**Playing Pyramid Solitaire Automatically**

Abstract

Solitaire is a genre of card games where the aim is to arrange the cards in some systematic order by following a restrictive 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 a third-party solitaire application for which Windows Solitaire has been chosen.

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 NN,NNN 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.

Table of Contents

* Introduction
* Project Objectives
* Software engineering process
* Ethics
* Design
* Implementation
* Testing
* Evaluation and critical appraisal
* Conclusion
* Appendix

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 the shape of a pyramid with all the cards facing up. There are 7 rows of cards 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 where it is still facing up and is playable. 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. The deck can be cycled through a total of three times before the no more cards can be drawn. The aim of the game is to remove all the cards in the pyramid before this happens.

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 ahead of time. Players are required to make strategic decisions to calculate the correct sequence of moves which leads to all the pyramid cards being cleared out without reaching a dead-end where no more moves can be played. This very fact implies that every game in Pyramid can be solved programmatically using systematic techniques. Therefore, I wanted to see how I could employ these techniques along with automation to develop a solver that can solve every 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 which relies on computer vision techniques. The second is that it needs to be able to use this information to calculate the necessary moves that lead to a solution. 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 the tree traversal 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 and 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. Because of all these reasons and because of past experiences with Scrum, I have decided to follow this methodology.

Initially, before starting with the implementation, 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 each 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.

4 Ethics

There are no ethical issues that have arisen through this project that need to be considered. Therefore, only a Self-Assessment Form has been submitted and is included as part of the report.

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 deck pile, 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 the UI and a discussion on how the pyramid and represented and stored internally has been provided.

5.1 Main game loop

The center point of the solver is the main game loop. 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.

5.2 Game state

As mentioned above, the game can be represented as a state which is done by representing it as a GameState object. This is where the three key components of the game are represented which are the pyramid of cards, the deck pile, 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.

5.3 Card state

In addition to having a representation of the state of the game, it is important to consider the fact that each card within each game state can have its own state. This is represented as a CardState object. 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.

5.4 Tree traversal

Implementation of the tree traversal functionality uses Depth-First Search, and this is represented in the DFS class. 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 heuristics that have been implemented. Additionally, backtracking may occur when there is a dead end that has been detected.

5.5 Screen interaction

The automation part of the solver consists of two main functionalities. 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 of cards and the deck of cards. The second functionality is the execution of the 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.

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 when searching for the optimal solution in the search tree. This would add a lot of overhead since GUI operations as well as the animations of making the moves happen at a much slower speed and so this would result in a performance bottleneck.

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 it. The relevant parts of the game that needed to be represented are the pyramid cards and the stock cards. The pyramid is represented as a 2-dimensional array of cards and the deck is represented as a 1-dimensional array of cards. Each card is represented as a string which concatenates the rank and its suit. Below is an example of what the initial state would look.

5.6.2 Solution moves

The solution is stored as a list of tuples where each tuple contains a pair of card strings.

5.7 User Interface Design

The solver comes with a GUI which allows it to be operated in a user-friendly manner.

5.7.1 Solver Window

The solver window has two buttons: ‘Scan Window’ and ‘Solve Game’. Pressing on the scan button initiates the process of recording the initial state of the game. This is displayed in the solver window which has a section representing the pyramid and a section representing the stock. The display is dynamically updated each time a card is read as the window is being scanned. Pressing on the solve button initiates the automated execution of the moves and each time a pair of cards is cleared in the game window, it is highlighted in the solver window. The UI also has a text box which displays output messages.

5.7.2 Verification Window

After the game window has been scanned, a pop-up window displays the recognized cards in the pyramid and deck. This prompts the user to verify that the cards read are correct and allows for the values to be edited if they are incorrect,

6 Implementation

As mentioned in the introduction section, there are three main components to the solver along with the UI. 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 deck 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 and deck 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 deck. 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 deck to be converted to the internal representation. This conversion is done in create\_initial\_game\_state() which appropriately formats the pyramid and deck as described in section 5.6 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 the verification window described in section 5.7 is relevant here. This pop-up window is displayed after all the cards have been identified and before converting the pyramid and deck 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 Card and Game State

The functionality for managing the state information for each card and the game in general 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. 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 deck. Now that the game has been represented programmatically, the search can now begin. For context, this is done in solver\_window.py right before entering the main loop of the game in solve(). The details of this part of the implementation will be explained in detail in section 6.4.

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

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, the moves can then be executed and the implementation for this is included in screen\_interaction.py. As described in section 5.6, 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 deck then, this needs to be identified which is done by checking If the card string is in the list of deck cards obtained from the initial state. If this is the case then check\_deck() is called to search through the deck for the relevant card. Each card can either be on top of the deck pile or on top of the deck pile, each of which is a separate region so both regions need to be checked. This check along with the clicking of the next deck 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 is played, it is entirely discarded so the deck list maintained in this algorithm needs to be dynamically updated which is done by removing the card each time it is found. The deck and waste cards are cyclic so indexes for the top deck card and top waste card need to be checked each time it is incremented. The check\_deck\_pos() function is responsible for this which sets the deck card to 0 each time the end of the deck is reached. The index for waste pile is always calculated by subtracting one from the index of the deck pile because it can be assumed that the waste card is always the previous card that was in the deck. Lastly, there are situations where there can either be no cards in the waste pile which happens at the start of the deck or no cards in the deck pile which happens at the end of the deck. 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 deck position is 0 and the empty deck pile is empty when it is set to -1 which is done if the index matches the length of the deck cards list.

6.5 Interaction with the User Interface

The user interface of the solver application is implemented in solver\_window.py. The interaction with the user interface mainly revolves around the ‘Scan Window’ and ‘Solve Game’ buttons which provide the two main functionalities of the application from the user’s perspective.

The ‘Scan Window’ button calls scan() which initiates the process of identifying the game window and scanning each card in the window which is the functionality explained in section 6.1. The intention is to display the card one at a time on the solver window each time it is recognized in the game window. To support this, scan\_window() and read\_cards() are executed in a separate thread from the main GUI thread. This is necessary because the scan process takes a considerable amount of time, and the application needs to remain responsive during this period. A callback function, scan\_card\_callback(), is passed to read\_cards() which is called each time a card is recognized. This triggers the UI to update its display which is precisely why multithreading is necessary. Without it, the UI would not respond to the callback and so the UI wouldn’t be updated with the newly detected card. In read\_cards() the card string along with an id for that card string is passed in the call back. The id value is a simple integer value incremented each time a card is recognized. Once these values are received, scan\_card\_callback() passes returned values to update\_game\_display() which is responsible for updating the display. The solver window has a placeholder for each card and the appropriate placeholder to update with the card string is identified using the pyraid\_cards dictionary where the id value for the card is used as the key. Additionally, the update\_queue is used to safely communicate between the background thread and the main thread. Each time a callback is made, the call to be made to update\_game\_display() isn’t immediately called but is instead added to the queue. Then in check\_queue(), the queue is constantly monitored and each time a call is added, this is retrieved and the call to update\_game\_display is made. Lastly, on\_scan\_complete is another function included in the background thread and is executed after read\_cards() has completed its execution. This is necessary because once the verification process is completed at the end of read\_cards(), the corrected card string needs to be displayed in the UI of the solver which often times won’t match the cards initially displayed.

The ’Solve Game’ button calls solve() which initiates the automated execution of moves which is the functionality explained in section 6.3. In this function, the solution moves needs to be generated first. This is done by first creating a GameState object with the initial game state followed by creating a DFS object with the GameState object. Then the main game loop is execute which repeatedly calls DFS.search\_move() to get the next best move. Once a solution is found, the game loop terminates, and the solution is retrieved. Following this, execute\_moves() is executed in a separate thread. The rest of the implementation follows the same exact principles established in scan(). The only difference is that the UI is updated by highlighting the card that has been played.

7 Performance Testing

As part of one 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. These performances metrics included, 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. Therefore, this section is dedicated to presenting the result of these test runs that demonstrate the performance of the solver as well as its correctness. The measurements of the ten winning instances for each difficult level have been provided in the form of a table and a few charts.

Unfortunately, due to the solver being unable to solve almost all master and grandmaster instances, these instances aren’t included. This is a limitation 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 which 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 a table which summaries the results for each of the instances.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **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 |

Below are two charts that are based on the measurements from the table above. The first plots the results regarding the tree traversal for each instance and the second plots the time taken for the tree traversal.

A graph of multiple colored bars

Description automatically generated with medium confidence

A graph of a solution

Description automatically generated

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, so yet again 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 are a few instances where the solver was unable to generate a winning sequence. Although these are uncommon since the solver is able to solve medium instances for the most part, this is still an occurrence that has been recorded.

Below is a table which summaries all the measurements made from the tests for each of the instances.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **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 |

Below are two charts that are based on the measurements from the table above. The first plots the results regarding the tree traversal for each instance and the second plots the time taken for the tree traversal.

A graph of multiple columns

Description automatically generated with medium confidence

A graph of a solution

Description automatically generated

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. However, 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 a table which summaries all the measurements made from the tests for each of the instances.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **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 |

Below are two charts that are based on the measurements from the table above. The first plots the results regarding the tree traversal for each instance and the second plots the time taken for the tree traversal.

A graph with different colored bars

Description automatically generated

A graph with blue and white bars

Description automatically generated

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. However, 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 a table which summaries all the measurements made from the tests for each of the 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 |

Below are two charts that are based on the measurements from the table above. The first plots the results regarding the tree traversal for each instance and the second plots the time taken for the tree traversal.

A graph of a graph with text

Description automatically generated with medium confidence

A graph with text on it

Description automatically generated

7.5 Comparison between 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.