Solving Peg Solitaire with Artificial Intelligence

Nathan Sylvia
Computer Science Department
University of Massachusetts Lowell
Lowell, Massachusetts
Nathan_Sylvia@student.uml.edu

Evan Mitchell
Computer Science Department
University of Massachusetts Lowell
Lowell, Massachusetts
Evan_Mitchell@student.uml.edu

Junghwan Kim
Computer Science Department
University of Massachusetts Lowell
Lowell, Massachusetts
Junghwan_Kim@student.uml.edu

***Abstract*—This document describes a semester-long study of solving peg solitaire puzzles with artificial intelligence. The project began with a literature review to gain familiarity with existing advancements in the field. The literature review revealed few existing similar studies, leaving an excellent opportunity for new approaches. In this project, four approaches to solving peg solitaire were evaluated and compared for viability on different types of puzzles. Three of the approaches utilized search algorithms, breadth-first, depth-first, and iterative deepening search. The fourth approach used a Q-learning algorithm to map actions and rewards for different puzzle states. A graphical interface with animated representations of the puzzles and a flexible program structure were created alongside the algorithms. These serve to facilitate comparisons of intelligent agents and allow for efficient, organized expansion of the project in the future. The performed literature review, implementation methodology, and study results are discussed in detail in this report.**

Keywords—Peg Solitaire, Artificial Intelligence, Breadth-First Search, Depth-First Search, Iterative Deepening Search, Q-Learning

I. Introduction

Peg solitaire is a problem-solving strategy game that has been known to exist for several hundred years and is commonly found on the tables of restaurants to entertain waiting customers. The premise of the game is simple, but solving it is quite complicated, making it an excellent subject for artificial intelligence studies. A game of peg solitaire consists of a game board with a set of holes and a set of pegs that fit into the holes. There are several variations of peg solitaire game board shapes, including different cross-shaped boards in Europe, and triangular boards in the United States. Despite their different shapes, the rules and solutions to the games are quite similar. The focus of this study is the American triangular peg solitaire game, pictured below.

Figure 1: A peg solitaire game board. Source: Joenord.com

The American peg solitaire board consists of a triangular arrangement of an arbitrary number of holes, N, and an arrangement of no more than N-1 pegs in those holes. At least one empty hole is required at the start or the game will have no possible moves. To make a move, a peg is taken from its hole and moved over the top of an adjacent peg to an open hole. The peg that has been jumped in the move is removed from the game board. This is the only type of valid move in the game and is repeated continuously until there are no more possible moves to be made. The goal is to finish with one peg. If the final move results in one remaining peg, the gameplay is considered a victory. If there is more than one remaining peg, the puzzle has not been solved, with each additional peg remaining at the end of play considered to be a worse result.

There are several features of peg solitaire that make it intriguing for an artificial intelligence investigation. The most important of these features is the great complexity in problem solving despite having a simple layout and only one type of available move. Another feature is how easily the game board can be adapted to create new puzzles. For example, for a board with N holes, any number or arrangement of pegs can be used so long as no more than N-1 pegs are placed on the board. Not every peg arrangement has a solution, but this is interesting in itself, as the intelligent agents can provide information on which games have solutions and why. Another complicating aspect of this problem is symmetry. A triangular game board has symmetry on three separate planes, creating redundant states. This can be helpful if utilized correctly but can also cause problems for an agent. Successfully solving peg solitaire puzzles with artificial intelligence can provide valuable comparisons between algorithms and also serve as a teaching tool, helping to instruct newcomers in how to beat the game.

The main goal of this project was to design a model that can be adapted to many different peg solitaire games and to develop a set of intelligent agents to solve the puzzles, providing comparisons of effectiveness. A flexible model was constructed, and three search-based agents, depth-first, breadth-first, and iterative deepening search were constructed to run on the model, allowing comparison between each approach in puzzle solving. A fourth agent, a Q-learning agent was constructed to demonstrate the tradeoffs of learning, with less required modeling, but also less certainty of reaching a solution under certain conditions. Along with the model and agent, a graphical user interface was constructed to serve as a research and teaching tool. The graphical interface allows selection of each different agent and a choice of custom and preset puzzles. From the interface, agent performance can be compared, including

elapsed time, states expanded, the set of moves returned as the solution, and an animated visual representation of the gameplay. This tool was utilized for analysis and debugging of agents throughout the project and can serve as a valuable teaching tool on multiple levels, both to artificial intelligence students and to people who simply want to learn to play peg solitaire.

II. Literature Review

Before beginning this project, a literature review was performed to become familiar with current and past investigations into solving peg solitaire with artificial intelligence. There were not many available published papers, but one paper was found that provided a quality in-depth analysis. This paper was "Modelling and Solving English Peg Solitaire", from the University of York [1]. This paper stated that the problem of solving peg solitaire is NP-complete, and discussed many important aspects of the game, some which make it harder to solve and some which make it easier.

A helpful aspect of peg solitaire that is explained in the paper is that the number of moves to a goal state is always known. Each move removes one peg from the board and the goal state has only one peg remaining on the board. This means that the number of moves to the board can be directly calculated and is always equal to one move less than the number of pegs. This is helpful for predicting effectiveness of search agents as well as checking on their progress. The most important side effect of the set number of moves is that every path to the goal state is optimal as they will all be the same length. A complicating feature discussed in the paper is the high level of symmetry in the game board. For example, the three-sided triangular board used for this project has three sides that are symmetrical. The board could be rotated 60 degrees either direction to get other equivalent states. This adds redundancy to the problem of solving the puzzle. For a human, this is a good thing. The symmetry is easily recognized, and the puzzle can simply be rotated to change the perspective. For an intelligent agent, this symmetry can be very difficult to recognize depending on the state space representation and can waste time and memory. If this symmetry can be exploited, however, efficiency can be improved.

The paper "Modelling and Solving English Peg Solitaire" focused on using complex planning algorithms to solve the puzzle. Single planning algorithms alone were found not to be very effective, but combinations of them were much more successful. The paper lamented, however, that modelling and solving such a game was very difficult due to its exponentially growing complexity, and limitations in memory and processing power when the study was performed in 2003, using 256 MB of RAM and a Pentium III processor [1]. This provided an excellent opportunity to expand on the modelling of such a game and developing agents to solve it.

In the time since 2003, great advancements have been made in computing technology, allowing much more memory use and processing power to be dedicated to solving peg solitaire puzzles. Additionally, that study used the English cross-shaped game board, which has more pegs, and a more complicated structure than the American style game board that is the focus of this project. The combination of more memory and a simpler game board led to the belief that this type of peg solitaire can be completely modeled and subsequently solved by search-based agents, something that was not attempted in the English Peg Solitaire study, and the main focus of this project. With the puzzle seemingly highly conducive to search based agents, the decision was made to make multiple search-based agents for comparison. A Q-learning agent was also decided on for comparison and to attempt to limit modeling needs for one approach such as has been required in past attempts at solving the game.

Upon the decision to use modeling and search to solve the game, the focus of the literature review shifted to attempts at solving such puzzles with search methods to see if it had been achieved. One report was found, titled "Trying to Solve Peg Solitaire by a Simple, Depth-First Search" [2]. This paper described success in using depth-first search to solve the problem, but intentionally left out important implementation details, appearing to be intended as a directed study more than a full report. Still, some useful information was gained. Success was described on some types of game boards and failures in others along with time and memory usage statistics. Once again, this paper differed in board type and state space representation from the objective of this project, focusing on specific pegs and their movements versus changes in the full game board layout.

To gain additional insight into the triangular version of peg solitaire, another study, on the theory of solving triangular peg solitaire, was examined. This study, [3] broke the game down into important algorithms and logic problems that helped in the creation of the agents. While it was not focused on using artificial intelligence to solve such a game, the provided background information was a useful supplement to the other studied literature.

The literature review provided much needed information on the dynamics of peg solitaire but left much room for expansion as well. The American triangle board was not the focus of the studied papers and the desired modeling of this project is different as well. The only search agent found to be used in the literature review was depth-first, with no attempts at breadth-first search or iterative deepening. Q-learning was not found to be used anywhere in past studies of the game either. All these differences, along with the aging nature of the reviewed studies and subsequent developments provided comfortable room for development of new approaches.

III. Methodology

The first step in solving peg solitaire with Artificial Intelligence was selecting the programming language platform. Two languages were considered, Python, and C#, both due to their easy syntactics, data management, and abundance of polymorphic data structures. Of these two languages, C# was selected for the project due to the extensive debugging and analysis features of Visual Studio which would help to step through the algorithms to examine their behavior and provide extremely useful information on run time and data usage. These features are part of the Diagnostic Toolbox, pictured below.

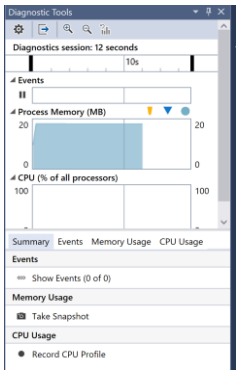


Figure 2: Visual Studio Diagnostic Tools

Other features of C# well-suited for this project are its structure of class objects and libraries that keep large coding projects well organized, and its extensive tools for graphical user interfaces, a requirement for this project. A class library was implemented for the project, with each piece of functionality, such as game states, game types, and agents having their own separate files and class structures.

The next step was representing the state space, which is crucial for this project. Game states and all necessary information to solve them are contained in an object named GameState. A GameState object contains an integer variable indicating the remaining number of pegs and a two-dimensional list of Booleans to represent peg and hole locations on the game board. These member variables provide all necessary information for solving a peg-solitaire puzzle. The remaining pegs variable is used for goal checking. At the goal state, one peg remains. If there is any other number of pegs, a goal has not been reached. The two-dimensional list of Booleans is used as a grid to indicate the locations of all pegs and holes on the board. A true value indicates a peg and a false value indicates a hole. This representation allows great flexibility in game board configurations, as lists can be resized, and Boolean values can be changed, allowing representation of nearly any peg-solitaire board with the same GameState object.

In addition to the state representing variables, the GameState object has a full set of functions that can be used by agents to solve the puzzle. Member functions of the GameState object return whether or not it is a goal state, the number of remaining pegs, the valid moves from the state, the next state after making a certain move, and all possible next states from the current state. The return values of all these functions are derived from the two member variables, remaining pegs, and the grid of peg locations. Moves returned by these functions are represented as a list of three coordinate pairs, representing the peg that is to be moved, the peg that is jumped, and the new location of the peg after it is moved.

After the state space model had been constructed, the next step was providing agents to solve puzzles given the state representations. An abstract Agent class was created, from which all agents are derived. This Agent class has two member variables, a GameState object, and an integer indicating expanded states. The GameState object is passed into the constructor for an agent and represents the game that is to be solved. The expanded states variable indicates the number of states that were explored by a search agent and is used for comparisons of data usage between agents.

The Agent class contains two functions, an accessor function for the expanded states, and a virtual function named solve. The solve function is overridden by each agent and using the information provided by the GameState runs the algorithm specified by the agent to attempt to find a solution. If the algorithm is successful in finding a solution, the move sequence found to reach the goal state is returned as a list. If every possible state is explored and no solution is found, an exception is thrown to be handled by the calling function. The solve function takes two parameters, both to implement a user-specified timeout. If a timeout is set and reached, a timeout exception is thrown, indicating that a solution was not found in the allowed time interval. Once again, this exception is to be handled by the calling function to continue proper operation.

With the abstract Agent class complete, the next step was to implement the different search agents to solve the puzzles. Three new class objects were added to the class library, each derived from the abstract Agent class to allow inheritance and interchangeability. Each agent has an overridden version of the solve function. The depth-first search agent uses a stack-based algorithm to perform a depth-first graph search on the states of the game. The breadth-first search agent uses a queue-based algorithm to perform a breadth-first graph search. The iterative deepening agent uses a modified version of the depth-first solve function, still using a stack by incorporating a loop and an incrementing depth to utilized iterative deepening.

With the search agents complete, the next step was to implement the Q-Learning agent. To support Q-Learning, the focus was turned back to the GameState object. Lists were added to the GameState object to store action and Q-value pairs for each state. This addition provides the necessary data for a Q-learning agent. Several functions were also added to the GameState class to support Q-learning. These functions include a Q-value update function that takes an action and a reward as an argument and updates Q-values appropriately, a function that returns the next move based on weighted exploration and Q values, and a function that returns the next move based solely on Q values with no exploration. Each of these additions was carefully made not to interfere with the already implemented search agents. With the necessary additions complete, a Q-learning agent could now be constructed.

In the interest of interchangeability, the Q-learning agent is derived from the same abstract Agent class as the search agents. The overridden solve function of the Q-learning agent is more reliant on the timeout argument than the search agents. For this agent, a timeout must be set, and Q learning is run for the amount of time specified by the timeout. At the end of the allowed time interval, the learning agent returns the best path it has found so far, based on Q values. During the learning process, Q-values are updated with next moves based on both existing Q-values and exploration. The updating of Q-values is dependent on rewards of the terminal state of the gameplay. If next moves are available, a reward of zero is given. If the game is over and one peg remains, a reward of 15 is given. If the game is over and more than one peg remains, the reward is the 1 minus the number

of pegs remaining, resulting in a negative reward, or a penalty. When the time limit is reached, the move sequence that is returned to the user is based solely on Q-values and not on exploration as the best-known path is desired at that point.

Although the three search agents and the Q-learning agent were the focus of this project, the class-based implementation makes new additions very simple, allowing many new agents to be added to this project in the future as desired. As long as they are derived from the Agent class, they will work interchangeably with any existing agents, with no disruptions to the program structure or operation.

With the state space and agents complete, work commenced on the construction of a graphical interface to represent the Peg-Solitaire games, the solutions found by the agents, and important comparison details between the operation of different agents. The graphical interface is a .NET Framework Windows Forms application. In the interface, drop-down menus allow the user to select premade or custom peg solitaire puzzles, the agent to solve them, and the time limit, if any, restricting agent runtime.

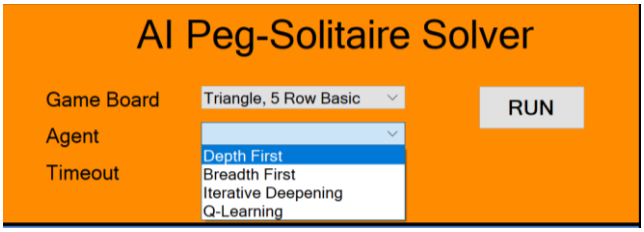


Figure 3: Configuration menu of the graphical user interface.

As peg solitaire games are selected, a visual representation of the game is displayed to the right of the configuration menu. For premade games, this display simply shows the user what the game looks like. If a custom game is selected, clicking the pegs and holes in the visual representation toggles the locations of the pegs. A dialog box is displayed with instructions for setting up a custom game.

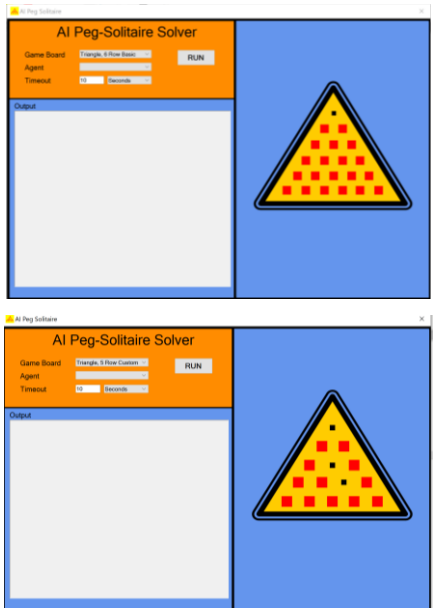


Figure 4: Visual display of premade and custom game boards.

When the run button is clicked, the selected agent is run on the selected puzzle and limited by the user-specified time limit. Depending on the outcome of the agent's attempt to solve the puzzle, there are three possible outcomes. If the agent finds a solution to the game, a dialog box is displayed, indicating success, and the number of states that were expanded to find the solution.

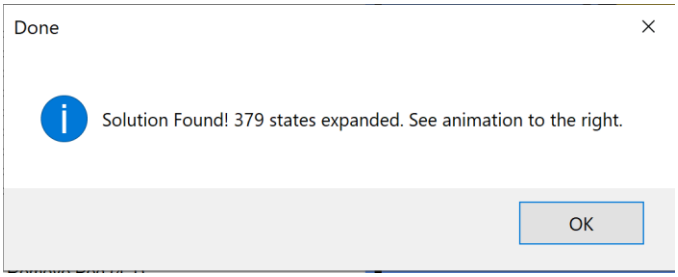


Figure 5: Dialog box after successful solution of a peg solitaire game

Once the dialog box is acknowledged, an animation of the full move sequence is played on the visual representation of the game. Additional information is given in the output window, including the elapsed time while solving the problem, the number of expanded states, and the full move sequence in coordinate form. The final display of the interface after all operations are complete is pictured below.

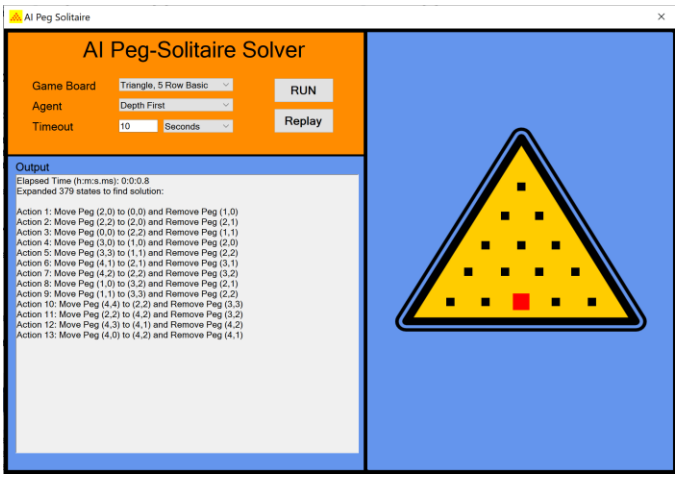
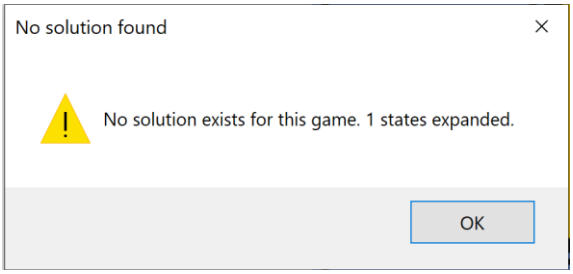


Figure 6: Full output after agent has been solved and animation showing the solution has finished.

If all possible states are searched and no solution is found, this no solution finding is displayed in a dialog box. Similarly, if the timeout is reached before there is a solution, another appropriate dialog box is displayed.



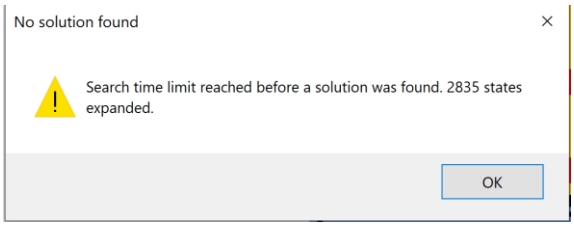


Figure 7: Dialog boxes produced when a solution is not found.

The completion of the graphical user interface finished the programming of the project and allowed easy analysis and comparisons of different agents and their abilities to solve different types of peg solitaire games. This interface also provides a user-friendly experience, allowing use as a teaching tool for artificial intelligence students and students learning to play peg-solitaire.

IV. Results

Once the game representation, agents, and graphical interface were completed, agents were compared for their performance on different types of puzzles, based on ability to find a solution, the time elapsed, and the number of states expanded to find the solution. Initial analysis was performed on the standard five and six row puzzles, moving later to custom puzzles. The standard puzzles are pictured below.

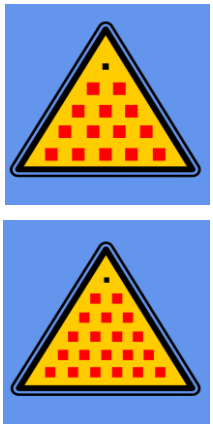


Figure 8: Standard 5 and 6 row peg solitaire.

The most effective agent in solving the peg solitaire puzzles of this project is the depth-first search agent. This agent solves the standard five and six row puzzles successfully on every run, never running for more than one second on the way to a solution. This agent explores 379 states before finding a solution to the five-row standard game and 1812 states before finding a solution to the six-row standard game. Both are remarkably efficient, given the exponentially growing complexity of the game as moves are made. The advantages of the depth-first search agent are clear. Depth-first search is known to be efficient in terms of time and memory, as it progresses rapidly through states expanding the deepest branch and moving to a solution. The drawback of such an agent is the lack of optimality in the result, often missing better routes that a breadth first agent would find. This lack of optimality is fundamentally never a problem for a peg solitaire game. This is caused by the nature of gameplay with each move removing a peg from the board. This means that any path to a goal state has the same length of one less move than the number of pegs. In other words, all paths to the goal are optimal, making the time and memory saving depth-first agent a perfect choice.

Despite the efficiency of the agent on the five and six row puzzles, it cannot solve anything more complex in a useful amount of time, running for over an hour with no found solution for puzzles of seven rows or more. With each state potentially having multiple pegs that can move in multiple directions, exponential growth of complexity is severe and very difficult to predict. It is for this reason that even a very efficient agent like depth-first search struggles beyond the six-row puzzle.

The next most effective agent was the iterative deepening search agent. This agent was found to be able to solve any puzzle that the depth-first search agent could solve, but with less efficiency. This agent solved the standard five-row puzzle in under a second but expanded 1109 states to reach its solution versus the 379 states of the depth-first agent. The iterative deepening agent also solved the six-row puzzle very quickly, taking less than a second but again was less efficient than the depth-first search agent, expanding 4674 states versus 1812. These differences can once again be attributed to the fact that all paths to a goal state are of the same length. Iterative deepening is intended to utilize some of the benefits of both depth-first and breadth-first search, with a better chance at finding an optimal path than depth-first search but less memory usage than breadth first. Once again, since all paths are the same length, the benefits of this agent are not able to be utilized. The result is no added benefit, but with some extra iterations at different depth limits, causing extra expansion of states.

The breadth first search agent was found to have significantly reduced ability compared to the depth-first and iterative deepening agent. This agent could not solve either the five or six row games in under an hour and when the tests were stopped after an hour of running, the search was only seven moves deep. With thirteen moves required to solve the five-row puzzle, nineteen to solve the six-row puzzle, and complexity growing exponentially, much more time would be required to reach a solution, if ever. The agent was not completely ineffective, however, successfully solving custom puzzles of reduced size such as the one pictured below in under a second of run time.

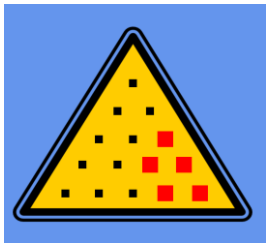


Figure 8: Dialog boxes produced when a solution is not found.

While successful, breadth-first search is still less efficient than depth-first search, expanding twenty-seven states to solve it versus twelve for depth-first. In a simple puzzle like this one, however, breadth-first search is more efficient than iterative deepening, which expands 30 states to find the solution.

Q-Learning shows some success, but not as much as the search agents. While it can quickly learn to solve simple puzzles, the agent struggles with larger, more complicated ones, often returning paths that lead to two or more remaining pegs at the finish. The exponential nature of the complexity of large peg solitaire games proves to be too much for the agent to be able to learn to play in a reasonable amount of time. The expansive amount of available states and moves requires too much exploring before reaching a goal state, greatly lengthening learning time. The below puzzle is solved consistently with only 10 seconds of learning time.

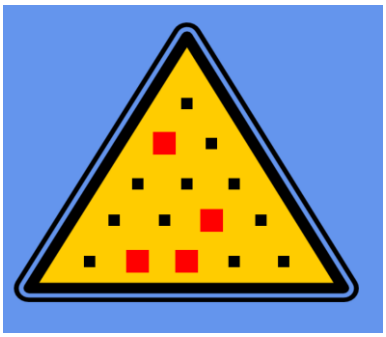


Figure 9: Puzzle consistently solved by Q-Learning.

The standard five and six row puzzles, and several simpler modifications were run multiple times, allowing up to five minutes of learning time. Success was not strong for this learning, with most paths resulting in three to four remaining pegs at the end of the game. This is not the worst possible result, but it is also not desirable. Learning for a longer period of time would likely improve results but was not feasible in the scope of this project.

The effect of symmetry is also very apparent, as suggested by the literature review and is clearly shown using the following games.

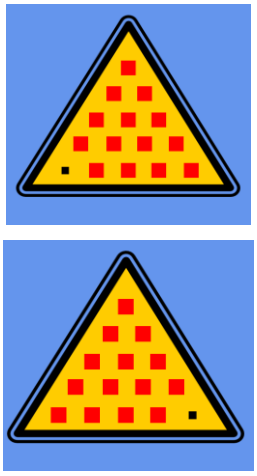


Figure 10: Symmetry variations on the standard puzzle.

While these games are truly the same as the standard five-row peg solitaire game due to rotational symmetry, they are not viewed the same by the agents solving the puzzles. A depth-first agent expands 7374 states to solve the puzzle with the open hole in the left corner and 7778 to solve the puzzle with the open hole in the right corner, taking one to two seconds to get to a solution. This is a very significant difference compared to the 379 expanded states for the puzzle with the open hole at the top. Iterative deepening is affected by this even more, expanding 14,946 states to solve the left hole puzzle and 15,650 states for the right hole puzzle, running for over four seconds before reaching a solution.

V. Discussion

The modeling of peg solitaire games and subsequent comparison of the effectiveness of agents was quite successful, providing much valuable information and areas for future project expansion.

The depth first search agent was the most successful by a significant margin. This was anticipated due to its low memory usage, quickly traversing the data set to reach a terminal goal state. The tendency of depth-first agents to return non-optimal paths is never a problem in peg solitaire, as path length to a goal state fundamentally cannot vary. As a result, all the benefits of a breadth-first search are enjoyed, without the main drawback of the approach. The efficient nature of this agent also made it particularly useful for learning about the game of peg solitaire, itself. The agent could be run many times on many puzzles, returning fast results. One major discovery made by the depth-first agent was that there are no solutions to a full triangle puzzle with less than five rows. This helped to direct the construction of the interface, and the test cases used with other agents. As efficient as the depth-first agent was, its limitations become apparent with puzzles larger than 6 rows. The inability to solve a seven or eight row puzzle was a surprising result given that six row puzzles were solved almost instantly. This demonstrated just how drastic the exponential increase in complexity is as the number of puzzle rows in increased.

The iterative deepening agent had largely the same capabilities as the depth-first agent, being found to solve any puzzle that the depth-first could solve. This makes sense, as iterative deepening is a modified version of depth-first search. While the agents are almost equally capable in solving ability, no true advantage was found to using iterative deepening to solve this type of game. Iterative deepening makes repeated runs of depth first search with an increasing depth limit with an intent of using less time and space than breadth first search but with a better chance at finding an optimal solution. Since optimality is not a concern in peg solitaire, the benefit of iterative deepening is lost. For this reason, the iterative deepening agent was found to provide no additional functionality, while expanding more nodes and taking longer to run than a depth-first agent, leaving depth-first search as a superior algorithm for solving peg solitaire.

Breadth first search was found not to be very well suited at all for solving this type of puzzle. The exponential complexity of the states and possible actions proved pathological for the breadth-first agent, seriously hindering its ability to solve the games. After an hour of running on the full five row puzzle, the agent was not even remotely close to finding a solution, searching only seven moves into a sequence of thirteen moves with exponential complexity. This type of time usage indicates that the agent will never find a solution in any useful time. While the breadth-first agent could solve small, simpler puzzles, these puzzles would also be relatively easy for a human to solve and

provide very little strain on another type of search agent. For these reasons, breadth-first search was determined to be very poorly suited for peg solitaire. The advantage of optimality is eliminated, and the drawback of time and data cost is very apparent.

Q-learning was found to be limited in its usefulness in solving the peg solitaire games but showed enough promise to encourage future investigations. The Q-learning agent could very quickly solve simple puzzles, with only ten seconds of learning, but it struggled with solving more complicated ones, unable to learn them in much longer periods of time. This data indicated that learning was occurring, but not at a fast-enough rate to make the agent as useful as or more useful than the search agents. The slow learning rate is likely due to the sheer magnitude of available states and actions that must be explored before a goal state can be found. There are very many possible paths and very few that lead to a goal, making exploration relatively ineffective. This limitation of Q learning led to analysis of how humans play the game, as a human can in fact learn to play peg solitaire in less attempts than the Q learning agent. This analysis revealed an important difference between the human approach and the Q-learning approach to peg solitaire. The Q-learning agent acts randomly when exploring, but humans can think ahead. If a heuristic could be developed that directed the exploration of Q learning toward states more likely to lead to a goal, the algorithm could be greatly improved. There is enough merit to this type of approach that while it was beyond the scope of this project, it should be explored in the future. If a proper heuristic can be developed, the Q learning agent could very possibly surpass the ability of the model-based search agent solutions.

The graphical interface proved very helpful, as expected. The animations of the actions taken to a goal state of a game were much easier to follow than the coordinate-based move sequences returned by the functions. Elapsed time and expanded states were also clearly displayed, providing an easy way to access most of the information needed for analyzing the agents. All functionality of the interface worked very well and made the project much more user-friendly than if it used a simple terminal window. One change could improve the functionality of the interface in future versions. If problem solving agents were run in the background on a separate thread from the interface, it would allow the interface to continue responding to user input while agents were running. This would provide a much better experience while running slow agents like breadth-first search or Q-learning. An attempt was made to implement this during the project, but it was found to be too difficult and time consuming of a task to implement without taking time away from more valuable artificial intelligence work. Future, less time sensitive versions, however, could certainly benefit from this feature.

VI. Conclusions

This project successfully modeled and solved peg solitaire with the use of artificial intelligence. The game board of interest was the American triangle game board, but the flexible state representation allowed many user-configurable variations of the game. These included two board sizes and many different starting peg configurations on the standard five-row board. Depth-first, breadth-first, and iterative deepening search agents were implemented and compared for their ability to solve different variations of peg solitaire. The depth-first agent was the most successful, able to solve full five and six row games. Iterative deepening was a close second, with the ability to solve the same games as the depth-first search agent but using more time and memory in the process. Breadth-first search was limited in capability, using large amounts of time and space, and struggling to solve anything more than a simple puzzle. It was determined that only the drawbacks of breadth-first search apply to peg solitaire and not the benefits, making it a poor option to solve the puzzles.

Q-learning showed some promise, able to quickly learn to solve simple puzzles, but struggling to learn the path to the goal of more complicated games. The Q-learning agent would benefit greatly from additional learning time as well as addition of heuristics to direct exploration toward the goal state rather than simply guessing randomly. The use of such a heuristic, coupled with large amounts of learning time should be the focus of future expansions of this project, and could find great success in solving peg solitaire.

In addition to the state representation and solving agents, a full graphical interface was constructed to allow user configuration of different games and agents. An animated representation of gameplay was successfully implemented such that every action an agent took to solve the puzzle could be visualized and compared. These additions were very helpful for the development and data collection process and make the project an appealing learning tool for students studying artificial intelligence or the game of peg solitaire.

All objectives of the project were achieved, with success in the state representation and all agents to operate in it. The Q-learning agent left room for improvement and should be the focus of a comprehensive long-term learning and heuristic study on peg solitaire in the future. The modeling of peg solitaire and comparison of solving agents sought out by this project were very successful and provided important data for future more expansive studies.

References

[1] C. Jefferson, A. Miguel, and Armagan Tarim, “Modelling and Solving English Peg Solitaire,” University of York, 2003 pp. 1-15.

[2] A. Matos, “Trying to Solve Peg Solitaire with a Simple Depth-First Search”, University of Portugal, November 2002, pp.1-15.

[3] G. Bell, “Solving Triangular Peg Solitaire”, Tech-X Corporation, Journal of Integer Sequences Vol. 11, 2008, pp.1-22.