Sixes wild agent

Group 7 -Nick Panzarino, Cameron Jones, Joey Perez, Jeff Sirocki

Worcester Polytechnic Institute

CS 4341 – Introduction to Artificial Intelligence

Table of Contents

[Project Overview 4](#_Toc432787635)

[Background on Sixes Wild 4](#_Toc432787636)

[Introduction 4](#_Toc432787637)

[Game Types 4](#_Toc432787638)

[Puzzle 4](#_Toc432787639)

[Lightning 4](#_Toc432787640)

[Elimination 4](#_Toc432787641)

[Release 5](#_Toc432787642)

[Moving and Scoring 5](#_Toc432787643)

[Special Moves 5](#_Toc432787644)

[Swap Two 5](#_Toc432787645)

[Remove One 5](#_Toc432787646)

[Shuffle Board 5](#_Toc432787647)

[User Interface 6](#_Toc432787648)

[Approach 6](#_Toc432787649)

[Problem Description 6](#_Toc432787650)

[Attempt #1 6](#_Toc432787651)

[Results for #1 6](#_Toc432787652)

[Idea behind attempt #1 8](#_Toc432787653)

[Attempt #2 8](#_Toc432787654)

[Results for #2 8](#_Toc432787655)

[Idea behind attempt #2 9](#_Toc432787656)

[Attempt #3 9](#_Toc432787657)

[Results for #3 9](#_Toc432787658)

[Idea behind attempt #3 10](#_Toc432787659)

[Attempt #4 11](#_Toc432787660)

[Results for #4 11](#_Toc432787661)

[Idea behind attempt #4 11](#_Toc432787662)

[Attempt #5 11](#_Toc432787663)

[Results for #5 11](#_Toc432787664)

[Idea behind attempt #5 12](#_Toc432787665)

[Analysis and Discoveries 13](#_Toc432787666)

[Graphs 13](#_Toc432787667)

[Conclusions 14](#_Toc432787668)

# Project Overview

The goal of this project was to take an existing platform, Sixes Wild, developed as part of Software Engineering, and apply the search techniques developed in Introduction to Artificial Intelligence, to create an agent capable of playing Sixes Wild, in all four of its game types. This is a particularly interesting problem, because it gives a real world application for the search algorithms to a unique problem. It also required us to integrate intelligent behavior into an existing application, but that application was one we were all familiar with from a previous course. This can be run using sixeswild.jar

# Background on Sixes Wild

## Introduction

Sixes Wild is a Candy Crush-esque game that includes multiple game modes where players remove numbered tiles by selecting tiles that add up to six. The application also features a robust level builder, but this is unrelated to our work. In our project, we expanded upon the framework to allow the player to select an agent move with the ‘Auto’ move button in the game. The auto move feature gives the agent full control to make the next move. Once the agent determines the move, it selects the tiles on the board, the move is executed, and the board is updated. In our project, we were successful in having the agent work properly for each of the four game types.

## Game Types

The game types in Sixes Wild include Puzzle, Lightning, Elimination, and Release. Each game type involves the same scoring system but the rules and objectives change for each game type. Our agent distinguishes game types from each other and acts accordingly to the semantics of the game type.

### Puzzle

The puzzle game type is the most basic version of the game. In this game the player is given a set number of moves to reach certain score thresholds by selecting tiles that add to six. As normal, each move removes squares from board and contents are repopulated by shifting downward to fill in holes, with new squares appearing at top. Our agent makes the greedy move to maximize its score for this game type.

### Lightning

The lightning game type is the most similar version to the puzzle game type. In this game the player is given a time limit and an unlimited number of moves to reach certain score thresholds by selecting tiles that add to six. As before, each move removes squares from board and contents are repopulated by shifting downward to fill in holes, with new squares appearing at top. Our agent will make moves to minimize time and maximize its score for this game type.

### Elimination

The Elimination game type is similar to the puzzle game type except for one new rule. In this game the player is given a set number of moves to reach score thresholds and with a catch that in order to finish the level the player has to additionally mark each square to complete the level. A player can mark squares by executing moves on any unmarked square. There are no restrictions on moves in this game mode, meaning a player can choose marked squares, unmarked squares, or a combination of both to make moves. As before, each move removes squares from the board and contents are repopulated by shifting downward to fill in holes, with new squares appearing at top. Our agent will make moves to maximize its score and ensure that it marks each square for this game type.

### Release

The release game type is another branch off the puzzle variation. In this game the player is given a set number of moves to reach score thresholds and additionally get rid or ‘release’ all “6” tiles on the board by dropping them into buckets. Once all the “6” tiles have been released the level completes. As before, each move removes squares from the board and contents are repopulated by shifting downward to fill in holes, with new squares appearing at top. Our agent will make moves to maximize its score and ensure that it releases all the “6” tiles to complete the level for this game type.

## Moving and Scoring

Standard moves in Sixes Wild are the same for all game modes. Moves are made by selecting adjacent squares that add up to 6. Each move is assigned points. For any move we calculate the base score with the equation:

Base = 10 x number of squares in move. (Base points are then from 20 to 60)

Additionally, some Squares may be given “bonus multipliers” (up to x3). The score is enhanced by product of bonuses. For example, if we selected tiles “2”, “2”, “2” and one two had a x2 multiplier and another had a x3 multiplier the score would be:

Example Score: 10 x 3 x (2 x 3) = 150

Each game type uses these scores to determine whether a player receives one, two, three, or no stars. The number of stars rewarded to a player for completing a level depends on the score thresholds. An example of this is, if a player gets 5,000 points and the threshold for 3-stars is 4,750, then the player will get a 3-star completion for that level. A minimum of one star is required to move on to the next level.

## Special Moves

Special moves are only available on a given level if the checkbox for that move is checked in the level builder. The score of all special moves is zero, however they can be used to create a situation which may be more advantageous to the player, or if there are no other valid moves remaining to be played. The types of special moves are as follows:

### Swap Two

This move selects two adjacent squares and simply swaps their positions.

### Remove One

This move removes a single square from the board. Cannot be applied to “Six” squares.

### Shuffle Board

Remove all tiles from the board and re-Distribute them randomly. “Six” square remain where they were, unaffected by this move.

## User Interface

# Approach

## Problem Description

The Agent’s task is to find the optimum move to make, given that there are multiple board types, each with different criteria for success. We want to optimize this agent by reducing the time required to select the move, so that the Agent plays at least as quickly as a human player would. To do this we will evaluate the time taken to select the optimum move, and see which approaches are most effective.

## Attempt #1

Initially we developed a brute force method which would iterate through every position on the game board for the simplistic Puzzle Level, and evaluate all possible move choices, storing the best choice as it went. This would provide a baseline for our AI, and a form to follow for developing the other game types and their auto moves. To determine a list of all possible moves, we start by finding all possible selections of incremental sizes. First each square of the largest possible board (9x9) is added to a list of selections. Then each selection in the list is expanded to create new selections by adding each adjacent square. This process is repeated until all possible selections of all possible shapes of a certain size (i.e 6) are generated and stored in the list. Then moves are generated from those selections, and the highest scoring move is selected and applied to the game.

### Results for #1

These results serve as a benchmark for the rest of the testing and to guide our search in areas to improve our algorithm.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Level | | | | |
| Trial # | Puzzle 1 | Puzzle 2 | Lightning 1 | Elimination 1 | Elimination 2 |
| 1 | 10 | 10 | 11 | 35 | 10 |
| 2 | 10 | 10 | 10 | 35 | 10 |
| 3 | 10 | 10 | 12 | 35 | 10 |
| 4 | 10 | 10 | 11 | 35 | 10 |
| 5 | 10 | 10 | 11 | 35 | 10 |
| Average Number of Moves | 10 | 10 | 11 | 35 | 10 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Level | | | | |
| Trial # | Puzzle 1 | Puzzle 2 | Lightning 1 | Elimination 1 | Elimination 2 |
| 1 | 11280 | 3530 | 240 | 4260 | 320 |
| 2 | 12800 | 2450 | 180 | 3950 | 310 |
| 3 | 17500 | 4850 | 290 | 4420 | 270 |
| 4 | 10370 | 3210 | 260 | 4230 | 310 |
| 5 | 14200 | 6010 | 230 | 4310 | 250 |
| Average Score | 13230 | 4010 | 240 | 4234 | 292 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Level | | | | |
| Move # | Puzzle 1 | Puzzle 2 | Lightning 1 | Elimination 1 | Elimination 2 |
| 1 | 112470522 | 2001773642 | 14329695651 | 4066400048 | 12649561859 |
| 2 | 94875426 | 1488329808 | 13705224921 | 3387971606 | 13692021099 |
| 3 | 90607712 | 1517088227 | 15134183718 | 3386467583 | 14233409758 |
| 4 | 80322722 | 1453150277 | 12662456192 | 3396399267 | 12968829617 |
| 5 | 106638386 | 1420197173 | 15148455751 | 3412553894 | 16932473317 |
| 6 | 89365413 | 1437837665 | 15947168134 | 3477914153 | 14678817278 |
| 7 | 83210761 | 1511598741 | 14271887877 | 3648816164 | 15811751986 |
| 8 | 82158340 | 1420007295 | 15937196185 | 3430343999 | 15653816942 |
| 9 | 87824677 | 1452445241 | 14886359815 | 3765373207 | 16430646777 |
| 10 |  | 1439897110 | 17069226901 | 4408031956 | 14051038881 |
| 11 |  |  | 15961934007 | 3385206336 |  |
| 12 |  |  |  | 3403677000 |  |
| 13 |  |  |  | 3459983910 |  |
| 14 |  |  |  | 3563416477 |  |
| 15 |  |  |  | 4444507474 |  |
| Average Time Per Move in NanoSecond | 91941551 | 1514232518 | 15004889923 | 3642470872 | 14710236751 |

It should be noted that all times are reported in nanoseconds, measured by getting the system time before the call to the function that determines the best move, and subtracting it at the end of the decision logic from the new system time.

### Idea behind attempt #1

The idea was to get a simple search algorithm operational. We succeeded but quickly determined that the time required was too long to be an effective approach so we would need to speed up the processing of the moves.

## Attempt #2

For the second attempt, we threaded the program, so that as the game is starting up, a second thread develops a look-up table of all possible move combinations, so that the expansion of the moves only happens once at the start, rather than every time an auto move is requested.

### Results for #2

This method succeeded in reducing the time complexity of our initial algorithm. The lookup table took approximately 15 seconds to build after the game started up, however after this time, auto-moves could be completed on any level almost instantaneously.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Level | | | | |
| Trial # | Puzzle 1 | Puzzle 2 | Lightning 1 | Elimination 1 | Elimination 2 |
| 1 | 10 | 10 | 38 | 35 | 10 |
| 2 | 10 | 10 | 37 | 35 | 10 |
| 3 | 10 | 10 | 39 | 35 | 10 |
| 4 | 10 | 10 | 40 | 35 | 10 |
| 5 | 10 | 10 | 45 | 35 | 10 |
| Average Number of Moves | 10 | 10 | 39.8 | 35 | 10 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Level | | | | |
| Move # | Puzzle 1 | Puzzle 2 | Lightning 1 | Elimination 1 | Elimination 2 |
| 1 | 9781832192 | 79345699 | 133260777 | 48461120 | 141529350 |
| 2 | 11682035 | 49650127 | 149760817 | 54655642 | 136596470 |
| 3 | 4159551 | 24531759 | 150000040 | 41743150 | 144890308 |
| 4 | 3969278 | 24181611 | 131210016 | 44763039 | 135153636 |
| 5 | 2290773 | 22061766 | 131374235 | 47439490 | 133188142 |
| 6 | 2228796 | 18758443 | 130993294 | 41884078 | 139805843 |
| 7 | 2245771 | 19455188 | 142518216 | 42678724 | 149027754 |
| 8 | 2244982 | 19164252 | 137158999 | 41102066 | 154032874 |
| 9 | 2179847 | 26546597 | 150001618 | 43776937 | 137190185 |
| 10 |  | 20654459 | 161008224 | 42481741 | 167215378 |
| 11 |  |  | 140631674 | 43381391 |  |
| 12 |  |  | 143493265 | 52115778 |  |
| 13 |  |  | 145147295 | 42626617 |  |
| 14 |  |  | 137138077 | 43307572 |  |
| 15 |  |  | 129892713 | 43360074 |  |
| 16 |  |  | 141247100 | 52787653 | 143862994 |
| 17 |  |  | 130897369 | 52321841 |  |
| 18 |  |  | 128697784 | 43168617 |  |
| 19 |  |  | 131249492 | 42667670 |  |
| 20 |  |  | 137781136 | 42236992 |  |
| 21 |  |  | 128827265 | 42832285 |  |
| 22 |  |  | 130180096 | 42548060 |  |
| 23 |  |  | 131599246 | 41996979 |  |
| 24 |  |  | 142480320 | 42043561 |  |
| 25 |  |  | 143396155 | 43585875 |  |
| 26 |  |  | 129522037 | 49526964 |  |
| 27 |  |  | 142308995 | 44983314 |  |
| 28 |  |  | 124550076 | 41597091 |  |
| 29 |  |  | 128459746 | 41494454 |  |
| 30 |  |  | 129872975 | 41106014 |  |
| 31 |  |  | 137982857 | 42784519 |  |
| 32 |  |  | 126157525 | 42284757 |  |
| 33 |  |  | 126440171 | 43264148 |  |
| 34 |  |  | 127893271 | 41529587 |  |
| 35 |  |  | 126354114 | 41895527 |  |
| 36 |  |  | 128605806 |  |  |
| 37 |  |  | 127357981 |  |  |
| 38 |  |  | 129737574 |  |  |
| Average Time Per Move in Nanoseconds | 1090314803 | 30434990.1 | 135399693.4 | 44298095.06 | 143862994 |

### Idea behind attempt #2

The idea behind the lookup table was to reduce the overhead of expanding all possible moves to only occurring once, rather than each time a move is called. Iterating through a list of all possible moves is a much faster operation than creating that list for every move. This behavior was implemented successfully. We are now able to make moves in an acceptable time frame, similar to what it would take a human player to determine a move.

## Attempt #3

We decided the next step was to implement the search for another one of the game types, Elimination, where not only did we need to maximize score, but in order to complete the level we needed to use every square at least once.

### Results for #3

For this attempt, the Lightning Level was not considered, because it runs on the same code and algorithm as the Puzzle Levels, and the goal of this attempt was to improve upon the efficiency of the Elimination Level algorithm compared to the baseline of no special algorithm other than the generic one used on Puzzle Levels.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Level | | | |
| Trial # | Puzzle 1 | Puzzle 2 | Elimination 1 | Elimination 2 |
| 1 | 10 | 10 | 16 | 10 |
| 2 | 10 | 10 | 14 | 10 |
| 3 | 10 | 10 | 18 | 10 |
| 4 | 10 | 10 | 20 | 10 |
| 5 | 10 | 10 | 17 | 10 |
| Average Number of Moves | 10 | 10 | 17 | 10 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Level | | | |
| Move # | Puzzle 1 | Puzzle 2 | Elimination 1 | Elimination 2 |
| 1 | 1685940267 | 84116727 | 50100545 | 144287910 |
| 2 | 11040950 | 68714112 | 58627288 | 134144242 |
| 3 | 3790848 | 22875361 | 40839553 | 132884969 |
| 4 | 3406750 | 23384991 | 41267864 | 132976947 |
| 5 | 3403987 | 18977533 | 39927665 | 133998183 |
| 6 | 2948043 | 22231907 | 43901680 | 133816200 |
| 7 | 2574998 | 24149240 | 41284048 | 132541136 |
| 8 | 5903587 | 20011400 | 43085323 | 134486891 |
| 9 | 3295034 | 20107326 | 40805209 | 131701488 |
| 10 | 2991466 | 20956843 | 39950166 | 135281142 |
| 11 |  |  | 41218124 |  |
| 12 |  |  | 40029512 |  |
| 13 |  |  | 42634117 |  |
| 14 |  |  | 41366947 |  |
| 15 |  |  | 39523434 |  |
| 16 |  |  | 40200441 |  |
| Average Time Per Move | 172529593 | 32552544 | 42797619.8 | 134611910.8 |

### Idea behind attempt #3

At first our attempt was not successful, because we based it entirely off the general search used for Puzzle and Lightning Levels, which although it was selecting the best move to maximize score, it had no knowledge of which squares it had already used. We tapped into the framework of the game, and let it check to make sure the move it thinks is best actually moves along towards the intended goal. We do this by checking the selection it made, and determining if there are any squares selected that haven’t been used yet. If not, it checks if there are no unused squares left, otherwise it attempts to shuffle the board, assuming a shuffle move is allowed on that board. Otherwise the AI reaches an impasse and throws an error because it cannot move forward.

## Attempt #4

The next step was to implement our search at depth greater than 1, which would give us the ability to forecast ahead to determine if a move that is good now causes future moves to be less beneficial, decreasing the overall optimality of the initial move.

### Results for #4

After evaluating the framework for the game, it became clear that it would prove to be too large an undertaking to revamp the framework to allow us to compare entire game states against one another, and to create all of the game states for all possible future moves. This is because we would need to de-reference the active board that is displayed, as well as all other visual elements, then store those games in a way which would allow us to compare them. Also, due to the number of possible moves, the exponential expansion of the search tree would be too costly to run in a reasonable amount of time.

### Idea behind attempt #4

We felt that looking ahead at moves would be a more optimal solution since it gave the agent a more holistic view of the impact a given move would make. This would be akin to the difference between a simple greedy algorithm which selected the best choice, and an A\* algorithm which could back off a move after finding a suboptimal path. This approach would be preferred, but because the map of the game is changing dynamically with every move, it requires the storing of each of the potential boards to go past a search of depth 1. Ultimately, we made the decision to not go further with this implementation since it would be too costly to implement in the scope of this project.

## Attempt #5

Moving forward, the next approach was to try to make the elimination algorithm more efficient, since a simple greedy search with the stipulation that at least on the squares in the selection was unmarked, could be improved by favoring selections with multiple unmarked squares, over ones with fewer unmarked squares.

### Results for #5

We again decided to focus our data collection on a smaller area, because our goal was to improve further on the Elimination efficiency. The following data only considers the Elimination Levels and their timer per move and number of moves required. We decided score was not a good metric because of the variance in the random tiles even with the same board layout.

|  |  |  |
| --- | --- | --- |
|  | Level | |
| Move # | Elimination 1 | Elimination 2 |
| 1 | 7762129886 | 142158593 |
| 2 | 68715296 | 134889543 |
| 3 | 42882023 | 133750276 |
| 4 | 42898604 | 134287539 |
| 5 | 40097805 | 129725336 |
| 6 | 42116985 | 133146693 |
| 7 | 41936187 | 131385288 |
| 8 | 42813336 | 133336175 |
| 9 | 44499342 | 132545478 |
| 10 | 42661355 | 132245068 |
| 11 | 42160014 |  |
| 12 | 41613670 |  |
| 13 | 42008033 |  |
| 14 | 42767939 |  |
| 15 |  |  |
| Average Time Per Move | 595664320 | 133746998.9 |

|  |  |  |
| --- | --- | --- |
|  | Level | |
| Trial # | Elimination 1 | Elimination 2 |
| 1 | 14 | 10 |
| 2 | 15 | 10 |
| 3 | 12 | 10 |
| 4 | 16 | 10 |
| 5 | 14 | 10 |
| Average Number of Moves | 14.2 | 10 |

### Idea behind attempt #5

The idea came from a meeting where we were talking about ways that we as humans stragized our moves to make the best use out of the tiles and moves we had at hand. Because the game limits the number of moves we can make, it isn’t realistic to assume there are always enough moves available to let the search algorithm only use one unmarked square at a time. We decided to adjust the score function to multiply the total score of the move by the number of tiles that are in the selection, and are unmarked. This will favor moves that although might not be the highest scoring, help reach the objective of marking all the tiles which is a requirement for advancing to the next level.

# Analysis and Discoveries

## Graphs

## Conclusions

After going through a number of iterations, we were able to take a broad problem of building and intelligent agent to play Sixes Wild, and create an efficient agent with a minimal time to calculate the next move. There were some issues along the way, where the framework wasn’t as robust as we thought, which led to difficulties getting the Release Level type working with the agent. However we had success in taking a generic algorithm implemented for the Puzzle Level, and see a significant increase in performance by going after more attributes rather than solely evaluating the score associated with a given selection.

One interesting thing that came out of the data collection was that the performance on Puzzle 1 was negatively impacted when we adjusted our algorithm to make use of the lookup table. We believe that anomaly is due to the cost of building up the lookup table at the start. Because our test procedure was to go through the levels in a given order, it would make sense that the initial loading of the table would primarily affect Puzzle 1. We were able to achieve a time reduction of at least a factor of 10 for all of the Puzzle Types with the lookup table.

It should also be noted that the graph of average number of moves shows the Elimination Levels decreasing in the number of moves, which is good, but conversely, Lightning Level’s number of moves is increasing, which is also a good thing. For the lightning level, increasing the number of moves made is an indicator of more points being accrued within the time limit. For Puzzle, the number of moves is unaffected, because there is a fix number of moves that are allowed before that level ends.