**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 such as Klondike, Spider, Freeecell and Pyramid, where each game has its own set of rules.

Pyramid has been chosen as the solitaire variant of choice and the aim is to develop a solver which solves the game automatically by interacting with a third-party solitaire application. The third-party application of choice for this project is Windows Solitaire.

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

Winning in Pyramid is not just about luck but is especially about having the necessary skills and strategies when solving it. Players are required to make strategic decisions to calculate the correct sequence of moves which lead to a winning state. This very fact implies that in theory, 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 to function. The first is that it needs to have the ability to scan the game window to detect and recognize each card through image processing. The second is that it needs to be able to use this information to calculate the necessary moves that lead to a solution by searching through the entire search space of the game. This relies on tree traversal techniques along with heuristics to guide the search, effectively pruning the search space to improve efficiency. Lastly, once a solution has been found, the moves specified in the solution need 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 main 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 the 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 in a particular part of the project can be carried onto following parts of the project. 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 amount of work required to implement 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 intended on implementing during that sprint. During each sprint the actual implementation of the features occurred 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. 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 a node to the one of its child nodes. This is at the heart of the solver and so the implementation works around this idea.

The structure of the implementation consists of five different components, each of which is represented in their respective program files. Each of these components implements a specific aspect of the solver which will be described in detail in the following subsections.

5.1 Main game loop

The starting point of the solver is in pyramid\_solver.py which contains the main game loop. The initial state of the game is read from a JSON file and then sent to tree\_traversal.py which is responsible for performing the tree traversal. During each iteration of the loop, a call is made to the relevant functions in tree\_traversal.py and this 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 in game\_state.py where it is treated as an object of the GameState class. 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 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.

5.4 Tree traversal

Implementation of the tree traversal functionality uses Depth-First Search, and this is represented in the DFS class in tree\_traveral.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 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 is implemented in screen\_interaction.py where there are 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 a solution has been found so that this solution can be passed to the relevant functions in screen\_interaction.py to perform the automated execution of the moves.

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 performance bottleneck.

The internal representation of the game is stored in the initial\_state.json file in the game\_state subdirectory 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 only relevant parts of the game that needed to be represented are the pyramid of cards and the deck of 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.

6 Implementation

For each node, all its possible child nodes are identified. This is done by making a copy of the current state and then making a move from the list of available moves of the current state. This is repeated for every move that is possible from the current state and so the number of child nodes created corresponds to the number of moves available from the current state.