UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

A novel computer Scrabble engine based on probability that performs at championship level

Alejandro González Romero

ADVERTIMENT La consulta d'aquesta tesi queda condicionada a l'acceptació de les següents condicions d'ús: La difusió d'aquesta tesi per mitjà del repositori institucional UPCommons (http://upcommons.upc.edu/tesis) i el repositori cooperatiu TDX (http://www.tdx.cat/) ha estat autoritzada pels titulars dels drets de propietat intel·lectual **únicament per a usos privats** emmarcats en activitats d'investigació i docència. No s'autoritza la seva reproducció amb finalitats de lucre ni la seva difusió i posada a disposició des d'un lloc aliè al servei UPCommons o TDX. No s'autoritza la presentació del seu contingut en una finestra o marc aliè a UPCommons (*framing*). Aquesta reserva de drets afecta tant al resum de presentació de la tesi com als seus continguts. En la utilització o cita de parts de la tesi és obligat indicar el nom de la persona autora.

ADVERTENCIA La consulta de esta tesis queda condicionada a la aceptación de las siguientes condiciones de uso: La difusión de esta tesis por medio del repositorio institucional UPCommons (http://upcommons.upc.edu/tesis) y el repositorio cooperativo TDR (http://www.tdx.cat/?locale-attribute=es) ha sido autorizada por los titulares de los derechos de propiedad intelectual **únicamente para usos privados enmarcados** en actividades de investigación y docencia. No se autoriza su reproducción con finalidades de lucro ni su difusión y puesta a disposición desde un sitio ajeno al servicio UPCommons No se autoriza la presentación de su contenido en una ventana o marco ajeno a UPCommons (*framing)*. Esta reserva de derechos afecta tanto al resumen de presentación de la tesis como a sus contenidos. En la utilización o cita de partes de la tesis es obligado indicar el nombre de la persona autora.

WARNING On having consulted this thesis you're accepting the following use conditions: Spreading this thesis by the institutional repository UPCommons (http://upcommons.upc.edu/tesis) and the cooperative repository TDX (http://www.tdx.cat/?locale- attribute=en) has been authorized by the titular of the intellectual property rights **only for private uses** placed in investigation and teaching activities. Reproduction with lucrative aims is not authorized neither its spreading nor availability from a site foreign to the UPCommons service. Introducing its content in a window or frame foreign to the UPCommons service is not authorized (framing). These rights affect to the presentation summary of the thesis as well as to its contents. In the using or citation of parts of the thesis it's obliged to indicate the name of the author.

A Novel Computer Scrabble Engine Based on Probability that Performs at Championship Level

Author: Alejandro González Romero

Thesis Director: René Alquézar Mancho

PhD. In Artificial Intelligence

Universitat Politècnica de Catalunya, UPC.

December 22th , 2021

Wholeheartedly dedicated

To God who is always with us, and helps us despite our lack of faith. Heavenly father receive my eternal thanks for allowing me to finish this thesis.

To my beloved parents, Susana and Fico, who have loved, helped and supported me unconditionally all my life. Thanks so much for believing in me, for bringing me to life, for everything.

To my thesis advisor, René, for all his huge help, support and friendship. Many thanks for your trust and believing in me despite all the adversities.

Finally, but not for that to a lesser extent, I wholeheartedly dedicate this thesis in memory of a very kind person, who recently passed away, to my beloved cousin, Dr. Manuel Romero Salcedo (Manolé), a great human being with a huge heart.

Chapter 1 Introduction 1
1.1 The Turing Test
1.2 Computer Chess
1.3 Games
1.4 About Scrabble
1.5 Some Characteristics of Scrabble
1.6 Computer Scrabble
1.7 Objectives and expected contributions of this thesis
1.8 Structure of this document
Chapter 2 Games and their Artificial Intelligence Fundamental Algorithms 7
2.1 Game Theory
2.2 History of some Games in AI
2.3 Fundamental AI Algorithms
2.4 State-of-the-Art Engines (and the Algorithms used)
Chapter 3 The Basic Elements 45
3.1 Letter Distribution
3.2 Scrabble Rules
3.3 What's different about competitive Scrabble?
Chapter 4 History and State of the Art of Computer Scrabble 53
4.1 Generation of valid moves
4.2 Scrabble Engines
4.3 Monte Carlo Simulation
4.4 Opponent Modeling
4.5 The Endgame
Chapter 5 The Beginning 69
5.1 The Lexicon
5.2 A Spanish Scrabble program
5.3 About Heuri
Chapter 6 Move Generator 77
6.1 Computer Lexicon
6.2 Preliminaries of Heuri's Move Generator
6.3 The Move Generator
6.4 Results and conclusions of Heuri's Move Generator
Chapter 7 Move Evaluator 95
7.1 Introduction to Heuri
7.2 Probabilistic Heuristic of Heuri's first engine
7.3 Heuri's second Engine
7.4 Improved Heuristics (third and fourth engines)
7.5 Fishing
Chapter 8 Beyond the Heuristics 101
8.1 HeuriSamp
8.2 HeuriSim
8.3 Opponent Modeling
Chapter 9 Results 105
9.1 Data Findings
9.2 Match Results
Chapter 10 Conclusions and Future Work 119
10.1 Contributions of this thesis
10.2 Future work
10.3 Publications derived from this thesis
Appendix and *References* 123 and 125

Chapter 1.

Introduction

In 1956 John McCarthy conceived the term *Artificial Intelligence* (AI) when he held the first academic conference on the subject. But the journey to understand if machines can truly think began much before that. In Vannevar Bush's seminal work "As We May Think" [Bush45] he proposed a system which amplifies people's own knowledge and understanding. Alan Turing, five years later, wrote a paper on the notion of machines being able to simulate human beings and the ability to do intelligent things, such as playing Chess [Turing50].

Expectations, in the field of AI, seem to always go faster than reality. After decades of research, no computer has come close to passing the Turing Test (a model for measuring 'intelligence'); Expert Systems have grown but have not become as common as human experts; and while we've built software that can beat humans at some games, open ended games are still far from the mastery of computers. Is the problem simply that we haven't focused enough resources on basic research, or is the complexity of AI one that we haven't come to grasp yet? (And instead, like in the case of computer Chess, we focus on much more specialized problems rather than understanding the notion of 'understanding' in a problem domain.)

1.1 The Turing test

The Turing test is a central, long term goal for AI research – will we ever be able to build a computer that can sufficiently imitate a human to the point where a suspicious judge cannot tell the difference between human and machine? From its inception it has followed a path similar to much of the AI research. Initially it looked to be difficult but possible (once hardware technology reached a certain point), only to reveal itself to be far more complicated than initially thought with progress slowing to the point that some wonder if it will ever be reached. Despite decades of research and great technological advances the Turing test still sets a goal that AI researchers strive toward while finding along the way how much further we are from realizing it.

In 1950 English Mathematician Alan Turing published a paper entitled "Computing Machinery and Intelligence" which opened the doors to the field that would be called AI. This was years before the community adopted the term Artificial Intelligence as coined by John McCarthy [Turing50]. The paper itself began by posing the simple question, "Can machines think?" [RussellNorvig09]. Turing then went on to propose a method for evaluating whether machines can think, which came to be known as the Turing test. The test, or "Imitation Game" as it was called in the paper, was put forth as a simple test that could be used to prove that machines could think. The Turing test takes a simple pragmatic approach, assuming that a computer that is indistinguishable from an intelligent human actually has shown that machines can think.

The idea of such a long term, difficult problem was a key to defining the field of AI because it cuts to the heart of the matter, rather than solving a small problem it defines an end goal that can pull research down many paths.

1.2 Computer Chess

Chess has long been considered a game of intellect, and many pioneers of computing felt that a chess-playing machine would be the hallmark of true artificial intelligence. While the Turing Test is a grand challenge to ascertain machine intelligence, chess too is a good pursuit, one which fortunately has been 'solved' by AI researchers; producing programs which can defeat the world's best chess players. However, even the best game-playing machines still do not understand concepts of the game and merely rely on brute force approaches to play.

1.2.1 Origins of Computer Chess

Chess and intelligence have always been linked; the ability to play chess was even used as a valid question to ask during a Turing Test in Turing's original paper. Many people envisioned machines one day being capable of playing Chess, but it was Claude Shannon who first wrote a paper about developing a chess playing program [Shannon50].

Shannon's paper described two approaches to computer chess: Type-A programs, which would use pure brute force, examining thousands of moves and using a min-max search algorithm. Or, Type-B, programs which would use specialized heuristics and 'strategic' AI, examining only a few, key candidate moves. Initially Type-B (strategic) programs were favored over Type-A (brute force) because during the 50s and 60s computers were so limited. However, in 1973 the developers of the 'Chess' series of programs (which won the ACM computer chess championship 1970-72) switched their program over to Type-A The new program, dubbed 'Chess 4.0', went on to win a number of future ACM computer chess titles. This change was an unfortunate blow to those hoping of finding a better understanding of the game of chess through the development of Type-B programs.

There were several important factors in moving away from the debatably more intelligent design of a Type-B program to a Type-A. The first was simplicity. The speed of a machine has a direct correlation to a Type-A program's skill, so with the trend being machines getting faster every year it is easier to write a strong Type-A program and 'improve' a program by giving it more power through parallelization or specialized hardware, whereas a Type-B program would need to be taught new rules of thumb and strategies – regardless of how much new power was being fed to it. Also, there was the notion of predictability. The authors of 'Chess' have commented on the stress they felt during tournaments where their Type-B program would behave erratically in accordance to different hard-coded rules. To this day Type-A (brute force) programs are the strongest applications available. Intelligent Type-B programs exist, but it is easier to write Type-A programs and get exceptional play just off of computer speed. Grandmaster-level Type-B programs have yet to materialize since more research must be done in understanding and abstracting the game of chess into (even more) rules and heuristics.

1.2.2 Realization

Perhaps the best known Type-A program is IBM's Deep Blue. In 1997 Deep Blue challenged and defeated the then world chess champion Gary Kasparov. The 3.5/2.5 match win wasn't a decisive victory but with machines continually increasing in power, many feel the match was just a taste of things to come.

Few surprised by a computer beating a world chess champion. Scientist David G. Stoke explained this notion of expected computer superiority with: "Nowadays, few of us feel deeply threatened by a computer beating a world chess champion – any more than we do at a motorcycle beating an Olympic sprinter." Most of this sentiment is due to Deep Blue being a Type-A program. Deep Blue evaluated around 200 million positions a second and averaged 8-12 ply search depth. (Up to 40 under certain conditions.) Humans on the other hand are generally thought to examine near 50 moves to various depths. If Deep Blue were a Type-B program then perhaps the win would have been more interesting from the standpoint of machine intelligence. [McGuireSmith06]

1.3 Games

Arthur Samuel is one of the pioneers of artificial intelligence research. Together with Claude Shannon [Shannon50] and Alan Turing [Turing53], he laid the foundation for building high-performance game-playing programs. Samuel is best known for developing his checkers program. He consistently sold his work as research in machine learning. His papers describing the program and its learning capabilities are classics in the literature ([Samuel59],[Samuel67]). These papers are still frequently cited today, five decades since the original research was completed. There are few computing papers around today whose lifespan is 10 years, let alone 50.

It is important to recognize that building high-performance game-playing programs has been of enormous benefit to the respective game-playing communities. The technology has expanded human understanding of games, allowing us to explore more of the rich tapestry and intellectual challenges that games have to offer. Computers offer the key to answering some of the puzzling, unknown questions that have tantalized game fans. For example, computers have shown that the chess endgame of king and two bishops versus king and knight is generally a win, contrary to expert opinion [Thompson86]. In checkers, the famous 100-year position took a century of human analysis to "prove" a win; the checkers program *Chinook* takes a few seconds to prove the position is actually a draw (it is now called the 197-year position) [Lafferty97].

Games in AI play a crucial role; they are often used as a testbed for new search algorithms. Programs designed to play games usually make their decisions using searches in the problem *state space*, which is the set of all possible *states*, or possible configurations, of a game. Since games have been used as a testbed for search algorithms for almost as long as computing science has been around, most games used are generally well-understood domains. These days, it is rare to find a search problem that has not been explored in the context of a game, or a game which has not been investigated using search. Of course, a fringe benefit of working with games as a research domain is that games are great fun.

When research is done with search algorithms, games are used often as the domain for testing new algorithms or improvements to old algorithms. Games are used because they are often well-understood domains and relatively "real-world". Games also have a good built-in performance measure: a win.

1.4 About Scrabble

Scrabble is a popular board game played by millions of people around the world. It is a competitive outlet for those who delight in word play, and has brought enormous pleasure to generations of families since it was created by Alfred Butts in the 1930s. Players from across the world enjoy Scrabble as a pleasant family board game for two to four players. In this thesis we consider its tournament variant, which is a two player game. Competitors make plays by forming words on a 15 x 15 grid, abiding by constraints similar to those found in crossword puzzles. Each player has a rack of seven letter tiles that are randomly drawn from a bag that initially contains 100 tiles.

Achieving a high score requires a delicate balance between maximizing one's score on the present turn and managing one's rack in order to achieve high-scoring plays in the future. Because opponents' tiles are hidden and because tiles are drawn randomly from the bag on each turn, Scrabble is a stochastic partially observable game [RussellNorvig09]. This feature distinguishes Scrabble from games like chess and go, where both players can make decisions based on full knowledge of the state of the game. [RichardsAmir07]

The game of Scrabble is a good platform for testing Artificial Intelligence (AI) techniques. The game has an active competitive tournament scene, with national and international Scrabble associations and world championship tournaments in many languages (being English, French and Spanish the most common). Creating a program that simply plays legal moves is a significant challenge. Once the program plays legal moves, you find that it is too weak to challenge top humans, and nothing obvious will address the shortcomings. [Sheppard2002a]

Even though Scrabble is one of the most popular games in the world, with millions of game sets sold annually, little has been published in the computer literature about the game. In part, this is because the game-AI tradition focuses on deterministic games. Chess, checkers, reversi, awari, hex, go, renju, amazons, Pente, etc., dominate the game AI literature. Some of these games did not even *exist* when serious investigations into Scrabble began in the mid-1970s, yet the literature still emphasizes deterministic games. The fact that Scrabble has a significant random component makes it very interesting for AI research. [Sheppard2002b]

Similarly, the game-AI literature focuses on perfect-information games. The games listed above are all perfect-information games. Additionally, there is a computer literature for backgammon, which is a stochastic game of perfect information. We are starting to see some literature about games with private information, like poker and bridge, and this thesis will add to that trend.

Scrabble, in contrast with other games like chess, go, draughts, etc. is a game of imperfect information. Techniques of Artificial Intelligence have been applied to games of imperfect information, mainly card games such as Bridge and Poker ([Ginsberg99], [BillingsDavidsonSchaeffer02]). The theoretical support for the games of strategy goes back to the work of Von Neumann and Morgenstern [VonNewmanMorgenstern53].

1.5 Some Characteristics of Scrabble

Scrabble has a large state space. The state space is much bigger than Chess, and is even bigger than Go. The number of states is of the order of 2^{1000}. The number of legal moves is large, too. An average game position has about 800 moves, and positions can have over 8000 moves.

However, these imposing parameters of the state space are perhaps misleading, since expert Scrabble play is not predominantly concerned with exhaustively searching the state space. In part, this is because Scrabble has a random component, as the players draw tiles blindly. The random component gives the state space a high variance, so that the way that a move turns out depends on many imponderable factors. Because you cannot calculate the outcome of a move in most cases, in Scrabble evaluation takes precedence over search. [Sheppard2002b]

Move generation in Scrabble is a challenging task. To figure a rough order of magnitude for the task, on each turn you can place any subset of your 7 tiles in any order either horizontally or vertically starting at any 225 squares of the board. A rough order of magnitude calculation shows 2 * 225 * 7! = 2,268,000 potential words to check for legality. When playing Scrabble in Spanish, checking a move for legality involves looking up all words created or modified by a play in a Spanish lexicon containing 637,000 words from 2 through 15 letters.

1.6 Computer Scrabble

The game of Scrabble has only been tackled with success by programs or engines which rely on classical brute force Monte Carlo simulation, therefore falling into the Type-A(brute force) category. This thesis gives a novel, and powerful probabilistic heuristic to play Scrabble, leading to the construction of Heuri which is a Scrabble Type-B(strategic) engine capable of defeating a human Scrabble world Champion in a match. The most important match victory of Heuri was against the two times (2006,2008) Spanish Scrabble World Champion, Enric Hernández from Spain. Heuri defeated Enric 6-0 !

Heuri's performance versus a very strong Type A engine like Quackle (one of the best Scrabble engines in the world) is rather good, especially in Spanish. Despite Heuri's lack of simulation it has proven to be a strong computer Scrabble engine, thus giving motivation to do more heuristic research in Artificial Intelligence.

The thesis also presents a nice, novel and very fast move generator, which is very different from the standard move generators used by the strongest Scrabble engines. An experiment involving a rack with two blank tiles (a blank tile stands for any letter in the alphabet) indicates that, for such a rack, Heuri's move generator compared to move generators using a DAWG (*Directed Acyclic Word Graph*) structure is more than 30 times faster!

Although the engine presented in this thesis can be used for any language, once having the corresponding lexicon and alphabet, including the letter distribution, the main concern in this thesis is Spanish Scrabble; most of our experiments have been done for Spanish. Recently, English and French have been incorporated, mostly to compare the performance of our engine Heuri versus Quackle.

1.7 Objectives and expected contributions of this thesis

One of the objectives is to have a better understanding of the game of Scrabble. The current state-of-the-art Scrabble programs (Maven and Quackle) do not give a good understanding of the game, they are a bit like those Type-A chess programs that rely on brute force. This objective is similar to that of understanding better the game of chess through Type-B programs, which use 'strategic' AI and specialized heuristics, (see section 1.2.1 Origins of Computer Chess).

Another objective is to build a very strong Scrabble engine capable of defeating human Spanish Scrabble World Champions, and capable of defeating the state-of.the-art Scrabble programs in Spanish, English and French. The engine Quackle was chosen to be our rival, because it plays stronger than Maven, and because it has an open source code.

A third objective is to learn, from massive playing of the developed engine, word statistics (such as frequencies of bingos) and knowledge that can help Scrabble human players in their training.

Another possible objective is to obtain, from the construction of the required lexicon, knowledge or tools that can assist either to the human learning of Spanish or to Spanish computational linguistics.

1.8 Structure of this document

The structure of the dissertation is as follows.

Chapter 1 deals with the beginning of Artificial Intelligence and how games played an important role, specially Chess. It also gives an introduction to the game of Scrabble, mentions state-of-the-art computer Scrabble programs and gives some characteristics of our Scrabble engine *Heuri*.

Chapter 2 gives some important notions of game theory like game trees, games in extensive form, pure and mixed strategies and so on; it also studies some games like Penalty Kicks, Eight-Tile Partnership Dominoes and Two-Person Memory Game, from the game theory perspective. It also gives some history about engines that played Chess and Go (mainly Deep Blue and AlphaGo). It also contemplates other games like Gomoku, Checkers, Backgammon and Poker. Besides it deals with Fundamental Algorithms and techniques used in games and in many other applications, some of these algorithms are: Minimax Algorithm, Alpha-Beta Pruning Algorithm, Expectiminimax, Heuristics and Evaluation Functions, B* Algorithm, Monte-Carlo Tree Search, Artificial Neural Networks, Convolutional Neural Networks, Reinforcement learning and Q-learning. Finally, Chapter 2 covers the algorithms inside some state-of-the-art engines, mainly engines that play Go: AlphaGo, Alpha Go Zero and AlphaZero. AlphaZero also plays Chess and Shogi (Japanese Chess).

Chapter 3 deals with the preliminaries of the game of Scrabble like the Letter Distribution (it gives the Spanish and English letter distribution), it also tells a brief history about the origins of Scrabble and the study of the English letter distribution. In order to play Scrabble, we need to know some rules; these rules are also mentioned in this chapter. Finally we mention some differences between friendly games and competitive Scrabble games.

Chapter 4 treats the generation of valid moves based on the data structure DAWG (Directed Acyclic Word Graph) and also the variant GADDAG, which duplicates the speed of move generation. It explains and gives examples of a DAWG and a GADDAG. It also contemplates the state-of-the-art Scrabble engines Maven and Quackle. It gives more details about Quackle, since it is our rival engine and it has an open source code. Finally it considers Monte Carlo simulation in Scrabble, Opponent modeling and the Endgame phase in Scrabble.

It is in Chapter 5 where the work concerning our contributions starts. It begins with the construction of a Spanish lexicon; the official Scrabble dictionary in Spanish (*Diccionario de la Real Academia Española* DRAE) has 88455 lemmas (entries); to construct the complete lexicon the following two subproblems had to be solved: 1)Conjugation of verbs, 2) Plurals of nouns and adjectives. Once the Spanish lexicon is completed, the first Spanish Scrabble program, which played with the FISE (*Federacion Internacional de Scrabble en Español*) rules, was constructed. This engine did not play so well, since it used a greedy strategy (always playing the highest scoring move on the board). This is not a good strategy, since no value is assigned to the remaining tiles on the rack after a move. Finally some words of motivation and explanation about our human-like Scrabble engine, *Heuri*, were written.

In Chapter 6 detailed explanation about our move generator is given, lots of descriptions with examples are used. This generator originated the publication of a paper shortly named: "The Anagram Method" [GonzalezAlquezar18]; this method is similar to the way Scrabble players look for a move, searching for *anagrams* (permutation of letters that form a valid word) and a spot to play on the board. Comparisons of the Anagram Method with other methods like the DAWG (Direct Acyclic Word Graph) and GADDAG, used in other engines like Quackle, are given.

In Chapter 7 we are concerned with the evaluations of moves when playing Scrabble; the quality of your game depends on the decisions you make in each situation, in other words, in deciding what move should be played given a certain board and a rack with tiles. This decision was made by Heuri trying several heuristics which ended up with the construction of several engines. This chapter gives the explanation of the heuristics used in the first, second and third Heuri engines. These engines produced publications of several papers; an important one is: "Heuristics and Fishing in Scrabble" [GonzalezAlquezar12]. All these heuristics do not use forward looking, they are static evaluators.

Chapter 8 deals with additional heuristics. To improve , even more, the quality of play of the Heuri engine look ahead had to be employed. This chapter describes the HeuriSamp engine which evaluates a two-ply search; this gave a defense value *d*. The chapter also explains the HeuriSim engine which uses a 3-ply adversarial search tree; it contemplates the best first moves (according to Heuri third engine heuristic) from Player 1, then some replies to these moves (Player 2 moves) and then some replies to these replies (Player 1 moves). Finally to improve these engines, Opponent Modeling is used; this technique makes predictions on some of the opponents' tiles based on the last play made by the opponent.

Chapter 9 presents results obtained by playing thousands of Heuri vs Heuri games, collecting important information like the frequency of certain words, the winning percentage of a player, a list of the most frequently played *Bingos* (words that use all 7 tiles of a player's rack). It also gives results of matches played by Heuri against humans, and other match results of different Heuri engines when played against the Quackle engine.

Chapter 10 mentions the achievements made in the thesis, some of the objectives of the thesis were accomplished entirely and others partially. From here, work for the future is envisaged.

Chapter 2.

Games and their Artificial Intelligence Fundamental Algorithms

2.1 Game Theory

Game Theory is a branch of mathematics dealing with situations of conflict (strategic situations), where a result of a participant depends on choices made by himself and others. Sometimes, it is also called the theory of rational behavior. Apart from computer science, it has been applied in the fields of sociology, economics, military (historically earlier).

Two important historical works are :

1) Émil Borel, Applications for random games (fr. Applications aux Jeux de Hasard), 1938.
2) John von Neuman and Oskar Morgenstern, Theory of Games and Economic Behavior, 1944.

2.1.1 Some Notions

Game

A situation of conflict, where:

- at least two players can be indicated,
- every player has a certain number of possible *strategies* to choose from (a strategy precisely defines the way the game shall be played by the player),
- result of the game is a direct consequence of the combination of strategies chosen by players.

Game Trees

For a certain state s assume there exists n possible choices (moves, manipulations, actions): $a_1, a_2, \ldots, a_n$ causing new states to arise from s, respectively: $s_1, s_2, \ldots, s_n$. For each of those states there again exist further possible choices. By continuing this procedure a *tree structure* arises naturally.

Game in extensive form

One may think that a mathematical definition of a game (in extensive-form) is not necessary but, even if our interest focuses on algorithms in Artificial Intelligence, the fundamental theorem of game theory, existence of optimal strategies in 2-person zero-sum games, is used in AI; also definitions and results of games of imperfect and perfect information are certainly relevant.

An n-person *game in extensive form* is

(a) a finite directed tree Γ with a root ;
(b) a *(payoff) function*, that assigns to each terminal vertex t an element of R^n whose i-th component is the *payoff to player i at t* ;
(c) a partition of the set of non-terminal vertices into $n+1$ sets $S_0, \ldots, S_n$;

(d) for each vertex v in S_0 , a probability distribution on the set of immediate succesors ;
(e) for $i=1, \ldots , n$ a partition of S_i into subsets S_i^j, called *information sets*, such that, if $v \in S_i^j$, no succesor of v belongs to S_i^j ;
(f) for each S_i^j an index set I_i^j and, for every $v \in S_i^j$, a bijection from I_i^j onto the set of immediate succesors of v .

The vertices of S_0 (see (d)) correspond to chance moves and, if $1 \leq i \leq n$,the vertices of S_i correspond to player i. In some games, such as Checkers, Chess and Go, S_0 is empty, that is, there are no chance moves.

If for every terminal point t , the sum of the players' payoffs at t is zero, the game is a *zero-sum* game.

In zero-sum 2-person games in extensive form it is customary to write at each terminal vertex t only the payoff to the first player.

A game is said to have *perfect information* if every information set S_i^j with $i > 0$ consists of a single vertex. These are games in which each player knows every move that has been made so far.

A game is of *perfect recall* if no player forgets what he did previously. Roughly partnership games, as Bridge and Dominoes, are games of imperfect recall.

Checkers, Chess, Go, Backgammon, Einstein Würfelt Niecht ! and The Memory game are some games of perfect information and perfect recall, the last 3 involving chance moves; Scrabble, Poker and Penalty Kicks are examples of games with imperfect information and perfect recall; Bridge and Partnership Dominoes are games of imperfect information and imperfect recall.

The *endgame* is the final stage of a game such as chess, checkers or Scrabble, when few pieces or tiles remain.

A *strategy* is a complete set of decisions (about choices or moves) that a player has to make for all possible states the game can reach. More formally we have:

Def. Let Γ be an n-person game (in extensive form).

A *pure strategy* σ_i *of player i* is a function that assigns to each information set S_i^j an element of I_i^j .
The set of pure strategies of player i is denoted by Σ_i .
The *normal form* of the game Γ is the function π: $\Sigma_1 \times \Sigma_2 \times \ldots \times \Sigma_n \rightarrow R^n$ with components $\pi_1, \ldots, \pi_n$ where $\pi_i(\sigma_i, \ldots, \sigma_n)$ is the expected pay to player i if the strategies $\sigma_i, \ldots, \sigma_n$ are followed.

A *mixed strategy for player i* is a probability distribution on Σ_i ; it can be thought of as a convex combination of the pure strategies of player i .

The payoff function π can be extended uniquely, in a natural way, to a function $\pi(\mu_i, \ldots, \mu_n)$ where μ_i is now a mixed strategy of player i .

In a zero-sum 2-person game there is a pair of optimal mixed strategies, (μ_1, μ_2) and $\pi(\mu_1, \mu_2)$, a well defined real number, is the *value* of the game.

We will be mainly interested in zero-sum 2-person games.

If the game has perfect information the optimal strategies can be chosen to be pure.
Example (game equivalent to Tic-Tac-Toe). Players I and II alternate picking unchosen integers of the interval [-4, 4]. The first player having 3 integers whose sum is 0 wins. An optimal pure strategy for I: Block; else choose 0; else choose odd; else choose even. An optimal pure strategy for II: Complete else Block; else choose 0; else if

I has {n, -n}, with n odd, choose even; else if I has {m,n}, with m odd and {|m|, |m-n| }={1,3} choose -m; else if I has {m,n}, with { |m|, |n| } = {2,4} choose (-m-n)/2; else choose odd; else choose even.

// Complete (resp. Block) means: if you have (resp. your opponent has) a and b ($a \neq b$) and -a-b is an unchosen integer of [-4, 4] then choose -a-b.

A *behavioral strategy for player i* is a probability distribution on I_i^j for every information set S_i^j of player *i* .
A 2-person zero-sum-game with perfect recall has a pair of optimal behavior strategies.

A 2-person zero-sum matrix game is defined by a real *m* x *n* matrix $M = (m_{ij})$ as follows: the row player I chooses a row *i* and the column player II chooses a column *j;* the payoff to I is m_{ij} .

2.1.2 Penalty Kicks

We consider a matrix game arising from penalty shootouts. The 68 penalty kicks of the World Cup in Russia 2018 were studied [GG]. Together with a triplication technique these were extrapolated to form 204 penalties. The following 3 x 3 matrix was obtained:

Kicker\Goalie	Natural (N)	Center (C)	Unnatural (U)
natural (n)	0.71	0.86	0.96
center (c)	0.87	0.41	0.84
unnatural (u)	0.87	0.68	0.36

This game , drawn as a tree , has a root R, belonging to I, 3 successors A_1, A_2, A_3 of R forming an information set S^1_{II} of II; the successors of the A_i's are 9 terminal nodes (with corresponding payoffs).

The value of the model turned out to be v = 0.77073 (indicating that 77.073% of the penalties should be goals with optimal performance). The optimal mixed strategy for the kicker was:

Kicker:
With probability n=0.62202 shoot towards the natural side (right of the goalie if the kicker is right-footed)
With probability c=0.07462 shoot towards the center.
With probability u=0.30336 shoot towards the unnatural side (left of the goalie if the kicker is right-footed).

The optimal mixed strategy for the goalie was:

Goalie:
Dive towards his natural side (to the right if the kicker is right-footed) with probability N= 0.67926 .
Dive towards his unnatural side (to the left if the kicker is right-footed) with probability U=0.11503 .
Stay in the Center with probability C=0.20572 .

A similar analysis was made for the World Cup Mexico 1986, and also an experimental finer study during 1996-1998; in the latter finer study 12 pure strategies were considered for the kicker and 5 for the goalie. Optimal (mixed) strategies and the value of the game were calculated. See [GonzalezGonzalez18].

As a result from this work a paper titled *Optimal Strategies for Penalty Kicks* was published see [GonzalezGonzalez18].

2.1.3 Eight-Tile Partnership Dominoes

Consider the following eight-tile model of partnership dominoes. Team A-A' plays against team B-B' and the eight tiles [0|0], [0|1], [1|1], [1|3], [3|3], [3|2], [2|2], and [2|1] are dealt "at random" two tiles for each agent. The order of play is A, B, A', B' . The winner is A-A' (resp. B-B') if A or A' (resp. B or B') is the first player that plays out all his tiles. Points of the loser are not counted.

This game is of imperfect information and imperfect recall. It has been solved in [Trigo77] . It has no solution using only behavioral strategies.

Part of the optimal strategy of A-A' is as follows:
A-A' constructs randomly before the game starts, a regular tournament ([J. W. Moon, Topics on Tournaments in Graph Theory, 4 Exercise 7]) with four vertices [0|0], [1|1], [2|2], [3|3]. This is a complete digraph whose outdegrees are 1,1,2,2. If A has{ [p|p], [q|q] }he plays [p|p] iff there is an edge from [p|p] to [q|q] .

Part of the optimal strategy of B-B' is as follows:
If after a [c|a][a|a][a|b] layout where A played [a|a] and B played [a|b], B' has [b|b] and [c|d] ,with {a, b, c, d}={0,1,2,3}, then B' plays [b|b] (resp. [c|d]) with probability 1/3 (resp. 2/3).

2.1.4 Two-Person Memory Game

In some games (see for example "Games of Exhaustion" in [Owen 5.2]) at terminal vertices there is either a true payoff or the obligation to play another (usually simpler) game.

Example Two-Person Memory Game with Perfect Recall.

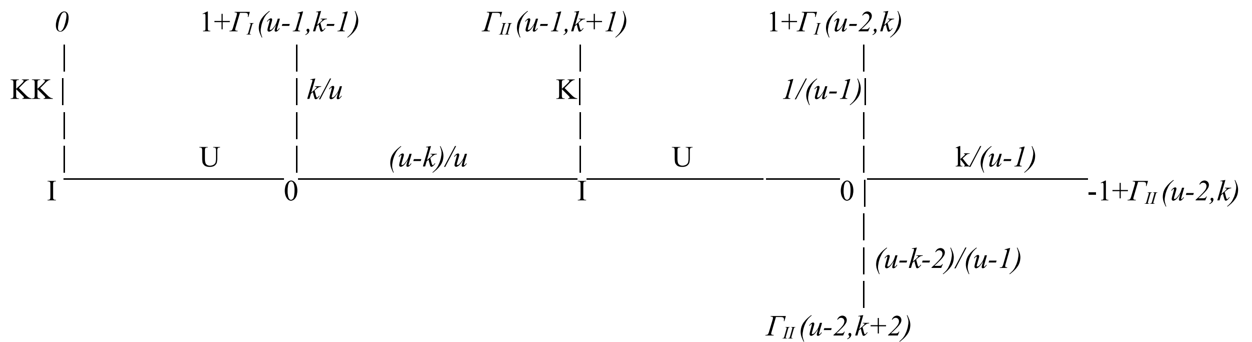
There are u unknown cards face down and k known cards also face down on the table. They form n different pairs of cards. The player in turn flips a first card and then flips a second one. If they form a pair they are removed from the table into the posession of the player who flipped them who gets another turn. Otherwise they are flipped back and the turn passes to the other player. The game is over when all cards are removed from the table and the winner is the player making more pairs whose payoff is the number of pairs he made minus the number of pairs his opponent made.

We denote this game by $\Gamma_I(u,k)$ (resp. $\Gamma_{II}(u,k)$) if I (resp. II) is the player who starts; $\varepsilon + \Gamma_i(u,k)$ is the game $\Gamma_i(u,k)$ with all payoffs at terminal vertices augmented by ε $(\varepsilon =1,-1)$ $(i=I, II)$.
It is a game with chance moves of perfect information and perfect recall. It was solved by Zwick and Paterson [ZP, Theoretical Computer Science 1993].

At every turn the three sensible strategies are:

KK) Flip two known cards; here we agree that the payoff is zero.
UU) Flip two unknown cards.
UK) Flip first one unknown; if it makes a pair with a known card make that pair. Othewise flip a (second) known card.

We give an extensive-form of the game $\Gamma_I(u,k)$. Notice that most terminal nodes have an obligation to play a simpler game rather than a true payoff . Non-terminal nodes with label 0 are chance moves.



(If $k < 2$ prune the branch KK; if $k=0$ prune the branch K).
Now we give an optimal strategy for the game assuming $0 < k < u$

If 4 divides u-k or $(u,k)=(11,1)$ play UK
else **if** $k > [(u+1) / 2]$ play KK
else play UU .

2.2 History of some Games in AI

2.2.1 Chess and Computer Chess

Chess has received by far the largest share of attention in game playing. Chess has long been considered a game of intellect, and many pioneers of computing felt that a chess-playing machine would be the hallmark of true artificial intelligence. While the Turing Test is a grand challenge to ascertain machine intelligence, chess too is a good pursuit, one which fortunately has been ‘solved’ by AI researchers; producing programs which can defeat the world’s best chess players.

Recently, two chess engines, starting from the basic rules, have incredibly improved their level, by learning with self-play using Deep Reinforcement Learning and other AI. techniques. AlphaZero [AlphaZero17] and LeelaZero [LCZero18] achieved amazing results; both engines defeated Stockfish (regarded as the strongest engine, before they arrived).

Early times of Computer Chess

Progress beyond a mediocre level was initially very slow: some programs in the early 1970s became extremely complicated, with various kinds of tricks for eliminating some branches of search, generating plausible moves, and so on, but the programs that won the ACM North American Computer Chess Championships (initiated in 1970) tended to use straightforward alphabeta search, augmented with book openings and infallible endgame algorithms. This offers an interesting example of how high performance requires a hybrid decision-making architecture to implement the agent function.

The first real jump in performance came not from better algorithms or evaluation functions, but from hardware. Belle, the first special-purpose chess computer (Condon and Thompson, 1982), used custom integrated circuits to implement move generation and position evaluation, enabling it to search several million positions to make a single move. Belle's rating was around 2250, on a scale where beginning humans are 1000 and the world champion around 2750; it became the first master-level program.

The HITECH system, also a special-purpose computer, was designed by former world correspondence champion Hans Berliner and his student Carl Ebeling to allow rapid calculation of very sophisticated evaluation functions. Generating about 10 million positions per move and using probably the most accurate evaluation of positions yet developed, HITECH became computer world champion in 1985, and was the first program to defeat a human grandmaster, Arnold Denker, in 1987. At the time it ranked among the top 800 human players in the world.

Deep Blue

The project was started as ChipTest at Carnegie Mellon University by Feng-hsiung Hsu, followed by its successor, Deep Thought. After their graduation from Carnegie Mellon, Hsu, Thomas Anantharaman, and Murray Campbell from the Deep Thought team were hired by IBM Research to continue their quest to build a chess machine that could defeat the world champion. Hsu and Campbell joined IBM in autumn 1989, with Anantharaman following later. Anantharaman subsequently left IBM for Wall Street and Arthur Joseph Hoane joined the team to perform programming tasks. Jerry Brody, a long-time employee of IBM Research, was recruited for the team in 1990. The team was managed first by Randy Moulic, followed by Chung-Jen (C J) Tan. [DeepBlue02]

The system Deep Thought 2 was sponsored by IBM, which hired part of the team that built the Deep Thought system at Carnegie Mellon University. Although Deep Thought 2 uses a simple evaluation function, it examined about half a billion positions per move, allowing it to reach depth 10 or 11, with a special provision to follow lines of forced moves still further (it once found a 37-move checkmate). In February 1993, Deep Thought 2 competed against the Danish Olympic team and won, 3-1, beating one grandmaster and drawing against another. Its FIDE rating is around 2600, placing it among the top 100 human players.

After Deep Thought's 1989 match against Kasparov, IBM held a contest to rename the chess machine and it became "Deep Blue", a play on IBM's nickname, "Big Blue". After a scaled-down version of Deep Blue, Deep Blue Jr., played Grandmaster Joel Benjamin; Hsu and Campbell decided that Benjamin was the expert they were looking for to develop Deep Blue's opening book, and Benjamin was signed by IBM Research to assist with the preparations for Deep Blue's matches against Garry Kasparov.

Deep Blue and Kasparov played each other on two occasions. The first match began on 10 February 1996, in which Deep Blue became the first machine to win a chess game against a reigning world champion (Garry Kasparov) under regular time controls. However, Kasparov won three and drew two of the following five games, beating Deep Blue by a score of 4–2 (wins count 1 point, draws count ½ point). The match concluded on 17 February 1996.

Deep Blue was then heavily upgraded (unofficially nicknamed "Deeper Blue") and played Kasparov again in May 1997, winning the six-game rematch 3½–2½, ending on 11 May. Deep Blue won the deciding game six after Kasparov made a mistake in the opening, becoming the first computer system to defeat a reigning world champion in a match under standard chess tournament time controls.

The system derived its playing strength mainly from brute force computing power. It was a massively parallel, RS/6000 SP Thin P2SC-based system with 30 nodes, with each node containing a 120 Mhz P2SC microprocessor, enhanced with 480 special purpose VLSI chess chips. Its chess playing program was written in C and ran under the AIX operating system. It was capable of evaluating 200 million positions per second, twice as fast as the 1996 version. In June 1997, Deep Blue was the 259th most powerful supercomputer according to the TOP 500 list, achieving 11.38 GFLOPS on the High-Performance LINPACK benchmark.

The Deep Blue chess computer that defeated Kasparov in 1997 would typically search to a depth of between six and eight moves to a maximum of twenty or even more moves in some situations. David Levy and Monty Newborn estimate that one additional ply (half-move) increases the playing strength 50 to 70 Elo points.

Deep Blue's evaluation function was initially written in a generalized form, with many to-be-determined parameters (e.g. how important is a safe king position compared to a space advantage in the center, etc.). The optimal values for these parameters were then determined by the system itself, by analyzing thousands of master games. The evaluation function had been split into 8,000 parts, many of them designed for special positions. In the opening book there were over 4,000 positions and 700,000 grandmaster games. The endgame database contained many six piece endgames and five or fewer piece positions. Before the second match, the chess knowledge of the program was fine tuned by grandmaster Joel Benjamin. The opening library was provided by grandmasters Miguel Illescas, John Fedorowicz, and Nick de Firmian. When Kasparov requested that he be allowed to study other games that Deep Blue had played so as to better understand his opponent, IBM refused. However, Kasparov did study many popular PC games to become familiar with computer game play in general.

Perhaps the first best known Type-A program is IBM’s Deep Blue. In 1997 Deep Blue challenged and defeated the then world chess champion Gary Kasparov. The 3.5/2.5 match win wasn’t a decisive victory but with machines continually increasing in power, many feel the match was just a taste of things to come.

Few surprised by a computer beating a world chess champion. Scientist David G. Stoke explained this notion of expected computer superiority with: “Nowadays, few of us feel deeply threatened by a computer beating a world chess champion – any more than we do at a motorcycle beating an Olympic sprinter.” Most of this sentiment is due to Deep Blue being a Type-A program. Deep Blue evaluated around 200 million positions a second and averaged 8-12 ply search depth. (Up to 40 under certain conditions.) [CambleHsuHoane02] Humans on the other hand are generally thought to examine near 50 moves to various depths. If Deep Blue were a Type-B program then perhaps the win would have been more interesting from the standpoint of machine intelligence. [McGuireSmith06]

Deep Blue employed custom VLSI chips to execute the alpha-beta search algorithm in parallel, an example of GOFAI (Good Old-Fashioned Artificial Intelligence) rather than of deep learning which would come a decade later. It was a brute force approach, and one of its developers even denied that it was artificial intelligence at all.

Deep Blue, with its capability of evaluating 200 million positions per second, was the fastest computer to face a world chess champion. Today, in computer-chess research and matches of world-class players against computers, the focus of play has often shifted to software chess programs, rather than using dedicated chess hardware. Chess programs like Houdini, Rybka, Deep Fritz or Deep Junior are more efficient than the programs during Deep Blue's era. In a November 2006 match between Deep Fritz and world chess champion Vladimir Kramnik, the program ran on a computer system containing a dual-core Intel Xeon 5160 CPU, capable of evaluating only 8 million positions per second, but searching to an average depth of 17 to 18 plies in the middlegame thanks to heuristics; it won 4–2.[WikiDeepBlue]

For more recent computer chess accomplishments see 2.4.1.3 AlphaZero, 2.4.2.1 Stockfish and 2.4.2.2 Leela Chess Zero.

2.2.2 Go and Computer Go

The game of Go is a board game for two players, in which the goal is to gain more territory and capture more stones than the adversary. It was invented in China more than 2500 years ago and is perhaps the oldest board game played to the present day. There are over 45 million people who know the rules and over 20 million current players must of them from East Asia.

The rules are simple, but Go is very complex. The branching factor is approximately 200 and the depth is roughly 120. A lower bound of the number of legal positions is 10^{170} .

Despite its popularity in East Asia, Go spread slowly to the rest of the world. It did not start to become popular in the West till the end of the 19th century when Korschell wrote on the ancient Han Chinese game. Edward Lasker learned the game in Berlin and when he moved to New York founded the New York Go Club with Arthur Smith who published *The Game of Go* in 1908. In 1934 Lasker published *Go and Gomoku*. World War II put a stop to Go activity, but after the war, Go continued to spread.

The first Go program was probably written by Albert Zobrist in 1968 as part of his thesis on pattern recognition. It introduced an Influence function to estimate territory and Zobrist hashing to detect ko. It could just beat a beginner.

The first computer-computer match was between the programs written by Jon Diamond (Institute of Computer Science, London University) and Jack Davies (University of Cambridge) in 1973 - the game was unfinished and no record of it has been found. Jon's program was probably the first to use the alpha-beta search algorithm and also beat a beginner. In strength it was about 20-25 kyu.

GNU Go was published in 1989 as the first open source program and a version for the Nintendo Games Machine by Allan Scarff sold 140,000 copies.

In 1998, very strong players were still able to beat programs while giving handicaps of 25–30 stones, an enormous handicap that few human players would ever take. There was also a case in the 1994 World Computer Go Championship where the winning program, Go Intellect, lost all 3 games against the youth players while receiving a 15-stone handicap. In general, players who understood and exploited a program's weaknesses could win giving much larger handicaps than typical players.

The Computer Go Olympiad, organised by the International Computer Games Association, was started in 1989 for 9x9 boards and in 2000 for 19x19, with the initial tournaments both being held in London and won by Dragon Go (9x9) and Goemate (19x19).

WinHonte in 2005 appears to be the first program using neural networks, although Allan Scarff was working on an Acolyte Artificial Neural Net System when he died.

In 2006, advances in the strength of Go programs were still being made, though the rate of advance had slowed. Processor speeds were continuing to double every two years in accordance with Moore's Law, but this did not help, as the algorithms used by the best programs did not scale well, if at all. However, in this year Kocsis and Szepesvári published their seminal paper *Bandit based Monte-Carlo Planning*. This describes an algorithm, now known as Monte-Carlo Tree Search (MCTS), that was effective for computer Go (in fact a French team was working on a closely-related algorithm at the same time). This not only led to a rapid advance in the strength of Go programs over the next few years, it allowed them to use a method that did scale well, so now Moore's Law was working with the programmers again. Crazy Stone used MCTS in winning the gold medal on a 9x9 board at the Computer Olympiad.

In 2008 MoGo, developed by the French team mentioned above, beat an 8-dan professional in a 9-stone-handicap 19×19 game. It was running on an 800-node supercomputer. He estimated the playing strength of Mogo as being in the range of 2–3 amateur dan. In the same year the program Crazy Stone running on an 8-core personal computer won against a 4-dan professional, receiving a handicap of eight stones.

Zen playing on the KGS Go server in 2010 achieved a rating of 3-dan, playing 19×19 games against human opponents. [The KGS rating scale is slightly weaker than the European rating scale, close to the American scale, and rather stronger than the Japanese scale.] MoGo and Many Faces of Go beat professionals taking a 7 stone handicap.

In March 2012 Zen beat top professional Takemiya Masaki 9p at 5 stones by eleven points, followed by a stunning twenty point win at a 4 stone handicap. Takemiya remarked "I had no idea that computer go had come this far." It also reached the rank of 6 dan on the KGS Go Server playing games of 15 seconds per move. However, it's not clear how seriously professionals have been taking these exhibition matches.

At the 27th Annual Conference of the Japanese Society for Artificial Intelligence in June 2013, Zen defeated another top professional with a 3 stone handicap with a time setting of 60 minutes plus 30 seconds byoyomi. In March Crazy Stone beat Yoshio Ishida with four handicap stones.

In 2014 for the *codecentric go challenge* a best-of-five match was played between Crazy Stone and eleven-times German Go champion Franz-Jozef Dickhut, 6 dan amateur, without a handicap. Dickhut won as was expected by most observers and himself before the match. However Crazy Stone won the first game by 1.5 points, which was a resounding mark that the top programs have reached top amateur level.

This was reprised in October 2015, this time with Zen playing and Dickhut won again 3-1 with Zen winning the first game, again by 1.5 points.

Zen has been champion of the Computer Olympiad from 2011 to 2015 in all board sizes, but it should be noted that Crazy Stone did not take part.

In January 2016 both Facebook and Google DeepMind publish papers about their programs. DeepMind reports that AlphaGo [AlphaGo16] beat the European Champion and Chinese professional Fan Hui by five games to none in an even game match, which took place in October 2015. A first for a computer program! AlphaGo used two types of Convolutional Neural Networks together with Monte-Carlo Tree Search.

In March 2016 AlphaGo beats the top 9 dan Korean professional Lee Sedol 4:1 in a five game match in Seoul to world-wide publicity.[AlphaGovsLeeSedol16]

January 2017 provides another land-mark, with AlphaGo playing anonymously as *Master* on several Oriental servers against the very top professionals scoring a resounding 60 wins and no defeats, albeit with short time-limits. So there is no longer any doubt that computers can now play Go better than humans.

March 2017 provided a final match for AlphaGo Master, being retired from competition after beating the acknowledged world's best player, the Chinese *Ke Jie*, 3-0.

In October 2017 it was announced that AlphaGo Zero, armed with just the rules, had in 40 days become even better at Go than the original AlphaGo, without the help of game records!

In January 2018 AlphaZero, an engine device to play 3 games (Go,Chess and Shogi), beat 60-40 AlphaGo Zero.

2.2.3 EinStein würfelt nicht !

EinStein würfelt nicht ! (EWN) is a game, invented by Ingo Althöfer (2005) , played on a 5x5 board with 6 stones for each player and a die. Player I (resp. II) places his six stones, numbered from 1 to 6, on the squares at taxi distance less than 3 from the NW (resp. SE) corner, see Fig. 2.1 below. The players take turns rolling the die and moving his matching stone or, if it is no longer on the board, a stone whose number is next-higher or next-lower to the rolled number. Player I (resp. II) may move his stone one square to the East, South, or SW (resp. West, North, or NE).

Figure 2.1. Initial position of the EWN game.

If there is a stone (of I or II) that lies in the target square it is removed from the board. Player I (resp. II) wins if a stone of I (resp. II) reaches the SW (resp. NE) corner, or if all the stones of II (resp. I) are removed from the board.

Reasonable evaluation functions can be found for this game and, indeed there are some strong minimax-based programs for EWN. There is also an MCTS program to play the game [Lorentz11].

In Bonnet, Viennot [ACG17] exact solutions to some instances of the game, on smaller boards, are computed.

Also in [Analytical Sol EWN with one stone] the version of the game in which the players start with only one stone is solved for any board size. This version may be compared with King Race [Gonzalez05].

2.2.4 Gomoku

The m,n,k-game is played on an m x n board in which two players allternate turns placing a stone on an empty intersection. The winner is the first player who forms k consecutive stones horizontally, vertically or diagonally, Tic-tac-toe is the 3, 3, 3 game and free-style Gomoku is the 15, 15, 5 game.

It is known that the second player does not win the m, n, k-game under optimal play (using strategy-stealing).

In 1994 L. Victor Allis showed that the first player wins , with optimal play,in Gomoku using proof-number search and dependency-based search.

AI methods have been used to see their performance in Gomoku.
In [ADP with MCTS Algorithm for Gomoku] the following algorithm is used.

1. From a neural network trained by Adaptive Dynamic Programming 5 Candidate moves C_i with their winning probabilities w_{ai} (i = 1,...,5) are obtained.

2. For i = 1,...,5 take C_i as the root of a MCTS and obtain a winning probability w_{mi} .

3. Solve the rectangular 2 x 5 matrix game with matrix

	C_1	C_2	C_3	C_4	C_5
ADP	w_{a1}	w_{a2}	w_{a3}	w_{a4}	w_{a5}
MCTS	w_{m1}	w_{m2}	w_{m3}	w_{m4}	w_{m5}

If, say, $\lambda C_1 + (1 - \lambda)C_2$ is optimal in the matrix game then C_1 (resp. C_2) is played with probability λ (resp, $1 - \lambda$).

ADP is used as follows:
The current board state $x(t)$ is fed forward to the Action Selection, which generates the control action $u(t)$. Under the action $u(t)$ we obtain the next step transition state $x(t+1)$ which is fedforward to the function r which produces a reward $r(x(t+1))$, The critic network is used to estimate the cost function V. Then the reward $r(x(t+1))$, the estimate $V(t)$ and the estimate $V(t+1)$ are used to update the weights of the critic network.

2.2.5 Checkers or Draughts

Beginning in 1952, Arthur Samuel of IBM, working in his spare time, developed a checkers program that learned its own evaluation function by playing itself thousands of times. Samuel's program began as a novice, but after only a few days' self-play was able to compete on equal terms in some very strong human tournaments. When one considers that Samuel's computing equipment (an IBM 704) had 10,000 words of main memory, magnetic tape for long-term storage, and a cycle time of almost a millisecond, this remains one of the great feats of AI.

Few other people attempted to do better until Jonathan Schaeffer and colleagues developed Chinook, which runs on ordinary computers using alpha-beta search, but uses several techniques, including perfect solution databases for all six-piece positions, that make its endgame play devastating. Chinook won the 1992 U.S. Open, and became the first program to officially challenge for a real world championship. It then ran up against a problem, in the form of Marion Tinsley. Dr. Tinsley had been world champion for over 40 years, losing only three games in all that time. In the first match against Chinook, Tinsley suffered his fourth and fifth losses, but won the match 21.5-18.5. More recently, the world championship match in August 1994 between Tinsley and Chinook ended prematurely when Tinsley had to withdraw for health reasons. Chinook became the official world champion.

This 8×8 variant of checkers was **solved** on April 29, 2007 by the team of Jonathan Schaeffer [Schaeffer et al 07]. From the standard starting position, both players can guarantee a draw with perfect play.[Schaeffer et al 07] Checkers is the largest game that has been solved to date, with a search space of 5×10^{20}. The number of calculations involved was 10^{14}, which were done over a period of 18 years. The process involved from 200 desktop computers at its peak down to around 50.

The crucial part of Schaeffer's computer proof involved playing out every possible endgame involving fewer than 10 pieces. The result is an endgame database of 39 trillion positions. By contrast, there are only 19 different opening moves in checkers. Schaeffer's proof shows that each of these leads to a draw in the endgame database, providing neither player makes a mistake.

Schaeffer was able to get this result by searching only a subset of board positions rather than all of them, since some of them can be considered equivalent. He carried out a mere 10^{14} calculations to complete the proof in under two decades. Schaeffer also released an updated version of Chinook [Chinook07]

2.2.6 Backgammon

The inclusion of uncertainty from dice rolls makes search an expensive luxury in backgammon. The first program to make a serious impact, BKG, used only a one-ply search but a very complicated evaluation function. In an informal match in 1980, it defeated the human world champion 5-1, but was quite lucky with the dice. Generally, it plays at a strong amateur level.

In 1992 Gerry Tesauro developed TD-Gammon, a computer backgammon program. Its name comes from the fact that it is an artificial neural net trained by a form of temporal-difference learning, specifically TD-lambda.

TD-Gammon was reliably ranked among the top three players in the world. It explored strategies that humans had not pursued and led to advances in the theory of correct backgammon play.

During play, TD-Gammon examines on each turn all possible legal moves and all their possible responses (two-ply look ahead), feeds each resulting board position into its evaluation function, and chooses the move that leads to the board position that got the highest score. In this respect, TD-Gammon is no different than almost any other computer board-game program. TD-Gammon's innovation was in how it learned its evaluation function.

TD-Gammon's learning algorithm consists of updating the weights in its neural net after each turn to reduce the difference between its evaluation of previous turns' board positions and its evaluation of the present turn's board position—hence “temporal-difference learning". The score of any board position is a set of four numbers reflecting the program's estimate of the likelihood of each possible game result: White wins normally, Black wins normally, White wins a gammon, Black wins a gammon. For the final board position of the game, the algorithm compares with the actual result of the game rather than its own evaluation of the board position. [Tesauro95]

2.2.7 Poker

Poker is a family of imperfect information games involving Bluffing. In 1950 H. Kuhn proposed and solved a 2-person 0-sum simplified form of the game in which the deck consisted of three cards (K, Q and J) where each of the two players was dealt one card, to be seen only by him/her, and they decided, in a betting round, whether to pass, bet, call or raise.

A common version of the game, nowadays, is Heads-Up Limit Texas Hold´em poker (HULHE). It has approximately 3.16×10^{17} states. It is played as follows:

From the usual 52-card deck two cards are initially dealt face down to each player;, three aditional cards are dealt face up in a first round, one aditional card is dealt face up in a second round and one last card is dealt face up in a third round. Before each round and after the third round the two players have a decision to make: pass, call, bet or raise.

In HULHE, but not in HUNL (the no-limit version), there are fixed bet sizes; HULHE has been solved in [Bowling et al 17] using CFR^+, an iterative algorithm that computes succesive approximations to a Nash equilibrium,which is an improved form of CFR (*Counterfactual Regret Minimization*).

To explain what CFR is we first define *regret for not having chosen an action.* Suppose $a = (s_1, s_2)$ is a *strategy profile*, that is, a pair of strategies (one for each player). Let s'_1 be a strategy of player I . Denote by $u(\sigma, \tau)$ the expected pay to I if stategies σ and τ are used, Then the regret of I for not having chosen s'_1 instead of s_1 is $u(s'_1,s_2) - u(a) = u(s'_1, s_2) - u(s_1, s_2)$.

Having computed regrets for all s'_1 one chooses actions with a distribution that is proportional to positive regrets (regret-matching)

Regret is the loss of utility an algorithm suffers for not having selected the single (unknown) best deterministic strategy. A regret-minimizing strategy is one that guarantees that its regret grows sublinearly over time, and so eventually achieves the same utility as the best deterministic policy. [An introduction to Counterfactual Minimization; Neller, Lanctot`] .

The following is now a pseudocode for CFR

For each player, initialize all cumulative regrets to 0.

- For some number of iterations
  - -Compute a regret-matching strategy profile
  - -Add the strategy profile to the strategy profile sum and select each player action according to the strategy profile
  - -Compute player regrets and add player regrets to player cumulative regrets
- Return the average strategy profile

Counterfactual regret minimization (CFR), in a game in extensive form, uses regret-matching and,
(1) the probabilities of reaching each information set given a strategy policy and,
(2) a passing forward of probabilities of player action sequences , and backpropagation of utilities.

In HUNL the algorithm DEEPSTACK [DeepStack17] played 44000 hands of poker against professional players winning with statistical significance. It combines CFR with recursive reasoning and deep learning self-play.

2.3 Fundamental AI Algorithms

2.3.1 Minimax Algorithm

Minimax dates back as far as 1928 when mathematician Von Neumann in the field of Game Theory discovered the following Theorem:

Minimax Theorem (von Neuman, 1928)
For every finite two-person zero-sum game there exists at least one pair of optimal mixed strategies. Therefore, there exists a game value v, such that by applying the optimal strategy the first player guarantees for himself a payoff not worse than v, while the second player guarantees for himself a payoff not worse than $-v$.

Let us consider the general case of a game with two players, whom we will call MAX and M1N . MAX moves first followed by MIN, and then they take turns moving until the game is over. At the end of the game, points are awarded to the winning player (or sometimes penalties are given to the loser).

A 2-person zero-sum game can be defined as a kind of search problem with the following components:

• **The initial state,** which includes the board position and an indication of whose move it is.

• **A set** of **operators,** which define the legal moves that a player can make.

• A **terminal test,** which determines when the game is over. States where the game has ended are called **terminal states.**

• **A payoff function (also called a utility function), which gives a numeric value for the** outcome of a game. In chess, the outcome is a win, loss, or draw, which we can represent by the values +1, —1, or 0. Some games have a wider variety of possible outcomes; for example, the payoffs in backgammon range from +192 to —192. [RussellNorvig09]

If this were a normal search problem, then all MAX would have to do is search for a sequence of moves that leads to a terminal state that is a winner (according to the utility function), and then go ahead and make the first move in the sequence. But, MIN has something to say **about it. MAX therefore must find a strategy that will lead to a winning terminal state regardless** of what MIN does, where the strategy includes the correct move for MAX for each possible move by MIN. We will begin by showing how to find the optimal (or rational) strategy, even though normally we will not have enough time to compute it.

The **minimax** algorithm is designed to determine the optimal strategy for MAX, and thus to decide what the best first move is. MAX (resp. MIN) nodes are those of even (resp. odd) depth. The algorithm consists of five steps:

• Generate the whole game tree, all the way down to the terminal states.
• Apply the payoff function to each terminal state to get its value.
• Use the value of nodes one level higher up in the search tree. If *v* is a MAX (resp. MIN) nodes its value is the maximum(resp. minimum) of the values of its children.
• Continue backing up the values from the leaf nodes toward the root, one layer at a time.
• Eventually, the backed-up values reach the top of the tree; at that point, MAX chooses the move that leads to the highest value.

```
Pseudocode :

function minimax(node, depth, isMaximizingPlayer):

if node is a leaf node :
  return value of the node

if isMaximizingPlayer :
  bestVal = -INFINITY
  for each child node :
    value = minimax(node, depth+1, false)
    bestVal = max( bestVal, value)
  return bestVal
else :
   bestVal = +INFINITY
   for each child node :
     value = minimax(node, depth+1, true)
     bestVal = min( bestVal, value)
    return bestVal

// Calling the function for the first time.
minimax(0, 0, true)
```

Algorithm 2.1. Minimax Algorithm. [GeeksMinimax_αβ]

Recall that MAX(resp. MIN) nodes are those of even (resp. odd) depth, and that the value of a non-terminal MAX(resp. MIN) node is the maximum (resp. minimum) of the values of its children. Thus in the tree shown below (see Figure 2.2) the values of D, E, F, G, B, C and A are respectively 5, 9, 2, 0, 5, 0 and 5.

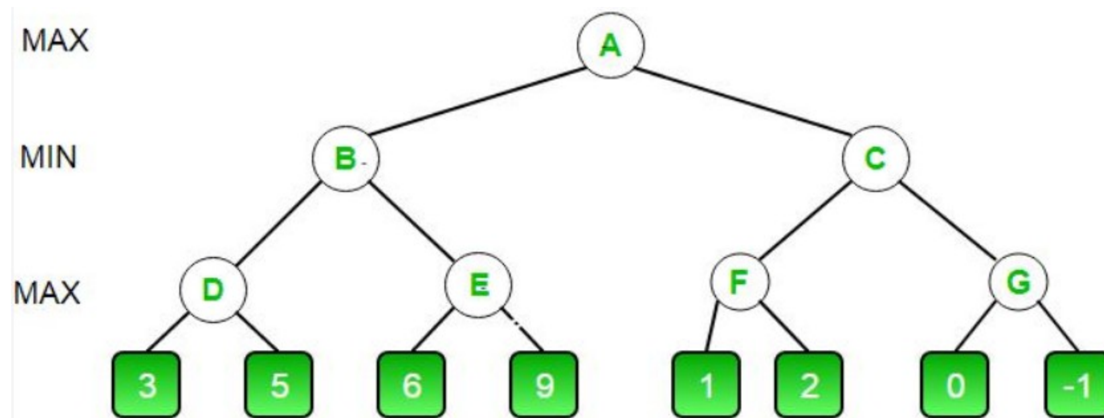


Figure 2.2. Tree. *Source:* [GeeksMinimax_αβ]

2.3.2 Alpha-Beta Pruning Algorithm

Minimax is a type of backtracking algorithm that is used in game theory , and finds an optimal move in a 2─person zero–sum game with perfect information . It is important theoretically ; for example it follows from it that such a game has a pair of optimal pure strategies. Chess, Checkers , Go and Backgammon are some examples.

However it is impractical because in a game with branching factor *b* and depth *d* there are $(b^{d+1}-1)/(b-1)$ nodes that need to be examined. To use it a more economical form is utilized: the alpha–beta algorithm with limited depth; the values at maximal depth are computed, for example, by an evaluation function.

The alpha–beta algorithm is essentially the minimax algorithm in which obvious bad moves have been eliminated. Using family-tree terminology, the following two rules are adopted:

- As soon as MAX finds a node *v* whose value is less than that of an uncle of *v* then *v´s* brothers are eliminated.
- As soon as MIN finds a daughter *v* whose value is greater than that of an aunt of *v* then *v's* sisters are eliminated.

If *p* is a parent of a node *v* of MAX, *g* is a parent of *p* and *u* with $u \neq p$ then *u* is an uncle of *v*; if *p* is a parent of a node *v* of MIN, *g* is a parent of *p* and *a* with $a \neq p$ then *a* is an aunt of *v* .

MAX´s nodes (resp. MIN´s nodes) are those with even (resp. odd) depth.
Alpha-Beta returns the same move as Minimax would.
The following is a pseudocode for the Alpha-Beta pruning algorithm.[GeeksMinimax_αβ]

```
Pseudocode :
function minimaxαβ(node, depth, isMaximizingPlayer, alpha, beta):

if node is a leaf node :
   return value of the node

if isMaximizingPlayer :
   bestVal = -INFINITY
   for each child node :
      value = minimaxαβ(node, depth+1, false, alpha, beta)
          bestVal = max( bestVal, value)
          alpha = max( alpha, bestVal)
          if beta <= alpha:
             break
    return bestVal
else :
   bestVal = +INFINITY
   for each child node :
      value = minimaxαβ(node, depth+1, true, alpha, beta)
      bestVal = min( bestVal, value)
      beta = min( beta, bestVal)
      if beta <= alpha:
          break
   return bestVal
// Calling the function for the first time.
minimaxαβ(0, 0, true, -INFINITY, +INFINITY)
```

Algorithm 2.2. Alpha-Beta Algorithm. [GeeksMinimax_αβ]

Using the tree shown in Figure 2.2, we will perform the Alpha-Beta Pruning algorithm:

- The initial call starts from A. The value of alpha here is -INFINITY and the value of beta is +INFINITY. These values are passed down to subsequent nodes in the tree. At A the maximizer must choose max of B and C, so A calls B first
- At B it the minimizer must choose min of D and E and hence calls D first.
- At D, it looks at its left child which is a leaf node. This node returns a value of 3. Now the value of alpha at D is max(-INF, 3) which is 3.
- To decide whether its worth looking at its right node or not, it checks the condition beta<=alpha. This is false since beta = +INF and alpha = 3. So it continues the search.
- D now looks at its right child which returns a value of 5. At D, alpha = max(3, 5) which is 5. Now the value of node D is 5.
- D returns a value of 5 to B. At B, beta = min(+INF, 5) which is 5. The minimizer is now guaranteed a value of 5 or lesser. B now calls E to see if he can get a lower value than 5.
- At E the values of alpha and beta is not -INF and +INF but instead -INF and 5 respectively, because the value of beta was changed at B and that is what B passed down to E
- Now E looks at its left child which is 6. At E, alpha = max(-INF, 6) which is 6. Here the condition becomes true. beta is 5 and alpha is 6. So beta<=alpha is true. Hence it breaks and E returns 6 to B
- Note how it did not matter what the value of E's right child is. It could have been +INF or -INF, it still wouldn't matter, We never even had to look at it because the minimizer was guaranteed a value of 5 or lesser. So as soon as the maximizer saw the 6 he knew the minimizer would never come this way because he can get a 5 on the left side of B. This way we didn't have to look at that 9 and hence saved computation time.
- E returns a value of 6 to B. At B, beta = min(5, 6) which is 5.The value of node B is also 5

So far this is how our game tree looks. The 9 is crossed out because it was never computed. See Fig. 2.3

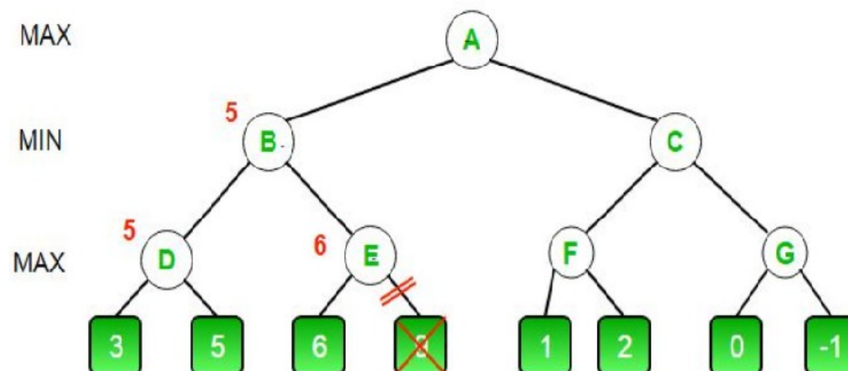


Figure 2.3. Pruned Tree (stage 1). *Source:* [GeeksMinimax_αβ]

- B returns 5 to A. At A, alpha = max(-INF, 5) which is 5. Now the maximizer is guaranteed a value of 5 or greater. A now calls C to see if it can get a higher value than 5.
- At C, alpha = 5 and beta = +INF. C calls F
- At F, alpha = 5 and beta = +INF. F looks at its left child which is a 1. alpha = max(5, 1) which is still 5.
- F looks at its right child which is a 2. Hence the best value of this node is 2. Alpha still remains 5
- F returns a value of 2 to C. At C, beta = min(+INF, 2). The condition beta <= alpha becomes true as beta=2 and alpha=5. So it breaks and it does not even have to compute the entire sub-tree of G.
- The intuition behind this break off is that, at C the minimizer was guaranteed a value of 2 or lesser. But the maximizer was already guaranteed a value of 5 if he choose B. So why would the maximizer ever choose C and get a value less than 2 ? Again you can see that it did not matter what those last 2 values were. We also saved a lot of computation by skipping a whole sub tree.
- C now returns a value of 2 to A. Therefore the best value at A is max(5, 2) which is a 5.
- Hence the optimal value that the maximizer can get is 5

This how our final game tree looks like. As you can see G has been crossed out as it was never computed. See Fig. 2.4 .

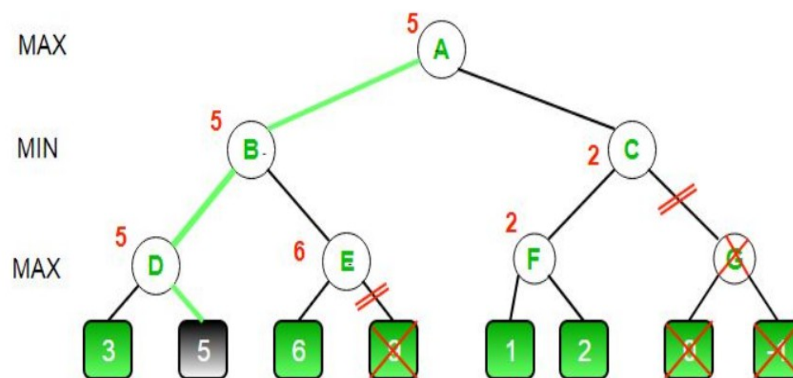


Figure 2.4. Pruned Tree (final stage). Source: [GeeksMinimax_αβ]

Effectiveness of alpha-beta pruning

The effectiveness of alpha-beta depends on the ordering in which the successors are examined. Under the strong assumption that the best succesors are examined first, it turns out that alpha-beta pruning only needs to examine $O(b^{d/2})$ nodes to choose the best move as opposed to $O(b^{d})$,the estimate with the minimax algorithm. Still $O(b^{d/2})$ is a large number unless one is happy with depth-limited search with, say, $d \leq 3$.
This means that the effective branching factor is $b^{1/2}$ instead of b—for chess, 6 instead of 35. Put another way, this means that alpha-beta can look ahead twice as far as minimax for the same cost. Thus, by generating 150,000 nodes in the time allotment, a program can look ahead eight plies instead of four. By thinking carefully about which computations actually affect the decision, we are able to transform a program from a novice into an expert.[RussellNorvig09]

The effectiveness of alpha-beta pruning was first analyzed in depth by Knuth and Moore (1975). As well as the best case described in the previous paragraph, they analyzed the case in which successors are ordered randomly. It turns out that the asymptotic complexity is $O((b/logb)^d)$, which seems rather dismal because the effective branching factor *b/logb* is not much less than *b* itself. On the other hand, the asymptotic formula is only accurate for $b > 1000$ or so—in other words, not for any games we can reasonably play using these techniques. For reasonable *b*, the total number of nodes examined will be roughly $O(b^{3/4})^d$. In practice, a fairly simple ordering function (such as trying captures first, then threats, then forward moves, then backward moves) gets you fairly close to the best-case result rather than the random result.
Another popular approach is to do an iterative deepening search, and use the backed-up values from one iteration to determine the ordering of successors in the next iteration.
It is also worth noting that all complexity results on games (and, in fact, on search problems in general) have to assume an idealized tree model in order to obtain their results. For example, the model used for the alpha-beta result in the previous paragraph assumes that all nodes have the same branching factor *b*; that all paths reach the fixed depth limit *d*; and that the leaf evaluations are randomly distributed across the last layer of the tree. This last assumption is seriously flawed: for example, if a move higher up the tree is a disastrous blunder, then most of its descendants will look bad for the player who made the blunder. The value of a node is therefore likely to be highly correlated with the values of its siblings. The amount of correlation depends very much on the particular game and indeed the particular position at the root. Hence, there is an unavoidable component of empirical science involved in game-playing research, eluding the power of mathematical analysis. [RussellNorvig09]

2.3.3 Expectiminimax

In real life, there are many unpredictable external events that put us into unforeseen situations. Many games mirror this unpredictability by including a random element such as throwing dice. In this way, they take us a step nearer reality, and it is worthwhile to see how this affects the decision-making process.

Backgammon is a typical game that combines luck and skill. Dice are rolled at the beginning of a player's turn to determine the set of legal moves that is available to the player.

The *expectiminimax* algorithm is a variation of the minimax algorithm, for use in artificial intelligence systems that play two-player zero-sum games, such as backgammon, in which the outcome depends on a combination of the player's skill and chance elements such as dice rolls. In addition to "min" and "max" nodes of the traditional minimax tree, this variant has "chance" nodes, which take the expected value of a random event occurring. In game theory terms, an expectiminimax tree is the game tree of an extensive-form game of perfect, but incomplete information. [RussellNorvig09]

In the traditional minimax method, the levels of the tree alternate from max to min until the depth limit of the tree has been reached. In an expectiminimax tree, the "chance" nodes are interleaved with the max and min nodes. Instead of taking the max or min of the payoff values of their children, chance nodes take a weighted average, with the weight being the probability that child is reached. [RussellNorvig09]

The interleaving depends on the game. Each "turn" of the game is evaluated as a "max" node (representing the AI player's turn), a "min" node (representing a potentially-optimal opponent's turn), or a "chance" node (representing a random effect or player).

For example, consider a game in which each round consists of a single dice throw, and then decisions made by first the AI player, and then another intelligent opponent. The order of nodes in this game would alternate between "chance", "max" and then "min".

The expectiminimax algorithm is a variant of the minimax algorithm and was first proposed by Donald Michie in 1966. A pseudocode is given below. [Michie66]

Pseudocode

```
function expectiminimax(node, depth)
   if node is a terminal node or depth = 0
      return the heuristic value of node
   if the adversary is to play at node
      // Return value of minimum-valued child node
      let α:= +∞
      foreach child of node
         α:= min(α, expectiminimax(child, depth-1))
   else if we are to play at node
      // Return value of maximum-valued child node
      let α:= -∞
      foreach child of node
         α:= max(α, expectiminimax(child, depth-1))
   else if random event at node
      // Return weighted average of all child nodes' values
      let α:= 0
      foreach child of node
         α:= α + (Probability[child] * expectiminimax(child, depth-1))
   return α
```

Algorithm 2.3. Expectiminimax Algorithm. [GeeksMinimax_αβ]

Note that for random nodes, there must be a known probability of reaching each child. (For most games of chance, child nodes will be equally-weighted, which means the return value can simply be the average of all child values.) [WikiExpecti18]

2.3.4 Heuristics and Evaluation Functions

The word "heuristic" is derived from the Greek verb *heuriskein,* meaning "to find" or "to discover." Archimedes is said to have run naked down the street shouting *"Heureka"* (I have found it) after discovering the principle of flotation in his bath. Later generations converted this to Eureka.

Heuristic techniques dominated early applications of artificial intelligence. The first "expert systems" laboratory, started by Ed Feigenbaum, Brace Buchanan, and Joshua Lederberg at Stanford University, was called the Heuristic Programming Project (HPP). Heuristics were viewed as "rules of thumb" that domain experts could use to generate good solutions without exhaustive search. Heuristics were initially incorporated directly into the structure of programs, but this proved too inflexible when a large number of heuristics were needed. Gradually, systems were designed that could accept heuristic information expressed as "rules," and rule-based systems were born. [RussellNorvig09]

In the specific area of search algorithms, it refers to a function that provides an estimate of solution cost. For most problems, the cost of reaching the goal from a particular state can be estimated but cannot be determined exactly. A function that calculates such cost estimates is called a *heuristic function.*

A problem with less restrictions on the operators is called a *relaxed problem.* It is often the case that the cost of an exact solution to a relaxed problem is a good heuristic for the original problem.

An *evaluation function*, also known as a *heuristic evaluation function* or *static evaluation function*, is a function used by game-playing programs to estimate the value or goodness of a position in the minimax and related algorithms. The evaluation function is typically designed to prioritize speed over accuracy; the function looks only at the current position and does not explore possible moves (therefore static). [RussellNorvig09]

One popular strategy for constructing evaluation functions is as a weighted sum of various factors that are thought to influence the value of a position. For instance, an evaluation function for Chess might take the form

c1 · material + c2 · mobility + c3 · king safety + c4 · center control + ...

Evaluation functions in Go take into account both territory controlled, influence of stones, number of prisoners and life and death of groups on the board.

The Scrabble engine Heuri, a crucial part of this work, presents a Probabilistic heuristic function, which is precisely a weighted sum of factors like: the score of the move and the expected value of the tiles left on the rack [GonzalezAlquezar12, GonzalezRamirez08, GonzalezAlquezar09, RamirezGonzalez09].

2.3.5 B* Algorithm

B* is a best-first graph search algorithm that finds the least-cost path from a given initial node to any goal node.

The algorithm stores intervals for nodes of the tree rather than a single point-valued estimates. Then, leaf nodes of the tree can be searched until one of the top level nodes has an interval which is clearly "best".

The interval is assumed to contain the correct value of that node. If all intervals attached to leaf nodes satisfy this assumption then, B* will find an optimal path to the goal state.

To regress the intervals within the tree a parent's upper (resp. lower) bound is set to the maximum of the upper (resp. lower) bounds of the children. Note that different children might supply these bounds. An upper bound is called an optimistic value and a lower bound a pessimistic value.

B* expands nodes in order to create *separation*, which occurs when the pessimistic value of a direct child of the root is greater than or equal to the optimistic value of any other direct child of the root. A tree that creates separation at the root contains a proof that the best child is at least as good as any other child.

There are time or memory limits. When a limit is encountered then you must make a heuristic judgement about which move to choose, having as evidence the intervals of root nodes.
There are two strategies to choose a node to expand: prove-best (choose the node with highest upper bound) and disprove-rest (choose the child that has second highest upper bound).

One suggestion is: expand the choice with smaller tree. Another suggestion from Sheppard is to alternate. Palay (1983) extended and developed B* to use probability distributions rather than optimistic-pessimistic ranges. Palay applied B* to chess; endpoints were assigned using null-moves searches.

Brian Sheppard in Maven applied B* to Scrabble Endgames [Sheppard03]; endpoint evaluations were assigned using a heuristic planning system.

An evaluation function must be invoked, to decide the approximate closeness to the goal of a given node at the periphery of the search. This or a similar function can also be used for deciding which node to expand next in a best-first search.

As Berliner states, the B* method assigns a greater responsibility for guiding the search, to the evaluation functions that compute the bounds than has been done before.

2.3.6 Monte-Carlo Tree Search

The idea to incorporate Monte Carlo simulations into the tree growing process introduced in [Jui99] was first adopted for Go by [Cou06] in the program Crazy Stone. This program has managed to outperform the conventional techniques and since then the Monte Carlo Tree Search (MCTS) has enjoyed great popularity in the computer Go community.

Briefly, in MCTS during the descent each move is chosen according to its value, which is accumulated by making random simulations, each one representing a complete game. The value for the move reflects the information of the number and the outcome of the simulations that have run through it.

This approximation is justified by a central limit theorem, which says that the Monte Carlo values (mean of all outcomes) converge to the normal distribution. If the tree is explored to a fair extent, the strategy using MCTS converges to the optimal strategy [KS06a].

MCTS has an important advantage over pruning [Bou06]. The values of the moves may change as the tree grows and initially inferior moves may eventually gain importance. In pruning, the branches are completely discarded, in contrast to MCTS, where they still have non-zero probability and a chance to come into play later.

Advantages of MCTS

The main advantage of MCTS is that it dramatically reduces the number of explored nodes in the tree. The search tree grows asymmetrically with a low branching factor allowing only for the promising branches [CM07]. Secondly, MCTS requires no domain knowledge. Thus, MCTS is very popular for general game playing when the rules are not known. But then, MCTS can be tailored to a certain application by inserting some domain-specific heuristics [DU07].

Moreover, there is no need to have a fixed number of simulations, in contrast to iterative deepening [Bou04]. The search can be stopped after arbitrarily many time steps, and the best result achieved so far is returned. This is profitable when the time for the move in the game is limited and it is necessary to return some approximation for the optimal move.

In addition to that, Monte-Carlo tree search is easy to parallelize because the simulations are independent. This advantage was utilized in the distributed version of AlphaGo [SHM+16].

Description of Monte-Carlo Tree Search

Let us describe Monte-Carlo Tree Search, a technique in AI especially useful with games with high branching factor such as Go and now Chess. It does not necessarily require knowledge about the game. It is often used together with UCB1 (Upper Confidence bound applied to trees).

Consider the multi-armed bandit problem. This is often described as follows. Suppose one is at a row of slot machines, you are given the probability distributions for payoffs in each machine, and you have to decide which machines to play, how many times, in which order and whether to continue with the current machine or try a different one.

One simultaneously tries to acquire new knowledge (called “exploration”) with little-used machines and

optimize one's decisions based on existing knowledge (called “exploitation”) over a period of time considered.

For a node *R*, all immediate successors are considered, *k* random games are played out to the end and scores are recorded. Then one applies successively the following four operation:

- Selection: This may be done choosing randomly, a successor of the node (pure MCTS). Instead one may use the UCB1 algorithm to choose a successor of the node; this means that from the set of successors $\{x_i\}$ of the node choose i such that $w_i/n_i + ((2ln(N_i))/n_i)^{1/2}$ is maximal where
  w_i is the number of wins for the node after move i.
  n_i is the number of simulations after move i.
  N_i is the number of simulations of the i-th move ran by the parent node.
- Expansion: occurs when an unvisited child is randomly chosen. In the expansion step the value for the newly added node can be initialized not randomly but according to the value function [GS07]. The *value function* $v: S \rightarrow \{\pm 1\}$ determines the outcome of the game starting from the current state.
- Simulation is a Monte Carlo simulation, either purely random (light playout) or guided by some, computationally expensive, evaluation function (heavy playout). Random rollouts are very unrealistic and provide a poor evaluation of the target function. The idea is to make them more sensible, either by using some domain knowledge or by learning tactics of the game. One example of the former is a *rule-based approach* [ST09]. Some examples of learning are *temporal difference learning* [FB10], *pattern matching* [WG07, DU07, Cou07], and *deep reinforcement learning* [AlphaGo16, AlphaGo Zero17, and AlphaZero18].
- Backpropagation (or update) . All the positions of the path to the terminal point increment their play count by 1, and the win count is incremented by 1 for those vertices of the path belonging to the winner.

There are still more enhancements to MCTS, see [BPW+12, Bai10]. For example in AlphaGo [AlphaGo16] Reinforcement Learning and Convolutional Neural Networks were utilized to enhance the performance of the MCTS achieving a truly amazing Go engine. In [AlphaZero17] Human Knowledge was removed from AlphaGo, and a Multipurpose Learning Algorithm achieved the World strongest engines for Go, Chess and Shogi.

An example is shown in Fig. 2.5:

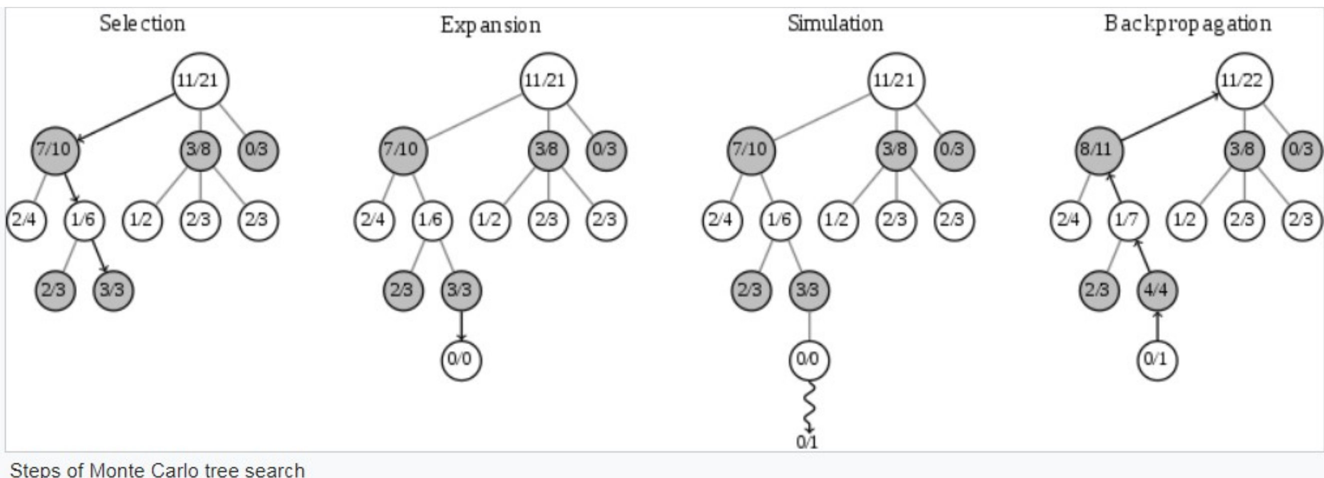


Figure 2.5 MCTS Steps. Source: [WikiMCTS19]

2.3.7 Artificial Neural Networks

Artificial neural networks and cognitive modeling are information processing paradigms inspired by the way biological neural systems process data. They try to simulate some properties of biological neural networks.

A biological neural network is composed of a group or groups of chemically connected or functionally associated neurons. A single neuron may be connected to many other neurons and the total number of neurons and connections in a network may be extensive.

The simplest version of the neuron in the human brain is the *perceptron model* [Fra57], depicted in Figure (2) inside the Fig. 2.6. As an input it gets environmental information and computes a weighted sum of these inputs, which is called a *potential* of the perceptron. After that, the *activation function* is applied to the potential which is a simple threshold in the case of the perceptron: $sign(\sum w_i x_i + b)$. The result of the output function basically shows if the perceptron reacted to the external impulse. The weights w_i and the bias b are the trainable parameters of this linear model.

Neural networks can be viewed as mathematical models that define a function.

A neural network is a directed (usually acyclic) graph whose set of vertices (nodes or *neurons*) is decomposed into disjoint subsets called *layers*. The first layer consists of inputs and the last one of outputs. Between them there might be middle layers called *hidden* layers. See Figure (3) inside the Fig. 2.6.

An edge (connection) from neuron i to neuron j has an *associated weight* $w_{ij} \in R$.

A neuron with label j has the following components:

- An *activation* $\alpha_j(t)$, the neuron's state , defined on a discrete set (time).
- A *threshold* $\theta_j \in R$.
- An *output* function f_{out} computing the output $o_j(t) = f_{out}(\alpha_j(t))$.
- An *activation function* f that computes $\alpha_j(t+1)$ by the formula $\alpha_j(t+1) = f(\alpha_j(t), p_j(t), \theta_j)$ where $p_j(t)$, the *propagation function*, is $p_j(t)=\Sigma_i o_i(t) w_{ij} + w_{0j}$ where i runs over the predecessors of j and w_{0j} is a bias.

Typically the weights and thresholds are continuously modified by an algorithm (*training*) called the *learning rule* .

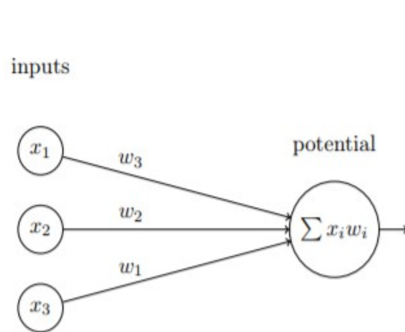


Figure 2: The perceptron

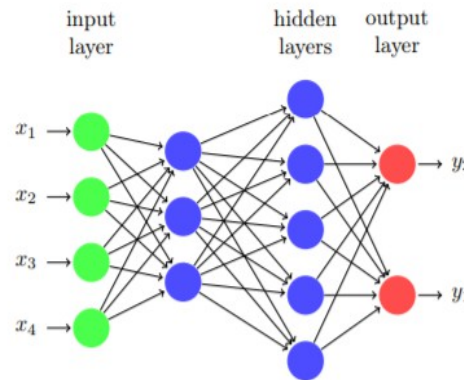


Figure 3: The multilayer feed-forward neural network

Figure 2.6 Perceptron and Multilayer feed-forward neural network. Source: [HM95]

The output function can be replaced by a differentiable one. The weights of the neuron with this function can be trained by the gradient method (minimizing the loss by making the derivative 0). The most popular output functions are *sigmoid* and *tanh*, their advantage is the ability to react to the slight changes in the input [HM95].

The neuron is the principal unit in the *Artificial Neural Network* (ANN) [MP43], which is a quite crude approximate model of the human brain. The ANN is a bunch of neurons computing a mapping $f : R^m \rightarrow R^n$. There exists a wealth of different kinds of ANNs, we would like to discuss only *feed-forward* NNs (Figure 3, inside the Fig. 2.6), as they are employed in AlphaGo.

In the feed-forward ANNs the neurons are arranged in layers, the connections exist only between consequent layers and there are no loops in the network. It consists of a single input and a single output layer and several hidden layers. The weights of the network are trained by *backpropagation* [WH86].

Feed-forward neural networks were applied to Go [Dah99]. However, due to the huge size of the Go board, back in 1999, it was highly difficult to incorporate it to the feed-forward neural network. For instance, if every intersection of the board was fed into a separate input neuron, it would yield about 100k connections to the first hidden layer. Moreover, in a feed-forward neural network each neuron on the current level is connected with each neuron in the previous level. Every connection has its weight, which means that a lot of parameters have to be estimated.

To tackle this, one can assume that the most informative part of the board is within a limited space around the previous move - a current local “fight”. Then it makes no sense to consider the whole board. This assumption was taken over in [Dah99].

Still, one can go further and introduce a whole new architecture of the neural network, that will take this idea into account. Here we are talking about *convolutional neural networks* that were implemented in AlphaGo.

Convolutional Neural Networks

In recent years the convolutional neural networks (CNNs) have been successfully applied in the domains of face and character recognition, image classification and other visual tasks [MHSS14]. Their operation is based on extracting local features from the input.

A convolutional neural network is based on two concepts: *local receptive fields* and *weight sharing*. The former concept means that each neuron is connected only to a small region of a fixed size in the previous layer. Every

neuron is responsible to its own subarea in the previous layer, but the set of weights of connections, known as *filter*, is the same for all neurons (*weight sharing* assumption).

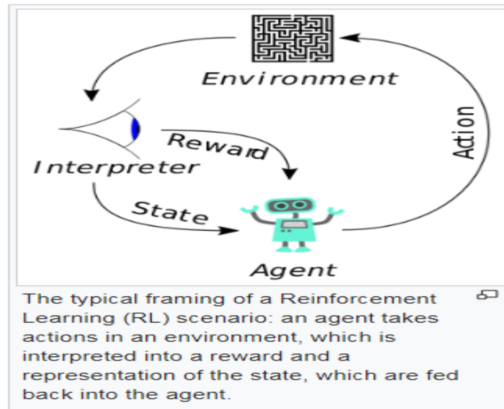
Each filter can be considered as a single feature extractor. The more different filters are applied, the more information we get from the input (256 filters are used in AlphaGo). The learning task for CNN is to find the weights within each filter.

One can observe that due to the fact that the filter is the same for all neurons in one layer and the size of the filter is restricted, the number of free parameters in the network is dramatically reduced [SN08].

CNNs have been very successfully applied in Go [MHSS14, SN08, CS15]. Due to the *translational invariance* [CS15] (the assessment of a position is not affected by shifting the board) and the local character of features in Go, CNNs come as the best choice.

2.3.8 Reinforcement learning, Q-learning

Reinforcement Learning is an area of machine learning concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward (see Fig. 2.7).



The typical framing of a Reinforcement Learning (RL) scenario: an agent takes actions in an environment, which is interpreted into a reward and a representation of the state, which are fed back into the agent.

Figure 2.7. Reinforcement Learning. Source: [WikiRL19]

It is one of 3 basic machine learning paradigms: supervised, unsupervised and reinforcement learning (RL).

RL differs from supervised learning in that labeled input/output pairs (training examples) need not be presented, and sub-optimal actions need not be explicitly corrected. Instead the focus is finding a balance between exploration (of uncharted territory) and exploitation(of current knowledge).[Kaelbling96]

In most AI topics, mathematical frameworks are built to tackle problems. For RL, the answer is the **Markov Decision Process (MDP)**. It produces an easy framework to model a complex problem. An **agent** (e.g. a human) observes the environment and takes **actions**. **Rewards** are given out but they may be infrequent and delayed. Very often, the long-delayed rewards make it extremely hard to untangle the information and traceback what sequence of actions contributed to the rewards. [Hui18]

Markov decision process (MDP), see Fig. 2.8, composes of:

$$(\mathcal{S}, \mathcal{A}, P, r, \rho_0, \gamma)$$

An MDP is defined by:

- Set of states S
- Set of actions A
- Transition function $P(s' \mid s, a)$
- Reward function $R(s, a, s')$
- Start state s_0
- Discount factor γ
- Horizon H

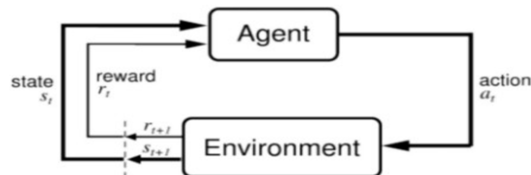


Figure 2.8. Markov Decision Process (MDP). Source: [Hui18]

States in MDP can be represented as raw images. See Fig. 2.9.

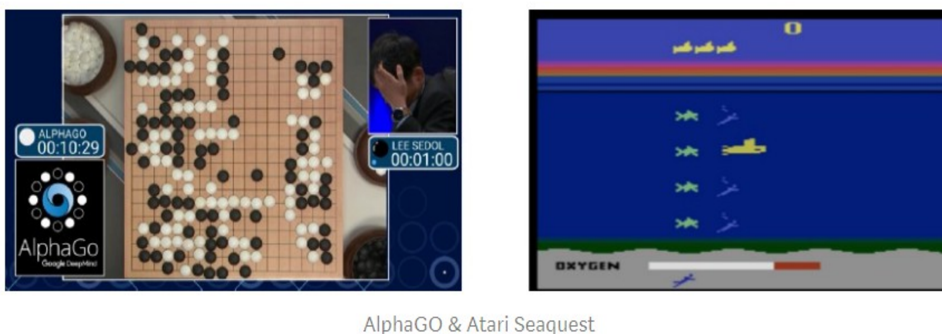


Figure 2.9. MDP states represented as images. Source: [Hui18]

- An **action** can be a move in a chess game or moving a robotic arm or a joystick.
- For a Go game, the reward is very sparse: 1 if we win or -1 if we lose. Sometimes, we get rewards more frequently. In the Atari Seaquest game, we score whenever we hit the sharks.
- The discount factor discounts future rewards if it is smaller than one. Money earned in the future often has a smaller current value, and we may need it for a technical reason to converge to the solution better.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

- We can rollout actions forever or limit the experience to N time steps. This is called the *horizon*.

The transition function is the system dynamics. It predicts the next state after taking action. It is called the **model** which plays a major role in the Model-based RL .

The concepts in RL come from many research fields including the control theory. Different notations may be used in a different context. The state can be written as s or x, and action as a or u. An **action** is the same as a **control**. We can maximize the rewards or minimize the costs which are simply the negative of each other.

In RL, our focus is finding an optimal **policy.** A policy tells us how to act from a particular state $a_t = \pi_\theta(s_t)$

Like the weights in Deep Learning Methods, this policy can be parameterized by θ . See Fig. 2.10.

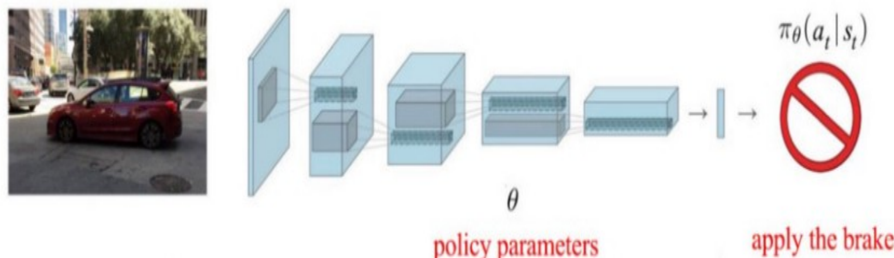


Figure 2.10. Optimal policy. Source: [Hui18]

and we want to find the policy that makes the most rewarding decisions:

$$\max_{\theta} E_{\tau \sim p_\theta(\tau)} \left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t) \right]$$

Find the policy with max rewards

expected rewards

Source: [Hui18]

In real life, very few things are absolute. So our policy can be deterministic or stochastic. For stochastic, the policy outputs a Probability distribution instead: $p(a_t \mid s_t) = \pi_\theta(a_t \mid s_t)$.

Finally,in RL, we want to find a sequence of actions that maximize expected rewards or minimize cost.

sequence of actions

$$\underbrace{p_\theta(\mathbf{s}_1, \mathbf{a}_1, \ldots, \mathbf{s}_T, \mathbf{a}_T)}_{p_\theta(\tau)} = p(\mathbf{s}_1) \prod_{t=1}^{T} \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$$

model

policy

probability of a sequence of actions

$$\max_{\theta} E_{\tau \sim p_\theta(\tau)} \left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t) \right]$$

rewards

Source: [Hui18]

There are many ways to solve the problem. For example, we can:

- Analyze how good is it to reach a certain state or take a specific action (i.e. value-learning. An example is the *Q-learning algorithm*).
- Use the model to find actions that have the maximum rewards (model-based learning), or
- Derive a policy directly to maximize rewards (policy gradient).

***Q*-learning** is a model-free reinforcement learning algorithm. The goal of Q-learning is to learn a policy, which tells an agent what action to take under what circumstances. It does not require a model (hence the connotation "model-free") of the environment, and it can handle problems with stochastic transitions and rewards, without requiring adaptations.

For any finite Markov decision process (FMDP) *Q*-learning finds a policy that is optimal in the sense that it maximizes the expected value of the total reward over any and all successive steps, starting from the current state. *Q*-learning can identify an optimal action-selection policy for any given FMDP, given infinite exploration time and a partly-random policy. "Q" names the function that returns the reward used to provide the reinforcement and can be said to stand for the "quality" of an action taken in a given state.

Like a child incentivized by spankings and candy, these algorithms are penalized when they make the wrong decisions and rewarded when they make the right ones – this is reinforcement.

Just as oil companies have the dual function of pumping crude out of known oil fields while drilling for new reserves; so too, RL algorithms can be made to both *exploit* and *explore* to varying degrees, in order to ensure that they don't pass over rewarding actions at the expense of known winners.

Reinforcement Learning is particularly well-suited to problems that include a long-term versus short-term reward trade-off. It has been applied successfully to various problems, incluiding robot control, elevator scheduling, telecommunications, Backgammon, Checkers and Go.

2.4 State-of-the-Art Engines (and the Algorithms used)

2.4.1 Go

Prior to 2015 the best Go programs only managed to reach amateur dan level. On smaller 9x9 and 13x13 boards or in handicap 19x19 games, computer programs performed better and were able to compare to professional players.

The number of legal moves, around 150-250 per turn, rarely falls below 100. In Chess it averages 37.
The placement of a single stone in the initial phase can affect the game a 100 or more moves later.

In capture-based games (such as chess) a position can be evaluated relatively easily. In Go there is no easy way to do it. However a 6-kyu human can evaluate a position at a glance, to see which player has more territory, and even beginners can estimate the score within 10 points. The number of stones in the board is only a weak indicator of the strength of the position, and a territorial advantage for one player might be compensated by the opponent's strong positions and influence all over the board.

Summary of Computer Achievements:
August 2008: A computer (Mogo) won a game against an 8-dan professional player with a 9-stone handicap.
2008: Crazy Stone won against a 4-dan professional receiving a handicap of 8 stones.
2010: Zen beat a professional with a 6-stone handicap.
March 2012: Zen beat a top professional Takemiya Masaki 9p with a 4-stone handicap by twenty points.
2013: Zen won a game against a professional player with a 3-stone advantage.
2014 Crazy Stone won a game without handicap against eleven-times German Go champion Franz-Josef Dickhut a 6 dan amateur.
October 2015: AlphaGo beat Fan Hui (European Go Champion) 5-0 with no Handicap.
March 2016: AlphaGo beat Lee Sedol (Top Player in the World) 4-1 with no Handicap.
May 2017: AlphaGo beat Ke Jie (World number 1 ranking) 3-0 with no Handicap.
October 2017: AlphaGo Zero beat AlphaGo 100-0.
January 2018: AlphaZero beat 60-40 AlphaGo Zero.

The game of Go has been around for centuries and it is indeed a very popular brainteaser nowadays. Along with Chess and Checkers, Go is a game of perfect information. That is, the outcome of the game solely depends on the strategy of both players. This makes it attractive to solve Go computationally because we can rely on a machine to find the optimal sequence of moves. However, this task is extremely difficult due to the huge search space of possible moves. Therefore, Go has been considered a desired frontier for AI, which was predicted to be not achievable in the next decade [BC01].

2.4.1.1 AlphaGo

Until recent time, many computer Go bots appeared, still they barely achieved the level of a master player, let alone play on a par with the professionals [RTL+10]. However, in the beginning of 2016, Google DeepMind published an article, where they stated that their program, AlphaGo, was able to win over a professional player [SHM+16]. Several months after that AlphaGo defeated the world Go champion in an official match, an event of a great importance, because now the “grand challenge” [Mec98, CI07] was mastered.

The problem with Go is the size of the board, which yields a 10^{170} positions state space [MHSS14, VDHUVR02]. In comparison, Chess’ state space is about 10^{43} [Chi96]. Such games are known to have a high branching factor - the number of available moves from the current position. The number of possible game scenarios in Go is greater than the number of atoms in the universe [TF06].

The authors of AlphaGo managed to solve this problem. The system they designed is based on tree search, boosted by neural networks predicting the moves. However, all these techniques are not novel in computer Go and have been utilized by other authors, too. So what makes AlphaGo so special? Here we discuss how AlphaGo is designed in the context of the history of computer Go. By unfolding the architecture of AlphaGo we show that every single detail of its implementation is a result of many years' research, but their ensemble is the key to AlphaGo's success.

Deep Reinforcement Learning to Support MCTS

To enhance the performance of MCTS AlphaGo utilizes Deep Convolutional Neural Networks and Reinforcement Learning. In this section we first provide the background of these approaches and later discuss how they are applied in the MCTS settings.

The Monte Carlo method has the nice property that the mean value of simulations converges to the true value; however to guarantee this, it should be provided with an infinite number of simulations. Of course, in no domain is this possible, let alone board games, where the thinking time is strictly limited [KS06b].

Therefore, if the quantity of the simulations is limited, one has to increase the quality of every single simulation. The more realistic the decisions are, the faster convergence. The side-effect of the reduced number of simulations can be poorer *selection* which will be prone to overlooking the good moves.

For tackling this problem one may refer to the way humans play Go. Clearly, they also pick a move and try to predict the outcome after playing it. But contrarily to MCTS, people do this not absolutely randomly. The experienced players use *intuition* to preselect an action [CS15]. Of course, if MCTS was guided by this intuition, it would converge faster. Therefore, many authors [Dah99, CS15, SN08] try to imitate the way people think. One of the possible means is an artificial neural network.

Neural Networks in AlphaGo

AlphaGo uses neural networks to *predict human moves*. For this reason the input to the AlphaGo's CNN is the current board setting and the output is the prediction of a move a person would make.

Let us describe it more precisely. To train the CNN, the authors took 30.000 games of Go professionals that were recorded on the Go server KGS [Sch10]. From each game random positions were selected together with the consequent actions of the player. This action was the objective of the network's prediction.

The input position was translated into 48 features, which indicate the color of the stone at each intersection, the number of free adjacent cells and some other information. These features have been selected according to the results of the previous research [CS15].

The input layer was, therefore, a 19×19×48 stack, carrying the value of every feature for each intersection. The CNN had 13 hidden layers with 256 filters on each layer. The output layer was of the size 19×19 and each cell in the output contained the probability that a person will put a stone in the corresponding intersection.

The neural network was trained by a standard backpropagation [WH86]. The scheme above represents a *supervised learning* approach, therefore, we refer to the resulting network as the SL network. However, in AlphaGo also *reinforcement learning* was used.

Reinforcement Learning of the Neural Networks

Even if the quality of the move prediction was 100% (and the accuracy of AlphaGo's CNN prediction was reported to be 57% in [SHM+16]), it will still be a score in the task of modeling peoples' moves. But the ultimate task is to win over human, therefore, some approach should be utilized that would enable AlphaGo to surpass the human abilities. To achieve this aim the *Reinforcement Learning* [Wil92] was applied in AlphaGo. Conceptually, reinforcement learning is a method of training an AI agent by not giving him explicitly the correct answer. Instead, the aim of the agent is to maximize the reward, which is a symbolical environment feedback to his actions.

Getting back to the setting of Go, the reward is expressed as ±1 for win/lose in the end of the game and 0 for all other steps. Now we iteratively increase the power of the SL network by letting it play against the other versions of itself and consequently learn from its own games.

At one iteration step the current version (in terms of current parameters) is opposed by some random instance of the SL network from the previous iterations. They play a match of Go and get the final reward. This reward tells the current SL network if its actions were correct and how the parameters should be accordingly adjusted. Every 500 iterations the version with the current parameters is added to the pool of previous versions. The reinforcement learning technique provided a significant improvement to the playing strength of the SL network.

Integrating Neural Networks with MCTS

So what are the neural networks used in AlphaGo for?

The SL network is used at the *selection* stage of MCTS to encourage exploration. Recall, that a good selection rule keeps a balance between selecting optimal known moves and investigating the new ones. AlphaGo uses a variant of the *UCT* rule to select the action a, maximizing the formula $x(a) + u(a)$, where $x(a)$ is the value of the move (which can be found with Monte Carlo rollouts) and $u(a)$ is proportional to $P(a)$ - the probability predicted by the SL neural network. In a sense, CNN biases MCTS to try out the moves which have been scarcely explored but which seem optimal to the CNN.

Although the reinforcement learning networks proved to be stronger than the SL networks, the overall performance of AlphaGo was better when the move selection was enhanced with the SL network predictions. It can be explained by the fact that the SL-network is more human-like, as it was trained on the real people's games [SHM+16]. People tend to explore more, either because of the mistakes during the game or out of ardor.

Nevertheless, the reinforcement learning networks (RL networks) found their application in the other component of AlphaGo, the *value network*, which is used to approximate the *value function*.

Value Networks

Let s denote a state in the game, then $v(s)$ is a *value function* ($v : S \rightarrow \pm 1$, where S is a state space). The value function predicts the game result (win/lose) for the current state s under the perfect play of both opponents.

Theoretically, value function can be found by the full traversal of the game tree, but as it was mentioned before, for Go it is not feasible. On the one hand, it is known that the mean value of the Monte Carlo rollouts converges to the value function [KS06b]. However, when too few simulations are made (due to the limited time for a move), the rollout results tend to be inaccurate [HM13].

To aid them, AlphaGo learns to approximate the value function by a powerful *value network*. This network has absolutely the same architecture as the SL network described above, however, given a position in the game as

input, it outputs a single value, denoting win or loss. Another difference is that the value network is trained not on the human games, but on the reinforcement learning network's games. The idea to approximate the value function utilizing reinforcement learning was also presented in [Lit94].

Therefore, the value, determining selection in the first stage of MCTS in AlphaGo, is a mix of both value network estimations and the Monte Carlo rollouts' results. Moreover, the rollouts were also improved in AlphaGo.

Guided Rollouts

So far we have not discussed, how AlphaGo aids the simulation stage of MCTS. The quality of rollouts can dramatically strengthen MCTS [SHM+16].

In Section 2.3.6 a few possible approaches were contemplated, such as *TD learning* and *pattern matching*. AlphaGo uses a relatively simple heuristic to guide rollouts: a linear classifier.

It works as follows: a small area around each legal move is taken and compared to one of the precomputed patterns. The input to the classifier is the array of indicators, saying that the area matches a particular pattern. The output of the classifier is the probability of the move being played. The classifier is again trained on the positions from KGS matches.

The advantage of the classifier described above is that it requires about 1000 times less computations than the neural network. It is very efficient when one needs to execute a great number of simulations in the limited time period.

AlphaGo takes every board state and runs the MCTS until it reaches 1600 simulations, and at that point the value network is used to decide which of the board positions are actually good ones, which ones are potential wins, and then it backtracks all those values all the way to the top of the network, then a very solid estimate of which moves are strong, and which moves are not so strong is obtained.

Learning Pipeline

To put it all together, the architecture of AlphaGo consists of two distinct processes: *learning* and *playing*. The former is an offline stage, during which the training of the neutral networks is performed. The output of this stage is the SL, RL and value network and rollout policy as well. The overview is in Fig. 2.11, see Figure 5. The second online stage is actually playing Go by traversing the game tree using MCTS. The search is enhanced by the predictors described above. The distinct stages of MCTS in AlphaGo are depicted in Fig. 2.11, see Figure 4.

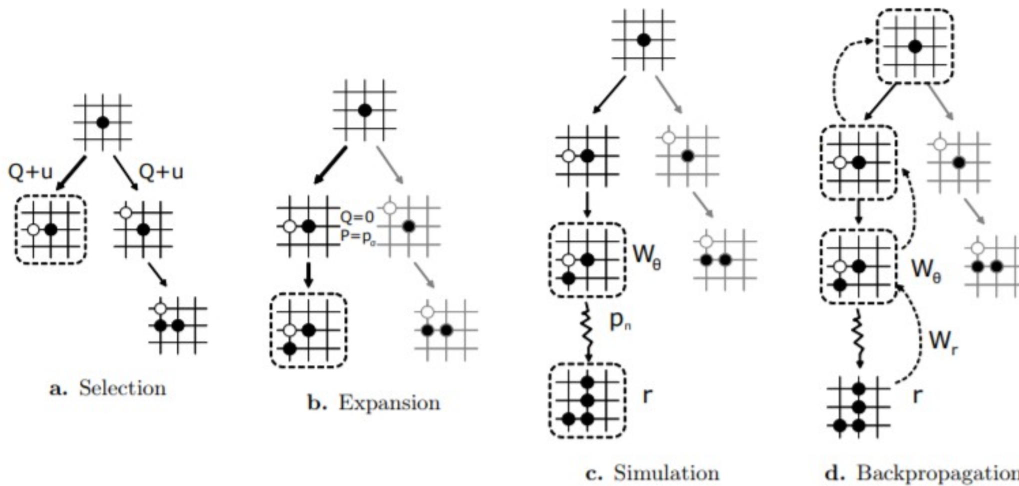


Figure 4: Monte Carlo Tree Search in AlphaGo. In the selection stage the decision is influenced by the prior probability from the SL network (a). The value of the node, Q, is calculated by both the rollout outcomes W_r and the value network estimations W_θ. The simulations are guided by the rollout classifier (c) and their result is then backpropagated (d).

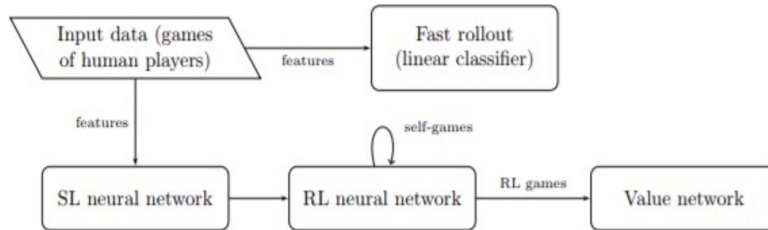


Figure 5: The learning pipeline of AlphaGo. SL denotes supervised learning, RL reinforcement learning

Figure 2.11. AlphaGo. Source: [AlphaGo16]

The Story of AlphaGo's Success

AlphaGo was launched as a research project by the Google team DeepMind in 2014. In October 2015 it became the first computer program to beat a human player in the full-sized game (on 19×19 board). That opponent was Fan Hui, a European Go champion, who possessed a rank of 2 dan out of 9 possible.

After that AlphaGo went on to play against one of the best players of Go, Lee Sedol, in March 2016. AlphaGo was able to defeat him winning four out of five games. That was a great milestone in the development of the AI.

In December 2016 Go community was baffled by an unknown player "Master" on the Go online server, who by the end of month managed to win over 30 top-ranked professionals without losing. In January 2017 DeepMind admitted that it was a new version of AlphaGo. In April 2017 AlphaGo defeated 3-0 Ke Jie, the last hope in Go for humankind.

2.4.1.2 Alpha Go Zero

AlphaGo Zero (AGZ) uses a technique which is a combination of deep learning with an improved Monte-Carlo Tree Search. As opposed to AlphaGo AGZ uses no human knowledge; it is trained solely by self-play reinforcement learning. AlphaGo uses a policy network and a value network, where as AGZ combines them into a single neural residual network which helps to avoid overfitting.

The Tree Search is simpler and does not perform any Monte-Carlo rollouts to evaluate positions, instead it evaluates them using the neural network.

Let us give more details on the engine.

It plays a game with subsequent states $s_1, \ldots, s_T$ (the final state s_T is a terminal position) as follows. A state s_1, together with its history, is given as input; it is accompanied by P a vector of probabilities and V, a winning probability from state s_1. P has as many components as edges starting from s_1; initially it is randomly given but P and V change with each iteration as a result of training.

In each state s_t a Monte-Carlo Tree Search MCTS (without roll out) is executed, as explained below, and an action a_t is executed using π_t as computed by the MCTS.

The neural network is trained taking s_t as input passing it through many convolutional residual layers with parameters θ, and outputting P_t, a probability distribution over the set of moves at s_t, and V_t the probability of the (current) player to win in position s_t. The neural network parameters θ are updated so as to minimize, using gradient descent, $l = (z - v(\theta))^2 + c|| \theta ||^2 - \pi' \log p(\theta)$, the first two summands of l giving the mean-square error and the last summand being the cross-entropy between π and p. The parameter c prevents overfitting.

The Monte-Carlo Tree Search is guided by the neural network approximately as follows. Each edge *(s,a)* stores a probability *P(s,a)*, a visit count *N(s,a)* and an action-value Q*(s,a)*. Starting from the root (state) it iteratively chooses moves that maximize $Q(s,a) + U(s,a)$, where $U(s,a)$ is proportional to $P(s,a)/(1+N(s,a)$ until a leaf vertex s' is encountered; s' is expanded and evaluated just once and the resulting probabilities and evaluation are stored. Then, backing up, to obtain the average of all evaluations V in the subtree below s .

The process is repeated; when the search is complete an output of search probabilities π is obtained proportional to $N^{1/\tau}$ where N is the visit count of each move and τ is a parameter controlling *temperature* (like quiescence denotes when a position has many threats or not).

2.4.1.3 Alpha Zero

In late 2017 AlphaZero was introduced, a single system that taught itself from scratch how to master the games of Chess, Shogi(Japanese Chess), and Go, beating a world-champion program in each case. The authors of AlphaZero were excited by the preliminary results and thrilled to see the response from members of the chess community, who saw in AlphaZero's games a ground-breaking, highly dynamic and "unconventional" style of play that differed from any chess playing engine that came before it. [AZblog18]

In [AlphaZero18] the full evaluation of AlphaZero is presented. It describes how AlphaZero quickly learns each game to become the strongest player in history for each, despite starting its training from random play, with no in-built domain knowledge but the basic rules of the game.

This ability to learn each game afresh, unconstrained by the norms of human play, results in a distinctive, unorthodox, yet creative and dynamic playing style. Chess Grandmaster Matthew Sadler and Women's International Master Natasha Regan, who have analysed thousands of AlphaZero's chess games for their book [SadlerRegan19] Game Changer (New in Chess, January 2019), say its style is unlike any traditional chess engine. "It's like discovering the secret notebooks of some great player from the past," says Matthew.

Traditional chess engines – including the world computer chess champion Stockfish and IBM's ground-breaking Deep Blue– rely on thousands of rules and heuristics handcrafted by strong human players that try to account for

every eventuality in a game. Shogi programs are also game specific, using similar search engines and algorithms to chess programs.

AlphaZero takes a totally different approach, replacing these hand-crafted rules with a deep neural network and general purpose algorithms that know nothing about the game beyond the basic rules.

To learn each game, an untrained neural network plays millions of games against itself via reinforcement learning. At first, it plays completely randomly, but over time the system learns from wins, losses, and draws to adjust the parameters of the neural network, making it more likely to choose advantageous moves in the future. The amount of training the network needs depends on the style and complexity of the game, taking approximately 9 hours for Chess, 12 hours for Shogi, and 13 days for Go.

The trained network is used to guide a search algorithm, known as Monte-Carlo Tree Search (MCTS), to select the most promising moves in games. For each move, AlphaZero searches only a small fraction of the positions considered by traditional engines. In Chess, for example, it searches only 60,000 positions per second, compared to roughly 60 million for Stockfish.

The fully trained systems were tested against the strongest hand-crafted engines for chess (Stockfish) and shogi (Elmo), along with the previous self-taught system AlphaGo Zero, the strongest Go player known.

- Each program ran on the hardware for which they were designed. Stockfish and Elmo used 44 CPU cores (as in the Top Chess Engine World Championship (TCEC)), whereas AlphaZero and AlphaGo Zero used a single machine with 4 first-generation TPUs and 44 CPU cores. A first generation TPU is roughly similar in inference speed to commodity hardware such as an NVIDIA Titan V GPU, although the architectures are not directly comparable.
- All matches were played using time controls of three hours per game, plus an additional 15 seconds for each move

In each evaluation, AlphaZero convincingly beat its opponent:

- In Chess, AlphaZero defeated the 2016 TCEC (Season 9) world champion Stockfish, winning 155 games and losing just six games out of 1,000. To verify the robustness of AlphaZero, we also played a series of matches that started from common human openings. In each opening, AlphaZero defeated Stockfish. We also played a match that started from the set of opening positions used in the 2016 TCEC world championship, along with a series of additional matches against the most recent development version of Stockfish, and a variant of Stockfish that uses a strong opening book. In all matches, AlphaZero won.
- In Shogi, AlphaZero defeated the 2017 Computer Shogi Association (CSA) world champion version of Elmo, winning 91.2% of games.
- In Go, AlphaZero defeated AlphaGo Zero, winning 61% of games.

Kasparov states in [Kasparov18]:

"I admit that I was pleased to see that AlphaZero had a dynamic, open style like my own. The conventional wisdom was that machines would approach perfection with endless dry maneuvering, usually leading to drawn games. But in my observation, AlphaZero prioritizes piece activity over material, preferring positions that to my eye looked risky and aggressive. Programs usually reflect priorities and prejudices of programmers, but because AlphaZero programs itself, I would say that its style reflects the truth. This superior understanding allowed it to outclass the world's top traditional program despite calculating far fewer positions per second. It's the embodiment of the cliché, "work smarter, not harder."

AlphaZero shows us that machines can be the experts, not merely expert tools. Explainability is still an issue—it's not going to put chess coaches out of business just yet. But the knowledge it generates is information we can

all learn from. Alpha-Zero is surpassing us in a profound and useful way, a model that may be duplicated on any other task or field where virtual knowledge can be generated. ” [Kasparov18]

As with Go, we are excited about AlphaZero’s creative response to chess, which has been a grand challenge for artificial intelligence since the dawn of the computing age with early pioneers including Babbage, Turing, Shannon, and von Neumann all trying their hand at designing chess programs. But AlphaZero is about more than Chess, Shogi or Go. To create intelligent systems capable of solving a wide range of real-world problems we need them to be flexible and generalise to new situations. While there has been some progress towards this goal, it remains a major challenge in AI research with systems capable of mastering specific skills to a very high standard, but often failing when presented with even slightly modified tasks. [Azblog18]

AlphaZero’s ability to master three different complex games, and potentially any perfect information game, is an important step towards overcoming this problem. It demonstrates that a single algorithm can learn how to discover new knowledge in a range of settings. And gives confidence in the mission to create general purpose learning systems that will one day help us find novel solutions to some of the most important and complex scientific problems.

2.4.2 Chess

2.4.2.1 Stockfish

The entirety of chess boils down to evaluating positions, and then searching for good moves. Let's start with evaluating positions. In order to tell how "good" a chess position is, Stockfish takes into account many things that human players do as well:

- Raw Material
  - Having more pieces is usually desirable
- General Piece Placement
  - Controlling center squares
  - Defending importat pieces
  - Attacking important enemy pieces
- Kings
  - Look for safety
- Pawn Formation
  - Double and tripled pawns are bad
  - Isolated pawns are bad.
  - Protected passed pawns are good
- Knights
  - Strong Knight in the center.
  - Knight on the rim is dim

- Bishops
  - Having the bishop pair
  - Controlling the diagonals
- Rooks
  - Rooks on (semi) open files

Each one of these attributes has its own unique weight that has been fine-tuned over the years. An exhaustive list of these attributes can be found in [StockfishEval19] .

Stockfish looks as far as it can (usually 30+ nodes deep), and uses heuristics based on the stage of the game to assign a value to how "good" the move really is based on the resulting position. To do this, Stockfish cleverly constructs a tree full of ever-increasing depth, consisting of moves followed by more moves, along with their values. While modern hardware can analyze hundreds of thousands of positions in seconds, multiple steps are still taken to "prune" and recognize patterns inside the tree.

A non-exhaustive list of techniques is as follows:

- Alpha-beta pruning
- Transposition tables
- Razoring
- Quiescence Search
- Opening heuristics
- Mid-game heuristics
- Iterative Deepening
- Aspiration Windows
- Parallel Search using Threads

For an exhaustive list of techniques see [StockfishEval19].

Most of the endgames are already solved. The engine simply compares its position to a table of possible positions (called 6-man Syzygy Tablebases), or uses yet another set of its own custom heuristics.

Let us mention that these techniques are found in nearly every decent chess engine nowadays, and are not what make Stockfish unique compared to other engines.

What truly makes Stockfish incredible compared to other engines is the community behind it, and very importantly its founders. Tord Romstad, Marco Costalba, and Joona Kiiski chose to open-source a top class engine in 2008 to the public so that anyone could contribute, and until 2014 Costalba led the project exceptionally well. Additionally, in 2013 the Stockfish community created Fishtest, which uses hundreds of thousands of dollars on hardware to test future versions of Stockfish against older versions of Stockfish to check performance gains in the newer version.

Stockfish is consistently ranked first or near the top of most chess-engine rating lists and was the strongest open-source conventional chess engine in the world until Leela Zero arrived in 2019. It won the unofficial world computer chess championships in season 6 (2014), season 9 (2016), season 11 (2018), season 12 (2018), season 13 (2018) and season 14 (2019). It finished runner-up in season 5 (2013), season 7 (2014) , season 8 (2015) and season 15 (2019) when it was defeated by Leela Chess Zero [WikiLCZ18].

2.4.2.2 Leela Chess Zero

Leela Chess Zero (**LCZero**, **lc0**) is a free, open-source, and neural network based chess engine and a distributed computing project. Development has been spearheaded by programmer Gary Linscott, who is also a developer for the Stockfish chess engine. Leela Chess Zero was adapted from the Leela ZeroGo engine, which in turn was based on Google's AlphaGo Zero project,also to verify the methods in the AlphaZero paper [AlphaZero17] as applied to the game of chess.

Like AlphaGo Zero, Leela Chess Zero starts with no intrinsic chess-specific knowledge other than the basic rules of the game. Leela Chess Zero then learns how to play chess by reinforcement learning from repeated self-play, using a distributed computing network coordinated at the Leela Chess Zero website.

As of June 2019, Leela Chess Zero had played over 226 million games against itself, and is capable of play at a level that is comparable with Stockfish, the leading conventional chess program.

The Leela Chess Zero project was first announced on TalkChess.com on January 9, 2018. This revealed Leela Chess Zero as the open-source, self learning chess engine it would come to be known as, with a goal of creating a strong chess engine.Within the first few months of training, Leela Chess Zero had already reached the Grandmaster level, surpassing the strength of early releases of Rybka, Stockfish, and Komodo, despite evaluating orders of magnitude fewer positions while using MCTS.

In December 2018, the AlphaZero team published a new paper in Science magazine revealing previously undisclosed details of the architecture and training parameters used for AlphaZero [AlphaZero18].These changes were soon incorporated into Leela Chess Zero and increased both its strength and training efficiency.

The basis of which Leela Chess Zero has used to self-learn, and play chess at a super human level is with reinforcement learning.This is a machine learning algorithm, mirrored from AlphaZero to be used by Leela Chess Zero, to maximize reward to make the engine a better chess player through self-play. From open-source, Leela Chess Zero has played hundreds of millions of games, run by volunteer users, in order to learn with the reinforcement algorithm. In order to contribute to the advancement of the Leela Chess Zero engine, the latest version of the Engine as well as the Client must be downloaded. The Client is needed to connect to the current server of Leela Chess Zero, which all of the information from the self-play chess games are stored, to obtain the latest network, generate self-play games, and upload the training data back to the server.[WikiLCZ18]

Some Competition Results.

In April 2018, Leela Chess Zero became the first neural network engine to enter the Top Chess Engine Championship (TCEC), during season 12 in the lowest division, division 4. Leela did not perform well: in 28 games, it won one, drew two, and lost the remainder. However, it improved quickly. In July 2018, Leela placed seventh out of eight competitors at the 2018 World Computer Chess Championship and in the next TCEC season, it won division 4 with a record of 14 wins, 12 draws, and 2 losses. After being promoted to division 3, Leela tied for 2nd place with Arasan, but did not advance.

By September 2018, Leela had become competitive with the strongest engines in the world. In the 2018 Chess.com Computer Chess Championship (CCCC), Leela placed fifth out of 24 entrants. The top eight engines advanced to round 2, where Leela placed fourth. Leela then won the 30 game match against Komodo to secure 3rd place in the tournament. Concurrently, Leela participated in the TCEC cup, a new event in which engines from different TCEC divisions can play matches against one another. Leela defeated higher-division engines Laser, Ethereal and Fire before finally being eliminated by Stockfish in the semi-finals.

In October and November 2018, Leela participated in the Chess.com Computer Chess Championship Blitz Battle. Leela finished third behind Stockfish and Komodo.

In December 2018, Leela participated in season 14 of the Top Chess Engine Championship. Leela dominated divisions 3, 2, and 1, easily finishing first in all of them. In the premier division, Stockfish dominated while Houdini, Komodo and Leela competed for second place. It came down to a final-round game where Leela needed to hold Stockfish to a draw with black to finish second ahead of Komodo. This was successfully managed, and therefore contested the superfinal against Stockfish. It narrowly lost the superfinal against Stockfish with a 49.5-50.5 final score.

In February 2019, Leela scored its first major tournament win when it defeated Houdini in the final of the second TCEC cup. Leela did not lose a game the entire tournament. In April 2019, Leela won the Chess.com Computer Chess Championship 7: Blitz Bonanza.Then on May 24, 2019 Leela lost to Stockfish in Computer Chess Championship 8: Deep Dive.

In May 2019, Leela defended its TCEC cup title, this time defeating Stockfish in the final 5.5-4.5 (+2 =7 -1) after Stockfish blundered a 7-piece tablebase draw. Leela also won the Superfinal of season 15 of the Top Chess Engine Championship 53.5-46.5 (+14 -7 =79) versus Stockfish. [Högy19]

Chapter 3

The Basic Elements

3.1 Letter Distribution

The game of Scrabble starts by drawing seven lettered tiles from a bag initially containing 100 tiles. In Spanish there are 44 vowels, 54 consonants, and two wild-card blanks. In English there are 42 vowels, 56 consonants, and two wild-card blanks. Each letter has an associated point value and a frequency, this is shown on Table 3.1 for Spanish and English. From now on, Spanish will be the language for our Scrabble examples and results, unless stated otherwise.

Spanish Distribution			English Distribution	
Letter	***Frequency***	***Value***	***Letter***	***Frequency***
# (Blank)	**2 tiles**	**0 points**	**# (Blank)**	**2 tiles**
A	**12 tiles**		**A**	**9 tiles**
E	**12 tiles**		**E**	**12 tiles**
I	**6 tiles**		**I**	**9 tiles**
O	**9 tiles**		**O**	**8 tiles**
U	**5 tiles**	**1 point**	**U**	**4 tiles**
L	**4 tiles**		**L**	**4 tiles**
N	**5 tiles**		**N**	**6 tiles**
R	**5 tiles**		**R**	**6 tiles**
S	**6 tiles**		**S**	**4 tiles**
T	**4 tiles**		**T**	**6 tiles**
D	**5 tiles**	**2 points**	**D**	**4 tiles**
G	**2 tiles**		**G**	**3 tiles**
B	**2 tiles**		**B**	**2 tiles**
C	**4 tiles**	**3 points**	**C**	**2 tiles**
M	**2 tiles**		**M**	**2 tiles**
P	**2 tiles**		**P**	**2 tiles**
F	**1 tile**		**F**	**2 tiles**
H	**2 tiles**	**4 points**	**H**	**2 tiles**
V	**1 tile**		**V**	**2 tiles**
Y	**1 tile**		**Y**	**2 tiles**
			W	**2 tiles**
CH	**1 tile**	**5 points**	**K**	**1 tile**
Q	**1 tile**			
J	**1 tile**		**J**	**1 tile**
LL	**1 tile**			
Ñ	**1 tile**	**8 points**		
RR	**1 tile**			
X	**1 tile**		**X**	**1 tile**
			Q	**1 tile**
Z	**1 tile**	**10 points**	**Z**	**1 tile**

Table 3.1. Distribution of letters in Spanish and English

When Alfred Butts invented the game, he initially experimented with different distributions of letters [Whitehill]. A popular story claims that Butts created an elaborate chart by studying the front page of *The New York Times* to create his final choice of letter distributions [ScraD].

In Spanish there are no news about a study to determine frequency and values of the letters. Table 3.1 suggests that letter distribution in Spanish seems to be mostly inherited from English. There are minor changes like having 12 A's instead of 9 (this is a good change since in Spanish there are more words containing an A than in English). For instance, approximately 1/10 of the words in the DRAE ("*Diccionario de la Real Academia*") start with the letter A, but probably the biggest mistake was the change of value of the letter Q. Inexplicably its value dropped from 10 points to only 5 points! This is hard to believe since it is much harder to place the Q on the board when playing in Spanish rather than in English. In Spanish there are no 2-letter words that contain Q, and in English there is one, the most important: QI (plural QIS) . In Chinese philosophy, *qi*, also spelled *chi* or *ch'i*, is the life force that every person and thing has.

Although it is most commonly spelled *CHI* in standard usage, the variant form *QI* is the single most-played word in English Scrabble tournaments, according to game records of the North American Scrabble Players Association (NASPA).

In English most of the words that contain Q are of the form QU + something else; nevertheless there are 101 exceptions (according to the 2015 Collins Words Dictionary (CSW15), there are exactly 101 words with q without u following). For example: QI/S, QAT/S, QAID/S, CINQ/S, TRANQ/S, SHEQEL/S, QABALA/S, SHEQALIM, etc. But in Spanish there are none!; in order to put the letter Q it has to be accompanied by the letter U, always QU + something else; this also happens in Catalan with one exception the word QATAR. Catalans wisely decided to make the tile Q equivalent to QU; therefore, if it were Spanish, you could put QE instead of QUE, QI instead of QUI, etc. The Spanish Scrabble community preferred to deal with the Q tile without including the letter U, as it is done in English. Despite the existence of some strategies to deal with this problem, many times, the player that has the Q tile at the endgame loses, regardless of being many points ahead.

To determine Scrabble's tile values, Alfred Butts carefully analyzed letter frequency in various periodicals, including the front page of the New York Times. Butts' original design aimed to create a game that would balance skill with luck, which he believed to be an important aspect of the game he loved [ScraTileValues].

But does Butts' original design stand up to rigorous computational analysis today? Two researchers recently developed programs to process and analyze far more data and variables than could have been considered by Butts 78 years ago. Director of Research at Google, Peter Norvig came up with a model for letter-frequency counts based on the Google Books English-language corpora; see Norvig's research [Norvig13]. More directly related to Scrabble, Joshua Lewis (a post-doc at the University of California, San Diego's Cognitive Science Department) proposed new tile values considering three variables: the frequency of letters by word length (the most valuable word lengths in Scrabble being two-,three-,seven- and eight-letter words), and the ease with which one can play a letter (the blank tile and S are far easier to play than Q). Using a program he created called Valett that accounts for these concerns, Lewis proposes 14 tile-value changes. These include X changing from 8 to 5 points, Z changing from 10 to 6 points and J changing from 8 to 6 points; see Lewis' full findings [Lewis12].

After playing thousands of games in Spanish, we have gathered Word Statistics which could be used to perform this kind of computer analysis and therefore give a more adequate frequency letter distribution and much better letter values for Spanish Scrabble.

3.2 Scrabble Rules

When playing Scrabble, two to four players may play the game. The object when playing is to score more points than other players. As words are placed on the game board, points are collected and each letter that is used in the game will have a different point value. Bonus squares enhance the value of your plays.

3.2.1 The Scrabble Board, notation and evaluation of words

A standard Scrabble board will consist of cells that are located on a square grid. The board offers 15 cells high and 15 cells wide. The tiles used on the game will fit in each cell on the board. The Scrabble board denotes columns by the labels A through O and the rows by labels 1 through 15. This labelling allows us to refer to squares and words on the board. There are premium squares, usually denoted with different colours according to its type. They are labelled on Fig. 3.1.

More formally, let us denote the unloaded board by $ß=\{1,2,\ldots,15\}\times\{1,2,\ldots,15\}$, which we also denote by $\{1,2,\ldots,15\}\times\{A,B,\ldots,O\}$ (See Fig. 1). Elements of ß are called cells or squares.

Four types of premium squares appear on the board: 2L, 3L, 2W, 3W, with the following meanings: 2L= Double Letter Value, 3L= Triple Letter Value, 2W= Double Word Value, 3W= Triple Word Value (See Fig. 3.1 for more details).

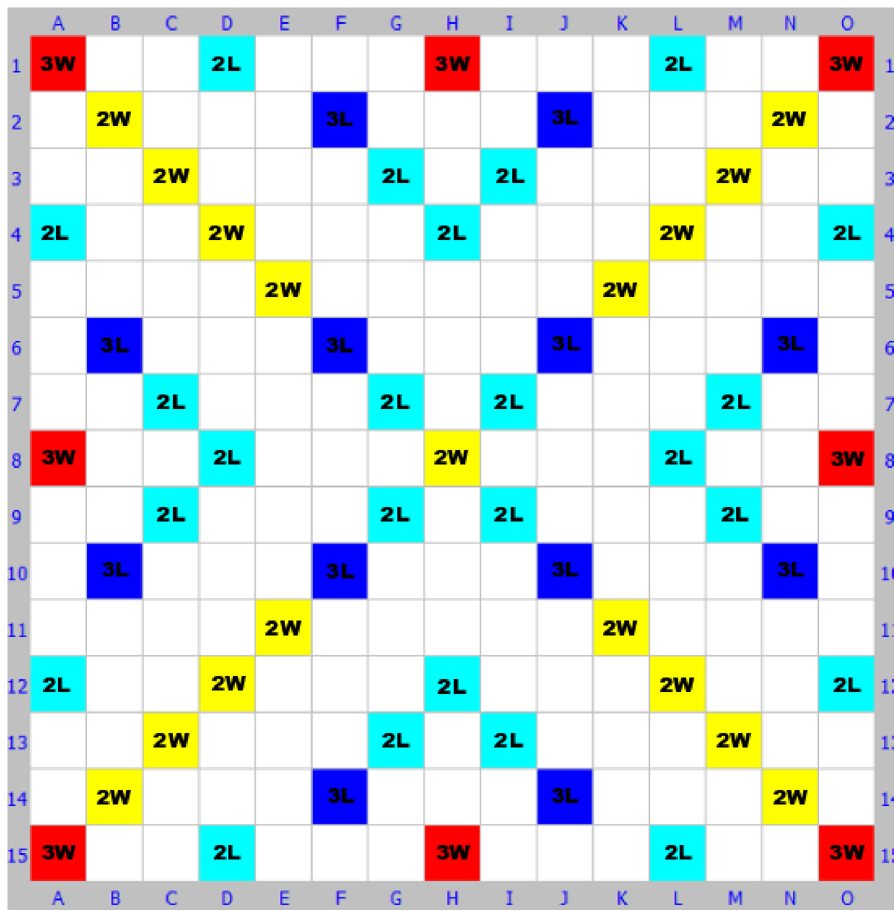


Fig. 3.1 Board Characteristics

A *played board* is a subset T of ß together with a function assigning to each element of T a letter of the alphabet used {A,B,…,Z}; the subset T alone is also called a *played board* .

When playing Scrabble a *lexicon* is used, which is the collection of *valid words*.

Word notation and evaluation

All words will have a left to right orientation, or an up to down orientation. When referring to words on the board the labelled coordinates of its initial letter are used. A horizontal word is denoted if the row number is first written followed by the column letter. A vertical word is denoted if the column letter is first written followed by the row number.

Let us define a *located string* as a word together with its coordinates. A located string is not necessarily a valid word.

For example 8D FORMAL is a horizontal word that starts at square 8D, H4 FORMAL is a vertical word that starts at square H4. 8D FORMAL, H4 FORMAL and H4 FRMLOA are examples of located strings; observe that FRMLOA is not a valid word since it is not contained in the lexicon. The first word has to touch the center, see 3.2.4 *The First Word Score*.

When evaluating a word, premium squares have to be taken into consideration. For every word formed in a move, if a tile contained in each word is placed for the first time on a 2L (resp. 3L) type of square, its tile value is multiplied by a 2 (resp. 3) factor. For each word formed in a move, if a tile contained in each word is placed for the first time on a 2W (resp. 3W) type of square, its word value (the sum of all its tile values) is multiplied by 2 (resp. 3) factor. After a premium square is used, its factor becomes 1, as all normal (non-premium) squares or cells.

Example, evaluate 8D FORMAL: Since the tile F hits a 2L square, then its value is 4*2=8, plus the contribution for all other tiles, that is, 4*2 + 1 + 1 + 3 + 1 + 1 = 15, now since a tile of the word formed hits a 2W square, the total evaluation is 15*2 = 30.

Suppose 8D FORMAL is already on the board, then if we placed the vertical move E4 CORPORAL the evaluation for this word is the sum of its tiles 3+1+1+3+1+1+1+1=12 multiplied twice by a 2 factor since it touches two 2W premium squares. Then we have 12*2*2=48, finally since the word CORPORAL has length 8, and there was only one letter on the board (letter O on E8), then this means that 7 letters from a player's rack were used; when this happens a 50 point reward is given (see 3.2.9 *The Fifty Point Bonus*), therefore the final score of the move is 48+50=98. Remember that these premium squares, once used, will no longer give a two factor reward, since its factor becomes one.

3.2.2 Scrabble Tiles

In Spanish and in English, there are 100 tiles that are used in the game; but in French 102 tiles are used. These tiles contain letters with point values. There are 2 blank tiles (with zero value) that can be used as wild tiles to take the place of any letter. When a blank is played, it will remain in the game as the letter it substituted.

Different letters in the game will have various point values and this will depend on how rare the letter is and how difficult it may be to lay that letter. For more information see the above section 3.1 Letter Distribution.

3.2.3 Starting the Game

Without looking at any of the tiles in the bag, players will take one tile at a time. The player that has the letter that is closest to "A" will begin the game. A blank tile will win the start of the game. The tiles are then replaced to the bag and used in the remainder of the game.

Every player will start the turn by drawing seven tiles from the Scrabble bag. There are three options in any turn. The player can place a word, he can exchange tiles for new tiles or he can choose to pass.

3.2.4 The First Word Score

In Scrabble (in any language) all words played or formed have to be of length greater than one and less than sixteen tiles. All words are read from left to right or from top to bottom.

The first horizontal or vertical word as well as all other words played or formed in the game must be valid words, they must be contained in the proper lexicon (see section 3.2.6 *Accepted Scrabble Words*).

When the game begins, the first move placed on the board must use the central square (8H) of the board. The central square is a double square and will offer a double word score. All players following will build their words off of this word, extending the game to other squares on the board. This is explained with more detail in the next section 3.2.5 *Beyond the First Word*.

3.2.5 Beyond the First Word

Let us write some definitions that will help us explain when a move is valid.

Let σ be the cell on row i and column j, and let σ' be the cell on row i' and column j'. Then the *Manhattan* (or *taxi*) *distance* from σ to σ' is $d(\sigma,\sigma') = |i - i'| + |j - j'|$, where $|\ |$ denotes absolute value.

If A and B are nonempty subsets of the board, the distance between them is: $d(A,B) = \min\{ d(a,b): a \in A, b \in B \}$.

Let T be the set of tiles which have been played at a certain point. A *molecule* of T on row i (resp. column j) is a maximal string of consecutive cells of T on row i (resp. column j).

If the first word has been made, then the following 3 properties have to be satisfied to make a valid move.

1) The set of tiles being played is contained in a row or a column.
2) At least one of the tiles being played is at (Manhattan or taxi) distance 1 from T (so the move is connected to the played board).
3) The vertical and horizontal located strings (of length greater than 1) formed by the moves must be valid words (see next section 3.2.6). If all tiles are played on row i (resp. column j) the horizontal (resp. vertical) located string formed is the union of the set S of tiles played with the molecules on row i (resp. column j) at distance 1 from S.

Some bilingual examples of valid moves in Spanish and English, using the board shown in Figure 3.2 and the rack: {aoodgln} are: H1 gondola(s), D8 (b)ongo, E7 d(o), F7 (o)x, F7 a(x), 7E nodal, 7G do, 7E dona.

Some examples of illegal moves are: D8 (b)ongo and F7 a(x) as a single move, because this move is not contained in a row or a column (violates 1); playing 11E gondola or J6 dona both moves have distance greater than 1 from a tile already on the board (violates 2) ; I7 gondola is an illegal move since 8D boxeso is not a valid word, 9D dona is an illegal move since D8 bd, E8 oo, F8 xn are not valid words (the previous moves violate 3).

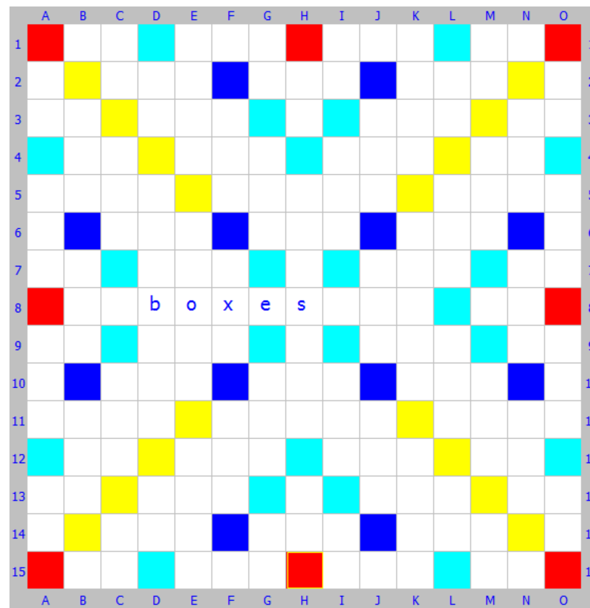


Fig. 3.2 rack: {aoodgln}; valid words are: H1 gondola(s), D8 (b)ongo, E7 d(o), F7 (o)x, F7 a(x), 7E nodal, 7G do, 7E dona.

3.2.6 Accepted Scrabble Words

When playing informal games of Scrabble in Spanish or English, any standard Spanish or English dictionary can be used to help us check the legality of the words played. Nevertheless for tournament games of Scrabble, there are Official Scrabble Dictionaries which served for the generation of lexicons that have to be used to check the legality of words.

The Official Spanish Dictionary is: "*El Diccionario de la Real Academia Española (DRAE), XXII Ed.*" from which the complete Spanish lexicon was constructed; the lexicon contains approximately 637,000 words.

In English, for serious games, things get more complicated since there is no unique lexicon.

OWL2 and OSPD5

The North American 2006 *Official Tournament and Club Word List Second Edition (OWL2)* went into official use in American, Canadian, Israeli and Thai club and tournament play on March 1, 2006 (or, for school use, the bowdlerized *Official Scrabble Players Dictionary, Fifth Edition (OSPD5)*).

The OWL2 and the OSPD5 are compiled using four (originally five) major college-level dictionaries, including Merriam-Webster (10th and 11th editions, respectively). If a word appears, at least historically, in any one of the dictionaries, it will be included in the OWL2 and the OSPD5. If the word has only an offensive meaning, it is only included in the OWL2. The key difference between the OSPD5 and the OWL2 is that the OSPD5 is marketed for "home and school" use, with expurgated words which their source dictionaries judged offensive, rendering the *Official Scrabble Players Dictionary* less fit for official *Scrabble* play.

Collins Scrabble Words

In all other countries, the competition word list is *Collins Scrabble Words 2015* edition, known as *CSW15*. Versions of this lexicon prior to 2007 were known as SOWPODS. The lexicon includes all allowed words of

length 2 to 15 letters. This list contains all OWL2 words plus words sourced from Chambers and Collins English dictionaries. This book is used to adjudicate at the World Scrabble Championship and all other major international competitions outside of North America [WikiScrabbleA].

3.2.7 Replacing Scrabble Tiles

Once tiles are played on the board, players will draw new tiles to replace those. Players will almost always have seven tiles during the game. Most of the endgame the players will have less than 7 tiles. The endgame begins when there are zero tiles left inside the bag. Drawing tiles is always done without looking into the bag so that the letters are always unknown.

3.2.8 Exchanging Tiles

The rule concerning when you can exchange tiles differ depending on the language played. When playing in Spanish it is possible to exchange tiles if the bag has at least one tile. In English and in French one can only exchange tiles if the bag has at least 7 tiles.

3.2.9 The Fifty Point Bonus

Exciting rewards can come when players use all seven tiles to create a word on the board. When this happens, players will receive a 50 point bonus, in addition to the value of the word. If the game is near the end and players are not holding seven tiles, they do not get the bonus for using all of their tiles. This is only collected when using all seven tiles from one's rack.

3.2.10 The End of a Scrabble Game

Once all tiles are gone from the bag and a single player has placed all of their tiles, the game will end and the player with the highest score wins. If both players are unable to place all their tiles, the game ends and both players subtract the values of their remaining tiles to their score to get the final score of the game.

3.2.11 Tallying Scrabble Scores

When the game ends, each player will count all points that are remaining on their tiles that have not been played. This amount will be deducted from the final score.

An added bonus is awarded to the player that ended the game and has no remaining tiles. The tile values of all remaining players will be added to the score of the player who is out of tiles to produce the final score of the game.

The Scrabble player with the highest score after all final scores are tallied wins [ScrabbleRules].

3.3 What's different about competitive Scrabble?

Friendly games of Scrabble may involve three or four players, but competitive Scrabble is strictly a one-on-one activity. Increasing the number of players drastically increases the luck factor, as you are only allocated a small handful of the 100 tiles to play with. Competitive Scrabble tournament games are played one-on-one, with a clock. In Spanish (resp. English) each player gets 30 (resp. 25) minutes to complete all of his/her turns, and a good player will average more than 450 (resp. 400) points a game.

3.3.1 Ending the Game

Besides the standard ways stated above to end the game, when playing in tournament, there are some special rules to end the game. Usually, these special rules differ according to the language of play.

When playing in Spanish or in English the following rules apply:

Twice Pass Rule (for both languages)

The game ends if all players have passed twice in consecutive turns.

Six Zero Rule (only when playing in English)

The game may also end by either player neutralizing the game timer after a sixth successive zero-scoring play from passes, exchanges, successful challenges, or illegal plays, regardless of the score.

Twelve Zero Rule (for Spanish play, in English six turns suffice to claim the end of the game)

The game may also end by either player neutralizing the game timer after a twelfth successive zero-scoring play from passes, exchanges, successful challenges, or illegal plays. This rule applies in Spanish play only if the score is different from 0-0.

If any of the above two rules occurs, each player´s final score is reduced by the total value of the tiles on his or her rack.

For more Information about the tournament rules of the game in Spanish refer to [RegFISE]. In English refer to [RulesNASPA] for North America type of rules (North America Scrabble Players Association, NASPA), or to [RulesWESPA] for almost the rest of the English speaking world (World English-Language Scrabble Players Association, WESPA).

Chapter 4.

History and State-of-the-Art of Computer Scrabble

4.1 Generation of valid moves

As to the generation of valid moves, Appel and Jacobson [AppelJacobson88] introduced an algorithm, which proved to be the fastest and more efficient at one time. It is based on the data structure DAWG (*Directed Acyclic Word Graph*) derived from the entries of a reference lexicon. Steve Gordon [Gordon94] introduced a variant of this data structure (GADDAG) which occupies 5 times more space than DAWG, but it duplicates the speed of move generation.

4.1.1 The DAWG

Definitions

An *alphabet* is a finite, non-empty set of symbols. Let us use the symbol Σ for an alphabet. A *word* (or sometimes *string*) is a finite sequence of symbols chosen from an alphabet. For a word *w*, we denote by $|w|$ the length of *w*. The empty word ε is the word with no symbols. Let *x* and *y* be words. Then *xy* denotes the concatenation of *x* and *y*, that is, a word formed by making a copy of *x* and following it by a copy of *y*. We denote as usual by Σ^* the set of words over Σ and by Σ^+ the set $\Sigma^* - \{ \varepsilon \}$. A word *w* is called a prefix of a word *x* if there is a word u such that x = wu. It is a proper prefix if $u \neq \varepsilon$.

A finite (not necessarily complete) deterministic automaton is specified by five pieces of information:
A = (S, Σ, ι, δ, F) where S is a finite set called the set of states (or vertices), Σ is the finite input alphabet, ι is a fixed element of S called the initial state, δ is a partial[1] function from S × Σ to S called the transition function, and F is a subset of S called the set of terminal states [Lawson04].

We think of A as a directed graph with S as its set of vertices and, if δ(s, a) is defined, there are as many arcs from s to δ(s, a) as there are elements $a' \in \Sigma$ such that δ(s,a')= δ(s,a).

We define L(A), the language accepted by A, to be the set of all strings in Σ* that label paths in A starting at the initial state ι and ending at a terminal state.

A DAWG D is a finite deterministic automaton with finite L(D), whose initial state ι has no incoming arcs and which is minimal, that is, no finite deterministic automaton with less vertices than D accepts the same language as D.

1A partial function from $S \times \Sigma$ to S is a function defined on a subset of $S \times \Sigma$ with values in S.

It is easy to see that, if F = ∅, then D = (S, Σ, ι, δ, F) has only one vertex and no arcs. In what follows we will assume F ≠ ∅.

The fact that ***D*** = (*S, Σ, ι, δ, F*) is minimal implies that the set *S'* of vertices that belong to some path from *ι* to a point of *F* is *S* , because if not , eliminating *S-S'* and the arcs with some endpoint in *S-S'* , one would obtain a deterministic automaton with less vertices than ***D*** with the same language as ***D***. Thus, for each vertex *v* of ***D*** there is at least one path from *ι* to *v* and one path from *v* to a vertex of *F* .

A consequence of this is that the set of strings corresponding to paths starting at *ι* is the set of all prefixes of words of *L(**D**)* .

D is acyclic because if there were a cycle (of positive length) *C* starting and ending at a vertex *v* then the concatenation $w_1cc...cw_2$, where the number of *c's* is arbitrarily large, would belong to *L(**D**)* contradicting that *L(**D**)* is finite (here w_1 , w_2 and *c* are the strings determined, respectively, by a path from *ι* to *v* , a path from *v* to a point of *F* and the cycle *C*).

The fact that the finite digraph ***D*** is acyclic implies that ***D*** has at least one *sink* (a vertex with no outgoing arcs) and, as ***D*** is minimal and *F* is nonempty, every sink *v* belongs to *F* ; otherwise, eliminating *v* and the arcs with *v* as an endpoint, one would have a deterministic automaton with less vertices and the same language as D . In fact, D has exactly one sink because if v1 , v2 were different sinks, identifying v1 with v2 and identifying arcs e1 , e2 with the same origin and the same label, ending respectively at v1 and v2 , one would have a deterministic automaton with less vertices and the same language as D . A similar argument shows that D has only one source (a vertex with no incoming arcs), namely ι .

Now consider the DAWG D whose accepted language is the lexicon of valid words in Spanish (or English, French, or any other language). This DAWG is unique by [Lawson04 Thm. 7.4.2].

We now describe how D is used in [AppelJacobson88] to generate words.

Suppose one has a nonempty played board T and w is a string contained in row i, not contained in T and at distance ≤ 1 from T. The anchor of w is the leftmost cell σ = (i,j) of w at distance 1 from T. That is, no cell (i, k) of w with k < j is at distance 1 from T.

For each cell σ at distance 1 from T:

(1) Find all possible "left parts" of strings w with anchor σ (a left part of w consists of those tiles to the left of σ).

(2) "For each left part" L found above, find all matching "right parts" R containing σ and possibly molecules of row i, that is, the concatenation of L and R as well as the vertical words formed of length greater than 1 must be valid words.

The characteristic of A mentioned above makes (1) and (2) an efficient task.

Let us look at an example of a Directed Acyclic Word Graph (DAWG) in Figure 4.1 [AppelJacobson88].

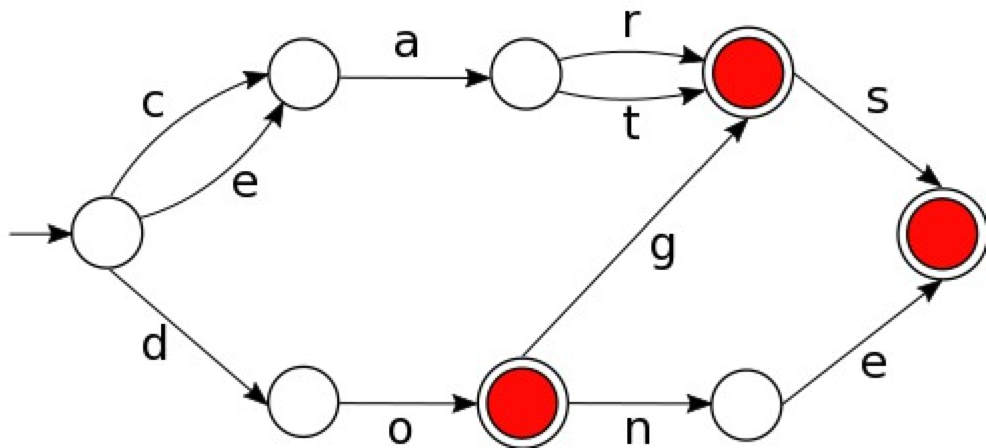


Fig 4.1. An example of a Directed Acyclic Word Graph (DAWG).

Notice that each word in the lexicon corresponds to a path from the root to a terminal node. When two words begin the same way they share the initial parts of their paths. The node at the end of a word's path is called a *terminal* node. Thus, observe that the small Lexicon above is represented by the DAWG in Figure 4.1.

The problem of move generation in the Scrabble game is complicated by the presence of *blank* tiles, which represent any letter from the alphabet used. The presence of blank tiles in the rack greatly increases the number of moves possible at a given turn. The time spent in searching increases accordingly [AppelJacobson88]. Appel and Jacobson write: "It is almost always possible to tell when our program holds a blank tile by the noticeable delay before a move is made. When the program gets both blanks simultaneously, it seems to slip into a coma for a few seconds."

4.1.2 The GADDAG

One may think of a GADDAG as a handy representation of a list of words, such that the following questions can be answered quickly.

- Given a substring of a word in the dictionary, which letters can be added to the left of it to stay a substring of a word in the dictionary?
- Given a prefix of a word in the dictionary, which letters can be added to the right of it to stay a prefix of a word in the dictionary?

How does the GADDAG do that? Well, it is a DAG (directed acyclic graph) where we start anywhere in the middle of the word, and read the letters right-to-left from there. Once we get to the left end, we switch directions and read the remaining letters left-to-right until we hit the right end [Wouter16].

Formally, the GADDAG of a collection L of words is the DAWG for {INV(x)◡y: xy ∈L and x is nonempty}. Here INV(x) is the string x written backwards and ◡(a delimiter indicating change of direction) is a symbol added to the alphabet of L, INV(x)◡y is the concatenation of INV(x), ◡and y; xy is the concatenation of x and y.

The GADDAG of {car, cars, cat, cats,do,dog,dogs,done,ear,ears,eat,eats} is displayed in Fig. 4.2. It uses the same lexicon as the DAWG example shown in Fig. 4.1, it has more paths for every word. Each word is represented in as many paths as the number of letters it has. Although in general the GADDAG occupies 5 times more space than the traditional DAWG, it duplicates the speed of move generation.

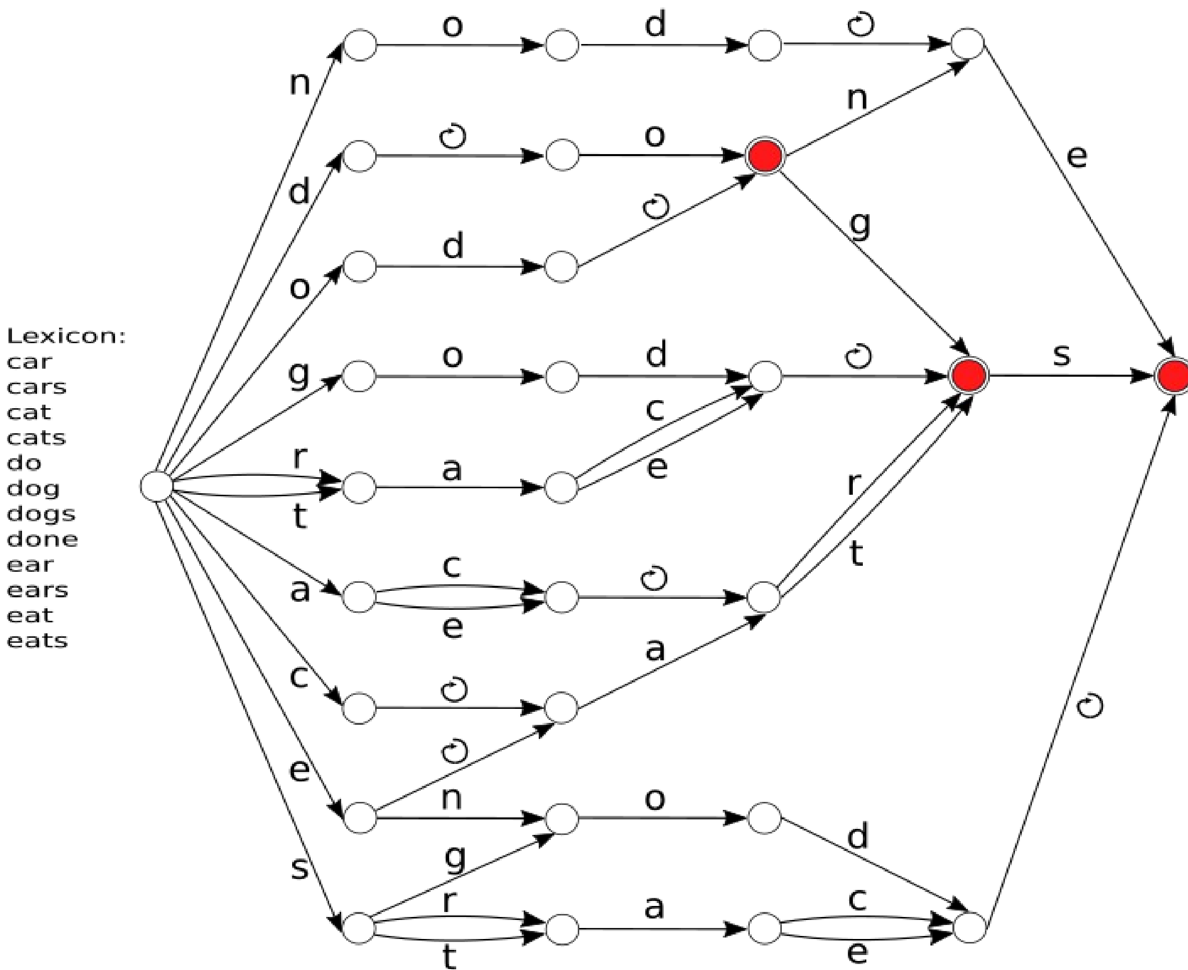


Fig. 4.2 The GADDAG of {car,cars,cat,cats,do,dog,dogs,done,ear,ears,eat,eats}

4.1.3 An example of a DAWG and a GADDAG

We now consider an example. Suppose that in Fig. 4.3 and with the rack { D, E, E, I, M, N, T}, we want to find all moves on row 11 having 11D as anchor.

The left part is empty or a string of length at most 3 at distance 1 from 11D occupying a subset of {11A,11B,11C}. Such a string must be a subset of the rack above. The number of such strings, including the empty one, is 173. Out of these, exactly 83, the ones which are prefixes of length ≤ 3 of words in the lexicon, appear as left parts. These are the alternating, the $C\ V\ V$, the V n C and the $V\ V$ strings with the following exceptions and additions:

Exceptions: ned, tid, tee.
Additions: eid, ein, etm, etn, mn, mne.

Here alternating means: no 2 vowels and no 2 consonants are adjacent; where V stands for vowel and C stands for consonant.

Of these 83 left parts, exactly 22 valid words arise by completing with right parts. The 22 valid words are: ideen, meen, dime, ende, ente, idee, mene, mete, mide, mine, neme, teme, tene, time, eme, ene, mee, mie, me, ne, de, te.

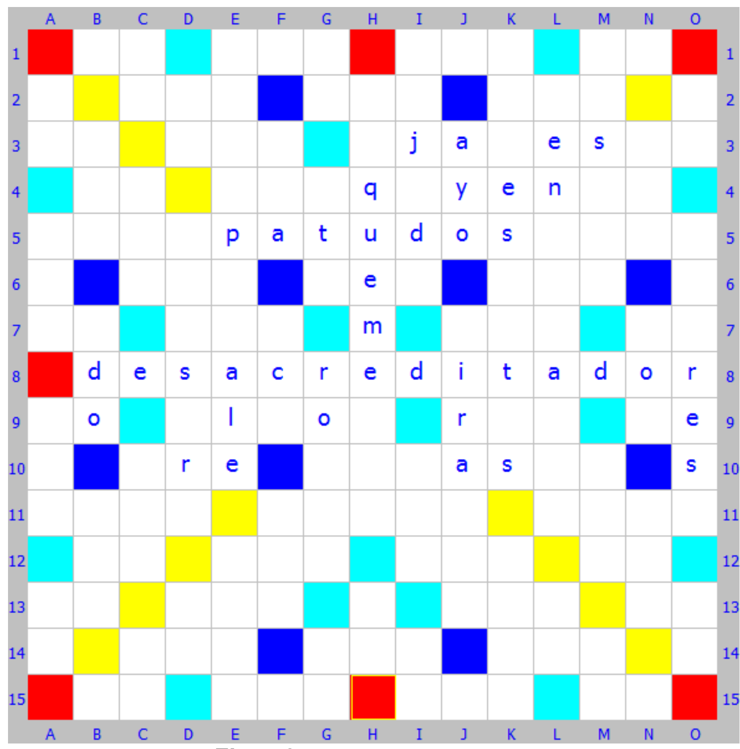


Fig. 4.3 A certain board position

If instead of DAWG one uses GADDAG one starts with “e”, in our example the only letter which can be placed on the anchor 11D, and then goes leftwards generating succesively the 47 strings: e, de, ee, ie, me, ne, te, dee, ... , time, tine; with no need to consider the remaining (83 – 47 = 36) prefixes; these 47 left parts yield, by completing with right parts as above, the 22 valid words listed previously. The reason for the economy is that in GADDAG one starts at the anchor.

4.2 Scrabble engines

Once we have a move generator, the implementation of basic engines simply based on the move that maximizes the score in each turn, is simple. Shapiro [Shapiro79] and Stuart [Stuart82] are examples of precursors in the development of engines for Scrabble. To solve the obvious deficiencies of this greedy approach, simulation techniques were applied. These techniques took into account the possible replies to a candidate move, the replies to those replies, and so on for many plies. This method, known as simulation, rests on Monte Carlo sampling for the generation of the opponent’s rack, in situations with uncertainty, and it has its theoretical base on the Minimax technique of Game Theory.

The program *Maven* [Sheppard2002a] developed by Brian Sheppard is one of the references for this paradigm and its excellent results against top-level players are the best demonstration of the appropriateness of this approach. Later Jason Katz-Brown and John O'Laughlin [KatzO’Laughlin06] have implemented *Quackle*, a powerful artificial intelligence Scrabble player, which also exploits the simulation technique.

4.2.1 Maven

Maven is an artificial intelligence Scrabble engine, created by Brian Sheppard. The first version dates back to 1986; then many other different versions came. Maven was sold to Hasbro in 1995. It is not clear the date of Maven´s last version, perhaps because it is not an open source code.

The first thing needed in a Scrabble engine to start playing is an algorithm to generate legal moves.

Move Generation

Maven's first move generator was named the Bit Parallel Generator, an explanation of this generator is given in [Sheppard02b] pp 50-51. In future versions the author switched to the Appel-Jacobson Move Generator which represents the lexicon as a directed acyclic word graph (DAWG). See [AppelJacobson88].

Maven's Game play

Maven's game play is sub-divided into three phases: The "regular" phase, the "pre-endgame" phase and the "endgame" phase.

The "regular" phase lasts from the beginning of the game up until there are nine or fewer tiles left in the bag. The program uses a rapid algorithm to find all possible plays from the given rack, and then part of the program called the "kibitzer" uses simple heuristics to sort them into rough order of quality. The most promising moves are then evaluated by "simming", in which the program simulates the random drawing of tiles, plays forward a set number of plays, and compares the points spread of the moves' outcomes. By simulating thousands of random drawings, the program can give a very accurate quantitative evaluation of the different plays. (While a Monte-Carlo search, Maven does not use Monte-Carlo tree search because it evaluates game trees only 2-ply deep, rather than playing out to the end of the game, and does not reallocate rollouts to more promising branches for deeper exploration; in reinforcement learning terminology, the Maven search strategy might be considered "truncated Monte-Carlo simulation". The shallow search is because the Maven author argues [Sheppard02a] that, due to the fast turnover of letters in one's bag, it is typically not useful to look more than 2-ply deep, because if one instead looked, e.g. 4-ply, the variance of rewards will be larger and the simulations will take several times longer, while only helping in a few exotic situations. As the board value can be evaluated with very high accuracy in Scrabble, unlike games such as Go, deeper simulations are unlikely to change the initial evaluation. [WikiMavenScrabble18]

The "pre-endgame" phase works in almost the same way as the "regular" phase, except that it is designed to attempt to yield a good end-game situation.

The "endgame" phase takes over as soon as there are no tiles left in the bag. In two-player games, this means that the players can now deduce from the initial letter distribution the exact tiles on each other's racks. Maven uses the B* search algorithm [Sheppard03] to analyze the game tree during the endgame phase.

Analysis of games played by Maven's simulation engine suggest that Maven, back in 2002, had come close to the skill level of human champions. See [Sheppard02a, Sheppard02b].

4.2.2 Quackle

Quackle is an open source crossword game engine and analysis tool developed by Jason Katz-Brown and John O'Laughlin in 2006 [KatzO'Laughlin06].

Quackle's AI in a Block Diagram

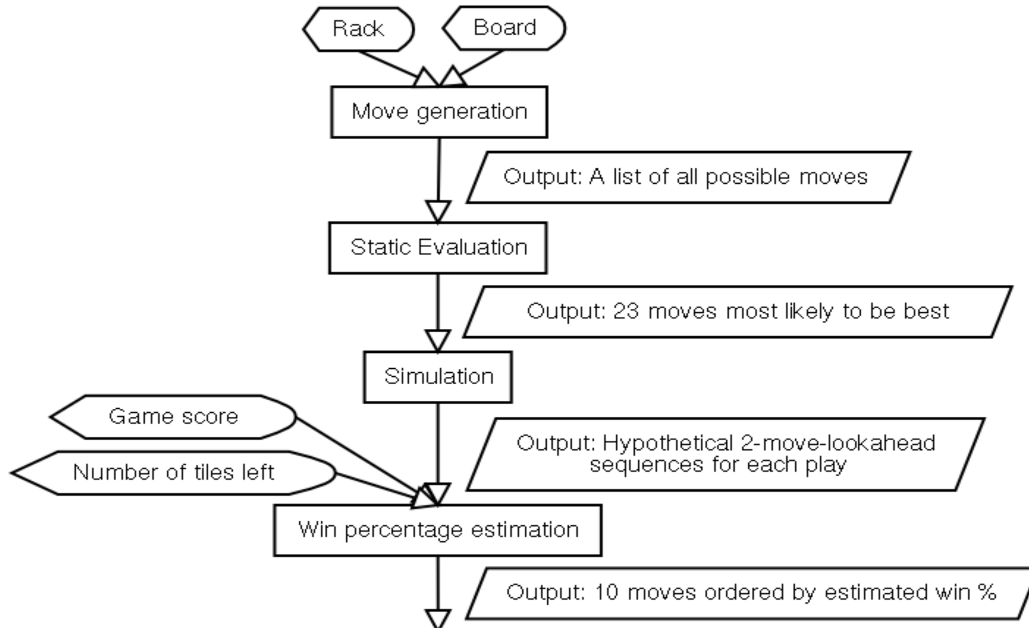


Fig. 4.4 How Quackle plays Scrabble [KatzO'Laughlin06]

The explanation of each block of Fig. 4.4 is given below. [KatzO'Laughlin06]

Move Generation

This block takes a rack and board as input and very quickly outputs a list of all possible plays that can be made. Quackle's ultrafast move generator uses the Scrabble lexicon converted into a GADDAG. The GADDAG data structure was originally designed by Steven A Gordon, and uses the move generation algorithm he outlined in *A Faster Scrabble Move Generation Algorithm* [Gordon94] .

Static Evaluation

This block takes a list of moves as input and very quickly outputs a list of moves ordered by a rough guess at how strong they are. For each play, this "static evaluation", so called because there is no look-ahead involved, is simply the score of the play plus an estimated "leave value" of the letters left on the player's rack after making the play. For instance, if the rack is ACEENOR, and Quackle wants to statically evaluate a play of OCEAN where it scores 19 on the board, the result is 19 plus the value of the leave ER. This leave value is gleaned from a table lookup into a large precomputed database of all possible one to six letter leaves.

For leave (rack) evaluation, Quackle uses an "exhaustive" architecture, where the program has a different leave (rack) evaluation parameter for each of the nearly a million (914,624) possible combinations of 1 to 6 tiles. With the advances in computer power over, it has become possible to tune such large parameter sets.

Here is a typical sequence of entries:

Leave	EQYZ	EQZ	ER	ERR	ERRR
Value	-4.68	-4.12	4.79	0.39	-9.02

So the final static evaluation of the OCEAN play is 19 + 4.79 = 23.79. The static evaluation is computed for all other plays and the 23 plays with highest static evaluation comprise the output list. The database of leave values distributes high values to leaves with good chance of allowing "bingos", or plays that use all 7 letters and score a 50-point bonus, in coming turns. The database also favors leaves that garnered high scores when they appeared on a player's rack in a large sample of Quackle versus Quackle games.

Simulation

This block takes a list of moves as input (let's call this the candidate list) and slowly outputs, for each candidate play, hypothetical future positions that are reached after the following sequence of events:

1. The current player puts the candidate play (let's call it ply_0) on the board;

2. The opponent draws 7 random tiles, performs a static analysis of this new position after ply_0, and puts the play (ply_1) with the highest static evaluation on the board;

3. The current player replenishes his rack with random tiles, performs a static analysis of this new position after ply_0 and ply_1, and puts the play with the highest static evaluation (ply_2) on the board.

4. The current player adds to his score the value of his rack leave after ply_2.

The end result of this process is that the current player's score is equal to his score in the original position plus the scores of ply_0 (the candidate), ply_2, and the value of his leave after making ply_2. The opponent's score is equal to his score in the original position plus the score of ply_1. Let's look at an example. Let's assume that the current player is ahead by 20 points and holds ACEENOR. Now let's say we're looking at the candidate OCEAN and iteration 1 goes:

1. Current player plays OCEAN for 19 points leaving ER

2. Opponent randomly draws AIIKNST and plays TANKINIS for 167.

3. Current player randomly draws PQXYZ to his ER so holds EPQRXYZ. He plays PR(O)XY for 50 leaving EQZ.

After this sequence, the current player is down by 86 points (20 + 19 - 167 + 50 = -78). The value of his EQZ leave is -4.12, so the final outcome of this iteration for OCEAN is a point differential of -82.12. This process is repeated many times to get many hypothetical positions for each candidate play. Quackle performs about 300 iterations of this process for each candidate move. Simulation is similar to the look-ahead algorithms used in chess artificial intelligence, but modified to handle the randomness inherent in the outcome of a Scrabble game. Instead of performing one look-ahead very many moves ahead, we perform many looks ahead of only 2 moves into the future. [KatzO'Laughlin06]

Win Percentage Estimation

This block takes many future positions for each candidate and very quickly outputs a list of plays ordered by estimated win percentage. For each future position, Quackle does a simple table lookup into a precomputed database that guesses the current player's chance of winning the game based on how many points he is ahead or behind and how many tiles are left to be played in the game (the sum of the 7 tiles on the opponent's rack and the number of tiles left in the bag). For each candidate, these win percentage estimates are averaged over all the future positions that started out with that candidate. [KatzO'Laughlin06]

Here is a sample of the win percentage estimation database:

Point differential	Number of tiles left	Estimated win probability
-82	65	0.203748
-82	66	0.20625
-82	67	0.208717
...		
9	15	0.784516
9	16	0.779248
9	17	0.774146

Let's say we're trying to guess the win percentage of the OCEAN play from before after our first iteration of simulation. Recall the point differential after iteration 1 was -82.12. Let's assume that after the last play of that iteration, PROXY, there were 66 tiles remaining in the game. Then we'll estimate that if the resulting game from OCEAN's iteration 1 were played to completion, the current player would win about 20.6% of the time.

The database of estimated win probabilities was made by analyzing the distribution of wins over many Quackle versus Quackle games.

This kind of win-percentage-based analysis is critical in a Scrabble AI when needing to erase a large deficit or protect a lead. For instance, when down by 70 points late in the game, a bingo is usually necessary to catch up. If the current player ever bingos in an iteration of a candidate's simulation the estimated win percentage of that candidate is boosted mightily, while most other iterations will have an estimated win percentage of about zero.

After averaging the estimated win percentages of all iterations for a candidate, the plays that have the greatest chance of a future bingo, will come out on top as desired.

The above module of Quackle is called Bogowin. Basically, the makers of Quackle decided that it would calculate values for your odds of winning the game on average if there are T tiles in the bag and you are ahead by G points. So they ran a gigantic simulation on many possible boards, and developed a large database consisting of every spread up to +200 or -200 points and your odds of winning with a given spread depending on how many tiles were remaining in the bag. Bogowin runs every single Spread at the end of the simulation into Bogowin, and it returns a win% according to its database. These win% numbers are then averaged.

While this seems like a good idea and it certainly is an approximation, it is a tricky approximation. Being up to 80 points with 50 tiles in the bag on a very closed board, is better than being up 100 points on an open board with 40 tiles left in the bag. Bogowin becomes more reliable as the preendgame approaches. Before the preendgame humans tend to be much better judges of win% than computers. [Matsumoto14].

Endgame

Quackle uses the B* search Algorithm to play the endgame. This endgame engine is much stronger than humans, and most of time plays perfect endgames.

Computers are far stronger and faster than humans, specially at endgames, due to its remarkable raw processing speed and its ability to go through each possible draw extremely quickly. Besides, the B* Algorithm suits very well for Scrabble endgames. Quackle solves around 98% of endgames correctly. [Matsumoto14]

Quackle Match Results

Quackle has participated in two machine-vs-man tournaments. The first one was the 2006 Toronto Scrabble Open and Human vs Computer Showdown, this tournament involved two Scrabble engines: Maven and Quackle and 92 humans. The humans played an 18 round tournament; everyone had two byes, during which they had the opportunity to play unrated games against Maven and Quackle. Thus, Maven and Quackle each played 36 games against human opponents (they did not play against each other).

The human player with the best record in the Open, won the right to challenge the best computer player in a best of five match, and $3,500. Dave Boys, the former Scrabble World Champion (in 1995), won the Open with a record of 16-2. Quackle finished with a record of 32-4, surpassing Maven´s record of 30-6.

Dave Boys, a computer programmer, won the first two games against Quackle, before Quackle evened the record at 2-2 setting up an exciting final game. Quackle's win in the fifth game featured a subtle strategic move worthy of the best human expert.

The second Human vs Computer event was the 2011 Toronto International Scrabble Open and Human vs. Computer Showdown. This tournament featured 21 players from 6 countries, including experts Nathan Benedict, John O'Laughlin (one of Quackle's authors), Geoff Thevenot, Adam Logan (Scrabble World Champion in 2005) and Nigel Richards (Scrabble World Champion in 2007, 2011, 2013 and 2018) . Quackle was the only computer program to participate.

Quackle won the final match against Adam Logan on a 28-point differential ! Quackle won the first two games 535-362, 453-390; Adam rallied to win the last two 500-323, 437-406, but it was not enough to win the match.

Tony Leah, a Scrabble expert and certified tournament director from Toronto, added: "The fact that both playoffs between Quackle and a human were close and suspenseful, and were decided only by the last plays of the last game shows how incredibly strong the best human players are. I think that humans so far are more competitive in Scrabble vs computer players than in chess or, now, Go."

It is generally recognised that there might only be a handful of players who could thwart the best artificial intelligence available in Scrabble over a sufficiently long series. Quackle - developed by John O'Laughlin and Jason Katz-Brown - is widely regarded at its championship level as the best computer program.[Carter16]

But few who know their strategy in the community of the world's most popular board game would doubt that a player of, say, Nigel Richards ability, would have a problem beating Quackle in a reasonably long series. The New Zealand maestro and three time world champion even commented in an unguarded comment some years back that he would not have difficulty beating Quackle, and Nigel is well known for his modesty [Carter16]

4.3 Monte Carlo simulation

Simulation goes by the names “rollout”, “Monte Carlo search”, and the generic name “stochastic lookahead” (Frank *et al.* 1998). The technique has been applied to the games backgammon (Galperin and Viola, 1998), bridge (Ginsberg, 1999), and poker (Billings et al., 1999), and in the fields of operations research and optimal control.

Monte Carlo simulations are those which involve stochastic sampling to give good approximations of difficult to measure quantities. For example, to compute stochastically the number pi, one would generate N random points within the unit box and calculate the ratio of how many of these points fell within the unit circle. Benefits of these methods to parallel computing are their easy parallelizability and typically linear scalability [Wickman08].

Simulations were used as an investigative tool for almost a decade before Maven applied them to selecting moves over the board. Maven [Sheppard2002a] was not the first to do this, but probably was the first to have a thorough implementation that hit enough fine points to clearly surpass what can be achieved without simulations [Sheppard2002b].

To adapt Monte Carlo methods to Scrabble, we literally have to do some guessing. Given a particular board and rack, we may guess with varying degrees of success what our opponent's rack is. The more tiles on the board, the better chance of guessing their tiles. Suppose we make 1000 guesses for what rack our opponent has. Now, take the top ten moves in consideration (most likely those with the highest score). For each of these moves, generate the opponent's highest-scoring response based upon what we guess their rack to be. This involves calling the move generator 10,000 times, but this should only take on the order of half a minute to do [Wickman08].

Now, for each of our ten best moves, we have the average score of our opponent's response. Using this average, we may choose which of those ten fares well both offensively and defensively [Wickman08].

4.4 Opponent modeling

Mark Richards and Eyal Amir, [RichardsAmir07], incorporated information about the opponent's tiles into the decision-making process. They quantify the value of knowing what letters the opponent has. Using observations from previous plays to predict what tiles the opponent may hold and then use this information to guide the play. Their model of the opponent is based on Bayes' theorem from Probability Theory.

Although they improved the performance of the *Quackle* engine by adding an inference agent with this approach, it seems that the inference player is too slow for tournament play.

The inference agent played 630 games against *Quackle's Strong Player*, the inference player won 51.43% of the games and had a mean score of 427 pts. per game. Five more points than *Quackle's Strong Player*.

4.5 The Endgame

4.5.1 Endgame Characteristics

The endgame refers to play when the bag is empty; therefore it becomes a game of perfect information. Knowing the initial distribution of tiles lets you deduce the opponent's tiles in the rack, by looking at the board and the tiles in your rack.

When the last tile is drawn out of the bag the endgame begins. The side to move will hold 7 tiles, and the opponent will hold at most 7 tiles. Neither side will be able to obtain more tiles for the rest of the game, since the bag is empty. The game will end when one player has used all his tiles, or when both players pass consecutively. If a player has tiles left when the game is over, then his score is reduced by the sum of the face values of those tiles. If a player plays out, then his score is increased by the sum of the face values of the opponent's tiles.
The branching factor of a typical endgame position is rather large: about 200 for a 7-tile versus 7-tile endgame. But there is a great deal of variations if blanks (we used the symbol "#" to represent a blank) are in play, then the branching factor can explode over 1000.

Endgame depths are never too large. Because the game ends on consecutive passes, and because each player starts with at most 7 tiles, the variations must stop by ply 14. In practice however, the search depth along the principal continuation are often just 3 ply, as the first player to move can usually play all of his tiles ("go out") in 2 turns.

Good endgame play in Scrabble is aided by two helpful properties. The first one is that Scrabble is a *converging game*. That is, the complexity declines along every variation as tiles are placed onto the board. Thus, it is possible that an analysis made at the root of the search tree will turn up information that is valid (or at least

useful) throughout the search. The second property is that the board is largely fixed during the endgame. An endgame adds at most 13 tiles to the board where there are already at least 86 tiles. That the board is largely frozen also supports the notion that a static analysis of the root will be useful in the search. Additionally, while every endgame has long variations, ideal play is usually characterized by *early termination* when a move plays out. Because early termination is normally ideal, the really deep variations are often cut off [Sheppard2003].

4.5.2 Human endgame strategies

First Principle: Go Out In One

Maybe it is obvious, but the priority in the endgame is almost always to use up all your tiles in one move, thus ending the game. (This principle might fail if your rival is stuck with a tile).

What are the benefits of this? Clearly, your opponent will not increase his score, and you will have the additional benefit of the face value of his rack being added to your own tally. If the main aim is to deny your opponent another play, then you do not have to concentrate on scoring particularly high or blocking off hotspots.

Sometimes it can be worthwhile not to go out in one. If your rival is stuck with a tile, it is likely that you can do better by going out slowly, and make several moves, since your rival would have to pass. Or maybe you can stick your opponent with an unplayable tile. Remember that the fewer moves your opponent has, the more likely you are to benefit.

Second Principle: Stop Your Opponent From Going Out In One

Bearing in mind that your opponent will also be doing his utmost to go out as quickly as possible, you should do your best to stop him. Perhaps there is just one place in which he can play a tricky tile, so the sensible thing to do is to play there yourself instead. Or maybe your opponent has a bingo, which would be playable but for your timely intervention. If you can prevent that, the effect on the final score could be crucial.

Stopping your opponent from going out in one will guarantee you an extra go, which is bound to increase your score and improve the spread position. You also ensure that he does not benefit form *countback,* adding the face value of your own tiles to his score and deducting it from yours.

Third Principle: Go Out In Two

If you can't go out in one move, the next best course might be to go out in two moves. For the reasons mentioned previously, you will be frustrating your opponent's plans and minimizing his scoring potential the faster you go out.

If he has a high-scoring tile, it should be worthwhile to focus on his best potential move –for example, a 50-point play involving the letter **X** and available in only one spot would almost certainly require blocking [FisherWebb04] .

4.5.3 The B* Search Algorithm

The strong Scrabble engines Maven and Quackle decided to use the B* Search Algorithm [Berliner79] in the endgame play. Although most of the endgames can be solved with 3 plies and they could be solved using the Alpha-Beta Pruning Algorithm, there are some endgames that require 14 moves to play out, and it becomes harder to search that deeply with limited tournament time.

The B* search algorithm has the distinctive feature that the evaluation of nodes are real intervals. The algorithm assumes that the real interval necessarily contains the true value of the node, but B* is fairly robust to occasional violations of this condition, as seen in endgames played by Maven and Quackle.

The lower bound of an evaluation is called the Pessimistic score, and the upper bound is called the Optimistic score. Another peculiarity of B* is that the optimistic and pessimistic bounds need not come from the same child. One of the strengths of the algorithm is that one child can provide a secure pessimistic bound, while another child provides upside.

In the B* algorithm *separation* is achieved when the pessimistic score of one move is at least as large as the optimistic score of all other moves. Under the assumption that the true value of every node is always contained within the real interval, the separation condition amounts to a proof that one move is at least as good as any other.

There is one last decision to make in a B* algorithm: which node should be expanded? B* provides two strategies, called Prove-Best and Disprove-Rest, for making this decision. The goal of Prove-Best is to raise the pessimistic bound of the move that has the highest optimistic bound. If the pessimistic bound of that move raises enough, then separation will be achieved. The goal of Disprove-Rest is to lower the optimistic bound of the move having the second-highest optimistic bound. There is no point in working on any other moves, since it is impossible to achieve separation until progress is made on at least one of these two goals.

If the strategy is Prove-Best, then the frontier node to expand is found by following the sequence of nodes that establish optimistic bounds for the root node with the highest optimistic bound until a frontier node is found. If the strategy selection is Disprove-Rest, then the node is found by tracing the second-highest optimistic bound. It follows that one of at most two nodes is the node to expand [Sheppard03].

4.5.4 An Endgame Example

The endgame, the terminal part of a Scrabble game, is the phase which begins when the bag of tiles is emptied. One of the players has 7 tiles in his rack and the other one at most 7. If no phony words have been played, the player to move, which we will call player I or White has 7 tiles while the other one, player II or Black, has 7 tiles or less.

Positive (resp. negative) numbers indicate an advantage for White (resp. Black).

In 75% of the games the side with the advantage has a score differential of more than 60 points [Thomas11], so even a modest strategy such as depth 2 (that is, the assumption that the game will last no more than 2 plies) is satisfactory. We make this assertion under the assumption that there are no stuck (that is, unplayable) tiles; if there are, then "one-tiling" (i.e. playing one or two tiles per turn) is usually preferable.

Let us now describe a possible strategy.

For $n = 1, 2, 3, \dots$ we find the best play assuming the endgame will last exactly n plies. For $n = 1$, this simply means finding the best bingo if there are any. For n odd (resp. even) we are considering games in which I (resp. II) goes out, i.e. it is I (resp. II) who plays his last tiles.

Denote by $|\mu|$ the score of half-move μ (positive if it is a white half-move and negative if it is black). Also denote by $||\mu||$ the *modified value* of μ, defined as $|\mu|$ plus (resp. minus) twice the total value of its residue if μ is a black (resp. white) half-move.

For n = 2 the plan amounts to maximizing $||\mu_1||+|\mu_2|$ where μ_1 is a half-move by White that allows Black a half-move μ_2 that plays all his tiles.

For n = 3 the plan is maximizing $|\mu_1|+||\mu_2||+|\mu_3|$ where μ_1 and μ_3 are White's half-moves that are compatible with Black's half-move μ_2.

It should be mentioned that when one is considering plans with depth exactly *n*, it is not necessary to consider choices which lead to a value not greater than the best value obtained with plans *m* for *m*<*n*.

We now analyze a tournament endgame. The position when the endgame starts is shown on Fig. 4.5 and the racks of I and II are respectively

rack I: **aabolsu** and rack II: **dghst** .

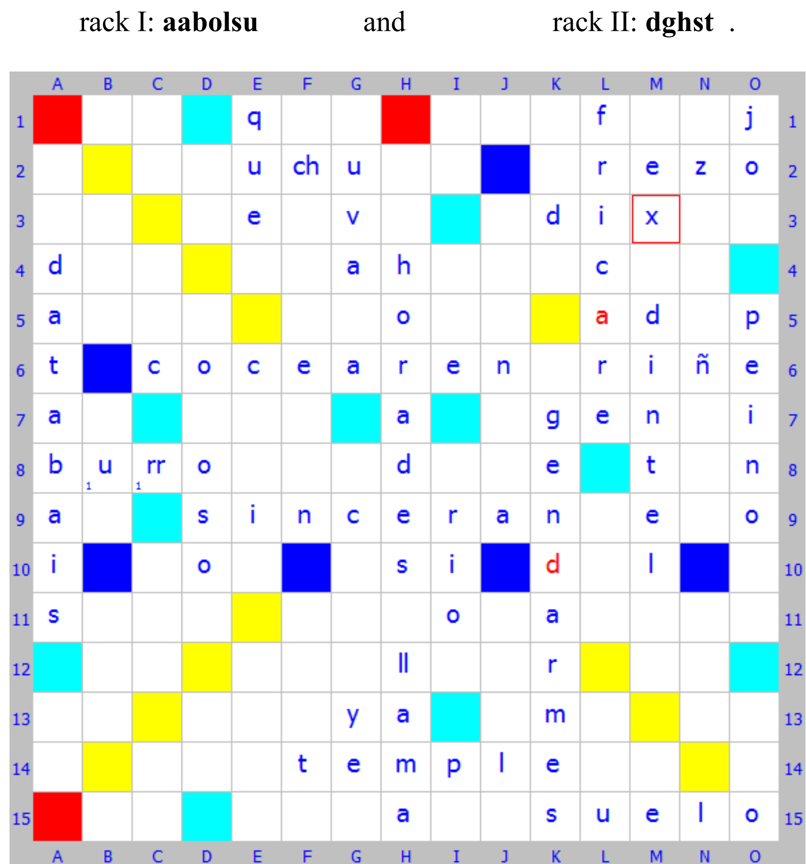


Fig. 4.5 An Example of an Endgame, White's rack aabolsu, Black's rack dghst. White to move

It is impossible for II to go out in less than 3 moves so it is sufficient to consider n-ply endings with n=3 and n>4.

We follow chess notation. The first row provides the move numbers with subsequent rows representing different variations. White half-moves are shown above black-half moves. Dots (**…**) represent half-moves that, for the variation, are identical to the variation above. A question mark ? indicates a mistake.
The final half-move of a variation, by White, in our example, is followed by {residue} where residue is the set of tiles black is left with; {residue} is followed by *m*+*p* where *m* is the number of points of White's final half move and *p* is twice the number of points of residue.

Table 4.1. 3-ply endings

	1	2	3	
1.	E6 (c)us(i)a 23 7G s(a)h* -23	13J a(m)blo{dgt} 20+10		**30**
2.	**…** 23 14F (temple)s ? -13	12K (r)obla{dght}16+18		**44**
3.	13J a(m)blo 20 7G s(a)h -23	E6 (c)us(i)a{dgt}23+10		**30**
4.	**…** 20 F6 (e)t ? -2	N8 usa{dghs}15+18		**51**
5.	8H (d)a 7 7G h(a) -18	15A sabulo{dgst} 29+12		**30**
6.	8G o(d)a 12 5G d(o)s -15	B1 bulas{ght}19+14		**30**
7.	M5 (dintel)aba 24 7H (a)h -18	B1 luso{dgst} 11+12		**29**
8.	**…** 24 7G s(a)h ? -23	E6 (c)us(i) 20 G2 (uva)d(as) -13	L11 lo{gt} 17+6	**31**
9.	**…** 24 8J t(e)s(t) ? -20	E6 (c)us(i) 20 7H (a)h -18	L11 lo{gt} 17+8	**31**
10.	13J a(m)obla 22 7G s(a)h -23	E6 (c)us(i){dgt} 20+10		**29**
11.	13J a(m)bulo 22 7G s(a)h -23	E6 (c)as(i){dgt} 20+10		**29**
12.	13J a(m)bula 22 7G s(a)h -23	E6 (c)os(i){dgt} 20+10		**29**
13.	**…** 22 **…** -23	E1 (que)o 8 ? G2 (uva)d(as) -13	1K u(f){gt} 5+6	**5**
14.	E6 (c)us(i) 20 7G s(a)h -23	13J a(m)obla{dgt} 22+10		**29**
15.	**…** 20 14F (temple)s ? -13	12J a(r)bola{dgt} 18+18		**43**
16.	E6 (c)os(i) 20 7G s(a)h -23	13J a(m)bula{dgt} 22+10		**29**
17.	**…** 20 14F (temple)s ? -13	15B baula{dght} 10+18		**35**
18.	E6 (c)as(i) 20 7G s(a)h -23	13J a(m)bulo{dgt} 22+10		**29**
19.	**…** 20 M14 g(e) ? -3	8G o(d)a 12 5G d(o)s -15	D3 bul(o){ht} 16+10	**40**
20.	E6 (c)as(i)a 23 7G s(a)h -23	C10 bulo{dgt} 17+10		**27**
21.	**…** 23 12L (a)h ? -5	I1 bulo{dgst} 13+12		**43**
22.	15A sabulo 29 7G s(a)h -23	13F a(ya){dgt} 9+10		**25**
23.	E6 (c)os(i)a 23 7G s(a)h -23	C10 bula{dgt} 17+10		**27**
24.	**…** 23	15C bula{dgst} 9+12		**39**

	5B (a)h ? -5			
25.	13J a(m)bla 20 7G s(a)h -23	N8 uso{dgt} 15+10		**22**
26.	15A abolsa** 29 7G s(a)h -23	1K u(f) {dgt} 5+10		**21**
27.	11A (s)ubsola 22 7G s(a)h -23	D8 (osos)a {dgt} 10+10		**19**
28.	13J a(m)bos 20 7G s(a)h -23	4A (d)ual {dgt} 10+10		**17**
29.	**…** 20 5A (a)d ? -3	F9 (n)ula{ghst} 6+16		**39**
30.	13J a(m)blas 22 7G s(a)h -23	F8 u(n)o{dgt} 5+10		**14**
31.	**…** 22 F6 (e)h ? -5	4A duo{dgst} 4+12		**33**
32.	13J a(m)bas 20 7G s(a)h -23	F9 (n)ulo{dgt} 6+10		**13**
33.	**…** 20 14F (temple)s ? -13	12K (r)ulo{dght} 8+18		**33**

*“s(a)h” is equivalent to “h(a)s”.
**”abolsa” is equivalent to “losaba”, “sabalo” and “solaba” .
A few 5 plies variants appear on the table above. These are bad variants that need 5 plies to be refuted.

Table 4.2. n-ply endings for n > 4

	1	2	3	
1.	E6 (c)us(i) 20 7G s(a)h -23	M5 (dintel)aba 24 G2 (uva)d(as) -13	L11 lo{gt} 17+6	**31**
2.	**…** 20 7H (a)h ? -18	13J a(m)obla{dgst} 22+12		**36**

Comments. The best 3-ply plans are variations 1,3, 5 and 6 in Table 4.1, which yield 30 points.

The best plan is a 5-ply plan which is variation 1 in Table 4.2. It yields 31 points.

These variations were found by analyzing those half-moves of White whose score is at least two thirds of the maximal one (in our example 29) and those which block the best half-move of Black; the rest of the half-moves gave poorer final scores for White. In this task it was found convenient to list the *independent sequences* of I (White) and II (Black). Here a *sequence* of I is an n-tuple $(w_1, w_2, \ldots, w_n)$, with $n \leq 3$, such that w_i is a valid initial half-move of I , occupying the subset S_i of the board; we call it *independent* if $d(S_i,S_j)>1$ for $1\leq i<j\leq n$.

Chapter 5.

The Beginning

This chapter deals about the construction of a Spanish lexicon. To build and engine that plays Scrabble, in any language, we need to know all the valid words for that language, in other words we need to have a Lexicon. In order to build an engine that played Scrabble in Spanish using the FISE tournament rules which use the DRAE "Diccionario de la Real Academia Española", we had to build from zero a Spanish lexicon, since there was no Spanish lexicon at that time, based in the DRAE dictionary that used the official tournament FISE rules. This was a big challenge, since it required a thoughtful development of morphological rules to build the verbal inflections of all verbs, and to make the plurals of many words (according to the FISE rules). We had to type all 88455 lemmas (entries) of the DRAE, to begin, and from them build the entire lexicon; the total number of valid Scrabble words in Spanish reached around 637000 words. Besides playing Scrabble in Spanish, the construction of the Lexicon could be used to learn the Spanish languague, and could possibly have other uses. This was the beginning of the construction of our *Heuri* Scrabble engine.

5.1 The lexicon

An important part of a program that plays Scrabble, in Spanish for our case, is the construction of a lexicon. The length of it is significantly larger than the length of a corresponding lexicon in English, being Spanish, unlike English, a romance language. A regular transitive verb in Spanish, as "amar", "temer" or "partir" has 46, 55 or 54 verbal inflections depending on whether it is an ar-verb, an er-verb or an ir-verb, whereas a verb in English, like "love", typically has only four verbal inflections (love, loves, loved, loving). Even though short words are more numerous in English than in Spanish -1350 words of length less than 4 in SOWPODS, the UK Scrabble dictionary, and 609 in Spanish- the number of words of length less than 9 (the important lengths are 8 and 7) is roughly 107000 in SOWPODS and 177000 in our Spanish dictionary. Accordingly, more bingos (seven tiles plays) are expected in a Spanish game than in an English game. The total number of words (of length less than 16) is, approximately, 246000 in SOWPODS and 637000 in our Spanish lexicon.

We now extract some important ideas from [Sheppard02a]:

"It is vitally important to know all of the words in order to avoid errors".

"We extensively discuss vocabulary because massive knowledge bases are an important component of AI systems".

"Computerizing the words was a big job. The OSPD (the American Scrabble dictionary) contained about 50000 main entries, and with inflected forms totalled about 95000 words averaging 8 letters long. At the authors´ typing speed the data entry task would have taken about 4 months of full time effort". We add: typing 637000 words would have taken more than two years of full time effort.

Lexicon Construction

The electronic version of DRAE (*Diccionario de la Real Academia Española XXII Ed.*), the official Scrabble dictionary in Spanish has 88455 lemmas (entries) from which one has to construct the complete lexicon. We therefore have the following two subproblems:

I. Conjugation of verbs.

II. Plurals of nouns and adjectives. (Note: Plurals of articles are explicit in DRAE and pronouns which admit plurals are few (roughly 20). Adverbs, prepositions, conjunctions, interjections, onomatopoeias, cannot be pluralized). Words in phrases cannot be pluralized either.

The FISE (*Federación Internacional de Scrabble en Español*) rules are clear as to which inflections of verbs are valid since old verbs cannot be conjugated (old words are those which in all its meanings have one of the abbreviations ant., desus. or germ.) and all verbs which are not old are explicitly conjugated in DRAE. Unfortunately all these conjugations cannot be copied in a text file as they are encrypted in a special format. Therefore we had to develop a way to conjugate verbs. The FISE rules indicating which nouns and adjectives admit a plural are less clear.

From the electronic version of DRAE, which we will refer to as eDRAE, one eliminates the words which contain “k”, “w”, a capital letter or a hyphen (-).

We define the *length* of a word as the number of letters of it, counting “ch”, “ll” and “rr” as one letter. Thus the lengths of “christmas”, “chorrillo”, “güisquis”, “guizguen”, “ñaque” and “hule" are, respectively, 8, 6, 8, 8, 5 and 4.
One eliminates all words of length greater than 15 with the exception of the 4 verbs of length 16 ending in “ar” and the 4 verbs of length greater than 15 ending in “arse”.

Words which in all their meanings have one of the abbreviations ant., desus. or germ. will be called *old* and all other words will be called *current*.

Old words can be used only as written in eDRAE, that is, they cannot be pluralized if they are nouns or adjectives in singular and they can be used only in infinitive if they are verbs.
Also words in italics (coming from different languages) can only be used as written; they cannot be pluralized.

I. Conjugation of verbs

The only inflections of verbs which are allowed are those indicated by eDRAE by a square mark.
Usually the root of a verb is defined as the infinitive form after the ending ar, er, ir (arse, erse, irse in the case of pronominal infinitives) is deleted. This, however, we call the *main* root because we will also consider *artificial* roots when dealing with irregular verbs or verbs which need orthographic modifications in their roots.
The verbs ending in “ar” (resp. “er”, “ir”) and “arse” (resp. “erse”, “irse”) are also called verbs of the first (resp. second, third) conjugation.

Regular verbs

The inflections of a current regular transitive verb needing no orthographic modifications are obtained by juxtaposing to the main root the following endings (we omit accents):

ar-verbs or arse verbs: o, as, a, amos, ais, an, aste,asteis, aron, aba, abas, abamos, abais, aban, aria, arias, ariamos, ariais, arian, ara, aras, aramos, arais aran, are, ares, aremos, areis, aren, ase, ases, asemos, aseis, asen, ar, ando, ado, e, es, emos, eis, en, ad, ada, ados, adas.

er-verbs or erse-verbs: es, e, emos, eis, en, i, iste, io, imos isteis, ieron, ere, eras, era, eremos, ereis, eran, ia, ias, iamos, iais, ian, eria, erias, eriamos, eriais, erian, iera,ieras, ieramos, ierais, ieran, iere, ieres, ieremos, iereis, ieren, iese, ieses, iesemos,ieseis, iesen, er, iendo, ido, o, a, as, amos, ais, an, ed, ida, idos, idas.

ir-verbs or irse-verbs: es, e, imos, is, en, i, iste, io, isteis, ieron, ire, iras, ira, iremos, ireis, iran, ia, ias, iamos, iais, ian, iria, irias, iriamos, iriais, irian, iera, ieras, ieramos, ierais, ieran, iere, ieres, ieremos, iereis, ieren, iese, ieses, iesemos, ieseis, iesen, ir, iendo, ido, o, a, as, amos, ais, an, id, ida, idos, idas.

According to FISE (*Federación Internacional de Scrabble en Español*) the following endings have to be eliminated for verbs which are not transitive:

a) ada, ados, adas, ida, idos, idas, for intransitive verbs.

b) ad, ed, id and the endings in a) for pronominal verbs which are not intransitive. For these verbs, however, the endings arse (resp. erse, irse) are allowed for verbs of the first (resp. second, third) conjugation provided this pronominal form appears as an entry in eDRAE.

We now explain the conjugation of regular verbs which need orthographic modifications in their roots.

Verbs ending in “car” (resp. “gar”, “zar”). They are conjugated as the regular ar-verbs above with the indication that when the ending “e”, “es”, “emos”, “eis”, or “en” is used the last “c” (resp. “g”, “z”) of the main root has to be replaced by “qu” (resp. “gu”, “c”).

Verbs ending in “ger” (resp. “gir”). They are conjugated as the regular er-verbs (resp. ir-verbs) above with the indication that when the ending “o”, “as”, “amos”, “ais”, or “an” is used the last “g” of the main root has to be replaced by “j”.

Verbs ending in “guir” (resp. “quir”). They are conjugated as the regular ir-verbs above with the indication that when the ending “o”, “as”, “amos”, “ais”, or “an” is used the final “gu” (resp. “qu”) in the main root has to be replaced by “g” (resp. “c”).

Verbs ending in “cer” (resp. “cir”). They are conjugated as the regular er-verbs (resp. ir-verbs) above with the indication that when the ending “o”, “as”, “amos”, “ais”, or “an” is used the last “c” of the main root has to be replaced by “z”.

Thus, for regular verbs needing orthographic modifications, it is convenient to use one artificial root besides the main root. For example, use the roots “saqu”, “pagu”, “cac” for the endings “e”, “eis”, “emos”, “eis”, “en” and the roots “sac”, “pag”, “caz” for the remaining ar-endings. Similarly, use “protej”, “dirij”, “disting”, “delinc”, “venz”, “zurz” for the endings “o”, “a”, “amos”, “ais”, “an” and “proteg”, “venc”, “dirig”, “distingu”, “delinqu”, “zurc” for the other endings (er-endings for the first two, ir-endings for the last four).

Irregular verbs

For the purposes of Scrabble we group the irregular verbs in 38 families. The last family consists of the verbs caber, caler, desosar, erguir, errar, estar, haber, ir, jugar, oler, poder and ser; no verb in this family is conjugated like any other verb in Spanish.

We give a model for each of the remaining 37 families. Each verb in a family is conjugated like the model belonging to it. In the list that follows we give the irregular inflections of each model and also the additional inflections which come from "voseo" (Spanish of some regions including Argentina). We use the following abbreviations of collections of endings;

E1={o, a, as, amos, ais, an}
E2={a, e, o, as, es, an, en}
E3={e, es, en}
E4={ieron, iera, ieras, ieramos, ierais, ieran, iere, ieres, ieremos, iereis, ieren, iese, ieses, iesemos, ieseis, iesen}
E4'={E4, iendo,io,amos, ais}
E4''={E4, e, o, iste, imos, isteis}
E5={eron,era,eras,eramos,erais, eran, ere, eres, eremos, ereis, eren, ese, eses, esemos, eseis, esen}
E5'={E5, endo, o}
E5''={E5, e, o, iste, imos, isteis}
E6={a, as, an, ia, ias, iamos, iais, ian}
E6'={E6, e, emos, eis}
E6''={E6, o, amos, ais}

We now give the 37 models, their irregular inflections, the inflections coming from voseo and the number of verbs in each family. See Table 5.1.

TABLE 5.1. Irregular verbs familiy models

Irregular verb model	Irregular inflections	Voseo inflections	Number of verbs
1. agradecer	agradezc(E1)		260
2. acertar	aciert(E2)	acert(a, as)	157
3. contar	cuent(E2)	cont(a, as)	126
4. construir	construy(E1, E3, E5')		55
5. sentir	sient(E2), sint(E4')		46
6. pedir	pid(E1, E3, E4´)		43
7. mover	muev(E2)	mov(e, es)	33
8. entender	entiend(E2)	entend(e, es)	30
9. mullir	mull(E5')		30
10. poner	pong(E1), pus(E4''), pondr(E6'), pon, p(uesto)		27
11. venir	veng(E1),vin(E4'', iendo), ven	ven(i)	18
12. conducir	conduzc(E1), conduj(E5'')		15
13. traer	traig(E1), traj(E5''), tray(endo)		13
14. ceñir	ciñ(E1,E3,E5')		12
15. tener	teng(E1), tien(E3), tuv(E4''), ten	ten(e,es)	11
16. leer	ley(E5')		10

Irregular verb model	Irregular inflections	Voseo inflections	Number of verbs
17. decir	dig(E1), dic(E3,iendo), dij(E5''), dir(E6'), d(icho)	dec(i)	8
18. lucir	luzc(E1)		8
19. sonreir	sonri(E1,E3,E5')		7
20. hacer	hag(E1), hiz(o), hic(E4''-{o}), har(E6'), h(az,echo)		6
21. ver	ve(E6''), v(isto)		6
22. caer	caig(E1), cay(E5')		5
23. dormir	duerm(E2), durm(E4')		5
24. querer	quier(E1), quis(E4''), querr(E6´)	quer(e,es)	4
25. oir	oig(E1), oy(E3, E5')		4
26. discernir	disciern(E2)		4
27. salir	salg(E1), saldr(E6´), sal		3
28. valer	valg(E1), valdr(E6')		3
29. yacer	yazg(E1), yazc(E1), yag(o), yaz		2
30. saber	s(e), sep(E1-{o}), sup(E4''), sabr(E6')		2
31. roer	roig(E1), roy(E1, E5')		2
32. dar	d(E4''-{o}, oy, i)		2
33. asir	asg(E1)		2
34. adquirir	adquier(E2)		2
35. andar	anduv(E4'')		2
36. tañer	tañ(E5')		2
37. bendecir	bendig(E1), bendic(E3, iendo), bendij(E5'')		2

Notes.

1. The verbs ending in “car”, “gar”, “zar”, “cer”, “gir”, “guir” appearing in the families “acertar”, “contar”, “pedir” and “mover” need the orthographic modifications explained when dealing with regular verbs.
2. The verb “roer” (and “corroer”) can also be conjugated as a regular verb in the present tenses, that is, the forms ro(E1) are valid.
3. The word “entredi”, from “entredecir”, is valid.

II. Plurals of nouns and adjectives

A difficult part of the construction of a lexicon is the determination of which words admit a plural inflection, among other factors because of the ambiguity of the FISE rules. Only words one of whose meanings is that of a noun, adjective or pronoun may be pluralized. The plurals of articles are all explicit as entries in eDRAE. The plurals of pronouns which need to be included are "aquesas", "aquesos", "aquestas", "aquestos", "aquellas", "cuales", "cuantas", "nuestras", "quienes", "quienesquiera", "suyas", "tuyas", "ustedes" and "vuestras". Adverbs, prepositions, conjunctions, interjections, onomatopoeias and contractions do not have plurals. If a word is not an entry but is contained in a phrase having the abbreviation expr. (*expresión*) or loc. *(locución*) it cannot be pluralized. However, an adjective contained in an "envío" having a little square and the abbreviation V. (*véase*) can be pluralized, for example "gamada".

An incomplete list of approximately 78000 words from DRAE appeared in INTERNET; there were about 10000 omitted words. A significant part of these consisted of words used in the American continent. In addition, words in masculine which appear in eDRAE together with a feminine inflection appeared in the list only in masculine. Thus, "niño, ña" of eDRAE appeared in the list as "niño". Those words of eDRAE had to be found and added to the list. Also, the words of the completed list were labelled (with an "f", "n" and other letters) depending on which inflections they had (feminine, masculine plural, feminine plural). This, of course required careful comparison with eDRAE and determination of which words were old.

Eventually a dictionary with inflections, consisting of approximately 636000 words, was obtained. More recently this dictionary was checked with the help of "*hypersnap*" which enabled one to write a textfile which contained all the entries of eDRAE.

The final "dictionary with inflections" (as of now) consists of 636561 words.

5.2 A Spanish Scrabble program

After a lot of work the huge task of building the lexicon was accomplished. After this we modified *Scrabbler*, a free English Scrabble source program made by Amitabh in 1999. The double letters CH, LL, RR were added as well as the letter Ñ. The distribution was also changed and finally some modifications had to be made in order to follow the Scrabble rules for tournaments in Spanish. These rules involved the capability of a player to change a certain number of letters or to pass.

Finally after all this work we had the first Spanish Scrabble program that played with the official rules of Spanish Scrabble tournaments (the FISE rules). Let us call this program *Scrabbler II*.

In order to check the performance of the new lexicon built and the functionality of the Scrabbler II program, a match was performed between Scrabbler II and an expert human Scrabble player who was ranked No. 28 in the Spanish Scrabble league and No. 90 in the world Spanish Scrabble rating list. The match followed the format of a baseball world-series match, that is, the player who arrives at 4 victories first wins the match. The human expert won the match 4 - 2.

Although at first sight one could think that the computer has a huge advantage against a human because it knows all the permissible words, it turns out that this is not sufficient to beat a Scrabble expert. The reason is that Scrabbler II employs a naïve greedy strategy. It always plays a move that maximizes the points at the current time. This is an optimal strategy for duplicated Scrabble where the aim of the game is to make the move that gives more points in every turn; nevertheless for normal Scrabble this strategy is not a good one, since it ignores

how this could affect the options in making better moves in future plays. Many times you will end up with low quality racks like {GGLLQPM} and since there is no change strategy these low quality racks will tend to prevail, thus giving the human lots of advantage. For instance, in Scrabble one usually tries to play a *bingo,* (playing all seven tiles in one turn), since this gives you 50 bonus points. Therefore it is convenient to preserve the blank tiles which are jokers that represent any chosen permissible letter. Since Scrabbler II plays always the highest point move, it tends to get rid of the blank tiles without making any bingo.

Since all these features were observed we decided to make an improved version of Scrabbler II with an inference strategy engine. This engine will follow a heuristic function that tries to play a high scoring move as well as balancing the rack in play.

5.3 About Heuri

Motivation

Even though Quackle is regarded as the best Scrabble engine, there is still room for improvement. There has been no official long series match between a human and a Scrabble engine, and there are reasons to believe that a handful of human players still dominate the game.

Let us read some words on this respect. Scrabble expert player and writer Kenji Matsumoto when comparing Quackle with humans stated: "in a long series of games against Quackle, Nigel Richards (5 times World Champion in English and twice World Champion in French) would be favorite," he also said: "Adam Logan (the Canadian mathematician and 2005 world champion) would also probably win and maybe one or two others."
Top international player David Eldar remarked that Nigel would be likely to win between 53% and 58% of games at Quackle's present skill level. "I am very curious to see a computer that could rival Nigel," said the Australian.

Top player Quinn James said it would be difficult for a computer to match the skillset of the top players' strategy. "I think there are probably clear ways in which Go is more complicated than Scrabble, but I still think it would be very tough to train a neural net, specifically, to play Scrabble effectively because of the way that imperfect information and the dictionary factor interact," he said.

It should be pointed out that in the very highest echelons of the game, at the level of Nigel and Adam for example, the vocabulary of the game would be virtually identical between man and machine. Occasionally the human might make a slip but in a long series it would make little tangible difference. The strategy is going to be the key. [Carter16]

Quackle is better than Maven. On page 46 in the book Knowledge-Free and Learning-Based Methods in Intelligent Game Playing, by Jacek Mandziuk, published on January , 2010; the following was written:
“ The crossword puzzle solving software named Quackle [166] outplayed Maven during the 2006 Human vs. Computer Showdown in Toronto. Both machines played against the same 36 human opponents achieving excellent scores: 32-4 and 30-6, respectively. Outplaying Maven gave Quackle the right to play a best-of-five match against the best human contestant David Boys. Boys – the former Scrabble World Champion (in 1995) - was outwitted by Quackle losing three out of five games”.

It is debatable whether a state-of-the-art computer program beats the best human. In [3] Howard Warner states: “Nigel Richards, the top player in the world, seems to have a computer for a brain. In many ways we think he could actually be superior to Quackle the program”.

“Editorial note: According to FiveThirty Eight's analysis of cross.tables.com's data, Richards is technically ranked higher than Quackle”.

There is only one head-to-head encounter between Nigel Richards and Quackle, it took place in the Toronto International Scrabble Open 2011, Nigel won 473-449 .

Heuri

Heuri is a human-like Scrabble engine, based on ideas used by humans when playing Scrabble. Starting from looking for legal moves on the board, Heuri uses "The Anagram Method" [GonzálezAlquézar18], this method resembles the way Scrabble players search for a legal move, by using Anagrams and spotting where to play on the board. After finding some legal moves, Expert Scrabble Players use a strategy to decide which move to play, by balancing the points made by a certain move, played on the board, and the potential points in the next move that the unplayed tiles , the *leave* (tile residue left on the rack), can make. Again Heuri's decision-making on what to play resembles this strategy. These ideas yielded the construction of the Computer Lexicon and Heuri's engine.

Chapter 6.

Move Generator

This chapter explains "the Anagram Method", which is a technique to generate all valid words in a given languague given a lexicon for that language; then this lexicon has to be inserted in several adequate files, in order to retrieve, later on, valid words. The method of construction of this *Computer Lexicon*, along with the way to retrieve the words and put them on the board, carefully searching for all possible spots in/on the entire board, has as a consequence the generation of all valid moves, and only those. The method is based on indexing all valid words in the lexicon by means of their anagrams, and then finding matches of words retrieved from anagrams formed by tiles in the rack and tiles in/on the board. An *anagram of a string w* is a valid word obtained by the rearrangement of the letters of *w*.

The algorithm used in the Anagram Method receives as an input a board and a rack containing usually seven tiles, sometimes less tiles (in many endgames). This two inputs must be converted into 2 related inputs. The board gives the first input, the number of *anchor squares*, that is the number of squares, on the board, where you can legally place a letter. The second input is given by the tiles on the rack, this multiset of tiles gives as an input, all the possible subsets of this multiset. The time complexity of our algorithm is linear since the running time increases at most linearly with the size of the input data, that is the number of anchor squares plus the number of subsets of the multiset given by the rack.

This method was an innovation in Scrabble move generators, before this, Scrabble move generators used a DAWG (Direct Acyclic Word Graph) or a GADDAG (similar to the DAWG, but with more paths) to index valid words in terms of their preffixes and suffixes. Several experiments indicate that the Anagram Method is faster than the past move generator methods that used DAWG or GADDAG.

6.1 Computer Lexicon

The lexicon used may contain special letters which do not have an ASCII value, in the case of Spanish the double letters ch, ll and rr. Therefore we replace those double letters with other characters. In our case, we decided to use k, w and % instead of ch, ll and rr. A name used for this type of lexicon transformed is the *raw lexicon*.

Once having a raw lexicon in a given language (Spanish, French and English are used in this thesis), the words contained in the lexicon are rearranged in different files. Some of these files will be used by Heuri's move generator in order to produce *candidate moves* (moves that can be played), and other files will be used to calculate the values of the moves.

6.1.1 Direct and Inverse files

Direct_n and Inverse_n files consist of lists having all valid words of length n ($2 \leq n \leq 15$), in direct and inverse order. Notice that there are no words of length 1 (see 2.2.4). Direct_n and Inverse_n files are used to check word placement more efficiently, they also serve to evaluate perpendicular moves.

Many times, when a word is played on the board, more than one word is formed. When more than one word is made, if more than one tile is played it is possible to distinguish between the principal or main word from the others. The *principal* or *main* word is the word formed using the tiles grabbed from your rack (*the complement*),

all the other words formed are perpendicular to the principal word. In Fig. 6.1 an example is shown. The rack is [doxtlcv] and the complement is {dx}.

Direct and Inverse files are used to check perpendicular words. In Fig. 6.1, where the main word is *d(i)x*, the perpendicular words *ad* and *ox* are contained in the Direct_2 file, these words in inverse order (*da* and *xo*) are contained in the Inverse_2 file. Part of the calculation of the score of a move involves finding these words in the mentioned files.

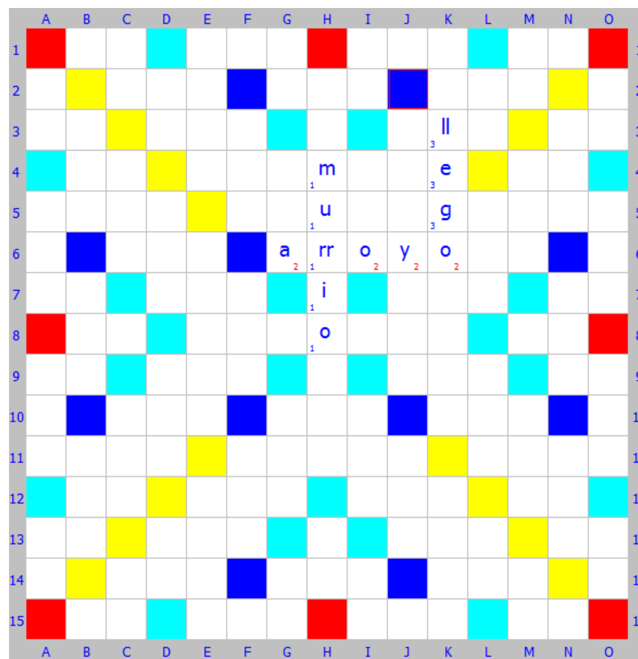


Fig. 6.1 Principal word 7G d(i)x , perpendiculars G6 (a)d , I6 (o)x

6.1.2 Index and Anagram files

Index files contain collections of strings of tiles that with an adequate permutation (or rearrangement) form at least one *valid word*, a word contained in the lexicon. Anagram files contain all the anagrams of a given string. An *anagram of a string w* is a valid word obtained by the rearrangement of the letters of *w*. Each member (a string) of an Index_n file is wisely linked to a member (set of anagrams) of an Anagram_n file (n stands for the length of the string or word). Here is an example: take the string "EOS" (member of the Index_3 file), this string points to a member of the Anagrams_3 file which is: {ESO, OES, OSE, SEO}. An important thing to remember is that, if a string of tiles does not have anagrams then it is not contained in any Index_n file. The Index and Anagram files are the soul of the move generator.

Let us give a more general view of this subsection.

Define Enlarged Direct_n, which we will write as Index_n, as the collection of strings obtained by replacing in each member of Direct_n 0, 1 or 2 letters by blanks. Each member of Index_n is written alphabetically, and Index_n is ordered lexicographically.

In this subsection we construct files which associate to each member w of Index_n the (nonempty) collections of words in Direct_n that can be constructed from the tiles of w. Anagrams_n is the set of all the words in Direct_n which can be constructed from the strings of Index_n .

The construction of the Index and Anagram files

In this subsection we construct files which associate to each member w of Index_n the (nonempty) collection of words in Direct_n that can be constructed from the tiles of w. Anagrams_n is the set of all the words in Direct_n which can be constructed from the strings of Index_n.

We strongly suggest looking at Table 6.1 to understand better the following descriptions.

Start with Direct_n, the words of the lexicon of length n; then concatenate each member of Direct_n with its associated string. The *associated string* of a word consists of the chain formed by the letters of the word; the letters are ordered alphabetically. The word concatenated with its associated string forms a member of the *Preliminar AIndex_n*; the collection of all these members form Preliminar AIndex_n.

To build *AIndex_n* we need to include blank tiles in every member of Preliminar AIndex_n, more specifically in every string associated to an anagram; but this time there will be no ordering yet of the associated strings. To achieve this task we proceed in the following way:

Suppose we already have a member without blank tiles, then to add other members to the file that contain exactly one blank tile, we make n copies of each member. Then for the ith copy of each member, the ith letter of the string part of each member, is replaced with a blank tile (#), i varies from 1 to n.

Now to add the members containing 2 blank tiles, for each member with no blank tiles, we make $n! / 2! (n-2)!$ copies of each member, then replace each string belonging to each member with a different possible placement without repetition of the subset of two blanks in the string of length n.

The collection of all the members containing zero, one and two blanks form the entire *AIndex_n.*

Due to the repetition of a certain letter in a string, when blanks are added several equivalent members appear. These members will become duplicated members once ordered in Ordered AIndex_n and duplications will disappear after being reduced.

Once having the AIndex_n files with all its members, the members are ordered by their associated strings components (this ordering occurs in Ordered AIndex_n) in a lexicographical manner (the blank tile "#" goes at the end of the alphabet). When ordered, duplicated members will appear one after the other. The ordering also causes that all members with an equal associated string component (string/ending substring of length n) are grouped one after the other; let us call this set of members a *class*. After the ordering of all members of AIndex_n we obtain *Ordered AIndex_n*.

Then duplicated members are eliminated and the classes are reduced by eliminating repeated strings, leaving only the string of the first member of the class, *the class representative*.
After these operations are performed, *Reduced AIndex_n* is formed.

Finally, the Reduced AIndex_n files are separated in two groups of files:

1) The Index_n files which consist of the associated string parts of the class representatives of the Reduced AIndex_n files.

2) The Anagrams_n files which consist of all the words from the Reduced AIndex_n files.

A very important pointer is added to each class representative string part (now a member of Index_n) so that it points to its past members of the class (which are words belonging to Anagrams_n).

Once all these files are constructed, an important function is developed that receives a subrack, and returns a key, a place in memory where the anagrams corresponding to the subrack start, and an integer (the number of anagrams found). Remember that if the subrack, of length *n,* does not match any string in the Index_n file, then this means that such a string has zero anagrams. Let us give a small pseudocode to denote such a function.

Function Find Anagrams

```
Find_Anagrams(Subrack)
Returns Key and NumberofAnagrams
```

The word NEUTRAL has the associated string AELNRTU, and the *AIndex_7* file contains the member NEUTRALAELNRTU; similarly the word BACALAO originates BACALAOAAABCLO $\in$*AIndex_7*. See Table 6.1.

An example is shown in Table 6.1. It shows a few of the 7–letter words in the Direct_7 file (which contains all 7-letter words) and their transformed corresponding members inside the Preliminar AIndex_7, AIndex_7, Ordered AIndex_7, Reduced AIndex_7, Index_7 and Anagrams_7 files. At the end, the very important and needed pointers between the files Index_7 and Anagrams_7 are added.

TABLE 6.1. Construction of Index_7 and Anagrams_7.

Direct_7	Preliminar AIndex_7	AIndex_7	Ordered AIndex_7	Reduced AIndex_7	Index_7		Anagrams_7
…	…	…	…	…	…	.	…
…	…	…	…	…	…	.	…
ACABALO	ACABALOAAABCLO	ACABALOAAABCLO	ACABALOAAABCLO	ACABALOAAABCLO	AAABCLO	→	ACABALO
…	…	ACABALO#AABCLO	ACOLABAAAABCLO	ACOLABA			ACOLABA
…	…	ACABALOA#ABCLO	ALOCABAAAABCLO	ALOCABA			ALOCABA
		ACABALOAA#BCLO	BACALAOAAABCLO	BACALAO			BACALAO
		ACABALOAAA#CLO	ACABALOAAABCL#	ACABALOAAABCL#	AAABCL#	→	ACABALO
		ACABALOAAAB#LO	ACOLABAAAABCL#	ACOLABA			ACOLABA
		ACABALOAAABC#O	ALOCABAAAABCL#	ALOCABA			ALOCABA
		ACABALOAAABCL#	BACALAOAAABCL#	BACALAO			BACALAO
			…	…	…	…	…
		ACABALO##ABCLO	ACABALOAABCLO#	ACABALOAAACLO#	AAACLO#	→	ACABALO
		ACABALO#A#BCLO	ACABALOAABCLO#	ACOLABA			ACOLABA
		ACABALO#AA#CLO	ACABALOAABCLO#	ALOCABA			ALOCABA
		…	ACOLABAAABCLO#	BACALAO			BACALAO
		…	ACOLABAAABCLO#	ACABALOAABCLO#	AABCLO#	→	ACABALO
			ACOLABAAABCLO#	ACOLABA			ACOLABA
		ACABALOAAABC##	ALOCABAAABCLO#	ALOCABA			ALOCABA
		…	ALOCABAAABCLO#	BACALAO			BACALAO
		…	ALOCABAAABCLO#				
		…	BACALAOAABCLO#				
			BACALAOAABCLO#				
			BACALAOAABCLO#				
ACOLABA	ACOLABAAAABCLO	ACOLABAAAABCLO	ACABALOAAABC##	ACABALOAAABC##	AAABC##	→	ACABALO
		ACOLABA#AABCLO	ACOLABAAAABC##	ACOLABA			ACOLABA
		ACOLABAA#ABCLO	ALOCABAAAABC##	ALOCABA			ALOCABA
		ACOLABAAA#BCLO	BACALAOAAABC##	BACALAO			BACALAO
		ACOLABAAAA#CLO	ACABALOAAABL##				
		ACOLABAAAAB#LO	ACOLABAAAABL##				
		ACOLABAAAABC#O	ALOCABAAAABL##				
		ACOLABAAAABCL#	BACALAOAAABL##				
		ACOLABA##ABCLO	…				
		ACOLABA#A#BCLO	…				
		ACOLABA#AA#CLO					

		… ACOLABAAAABC##					
ALOCABA	ALOCABAAAABCLO	ALOCABAAAABCLO …	…	…		.	
BACALAO	BACALAOAAABCLO …	BACALAOAAABCLO …					
ENLUTAR … …	ENLUTARAELNRTU … …	ENLUTARAELNRTU ENLUTAR#ELNRTU … ENLUTARAELNRT# ENLUTAR##LNRTU … ENLUTARAELNR## … …	ENLUTARAELNRTU LENTURAAELNRTU NEUTRALAELNRTU ENLUTARAELNRT# LENTURAAELNRT# NEUTRALAELNRT# … ENLUTARELNRTU# LENTURAELNRTU# NEUTRALELNRTU#	ENLUTARAELNRTU LENTURA NEUTRAL ENLUTARAELNRT# LENTURA NEUTRAL … ENLUTARELNRTU# LENTURA NEUTRAL	AELNRTU AELNRT# … ELNRTU#	→ → … →	ENLUTAR LENTURA NEUTRAL ENLUTAR LENTURA NEUTRAL … ENLUTAR LENTURA NEUTRAL
LENTURA …	LENTURAAELNRTU …	LENTURAAELNRTU LENTURA#ELNRTU … LENTURAAELNRT# LENTURA##LNRTU … LENTURAAELNR## …	ENLUTARAELNR## LENTURAAELNR## NEUTRALAELNR## ENLUTARAELNT## LENTURAAELNT## NEUTRALAELNT## …	ENLUTARAELNR## LENTURA NEUTRAL ENLUTARAELNT## LENTURA NEUTRAL …	AELNR## AELNT## …	→ → …	ENLUTAR LENTURA NEUTRAL ENLUTAR LENTURA NEUTRAL …
NEUTRAL …	NEUTRALAELNRTU …	NEUTRALAELNRTU NEUTRAL#ELNRTU … NEUTRALAELNRT# NEUTRAL##LNRTU … NEUTRALAELNR## …	…				

Once the part of the Preliminar AIndex_7 is constructed, as explained above, we proceed to the addition of strings containing blank tiles to complete the entire AIndex_7 file. To obtain the strings with only one blank, we proceed as follows: for every string belonging to Preliminar AIndex_7, replace by a blank tile (the symbol # is used to represent a blank tile) one letter of the string; therefore for every string in Preliminar AIndex_7 we obtain 7 more strings. For instance, the string AAABCLO which forms the word BACALAO originates the following set of strings: {#AABCLO, A#ABCLO, AA#BCLO, AAA#CLO, AAAB#LO, AAABC#O, AAABCL#}. The concatenation of BACALAO with each of these strings becomes a new member of the AIndex_7 file. Therefore the new members of AIndex_7 are: {BACALAO#AABCLO, BACALAOA#ABCLO, BACALAOAA#BCLO, BACALAOAAA#CLO, BACALAOAAAB#LO, BACALAOAAABC#O, BACALAOAAABCL#}. There is no need to worry about equivalent members like BACALAO#AABCLO and BACALAOA#ABCLO since they become duplicated members, once ordered, in Ordered AIndex_7 (See Table 6.1), duplications no longer appear in Reduced AIndex_7.

To help eliminate these duplications and to group words produced by the same string the lexicographical order is essential (the blank tile "#" goes at the end of the alphabet). The blank tiles are introduced in AIndex_7 without ordering the associated string. The orderings of the associated strings of the members of AIndex_7 and of the members of AIndex_7 is performed in Ordered AIndex_7. For more examples see Table 6.1.

Next the strings containing 2 blanks are produced; there should be = 7! / 2!5! = 21 for every string contained in AIndex_7. In our BACALAO example the added strings are: {##ABCLO, #A#BCLO, #AA#CLO, #AAB#LO, #AABC#O, #AABCL#, A##BCLO, A#A#CLO, A#AB#LO, A#ABC#O, A#ABCL#, AA##CLO, AA#B#LO, AA#BC#O, AA#BCL#, AAA##LO, AAA#C#O, AAA#CL#, AAAB##O, AAAB#L#, AAABC##}. Remember that after being ordered, all duplicated member are eliminated in Reduced AIndex_7. See Table 6.1 for more examples.

When blanks appear, there are many repetitions of words in the Anagrams files, since the blank tiles can be replaced for any letter in the alphabet. The Anagrams files become larger, but in return the speed of searching all valid moves, when blanks appear on a player's rack, becomes very fast.

6.1.3 Septets and Septets9 files

When the board is clear, only seven-letter bingos can be played. The septets file contains all possible strings of 7 letters that produce a 7-letter bingo, and serves to calculate the expectation of making a bingo. If the board is not clear, it is possible to make 7-letter and 8-letter bingos. Septets9 are used to calculate the expectation value e of making a 7-letter or 8-letter bingo. A member of septets9 contains: 1) a string of 7 tiles, the pure septet or the septet of the reduced octet and 2) the sum of its tiles and 3) a delta value that indicates if the member can be placed on the board. Given a board, the deltas for all members of Septets9 are calculated very fast using the Halo of the played board T and the horizontal and vertical sets (H_σ and V_σ) for a given cell σ (they indicate the permisible letters for a given square). The septets and, above all, the septets9 files are the heart and soul of Heuri's heuristic to evaluate a move.

6.2 Preliminaries of Heuri's move generator

In this section Heuri's method for generating all possible moves is described. This is independent of the strategy for selecting the move to be played, which involves the evaluation function to be described later on. The pure beauty of Heuri's move generator lies on its simplicity and elegance and its efficiency relies on the power of anagrams which are the heart and soul of the generator.

Heuri's novel move generator is very different than the ones used by the most known Scrabble engines found in Computer Scrabble literature. Most engines use a Directed Acyclic Word Graph (DAWG) to somehow allocate words, and then they "walk the dawg" (follow a path of the graph to check for the existence of words). Some engines also used GADDAG (see 4.1.2); when using GADDAG, moves are produced approximately twice faster than when using DAWG (see 4.1.1).

Instead, Heuri's generator takes full advantage of the power of Anagrams. The key two ideas are:

1) The storage of words into two groups of files (Index_n and Anagrams_n), where n stands for the number of letters of the strings and words (16 > n > 1);

2) The wise link between Index_n and Anagrams_n (Index_n are the group of strings of length n that have at least one anagram, Anagrams_n are the group of Anagrams of length n). This link will help to walk through the Computer Lexicon to generate candidate moves. After defining the concepts a better explanation of how this move generator works will be given.

6.2.1 Some useful concepts

Let us define and recall some concepts that will be useful in the explanation and understanding of how Heuri's move generator works.

For a given played board T we will define a series of concepts.

Intervals

An *interval* in row r (resp. column c) is a collection of, more than one, consecutive unoccupied squares on the board, that are *not next to* (have distance greater than one) an occupied square of row r (resp. column c).

Molecules

Let us recall the definition of a *molecule*:
A *molecule* is a maximal collection of *consecutive* tiles, they have distance one between them, on the played board T.

The Geometric Halo or Halo

The definition of Halo taken from Collins dictionary (in English, the official reference on Scrabble) is

1. A disc or ring of light around the head of an angel, saint, etc., as in painting or sculpture

From the rules of Scrabble (section 3.2, *Scrabble Rules*), if the board is not empty, in order to place a word, it necessarily has to be connected to at least one word on the board. But this is not sufficient, in order to be a valid move all the words formed by the move have to be valid words contained in the lexicon (from Heuri's point of view contained in the computer lexicon).

A useful tool to check the validity of moves was developed; it consists in pre-calculating the possible letters that preserve word validity in a given collection of empty squares that correspond to the *1-neighborhood set* of a given nonempty board.

The *1-neighborhood set* of a given nonempty board is the set S of empty squares σ which are *next to* (have taxi or Manhattan distance 1) to the played board. Let us call this set the *Halo* of the played board. See Fig. 6.2 to geometrically see the Halo of a certain board position; notice that it resembles the common notion of Halo (a ring of light) also known as aureole. The Halo consists of the empty squares surrounded by the green ring and the played board in Fig. 6.2.

A special case arises when the board is empty. Here the *Halo* is defined as a single empty square σ = (8,8)= 8H=H8. In other words the *Halo* is the central cell of the board. Thus, the first move on the board, as well as any move, must intersect the Halo.

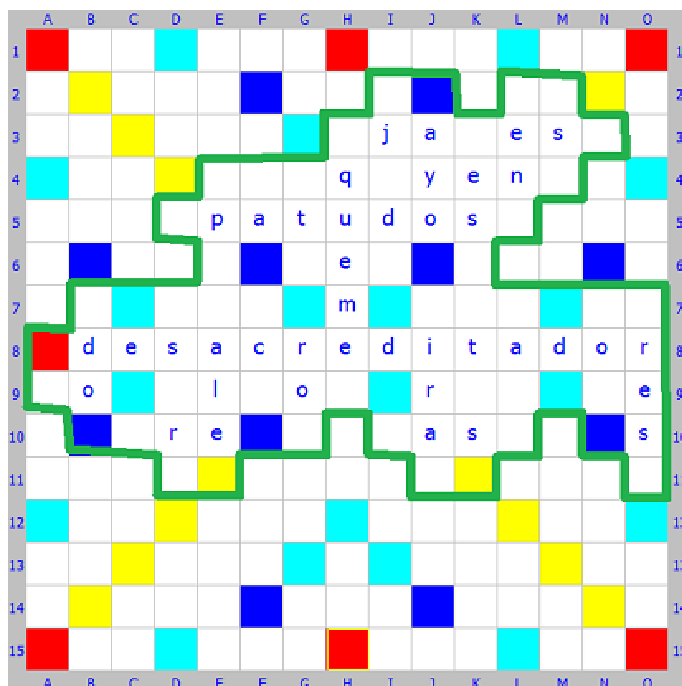


Fig. 6.2 Representation of the Halo of a played board

6.2.2 Calculating valid moves

Recall that the direction of play in Scrabble is north to south and west to east.

Let *m* and *n* be strings; then *mn* will indicate the concatenation of *m* and *n*. This concatenation is associative but, in general, not commutative. For example if *m*=SO and *n*=L then *mn*=SOL and *nm*=LSO.

For each empty square σ of the board we collect information on the tiles played above it (and below it, and to the right and left of it) including two collections of letters H_σ and V_σ.

Let us give the definitions of H_σ and V_σ:

Let *T* be a played board and let $\sigma=(i, j)$ be a cell of *ß* that does not belong to *T*. If there is a cell σ' in *T* directly

above σ (resp. below σ), that is, if $\sigma'= (i\text{-}1, j) \in T$ (resp. $\sigma'= (i+1, j) \in T$) then define n_σ, the north of σ (resp.

s_σ, the south of σ) as the molecule of column *i* containing σ'; otherwise n_σ (resp. s_σ) is the empty string.

Now, if n_σ or s_σ is nonempty, define H_σ as the set of letters λ such that $n_\sigma \lambda s_\sigma$ is a word of the lexicon; otherwise H_σ is the whole alphabet; in this thesis the alphabets used consist of the set of letters {A, B,…, Z}. A formal definition of an alphabet is given in section 4.1.1 The DAWG.

In a similar way one defines w_σ and e_σ, the west and east of σ, and V_σ.

Notice that if σ is not in the Halo of *T* then H_σ , and V_σ , is the whole alphabet. If σ belongs to *T* then one defines H_σ and V_σ to be the set with only one letter, namely, that represented by the tile on σ.

A horizontal (resp. vertical) word of the lexicon occupying the consecutive cells $\sigma_1 , \ldots , \sigma_r$ constitutes a valid move iff the letter represented by the tile on σ_i $(i \in [1, r])$ belongs to $H_{\sigma i}$ (resp. $V_{\sigma i}$) and some $\sigma_i \in$ Halo .

If *T* is the empty board the Halo is the central cell. Therefore any horizontal or vertical word of the lexicon that crosses the central cell is a valid move.

We will call H_σ (resp. V_σ) the *set of admissible letters horizontally* (resp. *vertically*) *at* σ, or short, the *horizontal* (resp. *vertical*) *set of* σ.

To produce horizontal (resp. vertical) moves H_σ (resp. V_σ) is used. Less intuitive are the one direction one tile moves since H_σ (resp. V_σ) corresponds to a vertical (resp. horizontal) formation of a word. The two direction one tile moves form two words (one horizontal and another vertical).

The following example shows how molecules, intervals and the horizontal and vertical sets of σ (H_σ and V_σ), are used to put valid words in Spanish on the board. See Fig. 6.3.

As an example, suppose a player has the rack {D E E I M N T} and the board shown in Fig. 6.3. If σ is an unoccupied square of row 10 then H_σ is the set of all letters {A,B,…,Z} with the following exceptions:

if σ = 10B (the 2nd square on row 10) then H_σ= {M, N, S, Y} since B8 (DO)λ is a valid word iff λ belongs to {M, N, S, Y}, and if σ = 10G (the 7th square) then H_σ= {A, B, E, I, L, N, O, S} since G8 (RO)A, (RO)B, (RO)E, (RO)I, (RO)L, (RO)N, (RO)O, (RO)S are valid words and (RO)λ is not if λ doesn't belong to{A,B,E,I,L,N,O,S}.

Some words which can be played on this row are: 10C I(RE) (a one-letter play containing the molecule RE), 10D (RE)MEND(AS)TEI(S) (a bingo containing the molecules RE, AS and S), 10A ENT(RE)MEDI(AS) (a bingo containing the molecules RE and AS), 10I M(AS)EEI(S) (containing AS and S), 10B MI(RE)N (containing RE), 10H ME(AS)EN (containing AS), 10M DE(S) (containing S).

Also 10A EN (in the interval preceding RE) and 10G ET (in the interval between RE and AS) can be played.
On column E, if σ is an unoccupied square then V_σ is the set of all letters {A, B,…, Z}, with no exceptions, since, when playing tiles only on column E, no horizontal words are formed.
If σ is an unoccupied square of column K, then V_σ is the set of all letters {A, B,…, Z} with the following exceptions:

if σ = K3 then V_σ = {D,L,M,S,T} since (3I (JA)D(ES), 3I (JA)L(ES), 3I (JA)M(ES), 3I (JA)S(ES), 3I (JA)T(ES) are valid words; and if σ = K9 then V_σ={E,O} since (9J (R)E, 9J (R)O) are valid words. See Fig. 6.3.

A couple of vertical words that can be played are: F4 M(A)TI(C)E (using molecules A and C), K3 D(ES)TE(T)E(S) (using molecules ES, T and S). A couple of horizontal words that use intervals are: 2J MIDEN on interval [2J, 2N] (a valid move since 2J MIDEN, J2 M(AYO), L2 D(EN), M2 E(S) are valid words) and 7J MEDI on interval [7J,7M] (valid since 7J MEDI, J7 M(IRA), K7 E(T), L7 D(A), M7 I(D) are all valid words).

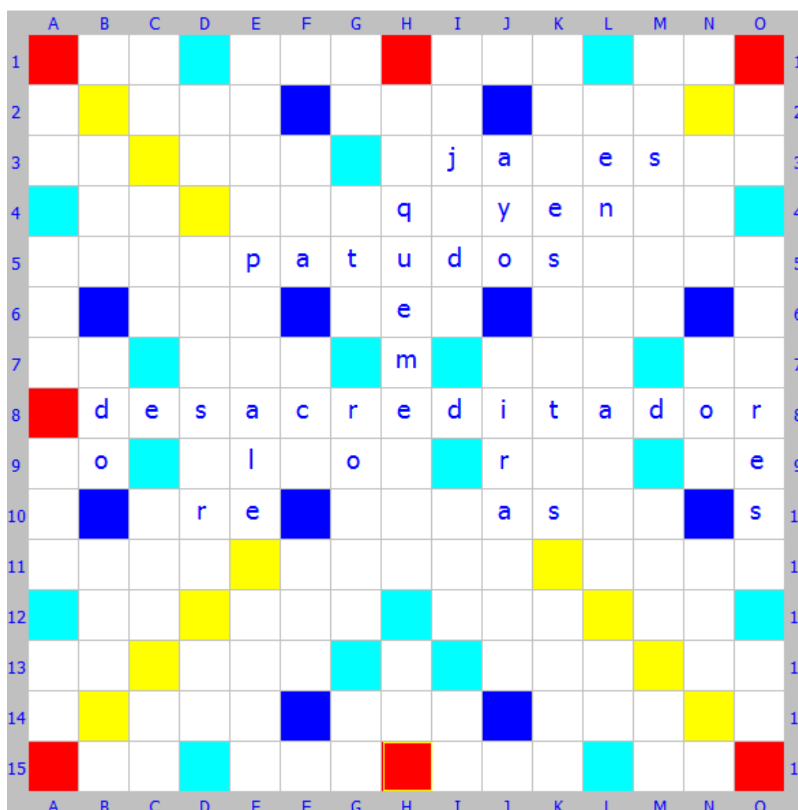


Fig. 6.3 An Scrabble Position illustrates how intervals, molecules and halos are used to place words

6.3 The Move generator

6.3.1 Intuition behind the Move generator

Let us describe how Heuri produces the valid candidate moves from a given position.

A board position and a rack are the only things needed to generate all possible moves on the board. When the board is empty the only restriction for every move is that it touches the center square. Due to the symmetry of the board let us only consider the horizontal moves; then a move consisting of a word of length n will produce n moves (one for every placement of each letter of the word on the center square).

To generate all horizontal moves from an empty board we proceed in the following manner:

All different rack subsets of cardinality greater than one are generated; then given these subsets we order them alphabetically and look for them inside the Index lists, if found they will point to the Anagrams lists which will give all anagrams of the given subsets. These anagrams give all the valid words of certain subsets. When the subsets are not found inside the Index lists this indicates that those subsets do not have anagrams, in other words, those subsets of letters have no valid words.

An important feature to remember is that all Index lists are linked with the Anagrams lists. To be precise, each member of an Index_n list points to a set of anagrams corresponding to the string of the Index_n member, where n stands for the length of this member.

To clarify things let us give some examples:

Given the rack {A, E, O, D, L, N, T} and an empty board let us give a few of the generated moves.
First let us search for all 7-letter words (bingos) from this rack; proceed as follows:

The generator first orders alphabetically the string corresponding to the rack; then it searches for the string ADELNOT inside the Index_7 list; once found it follows its pointer to the Anagrams_7 list which will give all anagrams (TOLENDA, DELTANO, ENTOLAD, ENTOLDA), see Table 6.2. Finally it produces all moves by placing each letter of a word, corresponding to an anagram, on the center of the board. For instance, the word TOLENDA yields 7 possible valid moves: (8B, 8C, 8D, 8E, 8F, 8G and 8H) TOLENDA.

Table 6.2, also shows the link between other strings and their anagrams; these strings follow the string ADELNOT (ADELNOU, ADELNOV, ADELNOZ). Since these strings are alphabetically ordered, an important thing to observe is that, in the Index_7 list, there is no string ADELNOX, it would be located after ADELNOV. This means that there are no anagrams corresponding to such string, in other words, there are no 7-letter words that can be formed with the rack {A, E, O, D, L, N, X}.

Table 6.2. Index_7 Anagrams_7

Index_7		Anagrams_7
		1 TOLENDA
		DELTANO
ADELNOT→1		ENTOLAD
ADELNOU→2		ENTOLDA
ADELNOV→3		2 LUNEADO
ADELNOZ→4		3 LEVANDO
		VELANDO
		NOVELAD DOVELAN 4 ENLOZAD

Now with the same rack {A, E, O, D, L, N, T} and an empty board, let us search for the moves of length 5 produced with the subset {A, E, O, L, T} (see Table 6.3).

Table 6.3. Index_5 Anagrams_5

Index_5		Anagrams_5
		1 LOTEA
		ATOLE
		ETOLA
		ELATO
		LATEO
AELOT→1		ALETO
AELOV→2		ALTEO
AELOZ→3		ALOTE
		2 ALVEO
		VOLEA
		OVALE 3 ALEZO AZOLE
		ZALEO

Similarly, the generator orders alphabetically the string AELOT; then it searches the ordered string AELOT inside the Index_5 list and then follows where it points to, inside Anagrams_5, thus obtaining the anagrams (LOTEA, ATOLE, ETOLA, ELATO, LATEO, ALETO, ALTEO, ALOTE), see Table 6.3. Then, it generates all moves placing every letter of a word, corresponding to an anagram, on the center of the board. For instance, the word LOTEA yields the following 5 possible valid moves: (8D, 8E, 8F, 8G and 8H) LOTEA.

A very important, useful and nice feature of Heuri's move generator is the treatment of the blank tiles. Blank tiles are treated as any other letter, in the sense that the process for looking for anagrams is exactly the same as the examples seen before. Therefore it is very fast to produce the anagrams corresponding to racks containing one or two blanks. In contrast with other move generators that involve DAWGs, these generators take more time to produce anagrams when the blank tiles appear, especially when a rack contains two blank tiles (See the end of subsection 4.1.1 *The DAWG*).

When constructing the computer lexicon, the strings that are members of the Index files that contain blank tiles were also linked to the list of all its anagrams in the anagrams files. This wise and convenient construction accounts for the speed of the move generator.

In Heuri's move generator, blank tiles appear in the Index lists, but they do not appear in the anagrams lists. Let us see an example to clarify things. Table 6.4 illustrates an example of a string DEEIMN# involving a blank tile (#) and shows its anagrams.

Table 6.4. Index_7 Anagrams_7

Index_7		Anagrams_7
		1 DIEZMEN
		2 MEDITEN
DEEIMNZ→1		MEDINES
DEEIMN#→2		INDEMNE
		EMIENDA
		REMIDEN
		ENDEMIA
		IMPENDE REDIMEN DIEZMEN

When the board is not empty, the generator first calculates the horizontal and vertical sets H_σ , V_σ of the board position. Then, using only the tiles from the rack, it finds all anagrams as described above. After the anagrams are found, the generator tries to put each anagram on the board. With each anagram the board is swept through all rows and columns; the intervals where each anagram fits are obtained using the board geometry and the HV-restrictions. The anagrams that fit in the intervals give all valid moves of an interval flavor. Examples using Fig. 6.4 are given.

Let us mention how moves of a molecule flavor are obtained. Using the letters of the rack and a collection of consecutive molecules an *extended rack* is made up and, as described above, anagrams are produced for this extended rack. Finally, using the geometry of the board and the HV-restrictions previously calculated, it determines which anagrams can be placed and gives the moves. Examples are shown using Fig. 6.4.

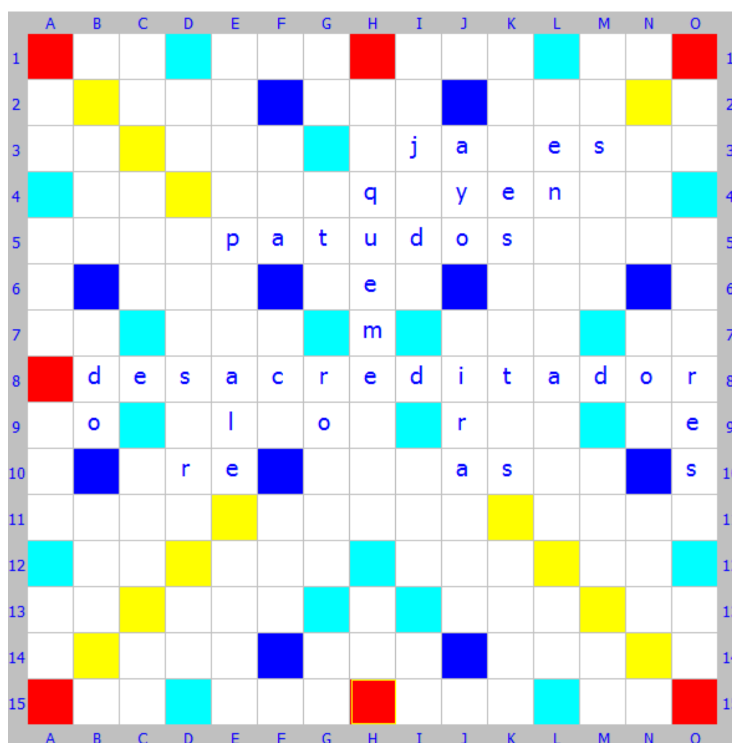


Fig. 6.4 A certain board position

Suppose we want to search for all 7-letter words (bingos) from the rack given {D E E I M N T} in the example corresponding to Fig 6.4. After searching the Index7 list and "walking to" the Anagrams7 list the only anagram found is MEDITEN, but it cannot be placed on the intervals of the board due to the geometry of the board and the HV-restrictions.

After observing the geometry of the board, the only three possible places to play the word MEDITEN are: 1) on column A, 2) on row 2 and 3) on row 11.

Let us analyze each of these possibilities using the HV-restrictions.

On column A, if σ is the empty square A8 then V_σ is the empty set, and when σ is the empty square A9 then V_σ={CH,D,F,J,L,N,Ñ,R,S,T,Y}. Therefore, due to the geometry of the board, the only candidate move to place MEDITEN on column A is A9 MEDITEN. But, since M does not belong to V_σ, A9 MEDITEN cannot be placed on the board, observe Fig. 6.4.

On row 2, due to the halo's restrictions and the geometry of the board we deduced that MEDITEN can't be placed. On row 11, if σ is the empty square 11D then H_σ = {E, O}, so we try placing on the board 11C MEDITEN, but if σ is the empty square 11E then H_σ = {A,E,F,O} and D does not belong to H_σ, therefore MEDITEN can't be placed on 11C. A similar analysis concludes that MEDITEN can't be placed on row 11, see Fig. 6.4.

Let us now search for all bingos on row 10, from the given rack {D E E I M N T}, in the example shown in Fig. 6.4. After finding that, due to the geometry on row 10, there are no intervals where to put a seven letter bingo, we proceed to look for longer bingos by using molecules.

First to get the extended rack, molecules on row 10 are detected and taking into account the geometry (number of spaces between the different molecules and the edges of the board) on column 10, it is deduced that the maximal collection of subsets of molecules that could make a bingo is: {(RE),(AS)}, {(RE),(AS),(S)}. Thus, the following extended racks are constructed: {DEEIMNTREAS}, {DEEIMNTREASS}; then using Index11 linked to Anagrams11, and Index12 linked to Anagrams12, the following anagrams are found: DETERMINASE, DEMENTAREIS, DESENTARIME, DEMERITASEN, AMEDRENTEIS, SEDIMENTARE, ENTREMEDIAS, ENMIERDASTE (in Anagrams11); SEDIMENTARES, ENMERDASTEIS, DETERMINASES, REMENDASTEIS, DESENTARIMES, DESESTIMAREN, MERENDASTEIS (contained in Anagrams12). Finally using the geometry of the board, and the halos restrictions, it is found that the valid moves that are bingos, on row 10, are: 10A_ENT(RE)MEDI(AS) and 10D_(RE)MEND(AS)TEI(S), observe Fig. 6.4.

In summary, using the letters of the rack and those of a (possibly empty) collection of consecutive molecules, from the board, an *extended rack* is made up and Heuri's generator finds using lookup tables (wisely linked), and therefore very fast, the list of anagrams of it; one keeps only the valid words that can be placed on the board, according to the geometry of the board (number of spaces between the different molecules and the edges of the board) and the HV-restrictions.

The use of the lookup tables, anagrams, molecules and intervals makes Heuri's move generator very fast. These tables use 338 MB, a reasonable amount of memory for present day computers.

6.3.2 A loaded board

We consider the unloaded board ß = {1,2,...,15}×{1,2,...,15}, which we also denote by {1,2,..., 15}×{A,B,...,O} (see Fig 3.1). Elements of ß are called cells. A loaded board is an assignment, to every cell $\sigma \in$ ß, of the following information:

1) An element λ_σ of {a,b, … , z, ñ, %, 0} . (The symbol % stands for rr; 0 represents an empty cell; in English we will not use ñ and %)

2) An element v_σ of {0,1,2,3,4,5,8,10} called the value of σ. (When a blank becomes a letter the value of the letter is 0).

3) An element LF_σ of {1,2,3} called the letter factor of σ.

4) An element WF_σ of {1,2,3} called the word factor of σ.

5) An element b_σ of {0,1} (a Boolean variable); it is 1 iff the distance from σ to the *played board* {$\tau \in$ ß : $\lambda_\tau \neq 0$ } is 1 (that is, iff σ belongs to the Halo).

6) Subsets H_σ and V_σ of {a,b,…,z,ñ,%} if $\lambda_\sigma = 0$.

A board is determined by (and determines) the played board. Every time the opponent plays the loaded board is recalculated.

6.3.3 Generating the moves

Moves that involve exchanges will be contemplated later on, in the evaluation process of the best move.

Generating the moves (The Anagram Method)

To explain the Anagram Method which is heavily based on anagrams, besides the essential word explanation, pseudocode is given in Table 6.5.

TABLE 6.5. The Anagram Method

```
Subracks=GenerateRackSubsets(Rack);
(H,V)=CalculateHandVSets(PlayedBoard,Rack,
Direct_n,Inverse_n);
AnagramsForIntervals=FindAllAnagrams(Subracks,
Index_n,Anagrams_n);
// If the Board is empty then //
If IsHalo(LoadedBoard,CentralSquare) then
  PlaceAnagramsInHorizIntervalCoveringCentralSquare
  (AnagramsForIntervals,8) // on row 8
Else
  // 1-tile moves
  For row=1 to 15
    For column=1 to 15
      If IsHalo(PlayedBoard,row,column) then
        PlaceLetter(H,V,row,column,rack);
      End If
    End For
  End For
   // Horizontal moves
  For row=1 to 15
      PlaceAnagramsInHorizontalIntervals
      (AnagramsForIntervals,H,PlayedBoard,row);
      m=CalculateNumberMoleculesHorizontal
      (PlayedBoard,row);
      PlaceAnagramsInHorizontalMolecules
     (Rack,Index_n,Anagrams_n,H,PlayedBoard,row,m);
  End For
   //  Vertical moves
  For column=1 to 15
      PlaceAnagramsVerticalIntervals
      (AnagramsForIntervals,V,PlayedBoard,column);
      m=CalculateNumberMoleculesVertical
      (PlayedBoard,column);
      PlaceAnagramsInVerticalMolecules
 (Rack,Index_n,Anagrams_n,V,PlayedBoard,column,m);
  End For
End If
```

Function FindAllAnagrams(Subracks, Index_n, Anagrams_n) would call Find_Anagrams(Subrack) for every Subrack of Subracks and would take the union of the resulting anagrams. Procedure PlaceAnagramsInHorizontalMolecules is given in Table 6.6. PlaceAnagramsInVerticalMolecules would be similar, replacing "H" by "V" and "row" by "column".

TABLE 6.6. PlaceAnagramsInHorizontalMolecules procedure

```
For i=1 to m
  For j=I to m
     Enlarged_rack=ExtendRackWithMolecules
     (Rack,PlayedBoard,row,i,j);
     Subracks=GenerateRackSubsets(Enlarged_rack);
     Anagrams= FindAllAnagrams
                   (Subracks,Index_n,Anagrams_n);
     PlaceAnagramsInHorizontalMoleculesSubset
     (Anagrams,H,PlayedBoard,row,i,j);
  End For
End For
```

Suppose a played board T and a rack are given. We want to collect all valid moves.

First using the given rack all possible subracks are generated. Next, the sets H_σ and V_σ are calculated, where σ is any cell.Then, using the subracks and the lists Index_n and Anagrams_n (where n is the length of the subrack, $2 \leq n \leq 7$) the anagrams for each subrack are found as explained earlier using the function Find_Anagrams(subrack).

If the central square belongs to the Halo (the geometric Halo) then this means that the played board T is empty. Then we only need to place all these possible anagrams horizontally and in such a way that the central square is covered by a tile belonging to the anagram. We decided not to include the vertical words for the first move. Else if the central square does not belong to the Halo then T is nonempty and we proceed in the following way:

If T is nonempty, we first consider 1-tile moves. A 1-tile move at a cell σ, at distance 1 from T, is valid iff it represents a letter of $H_\sigma \cap V_\sigma \cap rack$.

From now on we assume that all words that are considered to be played belong to the lexicon and the tiles played on empty cells constitute a subrack of the given rack.

Next we consider moves of length k>1 on row r ($1 \leq r \leq 15$) and column c ($1 \leq c \leq 15$) contained in intervals. A located string w ($w=w_1 w_2 \ldots w_k$) of length k>1 occupying the consecutive cells σ_1 , σ_2 , …, σ_k of row r (resp. column c) is a valid move iff $w_i \in H_{\sigma_i}$ (resp. $w_i \in V_{\sigma_i}$), i=1,…,k. We will refer to this condition as the HV-restriction. This is very fast since we already computed the sets H_{σ_i} and V_{σ_i}.

Some H_σ and some V_σ may not intersect the rack. This information and the fact that an interval on row r (resp. column c) cannot be at distance one from a molecule on row r (resp. column c) help a lot the calculation of the geometry of where to place an anagram, leading to a fast generation of valid moves.

Suppose a board and a rack are given. We want to collect, for every row r, ($1 \leq r \leq 15$) and two cells (r,j), (r,j') (j< j'), the valid moves starting at (r, j) and ending at (r, j').

Discard all j such that (r, j-1) lies on the played board and all j' such that (r , j'+1) lies on the played board; that is (r, j) (resp. (r, j')) is immediately to the right (resp. left) of a molecule.

Let E, the enlarged rack, be obtained by adjoining to the rack R (which may contain blanks) the letters on $\sigma=(r,k)$ with $j \leq k \leq j'$ such that $\lambda_\sigma \neq 0$. Now consider subracks of E of length n = j'+1-j and the set of members of Anagrams_n, obtained looking up the subracks in Index_n, which are anagrams of such subracks. Such members we call candidates. To see if a candidate *w* gives a valid word we do the following. Place the candidate *w* on row r between (r,j) and (r,j'). If $\sigma=(r,k)$, with $j \leq k \leq j'$ then the letter of *w* on σ must belong to H_σ (resp. $\{\lambda_\sigma\}$) if $\lambda_\sigma = 0$ (resp. ($\lambda_\sigma \neq 0$)). If the candidate satisfies this we call it a valid horizontal candidate on row r, coordinates (r,k).

In a similar way we define valid vertical candidates on column c.

The list of valid words is the list of all valid horizontal and vertical words.

6.4 Results and conclusions of Heuri's move generator

To illustrate Heuri's move generator speed, we compared it with Quackle 2015 engine by performing 3 experiments using an English lexicon and an empty board. The experiments consisted in producing all the possible moves for 3 given different racks. The first experiment involved a rack with 2 blank tiles; in this experiment Heuri turned out to be (when Quackle used a GADDAG to store the lexicon) 30 times faster than Quackle. The other experiments involved racks with one and zero blanks; refer to Table 6.7 for the results.

TABLE 6.7. Shows the times to generate all moves on an empty board (ms stands for milliseconds).

Rack	*Quackle's time*	*Heuri's time*	*Speed Ratio*
AENRS##	**480.7 ms**	**15.9 ms**	**30.2**
AEINRS#	96.6 ms	3.4 ms	28.4
AEINRST	11.5 ms	0.5 ms	23

The last column shows how many times faster is Heuri's move generator compared to Quackle's for a specific example.

For a nonempty board with a single word on the board which is 8F ANEROID the following table 6.8 arises:

Table 6.8. Shows the times to generate all moves on a non-empty board with 8F ANEROID (ms stands for milliseconds).

Rack	*Quackle's time*	*Heuri's time*	*Speed Ratio*
AENRS##	**1431.1 ms**	**45.7 ms**	**31.3**
AEINRS#	297.2 ms	10.9 ms	27.3
AEINRST	25.3 ms	2.1 ms	12

After playing 1000 games Quackle vs. Heuri, playing in English, using the lexicon twl06 and Quackle using GADDAG to store the lexicon, the following results were obtained:

- The average number of turns was 24.25 turns.
- The maximal number of turns was 37 turns.

- Average number of legal moves analyzed per game was: 23541.95 moves.
- The branching factor (average number of legal moves per turn) was 970.88
- Average Quackle time per game: 331.26 ms.
- Average Heuri time per game: 126.69 ms.
- Time ratio Quackle/Heuri (Q/H) per game:2.61
- Time ratio Q/H of the first 12 moves: 5.14

Time comparisons per move between Quackle and Heuri, after playing 1000 games, are shown in Table 6.9.

TABLE 6.9. Time comparisons per move between Quackle and Heuri

Move Number	Quackle's time	Heuri's time	Time ratio Q/H
1	5.93 ms.	0.3 ms.	19.58
2	9.59 ms.	0.95 ms.	10.07
3	13.14 ms.	1.46 ms.	8.99
4	13.71 ms.	1.87 ms.	7.33
5	14.49 ms.	2.28 ms	6.37
6	13.03 ms.	2.64 ms.	4.93
7	14.93 ms.	3.13 ms.	4.77
8	17.08 ms.	3.65 ms.	4.68
9	15.7 ms.	4.02 ms.	3.91
10	17.15 ms.	4.59 ms.	3.74
11	14.45 ms.	4.91 ms,	2.94
12	19.43 ms.	5.61 ms.	3.46
13	13.97 ms.	5.81 ms.	2.4
14	16.55 ms.	6.42 ms.	2.58
15	15.63 ms.	6.82 ms.	2.29
16	16.67 ms.	7.47 ms.	2.23
17	15.18 ms.	7.66 ms.	1.98
18	15.16 ms.	8.42 ms.	1.8
19	13.14 ms.	8.47 ms.	1.55
20	13.83 ms.	9.14 ms.	1.51
21	9.02 ms.	7.83 ms.	1.15
22	8.33 ms.	7.3 ms.	1.14
23	4.06 ms.	4.53 ms.	0.9
24	3.23 ms.	3.33 ms.	0.97

On average when comparing the Anagram Method used by Heuri to the GADDAG Method used by Quackle, the Anagram Method turns out to be 2.61 times faster than the GADDAG Method when playing a complete game.

The time ratio Q/H decreases as the move number increases (See Table 6.9) being the first 12 moves the most profitable time saving for the Anagram Method used by Heuri compared with the GADDAG method used by Quackle. The first 12 moves are more significant than the moves that follow; this is due to a characteristic found in Scrabble games: a player often gains advantage in the first moves and it is difficult for the other player to recover from this disadvantage. After playing 1000 games, the player that was winning at move 12 finally won the game in 75% of the times. In the first 12 moves the time ratio Q/H equals 5.14 Therefore the Anagram Method would allow more time to analyze better the first 12 moves than the GADDAG method resulting in better play.

The GADDAG and DAWG methods have many dead ends; the GADDAG has more dead ends than the DAWG. These dead ends cost time in move generation. An advantage of the Anagram Method is that it has NO dead ends, therefore it is faster than the other methods.

The GADDAG algorithm is still non-deterministic in that it runs into many dead-ends. Nevertheless, it requires fewer anchor squares, hits fewer dead-ends, and follows fewer arcs before detecting dead-ends than the DAWG algorithm [Gordon94].

The Anagram Method is a deterministic algorithm, there is no luck involved. It has NO dead-ends, it only checks if the anagrams found with the tiles in the rack and on the board fit in the board as legal moves. For more info about the Heuri's Move generation phase see “The Anagram Method” in [GonzalezAlquezar18] .

Chapter 7.

Move Evaluator

Once we have all valid moves, it is necessary to decide which move to make. In this chapter we tackle this issue by introducing and explaining several heuristics that indicate us what move to make. These heuristics make a balance between the points score by the move, and the potential points that could be made, in our next turn, with the tiles that remain in our rack, after playing our move. In order to achieve this task, the heuristics take into account our tiles in the rack, the tiles on the board and their position, and the tiles left to be played, that is the union of the tiles in the bag and the tiles on the opponent's rack.

7.1 Introduction to Heuri

The first steps towards building a Scrabble engine, with an inference strategy which could balance the rack in play, was a **C** function that received a 7 letter string and returned the probability of making a bingo, for all 128 possible exchanges of the given string. In order to do this, we built a file, the *septets*, that contains all 7 letter strings that produce a bingo. An example of how to calculate this probability is given in section 7.2.

An essential part of the evaluation of a certain move, involves a method to quantify what is left on the rack after the move is made (the rack *residue* or *leave*). Since there is a 50 bonus point when you use all your tiles (see subsection 3.2.9 The Fifty Point Bonus) , a natural way to evaluate a move is by adding its score to its residue's expectation of making a bingo in the next move.

Several ideas about the heuristics used by humans when playing Scrabble, for instance the use of anagrams, yielded the construction of the Computer Lexicon and the *Heuri* engine.

A difference between Heuri and other Scrabble engines, like Maven and Quackle, is that the evaluation of a *leave* (rack residue) is calculated as a sum of products of probabilities times expected rewards (see sections 7.2, 7.3 and 7.4) rather than calculating values of individual tiles, summing them, and finally adjusting for synergy and other factors ([Sheppard, 2002b], Chapter 5), or obtaining precalculated values of multisets of tiles, by playing thousands of Quackle versus Quackle games [KatzO'Laughlin06].

7.2 Probabilistic Heuristic of Heuri's first engine

An important part of a program that plays Scrabble in Spanish (or any other language) is the decision of what to leave in the rack. In [GonzálezRamírez08], we propose to give a numerical evaluation as follows:

$$v = j + p * b - d \qquad (7.1)$$

where j is the number of points made by the move, in which t tiles are played (it is assumed here that the bag has at least t tiles) ; $j = 0$ if the t tiles are changed rather than played on the board; p is the probability of obtaining a 7-letter bingo if t tiles are drawn at random from a set that is the union of the bag and the opponent's rack (which we call the "augmented bag"); b, the expected value of a bingo, is 77 (an average taken from 1000 games) or better:

$$b = 50 + 2.5(r + 1.92t) \qquad (7.2)$$

where t is the number of tiles drawn from the bag and r is the total number of points of the leave; d is a nonnegative number which is zero if the move is not weak from a defensive point of view. It is 20, say, if one puts a tile on the edge allowing a possible nine-timer; it is 10 if one plays a vowel next to a premium square allowing a sixtimer; it is 5 if one plays allowing a possible four-timer.

The equality for b was obtained as follows. First we wrote $b=50+k*e$ where e is the expected total of points (without premiums and bonus) of a bingo. The sum of the values of the 100 tiles is 192 and so the average value of a tile is 1.92 and $e=r+1.92t$. As an example, if the leave is{ZADO}, with $r=14$, and we take $t=3$ tiles from the bag we would expect these tiles to add, on the average, $t*1.92$ $(=3*1.92)$. Say these tiles are {VES} (adding 6 points). Together with {ZADO} these tiles form VEZADOS giving $e=20$ (approximately $r+1.92t$). But most of the bingos have premiums (very often, besides the 50 points bonus, the word score is multiplied by 2 and sometimes 3, 4 or 9).

Hence it is reasonable to multiply e by a constant k and we took $k=2.5$ because very often it gives for b a value close to the experimental bingo average 77. For example, in the frequent case $r=3,$ $t=4$ one gets $b=50+2.5(3+1.92*4)=76.7$.

It is better to explain the calculation of p using an example:

What is the probability p_s of obtaining the "septet" s={AAAAÑRR} (from which one can form the 7_letter bingo ARAÑARA) if one has the leave {AAAÑ}, one exchanges {HQV}, the augmented bag is the initial one (that is, equals "total bag – {AAAÑHQV}") and one draws 3 tiles from it?

Answer: If {AAAÑ} were not contained in {AAAAÑRR} p would be 0. However {AAAÑ} is contained in {AAAAÑRR} so we consider the difference set {AAAAÑRR}-{AAAÑ}={ARR}={AR^2} and the augmented bag: {AAAAAAAAARRRBBCCCC…XYZ}={A^{9}R^{3}B^{2}C^4…XYZ} and one then computes $p_s=C(9,1)*C(3,2)/C(93,3)$ (93 is the number of tiles of the augmented bag and the 3 in $C(93,3)$ is the number of tiles that are taken from the bag) where $C(m, n)$ is the binomial coefficient:

$$C(m,n)= m(m-1)(m-2)\dots(m-n+1)/n! \qquad (7.3)$$

The numerator has n factors and $C(m,n)$ is the number of n-element subsets of a set consisting of m elements. Notice that the denominator $C(93,3)$ does not depend on the septet s.

The probability p of obtaining a 7-letter bingo if one has the leave {AAAÑ} and the augmented bag: "total bag– {AAAÑHQV}" is then the sum of all p_s when s runs over all septets.

7.3 Heuri's second engine

Although Heuri's first theoretical engine has the essential part of the algorithm used by Heuri to evaluate a move, it needed some improvements. These improvements besides making the engine play stronger, allowed the engine to start playing automatically.

In [RamírezGonzález09], it was proposed to give a numerical evaluation of all potential moves as follows:

Let us recall some essential ideas from last section and introduce some new features.

An important part of a program that plays Scrabble is the decision of what to leave on the rack. In a move t tiles are played or exchanged and n-t tiles make up the leave r (excluding the endgame, $n = 7$).

The following heuristic function is used to evaluate all potential moves:

$$v = j + e - d \tag{7.4}$$

where j is the number of points made by the move, in which t tiles are played (it is assumed here that the bag has at least t tiles; j=0 if t tiles are changed rather than played on the board); e is the expected value of a bingo, given a leave r, if t tiles are drawn randomly from the *augmented bag*, that is, the union of the bag and the opponent's rack; d is a nonnegative number which is zero if the move is not weak from a defensive point of view. Presently, d=0. From all potential moves one with maximal v is chosen.

To explain our estimate of e define a *septet* to be a lexicographically ordered string of seven characters of {A,B, …,Z,#} (where # is a blank) from which a 7-letter word can be constructed; for example {AAAAÑRR} (yielding ARAÑARA) and {ACEINR#} (yielding RECIBAN) are septets but {AEEQRY#} and {ADEILOS} are not.

There are 126,972 septets. For the calculation of e the following formula was used

$$e = \sum_{i=1}^{126972} p_i (50 + k\sigma_i) \qquad (2)$$

(7.5)

Let us call this heuristic **Heuri_2.**
Heuri_2 is the heuristic used in the program Heuri presented in [RamírezGonzález09].

Here p_i is the probability (which might be zero) of obtaining the i-th septet, given a leave r consisting of 7–t tiles, if t tiles are drawn randomly from the augmented bag; σ_i is the sum of the values of the characters of the i-th septet. The existence of premium squares, hook words, bingos of length greater than 7 and experimentation have led us to take presently 2.5 as the value of the constant k.

The calculation of p_i was explained (using an example) in the previous section 7.2.

7.4 Improved heuristics (third and fourth engines)

7.4.1 Heuri's third engine

Although the formula (2) is a good estimate of e when the board is open, it ignores the fact that one may have a septet in the rack which cannot be placed as a bingo on the board. This often happens when the board is closed. To account for this, one can write:

$$e = \sum_{i=1}^{126972} \delta_i p_i (50 + k\sigma_i) \qquad (3)$$

(7.6)

where δ_i, which depends on the board, is 1 if the i-th septet can be placed as a bingo on the board and 0 otherwise. The calculation of δ_i, for all i, which is independent of the rack, is a task that is feasible because Heuri has a fast move generator based on anagrams.

7.4.2 Heuri's fourth engine

A collection of 1000 Scrabble games was gathered with the purpose of classifying the bingos made in those games into 3 categories according to its length : 1) seven-letter bingos, 2) eight-letter bingos and 3) X-letter bingos, with X > 8 . The results gave:

The total number of bingos from the collection of 1000 games was 6073 bingos, divided as follows: 2003 seven-letter bingos, 3972 eight-letter bingos and 98 bingos with more than 8 letters. This reinforced a previous thought: the heuristic function should take into account the probability of making 8-letter words besides the 7_letter words.

Let us define an *octet* as a lexicographically ordered string of eight characters of {A,B,…,Z,#} (where # is a blank) from which an 8-letter word can be constructed. A *reduced octet* is a lexicographically ordered string of seven characters of {A,B,…,Z,#} which forms an octet when adding a certain new character.

Let a *pure septet* be a septet which is not a reduced octet, in other words when added any character to the pure septet it does not become an octet (there is no valid 8-letter word).

There are 126,972 septets, 5,830 pure septets and 282,214 reduced octets. Notice that the set of septets contains all pure septets and 121,142 reduced octets which are also septets.

One desires to give an estimate of the probability of obtaining a 7-letter or 8-letter bingo given a leave and a board. The pure septets and the reduced octets give a total of 288,044.

Then the calculation of e is given in [GonzálezAlquézar12] by the formula:

$$e = \sum_{i=1}^{288044} \delta_i p_i (50 + k\sigma_i) \qquad (4)$$

(7.7)

where p_i , σ_i , k and δ_i were explained in (2) and (3).

To verify that Heuri was improving, when constructing the new engines, we performed two matches, each match consisting in 10,000 games. The first match confronted Heuri's third engine against Heuri's second engine; and the second match was Heuri's fourth versus Heuri's third engines. These matches indicated the improvement of the Heuri's third engine over the second engine, and showed how Heuri's fourth engine surpassed the third engine. To see more details see tables 9.4 and 9.5 in subsection 9.2.3.

7.5 Fishing

Intuitively a *fish* is a move which seeks a bingo in the next turn. More precisely, using Heuri's probabilistic heuristic, $v = j + e - d$, let us define a *fish* as a move where $e > j$ (its expected value of a bingo is greater than the points made by the move). The most common example of a fish is an exchange with $e > 0$. Almost all exchanges are fishing moves; however in the pre-endgame (when there are few tiles in the bag) we might change a Q tile to

avoid getting stuck with it at the end; it is likely that $e = 0$ since no more bingos can be placed on the board, or it is impossible to construct a bingo with your rack leave and the tiles left inside the bag [GonzálezAlquézar12].

The following is an example of a possible line of play using Heuri that shows and involves fishing moves:

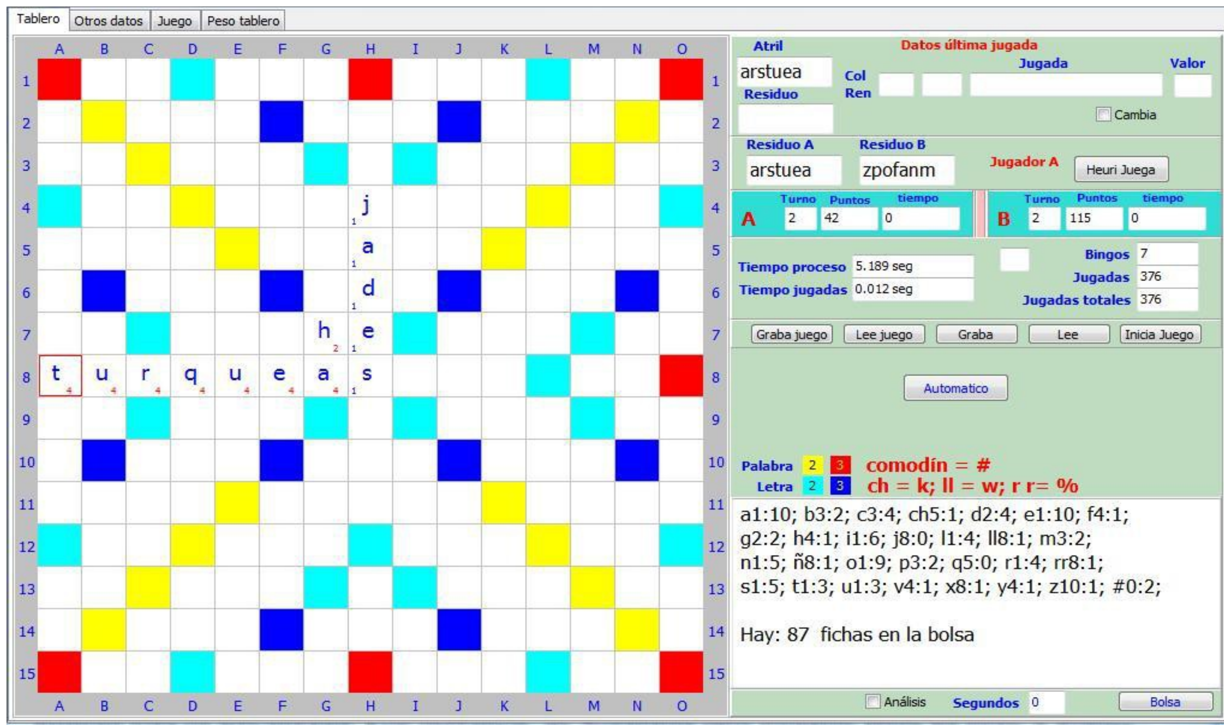


Fig. 7.1 Example of fishing moves using Heuri's engine

The first player has played (H4) jades 42pts. Then it is Heuri's turn. Heuri has the following rack: {r h t u u q e}. Using (1) $v = j + e - d$ with $d = 0$, and using (4) to calculate e, Heuri's engine gives us the following fishing moves:

Table 7.1. Examples of fishing moves

Coordinate and word or Exchange and leave	v	j	e	d
7G he	43.39	9	34.39	0
7H eh	43.39	9	34.39	0
I4 uh	39.85	19	20.85	0
5G ah	39.39	5	34.39	0
5H ah	39.39	5	34.39	0
Exch. h, Leave: (e q r t u u)	34.39	0	34.39	0
5E huta	33.21	14	19.21	0

Therefore Heuri plays 7G he for 9 points and takes an a out of the bag. The first player exchanged letters and Heuri with the rack {r a t u u q e} plays 8A turqueas 106 points!

Fishing plays are important because they seek high scoring moves in next turns. Brian Sheppard, the developer of Maven, writes the following in [Sheppard2002b].

“Maven’s lack of a move generator for fishing may be its biggest weakness. All of the moves that Maven overlooked in its match against Adam Logan were fish.”

Brian Sheppard [Sheppard2002b] also mentions that there are less than 4 million distinct racks of 7 or fewer tiles, and proposes to learn a value for every single one. Quackle follows this advice by playing thousands of Quackle vs Quackle games, to learn these values for different languages and lexicons. They are known as the *superleaves* and they are pre-calculated before the engine plays. Quackle can play without them but the quality of play drops considerably.

Chapter 8.

Beyond the Heuristics

Besides the heuristics given in the previous chapter, in order to make a stronger engine, it is convenient to foresee one or two moves ahead. These helps us in estimating a defense value; when foreseeing one move ahead this value contemplates some possible opponent's moves to estimate a defense value, the idea is to prevent the opponent on making many points in his next turn. When looking two moves ahead, we estimate a value which balances the possible opponent's points in his next turn and our possible points in our next move. To make a better estimation of these values, we also contemplate the quality of the opponent´s rack, that is the possible tiles that our opponent holds, based on his last move; this is named *opponent modeling*.

All the previous heuristics do not involve samplings or simulations to calculate d (the defense value), in fact d=0 almost always. To improve Heuri's engine, we need to perform samplings (see 8.1 HeuriSamp) or do simulations (see 8.2 HeuriSim), to achieve a good approximation of the defense value of a certain position, and to foresee other strengths of a particular move. When using the engine HeuriSamp the defense value will be always positive, since d will be the average of certain values corresponding to the opponent's moves; the HeuriSim engine will admit positive and negative values for the defense value, since $d = k - l$, where l is a value corresponding to a second move of the first player, and k is a value corresponding to the first move of the second player. Thus $d \geq 0$ if $k \geq l$ and $d < 0$ if $k < l$.

8.1 HeuriSamp

The HeuriSamp engine evaluates a two-ply search, taking into account some of the first player's moves, and the opponent's greedy replies to those moves (the second player's maximal scoring replies); to evaluate the defense value d, the average of these scores is calculated. Once d is calculated we reevaluate the first player moves using the formula $v = j + e - d$ (explained in [GonzálezAlquézar12] and in section 7.3 Heuri's second engine). In other words, the *HeuriSamp engine* takes a sampling of Heuri's moves and, for each move evaluates an average maximal score value d of the opponent's move; finally v is reevaluated using formula $v = j + e - d$.

Explanation of Heuri and HeuriSamp engines

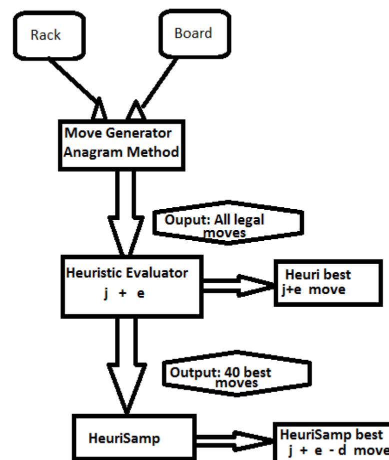


Figure 8.1. Heuri and HeuriSamp flow

Figure 8.1 shows the flow of two engines. The Heuri and HeuriSamp engines. HeuriSamp and Heuri share their first steps, both receive a rack and a board and using the Anagram Method [GonzalezAlquezar18] output all legal moves, then Heuri's engine performs a static evaluation of all legal moves using the heuristic function: $v=j+e-d$ (here $d=0$) with e = $\Sigma\, \delta_i p_i(50+k\sigma_i)$ i runs from 1 to 288,044 and then outputs Heuri's best move (the move that maximizes v). See [GonzalezAlquezar12] and (sections 7.3 and 7.4) .

Now if we are using HeuriSamp's engine we take the best 40 moves according to Heuri and for each one of them calculate its d (defense value). This d value is calculated in the following manner: For each one of the 40 candidate moves given by Heuri, we sample 300 racks (now they are randomly generated, but in the future they will be generated with Opponent Modeling); now for each of the 300 racks the highest value move of the opponent is computed obtaining a d_i (i runs from 1 to 300) , then d is computed by taking the average of all d_i's ; finally HeuriSamp outputs the best move, that is the one that maximizes $v_n = j_n + e_n - d_n$ where n runs from 1 to 40.

The following heuristic function is used in Heuri to evaluate all potential moves:

$$v = j + e - d \qquad (8.1)$$

where j is the number of points made by the move, e is the expected value of a bingo in the next turn and d is a non-negative number used for a defensive purpose (In **Heuri** $d = 0$ and in **HeuriSamp** $d > 0$).

The calculation of e is given by the following formula:

$$e = \sum_i^{totseptroct} \delta_i\, p_i\, (50 + k\sigma_i) \quad \text{(where } i \text{ runs from 1 to } \textbf{totseptroct}\text{)} \qquad (8.2)$$

where :

σ_i is the sum of the letters of the i-th septet.
k is a constant (usually 2.5) to account for board rewards
p_i is the probability to form the i-th septet.
δ_i equals 1 if the i-th septet can be placed on the board , otherwise $\delta_i = 0$.
totseptroct is the total number of strings which are either septets or reduce octets.

For the results using this engine, see 9.2.9 Match Results of HeuriSamp Engine.

8.2 HeuriSim

Let us explain the HeuriSim engine.

The main idea is to consider a 3-ply (depth 3) adversarial search tree, with leaf node terminal evaluation, in the following way: Player 1 analyses a move, Player 2 contemplates replies to Player's 1 move, and finally Player 1 analyses a reply move to Player's 2 move; the value calculated of this chain of moves is given a value and it is stored in a leaf node (a terminal evaluation corresponding to a branch of the tree).

Steps

1) Heuri generates and evaluates all possible moves of Player 1, given a certain board position and a rack.
2) The Player 1 top 40 moves, using the heuristic $j+e-d$ *where* $d=0$, are contemplated.
3) For each one of these 40 moves a value d_i (i runs from 1 to 40) is calculated in the following manner:
   3.1) Player 1 pseudo-plays move i , a new temporal board position $Tboard_i$ is obtained and the score is stored in $score_i$
   3.2) Player 2 generates 300 random racks, sum2=0 , avg2=0
   For each one of these 300 racks r_m (m runs from 1 to 300)
      Using $Tboard_i$ and rack r_m all possible moves for Player 2 are generated and evaluated
      Player 2 pseudo-plays move m, the move played is one with maximal j (pts. on board)
      a new temporal board position $Tboard_m$ is obtained and
      the score is stored in $score_m$
      Player 1 replenishes 100 times, obtaining 100 racks $r1_k$ (k=1 to 100)
      $sum1 = 0, avg1 = 0$
      For each $r1_k$ (k runs from 1 to 100)
         Using $Tboard_m$ and rack $r1_k$ all possible moves for Player 1
         Player 1 pseudo-plays move k, with maximal j (pts. on board)
         the score is stored in $score_k$
         $sum1 = sum1 + score_k$
      End For
      $avg1= sum1 / 100$ // calculates the average of the second move of Player 1
   $score_m = score_m - avg1$
   $sum2 = sum2 + score_m$
   EndFor
   $avg2= sum2 / 300$ // calculates the average value of the first move of Player 2 minus the
   d_i = avg2 // average value of the second move of Player 1. Then stored in d_i
   $v_i = j_i - d_i$ // A knew value v_i for each of the 40 moves is obtained using this 3 plies chain.
   EndFor
4) Sort(v_i) // Finally the values are ordered in decreasing order . Therefore the best move corresponds
   // to the move i with the greatest v_i

For the results using this engine, see 9.2.10 Match Results of HeuriSim Engine.

8.3 Opponent Modeling

Opponent modeling is a technique to recognize the strategy of an opponent and make predictions about their behaviour. It is the ability to recognize and anticipate the moves of an opponent. Refering to Scrabble, we observe the opponent's last move, to make predictions of some of the tiles the opponent might hold in the rack; the rest of the tiles are completed randomly (a rack contains a total of 7 tiles, except in many endgames).

To improve HeuriSamp and HeuriSim performances, it is convenient to use a certain form of opponent modeling: HeuriSamp and HeuriSim should play with the R_n-assumption. This is explained as follows: when it is HeuriSamp's or HeuriSim's turn to play they will look at the opponent's last move. The idea is to try to make an educated guess of the opponent's residue or leave (the tiles left on the opponent's rack after the move). Thus, they should estimate a number n , $0 \leq n < 7$, such that n of the (usually 7) tiles of its opponent are good tiles. For every n, with $0 \leq n < 7$ a collection R_n of good n-tiles racks has been previously constructed. For example, in English, $|R_0| = 1, |R_1| = 8 , |R_2| = 43, |R_3| = 168 , |R_4| = 407 , ... , |R_5| = 856 , |R_6| = 1769$,where | | denotes cardinality. Now we give part of the collection of the multisets of good n-tiles racks that conform R_n , for every n, with $0 \leq n < 7$,when playing in English.

R_0 Only consists of the empty set.
R_1 = { #, e, a, i, s, t, r, n }
R_2 ={ ## , #e, #a, #i, #o, #u, #s, #r, #n, aa, ae, ai, as, ar, an, ac, at, ad, al, ei, es, er, en, ec, et, ed, io, is, ir, in, ic, it, id, os, or, on, oc, ot, od, ol, om, sr, rn }
R_3 ={ ##e, ... , ocd }
R_4 ={ ##ee, ... , osnt}
R_5 ={ ##eea, ... , ouncd}
R_6 ={ ##eeea, ... , ourntd}

The multisets R_i ($i=1$ to 6) with cardinality i, were previously constructed. To construct them, we used the multisets M_i ($i=1$ to 6) with cardinality i. These multisets were constructed using all possible tiles and their Scrabble distributions. Then, using and empty board and a full bag of tiles, the Bingo probabilities of each member of the multisets M_i ($i=1$ to 6) were calculated. Finally the members with the highest probabilities of Bingo conformed the R_i ($i=1$ to 6).

Making an educated guess of the *n* value

The idea is to try to make an educated guess on the number of good tiles n , that the opponent keeps on the rack after playing.

To make educated guesses on the opponent's tiles, the following assumptions are made:

1) If the opponent has just played a Bingo (a move using all the 7 tiles on the player's rack), then the opponent kept zero good tiles, thus (the number of tiles played on the board is c=7 and $n=0$) .
2) If in the last move the opponent changed c tiles then,the number of good tiles the opponent has is $n=7-c$.
3) If in the last move the opponent played c tiles on the board making less than 30 points, then $n=7-c$.
4) If in the last move the opponent played c tiles on the board making 30 or more points, and if $7 > c > 3$, then $n=1$; if $c < 4$, then $n=2$.

Now, when using opponent modeling in HeuriSamp or HeuriSim, instead of taking 7 tiles randomly from the bag to generate the opponent's rack, we will look at the opponent's last move and using the above assumptions, we will estimate n . After estimating n, we will use the *augmented bag* (full bag minus the tiles on HeuriSamp or HeuriSim rack minus the tiles on the board), to calculate the most probable 10 racks belonging to R_n . Then, instead of generating the 300 random racks for Player 2, we will repeatedly take the 10 most probable members of R_n , and complete them randomly to form 7-tile racks, then we will have 300 educated guesses of racks.

The engines HeuriSamp and HeuriSim, improved very much, with the aid of Opponent Modeling, look at the results in Sections 9.2.9 and 9.2.10, Tables 9.17 and 9.20, and compare them with the results in Tables 9.15 and 9.19.

Chapter 9.

Results

This chapter presents results obtained by playing thousands of Heuri vs Heuri games (using Heuri’s third engine, see section 7.4), collecting important information and then calculating some statistics like the winning percentage of a player, the average final score or the frequency of certain words. These results are divided into two groups: the first group, presented in (9.1.1 General Statistics), which gives general figures of Spanish Scrabble and the second group, covered in (9.1.2 Word Statistics), which gives an important list of the most frequently played Bingos (words that use all 7 tiles of a player’s rack awarding them with a 50 bonus point). Section 9.2 contains match results of Heuri against humans and against Quackle. It also contains results showing the improvements made on Heuri's fourth engine, besides there are also results that indicate that Heuri plays better in Romance languagues; for these purposes matches in English, French and Spanish were performed. Then six more engines were constructed: the first one, Heuri_5, uses a non-constant FactorAtril value; the second one, Heuri_6, applies modified heuristics for the prefinal phase; the third engine, HeuriSamp, takes several random samples of the second player’s rack, they are used to do a 2-ply guided search, obtaining a defense value *d* and reevaluating the best first player’s moves; the fourth engine, HeuriSim, performs a 3-ply guided search to observe deeply into the first player’s moves. Finally, the fifth and sixth engines added Opponent Modeling to HeuriSamp and HeuriSim engines; they used the opponent's last move to make educated guesses of the opponent's tiles; this was a key factor to improve a lot, the results of both engines.

9.1 Data Findings

9.1.1 General Statistics

After playing 104,035 Heuri vs Heuri games the following statistics were obtained:

- The first player wins 57,221 games, ties 303 and loses 46,511. In other words :
  Player A wins 55.0017%, ties 0.2912% and Player B wins 44.7071%

- The average points per game were: Player A: 518.03 points and Player B: 501.73 points.

- An average game lasts 24.48 turns. This means that approximately half of the games each player plays 12 turns, but in the other half the first player gets one more turn.

- Exchanges occur 1.9 times in a game. Each player exchanges once each game.

After observing this behavior we wondered what we could do to make a more even game. Then we thought we could change the initial score. Instead of starting the game 0 vs 0 we could give an edge to player B by starting 0 vs 1, 0 vs 2 , etc… Then we could recalculate the stats and watched what would be the handicap for B in order to achieve a closer game. When starting the game with a score of 0 vs 0 the stats were: Player A won 55.00%, tied 0.29 % and Player B won 44.71% (see Table 9.1). It turned out that giving a 16 point lead to player B achieves the closest game (see the almost equal length of the bars in Figure 9.1). The closest margin between victories of players A and B was achieved with a 16 point handicap. Player A won 51,944 games tied 350 and Player B won 51,741 (see Figure 9.1). In other words Player A won 49.93% games, tied 0.34% and Player B won 49.73% (see Table 9.1).

Table 9.1. Handicap

Handicap	A win %	ties %	B win %
0	55.0017	0.2912	44.7071
1	54.7047	0.297	44.9983
2	54.3625	0.3422	45.2953
3	54.0222	0.3403	45.6375
4	53.7098	0.3124	45.9778
5	53.3926	0.3172	46.2902
6	53.0927	0.2999	46.6074
7	52.7621	0.3307	46.9073
8	52.4208	0.3412	47.2379
9	52.0777	0.3432	47.5792
10	51.7566	0.321	47.9223
11	51.4529	0.3037	48.2434
12	51.1462	0.3066	48.5471
13	50.854	0.2922	48.8538
14	50.5551	0.2989	49.146
15	50.2658	0.2893	49.4449
16	49.9294	0.3364	49.7342
17	49.6246	0.3047	50.0706

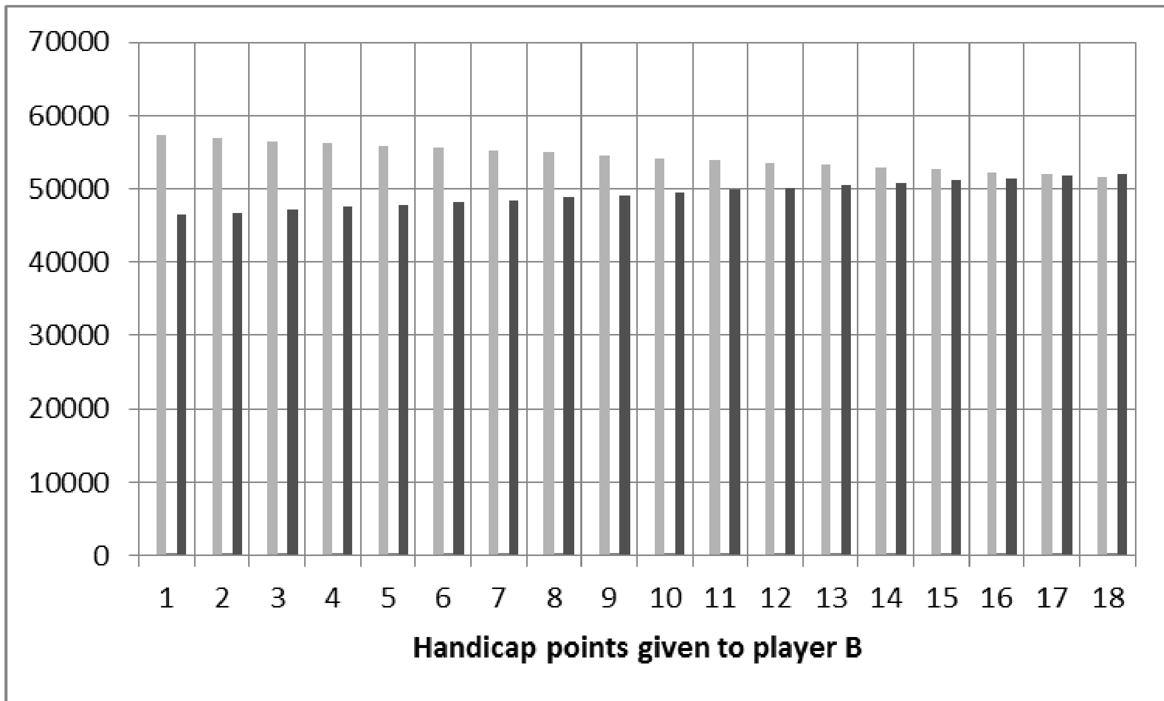


Fig. 9.1. Handicap points given to player B

Regarding the number of Bingos made by both players we got the following results:

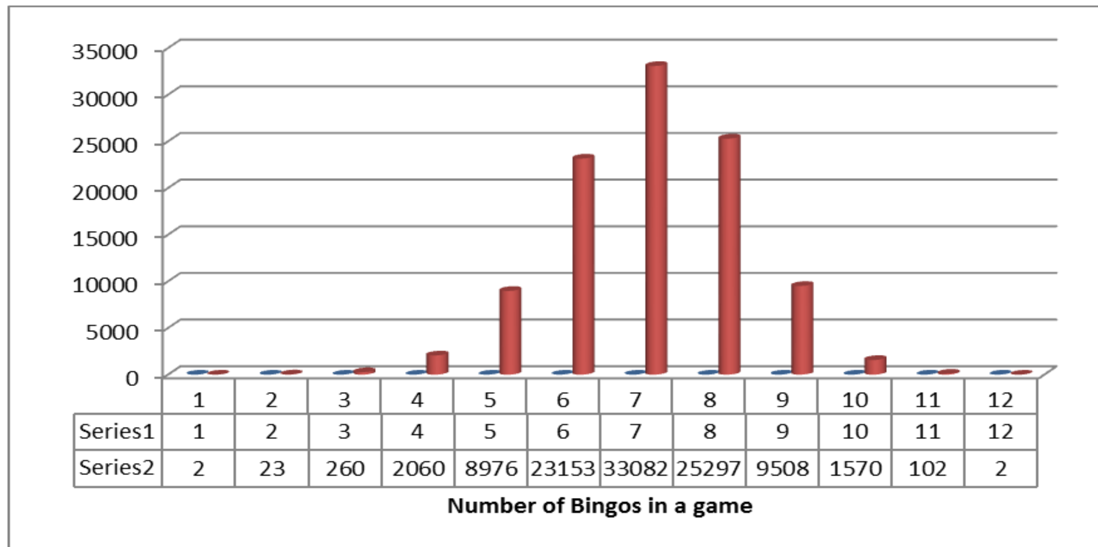


Fig. 9.2 Number of games with n Bingos

In Figure 9.2 the number of Bingos played in a game is shown. Series 1 indicates the number of Bingos in a game, while Series 2 indicates how many games from the total pool of 104,035 games contain exactly *n* Bingos, (1<= *n* <=12). From Figure 9.2 we can see that the number of games having exactly *n* Bingos is maximal when *n*=7 (there are 33,082 games with 7 Bingos).

9.1.2 Word Statistics

We saved the words in each of the 104,035 games played. Unlike humans, Heuri does not struggle in remembering all the valid words in the lexicon. A natural way to help humans with words is to make a list of the most important words. Bingos are important since a 50 bonus point is awarded to a player who uses all the 7-tiles in the rack. Then the most frequently played bingos were collected and ordered from highest to lowest frequency. The following tables present the 200 most frequent Bingos.

Table 9.2. Word Frequency and Bingo Frequency

Word	Frequency	Bingo Frequency	Word	Frequency	Bingo Frequency
1. OSEARIA	167	156	51. AQUIETO	75	46
2. ASAETEO	144	138	52. ALOTAIS	75	69
3. ALETEADO	127	110	53. ALETEOS	75	63
4. SAETEADO	121	110	54. AIREASE	75	75
5. ECUOREAS	121	86	55. AIREAIS	75	75
6. TEODICEA	110	110	56. ADENOSO	75	69
7. LESEADO	110	110	57. TUNEAIS	69	69
8. OLEACEA	104	81	58. TRONCAD	69	52
9. ESTURADO	104	104	59. TALONEA	69	69
10. AZOQUES	98	6	60. TARRECER	69	52
11. ANEROIDE	98	98	61. RESUENA	69	69
12. ONDEASE	92	92	62. OSEANDO	69	63
13. NAUSEOSO	92	92	63. ONDEARE	69	69
14. AUDITEN	92	69	64. OLEARIA	69	69

15. ASAETADO	92	63	65. OLEACEO	69	52
16. OLEACEAS	86	69	66. NAUSEADO	69	63
17. LANDRES	86	86	67. NAIPEADO	69	69
18. ECUOREA	86	86	68. LUNEADO	69	63
19. DOCILES	86	86	69. LETRUDA	69	63
20. AUREOLA	86	75	70. IDEARIO	69	58
21. ATIESADO	86	75	71. IDEARIAN	69	58
22. ADONIOS	86	86	72. HULEADO	69	52
23. ACODASE	86	86	73. EXIGUOS	69	6
24. SAETADO	81	81	74. ENCOGED	69	35
25. PEDACEO	81	63	75. EDITORA	69	63
26. EUBOLIAS	81	52	76. ECUOREOS	69	46
27. EUBOLIA	81	52	77. DUENARIO	69	69
28. DESOIGA	81	81	78. DESEARA	69	69
29. AUREOLE	81	75	79. DESAHITE	69	69
30. AROIDEO	81	75	80. CURIOSEA	69	69
31. ARENADO	81	75	81. CUEREADO	69	69
32. AQUELLOS	81	12	82. CONDUTA	69	52
33. ALIACEOS	81	46	83. COLOIDEA	69	69
34. ALIACEO	81	63	84. CODEASE	69	69
35. SACONEA	75	75	85. CAITEADO	69	69
36. RODEASE	75	63	86. ALLEGADO	69	3
37. OTEARIA	75	69	87. AUSONIA	69	63
38. OLEARIO	75	69	88. AUSETANO	69	69
39. NOQUEADO	75	52	89. ATUENDOS	69	52
40. LOLEARIA	75	63	90. ATARACEO	69	63
41. EROSIONA	75	69	91. ASOLEARE	69	69
42. ENDOSELO	75	75	92. ASEDIADO	69	58
43. CUITEADO	75	69	93. ASAETEE	69	58
44. CODEEIS	75	63	94. HARINEA	69	69
45. COALIGUE	75	75	95. ALOTASE	69	69
46. AUREOLEN	75	75	96. ALISEDA	69	63
47. AUREOLAS	75	69	97. ALEONASE	69	69
48. ASOLEADO	75	58	98. ALEONADO	69	58
49. ASEADORA	75	63	99. AIREASEN	69	35
50. AROIDEA	75	69	100. AIREADO	69	69

Word	Frequency	Bingo Frequency	Word	Frequency	Bingo Frequency
101. AHELEADO	69	63	151. ACIANOS	63	63
102. ADUANERO	69	69	152. ACEITOSA	63	63
103. ACUNAIS	69	69	153. ARROGUES	63	29
104. ACUERNO	69	69	154. ALEUDADO	58	6
105. XECUDOS	63	6	155. USUREADO	58	58
106. TENDRAS	63	58	156. TRONIDOS	58	52

107. TELENDOS	63	52	157. TRESNAL	58	52
108. TEHUANOS	63	40	158. TOLETEAD	58	58
109. SOTANEO	63	63	159. TOASEIS	58	58
110. SAINETEO	63	58	160. TERREADO	58	29
111. RESUENO	63	40	161. TANDEMS	58	46
112. REANUDO	63	63	162. SUELDEN	58	52
113. PEDANEOS	63	35	163. SOLIDAR	58	58
114. OTEARIAN	63	58	164. SOGUEADO	58	58
115. MATEEIS	63	63	165. SILUETA	58	58
116. LEUDANTE	63	63	166. SACONEO	58	58
117. LETUARIO	63	63	167. ROSACEA	58	58
118. LAURINEO	63	63	168. RECAIDO	58	58
119. LAUREOS	63	63	169. POSEERIA	58	58
120. HORDIATE	63	63	170. OTEADORA	58	40
121. HALOQUE	63	17	171. OPIACEOS	58	46
122. GEOIDES	63	23	172. ONEROSA	58	58
123. EUBOICA	63	58	173. ONDULARE	58	58
124. ESTURADO	63	58	174. ONCEASE	58	58
125. ENACEITA	63	63	175. ONCEADO	58	58
126. ELUDIAN	63	46	176. NEORACEO	58	52
127. ELEATICO	63	63	177. LOQUEAS	58	35
128. ELATERIO	63	58	178. LEÑADORA	58	17
129. EDETANA	63	63	179. LEÑADOR	58	17
130. DIOSTEDE	63	58	180. IDEARIA	58	58
131. CUOTEASE	63	46	181. HEBETADO	58	52
132. CALENTAD	63	46	182. GRANDES	58	29
133. BRESCAD	63	46	183. GOTEANDO	58	52
134. ATESORO	63	63	184. EXERGOS	58	12
135. ATESADO	63	63	185. CUSIERA	58	52
136. ATAREEN	63	58	186. EUROPEAS	58	46
137. ASESORO	63	63	187. EQUIDAD	58	6
138. ASEARIA	63	63	188. ENTIESO	58	58
139. AROIDEOS	63	35	189. ENEALES	58	58
140. ANELIDO	63	63	190. ENACIADO	58	58
141. ANADEEIS	63	63	191. ELISANA	58	58
142. ALIENTO	63	63	192. DUODENA	58	52
143. ALEUDADO	63	52	193. DIPETALA	58	58
144. ALEONEIS	63	63	194. DESTRAL	58	58
145. ALEONARE	63	63	195. DELICADA	58	58
146. ALANCEE	63	63	196. DECATLON	58	58
147. AHULADO	63	35	197. COQUETAS	58	52
148. ADOLECIA	63	63	198. CONSTAD	58	46
149. ADICIONE	63	63	199. CLOTEADO	58	29
150. ACUSONA	63	58	200. ESCOÑADO	58	12

Other words of lengths 6,5,4,3 and 2 which are not bingos, but occur frequently and therefore are useful when playing are given in the following table. For a Scrabble tournament player it is essential to know all 2-letter words and it is very convenient to know all the 3-letter words.

Table 9.3. Frequent words

acoged	haced	quid	tex	oc
azoque	hoque	oxeo	oxe	ad
herrado	heded	oxea	ahe	ex
acuñen	coged	oxee	elle	ax
hoques	xecas	halle	box	ox
equeco	quedo	xeca	dad	es
exhalo	cuello	oque	oca	en
heñido	eubeo	hede	iza	ar
aquello	herrad	duho	dux	eh
axiote	aqueo	diez	fax	os
herrada	acueo	huao	zar	oh
ocurres	corred	sexy	aje	lle
teñido	hertz	dond	jet	id
adeudo	querre	holle	año	ce
ceñido	codez	luxe	ñor	et
equino	herren	años	ahi	as
acorred	yerros	hoye	del	el
achoque	helad	luxo	ajo	ca
allegad	ceñid	hove	dix	do
cañedo	horrad	quio	bah	de
ocurren	cerril	yerro	aho	za
boxead	hocen	eneo	ños	ah
cloque	hotel	huye	ñas	un
colgad	llegad	haza	hay	al
gemido	yaque	hallo	hall	ño
jiñado	hallen	hiño	hez	da
ñaques	meced	xolo	cid	he
amoved	cerrad	hube	uño	re
currado	queco	trox	hin	uh
garrido	xolos	hayo	zas	ña
quedos	ceñen	hurra	her	aj
zooide	herron	hoce	ñus	ro
ahueco	index	hozo	aja	ir
ceñida	telex	lady	aña	ge
eduque	zolles	ñora	aji	ea
exiguo	debut	queo	huy	ha
heñida	exudo	hupe	ido	be

hierros	hurras	hice	den	se
hipado	oxead	hogo	que	ni
hiñese	xolas	exir	dey	oa
jaldos	ñecla	ohms	ene	pe
oxeada	eolio	bloc	gel	lo
curalle	quilo	error	tell	le
ñoclos	hollad	abey	ayo	xi
ahoyen	hundo	herre	oye	ne
apurrir	quede	selle	cay	to
calque	xiote	apex	has	so
cepazo	ñochas	aque	ele	me
equido	exude	dello	ved	ay
geoide	hedio	hedi	lux	cu

9.2 Match Results

9.2.1 Match Results of Heuri's First Engine

Due to the unfinished engine we could only perform a few games to test our first heuristic; let this heuristic be called ***Heuri***. The first match Heuri played, was a match against an engine who always played the highest move, Heuri won 4-0. Matches against humans were performed by internet, this gave the humans an extra force since they could consult the words before they actually put them on the board, they also had a lot of time to play. In its first match against a human Heuri won 4-2 against one of the best players of Colombia, Fernando Manríquez with an ELO 2110 (the highest ELO in the FISE list was 2191), ranked No.20 in the FISE list of Spanish Scrabble players. Another important game victory of Heuri was against Benjamín Olaizola, from Venezuela, who was the current (2007) world champion!, he also won the world championship in 2001, his ELO was 2169, he occupied the 2nd place in the international ranking list. It was a very close game! (Heuri 471 Benjamín Olaizola 465). Eventually Heuri lost the match against Benjamín Olaizola 2-4. After playing 17 games against several top class players Heuri results were: Heuri 10 wins and 7 loses, average points per game: 509 pts.

9.2.2 Match Results of Heuri's Second Engine

After playing 40 games against several top class players Heuri results using the heuristic described in section 7.4 were: Heuri 27 wins and 13 loses, average per game: 511 pts. Matches against humans were performed by internet, this gave the humans an extra force since they could consult the words before they actually put them on the board, they also had a lot of time to play. Almost all matches followed the format of a baseball world-series, the first opponent arriving at 4 victories won the match. The most important match victory of Heuri was against the, then current, 2008 World Champion Enric Hernández, from Spain, also World Champion in 2006. This was a longer match, it consisted in playing a maximum of eleven games to decide the winner. Heuri defeated the World Champion 6-0! An excellent result! Heuri also won 4-1 a match against Airan Pérez, from Venezuela, one of the best players in the world, 2007 and 2008 Runner-up. He has won two World Championships (2013 and 2015). Heuri also defeated 4-1 Aglaia Constantin, at that time the current best player of Colombia, twice National Champion of Colombia (2006 and 2009). Heuri won four matches and lost one, the rest of the games were incomplete matches against past ex world champions and national champions.

9.2.3 Match Results of Heuri's engines (Third vs Second, and Fourth vs Third Heuri's engines)

To verify that Heuri was improving, when constructing the new engines, we performed two matches, each match consisting in 10,000 games. The first match confronted Heuri's third engine against Heuri's second engine; and the second match was Heuri's fourth versus Heuri's third engines. These matches indicated the improvement of Heuri's third engine over the second engine, and showed how Heuri's fourth engine surpassed the third engine. More details are given in the next tables 9.4 and 9.5.

Table 9.4. Matches in Spanish between Heuri's third and second engines

Number of games	Third engine games won	Second engine games won	Confidence interval
10000	5721	4279	(0.5624,0.5818)

Table 9.5. Matches in Spanish between Heuri's fourth and third engines

Number of games	Fourth engine games won	Third engine games won	Confidence interval
10000	5534	4466	(0.5437,0.5631)

In table 9.4, the result of games won by Heuri third engine is statistically significant: with 95% confidence, the true value lies in the confidence interval (0.5624, 0.5818), that is, if the experiment is repeated many times the resulting proportion of Heuri wins is/lies in this interval with probability 0.95% . Likewise, in table 9.5, the result of games won by Heuri fourth engine is statistically significant: with 95% confidence, the true value lies in the confidence interval (0.5437, 0.5631).

9.2.4 Match Results of Heuri's Third and fourth Engines vs Quackle (Speedy Player and 20sec Ch. Player)

Using Quackle open source code and Heuri, connection was achieved to play Spanish Scrabble games between the two engines. Quackle is not as strong in Spanish as it is in English. Now we present the results, see Tables 9.6 and 9.7, from two, 10000-game, matches between Heuri's third and fourth engines and Quackle (Speedy player). The Speedy Player evaluates potential moves without any active forward looking, calculating only the short-term utility of a move: the value of the played word plus a pre-computed leave value, or the estimated value of the remaining tiles in combination with each other, plus a small adjustment for the number and quality of locations that are now accessible to the opponent [Thomas11]. The Speedy Player is weaker in Spanish since there are no pre-computed leave values for Spanish.

Table 9.6. Matches in Spanish between Heuri's third engine (H_3) versus Quackle (Speedy Player)

Number of games	H_3 engine games won	Quackle games won	Confidence interval
10000	5987	4013	(0.5891,0.6083)

Table 9.7. Matches in Spanish between Heuri's fourth engine (H_4) versus Quackle (Speedy Player)

Number of games	H_4 engine games won	Third engine games won	Confidence interval
10000	6134	3866	(0.6039, 0.6229)

In table 9.6, the result of games won by Heuri's third engine against Quackle (Speedy Player) is statistically significant: with 95% confidence, the true value lies in the confidence interval (0.5891, 0.6083). Likewise, in table 9.7, the result of games won by Heuri's fourth engine is statistically significant: with 95% confidence, the true value lies in the confidence interval (0.6039, 0.6229).

Heuri's third Engine (Heuri_3) also played a 1014-game match against Quackle using 20 Sec. Champ Player (see Table 9.8). This is a much stronger bot, since it uses 20 seconds simulation for every move. The results indicate that Heuri wins 53.35% of the games. A *turnover* is a game, where one player is losing when the endgame begins, but wins after the endgame ends; (the endgame starts when there are no tiles inside the bag, and ends when a player has no more tiles in her/his rack or no move can be made, and the bag is empty). Thus, Table 9.8 shows that out of the 507 games in which Heuri started, 285 were won by Heuri and in 47 of these 285 Heuri was behind when the endgame started and ended up winning after the endgame finish. Table 9.8 also shows that out of the 507 games in which Quackle started, 251 were won by Quackle and in 54 of these 251 Quackle was behind when the endgame started and ended up winning after the endgame finish. Turnovers indicate an estimation of the quality of play in the endgame. The first player always has a big edge, and as can be seen from the table, Heuri plays better in Spanish than Quackle; nevertheless Quackle plays better the endgames.

Table 9.8. Matches in Spanish Heuri_3 vs Quackle, Quackle vs Heuri_3 (Quackle uses 20 Sec Champ Player)

1014 Games 507 Seeds	Heuri_3	Quackle	Quackle	Heuri_3
Wins	285	222	251	256
Turnovers	47	12	54	5

Heuri_3 result against Quackle (20 Seconds Champ Player) is significant statistically: with 95% confidence, the true value lies in the confidence interval (0.5028, 0.5642).

Heuri's fourth engine (Heuri_4) also played a 1014-game match against Quackle (20s Ch Player), Heuri won 56.71% of the games (Heuri_4 won 575 games) . This is significant statistically: with 95% confidence, the true value lies in the confidence interval (0.5574, 0.5768).

9.2.5 Heuri plays better in Romance languagues

Using Heuri's fourth heuristic, the Scrabble engine Heuri was set up to play in French; then matches in French between Heuri and Quackle were played. The main reason to perform these experiments is to prove the following belief: "Due to Heuri's bingo-oriented strategy, Heuri plays better in Romance languagues like Spanish and French than in English".

Let us explain: being Spanish and French Romance languagues they have a high verb conjugation, in contrast to non-Romance languages like English. In other words, a conjugation of a verb originates many words. For example the verb amar originates nearly 50 different words, in contrast to the verb love which originates only 4 words. Most of the 7-letter and 8-letter words come from verbs. The most common bingos are 7-letter and 8-letter words. Recall that a *bingo* is a move in Scrabble which uses all the tiles of a player's rack, and most importantly, gives a 50 bonus points reward; that is why making bingos is very important to win a game.

Due to the verb nature of romance languagues, games played in those languages tend to have more 7-letter and 8-letter words than those played in non-Romance languages. Besides, most of the verbs in Romance languages contain highly frequent letters in the Scrabble game; therefore the 7 and 8-letter words that come from verbs are likely to appear in Scrabble games. For example the must frequent bingos in Spanish are: OSEARIA, ASAETEO, ALETEADO, which are words that come from the verbs OSEAR, ASAETEAR and ALETEAR.

The results of matches between Heuri and Quackle with greedy approach are presented in Table 9.9 below. As can be seen from the following two tables (Tables 9.9 and 9.10) Heuri plays stronger in Spanish and French than in English. Performance of Heuri vs Greedy Opponent ,Quackle (using a Greedy variant), after playing 10,000 games in each of 3 different languages (English, French and Spanish). See Table 9.9.

Table 9.9. Matches Heuri_4 vs Quackle Greedy

Heuri_4 vs Quackle Greedy	***Spanish***	***French***	***English***
Heuri_4 winning percentage	61.82%	61.11%	60.17%

The results of Heuri_4 against Quackle Greedy in Spanish, French and English are all significant statistically: with 95% confidence, the true values lay respectively in the intervals (0.6087, 0.6277), (0.6015, 0.6207) and (0.5921, 0.6113).

Let us recall some characteristics of the engine Quackle Speedy. Quackle Speedy Player is a Scrabble engine that does not use simulation, but it does use *Quackle's Superleaves* when playing in English and French. *Quackle's Superleaves* are numerical values corresponding to different multisets of tiles (with cardinality between 1 and 6). These Superleaves are precomputed values, they were calculated by playing hundreds of thousands of Quackle vs Quackle games. They were precomputed in English and French, but there are no Superleave values in Spanish.

Quackle Speedy Player outperforms Heuri_4 in English and French thanks to the Superleaves, but Heuri wins in Spanish against Quackle Speedy Player because of the lack of Superleaves in Spanish. See next table (Table 9.10). Performance of Heuri vs Quackle Speedy Player after playing 30,000 games (3 matches in 3 different languages consisting of 10,000 games each match).

Table 9.10. Matches in Spanish Heuri_4 vs Quackle Speedy Player

Heuri_4 vs Quackle Speedy Player	***Spanish***	***French***	***English***
Heuri_4 winning percentage	61.97%	47.35%	45.58%

The results of Heuri_4 against Quackle Speedy in Spanish, French and English are all significant statistically: with 95% confidence, the true values lay respectively in the intervals (0.6102, 0.6292), (0.4637, 0.4833) and (0.446, 0.4656).

9.2.6 Heuri produces and validates Scrabble expert Strategy in the opening

The following heuristic function is used in Heuri to evaluate all potential moves:

$$v = j + e - d$$

where j is the number of points made by the move, e is the expected value of a bingo in the next turn and d is a non-negative number used for a defensive purpose (In **Heuri** $d = 0$ and in **HeuriSamp** $d > 0$).

The calculation of e is given by the following formula:

$$e = \sum_i^{totseptroct} \delta_i \, p_i \, (50 + k\sigma_i) \quad \text{(where } i \text{ runs from 1 to } \textbf{totseptroct}\text{)}$$

where :

σ_i is the sum of the letters of the i-th septet.
k is a constant (usually 2.5) to account for board rewards
p_i is the probability to form the i-th septet.
δ_i equals 1 if the i-th septet can be placed on the board , otherwise $\delta_i = 0$.
totseptroct is the total number of strings which are either septets or reduced octets.

Let us refer to k as FactorAtril .

From many experimentation matches between Heuri vs Quackle Speedy and Heuri vs (Heuri/HeuriSamp) we discovered that the best results came when FactorAtril = 5 if the board is empty. When seeing the beginnings of these games, where FactorAtril = 5, we observed that this resulted in the following strategy: When playing the first move on the board try not to increase the opponent's chances to put a Bingo or a valuable play; this happens when playing on the board good tiles near hot spots (this is known as opening the board). Only open the board if your play makes many points (more than 30) or your leave (the rack residue after your play) is good; if none of these options is possible it is preferable to change rather than opening the board to the opponent without any gain. This is a strategy that expert Scrabble players are familiar with.

9.2.7 Heuri improves when FactorAtril is a non-constant value

Another improvement of Heuri's game playing came when we experimented with a non-constant value for FactorAtril. We used the total number of $\boldsymbol{\delta_i}$ (known as deltas) in a certain position; this number indicates the degree of openness of a position.

Then we calculated FactorAtril in such a way that it is directly proportional to the total number of deltas equal to one, of the position in play. The idea here is to have a greater (resp. smaller) value of FactorAtril corresponding to a more (resp. less) open position of the board in play.

Let us call Heuri_5 the engine obtained adding to the engine Heuri_4 the characteristic of FactorAtril non-constant making it directly proportional to the total number of deltas equal to one, starting with FactorAtril=5 when the board is empty. To check if Heuri_5 improves its play, we decided to performe six 10000-game matches: the first three matches are between Heuri_5 and Heuri_4, playing in Spanish, French and English, the other three matches are between Heuri_5 and Quackle (Speedy Player); confidence intervals are also given.

Table 9.11. Matches in Spanish, French and English. Heuri_5 vs Heuri_4

Heuri_5 vs Heuri_4	*Spanish*	*French*	*English*
Heuri_5 winning percentage	53.52%	52.58%	51.69%
95% Confidence interval	(0.5254, 0.545)	(0.516, 0.5356)	(0.5071, 0.5267)
99.9% Confidence interval	(0.5188, 0.5516)	(0.5094, 0.5422)	(0.5005, 0.5333)

Table 9.12. Matches in Spanish, French and English. Heuri_5 vs Quackle Speedy Player

Heuri_5 vs Quackle Speedy Player	*Spanish*	*French*	*English*
Heuri_5 winning percentage	64.22%	49.58%	47.78%
95% Confidence interval	(0.6328, 0.6516)	(0.486, 0.5056)	(0.468, 0.4876)
99.9% Confidence interval	(0.6264, 0.658)	(0.4794, 0.5122)	(0.4614, 0.4942)

9.2.8 Heuri version adapted to the prefinal phase

The following heuristic function is used in Heuri to evaluate all potential moves:

$$v = j + e - d \qquad (9.1)$$

where j is the number of points made by the move, e is the expected value of a bingo in the next turn and d is a non-negative number used for a defensive purpose (In **Heuri** $d = 0$ and in **HeuriSamp** $d > 0$).

The calculation of e is given by the following formula:

$$e = \sum_i^{totseptroct} \delta_i\, p_i\, (50 + k\sigma_i) \quad \text{(where } i \text{ runs from 1 to } \textbf{totseptroct}\text{)} \qquad (9.2)$$

where :

σ_i is the sum of the letters of the i-th septet.
k (also known as FactorAtril) is 5 if the board is empty, else it is directly proportional to the total number of deltas equal to one ($\delta_i = 1$); to account for board rewards
p_i is the probability to form the i-th septet.
δ_i equals 1 if the i-th septet can be placed on the board , otherwise $\delta_i = 0$. (except at the Prefinal phase (see, Prefinal phase, below)
totseptroct is the total number of strings which are either septets or reduced octets.

Let us refer to k as FactorAtril.

In chess it is known that some moves just before reaching the endgame are important to enter a favorable endgame. This also happens in Scrabble; we called this phase ***Prefinal*** and the best results were obtained when Prefinal was defined as the phase where $0 < \#bag < 15$ (where ***#bag*** is the cardinality of the bag in play). **The strategy used to play in the Prefinal phase was: if** $0 < \#bag < 15$ then use the heuristics given in the above formulas (9.1) and (9.**2**) with the following changes in (9.2) $\delta_i = 1$ and $k = 0.5$. Let us call it Heuri_6.

Table 9.13. Matches in Spanish, French and English. Heuri_6 vs Heuri_5

Heuri_6 vs Heuri_5	***Spanish***	***French***	***English***
Heuri_6 winning percentage	52.95%	52.18%	51.73%
95% Confidence interval	(0.5197, 0.5393)	(0.512, 0.5316)	(0.5075, 0.5271)
99.9% Confidence interval	(0.5131, 0.5459)	(0.5054, 5382)	(0.5009, 0.5337)

Table 9.14. Matches in Spanish, French and English. Heuri_6 vs Quackle Speedy Player

Heuri_6 vs Quackle Speedy Player	***Spanish***	***French***	***English***
Heuri_6 winning percentage	65.93%	51.71%	49.54%
95% Confidence interval	(0.65, 0.6686)	(0.5073, 0.5269)	(0.4856, 0.5052)
99.9% Confidence interval	(0.6437, 0.6749)	(0.5007, 0.5335)	(0.479, 0.5118)

9.2.9 Match Results of HeuriSamp Engine

Stability to run on the SuperComputer El Insurgente was achieved in HeuriSamp, allowing to perform two 10,000 game matches in English; Heuri_6Samp vs Heuri_6 and Heuri_6Samp vs Quackle SP, obtaining the following results:

Table 9.15. Matches in English Heuri_6Samp vs Quackle SP=Speedy Player, and Heuri_6Samp vs Heuri_6

HeuriSamp versus Heuri, and HeuriSamp versus Quackle Speedy Player	*Quackle Speedy Player*	*Heuri_6*
HeuriSamp winning percentage	48.73%	49.14%

Recall that when playing Heuri_6 vs Quackle Speedy Player in English, Heuri_6 winning percentage was 49.54% See Table 9.14. We believe that Heuri_6Samp plays weaker than Heuri_6 because it is not convenient to play always in a defensive manner.

After careful study of some games, it was concluded that Heuri_6Samp many times, unnecessarily, blocks a spot on the board, which is not used by the rival, but it could have been used by Heuri_6Samp, in other words, Heuri_6Samp many times self-blocks its own moves. To improve the performance of Heuri_6Samp, opponent modeling was incorporated into Heuri_6Samp, see Tables 9.16 and 9.17. Opponent modeling helped Heuri_6Samp in deciding when it is convenient to play defensively and when it is not.

Table 9.16. Matches in English HeuriSamp OPM vs (Quackle SP=Speedy Player, HeuriSamp and Heuri)

HeuriSamp OPM vs (Quackle SP, HeuriSamp and Heuri)	Quackle Speedy Player	HeuriSamp	Heuri
HeuriSamp OPM win %	50.68%	52.32%	51.12%
95% Confidence interval	(0.5,0.5196)	(0.5134,0.533)	(0.5014,0.521)

Table 9.17. Matches in Spanish, French and English. Heuri_6Samp with OPM vs Quackle Speedy Player

HeuriSamp OPM vs Quackle Speedy Player	*Spanish*	*French*	*English*
HeuriSamp OPM winning percentage	67.12%	52.84%	50.71%
95% Confidence interval	(0.662, 0.6804)	(0.5186, 0.5382)	(0.4973, 0.517)
99.9% Confidence interval	(0.6557, 0.6867)	(0.512, 0.5448)	(0.4907, 0.5235)

9.2.10. Match Results of HeuriSim Engine

Another improvement in Heuri's favour comes from the *HeuriSim* engine, this engine contemplates the first move of Player 1, replied by the first move of Player 2, replied by the second move of Player 1 (let us call this a 3-ply search); see Tables 9.18 and 9.19. HeuriSim avoids self blocking moves thanks to the 3 plies lookahead. The best results come when HeuriSim incorporates opponent modeling (see Table 9.20).

Table 9.18. Matches in Engish Heuri_6Sim vs (Quackle SP=Speedy Player, HeuriSamp and Heuri)

Heuri_6Sim vs (Quackle SP, Heuri_6Samp and Heuri_6)	Quackle Speedy Player	Heuri_6Samp	Heuri_6
Heuri_6Sim winning %	52.04%	55.23%	54.62%

Table 9.19. Matches in Spanish, French and English. Heuri_6Sim vs Quackle Speedy Player

Heuri_6Sim vs Quackle Speedy Player	***Spanish***	***French***	***English***
Heuri_6Sim winning percentage	68.63%	54.28%	52.17%
95% Confidence interval	(0.6772, 0.6954)	(0.5526, 0.533)	(0.5119, 0.5315)
99.9% Confidence interval	(0.671, 0.7016)	(0.5264, 0.5592)	(0.5053, 0.5381)

Table 9.20. Matches in Spanish, French and English. Heuri_6Sim with OPM vs Quackle Speedy Player

Heuri_6Simwith OPM vs Quackle Speedy Player	***Spanish***	***French***	***English***
Heuri_6Sim with OPM winning percentage	69.87%	55.45%	53.69%
95% Confidence interval	(0.6897, 0.7077)	(0.5448, 0.5642)	(0.5271, 0.5467)
99.9% Confidence interval	(0.6836, 0.7138)	(0.5284, 0.5612)	(0.5205, 0.5533)

The best results were achieved using Heuri_6Sim with OPM to make an educated guess about the opponent's tiles, and thus deciding when is worth to defend, blocking a spot on the board that could be used to make many points by the opponent, and when is better to leave it open so Heuri can possibly use it in the next turn.

Chapter 10.

Conclusions and Future Work

In Spanish Scrabble, the first player has an advantage of approximately 16 points. This is due to the fact that, statistically, the first player has one more turn in half of the games played. If a more even game is wanted we could start the game with a score of 0 vs 16 (a 16 point lead to player B). Another way to try to balance things is to make a rule forcing each player to have the same amount of turns in a game. Stats could be recalculated to see how this works.

Many humans have a tendency to change tiles too much in a game. As indicated by the 1.9 exchanges per game found and knowing that Heuri has had excellent results, humans should try to better look for non-exchanging plays. One might argue that this is due to the fact that Heuri knows all the words in the lexicon. But many times, a deficiency in human search is the reason to fail to find words that we do know. However, experiments for measuring Heuri’s strategy with a limited lexicon are desirable.

There are 3 factors that influence in the outcome of a game: 1) Chance, 2) Lexicon knowledge and 3) Strategic play. It is an open question to determine how much weight should be given to each of these factors. Some experts think that each one of them weights 1/3.

Besides performing experiments with different lexicons to try to determine its importance in the outcome of a game, we should also try to somehow develop experiments that could also measure how much chance and strategic play affect the game. For instance playing with different strategies and giving handicap to player B to minimize chance. It is also possible to give specific tiles to each player in such a way that every player has similar chances of making good moves.

The frequency word lists obtained are very helpful for humans in learning the appropriate sub-lexicon, but they are too long and it is difficult to memorize so many words. One way to help humans would be to have group of words instead of single words. We could group many words if they look alike. A distance function could be defined between two words and we could group words with small distance, then we could identify words which are inflections of the same verb, and only put the verb in infinitive, or if we have singular and plural of a word, just put the singular word.

Heuri’s move generator has an excellent performance, thanks to the wise and adequate Anagram methodology employed in building the Computer lexicon, and to the ingenious pre-calculations of multiple information of the cells of the board; proof of this is the comparative results against other move generators shown on section 6.4.

An important thing to mention is that Heuri allows to play in many different languages; Heuri only needs the lexicon, the letter distribution with the letter values and the rules to end the game. Then in only 30 minutes or less, the Computer Lexicon is constructed and Heuri is ready to play with full strength. Quackle needs to somehow pre-calculate its Superleaves to play at full strength; we do not know how long they take to calculate them, but it seems to be complicated, since despite playing in many languages, we have only seen Superleaves data for English lexicons.

Heuri's probabilistic heuristic turned out to be of championship caliber. It managed to defeat several Spanish Scrabble world Champions, without any forward looking. Then the strongest opponent came, Quackle. In Spanish Heuri defeated Quackle because of Quackle's lack of Superleaves in Spanish. Quackle's strength relies

in the Superleave values; these are pre-calculated values of every possible rack, with cardinalities 1 to 6, obtained by playing thousands of Quackle vs Quackle games. Heuri could not defeat Quackle in English when not using any forward lookahead. To play at Quackle's level, Heuri had to incorporate a 3-ply guided search using Opponent Modeling to make educated guesses about Quackle's tiles. Opponent Modeling along with the 3-ply guided search was the key. In the future to improve Heuri, we should try to make the heuristic not only bingo oriented, but also contemplating the probability of big scoring moves that are not bingos; also performing a 4-ply guided search using Opponent Modeling would be very convenient.

In order to play at championship level, humans need to memorize many words. But the vocabulary is huge, so it is desirable to know which words are more important to know. Our Scrabble engine Heuri could be very useful, since after playing thousands of games these words are discovered by their higher frecuency. A list of some of these words is given in Chapter 9.

In the process of building a vocabulary for Heuri in Spanish (The Spanish Lexicon), it was found that it is convenient to classify irregular verbs as forming 38 disjoint families (the most numerous is the family “agradecer” consisting of 260 verbs). This could be useful in teaching Spanish as a foreign language, or it can be useful for Spanish computational linguistics.

Trap Moves and Sacrifice Moves

In the paper “On adversarial search spaces and sampling-based planning” by Ramanujan et al. 2010 [Rama10], it was found that *trap moves* are very common in Chess games at various levels of play. *Trap moves* are moves which appear to be good, but are in fact bad. Monte-Carlo Tree Search (MCTS) valued these moves almost as highly as the strongest move available, indicating that it was unable to detect the trap.

A *sacrifice move* is the converse of a trap move; a s*acrifice move* is the deliberate loss of some resource, such as a piece in chess, in order to create a situation where no matter how the opponent plays, the player making the sacrifice can guarantee a good outcome. In other words, a sacrifice move is a move which looks bad, but is good [NatAld16].

In Scrabble trap moves and sacrifice moves appear frequently. Sacrifice moves are similar to Fishing moves in Scrabble, see [GonzalezAlquezar12], these moves sacrifice doing few points on the board, in order to make a big scoring move in the next move, sacrifice moves are detected by Heuri's heuristic $v = j + e - d$, where $d = 0$, see [GonzalezAlquezar12]. Nevertheless, sometimes this heuristic may fall into trap moves, originating moves that seem good, but are bad. Fortunately trap moves are detected using the combination of Heuri's heuristic, Opponent Modelling (OPM) and looking ahead 3 plies; in other words, the HeuriSim with OPM engine detects these trap moves. We decided not to use MCTS in Heuri, due to the good results obtained by the HeuriSim with OPM engine, and from the study of Ramanujan et al. 2010 in the paper “On adversarial search spaces and sampling-based planning” [Rama10], which indicates that MCTS was unable to detect the trap moves.

10.1 Contributions of this thesis

1) A novel Scrabble move generator based on anagrams has been designed and implemented, which has been shown to be faster than the GADDAG-based generator used in Quackle engine.

2) A novel Computer Scrabble engine called Heuri (actually a family of engines) has been designed and implemented, which has been tested to perform at championship level in different languages (Spanish, French and English).

3) Several heuristic evaluation functions of increasing playing performance, all of them based on probabilities, have been proposed and tested, which allow Heuri to beat (top-level) expert human players without the need of using sampling and simulation techniques.

4) Some extensions to the basic Heuri engine have been implemented that incorporate sampling, simulation and opponent modeling techniques, leading to the engines called HeuriSamp, HeuriSamp-OPM, HeuriSim and HeuriSim-OPM.

5) It has been demonstrated that even the basic strategic engine Heuri is capable of outperforming Quackle engine playing in Spanish, while approaching a slightly superior performance of Quackle in French and English.

6) The more recent and extended engine HeuriSim-OPM has been tested to achieve a similar performance to Quackle's playing in English.

7) From massive playing of Heuri-vs-Heuri games, general statistics of Scrabble game, like a 16 point handicap of the second player, and word statistics in Spanish, as the most frequently played bingos, have been collected. This information can help the training of Scrabble human players.

8) A Spanish lexicon for playing Scrabble has been built that is used by Heuri engines. From this construction, a detailed study and classification of Spanish irregular verbs has been provided.

10.2 Future Work

To improve our engine Heuri, a possible path would be to combine techniques of Heuri and Quackle. Heuri calculates very well the expectation of making a Bingo in the next turn, using the tiles left on the rack after a move has been made, among other things, because it uses the exact position on the board in play. Nevertheless, Heuri does not calculate the point expectation of a non Bingo move in the next turn, using these same tiles left on the rack. Quackle estimates both of these expectations, the Bingo and non Bingo value points of multiset of tiles, but it does not use the board in play. These values are precomputed by playing hundrends of thousands of Quackle vs Quackle games. They are an average of the contribution of points of every multiset of tiles of length i where $(1 \le i \le 6)$, and they are stored in what they named Superleaves.

A possible line of work is to combine both approaches. To do this, we would have to play hundred of thousands of Heuri vs Heuri games and calculate the expectation of points of non-Bingo moves, for every multiset of tiles of length i where $(1 \le i \le 6)$, see the Appendix for a small advance in this direction.

After having these Heuri leaves, we would have a more robust Heuristic that contemplates Bingo and non Bingo expectations, of the residue, and also takes into account the points made by the move. Two other important important factors to take into account would be the point-spread between the scores of the players, and measuring how near is the end of the game, this last factor can be measured by counting the tiles left in the bag. We could adjust the parameters in the resulting heuristic by using the Reinforcement Learning, for instance, using the Deep Q-Learning Algorithm.

Another path is to emulate AlphaZero technique. That is replacing hand-crafted rules with a deep neural network and general purpose algorithms that know nothing about the game beyond the basic rules.

To learn the Scrabble game, an untrained neural network would play millions of games against itself via reinforcement learning. At first, it would play completely randomly, but over time the system would learn from wins and losses, adjusting the parameters of the neural network, making it more likely to choose advantageous moves in the future.

10.3 Publications derived from this thesis

[GonzalezAlquezar21] *Heuri: A Scrabble engine inspired by a Heuristic based on Probability*. Alejandro González Romero, René Alquézar Mancho, Arturo Ramírez Flores, Francisco González Acuña, Ian García Olmedo. (Under review in International Computer Games Association, ICGA Journal (preliminary accepted with major revisions)), 2021.

[GonzalezAlquezar18] *The Anagram Method : A Fast and Novel Scrabble Move Generator Algorithm.* Alejandro González Romero, René Alquézar Mancho, Arturo Ramírez Flores, Francisco González Acuña, Ian García Olmedo. Accepted and Presented at the “Congreso Mexicano de Inteligencia Artificial 2018 (COMIA 2018)”. Published in a special number of the magazine *Research in Computing Science (ISSN 1870-4069)*. This paper was presented in Merida, Yucatan, Mexico. June 2018.

[GonzalezAlquezar12] González Romero Alejandro, Alquézar René, Ramírez Flores Arturo, González Acuña Francisco. Heuristics and Fishing in Scrabble. Chairs : Tristan Cazenave, Jean M´ehat, Mark Winands (Eds.) *Proceedings of the European Conference on Artificial Intelligence, ECAI'12. Computer Games Workshop CGW12*; pags. 62-70 http://www.lirmm.fr/ecai2012/images/stories/ecai_doc/pdf/workshop/W33_ComputerGames.pdf Montpellier,2012.

[RamirezGonzalez09] Arturo Ramírez, Francisco González Acuña, Alejandro González Romero, René Alquézar, Enric Hernández, Amador Roldán Aguilar and Ian García Olmedo. A Scrabble Heuristic Based on Probability That Performs at Championship Level. In *Proceedings of the 8th Mexican International Conference on Artificial Intelligence MICAI*, pags.112-123. Springer, 2009.

[GonzalezAlquezar09] González Romero Alejandro, Alquézar René, Ramírez Flores Arturo, González Acuña Francisco, García Olmedo Ian. Human-like Heuristics in Scrabble. In *Artificial Intelligence Research and Development. Proceedings of the 12th International Conference of the Catalan Association for Artificial Intelligence*; S. Sandri et al. (Eds.) *Frontiers in Artificial Intelligence and Applications*. 381-390, 2009.

[GonzalezRamirez08] González Romero Alejandro, González Acuña Francisco, Ramírez Flores Arturo,Roldán Aguilar Amador, Alquézar René, Hernández Enric. A heuristic for Scrabble based in probability. Adi Botea and Carlos Linares Lopez, (Eds.) *Proceedings of the European Conference on Artificial Intelligence, ECAI'08 Workshop on Artificial Intelligence in Games, AIG'08*; pags 35-39. http://abotea.rsise.anu.edu.au/data/W9.pdf Patras, Greece. 2008.

[GonzalezAlquezar08] González Romero Alejandro, and Alquézar René. Building policies for Scrabble. In *Artificial Intelligence Research and Development. Proceedings of the 11th International Conference of the Catalan Association for Artificial Intelligence*; T. Alsinet et al. (Eds.); IOS Press. *Frontiers in Artificial Intelligence and Applications*. Vol 184; pags. 342-351. 2008.

Other publications not derived from the thesis

[GonzalezGonzalez18] *Optimal Strategies for Penalty Kicks.* Francisco González Acuña, Alejandro González Romero. Accepted for the MICAI 2018, 17th Mexican International Congress on Artificial Intelligence. To be published in IEEE Proceedings of the 17th Mexican International Congress on Artificial Intelligence. Preprint found in https://www.cicling.org/micai/2018/papers/cps/Optimal%20Strategies%20for%20Penalty%20Kicks.pdf . October 2018.

[Gonzalez05] *King Race*. Alejandro González Romero. Advances in Computer Games: 11th International Conference, ACG; pp. 155-164. Taipei, China. 2005.

[GonzalezAlquezar04] *To Block or Not to Block?*. Alejandro González Romero, René Alquézar. In: Lemaître C., Reyes C.A., González J.A. (eds) Advances in Artificial Intelligence – IBERAMIA 2004. IBERAMIA 2004. Lecture Notes in Computer Science, vol 3315. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-30498-2_14

Appendix

Heuri Leaves

We calculate here the number of Scrabble hands, 7-letter strings contained in a bag, including the initial bag. This is done in the Spanish, English and French versions. Also, we do the same for *i*-letter hands ($1 \leq i \leq 6$).

The results are given in the following table (see Table A.1).

Table A.1. Number of *i*-letter hands

i	Spanish	English	French
7	4167929	3199724	2954029
6	947972	737311	693825
5	186939	148150	141871
4	31056	25254	24562
3	4170	3509	3457
2	424	373	371
1	29	27	27
Total	5338519	4114348	3818142

The number of *i*-letter hands are useful to begin the compilation of superleaves especially in Spanish where this has not been done so far. Let us consider first the case of 7-letter hands.
We begin by listing all (15) partitions of 7 together with some associated numbers. We write a partition as a non increasing finite sequence p(i) = ($p_1(i)$, $p_2(i)$, ... , $p_\lambda(i)$) of positive integers (i=1, ... ,15) where $\lambda = \lambda(p_i)$ is the number of components of p(i). The *repetition index* r(i) of p(i) is $\prod^7_{j=1} c(j)!$ where c(j) is the number of components of p(i) equal to j.

We list the 15 partitions of 7 together with its repetition index.

p(1)=7, λ(1) =1, r(1)=1
p(2)=(6,1), λ(2) =2, r(2)=1
p(3)=(5,2), λ(3) =2, r(3)=1
p(4)=(5,1,1), λ(4) =3, r(4)=2
p(5)=(4,3), λ(5) =2, r(5)=1
p(6)=(4,2,1), λ(6) =3, r(6)=1
p(7)=(4,1,1,1), λ(7) =4, r(7)=6
p(8)=(3,3,1), λ(8) =3, r(8)=2
p(9)=(3,2,2), λ(9) =3, r(9)=2
p(10)=(3,2,1,1), λ(10) =4, r(10)=2
p(11)=(3,1,1,1,1), λ(11) =5, r(11)=24
p(12)=(2,2,2,1), λ(12) =4, r(12)=6
p(13)=(2,2,1,1,1), λ(13) =5, r(13)=12
p(14)=(3,2,1,1), λ(14) =4, r(14)=2
p(15)=(1,1,1,1,1,1,1), λ(15) =7, r(15)=5040

Now, denote by B the multiset of characters of the complete Scrabble bag. Let S be any multiset contained in B. The number H(S) of Scrabble 7-letter hands contained in S is:

$H(S)=\sum^{15}_{j=1}H_j(S)$ where $H_j(S)=S(1)*(S(2)-1)*(S(3)-2)*...*(S(\lambda)-\lambda+1)/r(j)$, S(i) is the number of characters of the alphabet contained in S with multiplicity greatear than or equal to i, λ is the number of components of p(j), and r(j) is the repetition index of p(j).

For example, for the complete bag B, in Spanish, we have B(1)=29, B(2)=18, B(3)=12, B(4)=12, B(5)=9, B(6)=5, B(7)=3 and H(B)=4167929, the total number of Scrabble hands in Spanish. In English the correspon-ding number is 3199724 , and in French 2954029.

Hand pairs

To calculate, exactly, the number of hand pairs, a hand for the first player and a hand for the second player, is more complicated. This number is $\sum_S H(B-S)$ where S runs over the set of 7-character hands (dealt to the first player) and B-S is the multiset difference of B and S. This expression has 4167929 summands; we estimate it to be approximately 10^{13} . To justify it we used two methods.

First, using cluster sampling, we took a sample of 1596 summands of $\sum_S H(B-S)$. The summands corresponding to S an S' belong to the same cluster iff there is a multiplicity-preserving permutation of the alphabet sending S to S'. The estimate of $\sum_S H(B-S)$ obtained was 9.75 trillion.

The second method resembles a census. We split the alphabet into two sets : the *extreme* characters, the 14 characters with multiplicity 1, 9 or 12, and the *mean* characters, the 15 characters with multiplicity 2, 4, 5 or 6. Now partition the set of 4167929 hands into 8 strata, the i-th stratum consisting of the hands with exactly i extreme characters (and 7 - i mean characters) i = 0, 1, 2, ..., ,7. We have calculated the cardinality of the 3-stratum to be 1170180 and the sum of the addends of $\sum_S H(B-S)$ corresponding to the elements S of this stratum to be 2784224896916. Assuming the 3-stratum to be a good representation of the whole sum, the estimate of $\sum_S H(B-S)$ turns out to be 9.92 trillion.

References

[AlphaGo16] David Silver, Aja Huang, Chris 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 & Demis Hassabis. *Nature* **529**, 484-489 (28 January 2016).

[AlphaZero17] David Silver, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lilicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel & Demis Hassabis. Mastering the game of Go without human knowledge. *Nature* **550**, 354-359 (19 October 2017).

[AlphaZero18] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 07 Dec 2018: Vol 362, Issue 6419, pp. 1140-1144. DOI: 10.1126/science.aar6404

[AZblog18] Alphazero: Shedding new light in the https://deepmind.com/blog/alphazero-shedding-new-light-grand-games-chess-shogi-and-go/

[AppelJacobson88] Andrew W. Appel, Guy J. Jacobson: The World's Fastest Scrabble Program, Communications of the ACM. v. 31 , Issue 5 (May 1988) p. 572 – 578, 1988.

[Bush45] Bush, Vannevar. As We May Think. *The Atlantic Monthly*. July 1945.

[BillingsDavidsonSchaeffer02] Darse Billings, Aaron Davidson, Jonathan Schaeffer, and Duane Szafron, The challenge of poker, Artificial Intelligence, v.134 n.1-2, p.201-240, January 2002.

[Bowling et al 17] M. Bowling, N. Burch, M. Johanson, O. Tammelin. Heads-up limit hold'em poker is solved. Communications of the ACM. Volume 60 Issue 11, p. 81-88, November 2017.

[CambleHsuHoane02] Cambell, Murray; Hoane, Joseph; and Hsu, Feng-hsiung. Deep Blue. *Artificial Intelligence. Volume 134, issue 1-2; pags. 57 – 83. 2002*

[Carter16] Gerry Carter. Man vs Machine. https://www.mindsportsacademy.com/Content/Details/1096?title=man-vs-machine , 2016.

[Chinook07] Chinook Checkers engine https://webdocs.cs.ualberta.ca/~chinook/play/ , 2007.

[DeepBlue02] Murray Campbell, A. Joseph Hoane Jr., Feng-hsiung Hsu. *Deep Blue*. Artificial Intelligence 134 , pp 57-83, 2002.

[DeepStack17] M. Moravcik, M. Schmid, N. Burch, V. Lisý, D. Morrill, N. Bard, T. Davis, K.Waugh, M. Johanson, M. Bowling. DeepStack: Expert-level artificial intelligence in heads-up no limit poker. *Science*, Vol 356, Issue 6337, 05 May 2017.

[FisherWebb04] Fisher Andrew, Webb David. How to win at Scrabble. 2014.

[GeeksMinimax_αβ] https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/

[Ginsberg99] Ginsberg M.L., GIB: Steps toward an expert-level bridge-playing program. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence* (IJCAI-99), *p. 584-589, 1999.*

[Gonzalez05] Alejandro González Romero. *King Race*. Advances in Computer Games: 11th International Conference, ACG; pp. 155-164. Taipei, China. 2005.

[GonzalezAlquezar18] *The Anagram Method : A Fast and Novel Scrabble Move Generator Algorithm.* Alejandro González Romero, René Alquézar Mancho, Arturo Ramírez Flores, Francisco González Acuña, Ian García Olmedo. Accepted and Presented at the "Congreso Mexicano de Inteligencia Artificial 2018 (COMIA 2018)". Published in a special number of the magazine *Research in Computing Science (ISSN 1870-4069)*. This paper was presented in Merida, Yucatan, Mexico. June 2018.

[GonzalezAlquezar12] González Romero Alejandro, Alquézar René, Ramírez Flores Arturo, González Acuña Francisco. Heuristics and Fishing in Scrabble. Chairs : Tristan Cazenave, Jean M´ehat, Mark Winands (Eds.) *Proceedings of the European Conference on Artificial Intelligence, ECAI'12. Computer Games Workshop CGW12*; pags. 62-70 http://www.lirmm.fr/ecai2012/images/stories/ecai_doc/pdf/workshop/W33_ComputerGames.pdf Montpellier, France. 2012

[GonzalezAlquezar04] *To Block or Not to Block?*. Alejandro González Romero, René Alquézar. In: Lemaître C., Reyes C.A., González J.A. (eds) Advances in Artificial Intelligence – IBERAMIA 2004. IBERAMIA 2004. Lecture Notes in Computer Science, vol 3315. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-30498-2_14

[GonzalezGonzalez18] *Optimal Strategies for Penalty Kicks.* Francisco González Acuña, Alejandro González Romero. Accepted for the MICAI 2018, 17th Mexican International Congress on Artificial Intelligence. To be published in IEEE Proceedings of the 17th Mexican International Congress on Artificial Intelligence. Preprint found in

https://www.cicling.org/micai/2018/papers/cps/Optimal%20Strategies%20for%20Penalty%20Kicks.pdf . October 2018.
[GonzalezRamirez08] González Romero Alejandro, González Acuña Francisco, Ramírez Flores Arturo,Roldán Aguilar Amador, Alquézar René, Hernández Enric. A heuristic for Scrabble based in probability. Adi Botea and Carlos Linares Lopez, (Eds.) *Proceedings of the European Conference on Artificial Intelligence, ECAI'08 Workshop on Artificial Intelligence in Games, AIG'08*; pags 35-39. http://abotea.rsise.anu.edu.au/data/W9.pdf Patras, Greece. 2008.
[GonzalezAlquezar08] González Romero Alejandro, and Alquézar René. Building policies for Scrabble. In *Artificial Intelligence Research and Development. Proceedings of the 11th International Conference of the Catalan Association for Artificial Intelligence*; T. Alsinet et al. (Eds.); IOS Press. *Frontiers in Artificial Intelligence and Applications*. Vol 184; pags. 342-351. 2008.
[GonzalezAlquezar09] González Romero Alejandro, Alquézar René, Ramírez Flores Arturo, González Acuña Francisco, García Olmedo Ian. Human-like Heuristics in Scrabble. In *Artificial Intelligence Research and Development. Proceedings of the 12th International Conference of the Catalan Association for Artificial Intelligence*; S. Sandri et al. (Eds.) *Frontiers in Artificial Intelligence and Applications*. 381-390, 2009.
[Gordon94] Steven A. Gordon. A Faster Scrabble Move Generation Algorithm. Software—Practice and Experience, v. 24(2), p.219–232, February 1994.
[HM13] Shih-Chieh Huang and Martin Müller. Investigating the limits of Monte-Carlo tree search methods in computer Go. In Computers and Games, volume 8427 of Lecture Notes in Computer Science, pages 39–48. Springer, 2013.
[HM95] Jun Han and Claudio Moraga. The influence of the sigmoid function parameters on the speed of backpropagation learning. In IWANN, volume 930 of Lecture Notes in Computer Science, pages 195–201. Springer, 1995.
[Högy19] Kevin Högy. A new age in computer chess? Lc0 beats Stockfish. https://chess24.com/en/read/news/a-new-age-in-computer-chess-leela-beats-stockfish . June 2, 2019.

[Hui18] Jonathan Hui. RL-Introduction to Deep Reinforcement Learning. Medium Machine Learning https://medium.com/@jonathan_hui/rl-introduction-to-deep-reinforcement-learning-35c25e04c199
[Jui99] Hugues Rene Juille. Methods for statistical inference: Extending the evolutionary computation paradigm. 1999. AAI9927229.
[Kaelbling96] Kaelbling, Leslie P. ; Littman, Michael L. ; Moore, Andrew W. "Reinforcement Learning: A Survey". Journal of Artificial Intelligence Research. 4: 237-285. 1996. arXiv:cs/9605103.
[Kasparov18] Garry Kasparov. Chess, a Drosophila of reasoning. Science 07 Dec 2018: Vol. 362, Issue 6419, pp 1087. DOI: 10.1126/science.aaw2221 2018.
[KatzO'Laughlin06] Jason Katz-Brown, John O'Laughlin, John Fultz and Matt Liberty, Quackle is an open source crossword game program released in March 2006. *How Quackle plays Scrabble* http://people.csail.mit.edu/jasonkb/quackle/doc/how_quackle_plays_scrabble.html
[Knuth and Moore 1975] Donald E Knuth and Ronald W Moore. An analysis of alpha-beta pruning. Artificial intelligence, 6(4):293–326, 1975.
[Kor85] Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. Artificial intelligence, 27(1):97–109, 1985.
[KS06a] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In ECML, volume 4212 of Lecture Notes in Computer Science, pages 282–293. Springer, 2006.
[KS06b] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In ECML, volume 4212 of Lecture Notes in Computer Science, pages 282–293. Springer, 2006.
[KSW06] Levente Kocsis, Csaba Szepesvári, and Jan Willemson. Improved Monte-Carlo search. Univ. Tartu, Estonia, Tech. Rep, 1, 2006.
[Lafferty97] Lafferty D. Chinook draws the hundred years problem. *The American Checker FederationBulletin*, (269):6. 1997.
[Lawson04] Lawson, Mark V. *Finite automata*. Chapman and Hall/CRC. 2004.
[LCZero18] Leela Chess Zero (2018). https://www.chessprogramming.org/Leela_Chess_Zero
[Lewis12] Joshua Lewis. Rethinking the value of Scrabble tiles. 2012 http://www.thelastwordnewsletter.com/Last_Word/Rethinking_tile_value_113.html
[Lit94] Michael L Littman. Markov games as a framework for multi-agent reinforcement learning. In Proceedings of the eleventh international conference on machine learning, volume 157, pages 157–163, 1994.
[Matsumoto14] Kenji Matsumoto "Breaking the game," 2014. [Online]. Available: http://www.breakingthegame.net/strategy
[McGuireSmith06] McGuire, Brian and Smith, Chris. The History of Artificial Intelligence. http://courses.cs.washington.edu/courses/csep590/06au/projects/history-ai.pdf. University of Washington. December 2006.
[Mec98] David A Mechner. All systems Go. The Sciences, 38(1):32–37, 1998.

[MHSS14] Chris J. Maddison, Aja Huang, Ilya Sutskever, and David Silver. Move evaluation in Go using deep convolutional neural networks. CoRR, abs/1412.6564, 2014.

[Michie66]. *Game Playing and Game Learning Automata.* Advances in Programming and Non-Numerical Computation, Leslie Fox (ed.), pp. 183-200. Oxford, Pergamon.» Includes Appendix: *Rules of SOMAC* by John Maynard Smith, introduces Expectiminimax tree

[Mül02] Martin Müller. Computer Go. Artificial Intelligence, 134(1-2):145–179, 2002.

[NatAld16] Nathan Companez & Aldeida Aleti. *"Can Monte-Carlo Tree Search learn to sacrifice?,"* Journal of Heuristics, Springer, vol. 22(6), pages 783-813, December 2016.

[Rai16] Tapani Raiko. Towards super-human artificial intelligence in Go by further improvements of AlphaGo.

[Rama10] Ramanujan, R., Sabharwal, A., Selman, B. : *On adversarial search spaces and sampling-based planning*. In: The International Conference on Automated Planning and Scheduling (ICAPS), pp. 242–245, 2010.

[RamirezGonzalez09] Arturo Ramírez, Francisco González Acuña, Alejandro González Romero, René Alquézar, Enric Hernández, Amador Roldán Aguilar and Ian García Olmedo. A Scrabble Heuristic Based on Probability That Performs at Championship Level. In *Proceedings of the 8th Mexican International Conference on Artificial Intelligence MICAI*, pags. 112-123, Springer, 2009.

[RichardsAmir07] Richards, M., Amir, E., Opponent modeling in Scrabble, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, p.1482-1487, Hyderabad, India, January 2007.

[RTL+10] Arpad Rimmel, Olivier Teytaud, Chang-Shing Lee, Shi-Jim Yen, Mei-Hui Wang, and Shang-Rong Tsai. Current frontiers in computer Go. IEEE Transactions on Computational Intelligence and AI in Games, 2(4):229–238, 2010.

[RussellNorvig09]Stuart J. Russell; Peter Norvig. Artificial Intelligence: A Modern Approach. Prentice Hall. 2009.

[SadlerRegan19] Matthew Sadler and Natasha Regan. *Game Changer: AlphaZero's Groundbreaking Chess Strategies and the Promise of AI*. New in Chess. January 2019.

[Samuel59] Samuel, Arthur. Some studies in machine learning using the ga[me of checkers: Recent progress. *IBM Journal of Research and Development*, 3:210-229. 1959.

[Samuel67] Samuel, Arthur. Some studies in machine learning using the game of checkers: II-Recent progress. *IBM Journal of Research and Development*, 11:601-617. 1967.

[Schaeffer et al 07] Jonathan Schaeffer, Neil Burch, Yngvi Bjornsson, Akihiro Kishimoto, Martin Muller, Rob Lake, Paul Lu and Steve Sutphen. "Checkers is Solved", *Science*, 2007.

[Sch10] W Schubert. KGS Go server, 2010.

[Shannon50] Shannon, Claude. Programming a computer for playing chess. *Philosophical Magazine, Ser. 7,* 41:256-275. 1950.

[Shapiro79] Shapiro, S.C. A Scrabble crossword game playing program. Proceedings of the Sixth IJCAI, p. 797-799, 1979.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. Nature, 529(7587):484–489, 2016.

[SN08] Ilya Sutskever and Vinod Nair. Mimicking Go experts with convolutional neural networks. In ICANN (2), volume 5164 of Lecture Notes in Computer Science, pages 101–110. Springer, 2008.

[ST09] David Silver and Gerald Tesauro. Monte-Carlo simulation balancing. In ICML, volume 382 of ACM International Conference Proceeding Series, pages 945–952. ACM, 2009.

[Stuart82] Stuart, S.C. Scrabble crossword game playing programs. SIGArt Newsletter,80. p.109- 110, April 1982.

[Sheppard02a] Sheppard Brian. World-championship-caliber Scrabble, *Artificial Intelligence*, v.134, n.1-2, p.241-275, January 2002.

[Sheppard02b] Sheppard Brian. *Towards Perfect Play of Scrabble*. PhD thesis, IKAT/Computer Science Department, Universiteit Maastricht, July 2002.

[Sheppard03] Sheppard Brian. Endgame play in Scrabble. *International Computer Games Association (ICGA) Journal*. Vol.26 No. 3. September 2003.

[Tesauro95] Tesauro, Gerald. “Temporal Difference Learning and TD-Gammon”. Communications of the ACM. 38 (3), March 1995.

[TF06] John Tromp and Gunnar Farnebäck. Combinatorics of Go. In Computers and Games, volume 4630 of Lecture Notes in Computer Science, pages 84–99. Springer, 2006.

[Thomas11] Thomas Andrew C. Variance Decomposition and Replication In Scrabble: When You Can Blame Your Tiles? http://arxiv.org/pdf/1107.2456.pdf . November 2, 2011.

[Thompson86] Thompson Ken. Retrograde analysis of certain endgames. *Journal of the International Computer Chess Association*, 9(3): 131-139. 1986.

[Trigo77] O. Trigo. Estrategias de señalamiento en juegos de memoria imperfecta. Un juego de dominó. Tesis, Facultad de Ciencias, UNAM. (1977)

[Turing50] Turing, Alan. Computing Machinery and Intelligence. *Mind 49*, 433 – 460. 1950.
[Turing53] Turing, Alan. Digital computers applied to games. In B. Bowden, editor, *Faster that Thought*, pages 286-295. Pitman. 1953.
[Upt98] Robin JG Upton. Dynamic stochastic control: A new approach to tree search & game-playing. University of Warwick, UK, 23, 1998.
[VDHUVR02] H Jaap Van Den Herik, Jos WHM Uiterwijk, and Jack Van Rijswijck. Games solved: Now and in the future. Artificial Intelligence, 134(1-2):277–311, 2002.
[VonNewmanMorgenstern53] John Von Newman, Oskar Morgenstern: The theory of Games and Economic behavior. *Princeton University Press*, 1953.
[WG07] Yizao Wang and Sylvain Gelly. Modifications of UCT and sequence-like simulations for Monte-Carlo Go. Pages 175–182, 2007.
[WH86] DRGHR Williams and GE Hinton. Learning representations by back-propagating errors. Nature, 323(6088):533–538, 1986.
[Whitehill] *Whitehill, Bruce. "Scrabble". The Big Game Hunter.*
[Wickman08] Wickman Brian J. Scrabble on the TeraScale. https://www.eecs.northwestern.edu/~robby/uc-courses/22001-2008-winter/scrabble-on-the-tetrascale-wickman.pdf University of Nebraska Lincoln. February 27, 2008 or https://pdfs.semanticscholar.org/2d5e/05c0b29a5889ee0801e1dcb0c1aeff12a69f.pdf?_ga=2.157197874.826441.1493920552-1036224300.1493920552
[WikiAlphaGo] Wikipedia entry on Alpha Go. https://en.wikipedia.org/wiki/AlphaGo
[WikiChess] Wikipedia entry on Computer Chess. http://en.wikipedia.org/wiki/Computer_chess
[WikiDeepBlue] Deep Blue (chess computer). https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer)
[WikiExpecti18] Expectiminimax. Wikipedia https://en.wikipedia.org/wiki/Expectiminimax 2018.
[WikiLCZ18] Leela Chess Zero (2018). Wikipedia https://en.wikipedia.org/wiki/Leela_Chess_Zero
[WikiMavenScrabble18] Maven (Scrabble). Wikipedia https://en.wikipedia.org/wiki/Maven_(Scrabble) last edited on 2018.
[WikiMCTS19] Monte-Carlo Tree Search. Wikipedia https://en.wikipedia.org/wiki/Monte_Carlo_tree_search last edited on 2019.
[WikiRL19] Reinforcement Learning. Wikipedia https://en.wikipedia.org/wiki/Reinforcement_learning last edited on 2019.
[WikiScrabbleA] Wikipedia entry on Acceptable words of Scrabble https://en.wikipedia.org/wiki/Scrabble#Acceptable_words Scrabble
[Wil92] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine learning, 8(3-4):229–256, 1992.
[Wojciech14] Wojciech Wieczorek, Olgierd Unold. Induction of Directed Acyclic Word Graph in a Bioinformatics Task. *The Journal of Machine Learning Research JMLR, Workshop and Conference Proceedings W&CP* 34: 207-217, 2014.
[Wouter16] Wouter M. Koolen, Steven de Rooij. The GADDAG. February 1, 2016. https://http://blog.wouterkoolen.info/Gaddag/post.html
[WZZ+16] Fei-Yue Wang, Jun Jason Zhang, Xinhu Zheng, Xiao Wang, Yong Yuan, Xiaoxiao Dai, Jie Zhang, and Liuqing Yang. Where does AlphaGo go: from church-turing thesis to AlphaGo thesis and beyond. IEEE/CAA Journal of Automatica Sinica, 3(2):113–120, 2016.