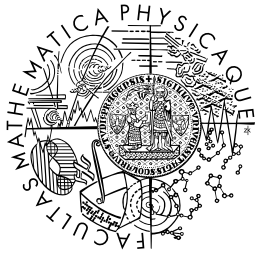


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Michal Počatko

AI for the Board Game Azul

Department of Software and Computer Science Education (KSVI)

Supervisor of the bachelor thesis: Adam Thomas Dingle, M.Sc.
Study programme: Programming and software systems
Study branch: Computer scienece

Prague 2020

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

Dedication.

Title: AI for the Board Game Azul

Author: Michal Počatko

Department: Department of Software and Computer Science Education (KSVI)

Supervisor: Adam Thomas Dingle, M.Sc., KSVI

Abstract: A comparison between three different approaches to developing an AI agent for the board game Azul and their implementation, testing and consequent results of said tests. A part of the thesis is also a simulator created in a game engine for playing against a local player or an artificial intelligence agent.

Keywords: artificial intelligence board game Azul

Contents

Introduction

0.1 About Azul

Azul is an award-winning board game for two to four players created by the German game designer Michael Kiesling and published by Plan B Games Inc. in 2018. The game has a simple set of rules that allow for the implementation of various interesting strategies. The goal of game is to acquire the highest possible score by placing a different colored tiles on a wall. The game involves an element of randomness, as at the beginning of each round, tiles are randomly placed on the factories.

0.2 Motivation

Since Azul is a fairly new board game (created in 2018), as far as I found, there are no papers that research AI for it and the few user interfaces that do exist are rather unintuitive and stiff. This creates a great opportunity to make something new and exciting.

0.3 Goals

The main goal for the thesis is to implement three different artificial intelligence agents for the board game Azul. A secondary goal is to also implement a graphical user interface to be able to play against an AI agent or locally against another human, and also a console user interface to be able to test different AI agents against each other.
Although according to the rules of the game it is possible for up to four players to play, in our AI research we will be focusing mainly on a game between two players. Although the GUI will support more players, the AI research will not be conducted beyond the game of two players.

0.4 Acknowledgment

I would like to give a special thanks to the leader of this project, Adam Thomas Dingle , M.Sc for his part of work on this project.

0.5 Intended audience

of this thesis is anybody interested in game theory and board game strategies. I also believe it would serve as an excellent GDNative and c++ usage example, for people struggling to use this relatively new and little documented library of Godot.

0.6 Thesis structure

Aside from the introduction and the epilog, the thesis will consist of four main chapters

1. Game rules	Detailed explanation of the rules of the game Azul
2. Analysis	An analysis of the game from a game-theoretic perspective
3. Implementation	Implementation details for the game framework and its user interface
4. Artificial intelligence	An in-depth look on the types of AI used, their success rates and testing results

1. Game rules

In this chapter we will go over the rules of Azul in detail. Note that this is not a comprehensive description of the rules, as we only describe the two-player game, though in the official rules it is possible for more than two players to play. Also, we purposely omitted some rules that can be added to the base rules for variation purposes. For a comprehensive manual, please refer to the Azul rulebook [Pla, 2018]

1.1 Terminology

Here we explain several basic terms we will be using later in the thesis. The terms used here come directly from the game manual for the English version of the game.

1. Tile	The most fundamental piece in the game. There are five different colors of tiles and one special starter tile.
2. Pattern line	A buffer area for tiles, where they are put before being placed on the wall at the end of a round.
3. Wall	An area on the board to which tiles are added at the end of each round.
4. Floor	The area where tiles are put when they overflow a pattern line or don't have any other place to go. For every tile on the floor, penalty points are given.
5. Factory	A drawing area for tiles.
6. Center	Another drawing area for tiles.

All terms described here can be seen in 1.1

1.2 Beginning of the game

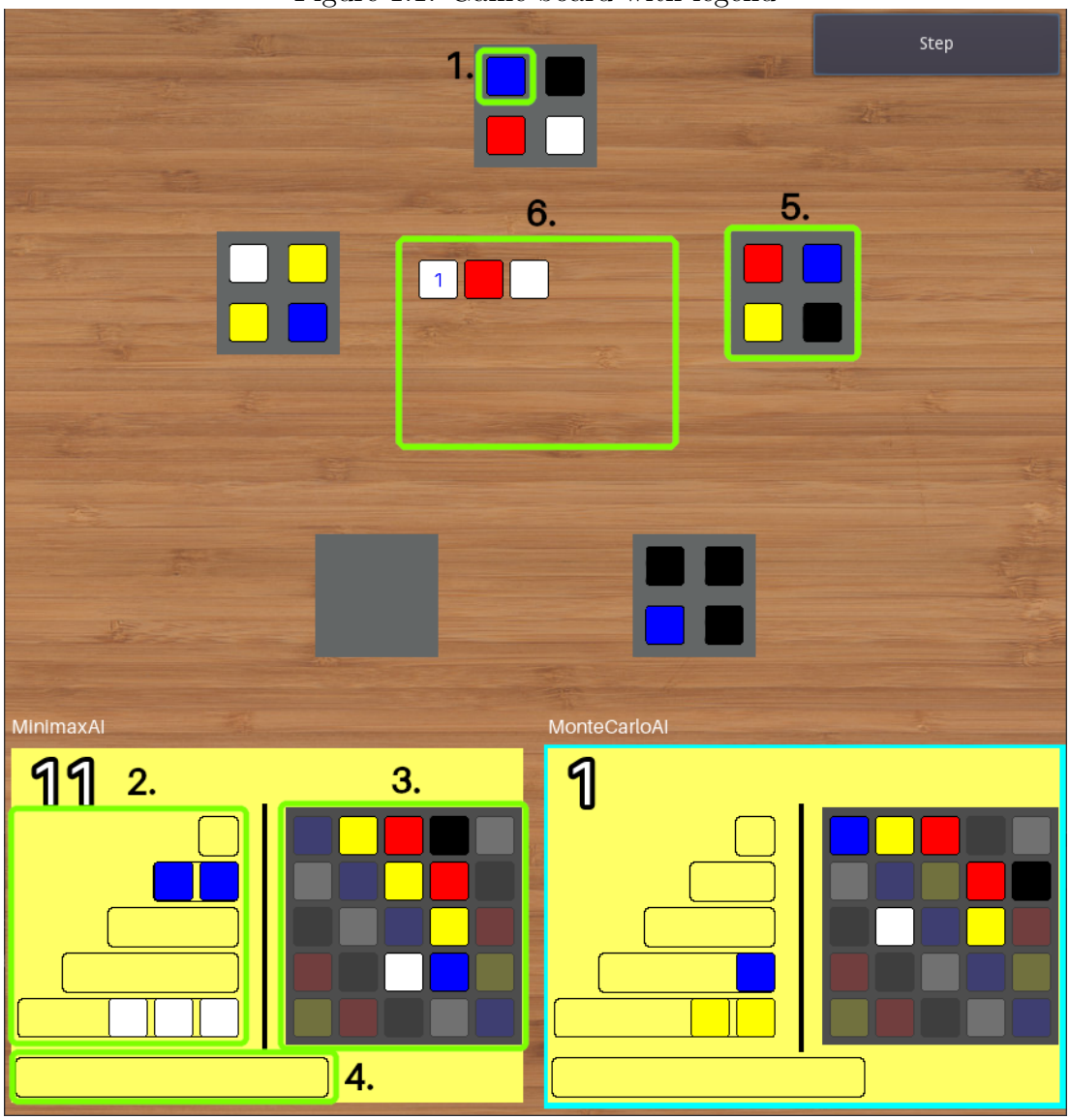
At the beginning of the game, there are 100 tiles in the bag, 20 of each color. Both players receive a board and 5 factories are placed in the center. The starting player is chosen and he receives a starter tile, which is a special sort of tile, that has a few functions throughout the game, but at the beginning of the round, it signifies that the player that holds it is the one that will be the first to move in the upcoming round.

1.3 Round

The game consists of a series of rounds. Each round consists of 3 stages:

1. Round setup

Figure 1.1: Game board with legend



2. Factory offer
3. Wall tiling

1.3.1 Round setup

At the beginning of the round, for each factory, we draw 4 random tiles from the bag and place them on the factory. See 1.2. If the bag is empty at any point, it is refilled with all tiles from the lid and then the filling of pattern lines resumes.

1.3.2 Factory offer

After the round is set up, players take turns taking tiles either from factories or from the center, starting with the player that holds the starter tile, and placing them on pattern lines of their choice.

On each turn, a player draws all tiles of one color from any factory he chooses or from the center. He may then choose a pattern line where he will place the tiles. However, he can only choose a pattern line that is not full and is either empty or contains tiles of the same color as the ones the player just drew. A pattern line may never contain tiles of a color that is already present on the wall in the corresponding line. If the number of tiles drawn is greater than the number of empty spaces in the pattern line, the tiles that won't fit on the pattern line must be placed on the floor. However, the floor can only hold seven tiles. When overflowing tiles no longer fit on the floor, they will be placed into the lid. A player can also choose to place tiles directly on the floor rather than on any pattern line.

When a player draws tiles from a factory, the tiles remaining in the factory are then placed in the center. If the player is the first in the round to move, he also places the starter tile into the center as well. If a player draws from the center while the center contains the starter tile, the player also takes the starter tile and places it on the floor. Since he now holds the starter tile, in the next round he will be the first to move.

Let's illustrate an example of a move a player could make in the situation at 1.2. Suppose that he decides to draw yellow tiles from the topmost factory and place them on the first pattern line. This is a legal move, as the first pattern line is empty and the first line of the wall doesn't contain a yellow tile yet. Since the number of tiles drawn is greater than the number of empty spaces in the pattern line, one tile overflows and must therefore be placed on the floor. The player also holds a starter tile, so he puts it in the center. The remaining tiles in the factory (red and black) are also placed in the center.

1.3.3 Wall tiling

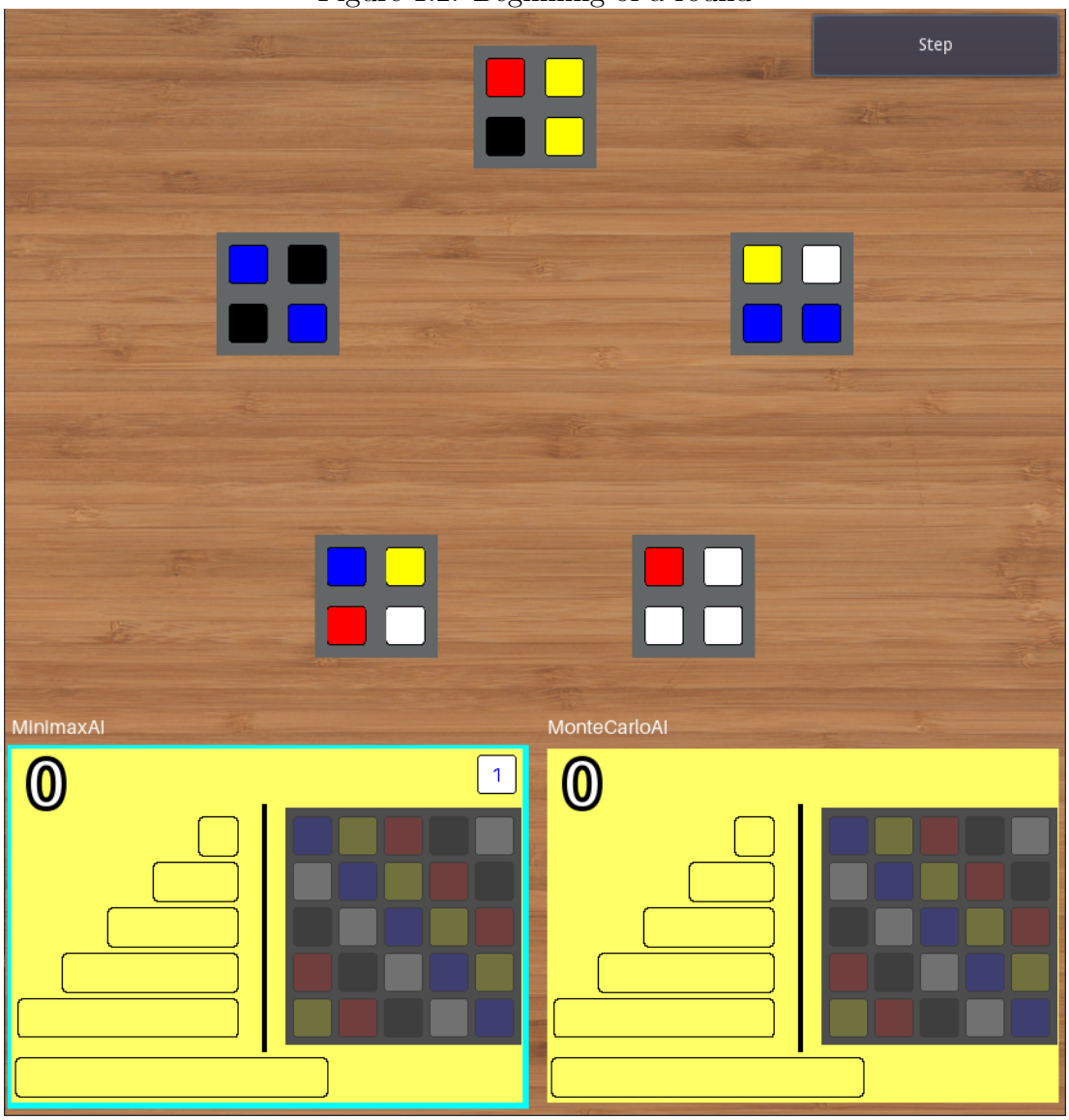
When all the factories and the center are empty, the factory offer phase of the round is over and the wall tiling phase begins. In this phase, tiles are moved over from pattern lines to the wall and points are awarded.

Each player goes through his pattern lines from top to bottom and for each one that is full, he takes the first tile on the pattern line and places it on the same line on the wall on the space with the corresponding color. He is awarded points and the remaining tiles from the pattern line are moved to the lid.

Scoring
A player is awarded points whenever he places a tile on the wall. If the tile that is placed has no adjacent tiles, it is worth only one point. If the tile has either vertically or horizontally adjacent tiles on the wall, it is worth the number of tiles that are in line with it. If, however, it has both vertical and horizontal adjacent tiles, the tile is worth the size of the line it forms with the horizontally adjacent tiles plus the size of the row it forms with the vertically adjacent tiles.

Let's explain the scoring with an example. In 1.3 the player on the left will be placing a red tile on the second line of the wall. As the tile will form both a

Figure 1.2: Beginning of a round



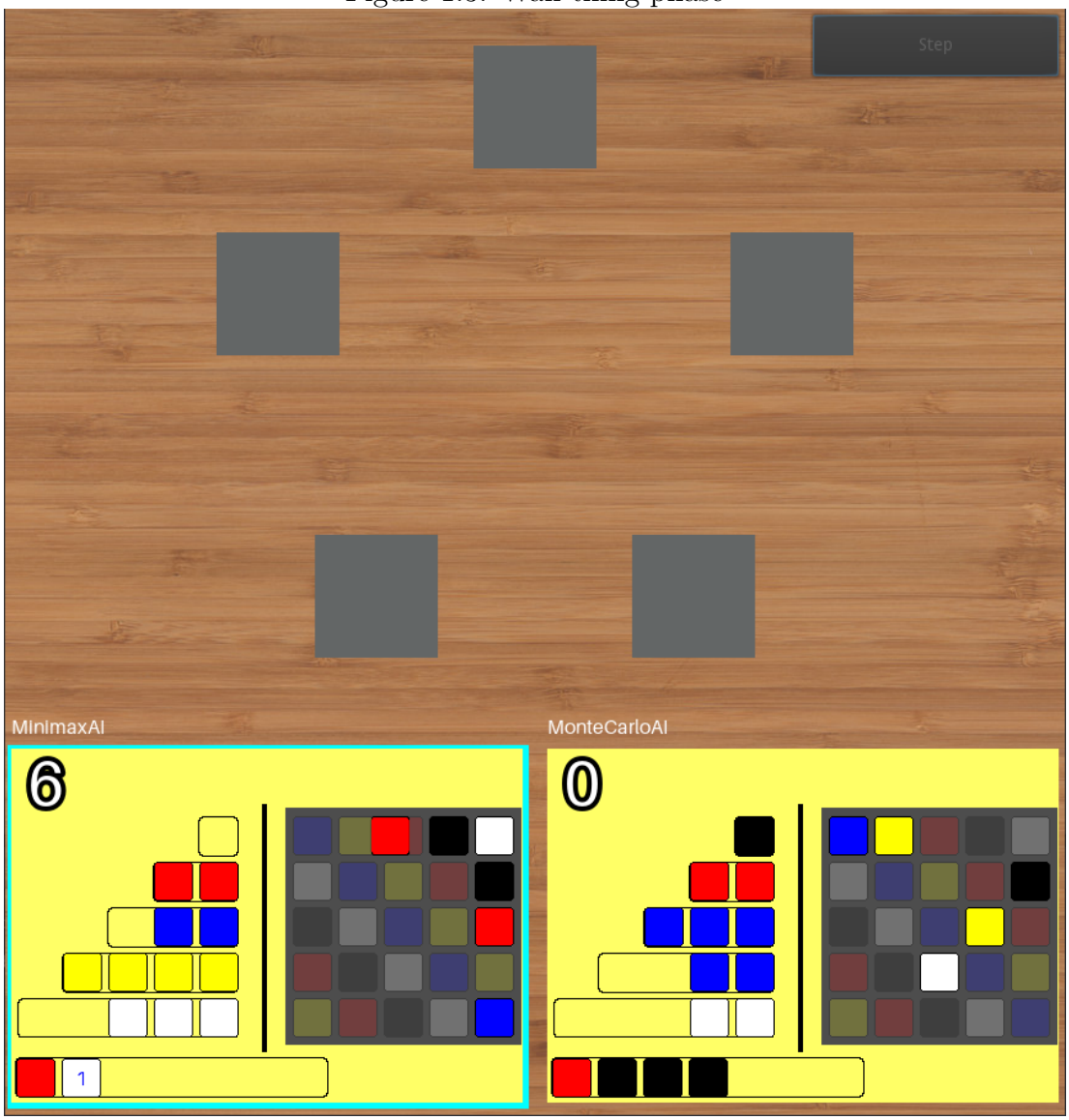
line of two and a row of two with the adjacent black tiles, it will receive 2 + 2 = 4 points for this one tile placed on the wall.

After resolving all the pattern lines, the player tallies all the penalty points he receives from the tiles he placed on the floor in that round. Each tile placed on the floor is worth a certain number of negative points. From left to right these numbers are -1, -1, -2, -2, -2, -3, -3. After adding penalty points from tiles on the floor, these tiles are all placed in the lid and the next round begins.

1.4 End of the game

The game ends when any of the players has all five tiles placed on any line of their wall. After the final wall tiling phase, bonus points for finished lines, rows and colors are awarded. For each line that the player was able to finish, they receive 2 points, for each finished row, they receive 7 points and for each color

Figure 1.3: Wall tiling phase



of which the player was able to place all five tiles on the wall they receive 10 points.

The winner is the player that has the higher number of points. If both players have the same number of points, the winner is the player with the highest number of complete horizontal lines.

2. Analysis

In this chapter we will focus on analysis of Azul from a game-theoretic perspective.

2.1 Classification

Combinatorial games can be classified according to several criteria.

- Zero-sum: Whether the reward to all players sums to zero (in the two-player case, whether players are in strict competition with each other).
- Information: Whether the state of the game is fully or partially observable to the players.
- Determinism: Whether chance factors play a part (also known as completeness, i.e. uncertainty over rewards).
- Sequential: Whether actions are applied sequentially or simultaneously.
- Discrete: Whether actions are discrete or applied in real-time.

Using these categories, we can classify Azul as a zero sum, perfect information, non-deterministic disrete game.

Zero sum is because there are no benefits to cooperation with other players and all players are in strict competition with each other.

We categorize it as perfect information, because there is no information that would be hidden from players and from the beginning to an end, all the boards of all players are available for observation.

The game is non-deterministic, as it involves a stochastic element in randomly setting up each round.

We categorize the game as sequential, since player are taking turns on all the moves they take.
Board games usually tend to be discrete. Azul is no exception to this rule and is discrete, unless an element of timer is introduced.

2.2 Complexity

Let's analyze how wide and deep can we expect Azul's search tree to be. At the beginning of the game There are 5 factories, each containing minimum 1 and maximum 4 different colors to choose from. Assuming it's a beginning of a game, each possible draw can be placed on any of the 5 pattern lines or to floor.
The largest number of possible moves will therefore be $5 \times 4 \times 6 = 120$ possible moves. The possible range for moves is between 1 and 120.

Through Testing the width of a tree throughout a game, we can draw some interesting observations. Here I will include an example of a game and how the width of a search tree changes throughout it

Table 2.1: Width of a search tree throughout a sample game

n	tw	sum(tw)	n	tw	sum(tw)
1	90	90	27	36	818
2	84	174	28	18	836
3	60	234	29	17	853
4	50	284	30	9	862
5	36	320	31	9	871
6	32	352	32	4	875
7	23	375	33	4	879
8	19	394	34	1	880
9	12	406	35	40	920
10	6	412	36	47	967
11	4	416	37	29	996
12	1	417	38	34	1030
13	46	463	39	19	1049
14	43	506	40	24	1073
15	29	535	41	8	1081
16	34	569	42	5	1086
17	22	591	43	3	1089
18	17	608	44	1	1090
19	10	618	45	32	1122
20	8	626	46	33	1155
21	5	631	47	22	1177
22	3	634	48	21	1198
23	1	635	49	16	1214
24	53	688	50	11	1225
25	56	744	51	11	1236
26	38	782	52	6	1242
			54	2	1249
			55	1	1250

From these data we can observe that the highest number of possible moves can be expected at the beginning of the first round. Then it decreases linearly until reaching 1. The average number of moves of this example game is 22.72.

3. Implementation

Technologies used:

- **Front end:** Godot engine
- **Back end:** C++

3.1 Front end

The front end was created in the Godot game engine. This technology was chosen as a simple multi-platform solution for creating simple games. It has a modern, simple to use GUI and using the new technology GDnative, the connection between the front end in Godot and the back end in C++ was relatively simple.

3.1.1 Graphics

The graphics for the GUI are a simple set of colored shapes created in Gimp. For game pieces combined with a preexisting library of functional blocks like buttons, drop down menus etc.

3.1.2 GUI functionality

The implementation of the GUI functionality such as board interactions for human players, game controlling and animations are achieved using GDnative, which is a module for the Godot engine which allows for execution of third-party code in the Godot environment. In our case, we used GDnative combined with a library cpp bindings pulled from Godot's public Github site.
Use of this technology was decided for two reasons:

1. It allows for an easy connection to the back end, which is also implemented in C++
2. C++ is a language with which the author of this thesis is best acquainted with

3.2 Back end

For the back end the language of choice is C++ for its speed and portability.

3.2.1 Structure

The entire back end is loosely based on the model-view-controller design pattern, from here referred to as MVC. The basic idea of MVC is to split the program logic into three main parts:

1. **Model** Represents the current state of a game and provides functions to manipulate the state

2. **Controller** Takes care of the game logic and interaction with the client

3. **View** Here we implemented two different views, one for a graphic user interface (GUI) and one for a console user interface (CUI)

3.2.2 Model

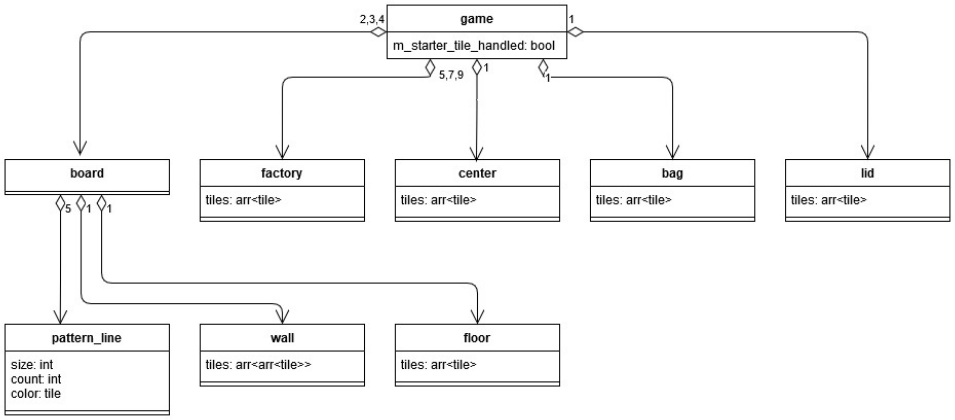


Figure 3.1:

The model is created with object-oriented programming in mind and it describes a current state of the game. Every class in the tree represents an actual object in a game, all of which are described in more detail in the chapter on Game Rules. Furthermore, classes in the model contain functions to manipulate the game state.

3.2.3 Controller

The controller keeps a reference to the model and performs operations on it. It also keeps track on these operations and if necessary can reverse them. To achieve this, it uses the Command design pattern. Every command represents an operation in the game.There are altogether eleven different commands. Four represent a player interaction with the game and are usually given to the controller by a view to perform. The remaining seven represents events that occur in a game without player interaction and are assigned by the controller itself when the game reaches an appropriate state (the end of a round or the end of a game).

Here is the name and a brief description of every one of these commands:

Player commands

factory_offer	Take tiles of a certain color from a factory and put them on a chosen pattern line
center_offer	Take tiles of a certain color from the center and put them on a chosen pattern line
drop_factory	Take tiles of a certain color from a factory and put them on the floor
drop_center	Take tiles of a certain color from the center and put them on the floor

Non-player commands

init_round	Initializes a new round (sets up factories from the bag)
score_wall_tile	Assigns score based on a tile position
tile_wall	Removes tiles from a pattern line, puts the first one on the appropriate position on the wall and the rest to the lid
tally_floor	Scores the tiles on the floor and moves them over to the lid
score_row	At the end of a game, assigns score for a row if it is fully filled up
score_column	At the end of a game, assigns score for a column if it's fully filled up
score_color	At the end of a game, assigns score for a color, if all tiles of that color are present on the board

3.2.4 View

There are two different viewing possibilities. The first one is a GUI created in Godot for either human players to play against each other, a human player to play against a computer, or to observe a game between two different AI players. The second one is a console user interface (CUI) used to test different artificial intelligence agents against each other and collect statistics of the game. It only provides the results and statistics of the games played.

In this section I'll briefly outline some basic implementation details for both of those. For a more detailed information about their information, see the source code.

GUI
A wrapper class for a model overrides all the functions that somehow change the visible state of a game and creates appropriate animations when these functions are called. This wrapper class is instantiated and passed to the controller, which the GUI view then uses to provide the client with interactions with the game.

CUI
The CUI can be given different options from the command line which it then parses and simulates an given number of games using a reference to a controller.

4. Artificial Intelligence

To develop an artificial intelligence agent that plays the game, I chose three different approaches for comparison:

1. Minimax
2. Monte Carlo Tree Search
3. Self-created strategy

In this chapter we will take a closer look at every one of these, their implementation, optimizations used and analysis of their success rate in the game.

4.1 Minimax

4.1.1 Algorithm description

A computer plays a turn-based game by looking at the actions available to it on this move and selecting one of them. In order to select one of the moves, it needs to know which moves are better than others. This knowledge is provided to the computer by the programmer using a heuristic called the static evaluation function.

4.1.2 Static evaluation function

A static evaluation function is arguably the most important part of the minimax algorithm.
Pseudocode for the evaluation function we used:

```
input: state
output: score
score = player.score()
foreach pl in state.pattern_lines
        if pl.is_empty() then continue
        frac = pl.count/pl.size
        if last_round() then frac = floor(frac)
        else score += player.has_starter_tile()
        score += frac * wall_tile_score(pl.index, pl.color)
foreach line in wall
        if line.is_full() score += 2
foreach row in wall
        if row.is_full() score += 7
foreach color in colors
        if wall.color_finished(color) score += 10
score += floor_penalty()
```

Here we calculate the score as a value of the current position. The parameters involved are the player's current score, score, the filled pattern lines will yield, a fraction of a score partially filled lines will yield and the score from finished lines,

rows and colors that will be awarded at the end of the game. An ownership of a starter tile is also accounted for as a half of a point.

In the last round, the partially filled pattern lines aren't worth any points, so the *floor()* function is used, result of which is that all non-fully filled pattern lines will be worth 0 points. Here the round is considered as "last" when in any line there are

4.1.3 Alpha-beta pruning

Alpha-beta pruning is an optimization of minimax that allows us to find branches that don't need to be examined anymore and prune them. This can happen when it is not profitable for a player to choose a move that leads into this branch.

4.1.4 Move ordering

Alpha-beta pruning can be made much more effective by examining the moves that are likely to yield higher score first. We can achieve this by a method called move ordering. Using a simple heuristic we can give every move a score and then examine these moves in descending order. The heuristic we used is basically a less time consuming portion of the evaluation function.
Here is its pseudocode:

```
input: move, state
output: score
if move is to floor
        score = 0
else
        pl = move.pattern_line()
        pl_count = state.pattern_line_count(pl)
        tile_count = move.tile_count
        overflow = pl_count + tile_count - pl - 1
        fraction = (pl_count + tile_count - overflow) / (pl + 1)
        penalty = state.floor_score(overflow)
        score = state.wall_tile_score(move.line(), move.color())
        score = score * fraction + penalty
        if score < 0 score = 0
```

4.1.5 Iterative deepening

To search move tree not only in depth in breadth, a method called iterative deepening is used.
Pseudocode:

```
depth = 1
while(time left)
        minimax to depth
        depth = depth + 1
```

Using this method we can be sure all moves were considered in as much depth as is possible the time we appointed for the algorithm to calculate a move.

4.1.6 Results

We can run different variations of MinimaxAI to see how they do against opponents with different parameters. We will use several different benchmark opponents for testing. First we will look at how minimax does against various opponents with an iterative deepening and a time limit of 500 ms per move

Table 4.1: Minimax with time 500 ms results against various opponents, sample size (games played) $n = 50$

Opponent	MinimaxAI time : 500 ms win rate
RandomAI	100%
GreedyAI	96%
MinimaxAI,depth : 3	67%
StrategyAI	92%
MonteCarloAI, iterations : 200	72%
MonteCarloAI, iterations : 500	19%

4.2 Monte Carlo tree search

4.2.1 Algorithm description

Monte-Carlo Tree Search (MCTS) is a best-first search method guided by the results of Monte-Carlo simulations. It is based on randomized exploration of the search space. Using the results of previous explorations, the method gradually builds up a game tree in memory and successively becomes better at accurately estimating the values of the most promising moves. MCTS has substantially advanced the state of the art in board games such as Go, Amazons, Hex, Chinese Checkers, Kriegspiel, and Lines of Action. MCTS pseudo code

```
Data: root
Result: bestMove
while(timeLeft()) do
currentNode <- root
/*the tree is traversed*/
while(currentNode in searchTree) do
lastNode <-currentNode
currentNode <- Select(currentNode)
end
/*a simulated game is played*/
r<-playOut(currentNode)
/*A node is added*/
lastNode<-Expand(lastNode, currentNode)
```

```
/*The result is backpropagated*/
currentNode <- lastNode
while(currentNode in searchTree) do
Backpropagation(currentNode, r)
currentNode <- currentNode.parent
return bestMove <-argmax of a in A(root)
```

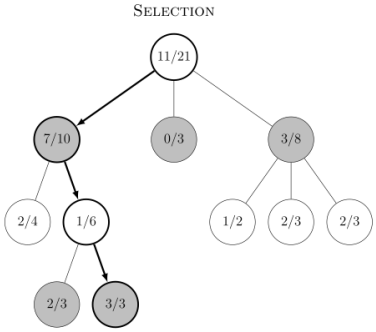
[Winands, 2017]

As we can see in the pseudocode, MonteCarloAI consists of four main parts: selection, expansion, simulation and backpropagation.

4.2.2 Selection

First, a node to expand is selected from already built tree, see 4.1. The selection is determined by a selection strategy UCT (upper confidence bounds to trees). It controls the balance between exploitation and exploration.

Figure 4.1: Selection [wik]



It applies following formula to calculate value of a node for selection, which is calculated as:

$$b \in argmax_(i \in I)(v_i + C \times \sqrt{\frac{\ln m_p}{m_i}})$$

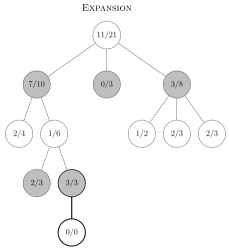
UCT selection strategy formula: [Winands, 2017]

where vi is the value of the node i, mi is the visit count of i, and mp is the visit count of p. C is a parameter constant, which can be tuned experimentally (e.g., C = 0.4). The value of vi should lie in the range [0, 1]. In case a child has not been visited yet, the maximum value that a node can have by sampling (i.e., vmax = 1) is assumed. The formula is applied recursively until an unknown position is reached.

4.2.3 Expansion

Afterwards an unknown node is selected randomly from all the possible moves available and added to a tree, see 4.2. Expansion is the procedure that decides whether nodes are added to the tree. The expansion strategy we

Figure 4.2: Expansion [wik]

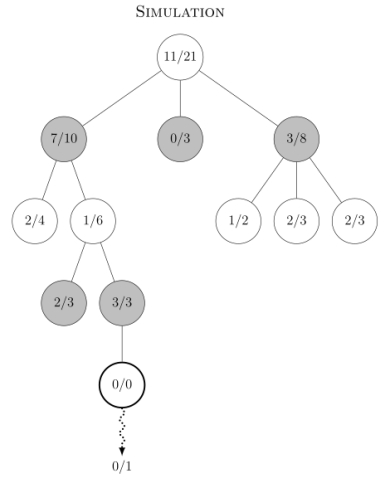


used was to add one node per simulation.

4.2.4 Simulation

To determine a value of the newly added node and to improve the value of its parents a random game is played. The game is scored one or zero points depending on whether the random game was won by current player, see 4.3.The simulation step begins when a position is entered that is not a part of the tree yet.

Figure 4.3: Simulation [wik]

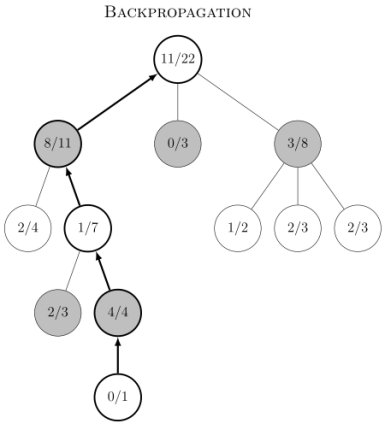


Moves are selected in self-play until the end of the game. This task might consist of playing plain random moves or – better – semi-random moves chosen according to a simulation strategy. Smart simulation strategies have the potential to improve the level of play significantly Gelly and Silver [2007]. The main idea is to play interesting moves based on heuristics. We achieve this by

4.2.5 Backpropagation

Backpropagation is the procedure that propagates the result r of a simulated game t back from the leaf node L, through the previously traversed nodes, all the way up to the root. If a game is won, the result of a player j is scored as a rt,j 1, in the case of a loss as rt,j 0, and a draw as rt,j 0.5. For a two-player game, propagating the values back in the tree can be performed similar to negamax [Knuth and Moore, 1975].

Figure 4.4: Backpropagation [wik]



4.2.6 Final Move Selection

The four steps are repeated either a fixed number of simulations or until time runs out. After the search is finished, one of the children of the root is selected as the best move to be played in the actual game. There are several final move selection strategies that determine the best child [G.M.J.-B. Chaslot, 2008].

1. Max child. The max child is the child that has the highest value.
2. Robust child. The robust child is the child with the highest visit count.
3. Robust-max child. The robust-max child is the child with both the highest visit count and the highest value. If there is no robust-max child at the moment, more simulations are played until a robust-max child is obtained.
4. Secure child. The secure child is the child that maximizes a lower confidence bound, i.e., which maximizes the quantity $v + \frac{A}{\sqrt{m}}$, where A is a parameter (e.g., 4), v is the node's value, and m is the node's visit count.

In practice, the difference in performance between these strategies is limited when a sufficient number of simulations for each root move have been played. In case there is a certain amount of time for the whole game and the player manages its own time for each move, the robust-max child is the most promising.

The version we used in our implementation was a robust child, where the child with the highest amount of visits is selected as a move.

4.2.7 Results

We measured the capability of MonteCarloAI against a MinimaxAI with a set search tree depth of three, which we chose as a benchmark AI for this test. We tested different amount of iterations and for each one, our testing sample size was $n = 100$:4.2)

4.3 Strategy

Strategy child is an AI based on a strategy developed through experience with game.

4.3.1 Strategy description

We decide which move we are gonna choose based on how this move can cooperate with other possible moves. Moves are first grouped by which tile on the wall they will contribute to filling. These moves are then sorted by what percentage of their pattern line they can fill.

For example, if all the moves that move red tiles to pattern line 1 can together move 4 tiles to the pattern line, it means, the value of all these moves together

Table 4.2: Monte carlo success rate against minimax depth 3 out of 100 games

Iterations	Win rate
100	37
150	38
200	49
250	58
300	71
350	61
400	68
450	81
500	81

is 4.0 / 1 = 4.0 and let's say all the moves of blue tiles to pattern line 5 can together fill only four out of five tiles, they are assigned a value 4.0 / 5 = 0.8. Finally, from all the moves that fill the pattern line with the same color tiles and can fill their respective pattern line to the highest percentage we choose the one that has the biggest line fill, but the smallest overflow.

4.3.2 Reasoning behind algorithm

Here we will outline what led to certain decisions made in the algorithm development.

- **Grouping moves:** The moves are grouped by the tile on the wall they will effect. This way, we can first choose the color we are most likely to finish for every pattern line.
- **First sorting:** The groups of moves are sorted by the portion of pattern line, they are together capable of filling. This way we hope to choose moves that are most likely to eventually fill the pattern line.

  As it is possible for opponent to steal moves that seem profitable, we choose the moves that have plenty enough profitable moves to render our opponent as unlikely as possible to interrupt our filling of a chosen pattern line.
- **Second sorting:** From the group we then choose the most profitable move. The assessment of which move is the most profitable consists of considering two main criteria, the first one being a portion of the pattern line that we fill, this assigns every move a value between zero and one which we try to maximize to gain points and the overflow, which we try to

minimize. We first sort the moves by the first criterion, then by the second one.

4.3.3 Pseudocode

Here is a strategyAI pseudocode for picking a best move

```
input: model
output:move
moves = get_moves()
grouped_moves = group_moves(moves)
sorted_groups = sort_gropus(grouped_moves)
move = sorted_groups[0].pick_move()
```

4.4 Results

The idea of a man-made strategy proved quite successful in the early stages of development, considering how simple the algorithm and idea behind it. In later phases of development the results of strategyAI playing against various opponents at the point of writing this thesis are as follows:

As we can see, strategyAI falls short of more developed AI variants. How-

Table 4.3: StrategyAI results against various opponents, sample size (games played) $n = 50$

Opponent	StrategyAI win rate
RandomAI	100%
GreedyAI	18%
MinimaxAI,depth : 3	16%
MinimaxAI,time : 500	8%
MonteCarloAI, iterations : 200	4%
MonteCarloAI, iterations : 500	2%

ever, it lead me to some good ideas on quick move developments and could serve as an interesting strategy to try and use and keep in mind. Despite its seemingly low success rate against more advanced opponents, I believe this method of artificial intelligence will at least pose a moderate challenge to a beginning human player.

4.4.1 Further development

In the future, StrategyAI could be improved in various ways. For example taking a score different tiles will acquire during wall tiling phase into consideration. Another improvement of strategy I had on my mind, but didn't manage to implement into strategyAI itself was using the amount of free neighboring tiles (2 for corner pieces, 3 for side pieces, 4 for center pieces) as a criterion when picking initial tiles.

4.5 Other

In this section I'll lightly touch on two other artificial intelligence agents, which were implemented and were mainly used as a benchmark to compare effectiveness of the three main ones and as debugging tool.

4.5.1 RandomAI

This chooses moves randomly, while not really a formidable opponent, it served an important debugging purposes in early stages of development.

4.5.2 GreedyAI

The algorithm used to decide move for greedyAI can be generally described as a minimax with a depth of one.

Conclusion

In this thesis we created a working simulation of a board game Azul. We also created different artificial intelligences using well known algorithms, such as MonteCarlo and minimax. We created a console user interface for testing purposes and tested how the artificial intelligences performed against each other.

Bibliography

Monte Carlo Tree Search phases. URL https://commons.wikimedia.org/wiki/File:MCTS-steps.svg#/media/File:MCTS-steps.svg.

S. Gelly and D. Silver. Combining online and offline knowledge in uct. *Proceedings of the International Conference on Machine Learning (ICML)*, page 273–280, 2007.

H.J. van den Herik J.W.H.M. Uiterwijk B. Bouzy G.M.J.-B. Chaslot, M.H.M. Winands. Progressive strategies for monte-carlo tree search. *New Math. Nat. Comput.*, pages 343–357, 2008.

D.E. Knuth and R.W Moore. An analysis of alpha-beta pruning. pages 293–326, 1975.

Azul Rulebook. PlanBGames, 2018. URL https://www.planbgames.com/en/index.php?controller=attachment&id_attachment=9.

Mark H. M. Winands. Monte-carlo tree search in board games. pages 51, 54, 2017.

List of Figures

List of Tables

List of Abbreviations

A. Attachments

A.1 Contents of the zip file:

- /bin: binary files and executables
- /text/text.pdf: thesis text
- /source: source code
- /source/doc/Developer documentation.odt: programmer documentation
- /source/doc/User guide.odt: user documentation