBoardProblem' class. Leo contributed to this project report through parts of the Background and Related Works, as well as the Abstract, Problem Statement, and Conclusion sections. He also constructed all of the citations used. Both Ryan and Leo collaborated on the Analysis section of this report.

References

[1] Ronald Bjarnason, Prasad Tadepalli, and Alan Fern. Searching solitaire in real time. 2007.

[2] Chapp Brown and Kylie Beasley. Freecell solitaire optimization. 2021.

[3] Julien Castiaux. So you want to solve freecell? an academic journey into crafting a solitaire game solver. 2021.

[4] Iloh Princess Chinemerem. A comprehensive and comparative study of dfs, bdf, and a* search algorithms in a solving the maze traversal problem. 2007.

[5] Achiya Elyasaf, Ami Hauptman, and Moshe Sipper. Ga-freecell: Evolving solvers for the game of freecell. 2011.

[6] Malte Helmert and Gerald Paul. Optimal solitaire game solutions using a search and deadlock analysis. 2016.

[7] Akihiro Kishimoto, Alex Fukunaga, and Yuima Akagi. On transposition tables for single-agent search and planning: Summary of results. 2010.

[8] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search*. 1985.

[9] Katy Lavallee. pyfreecell, 2022.

[10] Ashwin Ram, Santiago Ontanon, and Manish Mehta. Artificial intelligence for adaptive computer games. 2007.

[11] V. Nageshwara Rao, Vipin Kumar, and K. Ramesh. A parallel implementation of iterative-deepening-a*. 1987.

[12] Luis Henrique Oliveira Rios and Luiz Chaimowicz. A survey and classification of a* based best-first heuristic search algorithms. 2010.

[13] Shimul Sutradhar, Shabrina Sharmin, and Saiful Islam. A review on ida* - iterative deepening algorithm heuristics search. 2022.

[14] Xiang Yan, Persi Diaconis, and Benjamin Van Roy Paat Rusmevichientong. Solitaire: Man versus machine. 2021.