KTH Computer Science
and Communication

Crushing Candy Crush

Predicting Human Success Rate in Candy Crush Saga using Monte-Carlo Tree Search

ERIK RAGNAR POROMAA

Master's Thesis at NADA
Supervisor: Jens Lagergren
Examiner: Olov Engwald

Abstract

The purpose of this thesis is to evaluate the possibility of predicting difficulty, measured in average human success rate (AHSR), across game levels of King's Candy Crush Saga (Candy) using Monte Carlo Tree Search (MCTS). King is very interested in agents using Artificial Intelligence (AI) that can play their games for both quality assurance and development purposes. We implemented and tested a simulation based bot using MCTS for Candy. Our results indicate that AHSR can be predicted accurately using MCTS, which in turn suggests that our bot could be used to streamline game level development at King. Our work is relevant to the field of AI, especially the subfields of MCTS and single-player stochastic games as Candy, with its diverse set of features, proved an excellent new challenge for testing the general capabilities of MCTS. The results will also be valuable to companies interested in using AI for automatic testing of software.

Referat

Förutse mänsklig framgång i Candy Crush Saga med Monte-Carlo trädsökning

Syftet med denna uppsats har varit att evaluera möjligheterna att med Monte-Carlo trädsökning (MCTS) förutse svårighetsgrad, mätt som antalet vinster kontra förluster för människor, för spelnivåer i Kings Candy Crush Saga (Candy). King är väldigt intresserade av program användandes av Artificiell intelligens (AI) som kan spela deras spel, dels för kvalitetssäkring samt utvecklingssyften. Vi implementerade och testade en simuleringsbaserad MCTS bot för Candy. Våra resulat visade att vår bot kunde med god noggranhet förutse svårighetsgrad i Candy. Våra resultat indikerar att mänskling svårighetsgrad kan förutses med hjälp av MCTS, vilket tyder på att vår bot kan anändas av King för att effektivisera utvecklingen av nya spelnivåer. Vårt arbete är relvant för AI fältet, speciellt delfälten MCTS och stokastiska spel för en spelare då Candy, med sitt breda utbud av funktioner, var en perfekt utmaning för att testa de generella egenskaperna av MCTS. Resultaten är även värdefull för företag som är intresserade av automatiserad mjukvarutestning genom användning av AI.

Contents

Introduction

Artificial Intelligence (AI) has been developing since World War Two, making it one of the newest fields of Science and Engineering. Today AI is making itself more present in our daily lives as AI applications now are embedded in all industries. This development has inspired many industries and companies to consider the potential benefits from using AI. Throughout the development of AI, games have been used as test environments and several groundbreaking feats have been accomplished in this subfield of AI. In 1997 IBMs Deep Blue beat Garry Kasparov in Chess and in 2016 Deep Mind's AlphaGo beat Lee Sedol in the game of Go [1].

This progress has led to King being interested in the use of AI. King, as a game developer, could potentially use bots—AI agents—that can play their games to improve quality assurance and development of their games, hopefully resulting in a better experience for their players. The purpose of this thesis is to investigate opportunities to apply AI in game development at King by answering the following question:

- Is it possible, using Monte-Carlo Tree Search (MCTS), to predict average human success rate (AHSR) of levels in Candy Crush Saga?

As King produces several different games we will strive towards using methods that will be general and applicable to different games.

1.1 Scope & Outline

The thesis work involves positioning Candy Crush Saga (Candy) in the field of AI and implementing a bot that playes Candy using knowledge gained from MCTS and that potentially could be used on other games as well. We chose only to investigate a subset of all possible MCTS setups. We also investigated state-of-the-art methods for predicting AHSR in Candy at King. No other games than Candy were investigated. Methods and other areas of interest that we considered to be relevant but are outside the scope of this thesis are presented in the Future Work section.

This report is organized as follows. Chapter 2 contains the background and describes the field of AI, our test domain Candy, our positioning of Candy in the AI field, the field of General Game Playing and King's current state-of-the-art methods for predicting AHSR. Chapter 3 describes MCTS, why and how we implemented our MCTS bot on Candy. How we decided to test the performance of our bot in respect to answering our research question is also described. After the method chapter we present our results. Finally, the report is concluded with a discussion about the results and possible future directions.

Background

This chapter first describes the field of AI, games, Candy and how to solve games. Secondly subfields of AI that were relevant to our work of creating a bot that could be used to predict success rate in Candy and potentially other games are described. Lastly, King's current state-of-the-art methods for predicting AHSR are presented.

2.1 Artificial Intelligence

We call ourselves *Homo sapiens*—man the wise—because our **intelligence** is so important to us. For thousands of years, we have tried to understand *how we think*; that is: how a mere handful of matter can perceive, understand, predict and manipulate a world far larger and more complicated that itself. The field of **artificial intelligence**, or AI, goes further still: it attempts not just to understand but also to *build* intelligent entities.

Stuart J. Russell and Peter Norvig, *Artificial Intelligence A Modern Approach*

AI has several definitions, some consider thought process and reasoning, while others consider behavior. One of the most popular definitions of AI is the one of *Acting humanly* which stems from the Turing test [2]. Another definition of artificial intelligence is *Thinking humanly.* As our goal with this thesis was to predict human success rate in Candy Thinking humanly is an aspect of AI which we argued was relevant to us. In order to state that an agent is *Thinking humanly* we need to be able determine how humans think. Suggested ways of doing this are; introspection, psychological experiments and brain imaging. Using information gathered from these tests an agent could potentially be created. If the created agent has a similar input-output behaviour as a human it is evidence that there are similarities in the mechanisms between the agent and the human [1]. We did not investigate the cognitive aspects, of which this definition relates to, of AI in this thesis. However, as our goal was to predict human success rate, we wanted to mimic human thinking and therefore this definition of AI was relevant to us.

The AI field is composed of several subfields. These subfields range from general learning and perception, to specific subfields such as playing games, driving vehicles, and diagnosing diseases. The field of AI is universal, however, all subfields have in common that they contain an intellectual task being solved [1].

The amount of possible AI implementations have increased due to the increase in available data-sources since the early 2000s and increase in computing power [1]. We believe that a driving factor of the AI field stems from its ability to provide better and more cost-effective solutions to problems that previously have been solved by humans. Thus, we argue that AI can have a substantial impact on companies competitive advantage.

Table 2.1: Candy in context of other game types regarding determinism and number of players.

	Single player	**Two Player**	**Multi-Player**
Deterministic	TSP, PMP	Go, Chess, Checkers	Chinese checkers
Stochastic	Sailing problem, Candy Crush Saga	Backgammon	Simplified Catan

Since the start of the AI field, games have acted as a test environment for AI . After the era of competing against Chess Grandmasters, the game of Go emerged as the new challenge for AI development. The game of Go proved extremely difficult to master with the traditional game-tree search algorithm used in chess. Knowledge gained from the previous battle against Chess proved to be inefficient as even human amateurs beat the AI agents created. In 2006 a completely new search method changed the landscape for the Go agents [3]. This was Monte-Carlo Tree Search (MCTS) [4], described in detail in Subsection 2.6. Ten years after MCTS emerged Deep Mind's AlphaGo beat Lee Sedol 4-1 in the game of Go using a combination of Deep Neural Networks and MCTS [5].

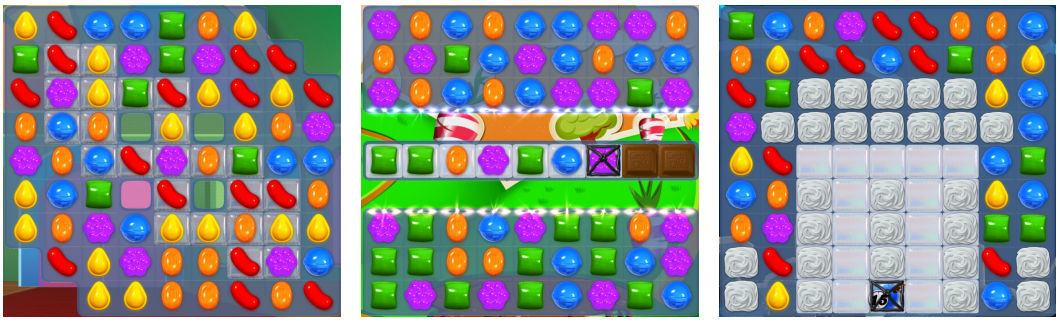
2.2 Game Types

Games are a subset of problems tested with AI. We sometimes refer to games as problems. There are several types of games; games can be deterministic, non-deterministic and/or have hidden information, however we give less attention to hidden information in this thesis. We also categorize games as follows: games without opponents (called single-player games, puzzles or optimization games), games with one opponent (two-player games) and games with several opponents (multi-player games), see Table 2.1 for the different game types.

Deterministic games have no element of chance, there is one state given each action, for example the game of Tic-Tac-Toe. In non-deterministic games a certain action does not correspond to a certain state. For example, in Candy the player needs to match candies. Matching candies removes them from the game board and creates space for new candies. The color of the new candies are chosen randomly from a set of colors specific to the level, each possible game board, containing a new set of candies, is a separate state. Hidden information in games can for example be the hidden cards of your opponent in Poker. Examples of deterministic one-player problems that have been subjects of AI tests are: Traveling Salesman Problem (TSP), Leftmost Path and Left Move Problems, Morpion Solitaire, Crossword construction, SameGame and Sudoku [6, 7, 8, 9]. Some stochastic games that have been tested are: Skat, Poker, and Backgammon [10, 11, 12]. Problems that share both the property of being stochastic and single player are fewer. Two problems that have been studied in this domain are the Sailing problem and Klondike Solitaire [13, 14].

Candy is a single-player non-deterministic game[1] and its placement in context to other games is shown in Table 2.1.

[1]Candy can have elements of hidden information on occasional levels.



(a) **(b)** **(c)**

Figure 2.1: Inital state of levels 13 (a), 77 (b) and 100 (c) in Candy. Shows diversity between levels of Candy, each levels features are described in detail in table 3.1

2.3 Candy Crush Saga

Candy Crush Saga (Candy) was released in 2012 by King. Candy is a single-player stochastic game with occasional hidden elements. The game board in Candy can contain up to $9x9$ positions. Every position on the game board contains a candy. Each level in Candy can have different numbers of available moves, different amounts of jelly, different blockers and other features, see Figure 2.1. The player's task in Candy is to complete levels. In order to complete a level the player needs to complete the level's objective. There are six different types of levels: *Moves levels*, *Jelly levels*, *Ingredients levels*, *Timed levels*, *Candy order levels* and *Mixed levels*. The objective for *Moves levels* is to obtain a certain score by using a predetermined amount of moves. The player's score on the level is the accumulated score of the individual scores for each of the moves the player makes. The objective for *Jelly levels* is to remove all the jelly on the game board, which is done by removing candies on top of the jelly. The objective for *Ingredient levels* is to move the ingredients on the board down to certain positions on the game board. In *Candy order levels* the objective is to remove certain types of candies. *Mixed levels* have more than one objective. Possible moves a player can make are matching three or more candies in different ways, when matching more than three candies special candies appear, matching four candies in a column or a row creates a Striped candy, matching five candies in a column or a row creates a Color bomb. The special candies are more powerful than regular candies; when matched with other candies or other special candies they have a greater effect on the game board. After removing candies on the game board new candies appear, the new candies presented are chosen randomly from a set of available candies for the level. Levels can also contain blockers. Blockers prevent the player from reaching candies, different blockers are: *Icing*, *Licorice lock*, *Chocolate* and many more. An attempt on a level is finished when the end state is reached. On all levels except *Timed levels* an end state is reached after a number of moves are expended. End state can also be reached prior to expending your available moves if the players either fails to remove elements such as Bombs from the board, see bottom of level 100 in Figure 2.1, or finishes the level's objective [2]. We believed that the diverse set of challenges and features that Candy presents provides a great challenge for any AI agent. In terms of complexity, Candy has been evaluated as non-deterministic polynomial-time hard (NP-hard) by previous research [15].

[2]For more information on Candy visit: `http://candycrush.wikia.com/wiki/Candy_Crush_Saga_Wiki`

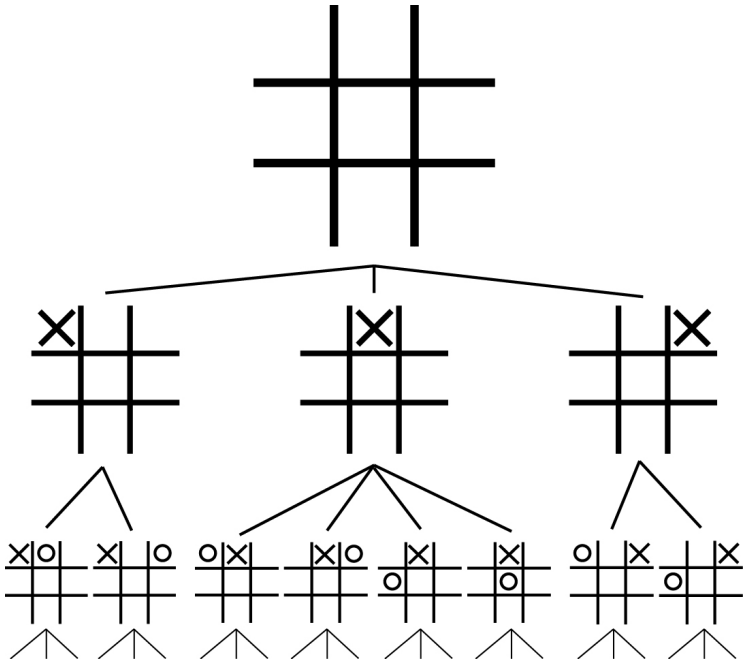


Figure 2.2: Overview of a search tree for Tic-Tac-Toe starting from initial state.

2.4 Solving Problems / Games

The state space for a game can be represented by a directed graph, as several paths might lead to the same state. However, when solving games, the state space is traversed using a tree structure. For example, in the game of Tic-Tac-Toe each state will be a different game board, see Figure 2.2. Problems where states can be represented by nodes of a tree and the possible actions can be represented by the edges of the tree are called tree search problems. In Tic-Tac-Toe the initial state would be an empty board and a goal state for either of the two players would be a board with their marks in a horizontal, vertical, or diagonal row. For every possibly state a player can do different moves, called actions. In order to determine the best possible action a player could try to estimate what sequence or path of actions leads to a win, a goal state. Figure 2.2 shows the first levels of a search tree for Tic-Tac-Toe. By exhaustively expanding a tree structure it is possible to determine the best possible action in every state of Tic-Tac-Toe. However, exhaustively expanding a tree structure is not possible in the vast majority of problems. The process of looking for a sequence that leads to a goals state is called a search [1].

All games of perfect information have an optimal value function $v^*(s)$. Given a certain state an optimal value function determines the outcome of the game under perfect play by all players. For example in Tic-Tac-Toe the optimal value function would allow the player taking the first action to know what action is most likely to lead to a win. For single-player games the optimal value function considers the chance of succeeding against the null opponent or the puzzle. Games of perfect information can be solved by recursively computing the optimal value function in a tree containing approximately $breadth^{depth}$ possible different paths of moves, where *breadth* is the average number of actions (and their respective states if it is a non-deterministic game) for each state in each path, and *depth* is the average number of states from the initial state to the end state of each sequence. For example, Chess

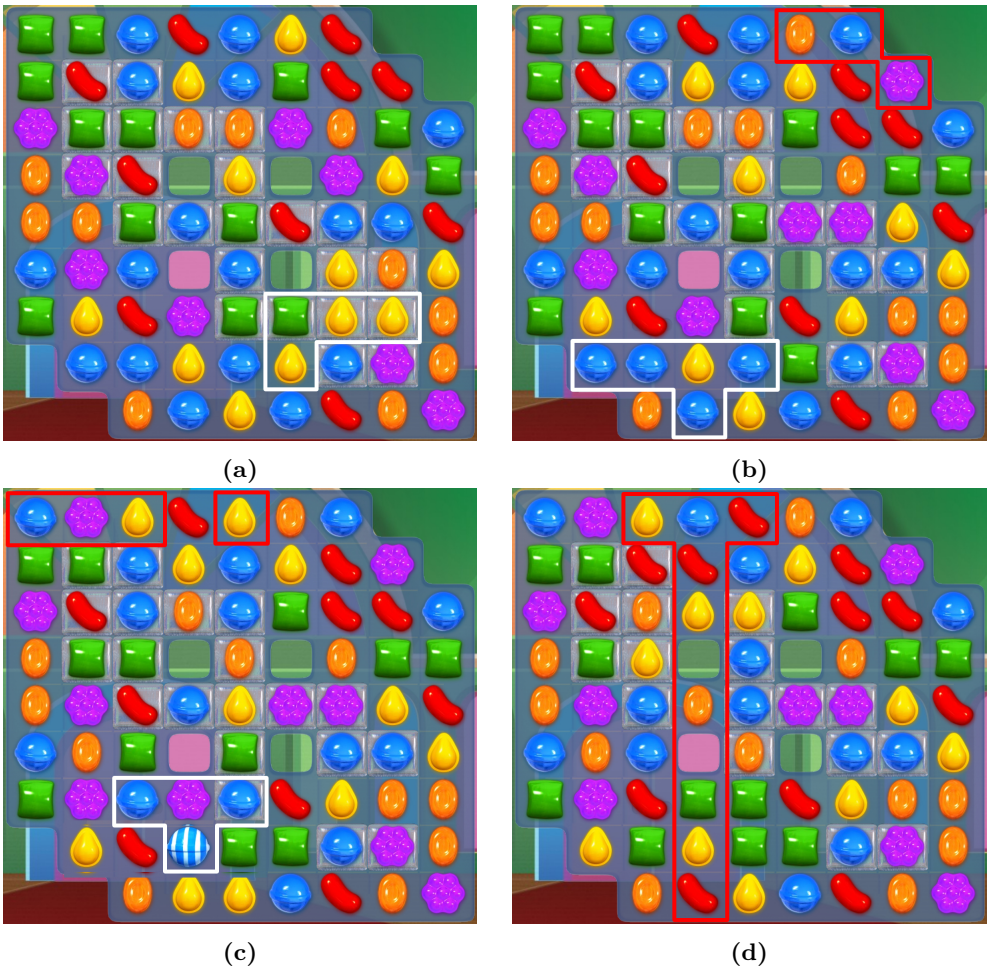


Figure 2.3: Area in white shows the chosen action for the state, area in red shows candies new to state. Starting from upper left corner (a) the first action is a three match from the initial state in level 13. The upper right plot (b) shows the succeeding state with it's new candies in red. Our next action is a four match. The lower left plot (c) shows that our four match resulted in four new candies, shown in red, and we created a striped candy. After matching the striped candy, bottom right plot (d), we see that the entire column has be changed as well as one candy on each of the two the adjacent columns.

has approximately a *breath* of 35 and a *depth* of 80, Go has approximately *breadth* of 250 and *depth* of 150, making an exhaustive search infeasible for both games. With exhaustive search infeasible two general principles can be used to reduce state space. First, the *depth* of the search tree can be reduced by replacing subtrees in the search by a value given from an approximate value function $v(s) \approx v^*(s)$. Second, the *breadth* of a search tree can be limited by not expanding all actions for each state using some policy$(a|s)$. A policy could be for example to only consider the ten most promising actions for each state, the value of the moves could be determined by an heuristic [5]. Candy has variable *breadth* and *depth*, depending on the level being played. An approximation of state space for three different levels is presented in the Results Chapter.

When trying to solve stochastic games using search trees it is not sufficient to use only nodes to represent states and their edges to represent actions. Without a random element

between states, each action will lead to a specific state, which is not the case in stochastic games. For example, after making the first move in level 13 of Candy a new state is given, this state has three new candies, see Figure 2.3. The colors of these new candies are chosen randomly, i.e the state succeeding the action is in part determined randomly. Similarly this continues through the following moves, see Figure 2.3. Despite this, a stochastic game can be transformed into a deterministic game by the act of determinization. Determinization can be done by sampling several deterministic instances in order to gain information about an action in a stochastic game. For example, the different combination of colored candies that can appear after making the three match, Figure 2.3, corresponds to several states. Sampling several of these different states will combined give an expected value of the action. Nonetheless, due to the large number of possible states, given the random element when searching in a stochastic domain, and the inability to chose what state to end up in after choosing an action, an agent will rarely play one of the sequences that it previously searched. In other words neither of the states the agent created after making the three match, Figure 2.3, while searching might be the actual state that an agent will play after finishing the search even if an agent chooses that action [16, 17]. An example of determinization can be seen in our bot implemented on Candy, described in the section 3.3.2.

In order to solve games or find the best action given a certain state several different search algorithms have been developed. A* is a traditional search algorithm for single-player deterministic games [18]. For two-player deterministic games a common algorithm was minimax [19], which later evolved into alpha-beta pruning [20]. The majority of algorithms were developed for the field of two player deterministic games and most of them depend on a state evaluation function [21].

Alpha-beta pruning has been the standard in two-player games for decades. However, in the game of Go, it was difficult to evaluate a certain state. Due to this, as of 2006, the best Go agents were a combination of alpha-beta search, expert systems, heuristics and patterns. In 2006 a new algorithm emerged. It emerged from substituting the state evaluation function in alpha-beta with Monte-Carlo simulations, referred to as playouts or rollouts [3]. Eventually, the Monte-Carlo Tree Search (MCTS) [4] was developed, described in detail in subsection 2.6.

2.5 General Game Playing

General Game Playing (GGP) is a subfield of AI in which the objective is to create AI agents that can successfully play not only one but several different games without human intervention [3]. The field of GGP is interesting to our thesis work as one of the objectives of our thesis is to, by the use of general algorithms, be able to use bot we created on several of King's different games.

For instance, Deep Blue could play chess on a world-champion level but has no idea how to play checkers and is therefore a terrible GGP agent. Improvements of AI agents performance in certain games have therefore been due to manual human input and not improvements in general AI. The GGP field tries to prevent this by creating an environment where agents require improvement in general knowledge handling in order to enhance performance rather than game specific improvements. This is achieved by letting agents play several different games and by not letting the agents know the game rules before the start of the games. Thus, the goal in GGP is to create intelligent agents that can successfully play different games using only knowledge of the game rules. In order for agents to play successfully they need to be able to learn game-specific strategies without any input from their developers [3].

In the field of GGP, MCTS has been used with great results and is therefor interesting to us. In turn, this is due to the MCTS algorithm's ability to perform well without external knowledge. However, having access to extra knowledge can greatly improve the performance. Gaining this additional knowledge is a big part of the field of GGP [3, 22]. Recent advances in the field of GGP are presented in [22].

2.6 Monte-Carlo Tree Search

This section briefly describes MCTS, where it has been successful and its strengths and weaknesses. A full algorithm description is presented in the Method chapter.

MCTS is a type of a tree search algorithm. Unlike traditional tree search algorithms such as alpha-beta pruning, MCTS does not need a heuristic or an approximate value function. Instead MCTS relies on Monte-Carlo playouts, playing randomly from a state until an end state is reached. The Monte-Carlo playouts enable MCTS to be more general than other tree search methods. In essence MCTS works by running several simulations of the games [21]. For example, when applying MCTS to the game of Tic-Tac-Toe the MCTS agent can simulate randomly to the end state after taking each action available from the initial state. Doing this once per action would provide the agent with an estimation, of low accuracy, of each actions expected value. By repeated simulations, the accuracy of the expected value is improved for each action. When the search end criteria is met, a final action is selected based on the best, for example, win ratio.

MCTS outperformed all classical techniques when first applied to the game of Go [5]. However, in other games such as chess, where it is possible to create a strong approximate state evaluation function MCTS was completely outperformed by alpha-beta algorithms [3]. During the last years the MCTS algorithm has evolved and become the focus of much AI research. Researcher are now refining the algorithm and investigating in what fields the MCTS succeeds [3, 21, 16].

Some of the strengths of MCTS are that effective game play can be achieved without any knowledge other than the rules of the game. MCTS is an anytime algorithm, meaning that all gathered information in the search can be used if search is terminated prematurely. The MCTS has a forward sampling approach which mimics human thinking, the algorithm will focus on the more beneficial paths of play while sometimes checking weaker alternatives. Some of the weaknesses of MCTS are that it might need a large amount of simulations to perform adequately, which can be hard to achieve if simulations are hardware demanding. MCTS requires allot of simulations to identify optimistic moves, moves in near future that seem to be good in the but actually are bad in the distant future, from good moves [23]. Also, if enhancements are made to the algorithm, similarly to other tree-search algorithms, no methods to study their influence on performance is available today except empirical testing. Similarly it is hard to predict the impact of changing parameters of the MCTS [16].

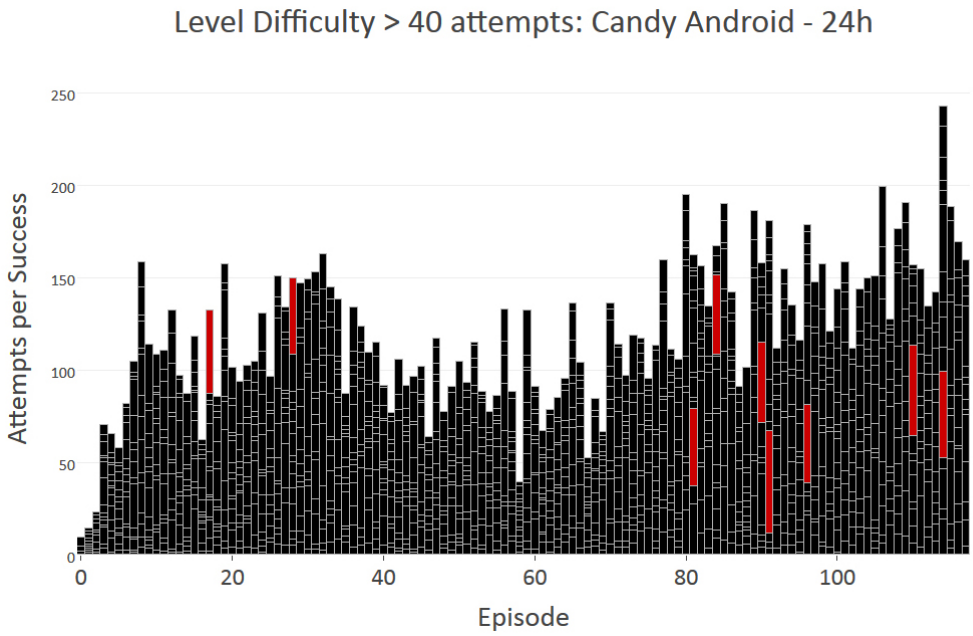


Figure 2.4: Overview of episodes in Candy. For each episode levels are stacked where the size of a stack shows average number of attempts accumulated for each episode. Levels with more than 40 attempts per success on average are shown in red.

2.7 State-Of-The-Art Methods for Predicting Success Rate

In order to get an estimation of difficulty for different levels in Candy King tracks the performance of players on all levels, see Figure 2.4. Each level's difficulty is measured in attempts per success or success rate. Throughout our tests we used the measurement of success rate. The average success rate of all tracked players on a level in Candy was what we compared our predictions with. This average success rate will be referred to as *average human success rate* (AHSR). The success rate of levels has a great impact on the player experience and keeping this success rate on moderate levels is therefore of importance to King. King would benefit from having an understanding of this difficulty prior to releasing new content. In order to get this understanding King needs to predict the difficulty. Current state-of-the-art methods for predicting the difficulty of new levels at King are:

- Handmade heuristic - Let a handmade heuristic bot play every level several times. The handmade heuristic, however, requires development as the game develops. The handmade heuristic is also useless in other games than the game it was created for.
- Play testers - A company has human testers playing levels and record their success rate. These human testers will be refereed to as *Play testers*.

Performance of these methods are presented in the results section.

Method

This chapter present why we choose to use MCTS. It also describes the MCTS algorithm, our implementation of an MCTS based bot for Candy, how our bot was affected by the Candy source code, how we implemented a non-deterministic search, non-binary signals (taking into account more than playout being wins or losses), guided playouts (playout guided by some type of heuristic rather than being random) and what tweaks to the MCTS we leave outside the scope of this project. Lastly we describe how we tested our bot.

3.1 Why MCTS

We chose to use MCTS for three reasons. First, King produces a diverse variety of games, of which most use different rules and logic and where game features might change over time. MCTS works without game specific knowledge and has been a successful algorithm in the field of GGP [16, 22]. Using MCTS as the underlying algorithm for our bot would hopefully enable us to use the same bot on several of King's games. Secondly, MCTS's forward sampling approach mimics human thinking and we reasoned that a bot using MCTS might predict human success rate accurately [16]. Third MCTS seemed a reasonable alternative due to the recent successes of MCTS in the game of Go (AlphaGo and the previous best Go programs were based on MCTS) [5]. Comparing Candy and Go, we believed that they were similar as they both share the difficulty of creating a state evaluation function [21]. For example an assumption in Candy could be that actions that create very powerful candies tend to be more likely to lead to a win. This is partly true. Having said that due to the large variety of game objectives and level specific details, it is not always the case that actions containing powerful candies lead to wins. For example: even if a player is in a state where he can create a Color bomb, a very powerful candy, the player could have blockers or some other objects that will prevent the player from completing the level, rendering the Color bomb useless in the context.

3.2 MCTS Algorithm

This section describes the an ordinary MCTS algorithm and then compares it to the MCTS algorithm we used.

To identify the best action in a given state the MCTS algorithm performs the following: Several simulations take place, with every simulation the search tree grows. The creation of the search tree ends by either limiting the search time or the amount of simulations. Figure 3.1 shows the procedure of running one simulation. Each simulation consists of four steps, *selection*, *expansion*, *playout* and *back-propagation*. When the search is ended a final move is selected using some policy [4, 13].

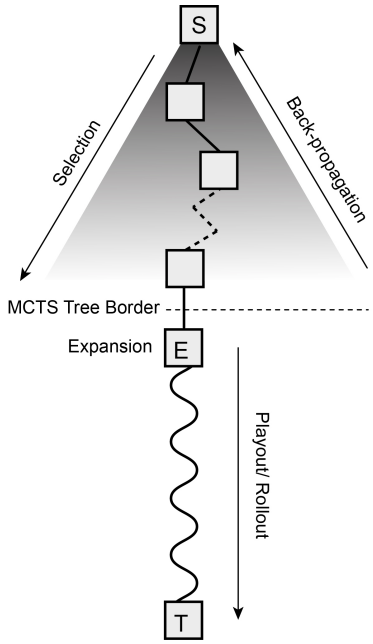


Figure 3.1: Overview of single simulation in an ordinary MCTS. Each simulation starts in the initial state (S) and ends in the end state (T). The search tree is represented by the gray shaded area. Selection in the search tree is done starting from the root of the search tree and moving downwards. The dotted line between the nodes in the search tree represents the possibility of additional nodes. Expansion takes place outside the border of the search tree. A playout continues until an end state is found. Back-propagation is done within the search tree, using the information gained from the playout, in the opposite direction of the selection.

Selection

Selection starts from the root node. A *selection strategy* is applied recursively until a state is reached outside the search tree. The selection strategy can either focus on exploitation or exploration, focusing on the best move so far or further trying less promising moves due to the uncertainty of the evaluation [21]. Several selection strategies have been developed, see [16]. Selection is often done using Upper confidence bound (UCB) [13]. When using UCB as a selection strategy the MCTS is called Upper Confidence Bounds for Trees (UCT). UCT is by far the most widely used MCTS algorithm. The UCB formula is presented below:

$$UCB = \bar{X_j} + C\sqrt{\frac{\ln(n_p)}{n_j}} \tag{3.1}$$

Where $\bar{X_j}$ is the value, often calculated as $wins/visits$ of a node j, n_j is the number of visits to node j, n_p is the number of visits to the parent node of j and C is a constant. In essence the UCB formula controls how much information is needed from one node before disregarding other nodes. A very small C-value would make the move selection greedy, searching only nodes with the highest expected value. A large C-value would make the selection less greedy, needing more visits to each nodes before disregarding nodes. Being less greedy can be beneficial as only focusing on the best possible move could lead to focusing resources towards non-optimal subtrees of the search tree. For example, sometimes taking out the queen in chess, a great move, can be succeeded by sacrificing less important pieces,

a path of weak moves. Without the UCB formula's second term these types of moves will never be explored. It has been proved that UCT converges to the best action of a given state provided sufficient number of simulations [13, 16]. Visualisations of search trees created with different C-values are shown in the Results Chapter, Figures 4.1 and 4.2.

Expansion

The expansion, adding a node to the MCTS tree, is made when the selection part of the simulation has led to an new state not previously part of the search tree. A common strategy is to add one new node for each simulation. This allows the tree to grow where it is most beneficial [16].

Playout / Rollout

The playout is started when the end of the search-tree is reached and a new node has been expanded. The playout can be random or guided. Starting from the expanded node the playout usually ends when the game gets to an end-state [16].

Back-propagation

When a node has been expanded and a playout has been made, the expanded node has been attempted once. The number of attempts and the score for all nodes preceding the expanded node are updated by iteratively moving back to the root of the search tree [16].

3.2.1 Final Move Selection

The act of selecting what move to make based on the information gained from the search. Final move selection is done after a search has been finished. Final move selection can be done using one of the following strategies:

1. Max child: Select the child with the highest win-ratio.
2. Robust child: Select the child that has been visited the most.
3. Robust-max: Select the child with both highest win-ratio and most visits [21].

3.3 MCTS based Bot for Candy Crush Saga

This section describes why and how we created a non-deterministic search, some of the limitations of implementing MCTS on Candy, our MCTS implementation and how it is different from ordinary MCTS (described in the previous section).

The bot was written in C++ and built on top of the Candy source code. In short, the Candy source code is created using a main object that controls the game logic, refereed to as the "Candy game logic". When playing Candy regularly, the game logic is given a seed at the beginning of the level, this seed then determines what set of new candies will appear after removing a set of candies, that is, what state the game will move to after an action has been made. In order to have our bot make intelligent evaluations that could represent *human thinking* [1], we argued that we needed to render the bot's search of the game non-deterministic. Meaning, as the Candy game logic contains all the information regarding the current state and is used in every step of the bot's simulation, this required us to change the seed given to the Candy game logic when using the game logic throughout the MCTS. If we were to simulate and search deterministically, without changing the seed

between states, we would have rendered the bot's search equivalent to knowing the future. Letting the bot use this information would have created an unfair advantage and we argued it would have been a bad representation of *human thinking*.

3.3.1 Limitations of Candy Crush Game Logic

The game logic limited us in many ways when implementing the bot, for example, the game logic does not support the possibility of moving backwards in the game, starting from other states than the initial state for a level or moving from one state to another non-consecutive state. This forced us to implement the bot to always simulate from the initial state of the game, even if it was searching from a late state of the game, rendering each simulation very time consuming, see the Results Chapter Figure 4.4, which in turn forced us to limit the amount of simulations for each search, impairing the performance of the MCTS.

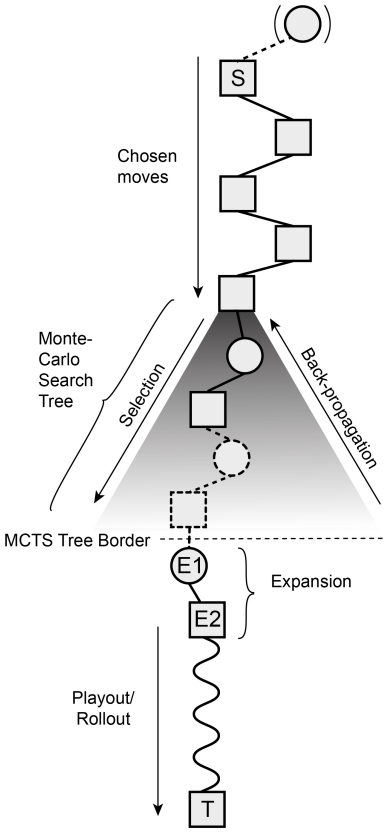


Figure 3.2: Overview of simulation with our bot implemented on Candy. Each simulation starts in the S-state and ends in the T-state. Circles represent chance nodes and squares represent decision nodes. The chance node in parenthesis in the beginning of the node sequence represented the seed of the level which the bot was playing. The two nodes with dotted stroke represent the possibility of additional nodes in the sequence. The simulation is divided into three parts, chosen moves, MCTS and playout. Expansion takes place outside the search tree border. The tree structure and simulation are described in detail in subsection 3.3.2.

3.3.2 Bot MCTS Implementation

The implementation used in the bot was a result of the requirements for the bot and the functionality the Candy game logic presented to us. On a high level the structure is a common one, single-player game tree with chance nodes [17]. The Candy game logic could only be started in the initial state of level. Needing to start in the initial state in every simulation forced us to create a tree structure that contained the nodes for the "Chosen moves" as well as the nodes for the "Monte-Carlo Search tree", Figure 3.2.

Chosen moves part

The first part of the tree structure consists only of decision nodes, these have no chance nodes in between them, see "Chosen moves" part of Figure 3.2. The nodes were used to move the game logic to the state where the bot needed to search, they were only created after a search had been finished and they represented moves already made. These nodes contained both the action made and the state given the action.

MCTS part

The search part of the simulation was based on Sparse UCT using chance nodes [16, 17]. The root of the search part of the tree structure had the same seed as the rest of the chosen nodes, Figure 3.2, enabling the bot to get the actions available at this state. The chance nodes connected to the root of the search tree, and used in other parts of the search tree, contained the actions available. The decision nodes connected to those chance nodes contained information regarding the states. The actual search tree can contain hundreds of nodes and tended not to be symmetric, see Results chapter Figures 4.2 and 4.1 for search tree visualisations. Compared to a regular MCTS, described previously, our bot's search was different in the following ways:

Selection in the search of our bot is made in two steps instead of one. First, choosing action from a given state is done using the UCB formula [13]. Second, given the chosen action, the following state chosen is done randomly. This required a chance node between every pair of decision nodes, see Figure 3.2.

Expansion in the search was, similar to the selection, done in two steps. First, using UCB a chance node representing an action is created. Second, using a random seed a decision node representing a state is created. In certain simulations expansion could occur without expanding an action. This happens when actions had been expanded in previous simulations and, depending on the parameters used, more than one state is required for each action.

Playout in our bot is, similarly to ordinary UCT, started after traversing the search tree. However, as the seed is changed throughout the states of our bot's search tree the last the state the bot will visits before the playout will have a different seed from the seed that the bot was originally playing. The seed of the state that the bot arrived at was then used throughout the playout. This results in that the bot's playout takes place in state space that the bot would never have entered if the bot was searching deterministically. No nodes are created in the playout.

Back-propagation in our bot traverses both decision and chance nodes. Both decision and chance nodes therefore contain information regarding their success rate. However, as stated before, this information is only considered when choosing actions in the search tree.

When searching two common stopping criteria are: either a certain number of simulations or a time limit [16]. Due to different hardware used and simulations being longer/shorter on different levels we chose to use the number of simulations as our stopping criteria.

Final Move

When the bot has finished a search it chooses a final move, this is done using the Max Child formula [21]. Using this action the bot then adds a new node to the chosen moves part of its tree structure. The new node contains both the action and the state. The creation of search trees and addition of nodes to the chosen moves part then continues until the tree structure consists solely of chosen moves. After the last search has been made a final simulation is made, using only chosen moves, the simulation moves the game logic to the level's end state. That completes one attempt at a level.

Acquiring sufficient data on a set of levels was quite hardware demanding. For example, if each MCTS search tree consists of 150 simulations it requires $1.8 \cdot 10^6$ simulations of Candy for 200 attempts on a single level of 60 moves.

3.3.3 Signal Improvement

A finished playout returns a signal. Originally the signal was either zero or one, representing a loss or a win. We developed the playout step of the MCTS algorithm, allowing the signal from the playout to return a continuous value from zero to one. Returning a value between zero and one from the playout gives the MCTS more information and improved the performance, see the Results Chapter Figure 4.11. The non-binary signal is influenced by either partial goals, such as amount of jelly removed, the score given at the end of the playout or a combination of the two. In order to distinguish between an actual win and a very high score we applied two policies. First, we chose to let all playouts return the value one if the playout ended in a win. Second, we used a shrinking factor, a value somewhere between one and zero, that was multiplied by the value provided by the non-binary signals. Using this we could create a larger or smaller distinction between a losing end state with a high score and a winning end state.

Partial Goals

Partial goals in Candy can be related to the different game modes, the amount of removed jellies and / or ingredients and likewise. Partial goals could potentially also be the number of removed blockers and / or other features. Partial goals are different in all games suggesting that creating signals influenced by partial goals in all games could prove impractical. We choose to use the amount of jellies removed to improve our signal. Using only jellies removed we argue would provide us with a sufficient indication of the effects of a signal based on partial goals. The value of our partial goals signal is calculated using the following equation:

$$\text{Signal} = \begin{cases} 1 & \text{if end state is a win} \\ sf \cdot (1 - (J_{end}/J_{init})) & \text{if end state is a loss} \end{cases} \tag{3.2}$$

Where sf is the shrinking factor, J_{end} is the number of jellies left at the end state and J_{init} is the number of jellies present at the initial state.

Score Influenced Signal

Playing Candy scores from a level can in rare cases be very high relative to the mean score, for distribution of scores see Appendix A.1 on page 41. As our search was non-deterministic and high scores were rare we reasoned that a high score could influence the search in an unfavorable way. Even if one simulation get a very high score our play might not ever end up in the state found by the non-deterministic search. There is no set range for scores in

Candy and the distribution of scores changes with different levels. In order to decrease the variance in signal values we therefore decided to remove outliers of high score and normalize the score to a value between zero and one. This was solved using a two part algorithm: First, the bot gets an approximate understanding of the range of the scores for the given level by collecting the scores from every simulation of a search. Second, the range defined from the previous search is used to remove outliers from influencing the bot and to normalize the scores to a value between zero and one. The following equation shows how we calculated the a signal value influenced by score:

$$\begin{aligned} \text{MaxScore} &= Q3 + 1.5 \cdot \text{IQR} \\ \text{MinScore} &= \max[0, Q1 - 1.5 \cdot \text{IQR}] \\ \text{ScoreN} &= \max[0, \min[1, \frac{\text{Score} - \text{MinScore}}{\text{MaxScore} - \text{MinScore}}]] \\ \text{Signal} &= sf \cdot \text{ScoreN} \end{aligned} \tag{3.3}$$

Where sf was the shrinking factor used. $Q1$ is the first quartile, $Q3$ is the third quartile and IQR is the inter-quartile range, all of these were from the distribution of scores from the previous search.

Combination of Score and Partial Goals

We combined the use of partial goals with score to influence our signal. We did this with equal weights.

3.3.4 Different Playouts

A guided playout is a playout that choses its moves in some other way than randomly. For example a playout could evaluate the possible actions in each state using a heuristic and then chose one of the actions based on those values. We experimented with the following four different playouts:

1. Random playout
2. Guided by Tree-Only Move-Avarage Sampling Technique (TO-MAST) table [3]
3. Guided by heuristic
4. Guided by clouded heuristic

The clouded heuristic was created by randomly picking each action with a probability proportional to its heuristic value, see the following equation:

$$p(a) = h(a) / \sum_{i}^{K} h(a_i) \tag{3.4}$$

Where $h(a)$ is the the value for an action given by the heuristic function and i is an action for the given state. K represents all actions for the given state.

A TO-MAST table was implemented using a matrix [3]. The game board positions were represented by positions in the matrix. Every position in the matrix contained a number of visits and a number of wins. After each finished playout the values for certain positions were updated with visits and wins when back propagating the search tree, the moves made in the playout. Actions were only considered in terms of their position on the game board, what state they came from was not considered.

3.3.5 MCTS Improvement and Tweaks

Several improvements on MCTS have been suggested by others [16]. Improvements of MCTS when applied in the field of GGP are MAST, TO-MAST, PAST, FAST, RAVE and keeping track of the best paths. Keeping track of the best paths could be advantageous as averaging simulations could lead to a strong play being discarded due to weak siblings when compared to a subtree where all paths are of medium strength [3]. More improvement to MCTS in the field of GGP are presented in [22]. MCTS has been adapted for single-player games using a modified selection strategy (adding a third term to the UCB formula that represents a possible deviation of the child node) and back-propagation. The adapted MCTS, called SPMCTS was tested on the single-player deterministic game SameGame [9]. Another improvement to MCTS is Beam search. Beam search is a version of MCTS where the amount of nodes per level of the search tree is limited, creating a narrow and deep search tree. Limiting the number of nodes to can be done for each node or for each tree depth level. What nodes to keep in the search tree can also be chosen retroactively after the nodes at a certain depth have been visited sufficient times or by a heuristic [24]. Another proposed improvement to MCTS is using nested searches. A nested search is a method for keeping track of and using previous simulation paths when it's advantageous [7]. Whether or not these would have been beneficial to Candy is arguable. We reasoned that many tweaks would be rendered useless when having a non-deterministic search, a very large state space and few simulations.

3.4 Bot Performance Tests

In order to test our bots behavior and performance we chose to divide our tests into four phases. We used different sets of levels for the different phases. The levels, faulty levels and tweaked levels (that we had to remove from our test sets) are shown in Table 3.2. The first phase was a preliminary phase where we looked at the state space of Candy, the bots behaviour and the bots performance in terms of simulation times. In order to get an understanding for the size of the state space in Candy we ran 10 000 simulations on levels 13, 77 and 100, see Table 3.1. For every state in the simulation the number of actions were recorded and the next action was chosen randomly. Using this information we calculated the state space using the following equation:

$$\prod_{i}^{D} A_i * K \tag{3.5}$$

Where D is the depth of a level, A is the number of available actions / moves for a certain depth and K is a constant for the possible available states given a certain action. Using a constant value of K is a simplification. The possible states available for each action is dependent on the number of candies cleared and the different candies available on the certain level. For example: if an action removes three candies, and there are 5 different candies available on the level then there can be $5^3 = 125$ unique states for that action. If 5 candies are cleared and there are 7 different candies available, there are $7^5 = 16807$ unique states for that action.

In the second phase we aimed to get a, for our scope, sufficient understanding of what parameters to use with the bot. We used three different levels: 13, 77, 100. The levels were chosen as they had different AHSR, they all contained jelly, they had some unique features and they all had a short depth, see Table 3.1 for details regarding the chosen levels. We chose levels containing jelly in order for our partial goals signal to work. The reason

Table 3.1: Features of levels chosen for phase one and two tests. AHSR is the average success rate for all King's tracked players. #J is the amount of Jelly. #Blockers is the number of different blocker types. #M is the predetermined number of moves available. #C is the number of different candies. #P is the number of positions on the game board.

Level	AHSR	#J	#Blockers	#M	#C	#P	Other features
13	20.83%	23	0	21	6	71	None
77	9.01%	9	2	25	4	63	Teleporter
100	4.93%	25	3	20	6	81	Sugar Drop Feature

we chose levels with short depth is because they took less time to simulate, Candy levels can have up to 60 moves of depth. Choosing only short levels could potentially affect the bots performance on them. However, we were mainly interested in testing different bot setups against each other and we argue that this would not be negatively affected by using short levels. To test the different setups we ran 200 attempts for each setup on each level. We were not able to test all parameters exhaustively, therefore we chose to test them in the order we believed would provide us with the greatest understanding of their individual influence on performance, this is presented in the Results Chapter Section 4.2.

For the third phase of testing we wanted to get an understanding of the bot's performance using different setups compared to human performance. We used the parameters we found to be beneficial from the second phase of tests. The only variable throughout the tests were the number of simulations per search and playout type. For these test we chose levels from the same period as our second phase, levels 50-100. For distribution of AHSR for the levels see Appendix A.2. We tested three different bot setups running 50 attempts on each level. To compare the performance of the different setups we used accumulated success rate and delta success rate between the bot and AHSR, referred to as the *delta distribution*.

Lastly, in our fourth phase of testing, we aimed to evaluate different bots' ability to predict AHSR. For these tests we choose a new set of levels. We choose levels 1100-1180 because we had consistent data from the Play testers and that the handcrafted heuristic had been shown to be insufficient at these levels. For distribution over AHSR for the levels we used see Appendix A.3. For each of these levels the Play tests consisted of three people playing each level 50 times each. We had to remove faulty and / or tweaked levels from the test sets. Faulty levels were levels that were either not optimally constructed or contained unique elements that made the bot crash on these levels. Faulty levels were rare, three levels out of 80 were faulty, see Table 3.2. Tweaked levels were levels that have been adjusted after being tested by the Play testers, 27 out the 80 levels were tweaked. Our bot ran on the altered version of these levels and we could therefore not compare different prediction methods on them, leaving 50 levels for our phase four tests, see Table 3.2. In order to test our bot's ability to predict AHSR we used accumulated success rate, the delta success rate between the tested bot and AHSR and an adjusted delta success rate. The adjusted delta took into account the AHSR for the level. The adjusted delta was created because of complications using the ordinary delta. For example, if we tested a bot that produced a lower success rate than the AHSR on a hard set of levels, then the delta between those success rates would still be rather small as the AHSR itself is very low on hard levels. To elaborate, a prediction off by 2% on a level where the AHSR is 20% is considered more accurate than a prediction off by 2% on a level where the AHSR was 4%. That leaves certain predictions to be falsely considered accurate. The adjusted delta was created to adjust for this problem. Adjusted delta was calculated using the following equation:

$$\begin{aligned} \sigma &= \sqrt{p(1-p)} \\ z &= \frac{\hat{p} - p}{\sigma} \end{aligned} \tag{3.6}$$

Where p was the AHSR. $\hat{p}$ was the predicted success rate. σ was used to adjust the delta between $\hat{p}$ and p to the AHSR on the levels.

Table 3.2: Sets of levels, number of attempts, problematic levels and tweaked levels for all test phases.

Test phase	**Levels**	**Attempts**	**Problematic levels**	**Tweaked levels**
1	13, 77, 100	Na	Na	Na
2	13, 77, 100	200	Na	Na
3	50-100	50	55, 88	Na
4	1100-1180	50	1106, 1140, 1160	See appendix

Results

4.1 PHASE 1: State Space and Bot Behavior

The objective of this phase was to get an understanding for the state space of Candy and the behavior of our bots in respect to different C-values used in the selection and time spent in different parts of the simulation. We simulated the levels 13, 77 and 100 10,000 times each and tracked the number of actions available at all states from the initial state to the goal state. Using the average number of actions available at each of the states we approximately calculated the state space in Candy, see Equation 3.5. The average number of actions for each state, the depth, the approximated state space and success rate for the random simulations for the levels 13, 77 and 100 are presented in Table 4.1.

Figures 4.1 and 4.2 on pages 22 and 23 are two visualizations of search trees, containing 100 simulations, created by our bot. The searches were performed from the initial state of level 13, using the same seed for the level for both trees. By using the same seed both searches had the same number of actions available from the initial state. The visualisations of the bot's search trees show how the use of different C-values affects the symmetry of the tree. The second tree, see Figure 4.2, simulated more times through a single move than the first tree, making it more asymmetric and deeper. The favored action for both search trees was a five match, see Figure 4.3. Matching five Candies creates a color bomb, which combined with another regular Candy removes all candies of that color on the board, giving the player a large score. By creating the color bomb it is likely that the future simulations have a higher probability of succeeding.

By tracking the amount of time spent in different parts on the simulation through 200 attempts on level 13 we found that simulations in later searches were roughly as time consuming as simulations in the early search trees, see Figure 4.4 on page 24. On level 13, a level of 21 moves, every simulation took around 0.15 seconds.

Table 4.1: Approximated state space for different levels of Candy. We simulated each level 10 000 times choosing moves randomly. Actions is the average number of actions recorded for each state from the initial state to the end state. The Depth is the number of states from the initial state to the end state. Approximated state space was calculated using formula 3.5. Success Rate is the number of Wins/Attempts on each level.

Level	Actions	Depth	Approx. State Space	Success Rate
13	8.93	21	10^{182}	2.0%
77	13.94	25	10^{103}	0.2%
100	5.20	20	10^{71}	0.0%

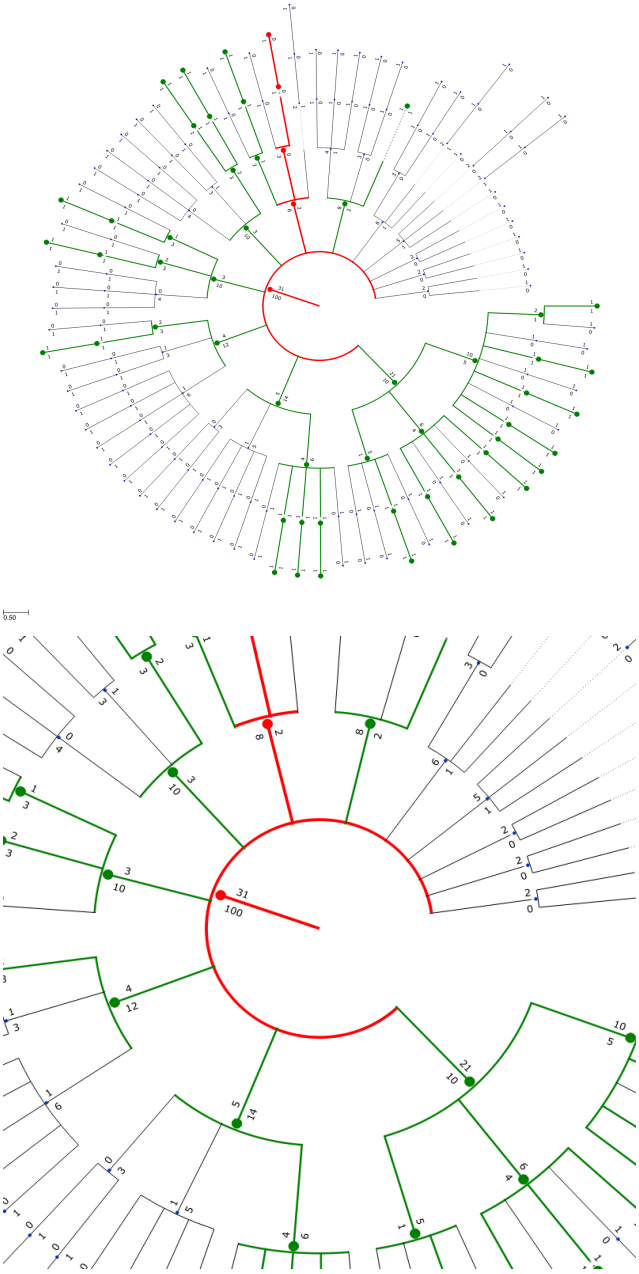


Figure 4.1: Search tree created using a C-Value of 0.6. Upper figure shows complete search tree, lower figure is zoomed in on the centre of the search tree. Paths that lead to leaves of the search tree from which winning playouts were started are presented in green. The red path is the last path before the search stopping criteria was reached. The initial state is represented by the tree root in the middle of the tree. The edges connected to the innermost circle represent actions. Every edge connects to a chance node which connects to one to three decision nodes that represent states. 31 out of 100 simulations in the search were successful. The numbers by each node represents visits and wins through the node. Using the Max Child formula the chosen action would be the action simulated 21 times with 10 wins.

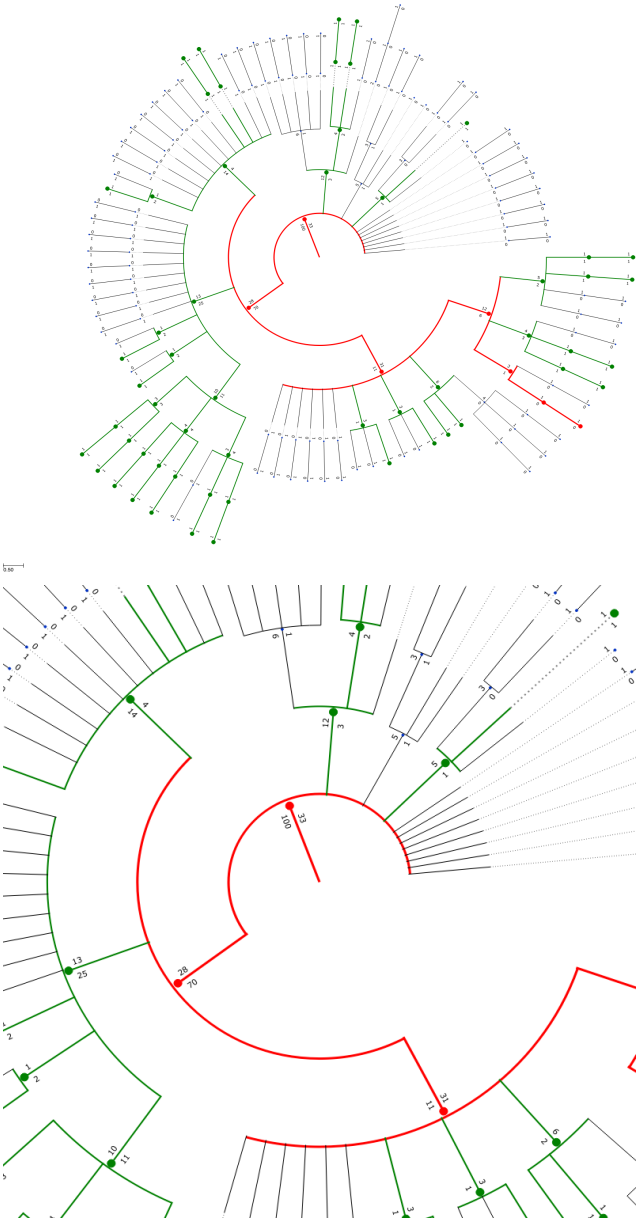


Figure 4.2: Search tree created using a C-Value of 0.1. Upper figure shows complete search tree, lower figure is zoomed in on the centre of the search tree. Paths that lead to leaves of the search tree from which winning playouts were started are presented in green. The red path is the last path before the search stopping criteria was reached. The initial state is represented by the tree root in the middle of the tree. The edges connected to the innermost circle represent actions. Every edge connects to a chance node which connects to one to three decision nodes that represent states. 33 out of 100 simulations in the search were successful. The numbers by each node represents visits and wins through the node. Using the Max Child formula the chosen action would be the action simulated 70 times with 28 wins.

Figure 4.3: Both search trees represented in Figures 4.1 and 4.2 focused a large portion of their simulations on one specific action. That action is shown in the white area above, the action is a match of five candies. Matching five candies in Candy produces a color bomb, a very powerful candy. See Subsection 2.3 for description of Candy.

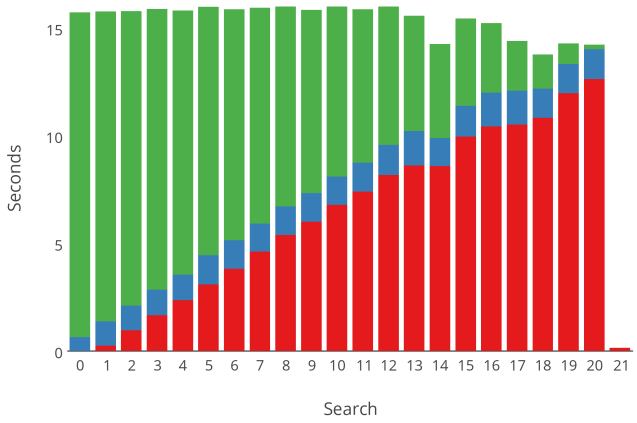


Figure 4.4: Time spent in the different parts of the simulation for a specific setup. The time spent was recorded through 200 attempts on level 13, with 100 simulations per search, using a random playout and a binary signal. Red indicates time of simulation spent in the chosen moves part, blue indicates time spent in the MCTS, green indicates time spent in the playout, see Section 3.3.2 for description of simulation. 108 of the 200 attempts were successful. Level thirteen contains 21 moves, but there were 22 searches made. This is because the last search consists only of moving the game logic from the initial state to the end state using chosen moves.

4.2 PHASE 2: Parameter testing

In this phase we aimed to get a, for our scope, sufficient understanding of what parameters to use with the bot. The parameters we tested were: branching factor / branching limit, the number of possible states for each action. C-Value of the UCB formula, see Equation 3.1. The Search Limit, the maximum depth of the search tree. Signal types, different ways of creating a non-binary signal, see Section 3.3.3. Shrinking, how we weight the value from the non-binary signal in comparison to a playouts win / loss. Playout types, different methods for creating a non-random playout, see Section 3.3.4.

We tested the parameters by running several setups using 100 simulations for each search. We tested the parameters in the order we believed would provide us with the most complete understanding of each parameter's influence on performance, see Table 4.2. We do not view the results from this phase to be significant, we argue that they at most can be used as a rough indication of the parameter's effect on performance. We choose not to exhaustively test all the parameters as we argued that a more thorough test of the parameters would require a far larger set of levels, which in turn also would expand the time needed, and that the potential gains in performance would be unsatisfactory given the amount of time they would require to be tested.

Table 4.2: Setups for different tests. Branching is the maximum number of states that we explored for each action. Search limit is the maximum depth allowed in the search tree. Shrinking is the shrinking factor used, see Subsection 3.3.3. The bold values on the diagonal represent the tested parameters in each test.

Test	Branching	C-Value	Search Limit	Signal Type	Shrinking	Playout Type
1	**1, 3, 5, 10**	0.6	False	Binary	False	Heuristic
2	3	**0.2, 0.4, 0.6, 0.8**	False	Binary	False	Heuristic
3	3	**0.2, 0.4, 0.6, 0.8**	False	Score	False	Random
4	3	0.6	**False, 1, 2, 3, 4**	Binary	False	Heuristic
5	3	0.6	**False, 1, 2, 3**	Score	False	Random
6	3	0.6	False	**Binary, PG, Score, Comb.**	False	Random
7	3	0.6	False	Score	**0, 0.25, 0.5, 0.75, 1**	Random
8	3	0.6	False	Score	False	**Random, TO-MAST, Heuristic, Clouded Heuristic**

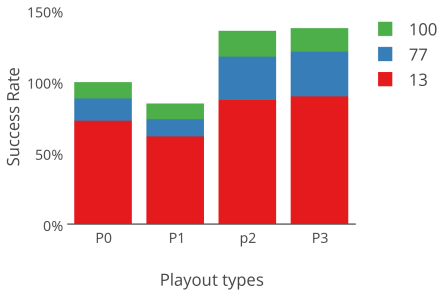


Figure 4.5: Test 8, shows performance on levels 13, 77 and 100. Colored respectively in red, blue and green. P0 means random playout, P1 means playout guided by TO-MAST. P2 means playout guided by clouded heuristic. P3 means playout guided by heuristic. All setups used a score influenced signal.

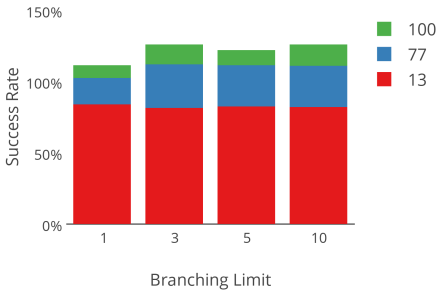


Figure 4.6: Test 1, shows performance on levels 13, 77 and 100. Colored respectively in red, blue and green.

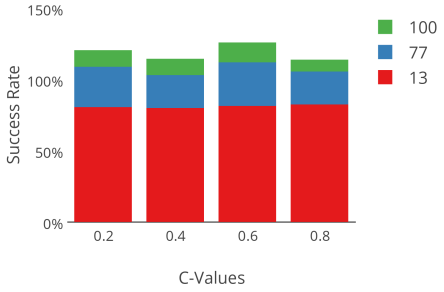


Figure 4.7: Test 2, shows performance on levels 13, 77 and 100. Colored respectively in red, blue and green. All bots were using a heuristic playout and binary signal.

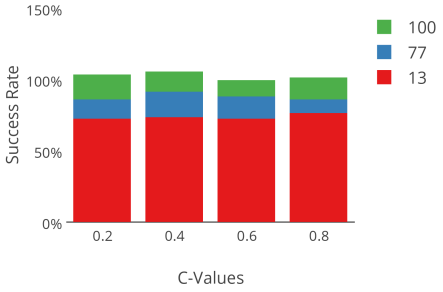


Figure 4.8: Test 3. shows performance on levels 13, 77 and 100. Colored respectively in red, blue and green. All bots were using using random playout and score influenced signal.

We tested four different playout types, the first one being a random one. The other playouts were guided by either a TO-MAST table, clouded heuristics or a heuristic. Using TO-MAST table to guide playout did not improve performance compared to a random playout, using the handcrafted heuristic or the clouded heuristic to guide the playout did improve performance, see Figure 4.5. We tested four different branching limits, the maximum number of states for each action, they were one, three, five and ten. The performance on the levels tested was not largely affected by the branching limit other than that using a branching limit of one showed to perform slightly worse, see Figure 4.10. For subsequent tests, we decided to use a branching factor of three. We tested the C-values of 0.2, 0.4, 0.6 and 0.8 for two bot setups, the first used a binary signal and heuristic playout and the second one used a score influenced signal and random playout. Neither of the two bot setups showed consistent difference in performance, see Figures 4.7 and 4.8. With no C-value concluded as superior, we chose to continue our tests with a C-Value of 0.6. Similarly, using the same two different bot setups we tested different search depth limits. For the bot using

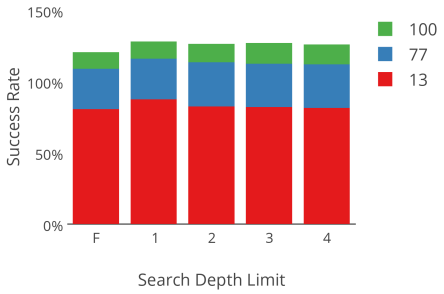


Figure 4.9: Test 4. shows performance on levels 13, 77 and 100. Colored respectively in red, blue and green. False means no limit to search depth. All bots were using a heuristic playout and binary signal.

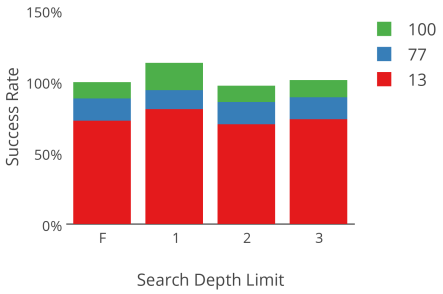


Figure 4.10: Test 5. shows performance on levels 13, 77 and 100. Colored respectively in red, blue and green. False means no limit to search depth. All bots were using a random playout and score influenced signal.

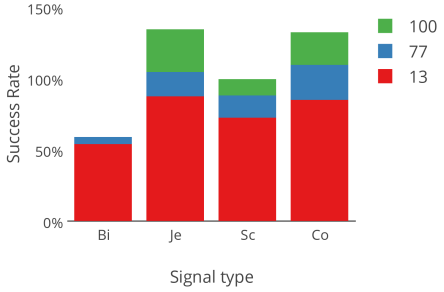


Figure 4.11: Test 6. shows performance on levels 13, 77 and 100. Colored respectively in red, blue and green. Binary signal performed worse, jelly influenced signal and combined signal performed best.

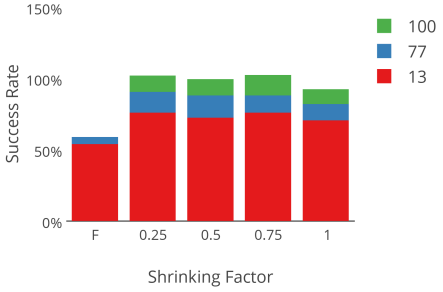


Figure 4.12: Test 7. shows performance on levels 13, 77 and 100. Colored respectively in red, blue and green. Using a shrinking value of 0 or 1 performed worse.

heuristic playout and binary signal we tested to use no limit to search depth as well as depth of 1, 2, 3 and 4. For the bot using random playout and score influenced signal we used the same limits except the depth of four as we found no great impact from different search depths. To conclude, the different search depths did not largely influence performance, see Figures 4.9 and 4.10. However, using a limited search depth of one showed to slightly increase performance. As these indications were weak we chose to continue our tests using no restriction as to search depth. We tested three different signal types besides the binary signal, considering only wins or losses. The other signals tested were influenced by either partial goals, score or a combination of the two of them. The binary signal, performed worse than the other signal types, see Figure 4.11. The signals that considered to what degree partial goals had been completed performed best, see Figure 4.11. However, moving forward we chose to use the signal that took into account the score of the Playout. The reason we chose to use a score influenced signal was because score is used in all levels of Candy, which is not the case for jelly, also score is a more common feature in all of King's games and using score to influence our signal we argued was the most general approach.

Last we tested the following different shrinking factors: False (Same as using a binary signal), 0.25, 0.5, 0.75 and 1 (no necessary distinguishing between losses with high scores and actual wins). The results show that distinguishing between actual playout wins and very good scores improved performance, see Figure 4.12. Unless stated otherwise all subsequent bot setups used the following setup: No limit to search depth, branching factor of three, C-Value of 0.6, shrinking factor of 0.5, score influenced signal, 100 simulations per search and random playout.

4.3 PHASE 3: Bot Performance

For this phase of testing we wanted to get an understanding of the different bot's performance compared to AHSR, in order to do this we used three different bot setups as well as the handcrafted heuristic. The bot setups used either 50 or 100 simulations per search. Two bot setups used 100 simulations per search, they were distinguished by using different playout types. The other parameters used for the bots were the parameters found to be most beneficial from the second phase of testing, see Table 4.3. The different setups were tested on 48 levels, from level 50 to 100, the removed levels were levels that the bot crashed on. The tests show that bot setups on average over performed compared to the AHSR, see Figures 4.13 and 4.14. The tests also indicate that an increased number of simulations and the use of a guided playout had an additive effect on performance, 4.13. On these levels the best setup was B3, that setup's success rate was on average 43% higher than AHSR, see Table 4.4 for mean and standard deviation of the delta distributions of different bot setups and AHSR. Delta success rate was smaller on hard and easy levels, indicating that an adjusted delta measurement might be needed for comparing predictions of success rate, see Figure 4.14. The linear trend line shows how the use of heuristics increased the delta on mainly the harder levels in the test sets, see Figure 4.14.

Table 4.3: Different MCTS bot setups used in PHASE 3 tests. *#B* is the maximum number of states explored for each action. *C* is the C-value used in the UCB formula. *S limit* is the maximum depth allowed in the search tree. *Signal* is the signal type used. *Shrinking* is the shrinking factor. *Playout* is the playout type. *#Sim* is the number of simulations for each search.

Bot setup	#B	C	S Limit	Signal	Shrinking	Playout	#Sim
B1	3	0.6	False	Score	False	Random	50
B2	3	0.6	False	Score	False	Random	100
B3	3	0.6	False	Score	False	Heuristic	100

Table 4.4: Mean and standard deviation (SD) for the delta distribution over all levels for the different setups. For description over bot setups see Table 4.3

	Mean	SD
H. Heuristic - AHSR	7.46%	13.5%
B1 - AHSR	19.4%	16.6%
B2 - AHSR	30.1%	16.2%
B3 - AHSR	42.9%	19.0%

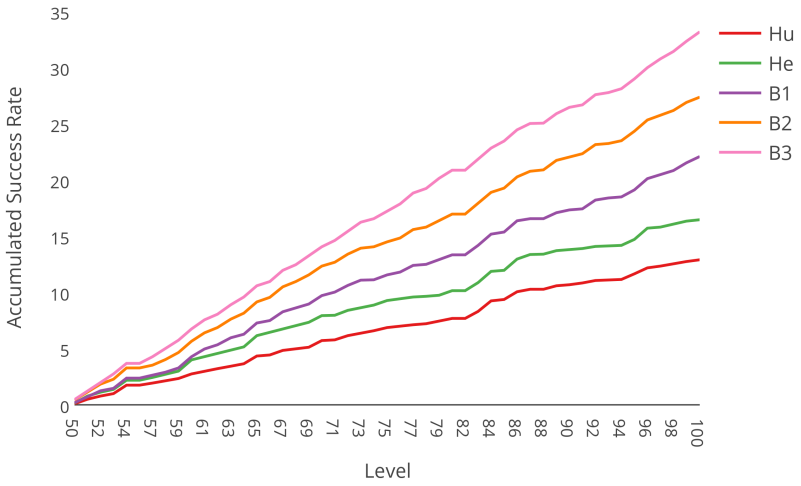


Figure 4.13: Accumulated success rate over the 48 levels tested from 50 to a 100. Levels are ordered in chronological order. AHSR (Hu), hand-crafted heuristic (He), bot using 50 simulation per search random playout and score influenced signal (B1), bot using 100 simulation per search random playout and score influenced signal (B2) and bot using 100 simulation per search playout guided by hand-crafted heuristic and score influenced signal (B3) are shown respectively in red, green, purple, orange and pink.

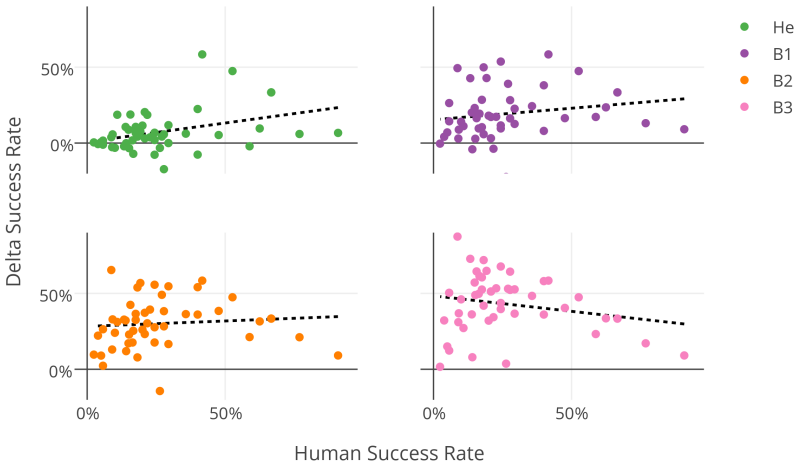


Figure 4.14: Delta success rate over levels tested from 50 to a 100. Delta between bot setup success rate and AHSR as a function of AHSR. Delta success rate between AHSR (Hu) and Hand-crafted heuristic (He), bot using 50 simulation per search random playout and score influenced signal (B1), bot using 100 simulation per search random playout and score influenced signal (B2), bot using 100 simulation per search playout guided by hand-crafted heuristic and score influenced signal (B3) are shown respectively in green, purple, orange and pink in separate plots. Dotted black line indicates linear trend line.

4.4 PHASE 4: Predicting AHSR

In this phase of testing, we aimed to evaluate the bots ability to predict AHSR. The test consisted of testing two bot setups on 50 different levels between level 1100 and level 1180, see Appendix A.2 for levels used. The two bot setups, B2 (same bot setup as used in phase 3 tests) and B4 used respectively 100 and 150 simulation per search. The other parameters used for the bots were those that showed to perform best in the second phase of testing, see Table 4.5 for bot parameters used. The two bot setups performed worse on average compared to AHSR, see Figure 4.15. However, using both absolute delta and adjusted delta showed that bot B4 predicted AHSR most accurately out of the methods tested, see Figures 4.16 4.17, 4.18 and 4.19. The mean and standard deviation (SD) for delta and adjusted delta between setups and AHSR show that bot B4 predicted AHSR most accurate, see Table 4.6. Using absolute delta as measurement the bot's predictions were most accurate on the easier and harder levels in the test set, see Figure 4.16. Using a linear trend line shows how the different prediction methods performed on average on levels of different AHSR, see Figures 4.16 and 4.18. Both the delta and adjusted delta indicate that B4 predicted both the easier levels of the tests set accurately compared to Play tester or heuristics, see Figures 4.16 and 4.18. The hand-crafted Heuristic underperformed on the whole set of levels, see Figure 4.16.

Table 4.5: Different MCTS bot setups used in PHASE 4 tests. *#B* is the maximum number of states explored for each action. *C-v.* is the C-value used in the UCB formula. *S. limit* is the maximum depth allowed in the search tree. *Signal* is the signal type used. *Shrinking* is the shrinking factor. *Playout* is the playout type. *#Sim* is the number of simulations for each search.

Bot setup	#B	C-v.	S. Limit	Signal	Shrinking	Playout	#Sim
B2	3	0.6	False	Score	False	Random	100
B4	3	0.6	False	Score	False	Random	150

Table 4.6: Mean and standard deviation (SD) for the delta and adjusted delta (adj) distribution over all levels between the different setups and AHSR. For description over bot setups see Table 4.5

	Mean	SD	adj Mean	adj SD
Play t. - AHSR	6.63%	16.6%	0.245	0.718
H. Heuristic - AHSR	-15.4%	10.7%	-0.406	0.182
B2 - AHSR	-3.12%	11.6%	-0.105	0.278
B4 - AHSR	-1.29%	9.97%	-0.041	0.247

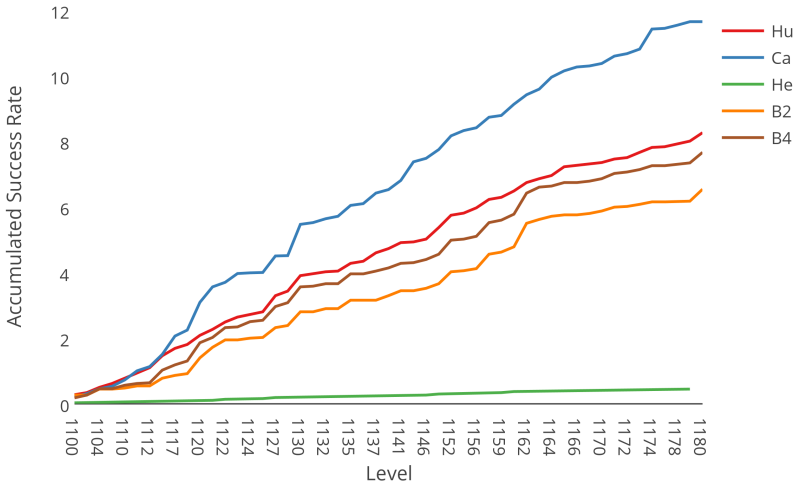


Figure 4.15: Accumulated success rate for, AHSR (Hu), Play tester (Ca), Hand-crafted heuristic (He), bot using 100 simulations (B2) and bot using 150 simulation (B4). Levels are ordered in chronological order. Shown respectively in red, blue, green, orange and brown. Both bots used random playout and signal influenced by playout score.

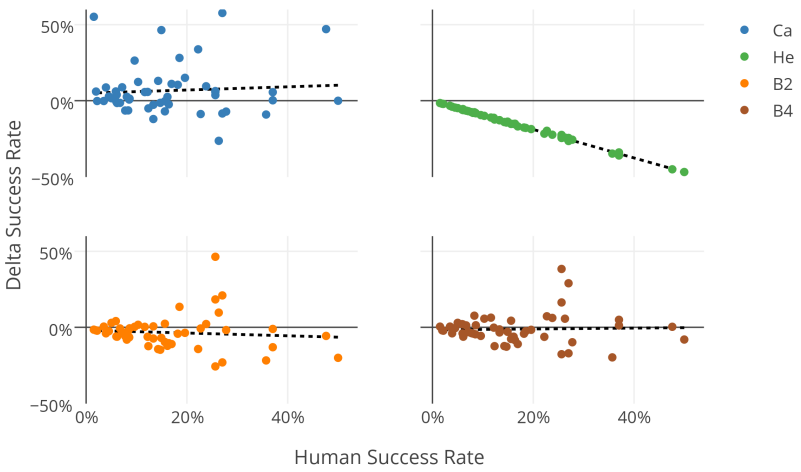


Figure 4.16: Delta success rate between bot setups success rate and AHSR as a function of AHSR. Delta success rate between AHSR (Hu) and Play testers (Ca), Hand-crafted heuristic (He), bot using 100 simulation (B2) and bot using 150 simulation (B4) are shown respectively in blue, green, orange and brown in separate plots. Both bots used random playout and signal influenced by playout score. Dotted black line indicates linear trend line.

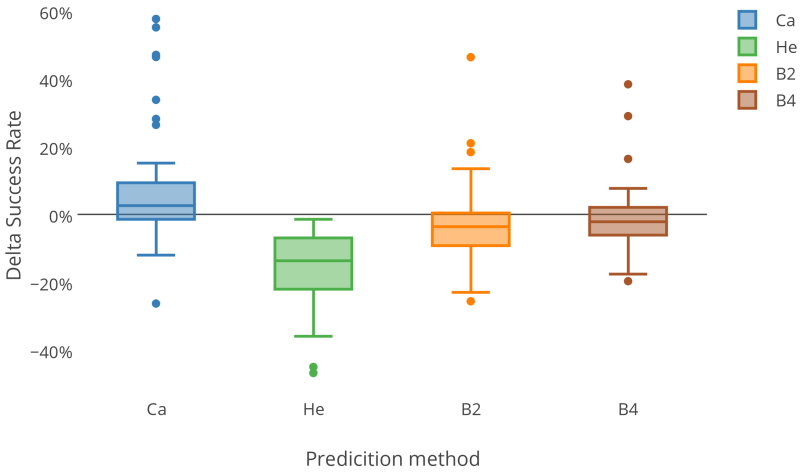


Figure 4.17: Boxplots over the delta between bot setups success rate and AHSR. Play tester (Ca), Hand-crafted heuristic (He), bot using 100 simulations (B2), bot using 150 simulations (B4) and are shown respectively in blue, green, orange and brown. Both bots used random playout and signal influenced by playout score.

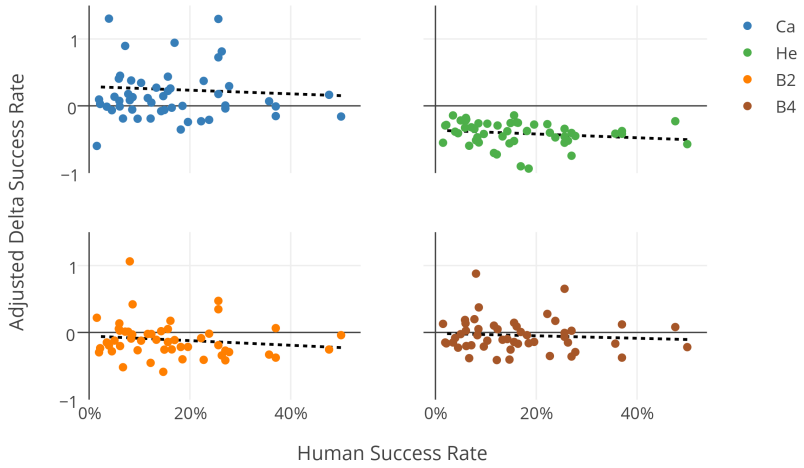


Figure 4.18: Adjusted delta success rate between bot setups success rate and AHSR as a function of AHSR. Adjusted delta success rate between AHSR (Hu) and Play testers (Ca), Hand-crafted heuristic (He), bot using 100 simulation (B2) and bot using 150 simulation (B4) are shown respectively in blue, green, purple, orange and brown in separate plots. Both bots used random playout and signal influenced by playout score. Dotted black line indicates linear trend line.

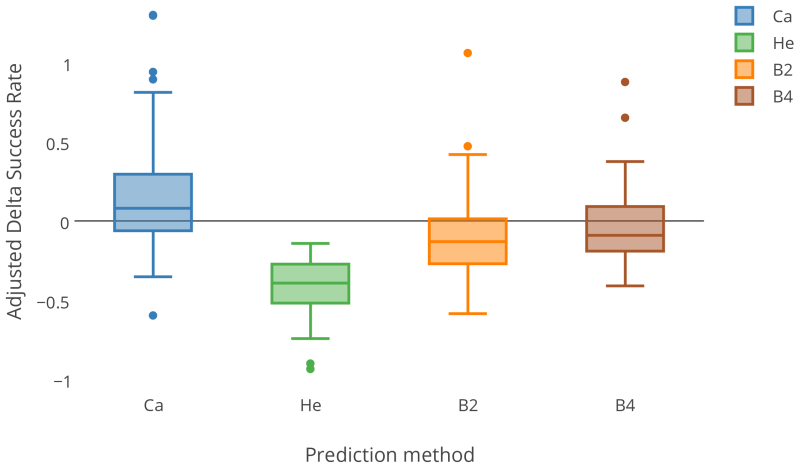


Figure 4.19: Boxplots over the adjusted delta, see section 3.4, between bot setups success rate and AHSR. Play tester (Ca), Hand-crafted heuristic (He), bot using 100 simulations (B2), bot using 150 simulations (B4) and are shown respectively in blue, green, orange and brown. Both bots used random playout and signal influenced by playout score.

Discussion / Future Work

We implemented and tested a MCTS based bot on King's Candy Crush Saga. Our results indicate that AHSR can be predicted more accurately using MCTS than current state-of-the-art methods. Our results also suggest that a MCTS based bot could be incorporated to streamline level development in Candy.

In addition to answering our research question, we argue that by testing MCTS on Candy, a unique single-player stochastic game with many different features and challenges, we have contributed to the understanding of MCTS in the following ways. First, the achievements in the GGP field obtained by using MCTS have shown that MCTS is a general algorithm. It has also been suggested that MCTS, with its forward sampling approach, in some sense mimics human thinking. This hypothesis is corroborated by our results as they indicate that MCTS performs well on the complex problem Candy and MCTS can accurately predict AHRS rate in Candy. Secondly, we believe that our tests have added to the knowledge regarding whether or not MCTS is an AI algorithm that is *thinking humanly*, see Section 2.1. We would not want to claim that we have in any way proved that MCTS is *thinking humanly*. However, we believe that Candy, due to its diversity and the amount of accurate human data available for the game, is a excellent problem for testing the degree to which algorithms are *thinking humanly*. We do claim that our results indicate that our bots predicts AHSR better than the Play testers. However we argue that the prediction of AHSR using Play testers is non-optimal as they are only three people playing 50 attempts on each level.

From the first phase of our tests, we can conclude that exhaustive search of the state space is infeasible. Our tests were affected by the fact that the majority of the simulations lead to loosing end states. A guided simulation, acquiring a higher success rate, would potentially had resulted in larger state space as fewer simulation would have terminated prematurely from for example not removing elements such as bombs. Therefore it is possible that the state space is larger than what our results indicate. Figure 4.4 shows that the time spent for each search could have been halved if we would have been able to move the game logic between non-consecutive states.

The idea behind the second phase of tests was not to exhaustively find the optimal setup for our bot. Rather the idea was to find the most beneficial changes within the scope of this project. Meaning that even if we could have done more tests to find a perfect C-value for Candy, which probably would depend on what levels were being tested as the other bot parameters, we argued that little benefits, when considering the research question and future tests, would come from this and chose use a C-value of 0.6. Preferably we should have enlarged all tests of parameters using a test set of levels similar to the levels used in the third and fourth test phases combined. Unfortunately this was not possible using the resources available.

Our third phase of tests showed that the number of simulations per search improved the

performance, see Figure 4.13. By presenting the delta success rate as function of AHSR we can see that delta success rate falsely classifies predictions on hard levels as more accurate, see Figure 4.14. We can also see that using a heuristic, used by bot B3, improved the performance mostly on the harder levels, suggesting that different ways of adjusting the MCTS bot can improve performance on different types of levels, see Figure 4.14. These results suggests that alterations can be made to the MCTS in order to more accurately mimic human playing. The tests also showed that having a heuristic guiding the playout was beneficial on these levels. However, we did not use the heuristic in the later levels for the following reasons. The heuristic had been optimized for early levels and would not increase performance on later levels, it requires constant maintenance as Candy develops, we acquired sufficient results without using it, and we wanted to remove all game-specific elements from the MCTS in order to prove that a general bot setup was sufficient to predict success rate.

Our fourth phase of tests indicate that using delta success rate as an measurement can be problematic. Presenting delta success rate as a function of AHSR, shows that having a prediction method consistently underperform, in this case the handcrafted heuristic, could falsely be considered accurate if only tested on a hard set of levels, see Figure 4.16. We can also see that the delta was smaller on hard and easy levels, see Figure 4.16. However, using both the delta and adjusted delta measurements indicate that bot B4 predicted success rate more accurate than other prediction methods, see Figures 4.16, 4.17, 4.18 and 4.19.

By comparing the bot B2's performance on the levels from the third and fourth phases, we can clearly see that the same bot setup performed worse on the later set of levels. The difference in bot performance raises the question whether or not this is due to the bot being more appropriate for the early levels or not. The reason for this difference in bot performance could be due to the score influenced signal. Potentially the score of early levels correlate more with actual wins rather than later levels. However, another hypothesis is that the player base has evolved throughout the different levels. It is not unreasonable to think that bad players have been removed from the player base after hundreds of Candy levels and that remaining players have gained in skill.

To conclude, we believe that our results indicate that AHSR in Candy can be predicted fairly accurately using bots, however we argue that extended tests containing more levels, with different AHSR, are needed in order to conclude this with confidence.

5.1 Future Work

There are several different directions for future work. In this section we present the following ones. Technical developments to our MCTS bot implementation, extended use of player data, investigating the aspect of *human thinking* and future work at King.

During our work we chose to keep our MCTS implementation simple. There were several MCTS adjustments that we did not implement and / or test. First, we could have developed the search tree creation. One way of doing this would have been to limit the number of actions expanded for each state using some type of Beam search [24]. The number of actions could either be limited to the search tree level or limited for each node. We believe that, using a rather simple heuristic, a subset of all available moves available from a certain state can be distinguished as more beneficial compared to all available moves. In addition to investigating different methods for choosing what actions to explore, we could potentially also dynamically determine the number of states to explore for each action. For example, if a playouts score is rare, based on the distribution of previous scores, further states could be expanded in order to give the action a more accurate estimated value for the action.

Second, we could further develop and test more Playout types. During our thesis work a separate team was working on creating a general heuristic using an Artificial Neural Network (ANN). We have combined the ANN with the MCTS, using the ANN to guide the playout. However these tests were left outside the scope of this thesis. In addition, if we develop the bot, allowing for more simulation per search, many methods in the field of GGP would be interesting to test. We could possibly develop an algorithm for classifying different types of levels allowing for different strategies to be used for different levels.

We could investigate methods for shortening search time. First, cutting simulations short when possible. Second, using a dynamic stopping criteria for the search. For example, the stopping criteria could be dependent on the win ratio of the most recent simulations. One hypothesis is that choosing several strong moves in the beginning of a level renders the following moves less important regarding the likelihood of successfully completing the level. If this hypothesis is true then, by creating shorter searches when a win is already secured, the average time for attempts could be shortened without significantly decreasing the success rate.

We would like to further investigate the bot's performance in compared to AHSR. For example, further investigating the difference in bot performance on early versus late levels. Potentially, using data from the players, a metric could be found and used to adjust the bot's search stopping criteria, resulting in a bot that can predict AHSR accurately whether the level is an early one or a late one. Also, as we are aiming to predict the average human success rate, rather than the success rate of the best human players, it would be interesting to test the effects introducing noise to different parts the MCTS algorithm. We would like to test the effects of using time as an stopping criteria for the bot's search rather than number of simulations. Using time as the search' stopping criteria would result in fewer simulations per search on deeper levels, perhaps predicting success rate more accurately. Observing the data gathered from players, we could investigate if there is a correlation between depth of level and success rate, if this is the case then a bot using level depth to influence the search stopping criteria could be beneficial.

We could further investigate what parameters of the bot makes the bot behave most humanly. Considering the definition of *thinking humanly*, see Section 2.1, and the results we present it could be argued that our bot is *thinking humanly*, especially when comparing the bot's performance to the performance of to the Play testers. We do not claim that this is the case, we do however argue that Candy, with its diverse range of features and challenges is an excellent proving ground for further investigation to what tweaks to the MCTS that would result in the most human behavior. By using a subset of the data gathered from King's players, we could also potentially identify certain types of players that are easier to predict. One hypothesis for this is certain types players are easier to predict, as their play potentially is more consistent.

Future work at King consists of exporting the bot to more powerful hardware. Using a server and a web interface we could create a easy-to-use tool for level developers, allowing them to get direct feedback on the difficulty of the levels they are creating. In addition we could continue our work by implementing the bot on other games besides Candy, such as Candy Crush Soda Saga, Candy Crush Jelly Saga, and Blossom Blast Saga. Blossom Blast Saga proposes an interesting challenge as every state has thousands of possible actions. We argue that King should consider creating an Application Programming Interface (API) for all games. Games with an API would simplify the use of bots. The current bot logic created for this thesis should be moved from Candy Crush Source code to external library in order to enable use in all games. Future work on Candy could include creating a Game logic copy constructor allowing for faster searches. Preferably all games should use a game logic that can be moved between any two states in the state space. New measurements for the

accuracy of success rate predictions on levels could be investigated.

Bibliography

[1] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[2] Alan M Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[3] Hilmar Finnsson. *Simulation-Based General Game Playing*. PhD dissertation, ReykjavÃk University, Computer Science Department, 5 2012.

[4] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and games*, pages 72–83. Springer, 2006.

[5] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[6] Arpad Rimmel, Fabien Teytaud, and Tristan Cazenave. Optimization of the nested monte-carlo algorithm on the traveling salesman problem with time windows. In *Applications of Evolutionary Computation*, pages 501–510. Springer, 2011.

[7] Tristan Cazenave. Nested monte-carlo search. *Int. Joint Conf. Artif. Intell.*, pages 456–461, 2009.

[8] Christopher D. Rosin. Nested rollout policy adaptation for monte carlo tree search. In Toby Walsh, editor, *IJCAI*, pages 649–654. IJCAI/AAAI, 2011.

[9] Maarten P. D. Schadd, Mark H. M. Winands, H. Jaap van den Herik, Guillaume M. J. B. Chaslot, and Jos W. H. M. Uiterwijk. *Single-Player Monte-Carlo Tree Search*, pages 1–12. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[10] Jan Schäfer, Michael Buro, and Knut Hartmann. The uct algorithm applied to games with imperfect information. 2008.

[11] Guy Broeck, Kurt Driessens, and Jan Ramon. Monte-carlo tree search in poker using expected reward distributions. In *Proceedings of the 1st Asian Conference on Machine Learning: Advances in Machine Learning*, pages 367–381. Springer-Verlag, 2009.

[12] François Van Lishout, Guillaume Chaslot, and Jos WHM Uiterwijk. Monte-carlo tree search in backgammon.

[13] Levante Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *European Conference on Machine Learning (ECML)*, pages 282–293, Berlin / Heidelberg, 2006. Springer.

[14] Ronald Bjarnason, Alan Fern, and Prasad Tadepalli. Lower bounding klondike solitaire with monte-carlo planning. In *Nineteenth International Conference on Automated Planning and Scheduling*, 2009.

[15] Toby Walsh. Candy crush is np-hard. *CoRR*, abs/1403.1911, 2014.

[16] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4:1–43, 03/2012 2012.

[17] Nicolas Jouandeau and Tristan Cazenave. *Technologies and Applications of Artificial Intelligence: 19th International Conference, TAAI 2014, Taipei, Taiwan, November 21-23, 2014. Proceedings*, chapter Monte-Carlo Tree Reductions for Stochastic Games, pages 228–238. Springer International Publishing, Cham, 2014.

[18] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(2):100–107, 1968.

[19] J. v. Neumann. Zur theorie der gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, 1928.

[20] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975.

[21] Guillaume Chaslot. *Monte-Carlo Tree Search*. PhD dissertation, Universiteit Maastricht, 9 2010.

[22] Maciej Â´Swiechowski, HyunSoo Park, Jacek Ma'ndzuik, and Kyuing-Joong Kim. Recent advances in general game playing. *The Scientific World Journal*, 2015:22, 2015.

[23] Hilmar Finnsson and Yngvi Björnsson. Game-tree properties and mcts performance. In *The IJCAI-11 Workshop on General Game Playing*, page 23. Citeseer.

[24] H. Baier and M. H. M. Winands. Beam monte-carlo tree search. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 227–233, Sept 2012.

Appendix

A.1 Distribution of Scores

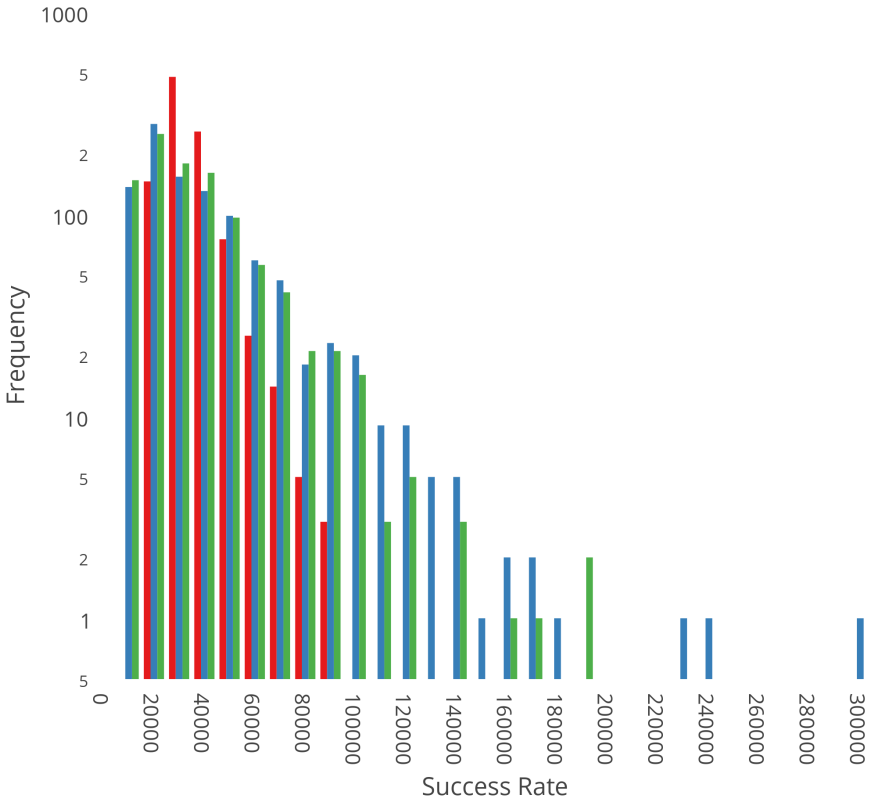


Figure A.1: Distribution of scores returned from random playouts on levels 13, 77 and 100. Shown respectively in red, green and blue.

A.2 Levels

A.2.1 PHASE 4 levels

For our fourth test phase we used levels in the range 1100-1180 that had not been adjusted post Canadian tests and didnÂ´t crash using the bot. These levels were:
1100, 1102, 1104, 1105, 1110, 1111, 1112, 1116, 1117, 1118, 1120, 1121, 1122, 1123, 1124, 1126, 1127, 1129, 1130, 1131, 1132, 1134, 1135, 1136, 1137, 1139, 1141, 1142, 1146, 1149, 1152, 1153, 1156, 1157, 1159, 1161, 1162, 1163, 1164, 1165, 1166, 1167, 1170, 1171, 1172, 1173, 1174, 1176, 1178, 1179, 1180

Problematic levels were:
1106, 1140, 1160

Tweaked levels were:
1101, 1103, 1107, 1108, 1109, 1113, 1114, 1115, 1119, 1125, 1128, 1133, 1138, 1143, 1144, 1145, 1147, 1148, 1150, 1151, 1154, 1155, 1158, 1168, 1169, 1175, 1177

A.2.2 PHASE 3 & 4 Level Difficulty

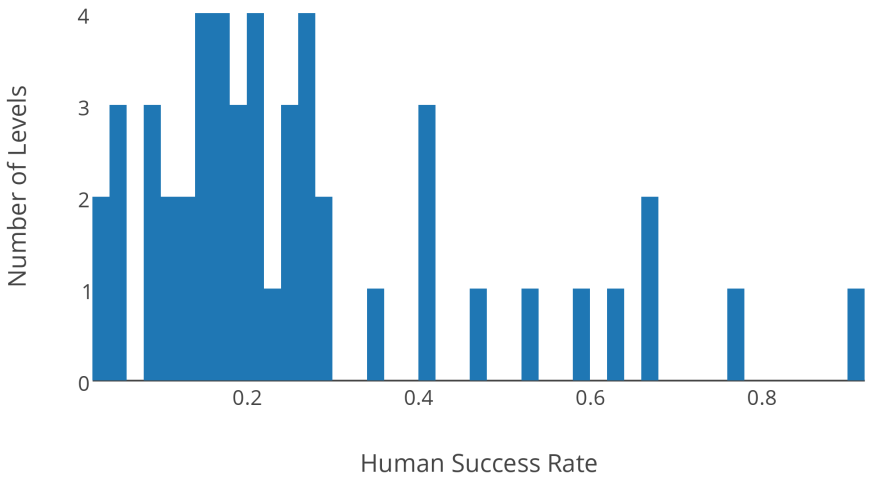


Figure A.2: Distribution of human success rate of levels for PHASE 3 tests.

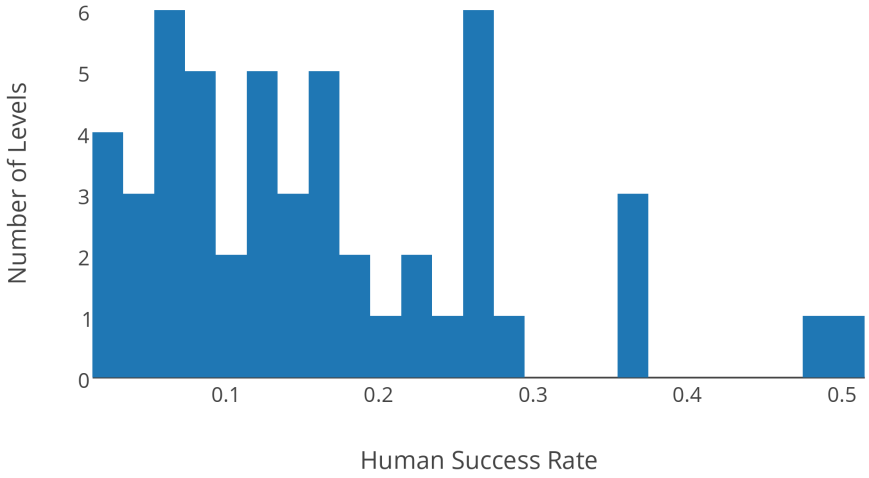


Figure A.3: Distribution of human success rate of levels for PHASE 4 tests.