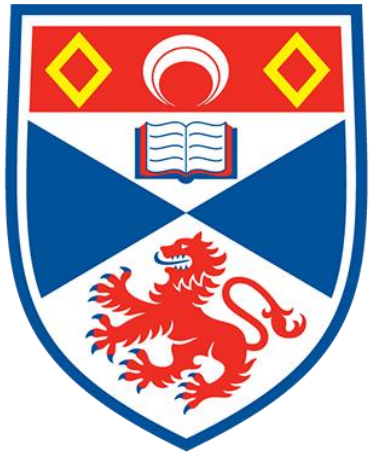
Playing Solitaire Automatically

Author: Santhosh Basina

Supervisor: Ian Gent



University of
St Andrews

March 22, 2024

Abstract

Solitaire is a genre of card games where the aim is to arrange the cards in some systematic order by following a set of rules which dictates how the game should be played. Originally known as Patience, this genre of games is believed to have originated in the late 1700s and early 1800s in Europe. Since then, it has exponentially grown in popularity and has even been featured in literature, film and art. This widespread popularity has been especially noticeable in the 20th century due to the rise in personal computers where it has become a staple game in operating systems such as Windows. There are many variations of Solitaire, but I was particularly interested in focusing on Pyramid for this project where the aim is to develop a solver which solves the game automatically by interacting with the Windows Solitaire application.

Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project report is 8,658 words long, including project specification and plan.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web.

I retain the copyright in this work.

Contents

1 Introduction

Pyramid is a card game played with a standard 52-card deck. The objective is to remove pairs of cards that add up to 13. The first 28 cards are dealt in such a way that it forms a pyramid with all the cards facing up. There are 7 rows of cards in this pyramid where the top row has one card, and the bottom row has 7. Cards in the upper rows are blocked by cards in the lower rows, so at the start only the first 7 cards in the bottom row are playable. But as the lower rows get cleared out, the cards in the upper rows become playable. The remaining 24 cards form the stock and are facing down except for the card on top of the stock which is facing up and is playable. Accessing the next card in the stock involves placing the current card that is on top of the stock to the waste pile which initially empty. This card this is transferred is still facing up and is playable in the waste pile. However, the next card that is moved to the waste pile will be placed on top of the current card that is on top of the waste pile and so it won't be playable anymore. Each time, one card from the stock and one card from the waste pile are available to be played. The stock can be cycled through a total of three times before no more cards can be drawn from it. The aim of the game is to remove all the cards in the pyramid before this happens.

Below is a screenshot of what the game would look like with the key components of the game labelled.

Figure 1: Game window screenshot

Winning in Pyramid is not just about luck but is especially about having the necessary skills and strategies when solving it, as moves need to be planned before being made. Players are required to make strategic decisions to calculate the correct sequence of moves that would lead to all the pyramid cards being cleared out without reaching a dead-end where no more moves can be played. This implies that every game in Pyramid

can be solved programmatically using systematic techniques. Therefore, I wanted to see how I could employ such techniques along with automation to develop a solver that can solve every instance of the game entirely on its own.

There are three main components that are required for the solver. The first is that it needs to have the ability to scan the game window to detect and recognize each card. The second is that it needs to be able to use this information to calculate the correct sequence of moves that would lead to a winning state. Lastly, once a solution has been found, the calculated winning sequence of moves needs to be executed using GUI automation.

The solver I developed implements and integrates these three components. At the core of the solver is a tree traversal functionality which searches for the optimal move and this component serves as the focus of this project. There are certain limitations in the functionalities for recognizing each card and finding the optimal solution, which will be discussed in detail later in the report.

2 Project Objectives

The requirements for the project have been split into primary and secondary objectives. The primary objectives mainly focus on having the baseline implementation of the solver, whereas the secondary objectives focus on more advanced implementations which are aimed at improving the performance of the baseline solver. Below is a list of these objectives which was originally included in the Description, Objectives, Ethics and Resources document.

- Primary objectives:
    - An algorithm that calculates potential moves and searches for the optimal move from the problem space using the appropriate tree-traversal techniques and heuristics.
    - Screen capture functionality to keep track of the game's current state from the application's graphical interface.
    - Move execution functionality for the solver to make moves within the application's interface by controlling the curser to click, drag and drop cards.
    - User-interface to allow players to interact with the solver.
    - Evaluation of the solver by implementing performance analytics to make measurements of its success rate, speed, and efficiency.
- Secondary objectives:
    - Optimisation of the solver to improves its performance by employing techniques such as caching to reduce the search space.
    - Integration of a Machine Learning model to enhance the solver's capacity to make the optimal move.
    - Extending the solver to be able to solve other card games.

3 Software Engineering Process

Throughout the software development phase, I followed the Agile methodology. I used Scrum as the Agile framework of choice for managing my project. This approach has several useful benefits. The two main benefits that I find appealing are the fact that it encourages iterative development and adaptivity. This is done by dividing the project into smaller and more manageable parts where each part is worked on one at a time. Each of these parts are planned out independently instead of having the entire development process planned out at the start. Another key benefit is that it encourages continuous improvement since the iterative approach enforces a feedback loop where previous failures when working on one part of the project during the assigned duration for that part can result in tasks be carried over to the next assigned duration. So, there is emphasis on continuous improvement. For these reasons and because of past experiences with Scrum, I have decided to follow this methodology.

Before starting with the development process, I created a product backlog which is a list of all the desired features that I intended on implementing. This was mainly based on the primary and secondary objectives of the project. Each of these features has a priority level associated with it, to indicate the difficulty of implementing the feature. The development then proceeded through a series of sprint cycles, where each is a time-boxed iteration which lasts 2 weeks. Before each sprint, I created a sprint backlog which is a list of a subset of the features from the product backlog which I intend on implementing during the sprint. During each sprint the actual implementation of the features takes place and then at the end of each sprint, a sprint review is done. During this event, the features that have been implemented are evaluated. Any failures in implementing a feature resulted in that feature being carried onto the next sprint as mentioned above. Through this repeated cycle, I was able to carry out the development of the Pyramid solver.

Below is screenshot of what the product backlog looked like at the start of the development process. This is from Jira which is the software I used to manage and track the development process.

Type	Key	Summary	Status
☑	SS-11	Game loop	DONE
☑	SS-1	Game state representation	DONE
☑	SS-2	Search algorithm for making optimal moves.	IN PROGRESS
☑	SS-3	Heuristics to guide the search algorithm	IN PROGRESS
☑	SS-4	Screen capture functionality to get the current state of the g...	TO DO
☑	SS-10	Mapping the screen state to the internal state of the game	TO DO
☑	SS-5	Move execution functionality using the cursor	TO DO
☑	SS-6	User-interface for the solver	TO DO
☑	SS-7	optimization of the solver's performance using techniques su...	TO DO
☑	SS-8	Integration of a Machine Learning model to improve the solv...	TO DO
☑	SS-9	Extend the solver to be able to solve other games	TO DO

+ Create

Figure 2: Product backlog

4 Ethics

There were no ethical issues that needed to be considered so only a Self-Assessment Form has been submitted and is included as part of the report in the appendix.

5 Design

Each time a move is made, this leaves the game in a different state since the cards that are playable, the moves that are available, the current card in the stock, the current card in the waste pile and the overall status of the game changes. Therefore, the main idea is to have the game represented as a state and each time a move is made this means a transition from the current state to a new state. This naturally means that we can reason about solving the game in terms of an abstract search tree which represents the entire search space of the game. So, in other words, the state of the game is a tree node and a transition between states is a traversal from the current node to a new node. This is at the heart of the solver and so the implementation works around this idea.

The structure of the solver can be split into five distinct parts, each of which implements a specific function of the solver, and these will be described in detail in the following

subsections. Alongside this, a description of how the pyramid cards and stock cards are represented and stored internally has been provided.

Below is the overall structure of the project presented as a UML diagram. As mentioned, each of these components are presented in more detail in the following subsections.

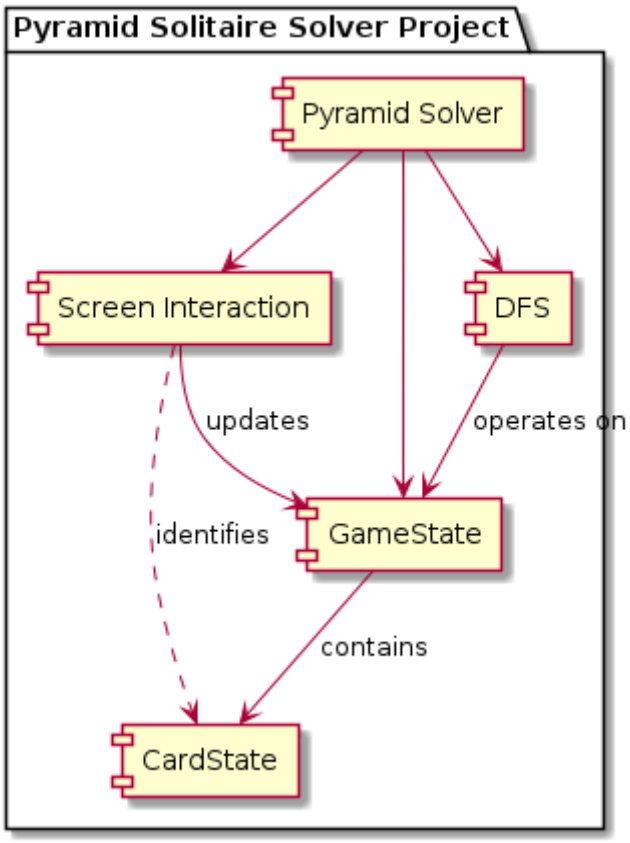


Figure 3: Project structure

5.1 Main game loop

The starting point of the solver is in *pyramid_solver.py* which contains the main loop of the game. First, the initial state of the game is read from a JSON file and then this loop is entered. During each iteration of the loop, the next best move to be made is calculated which results in a new node within the tree to be entered. This process continuously occurs resulting in the search tree being traversed, and the loop only terminates when a leaf node has been reached which can either be when a solution has been found or until there are no more moves that can be made.

Below is a UML diagram representing *pyramid_solver.py*.

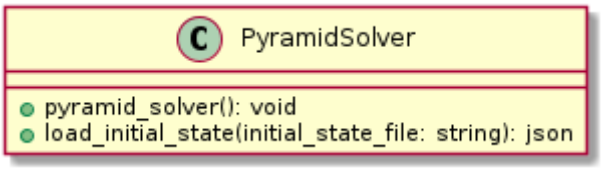


Figure 4: UML diagram for pyramid_solver.py

5.2 Game state

The state of the game is represented using the *GameState* class which is implemented in *game_state.py*. So, each game instance is an object of this class. This is where the three key components of the game are represented which are the pyramid, the stock and the waste pile. Additionally, a list of the moves already made and the moves that are available are maintained. All these fields will continue to change as moves are made and the game progresses, so it is necessary to update them while transitioning from one state to another.

Below is a UML diagram representing *game_state.py*.

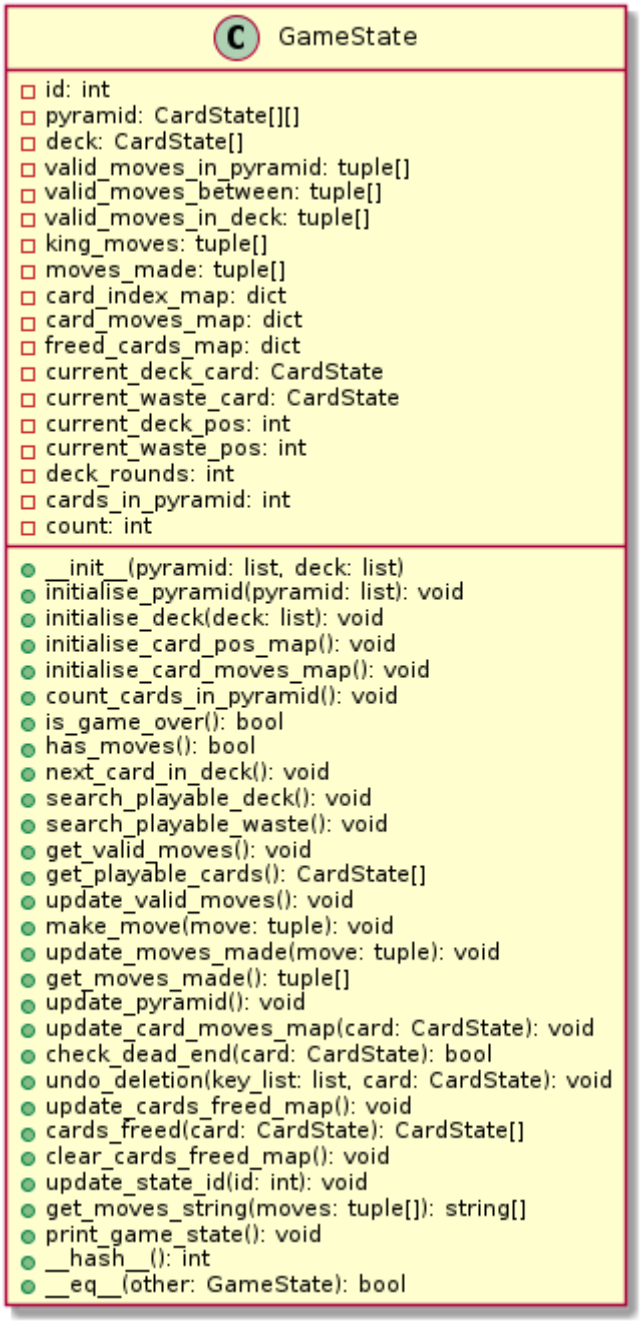


Figure 5: UML diagram for game_state.py

5.3 Card state

In addition to having a representation for the state of the game, it is important to consider the fact that each card within each game state can have its own state. So, each card is represented as a *CardState* object, and this class is implemented in *card_state.py*. There are three main fields for each card object which are the rank, the suit, and its playability status. The playability status represents whether the card can be played, has been played

or is blocked by another card. This is the field that will change as the game progresses and so it will have to be updated.

Below is a UML diagram representing *card_state.py*.

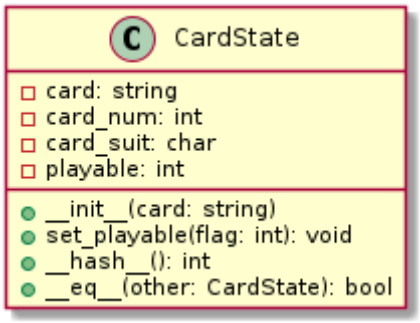


Figure 6: UML diagram for card_state.py

5.4 Tree traversal

Depth-First Search is the chosen tree traversal technique, and this is represented in the *DFS* class in *tree_traversal.py*. For each node, all the possible child nodes are identified and one of the nodes is chosen to be traversed to. This decision is made based on the heuristics that have been implemented. Additionally, backtracking may occur when there is a dead end that has been detected.

Below is a UML diagram representing *tree_traversal.py*.

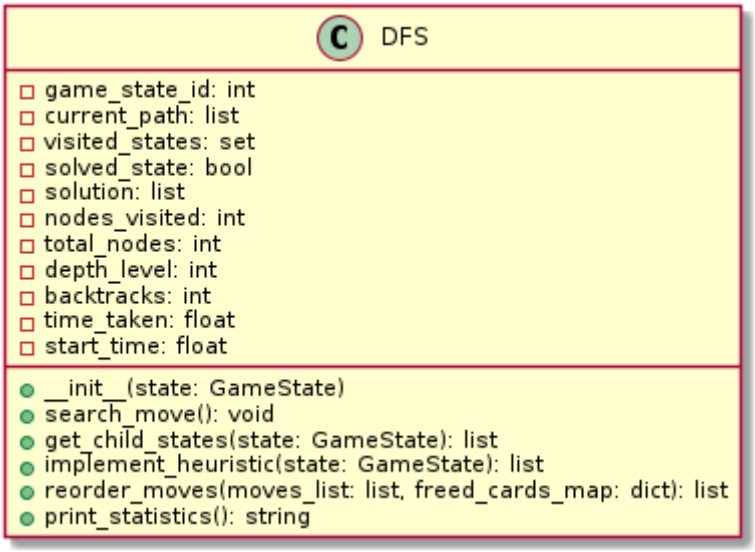


Figure 7: UML diagram for tree_traversal.py

5.5 Screen interaction

The automation part of the solver consists of two main functionalities and the implementation for these are included in *screen_interaction.py*. The first functionality is identifying the game window so that it can then proceed to scan the initial state of the game. This involves detecting and recognizing each of the cards in the pyramid and the stock. The second functionality is the execution of moves through mouse clicks on the screen. The first functionality occurs right before the main loop is entered because the initial state of the game in the JSON file needs to be written to, and the second functionality occurs after the loop has been exited which happens when a solution has been found.

Below is a UML diagram representing *screen_interaction.py*.

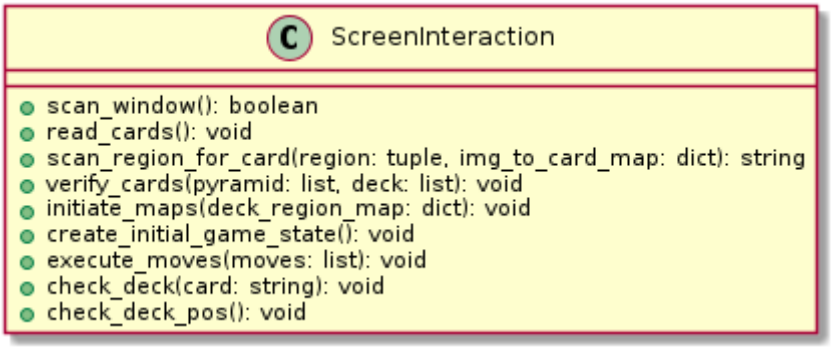


Figure 8: UML diagram for screen_interaction.py

5.6 Internal representation of the game

Another key aspect of the implementation is to have an internal representation of the game. This is necessary so that the solver doesn't make the moves on the screen as the optimal solution being searched for in the search tree. This would add a lot of overhead since GUI operations as well as the animations of making the moves happens at a much slower speed and so this would result in a performance bottleneck. Instead, the solver uses the initial state of the game and converts this to an internal representation which can programmatically be manipulated during the search.

5.6.1 Initial Game State

The internal representation of the game is stored in *game_state /initial_state.json* and this is only done for the initial state of the game since the later states are generated by making a copy of this initial state and then modify. The relevant parts of the game that need to be represented are the pyramid cards and the stock cards (the waste pile is irrelevant since it is initially empty and is treated as being part of the stock in the implementation). The pyramid is represented as a 2-dimensional array of cards and the stock is represented as a 1-dimensional array of cards. Each card is represented as a string which concatenates the rank and suit. For example, the Jack of Spades has the string value 11S (11 being the

numerical equivalent of Jack and S being the first letter in Spades). This pattern of string representation is followed for each card.

Below is an example of what an initial state would look in *initial_state.json*.

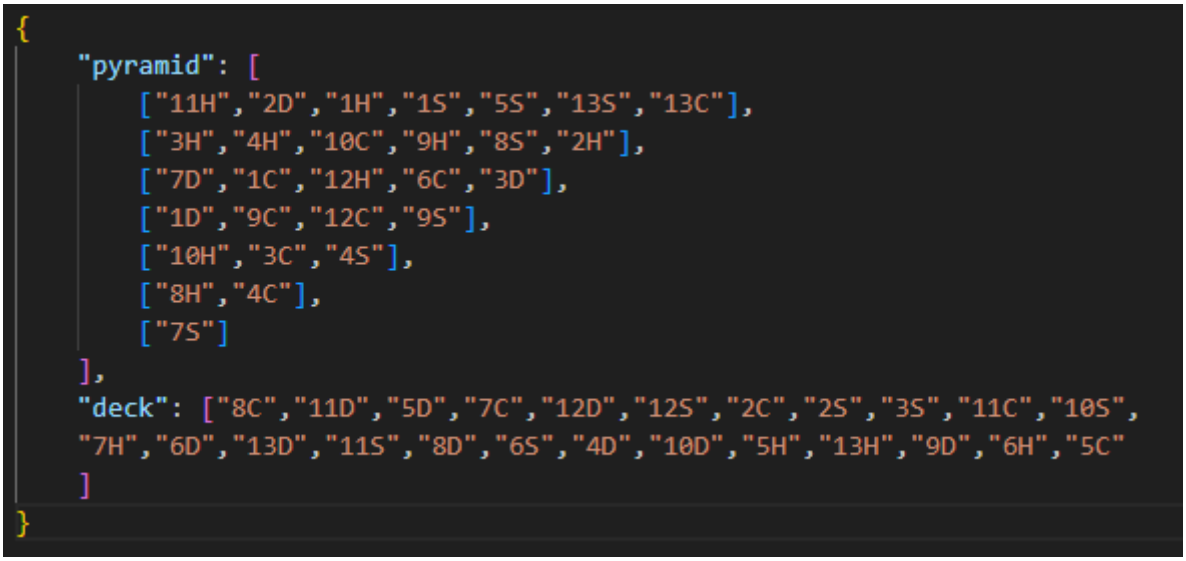


Figure 9: Example initial_state.json

5.6.2 Solution moves

The solution is stored as a list of tuples where each tuple contains a pair of card strings. When a move involves a king, only the second value in the tuple is empty. Below is an example of what a solution would look like.

```
[('7D', '6S'), ('3S', '10C'), ('13H',), ('13C',), ('8S', '5D'), ('9C', '4H'), ('2S', '11H'), ('8D', '5S'
), ('12C', '1C'), ('2H', '11D'), ('5C', '8H'), ('2C', '11C'), ('13S',), ('10D', '3H'), ('1H', '12H'), ('
1S', '12D'), ('7H', '6H'), ('12S', '1D')]
```

Figure 10: Example solution

6 Implementation

As mentioned in the introduction section, there are three main components to the solver. How each of these are implemented and integrated together to provide the necessary functionality will be explained in this section. Focus will be made particularly on key algorithms and data structures.

6.1 Card Recognition and Detection

The first step in calculating a solution to the game is identifying each card. The implementation of this functionality is included in *screen_interaction.py* which imports two key modules. The first is pygetwindow which makes it possible to manage application windows. The second is pyautogui which supports automating graphical user interface interactions.

Before any of the cards can be recognized, the game window needs to be identified and this is done in *scan_window()* using the pygetwindow module. A key point to note is that the window is moved to a specific location on the screen (0, 0) and is resized to specific aspect ratio (1280, 720). Why this is done will be discussed shortly.

Following this, *read_cards()* can then be called to detect each of the cards within the window. It performs template matching which is an object detection technique used to recognize a sub-image (template) in target image (where the template is being searched for). The template is slid across the target one pixel at a time in all directions and in each position a similarity measure is calculated for the overlapping region. This measures how well the template and target match in terms of their individual pixels. A match is found if the similarity score is higher than a specified threshold value. By default, this value is set to 1 which indicates a perfect match, but I chose a custom threshold value of 0.86 as this produced the best results based on my testing. In terms of my implementation, each card acts as a template, more specifically the rank and suit of each card which is stored in the *resources/cards/* subdirectory. Additionally, screenshots of the next stock button and undo all button are also included in *resources/buttons/* as they are an integral part in playing the game. The pyautogui module is used for this functionality.

Instead of having the template matching against the entire game window, I chose to mark specific regions within the game window where template matching is done. This is because using the entire screen window would be extremely inefficient since the template Is being slide one pixel at a time. Since each card is always located in a fixed position within the game window, the process of identifying each region was relatively straightforward. I manually measured the coordinates of each card in the pyramid, stock, and waste pile, along with the buttons and stored them in *resources/regions.json*. The data in this file is represented as a dictionary which maps each card and button to their respective coordinates. Because of how each card is contained within this region, this is sufficient for template matching. For this very reason, it was important to ensure that the game window has a fixed position and aspect ratio since the region coordinates depend on these two factors.

After establishing these requirements, the detection process can then occur, and it works by going through each card region for both the pyramid and stock. And for each region each of the card screenshots are used as templates. The card screenshot which gave the best similarity score is then chosen as the matching card for that specific region. Once each card has been detected, it then needs to be recognized. To aid this process, the

img_to_card_map dictionary, which is stored in *resources/image_to_card.json*, maps the image of each card to the string representation the card. The string equivalent of each card image is how the game is represented internally. Therefore, once the template with the best match has been identified, the string representation of the card can then be obtained by using it as a key to the dictionary, and this then allows for the pyramid and stock to be converted to the internal representation. This conversion is done in *create_initial_game_state()* which appropriately formats the pyramid and stock as described in section 5.6.1 and then writes it to *game_state/initial_state.json* representing the starting state of the game. Additionally, the card_to_region_map dictionary is created using the card strings and region data which will be relevant when executing the moves later.

Lastly, it is important to mention that a verification is displayed after all the cards have been identified and before converting the pyramid and stock into its internal representation of the game state. This is necessary because the accuracy of the recognition isn't high enough for the solver to reliably identify the correct cards every time.

6.2 Depth-First Search for Solution Generation

The initial state that has been created serves as the starting node in the search tree and so the search can now be initiated. The entire process of generating a solution can be summarized into five steps. First is instantiating the card and game states. From this, a list of playable moves can be generated and using heuristics, the most optimal move is given priority. The move is then made, and this entire process is repeated until a winning state has been reached using Depth-First Search. Below is a detailed explanation of how each of these steps has been implemented.

6.2.1 Instantiating Game State and Card States

The functionality for managing the state information for each card and the game is implemented in the *CardState* and *GameState* classes. Before the search can begin these objects need to be instantiated, which is done using the internal representation of the initial state obtained after the window has been scanned as explained in section 6.1. A *CardState* object is created for each of the cards in the pyramid and stock by going through the string representation of the cards. Then using these card objects, a *GameState* object is created where the array of card strings is converted into an array of card objects for both the pyramid and stock. Now that the game has been represented programmatically, the search can now begin. For context, this is done in *pyramid_solver.py* right before entering the main loop of the game.

6.2.2 Identifying Playable Moves

As mentioned in section 5.3, each card is represented as a *CardState* object and has a field to represent its playability status. Playable cards are marked by the playable variable having a value of 1. So, the first step in obtaining the list of valid moves is identifying which

cards are playable in the pyramid and storing them all in a list. This is implemented in *get_playable_cards()* which retries the list of playable cards from the pyramid. As for the playable cards in the stock and waste piles, the top card in each of these is always playable, so these cards are also taken into consideration.

Following this, in *get_valid_moves()*, every possible pair of cards is generated and the numerical ranks of the cards in each pair are added to see if they equate to 13. The card objects that do are stored in a list as a tuple which represents a move. Note that whenever a king is identified, it is stored as a tuple where the second element is empty. It is important to mention that moves have been categorized into different types and each type has its own list where the moves are stored. There are moves that can be made within the pyramid itself. There are moves that can be made between the stock and waste pile. There are moves that can be made between the pyramid and the stock or waste piles. There are also moves that only consist of kings. This is relevant for implementing the heuristics and will be explained in detail in section 6.3.4.

6.2.3 Making a Move

Making a move consists of changing the playability status from playable (value of 1) to played (value of 0). This is done in *make_move()* where it receives a move tuple. A key data structure that facilitates this is the *card_index_map* dictionary which maps the card objects to their index value. This specifies the position of each card in either the pyramid array or the stock array, and this is what is used to access each card that has been played as part of the move so that its playability status can be changed. Note that this dictionary is initiated in the constructor of *GameState* which goes through the pyramid and stock arrays, extracting the index value for each of the cards. The index values are stored as tuples which makes sense for the pyramid array since it is 2-dimensioanl, but since the stock is 1-dimensional, the first value in the tuple is aways -1 which marks the card as being in the stock/waste pile. Therefore, the first value in the index tuple is used when deciding whether the card played is in the pyramid or the stock so that the appropriate array is used in accessing the card.

After a move has been made, this causes cards in the upper rows to be unblocked if the cards played are in the pyramid. To capture this behavior, *update_pyramid()* is called each time a move has been made to update the pyramid. The logic here is that for every adjacent pair of cards in each pyramid row, if both the cards have a playability status of 0, the card in the row above that is being blocked by the pair below should have its playability status set to 1.

6.2.4 Depth-First Search Algorithm

The DFS class implements the tree traversal, and an object of this class is created at the beginning when the game state object is instantiated. The core functionality is facilitated using the *current_path* stack which stores the states to visit next and the *visited_states* set which stores the states that have already been visited. In the main game loop, during each iteration, a call is made to *search_move()* where the next state to visit is popped from the *current_path* stack. This basically symbolizes the traversal into a new node in

the search tree. After getting the new node representing the current state, the list of available moves for that state is generated as explained in section 6.2.2. However, the implementation isn't as simple as calling *get_valid_moves()*. First *update_valid_moves()* which is in GameState is called. This function then makes the call to *get_valid_moves()*, inside a loop, to retrieve the list of valid moves. It is then checked that among the different moves list representing different types of moves, at least one move is available in one of the lists. Then the loop can be terminated, and the available moves list is returned. Otherwise, a call is made to *next_card_in_deck()* which updates the current card in the stock and waste pile to the card that is next in the order and this is stored in separate variables for the top stock card and stop waste pile card. This process symbolizes pressing the next stock card button in the game. The entire process then repeats until a non-empty moves list is generated. This is arguably the most important part of the implementation of the solver since it has the largest consequence.

There are a few things to take into consideration here. Firstly, is that the playability status of the next card in the stock and waste pile needs to be checked to see that it is 1 and not 0. This is because there can be cards that have already been played in the stock and waste pile since the array for the stock isn't dynamically updated by removing them and are simply marked. Secondly, before entering the loop in *update_valid_moves()*, the available moves lists need to be emptied so that any moves that have been generated from the previous state isn't carried over to the current state. Thirdly, the rule of the game states that the stock can only be cycled a total of three times. So, if this limit is reached then an empty moves list is returned.

The more straightforward approach would be to implement the action of clicking the next stock card button as a move itself which would lead to a transition to a new state. Instead, I decided to only consider the action of clicking on a pair of cards as a valid move. The logic behind this is that progress in the game is only really made when a pair of cards is removed causing a reduction in the number of cards in the pyramid. And every time there are no more valid moves that can be played in the current state of the game, the purpose of the next stock card button is to find a card that would result in a playable move. So, the more efficient approach would be to have a loop which iteratively checks the next card in the stock and waste pile until an available move is found for the current state. For this very reason *update_valid_moves()* has been implemented. As a result of this approach, the size of the search tree is significantly reduced which I was able to conclude by testing both approaches. However, this approach leads to an issue in very specific situations when the only available move is between the stock and waste pile, but this move shouldn't be played because it results in a dead-end where one of the cards in the pyramid loses all its support cards and so can't be removed. To get around this issue I have heuristics in place which decided whether to add this move to the moves list or to ignore it and proceed to the next card in the stock and waste pile. The implementation of this will be explained in detail in section 6.3.

After the available moves list has been returned, a check is done to determine the next set of operations to perform based on three conditional branches. If all the cards in the

pyramid have been played in the current state, this is a winning state so the search ends and the *solved_state* variable is marked to be true so that the main loop of the game terminates. If not, then a check is done to see whether there are available moves and that the current state isn't in the *visited_states* set. This establishes the current state as a valid node in the tree and so *get_child_states()* is called which returns all the possible child nodes from this parent node. The parent node is then added to the set and the child nodes are added to the stack to be popped later. Each child node is generated from each move in the available moves list, where first the parent node representing the current state is copied. Then using the copied node, the move is made by calling *make_move()* as explained in section 6.2.3. This is repeated for all the available moves in the current state. Note that the ordering of the list of child nodes is determined based on the implemented heuristics which reorders the generated child nodes before pushing them to the stack. The implementation for this is in *implement_heuristic()* the details of while will be explained in section 6.3. If the first two condition branches aren't entered, this means that the current node is a dead-end so a backtrack needs to be done. This means that the current node is simply disregarded and the next time *search_move()* is called the next node that is popped is the node that has been backtracked to.

6.3 Heuristics

To improve efficiency and speed up the process of finding a solution, a set of heuristics have been employed so that the search doesn't entirely resort to an exhaustive search of the problem space. Since there are different types of moves lists, they are simply combined to form one list of available moves. The idea then is to reorder this combined list before making each of the moves so that the ordering of the child nodes will follow this order.

6.3.1 Prioritizing Kings

The simplest heuristic is to have the solver prioritize moves involving a king since these moves don't depend on other cards and it might also result in a card being unblocked. The implementation for this simply involves placing the list king moves at the start when concatenating the different lists. The means that in the next iteration, the nodes generated from this list will be popped out of the stack.

6.3.2 Prioritizing moves within the pyramid

Since the objective of the game is to clear out all the cards in the pyramid, another obvious heuristic would be to priorities moves that are made only involving pyramid cards. Then moves between a pyramid card and a stock/waste pile card are second in the priority list followed by moves made only between the stock and waste pile. The implementation of this follows the same principle as before where the list with the higher priority is higher up in the ordering. Note that the list of king moves will still have priority over these three lists.

6.3.3 Prioritizing moves that unblocks more cards

An ordering can not only be placed between the different lists but also within each list. This is because each move can differ in how many cards it unblocks, so cards that unblock more moves are given priority and this is what the ordering within each list is based on. Since only the cards in the pyramid can be unblocked, the list involving moves within the pyramid and the list involving cards in the pyramid and stock/waste pile are relevant here.

The relevant data structure for this heuristic is the cards_freed dictionary and the implementation is in *updated_cards_freed()*. This dictionary is updated for each state after calculating the available list of moves. It maps each move from that stat to a list of cards that will be unblocked as a result of making that move where this list is obtained by *calling cards_freed()* for each move.

Before all the moves lists are combined in *implement_heuristics()*, *reorder_moves()* is called for the two relevant lists and their moves are reordered based on the length of the list that each move maps to in the *cards_freed* dictionary.

6.3.4 Identifying moves that lead to a dead-end

Sometimes there are situations where the only move that can be played in the current state is a move consisting of a card in the stock and a card in the waste pile. This issue with this is the sometimes this could lead to a situation where one of the cards in the pyramid loses its support and so that card won't be playable leading to a lost state. Therefore, it is necessary to calculate ahead of time whether this type of move should be made or ignored.

This heuristic relies on the *card_moves_map* dictionary which maps each card in the pyramid to a list of all the card cards that can be played with it. For example, for the card 5S, the cards that can be played with it are 8S, 8C, 8D and 8H. This dictionary is initialized at the start of the game in *initialize_card_moves_map()* and each time a move is made this dictionary is updated in *update_card_moves_map()*. The process of updating the dictionary involves either removing the key-value pair entirely if the card that has been played is card in the dictionary or removing one of the cards in the value list if the card played isn't in the pyramid. One thing to take into consideration is that not all the cards that can be paired in theory with a card in the pyramid will be playable in practice. There are scenarios where one card in the potential pair might be blocked by the other card in the pair. So to take this into consideration, in *initialize_card_mvoes_map()*, each time a playable card is identified for a card in the pyramid, before adding it to the list it is passed to *validity_check()* ensure that this card can actually be played in practice. Then before adding a move to the list for cards between the stock and waste pile in *get_valid_moves()*, a conditional check is done where each card in the move is passed to *check_deadend()* and if one of cards is being depended on then this move is ignored.

There are different ways in which the presence of a dead-end can be identified. The first scenario is when there is only one card in the playable-with list of a conflicting card. The

second scenario is when there are multiple conflicting cards which share the same exact playable list and when this list has less cards than the number of key cards. For example, let's say the pyramid cards are 7S, 7C and 7D which all have 6S, 6D and 6H in their playable-with list. And if stock and waste pile move that's being checked is 7H and 6S, then this would result in there only being two rank 6 cards left but there are three rank 7 cards. So, one of the cards won't have any support. Both these scenarios are taken into consideration into consideration when implementing *check_dead_end()*.

6.4 Automated Move Execution

Once the solution has been found and the main game loop is terminated, the moves can then be executed and the implementation for this is included in *screen_interaction.py*. As described in section 5.6.2, the internal representation of the solution consists of a list of moves where each move is a tuple of card strings. Once again, the card and button regions are an essential part of the automation since the coordinates are utilized when specifying which area on the screen to click on. In *execute_moves()*, for each move from the moves list, each of the card strings are used to obtain the region of those cards using the *card_to_region_map* dictionary. Then a click action can be automatically initiated based on the coordinated which the pyautogui module provides support for. However, it is only this straightforward if the card is located within the pyramid. If it happens to be in the stock or the waste pile, then this needs to be identified which is done by checking If the card string is in the list of stock cards obtained from the initial state. If this is the case then *check_deck()* is called to search through the stock/waste pile for the relevant card. Each card can either be on top of the stock or on top of the waste pile, each of which is a separate region so both regions need to be checked. This check along with the clicking of the next stock card button is done repeatedly until the relevant card is found. The implementation of this isn't as straightforward as it sounds since there are a few things to consider. Each time a card from the stock/waste pile is played, it is discarded so the stock/waste pile list maintained in this algorithm needs to be dynamically updated which is done by removing the card each time it is found. The stock/waste pile cards are cyclic so indexes for the top stock card and top waste pile card need to be checked each time it is incremented. The *check_deck_pos()* function is responsible for this which sets the stock card to 0 each time the end of the stock is reached. The index for waste pile is always calculated by subtracting one from the index of the stock because it can be assumed that the waste card is always the previous card that was in the stock. Lastly, there are situations where there can either be no cards in the waste pile which happens at the start of the stock or no cards in the stock which happens at the end of the stock. For such cases there are conditional checks that are done to ensure that only the appropriate regions are checked each time in the loop. The waste pile is empty when the stock index is 0 and the stock is empty when it is set to -1 which is done if the index matches the length of the stock cards list.

7 Performance

As part of the primary objectives, the implementation of the tree traversal involved certain measurements being recorded. The intention of this was to highlight the performance of the solver when searching for the winning sequence of moves. There is a total of five performance metrics being recorded which includes the total number of nodes generated which corresponds to the total number of nodes in the tree, the number of nodes visited before reaching the winning state, the depth level of the winning state, the total number of backtracks performed and the time taken to find the solution.

The game has six difficulty levels: easy, medium, hard, expert, master, and grandmaster. I ran a series of tests where the objective was for the solver to correctly solve ten random instances for each of these difficulty levels. I recorded the number of games it took for the solver to win ten instances as well as the performances metrics for each of the instances for each of the difficulty levels. This section is dedicated to presenting the results from these test runs that demonstrate the performance of the solver as well as its correctness. Only the results of the ten successfully solved instances for each difficult level have been provide. The results are presented in the form of a table which includes the performance metrics recorded. Additionally, two charts are provided. The first plots the results regarding the tree traversal for each instance and the second plots the time taken for the tree traversal for each instance.

Unfortunately, due to the solver being unable to solve almost all master and grandmaster instances, the results for these two difficulty levels are included. This is a limitation of the solver and will be discussed further in section 8.3.

7.1 Easy Instances

Based on all the test runs, the solver was able to solve every easy instance. This means that it took ten tries to meet the threshold of ten successfully solved instances. On average, a total of 37 nodes were generated of which 26 of them were visited. The average depth-level of the winning node is 26 and 0 backtracks were made on average. The average time it took to find the winning sequence was 100ms.

Below is the performance metrics table for easy instance.

Game instances	Total nodes	Visited Nodes	Depth-level	Backtracks	Time taken (ms)
easy1	39	29	29	0	137.48
easy2	30	24	24	0	83.24
easy3	35	27	27	0	100.54
easy4	37	25	25	0	97.7
easy5	40	29	29	0	111.08
easy6	36	27	27	0	106.38
easy7	38	25	25	0	92.51
easy8	39	23	23	0	85.63
easy9	36	27	27	0	92.83
easy10	37	24	24	0	93.07
Average	37	26	26	0	100.046

Figure 11: Easy instances table

Below are two charts for easy instances.

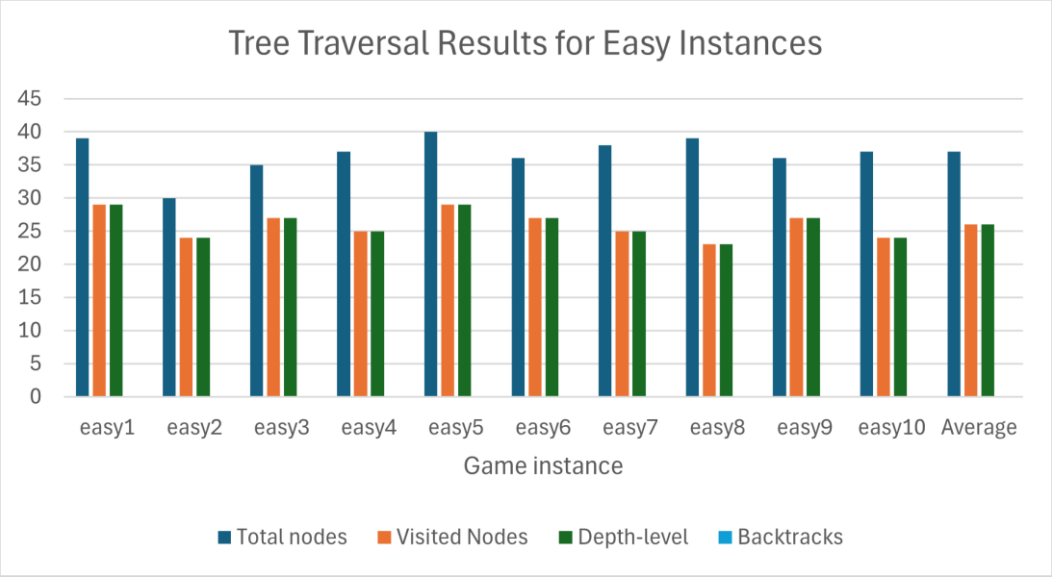


Figure 12: Tree traversal chart for easy instances

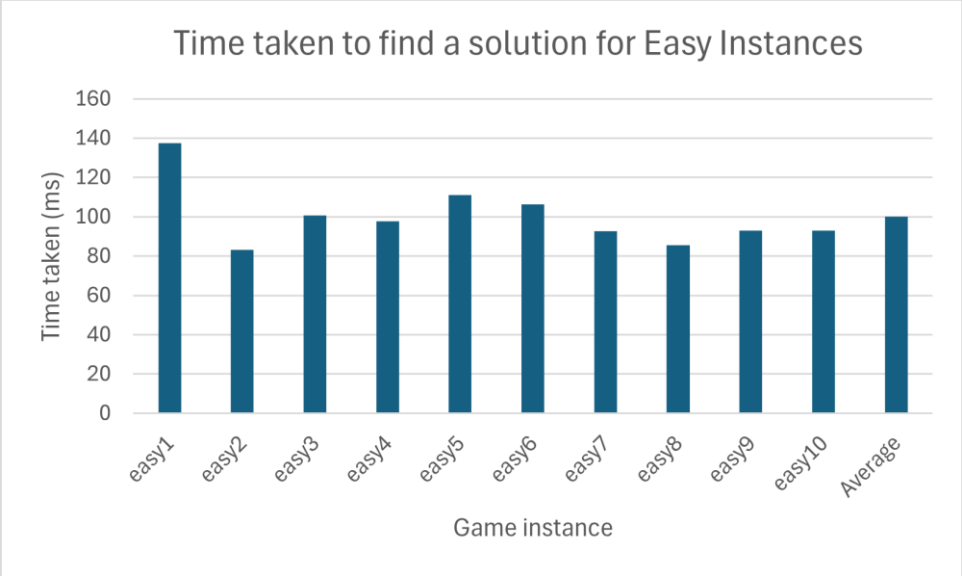


Figure 13: Time taken chart for easy instances

7.2 Medium Instances

The performance when it comes to medium instances seems to show similar results as it was able to pass all the test runs. This again means that it took ten tries to meet the threshold of ten successfully solved instances. On average, a total of nodes 43 node were generated of which 33 of them were visited. The average depth-level of the winning node is 32 and 1 backtrack was made on average. The average time it took to find the winning sequence of moves is 114ms.

However, based on testing done outside of these specific test runs, there were a few instances where the solver was unable to generate a winning sequence. Although these are uncommon since the solver can solve medium instances for the most part, this is still an occurrence that has been recorded.

Below is the performance metrics table for medium instance.

Game Instance	Total nodes	Visited Nodes	Depth-level	Backtracks	Time taken (ms)
medium1	35	25	25	0	86.25
medium2	35	27	27	0	91.76
medium3	39	29	29	1	104.68
medium4	40	26	26	0	96.36
medium5	44	32	31	1	107.08
medium6	37	28	28	0	97.68
medium7	87	77	73	4	267.78
medium8	41	27	27	0	95.31
medium9	36	27	27	0	100.86
medium10	35	27	27	0	95.24
Average	43	33	32	1	114.3

Figure 14: Medium instances table

Below are the two charts for medium instances.

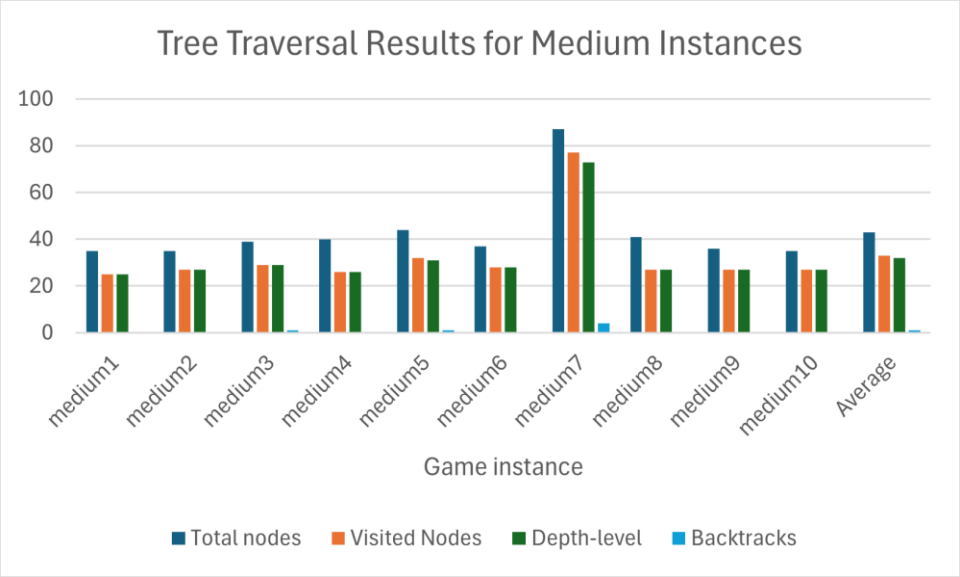


Figure 15: Tree traversal chart for medium instances

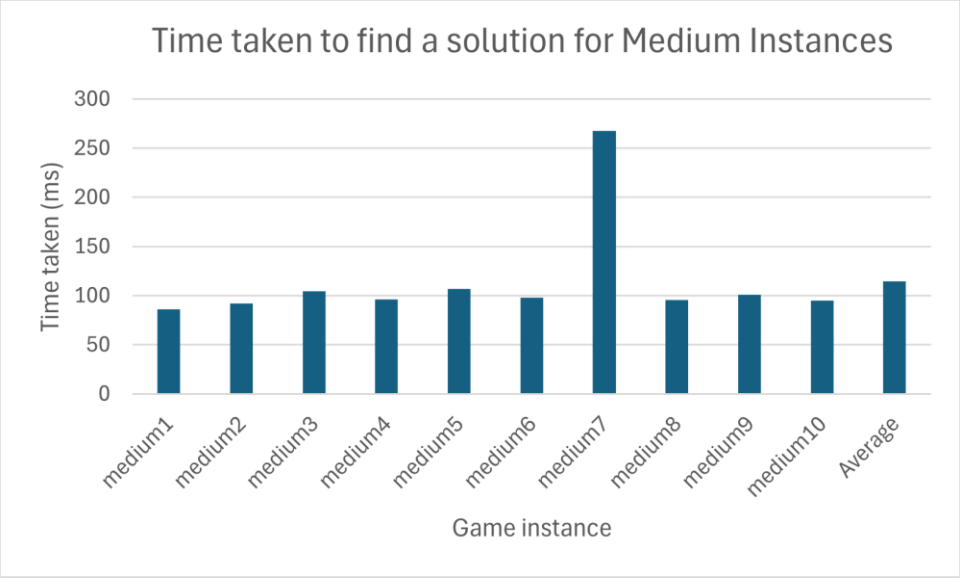


Figure 16: Time taken chart for medium instances

7.3 Hard Instances

It took a total of twelve tries to meet the threshold of ten successfully solved instances. This means that there were two instances where the solver was unable to find the solution. On average, a total of nodes 1275 node were generated of which 1292 of them were visited. The average depth-level of the winning node is 1041 and 225 backtrack was made on average. The average time it took to find the winning sequence of moves is 4346ms. It can be observed that there was some inconsistency in the performance since some of the instances took longer and required more nodes to be visited before a solution was found.

Below is the performance metrics table for hard instance.

Game Instance	Total nodes	Visited Nodes	Depth-level	Backtracks	Time taken (ms)
hard1	32	28	28	0	104.02
hard2	1086	1079	868	211	3744.44
hard3	50	26	26	0	95.17
hard4	33	26	26	0	115.67
hard5	36	288	28	0	97.63
hard6	2968	2956	2415	541	10095.96
hard7	36	29	29	0	115.41
hard8	65	45	41	4	158.31
hard9	8397	8397	6910	1487	28796.88
hard10	45	41	39	2	136.53
Average	1275	1292	1041	225	4346.002

Figure 17: Hard instances table

Below are the two charts for hard instances.

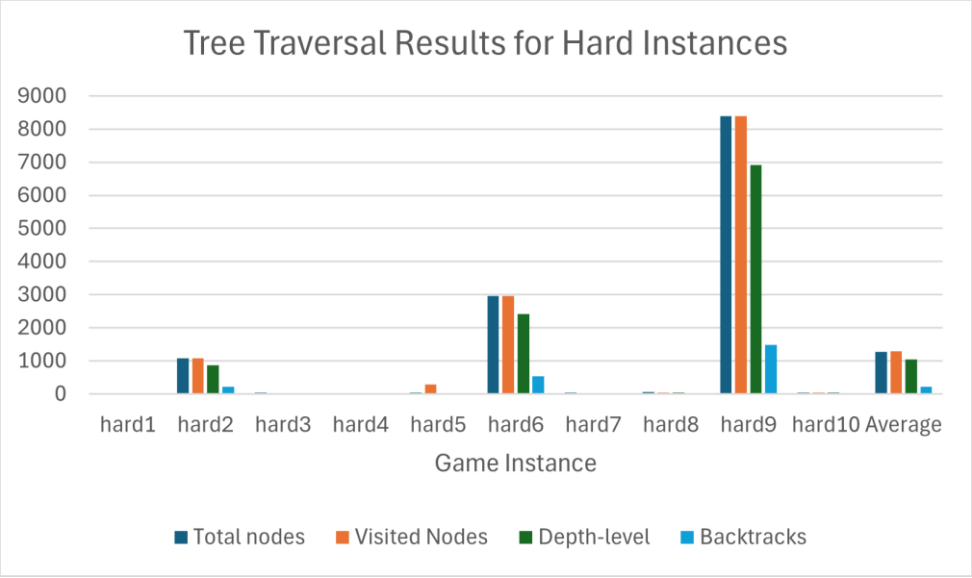


Figure 18: Tree traversal chart for hard instances

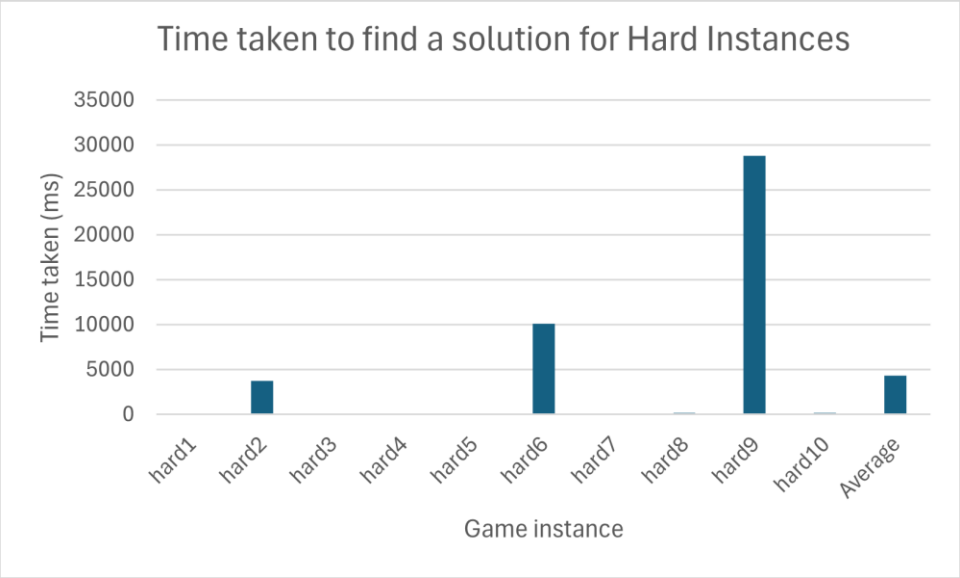


Figure 19: Time taken chart for hard instances

7.4 Expert Instances

The test runs for expert instances also took a total of twelve tries until it was able to meet the threshold of ten successfully solved instances. On average, a total of nodes 459 node were generated of which 450 of them were visited. The average depth-level of the winning node is 351 and 99 backtrack was made on average. The average time it took to find the winning sequence of moves is 1583. Again, it can be observed that there were inconsistencies in the performance where some of the instances required a lot more nodes to be visited and so it took a much longer time before a solution was found.

Below is the performance metrics table for expert instances.

Game Instance	Total nodes	Visited Nodes	Depth-level	Backtracks	Time taken (ms)
expert1	61	52	48	4	179.2
expert2	53	48	46	2	160.92
expert3	44	34	33	1	117.36
expert4	37	27	27	0	97.73
expert5	435	427	379	48	1533.44
expert6	3528	3514	2600	914	12318.48
expert7	75	61	57	4	232.76
expert8	269	268	250	18	936.07
expert9	39	28	28	0	101.15
expert10	51	42	41	1	162.73
Average	459	450	351	99	1583.984

Figure 20: Expert instances table

Below are the two charts for expert instances.

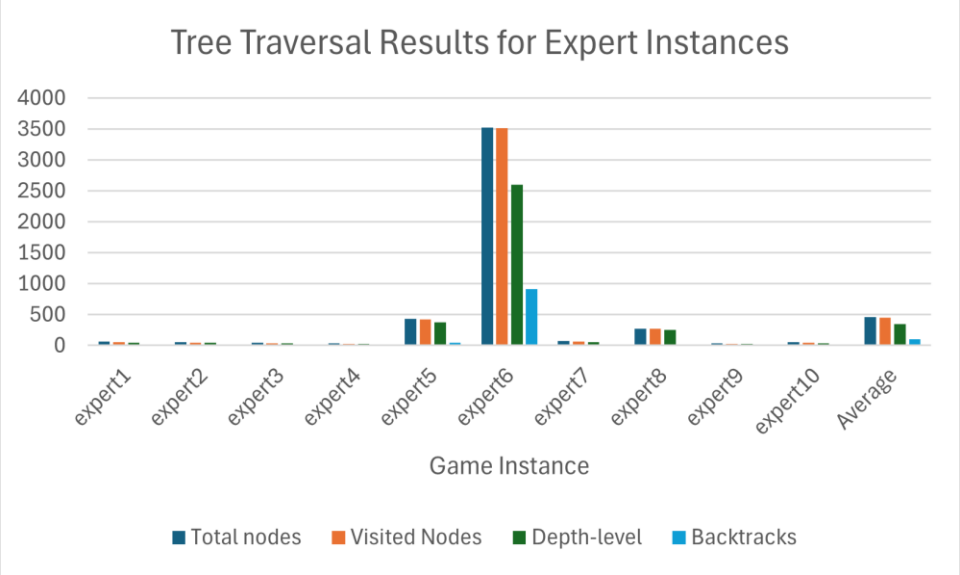


Figure 21: Tree traversal chart for expert instances

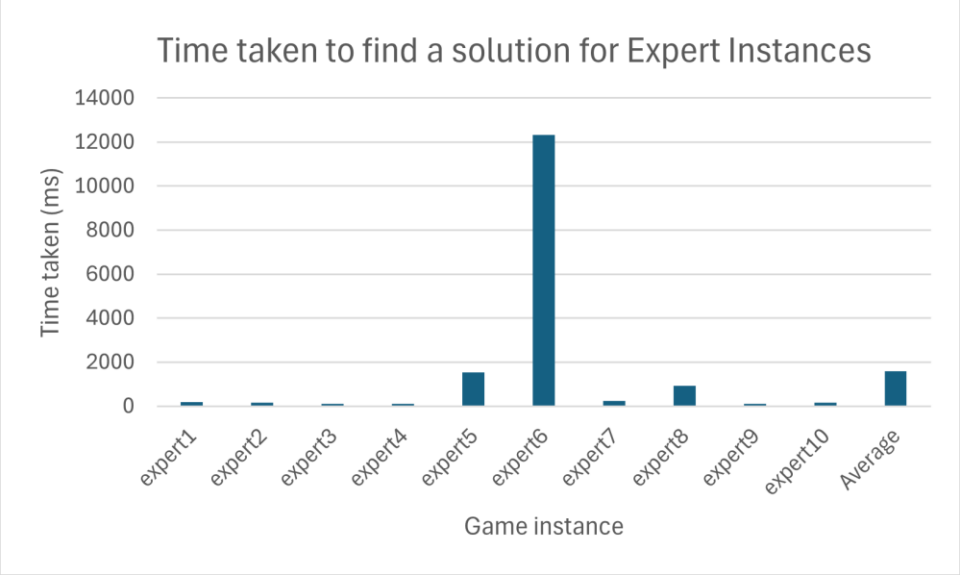


Figure 22: Time taken chart for expert instances

7.5 Comparison between difficulty levels

Below is a chart that compares the performance between the different difficulty levels. Easy and medium instances seem to have similar results for the most part. However, a huge jump can be observed for hard and expert instances which is to be expected since hard difficulties would require more traversal. Oddly, hard instances seem to perform worse than expert instances when it should be the other way around.

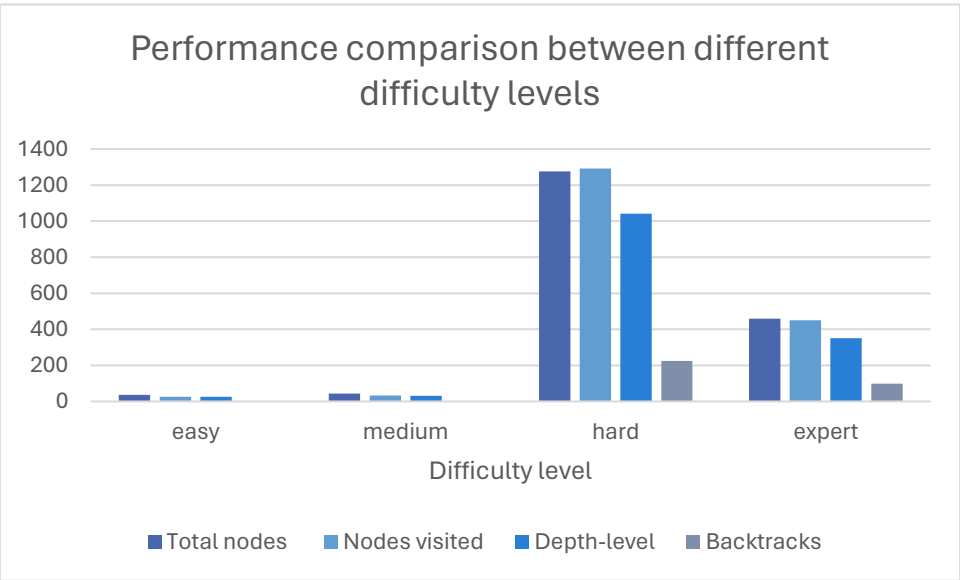


Figure 23: Comparison between different difficulty levels

8 Evaluation and Critical Appraisal

8.1 Objectives completed

My implementation of the pyramid solver has met most of the primary objectives from the original set of objectives. The solver uses Depth-First Search as the tree traversal techniques to search through a problem space for the winning sequence of moves. Using heuristics such as prioritizing moves from the pyramid, I was able to guide the search by prioritizing moves that are more likely to lead to a winning state. The solver can scan the application window of the game to obtain the initial state of the game, providing a verification window to allow the user to correct any potential mistakes made when recognizing each card. Once a solution has been found, it is able to execute the sequence of moves automatically by controlling the cursor to click cards and buttons. Lastly, there are certain measurements which are taken during the tree traversal which show the performance of the solver when searching for the solution. Out of the

8.2 Objectives failed to meet

Unfortunately, none of the secondary objectives have been met. There are no optimization techniques such as caching implemented to improve the performance of the solver. A Machine Learning model isn't integrated into the solver to enhance its capacity to solve problems. The solver isn't extended to be able to solve other variants of solitaire. Furthermore, one of the primary objectives isn't met, which is having a user-interface for the solver. I do have an implementation for this included in solver_window.py, but due to this being a last-minute implementation, it wasn't functioning as well as I would like it to. For this reason, I haven't included it as part of the final implementation of the solver, but it is still available to be run. The issue with it is that it sometimes disrupts the execution of moves by skipping some of the automated actions. As far as I'm aware, this unpredictability is an issue with multithreading. The user-interface displays the cards involved in the game instance and this is updated to highlight the cards that have been discarded as the moves are played. The main event loop for the user-interface is run in the main thread and the automated execution of moves is executed in a sperate background thread. These two threads being executed seem to result in some sort of resource contention but I'm not entirely sure again due to this being a last-minute implementation.

8.3 Limitations

Out of the objectives that I was able to meet, two of them have limitations in their functionalities. The first limitation is with the card recognition when getting the initial state of the game. Due to template matching not being a comprehensive enough approach, the accuracy of the recognition isn't high enough. The recognition of each card rank seems to be fairly accurate for the most part, but the issue is with recognizing each

card suit where it seems to perform poorly. Oddly, when using screenshots of cards from the same game instance, the accuracy of the recognition is accurate. But once a new instance is loaded, the same screenshots perform poorly. This seems to because due to slight differences in how the game is rendered for different instances. One potential solution to get around this issue would be to use OpenCV to perform template matching which is a more powerful and more flexible approach. This is because it provides ways in which images can be preprocessed and allows fine tuning through additional parameters. An alternative solution would be to train and fine-tune a pre-trained object detection model using TensorFlow, which was my initial plan. I have gone through this entire process which involved setting up the training environment, collecting training data, labelling training data, running the training process, and then evaluating the model's performance. However, with the available time, I couldn't train the model well enough for it to be imported into my project. The second limitation is with the solution generation process as my implementation can't solve master and grandmaster instances. There are also a few instances from the other difficulty levels that the solver can't solve, although rare. Further testing and debugging would have helped with identifying a solution, but I wasn't able to do this with the available time.

9 Conclusion

In conclusion, I was able to create an automated solver for pyramid that solves the game by interacting with the Windows Solitaire application. I achieved this by implementing functionalities which involved automating the card recognition process for each card in the game window, calculating the correct sequence of moves that lead to a winning state and automating mouse clicks to perform the moves in the game window. Despite limitations in the implementation, I have a baseline solver which performs reasonably on difficulty levels ranging from easy to expert. However, I would have liked to further improve the current implementation by addressing the limitations discussed in section 8.3. This means having a well-trained object detection model to reliably identify each card, having heuristics that perform a more comprehensive look-ahead mechanism and having a fully functional user-interface. Furthermore, I would have liked to extend the solver to work with the other variants of solitaire.

10 Appendix

10.1 Testing Summary

Below I have included the output generated when testing the solver with a random game instance:

- The output generated when recognizing each card in the game window where it displays the card string identified for each card region:

```
Scanning game window...
9H found in region (600, 153, 80, 30)
13H found in region (544, 181, 80, 30)
4S found in region (654, 181, 80, 30)
6S found in region (490, 212, 80, 30)
11S found in region (600, 212, 80, 30)
7S found in region (710, 212, 80, 30)
6H found in region (434, 240, 80, 30)
7D found in region (544, 240, 80, 30)
5S found in region (654, 240, 80, 30)
5D found in region (764, 240, 80, 30)
13S found in region (379, 270, 80, 30)
1S found in region (489, 270, 80, 30)
2S found in region (599, 270, 80, 30)
9S found in region (709, 270, 80, 30)
2D found in region (819, 270, 80, 30)
6S found in region (325, 300, 80, 30)
3S found in region (435, 300, 80, 30)
7H found in region (545, 300, 80, 30)
8S found in region (655, 300, 80, 30)
11D found in region (765, 300, 80, 30)
6S found in region (875, 300, 80, 30)
12S found in region (269, 330, 80, 30)
11H found in region (379, 330, 80, 30)
5S found in region (489, 330, 80, 30)
3S found in region (599, 330, 80, 30)
11S found in region (709, 330, 80, 30)
4S found in region (819, 330, 80, 30)
3H found in region (929, 330, 80, 30)
4S found in region (528, 464, 80, 103)
13S found in region (528, 464, 80, 103)
12S found in region (528, 464, 80, 103)
4H found in region (528, 464, 80, 103)
1H found in region (528, 464, 80, 103)
9S found in region (528, 464, 80, 103)
5H found in region (528, 464, 80, 103)
3D found in region (528, 464, 80, 103)
8S found in region (528, 464, 80, 103)
1S found in region (528, 464, 80, 103)
```

Figure 24: Terminal output from scanning each card

- The output generated when searching for the solution. While traversing the tree, for each node visited, the state of the game is displayed. This is the winning state of the game which is the final node.

```
Pyramid:
                    9H (0)
                13H (0) 4C (0)
             6S (0) 11C (0) 7C (0)
          6H (0) 7D (0) 5C (0) 5D (0)
       13S (0) 1S (0) 2C (0) 9S (0) 2D (0)
    8S (0) 3C (0) 7H (0) 8C (0) 11D (0) 6C (0)
 12S (0) 11H (0) 5S (0) 3S (0) 11S (0) 4D (0) 3H (0)

Deck:
4S (0) 13C (0) 12C (1) 4H (0) 1H (0) 9C (0) 5H (0) 3D (1) 8D (0) 1C (1) 9D (0) 10C (0) 12D (0) 1D (0) 2S
 (0) 7S (0) 13D (0) 10D (0) 2H (0) 10S (0) 8H (0) 6D (0) 12H (0) [10H (1)]

Cards in Pyramid: 0
Cards in deck: 1
Current card in deck pile: 4S
Cards in waste pile: 0
Current card in waste pile: 0
Deck round(s): 3
Is game over?  True
Are there any moves left?  False

Available moves:
[]

Moves made:
[('13C',), ('1H', '12S'), ('9C', '4D'), ('8D', '5S'), ('10C', '3H'), ('1D', '12D'), ('2S', '11H'), ('7S'
, '6C'), ('13D',), ('10D', '3S'), ('2H', '11S'), ('10S', '3C'), ('6D', '7H'), ('12H', '1S'), ('5H', '8C'
), ('11D', '2C'), ('4H', '9S'), ('8S', '5C'), ('13S',), ('6H', '7D'), ('2D', '11C'), ('8H', '5D'), ('6S'
, '7C'), ('13H',), ('9D', '4C'), ('4S', '9H')]

Solution found!

Statistics:
Total nodes generated: 32
Number of nodes visited: 27
Depth-level reached: 27
Number of backtracks: 0
Time taken to solve: 95.78085 ms
```

Figure 24: Terminal output of the winning state of the game in the search

- The output generated while executing the moves in the game window.

```
Executing moves...
Click on: 13C
Draw
Click on: 1H, 12S
Draw
Draw
Click on: 9C, 4D
Click on: 8D, 5S
Draw
Draw
Click on: 10C, 3H
Draw
Draw
Click on: 1D, 12D
Draw
Click on: 2S, 11H
Click on: 7S, 6C
Click on: 13D
Click on: 10D, 3S
Click on: 2H, 11S
Click on: 10S, 3C
Click on: 6D, 7H
Draw
Click on: 12H, 1S
Click on: 5H, 8C
Draw
Draw
Draw
Draw
Click on: 11D, 2C
Click on: 4H, 9S
Click on: 8S, 5C
Click on: 13S
Click on: 6H, 7D
Click on: 2D, 11C
Click on: 8H, 5D
Draw
Draw
Draw
Click on: 6S, 7C
Click on: 13H
Click on: 9D, 4C
Click on: 4S, 9H
Draw
```

Figure 25: Terminal output from the automated execution of moves

10.2 User Manual

Below are instructions on how to run the solver:

1. The Windows Solitaire application should be opened and shouldn't be minimized. After opening the application, select Pyramid and choose a difficulty level to start a game instance.
2. In the terminal, navigate to the project directory and install *pygetwindow* and *pyautogui* dependencies. This can be done by running *pip install -r requirements.txt*.
3. Now the solver can be started which is done by running *python pyramid_solver.py*.
4. Make sure not to move the game window while the solver is running.

10.3 Self-Assessment Form

UNIVERSITY OF ST ANDREWS
TEACHING AND RESEARCH ETHICS COMMITTEE (UTREC)
SCHOOL OF COMPUTER SCIENCE
PRELIMINARY ETHICS SELF-ASSESSMENT FORM

This Preliminary Ethics Self-Assessment Form is to be conducted by the researcher, and completed in conjunction with the Guidelines for Ethical Research Practice. All staff and students of the School of Computer Science must complete it prior to commencing research.

This Form will act as a formal record of your ethical considerations.

Tick one box

☐ **Staff Project**
☐ **Postgraduate Project**
☑ **Undergraduate Project**

Title of project

Playing Solitaire Automatically

Name of researcher(s)

Santhosh Basina

Name of supervisor (for student research)

Ian Gent

OVERALL ASSESSMENT (to be signed after questions, overleaf, have been completed)

Self audit has been conducted **YES** ☐ **NO** ☐

There are no ethical issues raised by this project

Signature Student or Researcher

Print Name

Santhosh Basina

Date

27/09/2023

Signature Lead Researcher or Supervisor

Print Name

Ian Gent

Date

27 Sep 2023

This form must be date stamped and held in the files of the Lead Researcher or Supervisor. If fieldwork is required, a copy must also be lodged with appropriate Risk Assessment forms. The School Ethics Committee will be responsible for monitoring assessments.

Computer Science Preliminary Ethics Self-Assessment Form

Research with secondary datasets

Please check UTREC guidance on secondary datasets (https://www.standrews.ac.uk/research/integrity-ethics/humans/ethical-guidance/secondarydata/ and https://www.st-andrews.ac.uk/research/integrityethics/humans/ethical-guidance/confidentiality-data-protection/). Based on the guidance, does your project need ethics approval?

YES ☐ **NO** ☑

** If your research involves secondary datasets, please list them with links in DOER.*

Research with human subjects

Does your research involve collecting personal data on human subjects?

YES ☐ **NO** ☑

If YES, full ethics review required

Does your research involve human subjects or have potential adverse consequences for human welfare and wellbeing?

YES ☐ **NO** ☑

If YES, full ethics review required
For example:

Will you be surveying, observing or interviewing human subjects?

Does your research have the potential to have a significant negative effect on people in the study area?

Potential physical or psychological harm, discomfort or stress

Are there any foreseeable risks to the researcher, or to any participants in this research?

YES ☐ **NO** ☑

If YES, full ethics review required
For example:

Is there any potential that there could be physical harm for anyone involved in the research? Is there any potential for psychological harm, discomfort or stress for anyone involved in the research?

Conflicts of interest

Do any conflicts of interest arise?

YES ☐ **NO** ☑

If YES, full ethics review required
For example:

Might research objectivity be compromised by sponsorship?

Might any issues of intellectual property or roles in research be raised?

Funding

Is your research funded externally?

YES ☐ **NO** ☑

If YES, does the funder appear on the 'currently automatically approved' list on the UTREC website?

YES ☐ **NO** ☐

If NO, you will need to submit a Funding Approval Application as per instructions on the UTREC website.

Research with animals

Does your research involve the use of living animals?

YES ☐ **NO** ☑

If YES, your proposal must be referred to the University's Animal Welfare and Ethics Committee (AWEC)

University Teaching and Research Ethics Committee (UTREC) pages http://www.st-andrews.ac.uk/utrec/