Credit Hours System
CMPN402
Machine Intelligence
Spring 2019

Cairo University
Faculty of Engineering

Wehrmacht Team

Scrabble game

Project Supervisor

Prof. Dr. Nevin Darwish

Computer Engineering Department

Authors:

Mustafa Mufeed

Abeer Mohamed

Project Leaders:

Ahmed Salem Elhady

Mina Ashraf Louis

Abstract

The report explains how Wehrmacht Team Designed and implemented the Scrabble game and the programming languages/ technologies used as well as the implementation of the game Graphical user interface tools and the testing and training that was done to the agent that would be used for the upcoming competition with the other team in the Machine Intelligence course.

In the first section we introduce what is the history of the scrabble game and the basic game rules, in order to build the main idea on how the work follow went through the project and the reasons for choosing the mentioned algorithms for Agent implementation. After that we demonstrate a market survey to show our customers who are willing to buy/play our game and the other games of scrabble that are currently in the market.

After the research phase we oved to the methods and algorithms used for implementation; we decided to implemented our board using Bitboard instead of 2D array and used Monte Carlo for the logic implementation. Then we designed the interface and implemented the Graphical user interface using unity since unity is the current dominate tool used in the game market now a days.

After finishing the implementation and the communication between the logic and the user interface, tests were done to the project units and the project as whole, and the agent successfully managed to beat the human player in the tests after being trained and the game simulate the original scrabble game but with better and quicker logic and interface.

It is hoped that our game will be able to perform as well as it did during the testing and simulations and that it will be considered to be released to the market in the future to be able to compete with the other scrabble games that are currently in the market.

Table of Contents

List of Figures

List of Tables

Team Contacts

Name	ID	Email	Team Role
Ahmed Salem Muhammad Elhady	1155070	ahelhady511@gmail.com	Leader
Mina Ashraf Lois	1152055	Minaashraf97@gmail.com	Co-Leader
Mustafa Ashraf Moharram	1134441	mustafaashraf@gmail.com	Research
Omar Osama Saleh	1132225	omarosamasaleh@gmail.com	Research
Ali Ahmed Mostafa	1152397	Ali_Ahmed97@hotmail.com	Implementation
Amr Khaled	1152003	amrkh97@gmail.com	Implementation
Andrew Baher Georgy	1152173	Andrew.baher@gmail.com	Implementation
Mohamed Basel Mohamed	1152253	Basselmoahmed@gmail.com	Implementation
Seif Eldien Adel	1153037	Seif199714@gmail.com	Implementation
Seif Eldien Tarek	1152013		Implementation
Wael Ashraf	1152195	wael.ashraf.anwar@gmail.com	Implementation
Yahia Mohamed Yahia	1155413	Yahia.M.Yahia96@gmail.com	Implementation
Mostafa Abdelgawwad Mohammed	1155112	Mostafaelgawwad1997@gmail.com	Integration
Nourhan Elsayed	1142061	Nourhaneattya@hotmail.com	Integration
Ahmed Hussein Fekry	1152259	hussein.ahmed50@gmail.com	Communication and Testing
Mohamed Anwar	1152055	Mohamed.anwar97@hotmail.com	Communication and Testing
Omar Yousry	1152065	Omaryousry97@yahoo.com	Communication and Testing
Ahmed Mohamed Salah Eldein	1152114	ahmed.mohamed973@eng-st.cu.edu.eg	GUI
Mahmoud Mohamed Morsy	1142019	mahmoud_morsy@live.com	GUI
Abeer Mohammed	1152077	Abeerrefay1@gmail.com	Documentation
Mostafa Mufeed	1162249	m.mufeed1996@gmail.com	Documentation

Table 1 -1 Team contacts

Section 1: Introduction

1. Introduction

1.1 About Scrabble

Scrabble is a word game in which two to four players score points by placing tiles bearing a single letter onto a board divided into a 15×15 grid of squares. The tiles must form words that, in crossword fashion, read left to right in rows or downward in columns, and be included in a standard dictionary or lexicon.

The name is a trademark of Mattel in most of the world, but of Hasbro, Inc. in the United States and Canada. The game is sold in 121 countries and is available in 29 languages; approximately 150 million sets have been sold worldwide and roughly one-third of American and half of British homes have a Scrabble set. There are around 4,000 Scrabble clubs around the world.

1.2 History of Scrabble

In 1938, American architect Alfred Mosher Butts created the game as a variation on an earlier word game he invented called Lexiko. The two games had the same set of letter tiles, whose distributions and point values Butts worked out by performing a frequency analysis of letters from various sources, including The New York Times. The new game, which he called "Criss-Crosswords," added the 15×15 game board and the crossword-style game play. He manufactured a few sets himself, but was not successful in selling the game to any major game manufacturers of the day.

1.3 Game Rules

1.3.1 Notation System

- In the notation system common in tournament play, columns are labeled with the letters "A–O" and rows with the numbers "1–15".
- A play is usually identified in the format xy WORD score or WORD xy score, where x denotes the column or row on which the play's main word extends, y denotes the second coordinate of the main word's first letter, and WORD is the main word.
- additional words formed by the play are sometimes listed after the main word and a slash. When the play of a single tile forms words in each direction, one of the words is arbitrarily chosen to serve as the main word for purposes of notation.
- When a blank tile is employed in the main word, the letter it has been chosen to represent is indicated with a lower-case letter, or, in handwritten notation, with a square around the letter. When annotating a play, previously existing letters on the board are usually enclosed in parentheses.

1.3.2 Sequence of Play

- Before the game, a resource, either a word list or a dictionary, is selected for the purpose of adjudicating any challenges during the game. The tiles are either put in an opaque bag or placed face down on a flat surface.
- Opaque cloth bags and customized tiles are staples of clubs and tournaments, where games are rarely played without both.
- Next, players decide the order in which they play. The normal approach is for players to each draw one tile: The player who picks the letter closest to the beginning of the alphabet goes first, with blank tiles taking precedence over the letter A.
- In most North American tournaments, the rules of the US-based North American Scrabble Players Association (NASPA) stipulate instead that players who have gone first in the fewest number of previous games in the tournament go first, and when that rule yields a tie, those who have gone second the most go first. If there is still a tie, tiles are drawn as in the standard rules.
- At the beginning of the game, each player draws seven tiles from the bag and places them on his or her rack, concealed from the other player(s).

1.3.3 Making a Play

- The first played word must be at least two letters long, and cover H8 (the center square). Thereafter, any move is made by using one or more tiles to place a word on the board. This word may use one or more tiles already on the board and must join with the cluster of tiles already on the board.
- On each turn, the player has three options:

    1. Pass, forfeiting the turn and scoring nothing.
    2. Exchange one or more tiles for an equal number from the bag, scoring nothing, an option available only if at least seven tiles remain in the bag.
    3. Play at least one tile on the board, adding the value of all words formed to the player's cumulative score.

- A proper play uses one or more of the player's tiles to form a continuous string of letters that make a word (the play's "main word") on the board, reading either left-to-right or top-to-bottom. The main word must either use the letters of one or more previously played words or else have at least one of its tiles horizontally or vertically adjacent to an already played word. If any words other than the main word are formed by the play, they are scored as well, and are subject to the same criteria of acceptability. See Scoring for more details.
- A blank tile may represent any letter, and scores zero points, regardless of its placement or what letter it represents. Its placement on a double-word or triple-word square causes the corresponding premium to be applied to the word(s) in which it is used. Once a blank tile is placed, it remains that particular letter for the remainder of the game.
- After making a play, the player announces the score for that play, and then, if the game is being played with a clock, starts his or her opponent's clock. The player can change his play as long as his or her clock is running, but commits to the play

when he or she starts the opponent's clock. The player then draws tiles from the bag to replenish his or her rack to seven tiles. If there are not enough tiles in the bag to do so, the player takes all the remaining tiles.

- If a player has made a play and has not yet drawn a tile, the opponent may choose to challenge any or all words formed by the play. The player challenged must then look up the words in question using a specified word source (such as OTCWL, the Official Scrabble Players Dictionary, or CSW) and if any one of them is found to be unacceptable, the play is removed from the board, the player returns the newly played tiles to his or her rack and the turn is forfeited. In tournament play, a challenge may be to the entire play or any one or more words, and judges (human or computer) are used, so players are not entitled to know which word(s) are invalid. Penalties for unsuccessfully challenging an acceptable play vary in club and tournament play, and are described in greater detail below.

1.3.4 End of Game

- Under North American tournament rules, the game ends when either

1. one player plays every tile on his or her rack, and there are no tiles remaining in the bag (regardless of the tiles on his or her opponent's rack)
2. at least six successive scoreless turns have occurred and either player decides to end the game
3. either player uses more than 10 minutes of overtime. (For several years, a game could not end with a cumulative score of 0–0, but that is no longer the case, and such games have since occurred a number of times in tournament play, the winner being the player with the lower total point value on his or her rack).

- When the game ends, each player's score is reduced by the sum of his or her unplaced letters. In addition, if a player has used all of his or her letters (known as "going out" or "playing out"), the sum of the other player's unplaced letters is added to that player's score; in tournament play, a player who goes out adds twice that sum, and his or her opponent is not penalized.

1.3.5 Scoring

- The score for any play is determined this way:

- Each new word formed in a play is scored separately, and then those scores are added up. The value of each tile is indicated on the tile, and blank tiles are worth zero points.
- The main word (defined as the word containing every played letter) is scored. The letter values of the tiles are added up, and tiles placed on Double Letter Score (DLS) and Triple Letter Score (TLS) squares are doubled or tripled in value, respectively. Tiles placed on Double Word Score (DWS) or Triple Word Score (TWS) squares double or triple the value of the word(s) that include those tiles, respectively. In particular, the center square (H8) is considered a DWS, and the first play is doubled in value.
- If any "hook" words are played (e.g. playing ANEROID while "hooking" the A to BETTING to make ABETTING), the scores for each word are added separately. This is common for "parallel" plays that make up to eight words in one turn.

- Premium squares apply only when newly placed tiles cover them. Any subsequent plays do not count those premium squares.
- If a player covers both letter and word premium squares with a single word, the letter premium(s) is/are calculated first, followed by the word premium(s).
- If a player makes a play where the main word covers two DWS squares, the value of that word is doubled, then redoubled (i.e. 4× the word value). Similarly, if the main word covers two TWS squares, the value of that word is tripled, then re-tripled (9× the word value). Such plays are often referred to as "double-doubles" and "triple-triples" respectively. It is theoretically possible to achieve a play covering three TWS squares (a 27× word score), although this is extremely improbable without constructive setup and collaboration. Plays covering a DWS and a TWS simultaneously (6× the word value, or 18× if a DWS and two TWS squares are covered) are only possible if a player misses the center star on the first turn, and the play goes unchallenged (this is valid under North American tournament rules).
- Finally, if seven tiles have been laid on the board in one turn, known as a "bingo" in North America and as a "bonus" elsewhere, after all of the words formed have been scored, 50 bonus points are added.
- When the letters to be drawn have run out, the final play can often determine the winner. This is particularly the case in close games with more than two players.
- Scoreless turns can occur when a player passes, exchanges tiles, or loses a challenge. The latter rule varies slightly in international tournaments. A scoreless turn can also theoretically occur if a play consists of only blank tiles, but this is extremely unlikely in actual play.

1.3.6 Acceptable Words

- Acceptable words are the primary entries in some chosen dictionary, and all of their inflected forms. Words that are hyphenated, capitalized (such as proper nouns), or apostrophized are not allowed, unless they also appear as acceptable entries; JACK is a proper noun, but the word JACK is acceptable because it has other usages as a common noun (automotive, vexillological etc.) and verb that are acceptable.
- Acronyms or abbreviations, other than those that have acceptable entries (such as AWOL, RADAR, LASER, and SCUBA) are not allowed.
- Variant spellings, slang or offensive terms, archaic or obsolete terms, and specialized jargon words are allowed if they meet all other criteria for acceptability, but archaic spellings (e.g. NEEDE for NEED) are generally not allowed.
- Foreign words are not allowed in English-language Scrabble unless they have been incorporated into the English language, as with PATISSERIE, KILIM, and QI. Vulgar and offensive words are generally excluded from the OSPD4 but allowed in club and tournament play.
- Proper nouns and other exceptions to the usual rules are allowed in some limited contexts in the spin-off game Scrabble Trickster. Names of recognized computer programs are permitted as an acceptable proper noun (For example, WinZip).
- The memorization of two-letter words is considered an essential skill in this game.
- There are two popular competition word lists used in various parts of the world:

- TWL (also known as OTCWL, OWL, or TWL).
- SOWPODS (also called "Collins" or "CSW").

The first is used in America, Canada, Jerusalem and Thailand, and the second in all other English-speaking countries.

1.3.7 Challenges

- The penalty for a successfully challenged play is nearly universal: the offending player removes the tiles played and forfeits his or her turn. (In some online games, an option known as "void" may be used, wherein unacceptable words are automatically rejected by the program. The player is then required to make another play, with no penalty applied.)
- The penalty for an unsuccessful challenge (where all words formed by the play are deemed valid) varies considerably, including:

- "Double Challenge", in which an unsuccessfully challenging player must forfeit the next turn. This penalty governs North American (NASPA-sanctioned) OWL tournament play, and is the standard for North American, Israeli, and Thai clubs. Because loss of a turn generally constitutes the greatest risk for an unsuccessful challenge, it provides the greatest incentive for a player to "bluff", or play a "phony" – a plausible word that they know or suspect to be unacceptable, hoping his or her opponent will not call him on it. Or a player can put down a legal word that appears to be a phony hoping the other player will incorrectly challenge it and lose their turn.
- "Single Challenge"/"Free Challenge", in which no penalty whatsoever is applied to a player who unsuccessfully challenges. This is the default rule in Ireland and the United Kingdom, as well as for many tournaments in Australia, although these countries do sanction occasional tournaments using other challenge rules.
- Modified "Single Challenge", in which an unsuccessful challenge does not result in the loss of the challenging player's turn, but is penalized by the loss of a specified number of points. The most common penalty is five points. The rule has been adopted in Singapore (since 2000), Malaysia (since 2002), South Africa (since 2003), New Zealand (since 2004), and Kenya, as well as in contemporary World Scrabble Championships (since 2001) and North American (NASPA-sanctioned) Collins tournaments, and particularly prestigious Australian tournaments. Some countries and tournaments (including Sweden) use a 10-point penalty instead. In most game situations, this penalty is much lower than that of the "double challenge" rule. Consequently, such tournaments encourage greater willingness to challenge and discourage playing dubious words.

- Under NASPA tournament rules, a player may request to "hold" the opponent's play in order to consider whether to challenge it, provided that the opponent has not yet drawn replacement tiles. If player A holds, player A's clock still runs, and player B may not draw provisional replacement tiles until 15 seconds after the hold was announced (which tiles must then be kept separate). There is no limit on how long player A may hold the play. If player A successfully challenges after player B drew provisional replacement tiles, player B must show the drawn tiles before returning them to the bag.

Section 2: Market Survey

2. Market Survey

2.1 Intended Customers

Our main targeted customers are:

- Gamers who love challenging games
- Lovers of Scrabble
- Gaming Software Developers

2.2 Online Gaming Marketing Analysis

Online gaming market is expected to witness substantial growth over the forecast period. This may be attributed to increasing number of users taking up online gaming as an entertainment tool, the following figure represents how much people have consumed in online Gaming during the past few years.

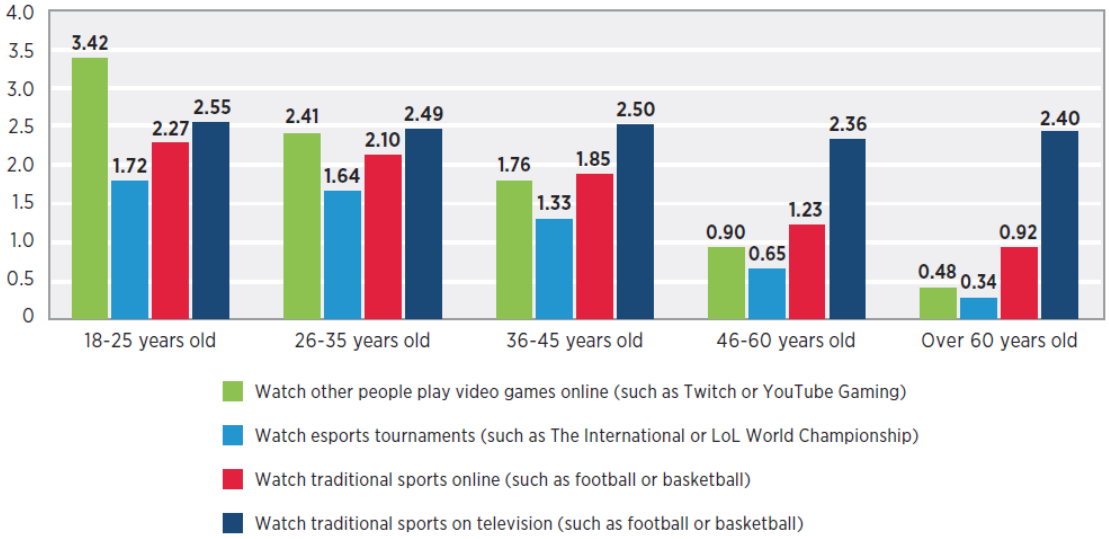


Figure 1: Online Gaming Statistics.

And according to a lot of Statistics:

- The average age of gamers: 35
- The average age of game purchasers: 38
- Households that own a device used for playing video games: 65%
- Households that own a device exclusively for playing video games: 48%
- The average number of years gamers have been playing: 13

2.3 Popularity of Scrabble

Scrabble is very popular among both men, women and children.

Figure 2: Family Playing Scrabble

2.4 Scrabble games in the Market

2.4.1 Overview of Maven and Quackle AI

Currently, Maven and Quackle are the leading Scrabble AI's. Maven was created in 2002 by Brian Sheppard whereas Quackle is an open source Scrabble AI developed by Jason Katz-Brown and John O'Laughlin in 2006.

2.4.1.1 Quackle

- Quackle is an open source Scrabble AI tool originally written in C++. We implemented the underlying heuristics. Quackle is similar to Maven in heuristic applications, but it has advanced tool for analysis.
- Quackle has two critical components called kibitzer and simulation engine. This section will overview the workflow, powerful features of Quackle and presents the strengths and limitation of the AI.

- A Quackle engine uses a program module called kibitzer that applies a different kind of static evaluation function than Maven to find the most promising candidate in the game. Figure 2 shows the Quackle flowchart.

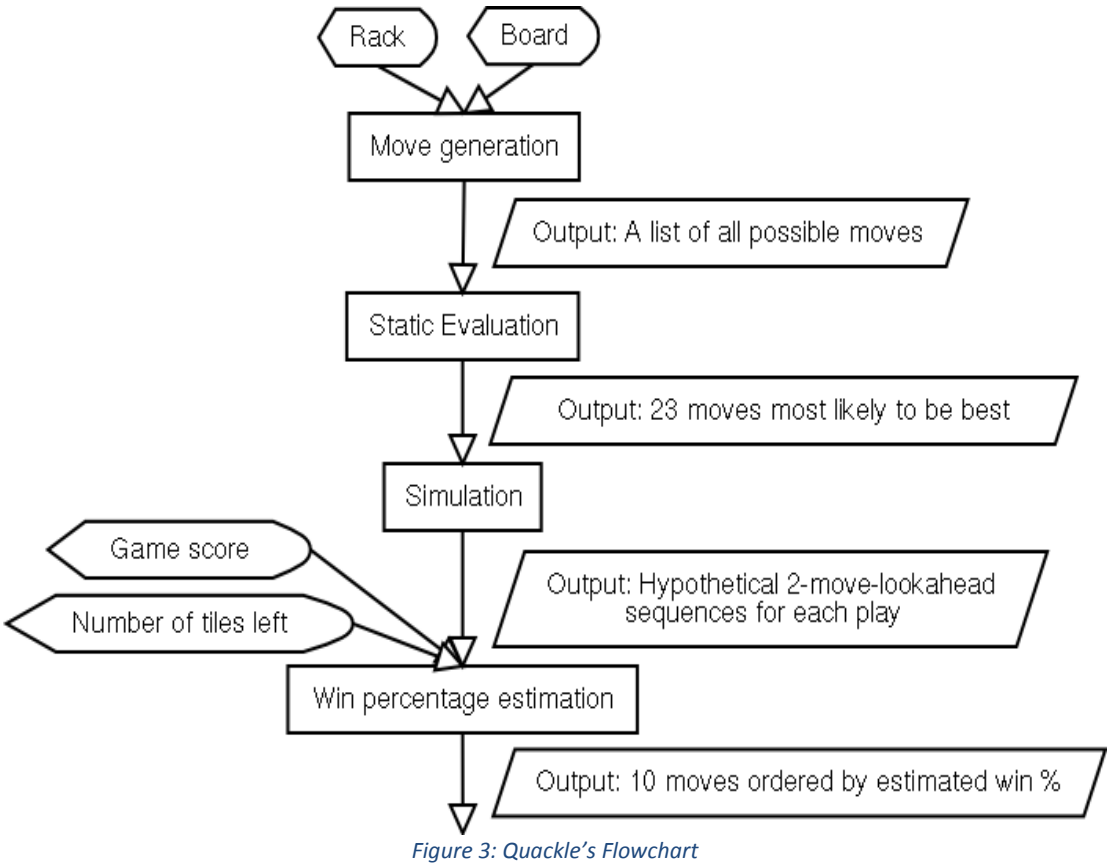


Figure 3: Quackle's Flowchart

2.4.1.2 Maven

It uses a Directed Acyclic Word Graph (DAWG) data-structure for dictionary representation and word generation. In Maven, each move is influenced by three main factors. These three elements are the current score, the player rack, and the current game board.

- Maven's game play is sub-divided into three phases:
  - Mid-game phase.
  - Pre-endgame phase.
  - Endgame phase.
- **The "mid-game" phase:**
  - The "mid-game" 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.
  - 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.
- **The "pre-endgame" phase:**
  - The "pre-endgame" phase works in almost the same way as the "mid-game" phase, except that it is designed to attempt to yield a good end-game situation.
- **The "endgame" phase:**
  - 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-star search algorithm to analyze the game tree during the endgame phase.

Section 3: Research

3. Research

3.1 Board Representation

A bitboard is a data structure commonly used in computer systems that play board games.
A bitboard, often used for board games such as chess, checkers, Othello and word games, is a specialization of the bit array data structure, where each bit represents a game position or state, designed for optimization of speed and/or memory or disk use in mass calculations. Bits in the same bitboard relate to each other in the rules of the game, often forming a game position when taken together. Other bitboards are commonly used as masks to transform or answer queries about positions.

3.1.1 Why Bitboard is used over 2D array

The advantage of the bitboard representation is that it takes advantage of the essential logical bitwise operations available on nearly all CPUs that complete in one cycle and are fully pipelined and cached etc. Nearly all CPUs have AND, OR, NOR, and XOR. Many CPUs have additional bit instructions, such as finding the "first" bit, that make bitboard operations even more efficient. If they do not have instructions well known algorithms can perform some "magic" transformations that do these quickly.

In terms of memory:

Bitboards are extremely compact. Since only a very small amount of memory is required to represent a position or a mask, more positions can find their way into registers, full speed cache, Level 2 cache, etc. In this way, compactness translates into better performance (on most machines). Also, on some machines this might mean that more positions can be stored in main memory before going to disk.

3.1.1 Usage of Bitboard and tile representation

This link explains the most important details:
http://boardword.com/static/bitboards.html

3.2 Move Generation

This project uses a data structure called a GADDAG for lexicon storage and a weighted heuristics method for word evaluation in a computer-based game of Scrabble. The GADDAG structure is assessed for its speed and performance in generating words. It was found to be well suited for this purpose facilitating rapid word generation. It required quite a large amount of system memory to be constructed however, which is highlighted to be a possible flaw. Weighted heuristics proved to be a fast and reliable method for move evaluation for the most part. However, it was prone to some strategic errors in judgement. An alternative method to address this is also discussed. Weighted heuristics could also be used to introduce other game elements such as realism to the game. A method for doing this using a word frequency data source is proposed.

3.3 Searching the Best State

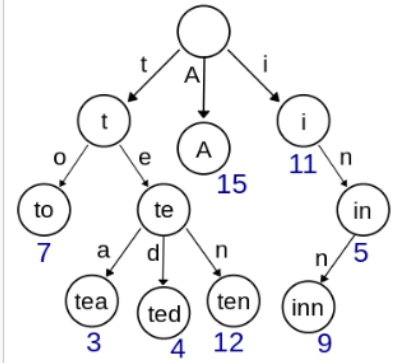
3.3.1 Search Tree

A **trie**, also called **digital tree**, **radix tree** or **prefix tree**, is a kind of search tree an ordered tree data structure used to store a dynamic set or associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string.

Keys tend to be associated with leaves, though some inner nodes may correspond to keys of interest. Hence, keys are not necessarily associated with every node. For the space-optimized presentation of prefix tree.

3.3.2 Search Algorithms

Trie as lexicon has a number of advantages over binary search trees. A trie can also be used to replace a hash table, over which it has the following advantages:



A trie for keys "A", "to", "tea", "ted", "ten", "i", "in", and "inn". Note that this example does not have all the children alphabetically sorted from left to right as it should be (the root and node 't').

Figure 5: Trie

- Looking up data in a trie is faster in the worst case, O(m) time (where m is the length of a search string), compared to an imperfect hash table. An imperfect hash table can have key collisions. A key collision is the hash function mapping of different keys to the same position in a hash table. The worst-case lookup speed in an imperfect hash table is O(N) time, but far more typically is O(1), with O(m) time spent evaluating the hash
- There are no collisions of different keys in a trie.
- Buckets in a trie, which are analogous to hash table buckets that store key collisions, are necessary only if a single key is associated with more than one value.

- There is no need to provide a hash function or to change hash functions as more keys are added to a trie.
- A trie can provide an alphabetical ordering of the entries by key.

Tries do have some drawbacks as well:

- Trie lookup can be slower in some cases than hash tables, especially if the data is directly accessed on a hard disk drive or some other secondary storage device where the random-access time is high compared to main memory.
- Some keys, such as floating-point numbers, can lead to long chains and prefixes that are not particularly meaningful. Nevertheless, a bitwise trie can handle standard IEEE single and double format floating point numbers.
- Some tries can require more space than a hash table, as memory may be allocated for each character in the search string, rather than a single chunk of memory for the whole entry, as in most hash tables.

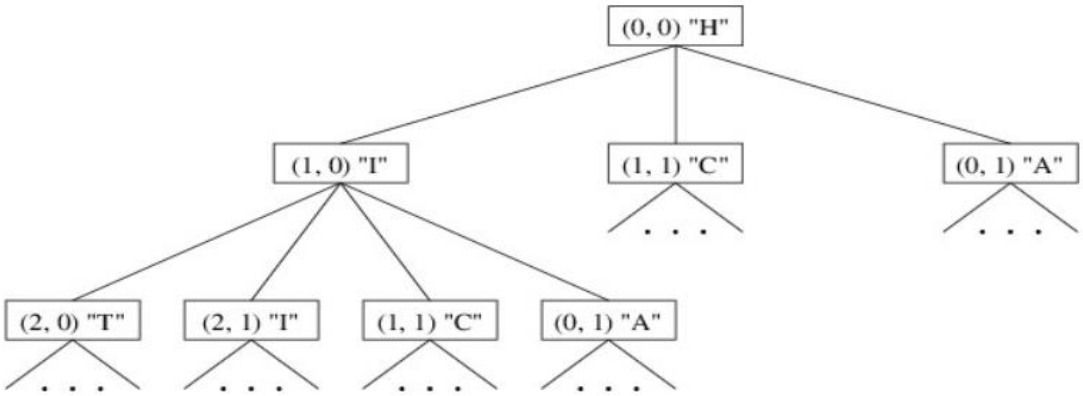


Figure 7: Example on long chains

3.3.3 How to use trie

Trie as a 2D array works similar to a DFA (Finite automata). Just assume that you have a variable named counter that starts from 0 and it tracks the total memory locations that you have used till now. Then if your 2d trie is like trie[i][j], it means that if you are at location numbered at i then if you are seeing the character j then what location number should I move to, the 2D trie stores this. So, the total size is N * K where N is the sum of length of all the strings and K is the size of the character set. Initially the trie is set to -1 i.e. if trie[i][j] is -1 then it means you can't perform any transition from state i if the next character is j. The variable counter is used to track how many locations you have used or simply it provides the index of the new node.

World Fastest Scrabble Program is based on lexicon trie:

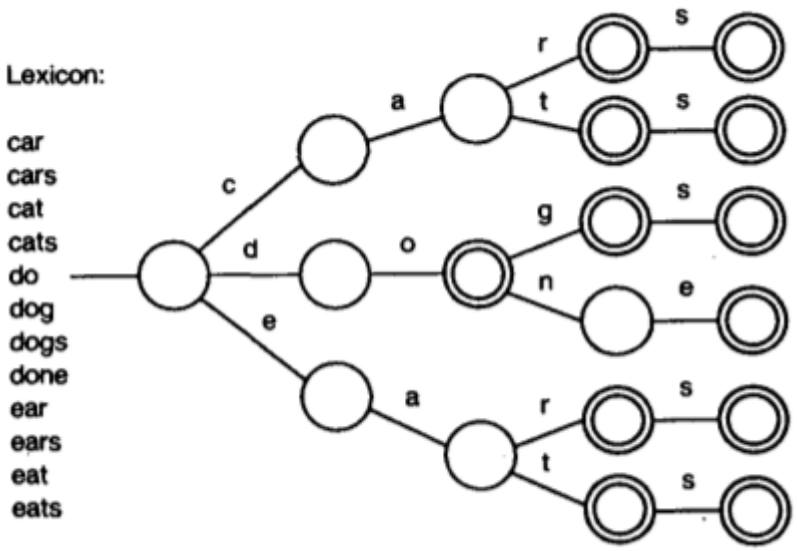


Figure 8: Lexicon Trie

3.4 Game Implementation

- **Artificial Intelligence of a Scrabble System**
    - Scrabble Bot
    - Artificial Intelligence of a Scrabble System

- **Task**
    - Create a comprehensive Scrabble system for a human player to compete against an agent.
    - Produce an Artificial Intelligence System that can determine possible words from a list of letters.
    - Develop an Open Source application easy for users to download and play.

- **Purpose Create a Scrabble agent to play against a human opponent**
    - Must be able to determine a word in a reasonable amount of time.
    - The bot will choose the best word point-wise with restrictions based on difficulty.
    - Agent must follow rules of Scrabble.
    - Word Choice.
    - Take into account board layout & special tiles.

- **Purpose Graphical User Interface**
    - Should be easy for a new player to learn and use
    - Playing should be intuitive
    - Aesthetically appealing
    - Other Functionality
    - Easy for a user to download and install the game
    - All words referenced against official Scrabble dictionary for correctness

- **Design: GUI > Interaction > Bot > Rules Engine**
    - The GUI receives interaction from the user, and in turn probes the bot for moves. The Bot contains the AI, and uses base functions from the Rules Engine to determine the rules it must follow.
    - Rules Engine
    - Contains the basic functions that determine whether a move follows the rules
        - Example: is [word] a valid word in the dictionary?
        - Example: what is the score of [word]?
    - Uses the SOWPODS standard scrabble dictionary

- **Implementation Board Graphical Structure building GUI**
    - Internal Representation of Board
    - Player
    - Generation of a hand
    - Finding the played word

- **Program Flow Board appears / updates on screen:**

    - Player moves (places tiles on board)
    - GUI registers move
    - GUI class probes bot for next move
    - Bot calculates (using brute force) best possible move
    - Bot sends to GUI
    - GUI registers on screen
    - Repeat from step 2 (Player's turn to move)
    - If no more tiles or space, game ends

- **Computer Bot Intelligence**

    - Checks occupied squares on the board
    - Finds various word combinations with given tiles
    - Checks if moves are valid
    - Ranks valid moves based on word length and score

- **Improving the Algorithm**

    - Two big options: DAWG or GADDAG.

        - DAWG: directed acyclic word graph
        - GADDAG: bi-directional acyclic word graph
    - Both offer tradeoffs for time and space
    - Each is significantly more powerful than brute force methods.

- GADDAG and DAWG:

GADDAG	DAWG
-bi-directional acyclic word graph -Pros: (when optimized) extremely fast, eliminates invalid words from even being considered. -Cons: storing the optimized data structure is ~5x bigger than the DAWG. -Very similar in form to the DAWG, but treats prefixes and suffixes in the same way. -Increased performance allows more time to make better moves.	-directed acyclic word graph -child: an appended letter. -next: an alternate letter. -Pros: low space requirement, reasonably fast, eliminates invalid words from even being considered. -Cons: Does not do well in multiple directions, treats prefixes differently than suffixes.

Table 1-2 GADDAG VS DAWG

- General Improvements
    - Balance use of vowels, consonants, and letters required for larger moves.
    - Predict opponent's hands based on what is on the board.
    - Avoid moves that make large bonus tiles easy to reach.

3.4.1 Programming Languages

After researching the tools used in similar projects we can recommend:

- C++ programming language for implementation because it is one of the fastest programming languages out there, so it is the best form optimizing performance.
- Python for the training algorithm. Although python is a slow language, we recommend using it for the training algorithm since it contains a lot of helper libraries and functions.
- Unity 3D&2D game engine to create the required Scrabble GUI because of its good compatibility with C++.
- We considered using Lisp language & Closure technique in our implementation, but decided against it due to the lack of knowledge and experience with them.

3.4.2 Libraries

- zeroMQ.
  ZeroMQ is a high-performance asynchronous messaging library, aimed at use in distributed or concurrent applications. It provides a message queue, but unlike message-oriented middleware, a ZeroMQ system can run without a dedicated message broker. The library's API is designed to resemble that of Berkeley sockets.
- RabbitMQ.

  RabbitMQ is an open-source message-broker software that originally implemented the Advanced Message Queuing Protocol and has since been extended with a plug-in architecture to support Streaming Text Oriented Messaging Protocol, Message Queuing Telemetry Transport, and other protocols.

- WebSockets/ws.

  WebSocket is a computer communications protocol, providing full-duplex communication channels over a single TCP connection. The WebSocket protocol was standardized by the IETF as RFC 6455 in 2011, and the WebSocket API in Web IDL is being standardized by the W3C.

3.4.3 Implemented Projects

- Wordsmith:
  Wordsmith plays by searching the board for all possible tile placements and then chooses the best one. It evaluates this not only based on how many points a particular move gives, but also by using heuristics to avoid technically higher-scoring moves which are probably detrimental in the long-run.

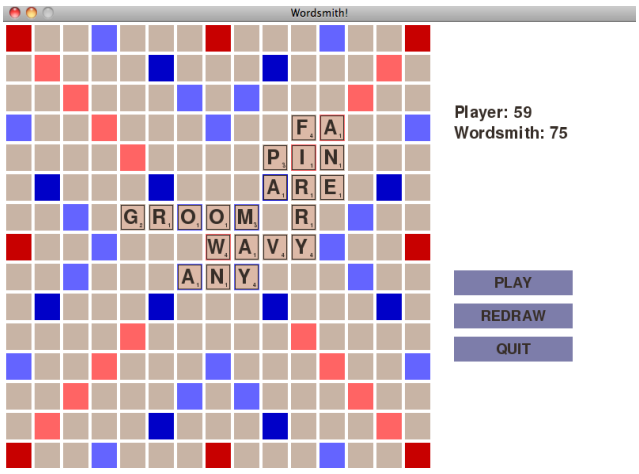


Figure 9: Wordsmith Game

1. Make a set of "slots" where all possible moves can be placed. Don't consider the individual tiles yet, just look at where it is possible to place tiles in a valid move and make a list of these.
2. For each slot, try placing all permutations of your tiles onto the board. For blanks, that also means trying every single one of the 26 possible letters.
3. For each possible play, check whether it forms all valid words (not only in the principle direction, but in all new crosswords) and calculate the points. Keep track of the highest scoring play while you proceed.

- This brute force algorithm is, obviously, very slow so how do we speed it up.

1. Scoring is complex, testing whether the main word is valid is easy (O (1) hash-table lookup, for the CS geeks). Only words that were valid along the principle direction would be scored thoroughly.
2. Before placing a blank, figure out if there are any words, which match that blank configuration. X_YGYK does not mean anything with any letter, so we can skip trying all 26 times.
3. Since blanks have zero points, we only need to figure out if one letter assignment is valid, since all others will result in the same score. WI_ is the same whether you played WIZ or WIN (provided all crosswords are also valid in both cases).
4. The algorithm for calculating possible slots can produce duplicates, so making sure every slot is unique can cut the processing by 30-50%.
5. Forcing a certain time limit and start the search with the shorter words.

3.5 Research Results

And here is what we decided to use in our project upon the research team's suggestions.

3.5.1 Programming Language

- Using C++ Programming Language because it is very fast programming language.

3.5.2 Approaches

- Maven Approach.
  - We choose this approach as our project mainly dependent on simulation.
  - our heuristic function is needed to be improved and due to the limited time, so we couldn't use Quackle leaf evaluation.

3.5.3 Project Main Modules

- GUI Module.
- Game Brain Module.
- Game Server Module.

Section 4: GUI

4. GUI

4.1 Phase 1: Preparation

One of the important (if not the most) factor of enjoying games is in its graphics and GUI. So, we decided to implement a simple yet beautiful design to our Scrabble game. While doing the market research, we found out that the Unity engine is one of leading platforms in the industry while also being easy to learn so we went with it.

4.2 Phase 2: Implementation

After making surveys and consulting few game developer's opinion regarding our game design, we decide to change our implementation to fit the easy yet catching design integrating beautiful wood materials as base for our board with dark and light colors to catch the users' attention during the game, with all those factors we came up with our Wood Board Design. Figure 7, Wehrmacht In-Game Design (p.28)

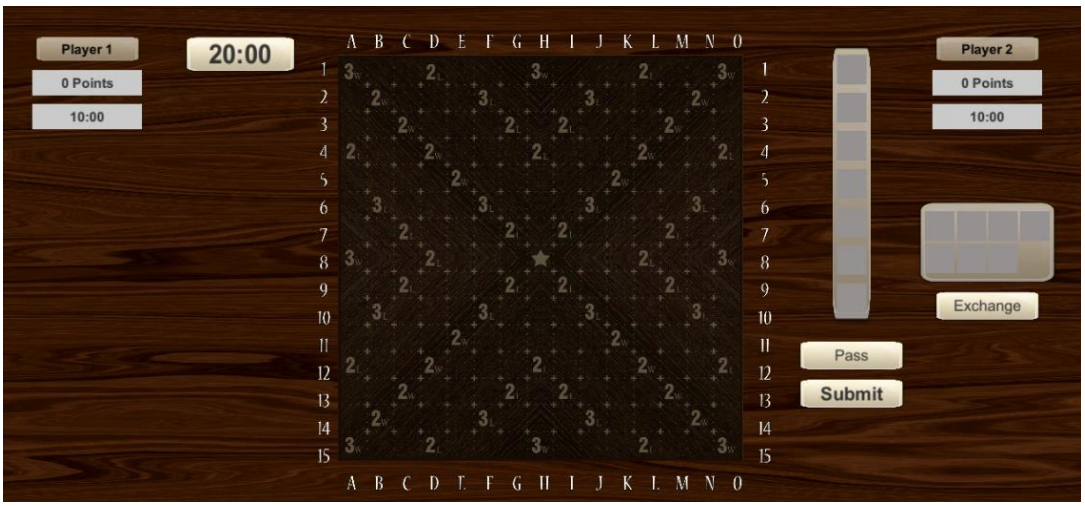


Figure 10: Wehrmacht In-Game Design

Using Unity Frame Work and C# programing language, the game succeeded in fulfilling the desired outcome and game flow that we hoped for, which are the following: -

- Player Area to carry player name and score and total played time.
- Timer limit of the whole game.
- Tiles shelf to carry the player tiles.
- An Exchange shelf to carry the tiles that the user might change his tiles with it.
- Opponent Area to carry opponent name and score and total played time.
- A pass button if the player wishes to forsake his turn.
- A submit button if the player decided to submit his gameplay after placing the tiles on board.
- The board itself which will carry both player's gameplay and will be used to show the game itself.

As our in-game mode seemed simple but yet catching, we should remember that Wehrmacht scrabble game support different modes e.g. AI mode, Training mode but in order for our users to have the ability to access these features, we provided our users with an easy and simple designed main screen. Figure 8, Main screen (p.29) that also inherent the board wood design to keep the design flow throw the whole game experience.

Figure 11: Wehrmacht Main Screen

Using also Unity Frame Work and C# programing language, our main screen allows the user to choose the game modes or just quite the game.

But before our users are able to experience the game, they must provide us with their name in order to help them keep track of their scores during the game, by going back to our in-game mode. Figure 1, Wehrmacht In-Game Design (p.28). we find that our user fields take the user name e.g. player 1, player 2, and for that feature we added a welcome screen. Figure 9, Wehrmacht Welcome Screen (p.29). to help our user to insert his name in the easiest way possible and help us provide our user with his score during the game.

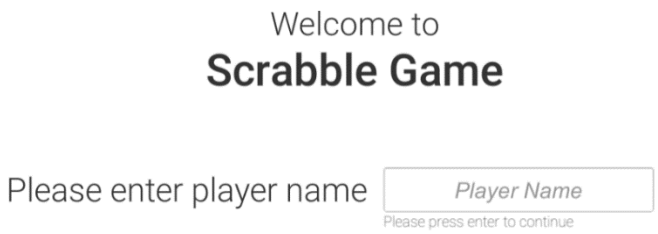


Figure 12: Wehrmacht Welcome Screen

Now after providing our user with all these features, and providing him with the best experience of playing scrabble game, we didn't forget to thank our user for his time playing our game and trying our scrabble approach. For that we also added an End game screen. Figure 10, Wehrmacht End Game Screen (p.30). where we thank our user for playing Wehrmacht scrabble game.

Figure 13: Wehrmacht Ending Screen

With all these features we are positive that our user will have the best experience playing scrabble and it will attract many players who seeks to play or understand scrabble game as well as fulfilling its role as base for our Artificial intelligent agent to play against other players.

But we can't say that everything is working fine without understanding the goal of this game, and for that we will take about the goals and the constraints of our game that the GUI supports and we will demonstrate this with a sample code from our GUI team:

- Goal: either one of the two players manage to finished his bag of tiles before the other one.
- Constraint: time limit of the game which is 20 minutes, after that the game will end and the player with the highest score wins.

This may sound hard to implement but thanks to GUI team we managed to implement an easy function to do that. **Figure** 11, Wehrmacht countdown function (p.31).

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;

public class CountdownTimer : MonoBehaviour
{
    public Text mTimerText;
    // Time in seconds
    public float mTime = 1200;

    //public GameObject mGameOver; // Assign in inspector
    public Scene mGameOver;
    void Start()
    {
        StartCoundownTimer();
    }

    void StartCoundownTimer()
    {
        if (mTimerText != null)
        {
            // Time in seconds
            mTime = 1200;
            mTimerText.text = "20:00";
            // Keep updating time
            // Will start at 0 seconds and repeat every 0.01667 seconds
            InvokeRepeating("UpdateTimer", 0.0f, 0.01667f);
        }
    }

    void UpdateTimer()
    {
        if (mTimerText != null)
        {
            mTime -= Time.deltaTime;
            if (mTime <= 0)
            {
                SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
                return;
            }
            string minutes = Mathf.Floor(mTime / 60).ToString("00");
            string seconds = (mTime % 60).ToString("00");
            mTimerText.text = minutes + ":" + seconds;
        }
    }

}
```

Figure 14: Wehrmacht GUI timer function

The code is simple and easy to understand and most importantly it preforms its required functionality in an efficient manner, then what about the score, how does our team calculate it, after all that's the main goal of the game, and for that it will be explained in Section 5, Implementation (P.34).

Section 5: Implementation

5. Implementation

5.1 Implementation Main Division

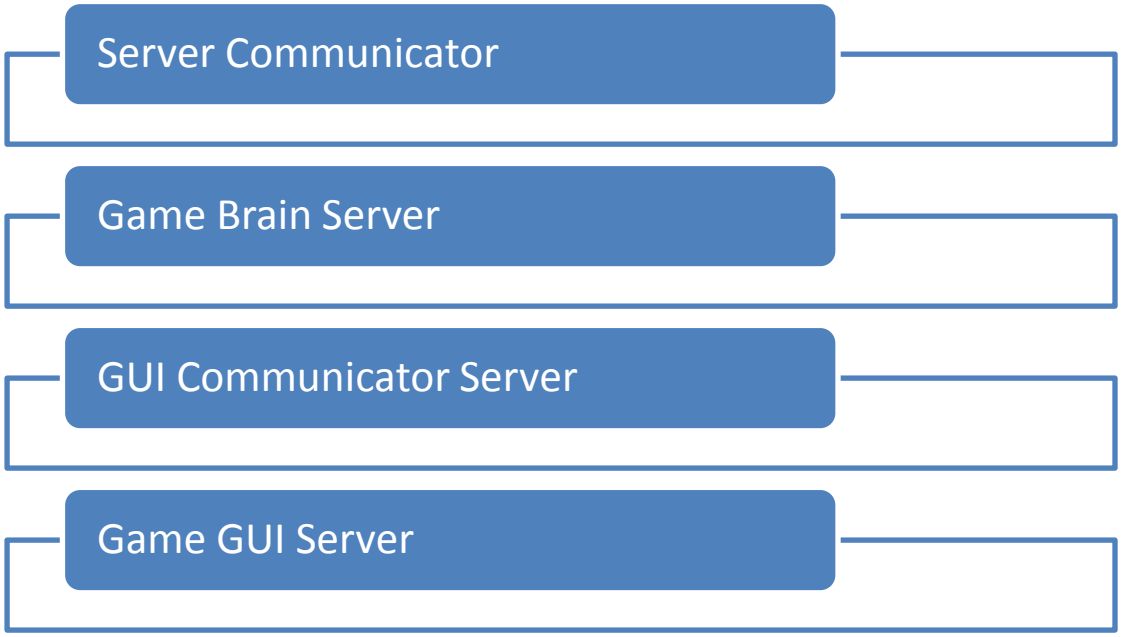


Figure 15: Implementation Division

5.1.1 Server Communicator

This server communicates with the game brain server to

- Provide the game with:
  - Tiles
  - The move we want to do
  - The other player's move.
- Keep up with the data that should be updated (time, tiles, score etc.).

5.1.2 Game Brain Server

By taking some of the data (tiles and board) provides us with the best move.

- This server is divided into three modules:

5.1.2.1 Move Generation

Given to it lack of tiles and the board state. And using GADAG and TRI data structures. It generates a list of all possible moves that can be done from our dictionary (Sowpods).

- For each move, it gives some information
  - Stop position (row and column) of the first letter.
  - This word is horizontal or vertical.
  - The score of the word.
  - This word is bingo or not.

5.1.2.2 Game Evaluation

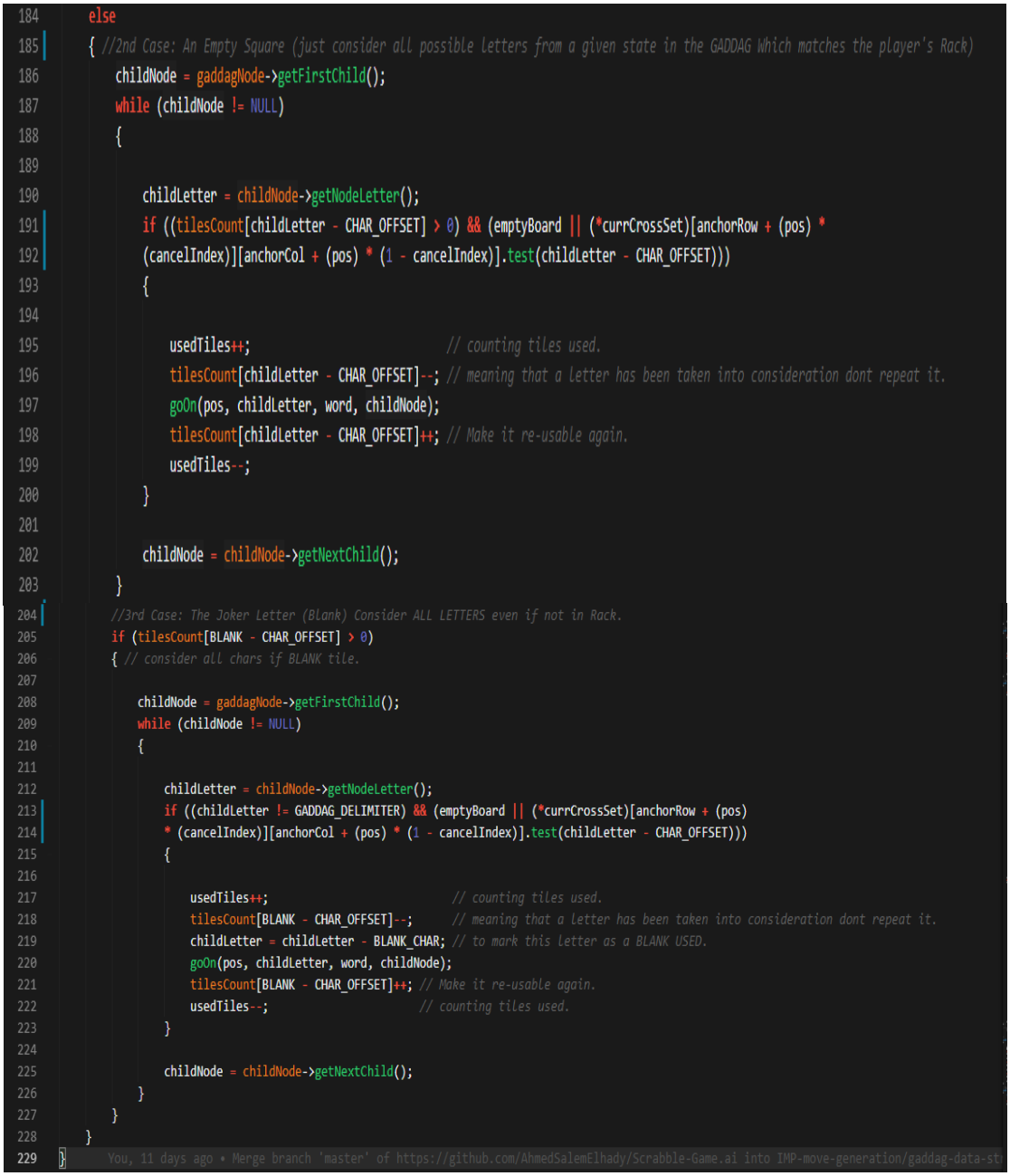
- This game is sub-divided into three main phases
  - Mid game.
  - Pre-end.
  - End game.
    - For the mid and pre-game phases:

Those two phases have some common steps and identities. Both use game simulation for Monte-Carlo. They are using "Maven-Approach" instead of "Quackle-Approach" because Maven has only two plies but Quackle has three plies and this is too much wasted time. As we don't need the 3rd ply.

- The two-ply simulation:
  - The move of our agent.
  - The player's response.

But for doing this simulation we need to know two main states of our game which is the board state and the player's rack in order to generate all the possible moves for our mid and pre-game phase and for that we decided to an implemented GADDAG data structure. **Figure** 16, Wehrmacht move generation function (P.34).

```
164  //This Gen funtion first function in Move Generation Algo.
165  void WordGenerate::gen(int pos, string word, Node *gaddagNode)
166  {
167      char childLetter; // init of needed paramters.
168      char boardLetter;
169      Node *childNode;
170      bitset<27> defaultCrossSet; // Pre-calculated To ensure valid connection of words when constructing horiz/vertical move.
171
172      //1st Case: When Character is Already onBoard : (Find It in the GADDAG From The pos you are on pass it to continue)
173      if (board->hasaTile(anchorRow + (pos) * (cancelIndex), anchorCol + (pos) * (1 - cancelIndex)))
174      {
175          boardLetter = board->getTileAtPosition(anchorRow + (pos) * (cancelIndex), anchorCol + (pos) * (1 - cancelIndex));
176           // a More General Approch.
177          childNode = gaddagNode->findChildChar(boardLetter);   // find letter in GADDAG.      You, a few seconds ago • Uncommitted changes
178          if (childNode == NULL)
179          {
180              return;
181          }
182          goOn(pos, boardLetter, word, childNode);
183      }
184      else
185      { //2nd Case: An Empty Square (just consider all possible letters from a given state in the GADDAG Which matches the player's Rack)
186          childNode = gaddagNode->getFirstChild();
187          while (childNode != NULL)
188          {
```



```
184     else
185     { //2nd Case: An Empty Square (just consider all possible letters from a given state in the GADDAG Which matches the player's Rack)
186         childNode = gaddagNode->getFirstChild();
187         while (childNode != NULL)
188         {
189
190             childLetter = childNode->getNodeLetter();
191             if ((tilesCount[childLetter - CHAR_OFFSET] > 0) && (emptyBoard || (*currCrossSet)[anchorRow + (pos) *
192             (cancelIndex)][anchorCol + (pos) * (1 - cancelIndex)].test(childLetter - CHAR_OFFSET)))
193             {
194
195                 usedTiles++;                          // counting tiles used.
196                 tilesCount[childLetter - CHAR_OFFSET]--; // meaning that a letter has been taken into consideration dont repeat it.
197                 goOn(pos, childLetter, word, childNode);
198                 tilesCount[childLetter - CHAR_OFFSET]++; // Make it re-usable again.
199                 usedTiles--;
200             }
201
202             childNode = childNode->getNextChild();
203         }
204         //3rd Case: The Joker Letter (Blank) Consider ALL LETTERS even if not in Rack.
205         if (tilesCount[BLANK - CHAR_OFFSET] > 0)
206         { // consider all chars if BLANK tile.
207
208             childNode = gaddagNode->getFirstChild();
209             while (childNode != NULL)
210             {
211
212                 childLetter = childNode->getNodeLetter();
213                 if ((childLetter != GADDAG_DELIMITER) && (emptyBoard || (*currCrossSet)[anchorRow + (pos)
214                 * (cancelIndex)][anchorCol + (pos) * (1 - cancelIndex)].test(childLetter - CHAR_OFFSET)))
215                 {
216
217                     usedTiles++;                          // counting tiles used.
218                     tilesCount[BLANK - CHAR_OFFSET]--;    // meaning that a letter has been taken into consideration dont repeat it.
219                     childLetter = childLetter - BLANK_CHAR; // to mark this letter as a BLANK USED.
220                     goOn(pos, childLetter, word, childNode);
221                     tilesCount[BLANK - CHAR_OFFSET]++; // Make it re-usable again.
222                     usedTiles--;                      // counting tiles used.
223                 }
224
225                 childNode = childNode->getNextChild();
226             }
227         }
228     }
229 }     You, 11 days ago • Merge branch 'master' of https://github.com/AhmedSalemElhady/Scrabble-Game.ai into IMP-move-generation/gaddag-data-st
```

Figure 16: Wehrmacht move generation function

But knowing only the player rack and the current board state is not enough for the agent to make a decision or to give the user an advice about his move, what we need is to predict our opponent rack as well to provide our simulation with enough information to support the user/AI decision and for that we implemented a function that generate rack of the opponent to be used in simulation. **Figure** 17, Wehrmacht opponent rack generation (P.36).

```
  //Function to create the rack to be used:
vector<char> OpponentRack::RackGenerator(unordered_map<char, int> Letters) {···
  }
```

```
//Function to create the rack to be used:
vector<char> OpponentRack::RackGenerator(unordered_map<char, int> Letters) {
  OpponentRack::Rack.clear();
  //Taking a copy of letters vector:
  std::vector<std::pair<char,int>> Copy_Letters = copy(Letters);
  //-------------------------------------------------------------------------------------------------------//
  //Remove any element with prob of zero:
  OpponentRack::RemoveZeroProbabilities(Copy_Letters);
  //Data Members where data will be returned in.
  vector<double> Letter_Probabilities; //Vector containing probability of finding a letter in a rack.
  int Number_Of_Remaining_Letters; //Number of Remaining Letters in the Bag and Opponent Hand.
  //Get Number of Remaining Letters that aren't on board or in my hand:
  Number_Of_Remaining_Letters = OpponentRack::LeftLetters(Copy_Letters);
  //Get Probability of each character being drawn from the bag:
  Letter_Probabilities = OpponentRack::GetProbabilities(Copy_Letters, Number_Of_Remaining_Letters);
  //std::mt19937 Generator;
  std::random_device Generator;
  //Populate one rack:
  for (int i = 0; i < 7;i++) {
    //Function that Generates integers based on their different probabilities:
    std::discrete_distribution<int> Distribution(Letter_Probabilities.begin(),Letter_Probabilities.end());
    //Distribution generates numbers from 0 --> 26:
    int Letter_Ascii = Distribution(Generator);
    //Push the selected letter into the suggested opponent rack:
    if (OpponentRack::Rack.size() >= 7)
      return OpponentRack::Rack;
    if (Copy_Letters[Letter_Ascii].second > 0) {
      OpponentRack::Rack.push_back(Copy_Letters[Letter_Ascii].first);
      //Decrement remaining Number of all characters:
      Number_Of_Remaining_Letters--;
      //Decrement Number of remaining instances of the character:
      Copy_Letters[Letter_Ascii].second--;
    }
    else {
     RemoveZeroProbabilities(Copy_Letters);
    }
    //Update New Probabilities of Letters after removing the current letter:
    Letter_Probabilities = OpponentRack::GetProbabilities(Copy_Letters, Number_Of_Remaining_Letters);
  }
  //Return the Generated Rack:
  return OpponentRack::Rack;
}
```

Figure 17: Wehrmacht opponent rack generation

Now you are probably wondering about all of those includes that you have seen so far in our code examples, well our game consists of tiles and a single board so we implemented classes to provide use with information about tiles, rack and board at any needed part of the code, e.g. Get Tiles, Get Rack, Add tiles, Exchange Tiles. **Figure** 18, Wehrmacht Tiles class implementation (P.37).

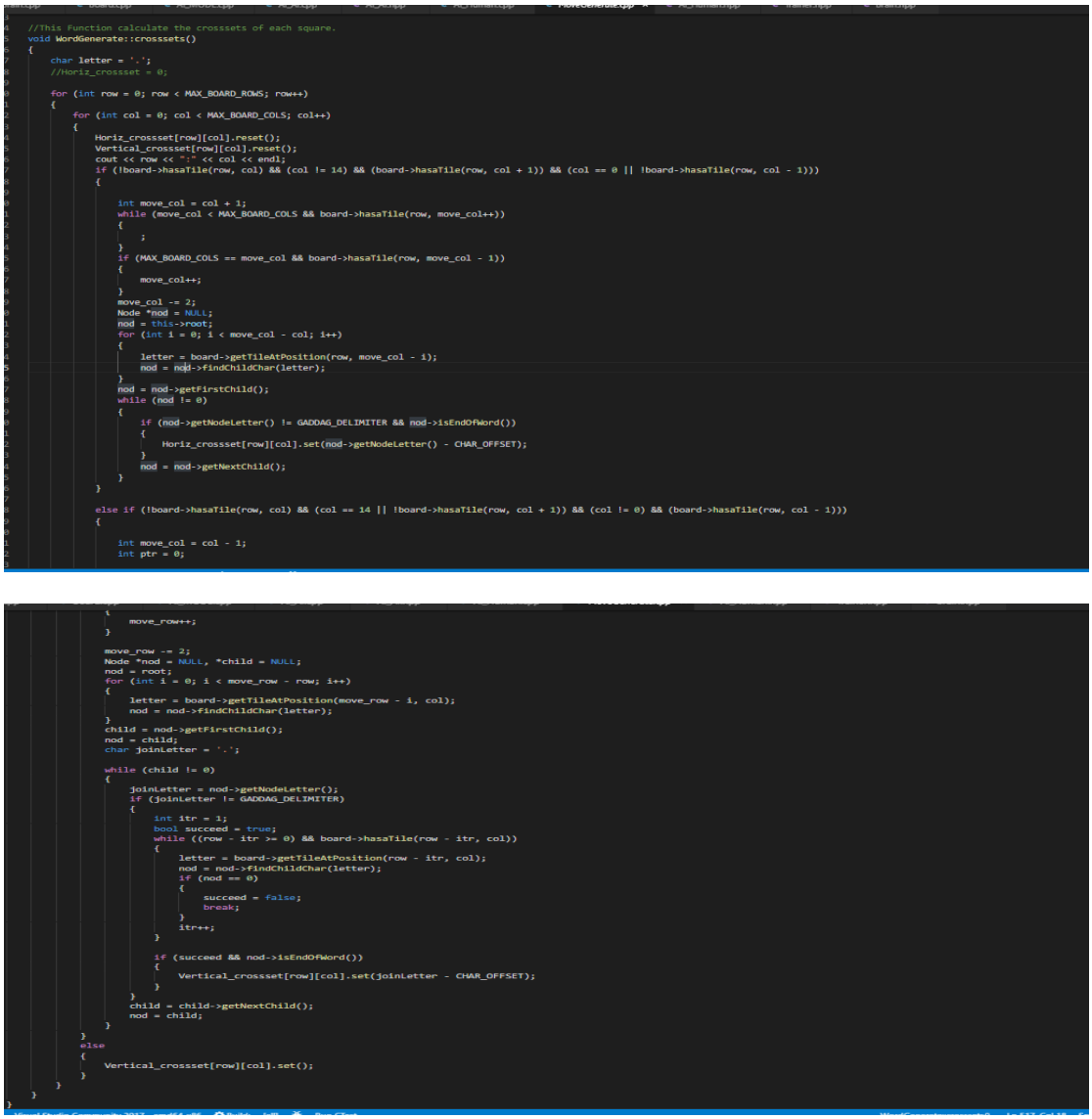
Tiles.h

```h
#pragma once

#include <vector>


class Tiles
{
private:


    std::vector<char> RackTiles;//Vector Containing character which is tiles
    Tiles();//Default Constructor
    static Tiles* TilesInst_;
public:


   static Tiles* getTiles();//this function to get current rack of tiles
   std::vector<char> getRackTiles();//This function return rack of tiles
   bool addTiles(char tile);//this function to add a tile to the rack
   bool exchngTiles(char Current,char New);//this function to exchange tiles with server

};
```

```cpp
//This Function return tiles
Tiles* Tiles::getTiles()
{

  if(TilesInst_==nullptr)
  {
    TilesInst_=new Tiles;
  }

  return (TilesInst_);
}

//This Function add tiles to the rack
bool Tiles::addTiles(char tile)
{

        if(RackTiles.size()<7)//if rack of tiles less than 7 then you can add other wise operation fail with false return
        {
          RackTiles.push_back(tile);
          return true;
        }
return false;
}

//This Function exchange tiles
bool Tiles::exchngTiles(char Current,char New)
{

std::vector<char>::iterator itr= find(RackTiles.begin(), RackTiles.end(), Current);
//if the current tile is found in the rack of tiles exchange is successful other wise fail with false return

if (  itr!= RackTiles.end() )
   {
        //here we may take the current tile to give it to server befor deleting it from the rack of tiles
        RackTiles.erase(itr);//remove the current tile
        RackTiles.push_back(New);//add the new one

        return true;
   }
return false;
}
```

Figure 18: Wehrmacht Tiles class implementation

But that's not quite enough, yes it may seem complicated but since we have 15 X 15 board, we have a huge number of moves that we can do but since one of the scrabble rules is that the new placed tiles should be next to the ones that already have been placed. We need a function to know the possible tiles that can be placed around the tiles already played for our simulation to decide on the best move to make. **Figure** 19, Wehrmacht tiles placement class (P.38).



```
//This Function calculate the crosssets of each square.
void WordGenerate::crosssets()
{
    char letter = '.';
    //Horiz_crossset = 0;

    for (int row = 0; row < MAX_BOARD_ROWS; row++)
    {
        for (int col = 0; col < MAX_BOARD_COLS; col++)
        {
            Horiz_crossset[row][col].reset();
            Vertical_crossset[row][col].reset();
            cout << row << ":" << col << endl;
            if (!board->hasaTile(row, col) && (col != 14) && (board->hasaTile(row, col + 1)) && (col == 0 || !board->hasaTile(row, col - 1)))
            {

                int move_col = col + 1;
                while (move_col < MAX_BOARD_COLS && board->hasaTile(row, move_col++))
                {
                    ;
                }
                if (MAX_BOARD_COLS == move_col && board->hasaTile(row, move_col - 1))
                {
                    move_col++;
                }
                move_col -= 2;
                Node *nod = NULL;
                nod = this->root;
                for (int i = 0; i < move_col - col; i++)
                {
                    letter = board->getTileAtPosition(row, move_col - i);
                    nod = nod->findChildChar(letter);
                }
                nod = nod->getFirstChild();
                while (nod != 0)
                {
                    if (nod->getNodeLetter() != GADDAG_DELIMITER && nod->isEndOfWord())
                    {
                        Horiz_crossset[row][col].set(nod->getNodeLetter() - CHAR_OFFSET);
                    }
                    nod = nod->getNextChild();
                }
            }

            else if (!board->hasaTile(row, col) && (col == 14 || !board->hasaTile(row, col + 1)) && (col != 0) && (board->hasaTile(row, col - 1)))
            {

                int move_col = col - 1;
                int ptr = 0;
```



```
                {
                    move_row++;
                }

                move_row -= 2;
                Node *nod = NULL, *child = NULL;
                nod = root;
                for (int i = 0; i < move_row - row; i++)
                {
                    letter = board->getTileAtPosition(move_row - i, col);
                    nod = nod->findChildChar(letter);
                }
                child = nod->getFirstChild();
                nod = child;
                char joinLetter = '.';

                while (child != 0)
                {
                    joinLetter = nod->getNodeLetter();
                    if (joinLetter != GADDAG_DELIMITER)
                    {
                        int itr = 1;
                        bool succeed = true;
                        while ((row - itr >= 0) && board->hasaTile(row - itr, col))
                        {
                            letter = board->getTileAtPosition(row - itr, col);
                            nod = nod->findChildChar(letter);
                            if (nod == 0)
                            {
                                succeed = false;
                                break;
                            }
                            itr++;
                        }

                        if (succeed && nod->isEndOfWord())
                        {
                            Vertical_crossset[row][col].set(joinLetter - CHAR_OFFSET);
                        }
                    }
                    child = child->getNextChild();
                    nod = child;
                }
            }
            else
            {
                Vertical_crossset[row][col].set();
            }
        }
    }
}
```

Figure 19: Wehrmacht tiles placement class

Now that we have our tiles and rack, what about our AI; of course, our AI module was implemented in class on its own to be used later in trainer and Ai mode using functions such as e.g. Set Communicator, Set Tiles, Set Board. **Figure 20,** Wehrmacht AI Human class (P.39).

AI_Human.hpp

```
#pragma once

#include "../AI_MODE/AI_MODE.hpp"
#include "../SharedClasses/TrainerComm.hpp"

class AI_Human
{
private:
    std::string messageToHuman;
    Board *MyBoard;
    vector<char> *HumanTiles;
    unordered_map<char, int> *Bag;
    BoardMask *BoardStatus;
    BoardToGrammer b2g;

    AiMode *AI_Agent;
    TrainerComm *Communicator;

    void exchange(std::vector<char> *, char, char);

public:
    AI_Human();
    void exchangeTiles(std::vector<char> *, char);
    bool SetBag(unordered_map<char, int> *);
    bool SetTiles(vector<char> *);
    bool SetCommunicator(TrainerComm *);
    bool SetBoard(Board *MyBoard);
    bool SetAgent(AiMode *AI_Agent);
    Move *DoWork(bool, int, LoadHeuristics *);
    std::string getString() const
    {
        return this->messageToHuman;
    }
};
```

AI_Human.cpp

```cpp
bool AI_Human::SetCommunicator(TrainerComm *communicator)
{
    try
    {
        this->Communicator = communicator;
        return true;
    }
    catch (const std::exception &e)
    {
        std::cerr << e.what() << '\n';
    }

    return false;
}

bool AI_Human::SetTiles(vector<char> *tiles)
{
    try
    {
        this->HumanTiles = tiles;
        this->AI_Agent->setTiles(*tiles);
        return true;
    }
    catch (const std::exception &e)
    {
        std::cerr << e.what() << '\n';
    }

    return false;
}

bool AI_Human::SetAgent(AiMode *AI_Agent)
{
    AI_Agent->setBagPointer(this->Bag);
    AI_Agent->setBoardToGrammar(b2g);
    this->AI_Agent = AI_Agent;
    return true;
}

bool AI_Human::SetBoard(Board *board)
{
    try
    {
        this->MyBoard = board;
        return true;
    }
    catch (const std::exception &e)
    {
        std::cerr << e.what() << '\n';
    }

    return false;
}
```

Figure 20: Wehrmacht AI Human class

- For the end-game phase:

  We know that we are in this phase when there are only 7 tiles left or there is no tile in the bag.

  In this phase we know the opponent's tiles in its bag by 75%. Almost know what tiles it have.se we can decide the best move for our agent and which move will make him lose the game using a "Greedy-Approach".

➢ Our evaluation function approximation is the "Rack-Leave-Evaluation".

- Rack-Leave Evaluation function:

  In this function we don't depend only on the move's score but also, we depend on what will remain in the rack after the current move.

  It's preferred that when the coming play has a better score and for that our team worked hard to write a function that calculate a suggested move score if it placed on a board. Figure 21, Wehrmacht Score Calculator (P.40).

```
#include "../Board/Board_and_tiles/Board_and_tiles/BoardConst.h"

int BoardCommunicator::calculateScore(string suggestedMove, int row, int col, bool horizontal)
{
    int intersectionScore = 0;
    int TileValues[26] = {1, 3, 3, 2, 1, 4, 2, 4, 1, 8, 5, 1, 3, 1, 1, 3, 10, 1, 1, 1, 1, 4, 4, 8, 4, 10};
    int offsit = row + col * 15;
    char charOffsit = ' ';
    bool Wsx3 = false;
    bool Wsx2 = false;
    int Wsx3M = 1; // multiplyer
    int Wsx2M = 1;
    int WordScore = 0;
    BoardMask CurrentBoard = BoardPtr->getBoardStatus();
    int preOffsit = (horizontal) ? offsit - 1 : offsit - 15;// to detect intersection in a suggestedMove
    int posOffsit = (horizontal) ? offsit + 1 : offsit + 15;
    int boarderCheck = (horizontal) ? row : col; // above or bellow and vice versa.

    bool blank = false;
    int x;
    for (std::size_t i = 0; i < suggestedMove.length(); ++i)
    {
        blank = (suggestedMove[i] >= 65 && suggestedMove[i] <= 90) ? true : false;
        charOffsit = (suggestedMove[i] >= 0 && suggestedMove[i] <= 25) ? 0 : 97;
        if ((suggestedMove[i] >= charOffsit && suggestedMove[i] <= (charOffsit + 26 - 1)) || blank)
        {
            if (!blank)
            {
                if (LetterScoreMultiplyBy2.getBit(offsit) && !CurrentBoard.getBit(offsit))
                {
                    WordScore += TileValues[suggestedMove[i] - charOffsit] * 2;
            {
                if (LetterScoreMultiplyBy2.getBit(offsit) && !CurrentBoard.getBit(offsit))
                {
                    WordScore += TileValues[suggestedMove[i] - charOffsit] * 2;
                }
                else if (LetterScoreMultiplyBy3.getBit(offsit) && !CurrentBoard.getBit(offsit))
                {
                    WordScore += TileValues[suggestedMove[i] - charOffsit] * 3;
                }
                else
                {
                    WordScore += TileValues[suggestedMove[i] - charOffsit];
                }
            }
            if ((WordScoreMultiplyBy2.getBit(offsit)) && !CurrentBoard.getBit(offsit)) // the start square
            {
                if (offsit == 7 + 15 * 7)
                {
                    if (!CurrentBoard.getBit(7 + 15 * 7))
                    {
                        Wsx2 = true;
                        Wsx2M *= 2;
                    }
                }
                else
                {
                    Wsx2 = true;
                    Wsx2M *= 2;
                }
            }
            else if (WordScoreMultiplyBy3.getBit(offsit) && !CurrentBoard.getBit(offsit))
```

```
                    intersectionScore += BoardPtr->calculateScore(offsit, horizontal, suggestedMove[i] + offset);
                }
            }
            blank = false;
        }
        //incremntal
        if (horizontal) // row
        {
            offsit = offsit + 15;
            preOffsit += 15;
            posOffsit += 15;
        }
        else
        {
            offsit++; //  move in coloum
            preOffsit++;
            posOffsit++;
        }
    }
    if (Wsx2)
    {
        WordScore = WordScore * Wsx2M;
    }
    if (Wsx3)
    {
        WordScore = WordScore * Wsx3M;
    }
    WordScore += intersectionScore;
    return WordScore;
}
```

Figure 21: Wehrmacht Score Calculator

Now before we move to our decision about what move to choose from, going back to our GUI section, Section 4 (P.28). we mentioned a timer function, but where does our time come from, how do we keep track of our game time in our back end of the game.

The answer for those questions is simple, a timer class, Figure 22, Wehrmacht timer class (P.41). which sends the server time to our GUI and keep track of the game time to be used in our logic in the back-end side.

```
class TimerGUI
{
private:
    TrainerComm *Communicator;
    bool resetted;
    bool running;
    unsigned long beg;
    unsigned long end;
    unsigned long finalendtime;

public:
    TimerGUI(TrainerComm *, unsigned long);
    void start();
    void stop();
    void reset();
    bool isRunning();
    unsigned long getTime();
    bool isOver(unsigned long seconds);
    void SendTime();
};
```

Figure 22: Wehrmacht timer class

5.1.2.3 Move decision

From the previous two simulated games, here we decide which move is better.

- In this stage we start to take actions
  - Send to the server to decide what will send for the other servers.
  - Send to the server the enough information to decide how it will represent the screen.

5.1.3 GUI Communicator Server

- Computer VS computer
  - Handovers our agent and the other agent's plays to represent to the user.
- Computer VS human
  - Takes the human play from the GUI and gives a response message:
    - When the player does the same move as the client sent to him, the client responds with (Congratulations!).
    - When the player does a better move, the client responds with (Congratulations, your play is the best!).
    - When the player does a bad move (you could do it in a better way!).

5.1.4 Game GUI Server

Represents everything in the game for the player. Implemented using unity.

Figure 23: Decision flow

Section 6: Communication

6. Communication

6.1 Send to the server

6.1.1 Game to server client

- This client communicates with the server to provide it with:
  - Game brain.
  - The flow of data control from the server to the game brain.
- Implemented with C++ programming language Using WebSocket library to communicate with the server.
- Used Callback interfaces to be able to respond to the server when data is received from the game brain.

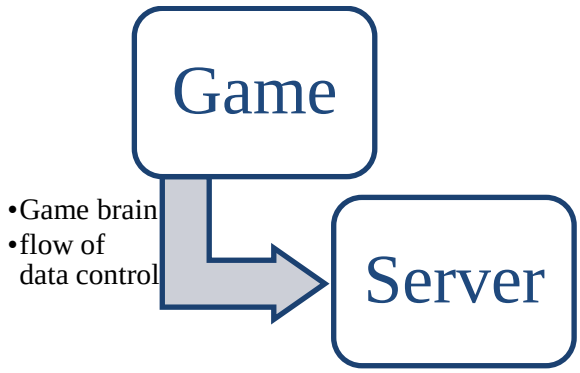


Figure 24: Game to server client

6.1.2 Game to GUI client

- Sends the minimal data needed to be rendered for users.
- Implemented with C++ programming language using ZeroMQ library to
  - To open message passing interface with the C# programming language.
- Used Callback interfaces to be able to respond to the server when data is received from the game brain.

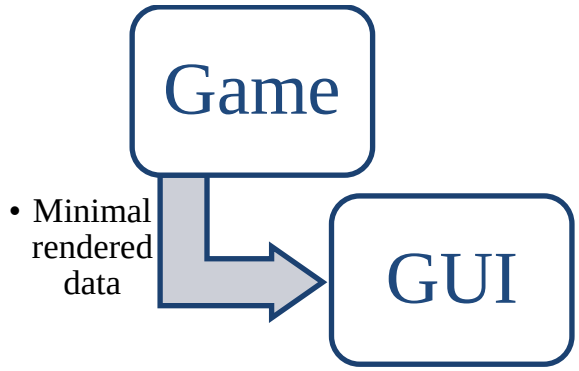


Figure 25: Game to GUI client

6.2 Receive from the server

6.2.1 GUI to Game client

- **AI mode:**
  - During this mode the client doesn't send anything to the game.
- **Teacher mode:**
  - In this mode the client provides the game with:
    - The player's name.
    - Plays in the trainer mode.
    - Play Response
      - Submission.
      - Pass.
      - Challenge.
    - Gives a response to human when one of these scenarios happens:
      - When the player does the same move as the client sent to him, the client responds with (Congratulations!).
      - When the player does a better move, the client responds with (Congratulations, your play is the best!).
      - When the player does a bad move (you could do it in a better way!).

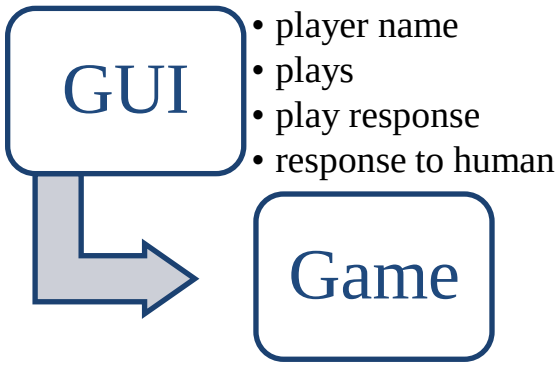


Figure 26: GUI to game client

Section 7: Integration

7. Integration

The integration team have used Unity for the GUI implementation and C# for the game logic.

Our integration team was trying to keep the project as modular as possible. They allowed other teams work without worrying about clashes or problem with other teams and this made them more comfortable and under less stress.

7.1 Design Pattern

In our project, we used strategy pattern for game flow as there are 3 phases in this game as described before in 2.5.2. so, each one has a different strategy in planning and evaluating the game.

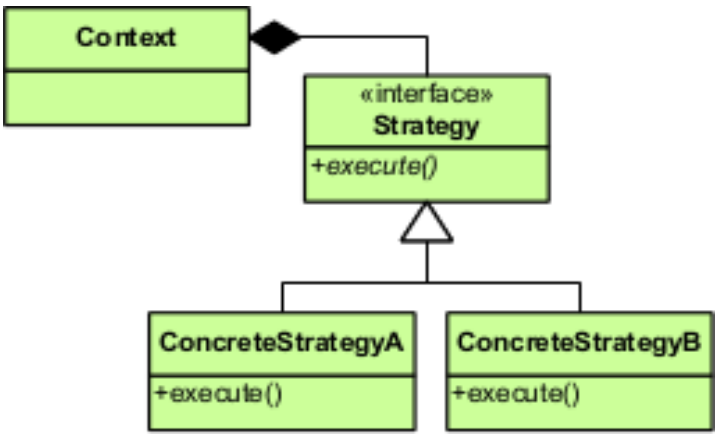


Figure 27: Strategy Pattern

7.1.1 About strategy pattern:

It is a behavioral software design pattern that enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use.

Strategy lets the algorithm vary independently from clients that use it. Strategy is one of the patterns included in the influential book Design Patterns by Gamma et al. that popularized the concept of using design patterns to describe how to design flexible and reusable object-oriented software. Deferring the decision about which algorithm to use until runtime allows the calling code to be more flexible and reusable.

7.2 Levels of Integration

- We have passed through three levels of integration:
  - Make sure that any module is built without any syntax or runtime errors.
  - Any module in this project is logically correct. And there will not be any clashes with other modules.
  - On integration time, there are not any differences in C++ and C# versions. And this will not cause any problem to the integrated modules during runtime.

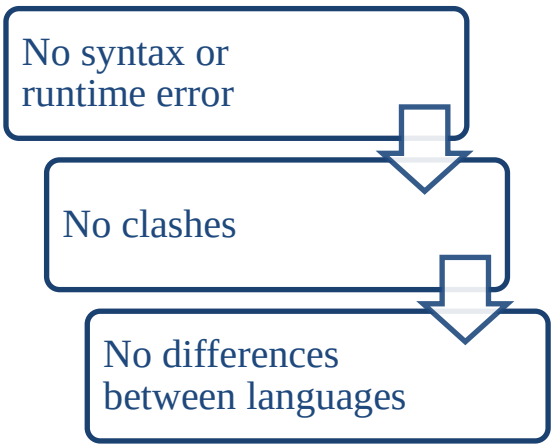


Figure 28: Levels of integration

Section 8: Testing

8. Testing

```
MCTreeNode* MCTSearch:: SelectBestMove(MCTreeNode* Parent){ ... }
```

Tester	Ali Ahmed Tolba
Input	Parent Children are Moves that are not visited Parent->Moves[i].moveScore={132,410,250}
Output	i=1
Comment	Correct Result as the second move has the highest score

```
void MCTreeNode::expandMidGame(){ ... }
```

Tester	Seif El Dien Adel
Precondition	PossibleMoves stack is not NULL
Output	Popped Possible Move is added to children
Comment	Correct Result as the tree is now expanded and the new move is added to the children of the current Monte Carlo Tree Node.

Section 9: Tools Used

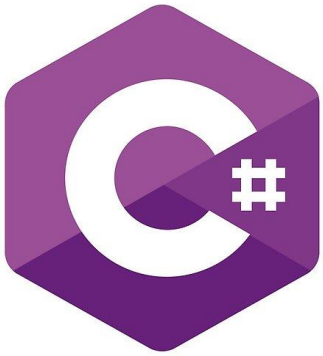
9. Tools Used

9.1 C++

C++ is a general-purpose programming language. It has imperative, object-oriented and generic programming features, while also providing facilities for low-level memory manipulation.

9.2 C#

C# is a multi-paradigm programming language encompassing strong typing, imperative, declarative, functional, generic, object-oriented, and component-oriented programming disciplines.



9.3 Unity

Unity is a cross-platform game engine developed by Unity Technologies, which is primarily used to develop both three-dimensional and two-dimensional video games and simulations for computers, consoles, and mobile devices.

Section 10: Conclusion and Future Scope

10. Conclusion and Future Scope

In a nutshell, we have presented a scrabble agent that provide the best alternative move to our user after the user player his turn in the teacher mode as well as playing against other agents/players in the AI mode. Additionally, our scrabble agent plays its moves in a good timely manner following its algorithm which was mentioned in the implementation part, since the main target is to do so in a small amount of time, since our scrabble agent shall be used in competitions. This has all been presented in a user-friendly GUI to make the game more appealing for our users.

Our future scope, in terms of design and implementation, includes going into a deeper depth to guarantee that this is a move towards winning. On the other hand, our future scope, in terms of GUI design, includes animating movement, reducing the rendering time for the GUI, adding music to the game, adding an option tab in the main menu to enable and disable music and adding effects when eaten or a special move is in place.

Section 11: References

11. References

11.1 Introduction References

- https://www.scotthyoung.com/blog/wordsmith-download/
- https://scrabble.hasbro.com/en-us/history
- https://en.wikipedia.org/wiki/Scrabble

11.2 Market Survey References

- http://people.csail.mit.edu/jasonkb/quackle/doc/how_quackle_plays_scrabble.html
- https://www.researchgate.net/figure/Video-game-consumer-market-value-worldwide-from-2011-to-2019-by-distribution-type-in_fig1_328902434
- https://www.limelight.com/resources/white-paper/state-of-online-gaming-2018/

11.3 Research References

- http://boardword.com/static/bitboards.html
- https://wikivisually.com/wiki/Bitboard
- https://www.youtube.com/watch?v=MzfQ8H16n0M&t=573s
- http://people.cs.ksu.edu/~rhowell/DataStructures/trees/tries/word-games.html
- https://discuss.codechef.com/questions/109504/how-trie-as-2d-array-work
- https://en.m.wikipedia.org/wiki/Search_data_structure
- https://en.wikipedia.org/wiki/Trie
- https://www.rabbitmq.com/
- http://zeromq.org/

11.4 Implementation References

- https://en.wikipedia.org/wiki/Monte_Carlo_method
- https://arxiv.org/pdf/1901.08728.pdf
- https://en.wikipedia.org/wiki/WebSocket
- https://www.rabbitmq.com/dotnet.html
- https://www.cs.cmu.edu/afs/cs/academic/class/15451-s06/www/lectures/scrabble.pdf
- http://boardword.com/static/bitboards.html