Universidad Carlos III de Madrid

Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

Analizador de posiciones del tablero del Go

Autor: Javier Huertas Tato

Tutor: Álvaro Torralba Arias de Reyna

Tutor adjunto: Daniel Borrajo Millán

Índice

CONTENIDO

ILUSTRACIONES

ECUACIONES

TABLAS

Abstract

Project description

Throughout humankind's history games have been a defining part of it. For humans, games are a unique phenomenon. They are a challenge established within a defined set of abstract rules, an example of what humans desire: overcoming obstacles and going forward. These challenges are not required for survival, solving and mastering them are its own rewards.

A great amount of games have been created during our history. Games are extremely varied, difficult, sophisticated, simple. The world nowadays has a lot of games to offer and one of its main categories is tabletop games. Again, most of them have been created and popularized throughout history. One of the better known games is Go.

Go's main feature is being fairly simple in terms of rules which can be explained within a short time. However, it may take a whole life for a normal person to master these rules to their maximum extent. This makes it very interesting and peculiar because it is one of the most difficult games ever created. It is a very complex game given its age, and after 2000 years it is still being studied.

In chess, another tabletop game, automatic players are able to play at a grandmaster level by using techniques based on the exhaustive exploration of possible moves, like min-max search with alpha-beta pruning, with the help of carefully designed evaluation functions. This approach is significantly less useful with Go. Go has too many possibilities in terms of movements so the approach taken in Chess gives far less advantage with Go.

Go is a complex game for humans to play, but it is even harder to play for computers. Despite its age there are ongoing investigations about Go's computer analysis. No optimal strategy has been found yet for Go. It is one of the games that has not been solved yet and it is considered one of the most difficult to handle.

Any effort made in this area is important because of its complexity. Any valuable addition will push further investigation forward. The motivation of this project is to improve our knowledge of the computational analysis of Go.

Following the reasoning made before, a Go tool will be developed. This tool will be able to analyze any given Go board using an algorithm known as Monte-Carlo Tree Search. Some playing agents used in Go competitions use this algorithm. However these agents use MCTS only to play, so this project will take a different take on this algorithm. MCTS will be used to retrieve information about influence and other different features of a Go board. Furthermore, the tool developed will be used in order to further analyze information retrieved with machine learning techniques.

Fuego

Fuego is a collection of C++ libraries for developing software for the game of Go. It includes a Go player that uses Monte-Carlo Tree Search.

All libraries and applications of *Fuego* are open-source.

The computer application that *Fuego* includes has competed against other Go programs with fairly good results, frequently ranking between first and third place. However, it has recently been demoted to lower ranks.

Additionally it includes comprehensive documentation and some extra manuals. This makes it more accessible to develop with. *Fuego*'s code has many classes and lines of code. This issue turns it into an intimidating application to program with, but its well-made documentation makes up for any inconvenience the clarity of the code may cause.

It requires some external libraries in order to be run, which makes it less portable.

Weka

Weka is a collection of machine learning algorithms for data mining tasks. The algorithms can be applied directly to a dataset or called from code. Using said algorithms it can fulfill tasks such as: pre-processing, classifying, regression, clustering, association rules and data visualization.

All algorithms and procedures of *Weka* can be freely used.

Weka is an approachable tool. For simple analysis, it is frequently the best option available. *Weka*'s main problem is that it is inefficient with big problems. Big sets of instances or attribute are highly time consuming for *Weka*.

Weka can be launched through commands in a *Linux* terminal using *Java*. Therefore there are ways to communicate between *Weka* and other programming languages (like C++) making use of the terminal and command lines.

Monte-Carlo Tree Search

Monte-Carlo Tree Search or MTCS is a heuristic search algorithm that executes a set of random plays and takes decisions making use of a great sample of randomly-generated games. It is greatly useful in the study of Go because the search space of this game is extremely extensive and it is difficult to be handled. In a 19x19 board (which is the standard size) there are approximately 3^{361} states in the search space.

For the default version of MCTS, each node contains two pieces of information: the game's state and the evaluation. The game's state is a snapshot in a given point in time of the game. The evaluation indicates who is the player with the most advantage over the game winning. Initially the evaluation is void and the state is the game's initial state.

The algorithm works as follows. First, an initial node has to be created with the previous characteristics. Once the first node is established, the tree can be built from that starting point. The first node is expanded making random moves.

If a terminal node is reached when expanding nodes, the terminal node's information is saved and propagated backwards in the tree. When all information is propagated the first node makes another random move and keeps playing.

Once a finite number of nodes has been explored the algorithm stops expanding nodes and starts the decision. Deciding which move to play depends on which game has a higher win/loss ration throughout all random games played. MCTS can be improved if the evaluation function for deciding which is the best child node is replaced.

Fuego uses the Upper Confidence Tree evaluation function in order to improve its performance.

Influence Map

In tabletop games, an influence map is a numerical representation of a board. Influence is an indicator on the control a player has on certain area of the board. This can be calculated through the pieces on the board and the influence function. Influence can be added by allied pieces or subtracted by enemy pieces. Influence can also be neutral if no player has control over a position.

Functionality

Fuego has a wide range of base functionality but there are 4 that take special interest in this project. These four functionalities are:

- Generate a movement (Search)
- Load a board
- Save a tree search
- Analyze a board

From this set of functionalities, two of them have been modified: generate a movement and save a tree search. Load a board has been excluded from modifications but it is used as part of the process of analyzing in order to load real board cases.

Generate a movement has been modified as follows. In order to generate a movement it is necessary to create a Monte-Carlo Tree Search. The tree search expands nodes with randomly generated movements until they find a terminal state. A terminal state is considered a situation when two passes have been made and therefore the game is over with a winner. All information of a terminal node is gathered in order to decide the best movement available from the set of movements made.

The tree that *Fuego* generates has been modified in order to introduce new statistical information about each game. In this project the most important information gathered is the influence map, with the influence information of every position in the board.

Calculating an influence map at a given point in the game is a difficult process. Go is a game with really complex strategies and tactics, and it is much easier for a human to evaluate compared to that of a computer. A Go board has a wide set of features and it is a complex task to identify all. In order to analyze the board another approach can be taken. Instead of calculating the influence in every point of the game, using a MCTS the terminal nodes can be analyzed and back propagated through the tree. Following this method the influence maps of the game are precise and easy to generate.

This way the tree has been modified in order to generate influence maps, back-propagate them and obtain an influence map of the root node. The initial node gets an influence map based on randomly generated games.

The other part of the tool is saving maps. Saving a map can be made in two different ways. The first one is saving the tree in a format recognizable by a graphical interface. This format is known as *sgf* and the base system can already save a tree as a *sgf* file. The function required to do this is unable to save influence maps or any other statistical information that is required. Because of this, it has to be modified in order to represent all new information desired.

Another required modification is creating a file recognizable by a data mining tool. Weka has been chosen to be used, so information must be represented as an *arff* file. Using existing functionality for *sgf* files it is possible to adapt it for an *arff* output.

Architecture

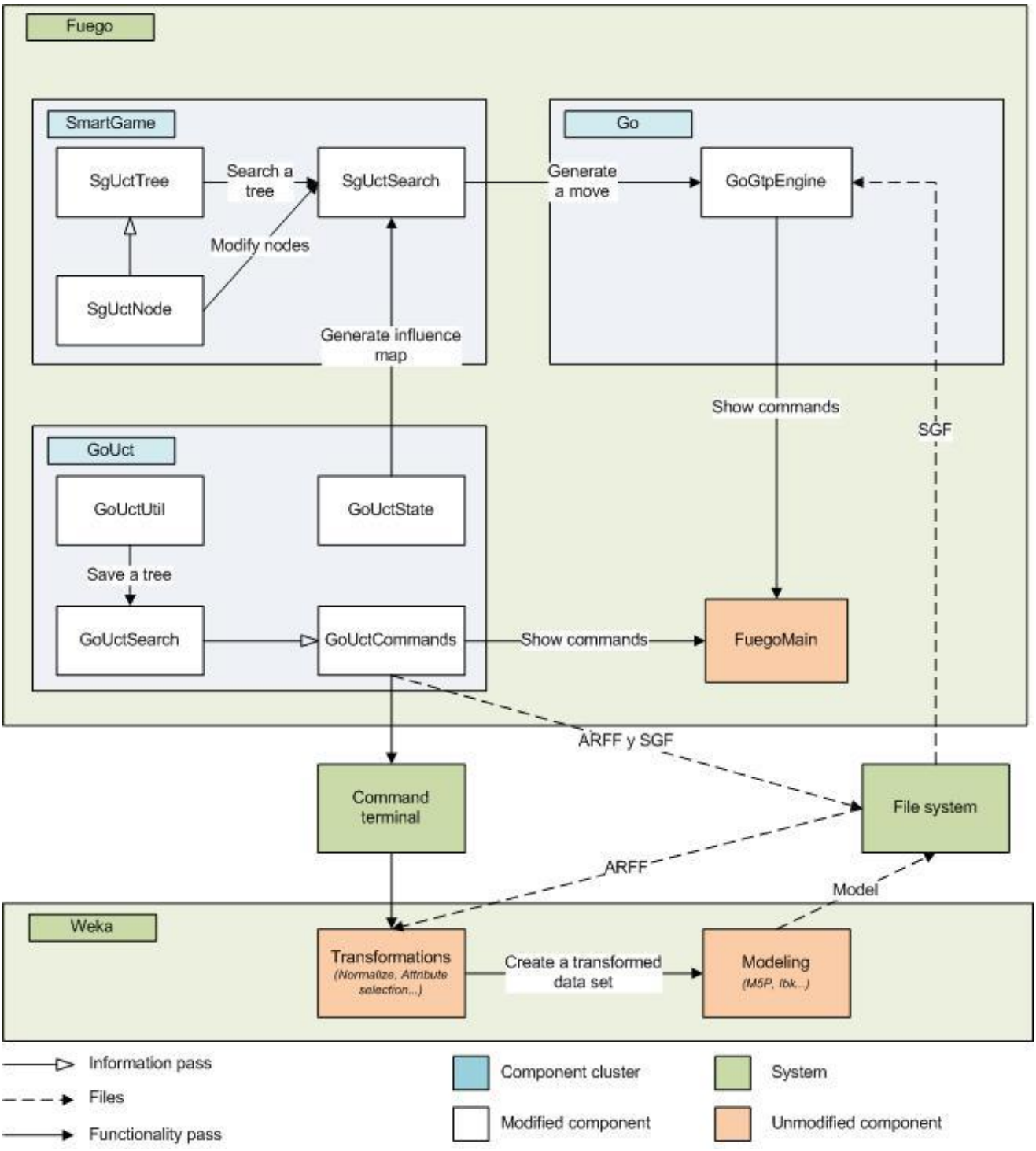


Figure 1: Systems architecture

The architecture depends on the interaction between four different basic systems:

- *Fuego*: The Go tool that has to be modified.
- *Weka*: Data analysis tool used for machine learning.
- File system: Place where all generated files are saved.
- Command terminal: Place where some instructions are executed for communication between systems.

These systems share data and, in some cases, they can interact. The most important part is the *Fuego* system. The *Weka* system is described in further detail in the experimentations, but both parts are independent for developing purposes.

Fuego contains several libraries: *SmartGame, Go, GoUct, SimplePlayers* y *GtpEngine.* From the set of libraries only three of them have to be modified. The libraries that have to be modified are: *SmartGame*, *Go, GoUct*. The first modification is changing the Monte-Carlo Tree Search.

- *SgUctTree*: Modify the tree update algorithm.
- *SgUctNode* : Modify the information each node can contain.
- *SgUctSearch*: Modify the way the search is made.
- *GoUctState*: Modify the information the search state can process.
- *GoGtpEngine*: Add the new options given to the user.

The search tree is compound by nodes so *SgUctTree*, *SgUctNode* and *SgUctSearch* are all related to each other. The search requires a search state so *SgUctSearch* and *GoUctState* are also linked. Finally, in order to have human-computer interaction *SgUctSearch* and *GoGtpEngine* are communicated.

After making the modifications on the search part it is required to save a search tree into the file system. For this purpose these classes must be modified:

- *GoUctUtil*: Modify the data printed and the printing format.
- *GoUctSearch*: Add the new file formats.
- *GoUctCommands*: Add the new options given to the user.

GoUctUtil communicates with *GoUctCommands* through *GoUctSearch* to offer its functionality. *GoUctCommands* needs a command to communicate with *Weka*. Communication between *GoUctCommands* and *Weka* has to be made in the command terminal.

Aside from all other relations, there are some special ones that show the communication through files. *GoGtpEngine* is able to load a game using a *sgf* file. *GoUctCommands* must be able to save *arff* and *sgf* files. *Weka* receives an *arff* file and it generates a model.

Experiments

After developing the analyzing tool, it will be used to make a set of experiments in order to test its effectiveness.

Creating the initial dataset involves using the modified *Fuego* tool. In the first place, the board must be loaded in *Fuego* with its *loadsgf* command. Then a movement is generated through the *genmove* command. The movement generation creates a new search, and once the search is finished the command *uct_savetree_arff* can be applied to save the generated tree.

When the initial dataset has been generated, some different transformations can be generated with the help of *Weka*. The original dataset can be transformed with 3 filters. These filters are applied to reduce the dimensionality of the dataset. The filters used are: Normalize, Remove useless and Remove "move". Normalize compresses the range of all attributes to 0 and 1. Remove useless removes irrelevant attributes to the class. Remove "move" discards the attribute from the dataset because it is incompatible with regression algorithms. "Move" is a String attribute so regression is unable to be applied.

After the dataset has been simplified another set of transformations can be applied in order to get different datasets. These filters will produce 3 datasets named after their last transformation and they are: PCA, Attribute selection and Random Projections.

Finally, after getting all datasets they can be introduced in the *Weka experimenter* and processed through machine learning algorithms. The algorithms used for the analysis are M5P, M5Rules, IBk and Linear Regression. From the experiments we get the correlation coefficient of each model.

In the first place and the most important part are the results obtained through attribute selection. From all the different transformations applied attribute selection offers a correlation coefficient consistently above 0.8. This is true for every algorithm used. In the dataset there are many unused attributes or that are unrelated to the class. However, all the data removed is different for each board used. The information removed with the "Remove useless" filter is completely circumstantial and specific to each board.

Then, in each experiment IBk obtains the worst results from the set of algorithms used. Even with attribute selection IBk is approximately 0.8 correlation while the rest of algorithms are around the 0.9 mark.

When PCA and *RandomProjections* are used, the results from these transformations never exceed the ones in attribute selection. PCA works better than Random projections. Both transformations offer results below 0.7 correlation coefficient so they are discarded in favor of attribute selection.

As observed, correlation coefficients for regression trees (M5P) have been above 0.9 they can be further analyzed. In the first place all trees generated show *raveValue* as the

root node. It always takes really important places in the tree layout. This implies that the RAVE value is very related to the class.

If the linear models are examined, attributes and coefficients can be seen together. In these models two values are always present: *moveCount* and *raveValue*. Both of them have always high coefficients in the linear models and they are important to the class. However the most interesting part to be analyzed are the board positions that appear in the models.

For example we have this linear model taken from one of the trees generated in the experimentation:

```
LM 7 => 0.0046 * inf7z7          - 0.0121 * infDiff11z14
        + 0.0924 * infDiff14z9   - 0.2243 * infDiff17z16
        - 0.0592 * board3z7      + 0.024 * board16z11
        + 0.0062 * lib9z6        + 0.0045 * lib18z15
        + 0.0178 * moveCount     - 0.2331 * raveValue
        + 0.539
```

With this information we can analyze each attribute and get some insight over the game's state. The following information must be taken into account: if the white player is winning, the class is close to 1 and the class approaches 0 when the black player is the one with the advantage. Some conclusions can be drawn from this node:

- Position (7,7) is strategically important for the white player.
- The black player is interested in controlling position (17,16)
- The white player is interested in controlling position (4,9)
- White player benefits from keeping free both (9,6) and (18,15) positions

This analysis can be made for each terminal node of the model tree.

Therefore, after the experimentation with the different data files:

- It is possible to predict who is winning the game through statistical information of the board. The correlation coefficient is approximately 90%.
- The algorithm that works better with this scenario is M5P for a dataset transformed through attribute selection.
- Attributes *raveValue* and *moveCount* have great impact on the decision tree.
- Using the linear regression models from the terminal nodes an analysis can be made. From this analysis important positions and tactical decisions are revealed.
- PCA and *RandomProjections* transformations perform far worse than attribute selection, and IBk is the worst performing of the regression algorithms used.

Conclusions

Go is an interesting game. It is simple and complex at the same time. A lot of hidden features and varied strategies turn an easy rule set into one of the less approachable problem in the world of computer science. Throughout the project a tool has been developed in order to study the Go problem in a different way. There was a high risk that it was impossible to apply the Monte-Carlo Tree Search effectively for analysis purposes.

In the end some positive results have been achieved. First, the tool described in this project shows a very useful vision of a board's influence, which helps evaluating the board as a human. It offers a great way to know how the game has developed so far and how a territory can be exploited to maximize a player's winning chance. When analyzing boards some good results have been achieved. It is possible to analyze a board with acceptable precision. There are some random elements but the influence maps are consistent. Through this analysis it is possible to visually distinguish alive groups of pieces, dead groups, conflict zones, mixed territory...

Results from experimentation also have been satisfactory. Experiments present high correlation coefficients, which implies that the win/loss ratio of a player at a given point is predictable and influence is related to it. Furthermore, thanks to attribute selection it is possible to see exactly which points of the map carry vital information about the game's progress.

1. Introducción

1.1. Descripción del proyecto

El mundo de los juegos ha sido uno de los aspectos que ha definido a la humanidad tal y como es. Un juego es un fenómeno único en el ser humano, donde se establece un desafío a partir de unas reglas abstractas. En definitiva los juegos no son más que un ejemplo del afán de avance del ser humano, pues son desafíos que no son necesarios para la supervivencia, sino autoimpuestos por el mero hecho de ser un obstáculo que superar.

A lo largo de la historia han surgido gran cantidad de juegos. Existen de todo tipo y a día de hoy existen gran variedad, juegos difíciles, sofisticados, sencillos, duraderos. Una de las ramas de juegos más antiguos que existen son los juegos de tablero. Se han creado muchos juegos de tablero, entre los más conocidos están el Ajedrez, las Damas o el Go.

El objeto de estudio del proyecto es el Go. El Go se caracteriza por ser un juego con unas reglas muy básicas, que pueden ser explicadas en cuestión de minutos, pero que puede tomar toda una vida aprender a un alto nivel. Es un caso muy peculiar de juego de mesa pues es uno de los juegos más complicados de dominar que existen. Incluso con su antigüedad sigue siendo un juego de tablero complejo. Con el ajedrez existen muchas estrategias distintas de apertura que son deterministas y, aunque dicha aproximación también se aplica al Go, la variabilidad de movimientos no proporciona tanta ventaja como en el ajedrez.

El juego es difícil de jugar para un humano, pero lo es todavía más difícil para un ordenador. Sigue siendo estudiado mediante técnicas informáticas, pero todavía no ha sido posible determinar la estrategia óptima. El Go es uno de los juegos de tablero que no se ha logrado resolver y se considera como uno de los más difíciles de tratar.

Su dificultad hace que cualquier esfuerzo en este área sea importante y cualquier contribución que añada algo de valor a este campo es crucial. Por ello existe este proyecto, para aportar algo nuevo a este campo en desarrollo.

Para expandir el trabajo sobre el Go se elaborará una herramienta capaz de analizar un tablero de Go mediante una técnica de búsqueda conocida como Árbol de Monte-Carlo[8]. Existen programas jugadores que utilizan esta técnica que participan en competiciones de Go, pero el algoritmo de búsqueda planteado puede tener muchas más aplicaciones. Se quiere utilizar esta técnica para analizar (no jugar) tableros de Go y entender el juego de una manera más analítica. Además, esta herramienta que se desarrollará se utilizará para ser examinada con técnicas de aprendizaje automático.

1.2.Estructura del documento

En el presente documento se detalla el trabajo realizado en el trabajo de fin de grado. Este capítulo 1 se dedica a introducir el documento y clarificar todo lo necesario antes de comenzar con el cuerpo del documento, aquí se explica cómo se va a detallar todo y en qué lugar se emplaza cada elemento del trabajo.

En el capítulo 2 se detalla el estudio realizado para entender la situación que rodea el trabajo. Se definen los factores más importantes del proyecto, como son: El juego del Go, el árbol de búsqueda de Monte-Carlo; así como se estudian trabajos y aplicaciones similares como por ejemplo *Fuego*. Con esta parte queda establecido el contexto que rodea al proyecto.

En el capítulo 3 se establecen los objetivos del proyecto, tanto generales como específicos. Con esta parte se fija la dirección del proyecto y qué se quiere conseguir de él.

En el capítulo 4 se analiza el problema que plantea el proyecto. Para dicho análisis se estudiará el juego del Go, su relación con computación y se creará un modelo abstracto a partir del cual posteriormente se construirá la aplicación. Dicha aplicación se basará en la herramienta elegida para el desarrollo.

En el capítulo 5 se describe el analizador de posiciones desarrollado a alto nivel. Se explicará con detalle cada método modificado y creado para el correcto funcionamiento de la aplicación. Es de tener en cuenta que todo lo expuesto en esta sección es una descripción general. Existirán métodos intermedios para el paso de datos entre clases que pueden ser omitidos por simplicidad del documento.

En el capítulo 6 se describe la experimentación realizada, tanto resultados como metodología. Aquí se exponen los resultados del proyecto y un análisis.

En el capítulo 7 se describe la organización del proyecto. Detalles sobre la organización, presupuesto, horarios e hitos se encuentran en esta sección.

En el capítulo 8 se habla de las conclusiones del proyecto, qué se ha obtenido de él, si los resultados han sido los esperados.

En el capítulo 9 se plantean futuros trabajos y mejoras a partir de la investigación realizada. Dada la naturaleza del proyecto es altamente probable que existan varias líneas por donde proseguir la investigación.

2. Estado del arte

El proyecto se centra en torno al juego del Go y su análisis mediante un árbol de búsqueda de Monte Carlo. Primero, es necesario entender qué es el Go y qué aporta respecto al mundo de la computación.

2.1. El juego del Go

El Go es un juego de mesa originario de China. Es un juego en el que dos jugadores tratan de dominar territorio y capturar piezas enemigas. Se basa por completo en la estrategia de manera similar a otros juegos de mesa clásicos como las damas o el ajedrez.

Sus bases son muy sencillas pero es muy difícil de dominar. En el juego hay un tablero con una cuadrícula de número de intersecciones variable (9x9, 13x13, 19x19 son tamaños comunes) en el que cada jugador puede colocar una pieza de su color (Negro o Blanco) para avanzar en la partida.

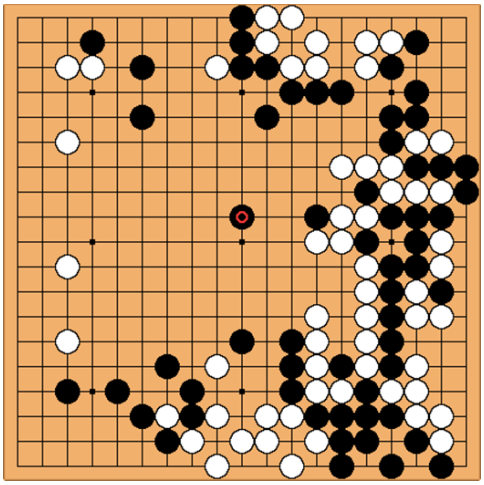


Ilustración 2: Tablero de Go 19x19. Punto intermedio en una partida

2.2. Reglas básicas

- Debe haber 2 jugadores, uno blanco y otro negro. Estos jugadores van alternándose el turno, empezando desde el negro y colocan una ficha en una posición libre o pasan su turno al siguiente.
- Un jugador puede capturar una o varias fichas del jugador enemigo privando a una ficha de movimiento.

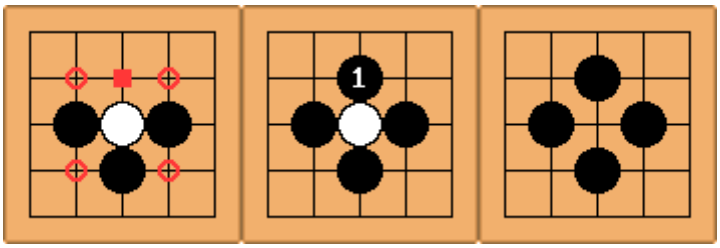


Ilustración 3: Ejemplo 1 de captura

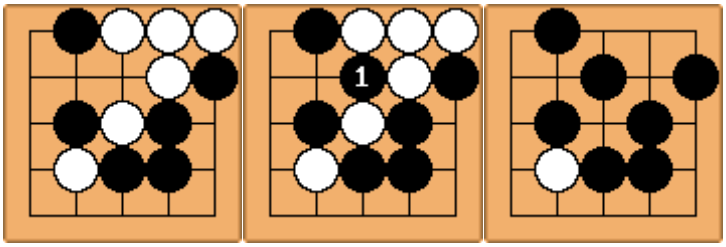


Ilustración 4: Ejemplo 2 de captura

- Un jugador no puede, mediante un movimiento, repetir una posición directamente anterior del tablero. Esta regla se llama regla del Ko.

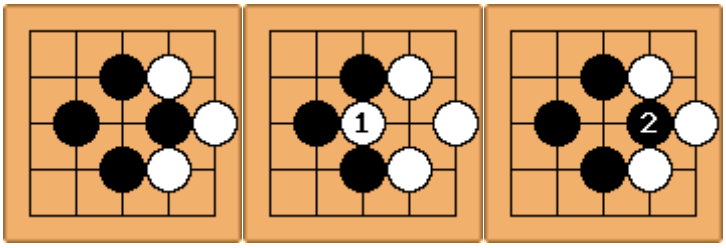


Ilustración 5: Ejemplo de movimiento ilegal. El movimiento 2 rompe la regla de no repetición volviendo a la posición directamente anterior.

- Es un movimiento ilegal colocar una ficha en una posición en la que sea capturada al entrar al juego salvo si contribuye a una captura.
- El juego acaba cuando ambos jugadores pasan o es imposible colocar más fichas. El criterio de victoria depende de qué reglas se usen. Un criterio muy popular es contar las posiciones dominadas por cada jugador.

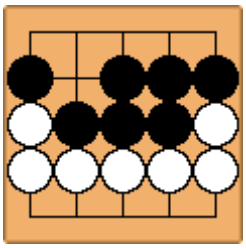


Ilustración 6: Ejemplo de victoria del jugador negro

- Existen variaciones para el juego pero estas reglas básicas se mantienen constantes.

2.3. Herramientas del Go

Existen muchas herramientas relacionadas con el juego del Go: programas jugadores de Go, editores e interfaces gráficas entre otros. Para el proyecto se examinarán programas jugadores. Este tipo de programas hacen uso de algún algoritmo de búsqueda para ser capaz de jugar de manera autónoma entre un humano y dicho programa, o entre dos programas similares.

Para definir qué programa es conveniente utilizar, primero hay que tener en cuenta lo que queremos lograr en el proyecto. Entre los criterios para evaluar como de útil nos será una herramienta podemos definir:

- Adaptabilidad/Flexibilidad: Capaz de incorporar nuevas funcionalidades sin hacer cambios radicales en la herramienta. Esto es para no dificultar la tarea de introducción de funcionalidades.
- Código abierto (*Open source*): Que el código del programa sea libre. Gracias a esta característica será posible acceder a la implementación del programa y, muy posiblemente, acceder a una documentación del programa.
- Documentación: La herramienta debe de tener un manual o comentarios para comprender el código. Esto va ligado a la adaptabilidad ya que mejorando la comprensión del código es mucho más sencillo realizar cambios sobre él.
- Búsqueda Monte-Carlo: Es imprescindible que la herramienta tenga la funcionalidad de hacer búsquedas Monte-Carlo.

Otros factores de interés pero secundarios:

- Eficacia: Capacidad de juego de la aplicación, si juega bien o mal.
- Eficiencia: Capacidad para jugar partidas en el menor tiempo posible.

2.3.1. Fuego

Fuego[9] es un conjunto de librerías C++ para el desarrollo de aplicaciones de Go. Incluye entre sus prestaciones aplicaciones para jugar y tiene incorporado un jugador usando el árbol de búsqueda Monte-Carlo.

Todas las librerías y aplicaciones de *Fuego* son de código abierto.

La aplicación adherida a las librerías ha sido utilizada en competiciones contra otros programas de Go con buenos resultados, acabando frecuentemente entre el tercer y el primer puesto. Sin embargo, recientemente ha quedado relegado a puestos más bajos.

Dispone de una amplia documentación y algunos manuales, en este aspecto es fácilmente accesible. Por otra parte, el código es extenso y la cantidad de clases que contiene es poco manejable. A primera vista es poco aproximable para alguien ajeno al código, pero la documentación y manuales atenúan este problema. Gracias a la extensión es fácilmente adaptable, hay más opciones a la hora de trabajar con estas librerías.

Requiere de algunas librerías externas para poder funcionar, lo que hace que su portabilidad sea más costosa.

2.3.2. Pachi

Pachi es un entorno modular para hacer programas que jueguen al juego Go. Incluye un motor de juego propio integrado en el entorno que, por defecto, juega con el árbol de búsqueda de Monte Carlo.

Todo el código de *Pachi* es de código abierto.

El jugador que utiliza *Pachi* es capaz de jugar en tableros de 19x19 con 2 Dan. En las competiciones contra otros computadores queda consistentemente entre el segundo y cuarto puesto. Es capaz de ganar a jugadores profesionales, ganando en una ocasión a un jugador de 7 Dan.

El código no dispone de documentación, sin embargo está comentado y es legible. No requiere una amplia búsqueda dentro de sus archivos para encontrar los métodos requeridos.

Aparte del código que es proporcionado no hace falta nada más para ejecutarlo. Es fácilmente portable.

2.3.3. Otros programas

Existen otros programas de Go que merecen especial mención. En primer lugar la aplicación *Zen*, capaz de alcanzar 6 Dan en una de sus versiones y de vencer a jugadores profesionales con 9 Dan. La aplicación se distribuye de manera comercial por eso no queda entre las opciones disponibles.

MoGo e *Indigo* son dos aplicaciones de Go que introdujeron conceptos importantes dentro del mundo del Go y la computación. *Indigo* comenzó la experimentación con árboles de búsqueda Monte-Carlo, popularizándolo como método principal de búsqueda en los jugadores actuales de Go. *MoGo* introdujo los conceptos de UCT y RAVE, dos refinamientos sobre la técnica de Monte-Carlo que mejoran la búsqueda. No disponen de documentación y están tecnológicamente obsoletos, por ello no se consideran como opciones viables.

El resto de las aplicaciones más populares, son comerciales o no superan lo que Pachi o Fuego tienen que ofrecer.

2.3.4. Herramienta elegida

En definitiva, a lo largo del proyecto se utilizará *Fuego* a pesar de ser mucho más extenso, menos eficaz y eficiente que *Pachi*, viene con una documentación mucho más completa y es mucho más flexible a la hora de realizar cambios debido a su consola de comandos.

El resto de programas no cumplían todo lo requerido en la primera descripción, por tanto no podrían ser utilizados para el proyecto.

2.4. Herramientas de aprendizaje automático

Existen muchas herramientas de aprendizaje automático. En el proyecto se utilizarán como parte del analizador de posiciones. Dado este caso es necesario encontrar una herramienta que se adapte a nuestras necesidades.

Dado que no se requiere alterar el código interno de la herramienta que se utilice, los requisitos que deba cumplir serán diferentes a los detallados anteriormente. Para el proyecto se examinarán herramientas con opción de regresión.

Como esta herramienta no tiene tanto peso sobre el proyecto como la herramienta de Go, se busca algo más sencillo de incluir y utilizar en el trabajo. Como tal, los requisitos reflejarán la relajación en las restricciones de uso.

Los requisitos que se buscan de la herramienta son:

- Integración: Capacidad para comunicarse con el lenguaje de elección, en este caso C++.
- Simplicidad de uso: Cómo de sencilla y rápida de utilizar es la herramienta.

- Gratuito: Para reducir los costes del proyecto. Existen herramientas perfectamente eficaces gratuitas.

2.4.1. Weka

Weka es una colección de algoritmos de aprendizaje automático para hacer minería de datos. Dichos algoritmos se pueden ejecutar desde una aplicación o mediante llamadas desde Java a través de líneas de comandos. Mediante estos algoritmos se pueden realizar tareas de pre-procesamiento, clasificación, regresión, agrupación, reglas de asociación y visualización de datos, aparte de ser también una posible plataforma para otros algoritmos de aprendizaje nuevos.

Todos los algoritmos y procedimientos de *Weka* pueden ser utilizados libremente.

En términos generales, resulta una herramienta muy accesible y para análisis pequeños resulta con frecuencia la mejor opción posible. El principal problema de *Weka* radica en su uso de Java. Para procedimientos pequeños es muy útil y rápido, pero es ineficiente con problemas de muchos atributos e instancias.

Como se ha mencionado antes existe integración con Java a través de la línea de comandos. Por tanto hay maneras de comunicarse con *Weka* a través de otros lenguajes de comandos. Se puede lograr la comunicación entre el programa llamando a un archivo ".sh".

2.4.2. R

R es un entorno software para computación y estadística. Se acerca mucho más a un lenguaje de programación que a una herramienta de aprendizaje en el sentido más tradicional. Cuenta con multitud de librerías y métodos para realizar análisis sobre conjuntos de datos.

Este entorno es utilizable de manera gratuita.

R es una herramienta compleja. Dado su parecido a un lenguaje de programación, es necesario tomar muchos más pasos para realizar un proceso sencillo y no guarda grandes parecidos con otros lenguajes populares como C o Java, por tanto requiere un proceso inicial de aprendizaje para ser usado. A largo plazo, resulta una herramienta muy potente y flexible, aunque poco accesible.

La comunicación entre R y C++, se basa en librerías para comunicar a ambos, por tanto la comunicación es sencilla pero poco accesible.

2.4.3. Herramienta elegida

De entre las opciones exploradas, Weka es, sin duda alguna, la opción más eficiente para el proyecto. Su simplicidad con análisis pequeños supone una ventaja muy

importante a tener en cuenta a la hora de realizar el trabajo y realizar una experimentación final.

2.5. Árbol de búsqueda Monte-Carlo

El árbol de búsqueda de Monte-Carlo es un algoritmo de búsqueda heurística que realiza jugadas ficticias al azar y toma decisiones en función de los resultados obtenidos de un gran número de partidas. Es especialmente útil en Go debido a que el espacio de búsqueda del juego no es manejable. En un tablero de 19x19 existen aproximadamente 3^{361} estados en el espacio de búsqueda.

Cada nodo del árbol de búsqueda contendrá dos piezas de información: el estado de la partida y la evaluación. El estado de la partida es un punto en el tiempo del juego, la evaluación indica a favor de qué jugador está la partida. Inicialmente la evaluación estará vacía y el estado será el estado inicial.

Partimos de un nodo inicial con el estado inicial. Esta jugada servirá para expandir el árbol de búsqueda. En la representación siguiente se representa solo la evaluación de la jugada con la proporción de victoria/derrotas en una partida hipotética de un juego de 2 jugadores.

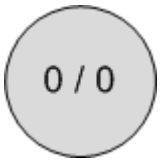


Ilustración 7: Nodo raíz del árbol

En el primer paso, se hacen X jugadas aleatorias, cada una sin información de evaluación y con una jugada diferente. A continuación repetimos esta acción recursivamente para cada nuevo nodo del árbol.

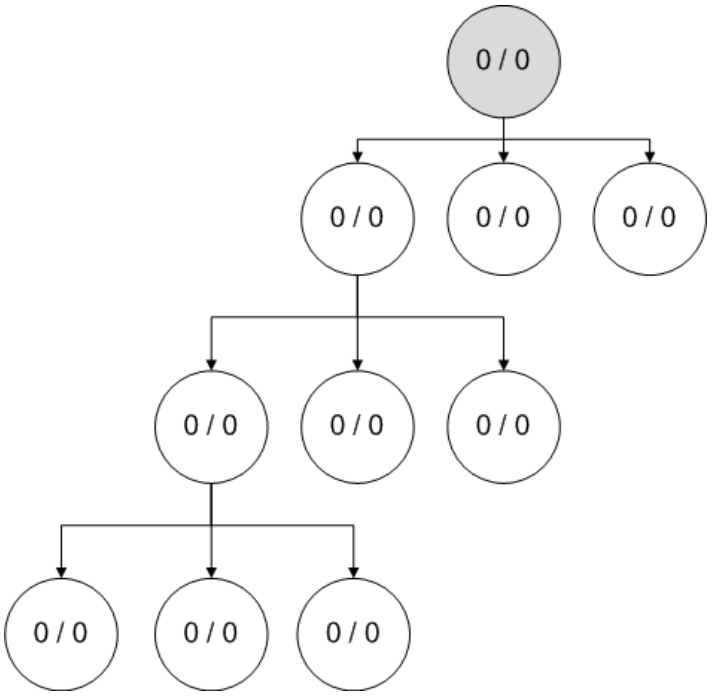


Ilustración 8: Ejemplo del árbol a profundidad 3

Si mientras estamos expandiendo el árbol alcanzamos un nodo hoja o estado final del juego, se recoge el resultado de la partida y se propaga hacia atrás en el árbol.

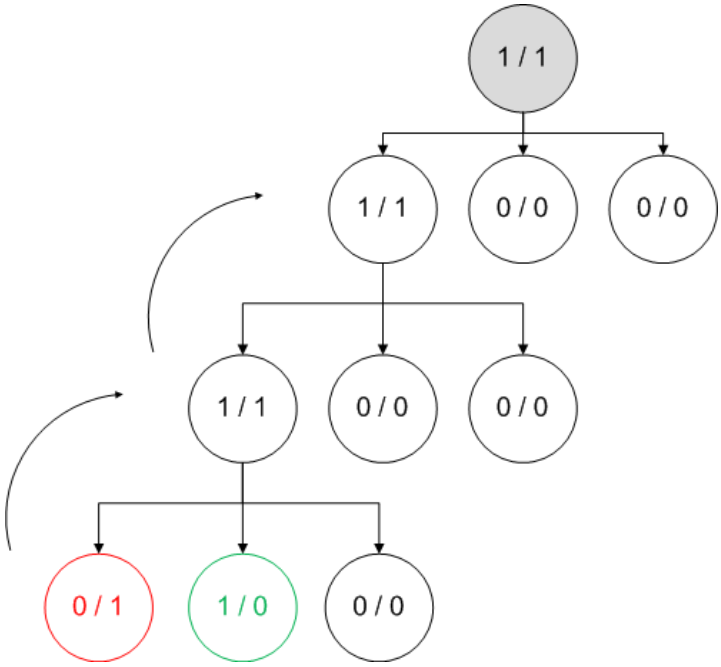


Ilustración 9: Ejemplo de la propagación del árbol a profundidad 3

Una vez se han expandido un número finito y manejable de veces los nodos del árbol, para el algoritmo y se comienza la decisión. Decidir el siguiente movimiento consiste en escoger el nodo con el valor de evaluación más favorable para el jugador.

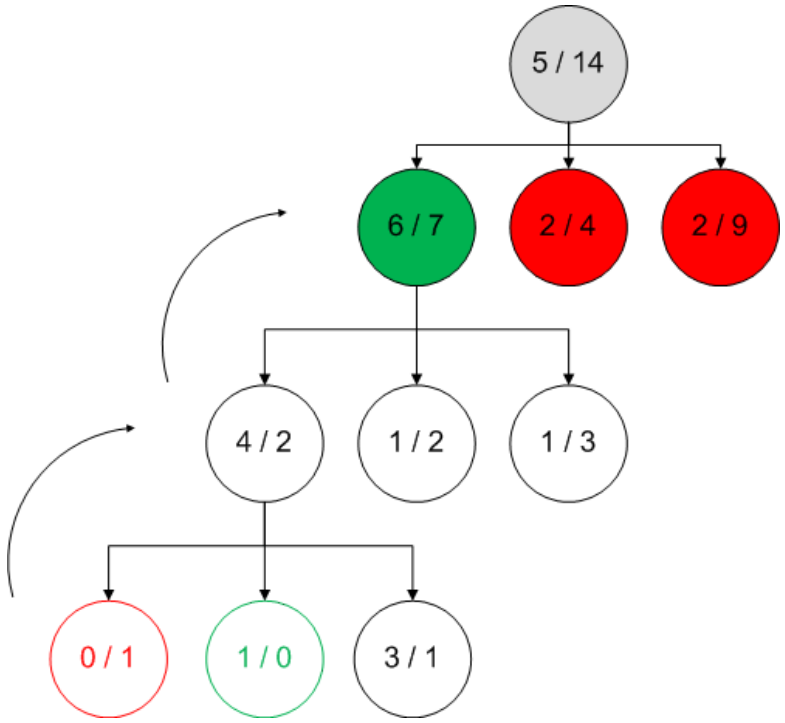


Ilustración 10: El nodo coloreado de verde ofrece la mejor relación Victoria/Derrota de todos, por tanto es el elegido.

Una de las posibles mejoras que se le puede aplicar a un árbol de Monte-Carlo es utilizar UCT[7]. A diferencia de la aproximación clásica de elegir la mayor proporción de victoria o derrota, es mucho más apropiado utilizar una función de evaluación más elaborada.

La función de UCT es mejorar el árbol de Monte-Carlo a partir de una función de evaluación que proporcione mayor información que la proporción entre victorias y derrotas.

Para ello se aplica la siguiente fórmula:

$$f_i = \frac{v_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}}$$

Ecuación 1: Función de evaluación UCT

- f_i: Evaluación para el nodo *i*.
- v_i: Número de victorias después del nodo *i*.
- n_i: Número de movimientos después del nodo *i*.
- c: Parámetro de exploración, en teoría $\sqrt{2}$, pero en la práctica se elige empíricamente.
- t: Simulaciones totales en un nodo, es la suma de todos los n_i.

2.6. Términos y expresiones frecuentes

En el mundo del Go y su tratamiento automático existen ciertas expresiones que se utilizan frecuentemente.

- Piedra:
  - Denominación que recibe una ficha cualquiera en el Go.
- Dan:
  - Denominación del nivel en el sistema de rangos *Dankyuisei*, empleado para catalogar la habilidad del jugador en el juego del Go. Dan implica un jugador avanzado y va de 1 a 9 en orden de habilidad, con 1 siendo el menor nivel de habilidad de rango Dan y 9, el de mayor nivel.
- Mapa de influencia:
  - En el ámbito de los juegos de mesa, un mapa de influencia es una representación de un tablero. La influencia es un indicador de cuál es el control de un jugador sobre una casilla. Esta se calcula a partir de las piedras que tenga un jugador sobre el tablero a partir de una función de influencia.

$$\boldsymbol{f_{i,j} = V_i * \frac{1}{(D(i,j)+1)}} \; ; \boldsymbol{D(i,j) \in \{0, N\}}$$

Ecuación 2: Influencia

- $\boldsymbol{f_{i,j}}$: Función de influencia para la ficha *i* en la posición *j*
- $\boldsymbol{V_i}$: Influencia en la posición dónde *i = j*.
- $\boldsymbol{D}$: Función de distancia de la casilla *j* que se quiere calcular a la ficha *i*
- $\boldsymbol{N}$: Distancia máxima a la que se calcula la influencia. si $D(i,j)>N$ entonces $f_i = 0$.

La influencia se puede superponer con otras influencias, si la influencia es enemiga se resta la influencia, si la influencia es aliada se suma.

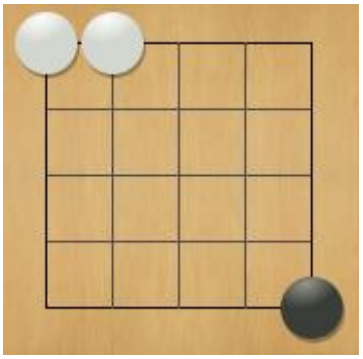
Un mapa de influencia es un conjunto de todas las influencias del mapa. En este caso solo se tiene en cuenta la versión para juegos de dos jugadores donde intervienen fichas blancas o negras (Ajedrez, damas, Go...). Por tanto:

$$\boldsymbol{I_j = \sum_{i=0}^{n} (f_{i,j} * C_i)}$$

Ecuación 3: Influencia total en un punto

- $\boldsymbol{I_j}$: Influencia total para la casilla j
- $\boldsymbol{f_{i,j}}$: Función de influencia para la ficha *i* en la posición *j*
- $\boldsymbol{C_i}$: Color de la ficha *i*. Si es blanco $C = -1$, si es negro $C = 1$
- $\boldsymbol{n}$: Número total de fichas sobre el tablero.

Por ejemplo, para el tablero propuesto, suponiendo $N = 2$ y para cada piedra del tablero $V_i=1$

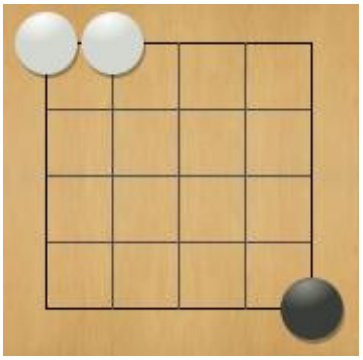


```
5| -1.5 -1.5 -0.8 -0.3   0
4|  -1   -1  -0.8 -0.3   0
3| -0.6 -0.6 -0.3   0   0.3
2|   0    0   0.3  0.5  0.5
1|   0    0   0.3  0.5   1
    [A]  [B]  [C]  [D]  [E]
```

Ilustración 11: Ejemplo de mapa de influencia

- Mapa de libertades
  - En el ámbito de los juegos de mesa, concretamente en el Go, un mapa de libertades es una representación de un tablero. Esta representación ofrece información del número de casillas libres alrededor de una piedra en todo el tablero. Cada piedra puede tener hasta 8 casillas libres alrededor, por tanto puede tener hasta 8 libertades. En los límites del tablero no se puede poner una piedra por lo que también limitan las libertades

    Un ejemplo de un mapa de libertades. En el caso de que las libertades no apliquen (es decir, haya una casilla vacía) se representa en el mapa como un "-"



```
5| 2  4  -  -  -
4| -  -  -  -  -
3| -  -  -  -  -
2| -  -  -  -  -
1| -  -  -  -  3
   [A][B][C][D][E]
```

Ilustración 12: Ejemplo de mapa de libertade

- Mapa de patrones:
  - En el ámbito de los juegos de mesa, concretamente en el Go, un mapa de patrones es una representación de un tablero. Esta representación ofrece información de un patrón concreto en cada posición del tablero. La representación tiene dos características particulares.

    La primera es que el tamaño del mapa es menor que del tablero. Se comprara un conjunto de casillas del tamaño del tamaño del patrón y se anota 0 si no existe y 1 si existe. Por ejemplo, si se quiere evaluar un patrón de 3x3 en un tablero de 5x5 el mapa de patrones sería de 3x3 porque son las combinaciones posibles con el tamaño del tablero indicado.

    En segundo lugar, para comprobar si existe un patrón es necesario, no sólo examinarlo según se representa, sino evaluar sus distintas rotaciones.

    Dado un tablero de 5x5 y un patrón de 2x2 se crea un mapa de 4x4 con las siguientes características:

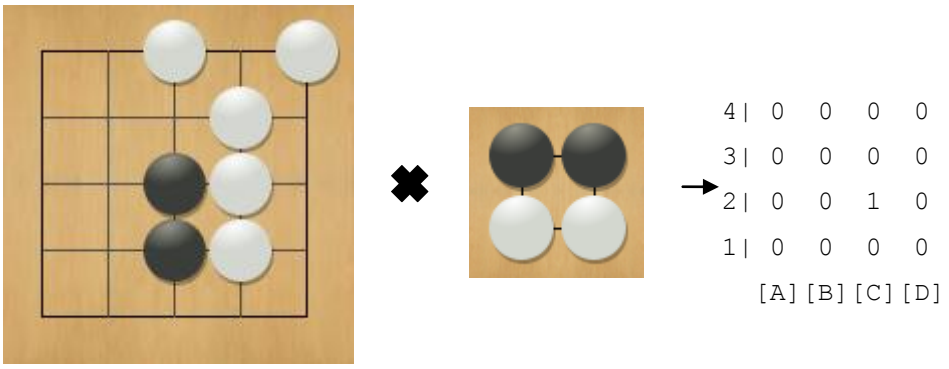


Ilustración 13: Ejemplo de mapa de patrones

- Formato sgf:
  - Formato de texto *Smart Game File*. Se utiliza para guardar partidas en programas de juegos de mesa.
- Formato arff:
  - Formato de texto *Attribute-Relation File Format*. Se utiliza para almacenar datos para aprendizaje automático.

3. Objetivos

En el siguiente proyecto se perseguirán objetivos relacionados al juego del Go y su análisis.

Desde un punto de vista general, se busca crear un analizador de posiciones para el juego del Go a partir de un árbol de búsqueda Monte-Carlo. A partir de crear un analizador, se podrán realizar experimentos utilizando técnicas de aprendizaje automático para extraer más información de cada posición.

Como objetivos generales se distinguen:

- Crear una herramienta capaz de extraer información a partir de un tablero de Go.
- Crear una herramienta capaz de representar información en un formato analizable.
- Experimentar con los datos obtenidos mediante aprendizaje automático.
- Conocer más profundamente el Go y su funcionamiento

En primer lugar se quiere crear una herramienta que, dado un tablero de Go cualquiera, recree un árbol de búsqueda de Monte-Carlo y, mediante los datos obtenidos del árbol de búsqueda, recoger un análisis exhaustivo de las posibles jugadas futuras y la situación del juego. Entre los datos que se pueden obtener están la relación entre victorias y derrotas, la influencia de un color en un punto específico del tablero, cuál es la profundidad de la rama en la que se está analizando o las diferencias entre dos puntos de tiempo en el tablero.

Seguidamente hay que crear otra herramienta para representar datos. Los datos que se han recogido requieren una representación concreta para ser analizados, por tanto es necesario adaptar los datos. Existen lenguajes para representar juegos de mesa como texto plano, que pueden ser muy útiles para una interpretación directa mediante una interfaz preparada para leerlos. Además es posible representar los datos como un conjunto de datos analizable por una herramienta de aprendizaje automático como *Weka* o R.

Con las herramientas requeridas para experimentar, se llevarán a cabo experimentos para analizar los datos obtenidos y comprobar si es posible extraer algún tipo de información útil de ellos. Para ello es necesario buscar y seleccionar una serie de atributos clave en el juego del Go.

Finalmente, se pretende intentar conocer más el juego del Go. Durante mucho tiempo ha sido objeto de estudio y todavía es un problema sin resolver. Con el proyecto se pretende aportar nueva información útil acerca de cómo funciona el Go.

Desglosando los objetivos generales en objetivos específicos:

- Crear una herramienta capaz de extraer información a partir de un tablero de Go.
  - Encontrar una herramienta que genere el árbol de Monte-Carlo.
  - Analizar la herramienta para su posterior modificación.
  - Adaptar la herramienta para que extraiga la información que se desee.
- Crear una herramienta capaz de representar información en un formato analizable.
  - Organizar los datos
  - Crear un formato de salida para la interpretación directa
  - Crear un formato de salida para el análisis mediante aprendizaje automático
- Experimentar con los datos obtenidos mediante aprendizaje automático.
  - Buscar atributos clave de un tablero de Go.
  - Encontrar información relevante de los experimentos realizados.
- Conocer más profundamente el Go y su funcionamiento.
  - Descubrir nuevas técnicas de análisis para el Go.

4. Análisis del problema

4.1. El Go y la computación

El Go es un juego sencillo a primera vista pero a la hora de ser tratado con un ordenador surgen varios problemas al utilizar aproximaciones clásicas. Comparándolo a otros juegos clásicos como por ejemplo el ajedrez es muy distinto y los algoritmos que funcionan en el ajedrez con gran precisión, demuestran un comportamiento mediocre en el juego del Go.

El primer desafío que supone el Go con técnicas clásicas es la función de evaluación. Una función de evaluación permite medir qué jugador tiene ventaja sobre la partida en términos numéricos. En el caso del ajedrez es posible desarrollar una función capaz de evaluar las condiciones de la partida para luego hacer un nuevo movimiento. Sin embargo, las posiciones en el Go dependen de un análisis complejo, incluyendo detalles como las fichas conectadas, si un grupo puede ser salvado o no, si tienen una posición cercana estratégicamente fuerte o si la jugada influirá en otro grupo de fichas. Para un ser humano estos últimos pueden juzgarse con relativa dificultad y se basan en movimientos estratégicos, no es fácil desarrollar una función que con precisión evalúe quien tiene ventaja sobre el otro.

El segundo problema es el tamaño del tablero de juego. Jugando en el tablero estándar (19x19) existen muchas posibilidades para hacer un movimiento. Esto se une a la posibilidad de colocar una ficha en cualquier lugar del tablero. En algoritmos de búsqueda que exploren todos los movimientos posibles (Mini-Max, Alfa-Beta) esto supone un factor de ramificación desproporcionando (factorial), ocupando un tamaño en memoria imposible de manejar.

La clase del problema del Go es *EXPTIME-Completo*[1] con las reglas japonesas del Ko (que son las que utilizaremos). Por tanto, su orden de complejidad será de $O(2^{p(n)})$. Como poco estaremos tratando con un problema que escala de manera exponencial conforme al tamaño de entrada. Esto, en términos de computación, lo convierte en un problema intratable. Sin embargo existen maneras de atajar este problema buscando aproximaciones.

Entre las soluciones existentes en el problema del Go en este caso usaremos la búsqueda en árbol de Monte-Carlo.

4.2. Funcionalidad

La herramienta *Fuego* cuenta con un amplio abanico de funcionalidades base, pero en nuestro interés entran las funcionalidades siguientes:

- Generar un movimiento (Una búsqueda)
- Cargar un tablero
- Guardar una búsqueda
- Analizar un tablero

De estas funcionalidades se han modificado generar movimiento y guardar una búsqueda. Cargar un tablero queda excluido de modificaciones, pero es utilizado como parte del proceso de análisis para poder cargar tableros de casos reales.

Generar un movimiento se ha modificado de la siguiente manera. Para generar un movimiento es necesario crear un árbol de búsqueda Monte-Carlo. Como se ha descrito anteriormente el árbol de búsqueda expande nodos con movimientos aleatorios hasta llegar a nodos hoja con la conclusión de una partida. Toda la información de las partidas aleatorias se recoge para estimar cuál es el mejor movimiento posible.

Este árbol se ha modificado para poder incluir nueva información estadística acerca de cada partida. En este caso la información estadística nueva es acerca de la influencia que afecta a cada posición del tablero. Influencia se describirá como qué jugador afecta a qué casilla en qué medida.

Calcular un mapa de influencia en un punto arbitrario de la partida es especialmente dificultoso debido a que el Go es un juego con muchos matices complejos. Para realizar un análisis estadístico del territorio y la influencia se puede adoptar otra solución, esto es, calcular el mapa de influencia solo en un estado final, que es un proceso trivial, y a través de un árbol de búsqueda propagarlo hacia atrás para obtener mapas de influencia de todos los estados anteriores.

De esta manera se ha modificado el árbol que genera la herramienta para generar un mapa de influencia y propagarlo hacia atrás en el árbol y obtener finalmente un mapa de influencia en el nodo raíz, que es el estado que se está analizando.

Por otra parte está el guardado de mapas, que a su vez se subdivide en dos partes. En primer lugar se encuentra guardar un árbol en un formato reconocible por una interfaz gráfica existente. Este formato se conoce como *sgf* y el sistema base ya es capaz de guardar un archivo de este tipo. Sin embargo no maneja el guardado de la influencia y de otra información estadística que se desea incluir. Por ello es necesario ejecutar cambios sobre esta funcionalidad base para representar toda la información nueva que se desea incluir.

La otra modificación necesaria es crear un archivo en un formato reconocible por una herramienta de minería de datos. En este caso se ha elegido *Weka* para desempeñar ese papel, por tanto se representa la información en formato *arff*. Haciendo uso de la funcionalidad existente para representar archivos en formato *sgf*, es posible crear una versión que pueda transformar el árbol en un fichero de tipo *arff*.

4.3. Diagrama de la arquitectura

A continuación se hace un diagrama del sistema diseñado, tomando en cuenta las conexiones con Weka y el sistema de archivos.

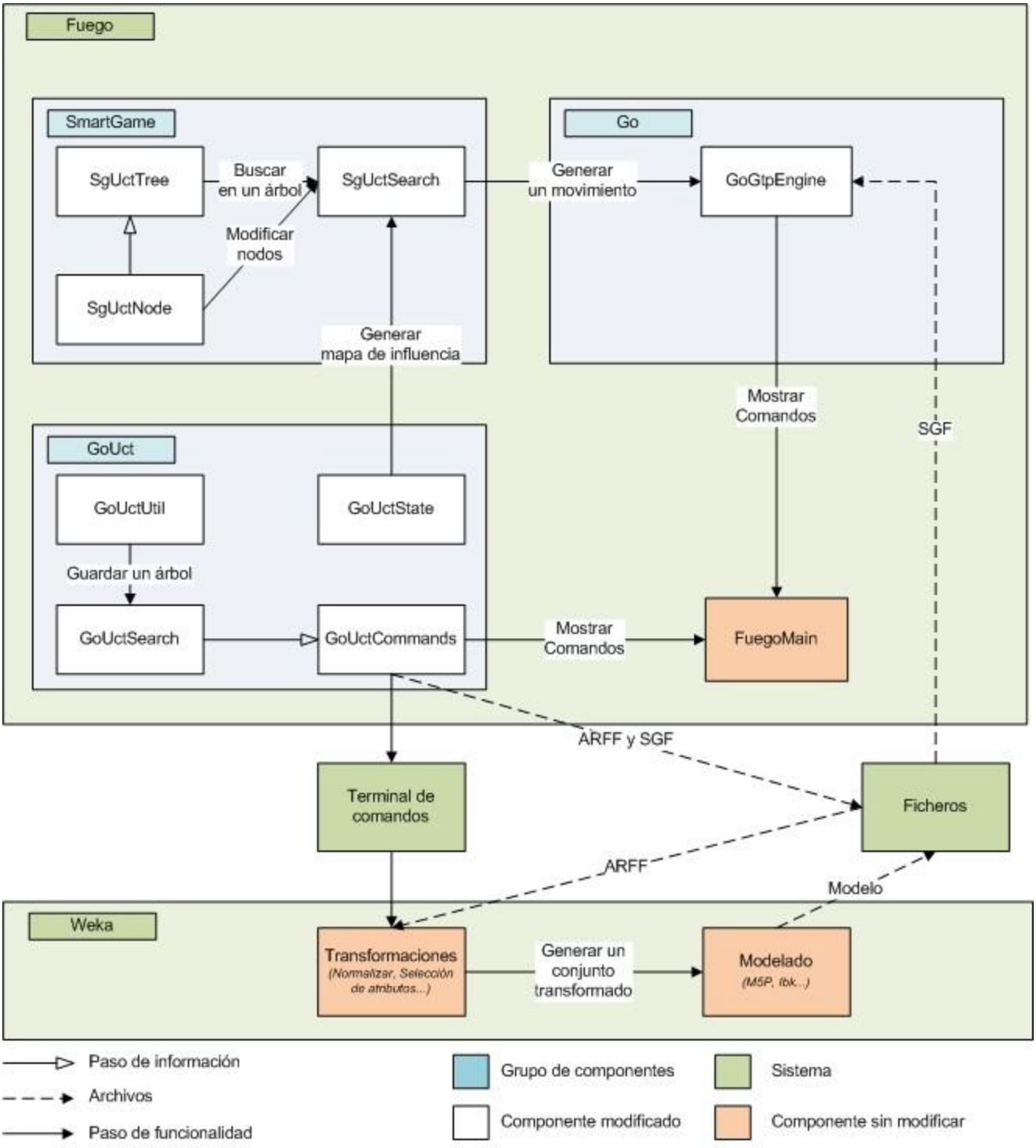


Ilustración 14:Diagrama de la arquitectura

En primer lugar se muestran 4 sistemas básicos:

- *Fuego*: La herramienta que se tiene de base para modificar.
- *Weka*: La herramienta de análisis de datos.
- Sistema de ficheros: Donde se almacenan todos los archivos que se generen.
- Terminal de comandos: Donde se ejecutan instrucciones para comunicarse entre herramientas.

Estos sistemas se comunican entre sí y pueden interactuar en algunos casos. La parte que más nos interesa es la de *Fuego*, pues la parte de *Weka* es la que se verá en la experimentación.

En *Fuego* existen varios módulos: *SmartGame*, *Go*, *GoUct*, *SimplePlayers* y *GtpEngine*. De estos módulos se requiere modificar únicamente los tres primeros. Lo primero que se quiere hacer es modificar el árbol Monte-Carlo por lo que es necesario modificar en las clases:

- *SgUctTree*: Para modificar la actualización del árbol.
- *SgUctNode* : Para modificar la información que almacena el árbol.
- *SgUctSearch*: Para modificar la búsqueda
- *GoUctState*: Para modificar el estado con la información nueva.
- *GoGtpEngine*: Para añadir las nuevas opciones que se ofrecen al usuario.

El árbol que utiliza la búsqueda se compone de nodos, haciendo que *SgUctTree*, *SgUctNode* y *SgUctSearch* estén relacionados entre sí. La búsqueda requiere de un estado por lo que *SgUctSearch* y *GoUctState* estén relacionados. Finalmente, para poder comunicarse con el usuario *SgUctSearch* y *GoGtpEngine* están relacionados.

En segundo lugar se quiere hacer que se guarden archivos, para ello es necesario modificar:

- *GoUctUtil*: Para añadir nueva información y formatos de impresión.
- *GoUctSearch*: Para añadir los nuevos formatos.
- *GoUctCommands*: Para añadir las nuevas opciones que se ofrecen al usuario.

GoUctUtil se comunica a través de *GoUctSearch* con *GoUctCommands* para ofrecer la funcionalidad deseada. Además *GoUctCommands* necesita de un comando mediante el que se comunique con *Weka* por lo que a través del terminal manda una instrucción.

También existen relaciones para guardar ficheros y leerlos. *GoGtpEngine* es capaz de leer un *sgf* para cargar una partida, mientras que *GoUctCommands* debe ser capaz de almacenar archivos *arff* y *sgf*. En *Weka* entra un archivo *arff* y se extrae un modelo.

4.4. Requisitos

A continuación se especifican los requisitos del programa final. Se contará especialmente con los requisitos funcionales ya que no se depende tanto de la experiencia a ojos del usuario como de lo que la aplicación sea capaz de ofrecer para la experimentación.

El modelo de representación de los requisitos será el siguiente:

Identificador	***Nombre-identificador***
Tipo	*Descripción*
Prioridad	

- Nombre-identificador: Número que identifica al requisito con el formato T-XX donde T es una letra representativa del tipo de requisito y XX es el número del requisito en la categoría del tipo T. Ej: F-01, NF-03…
- Tipo: Tipo del requisito. Ej. Funcional, Diseño, Usuario…
- Prioridad: Dentro del proyecto, cuando debe hacerse. Ej: Alta, Media, Baja, Muy Baja
- Descripción: La descripción del requisito.

A partir de la descripción del proyecto y los objetivos, se identifican los siguientes requisitos:

Identificador	***F-01***
Funcional	*Extraer mapas de influencia a partir de estados finales*
Prioridad muy alta	

Identificador	***F-02***
Funcional	*Propagar mapas de influencia a través de un árbol Monte-Carlo*
Prioridad muy alta	

Identificador	***F-03***
Funcional	*Representar tableros de Go*
Prioridad alta	

Identificador	***F-04***
Funcional	*Almacenar tableros de Go en un archivo*
Prioridad alta	

Identificador	***F-05***
Funcional	*Almacenar mapas de influencia de Go en un archivo*
Prioridad muy alta	

Identificador	***F-06***
Funcional	*Almacenar un árbol de búsqueda Monte-Carlo en un archivo*
Prioridad muy alta	

Identificador	***F-07***
Funcional	*Limitar la profundidad de un árbol de Monte-Carlo al ser almacenado.*
Prioridad media	

Identificador	***F-08***
Funcional	*Generar diferenciales entre mapas de influencia de un nodo padre y un nodo hijo en un archivo.*
Prioridad baja	

Identificador	***F-09***
Funcional	*Almacenar diferenciales entre mapas de influencia de un nodo padre y un nodo hijo en un archivo.*
Prioridad baja	

Identificador	***F-10***
Funcional	*Crear un mapa de libertades a partir de un tablero de Go.*
Prioridad baja	

Identificador	***F-11***
Funcional	*Almacenar un mapa de libertades de un tablero de Go en un archivo.*
Prioridad baja	

Identificador	***F-12***
Funcional	*Identificar patrones en un tablero de Go*
Prioridad baja	

Identificador	***F-13***
Funcional	*Crear un mapa de patrones a partir de un tablero de Go.*
Prioridad baja	

Identificador	***F-14***
Funcional	*Almacenar un mapa de patrones en un archivo*
Prioridad baja	

Identificador	***F-15***
Funcional	*Representar la información de un árbol en formato Weka*
Prioridad alta	

Identificador	***F-16***
Funcional	*Representar la información de un árbol en formato sgf*
Prioridad alta	

Identificador	***F-17***
Funcional	*Predecir el jugador con mayor probabilidad de victoria a partir de información de un árbol Monte-Carlo*
Prioridad alta	

Identificador	***F-18***
Funcional	*Mostrar comando para cargar un tablero de Go*
Prioridad media	

Identificador	***F-19***
Funcional	*Mostrar comando para generar un árbol de Monte-Carlo*
Prioridad media	

Identificador	***F-20***
Funcional	*Mostrar comando para guardar un árbol de Monte-Carlo con información en un fichero*
Prioridad media	

Identificador	***F-21***
Funcional	*Mostrar comando para analizar un fichero con un árbol de Monte-Carlo*
Prioridad media	

Identificador	***NF-01***
No Funcional	*Utilizar la herramienta Fuego*
Prioridad alta	

Identificador	***NF-02***
No Funcional	*El tiempo de análisis de un fichero debe ser inferior a diez minutos*
Prioridad alta	

Identificador	***D-01***
Diseño	*Hacer uso de comandos para mostrar la funcionalidad*
Prioridad alta	

4.5.Diseño de la aplicación

4.5.1. Análisis de la herramienta base

Para el desarrollo de la aplicación analizadora de tableros haremos uso de la herramienta Fuego como base para la construcción. La parte de la herramienta que es necesaria es la que se encarga de generar un movimiento.

La documentación de la herramienta no dispone de un diseño de clases, pero viene complementada con sus propios diagramas para comprender el funcionamiento interno de su principal funcionalidad.

En este caso la aplicación sigue un proceso complejo para generar un nuevo movimiento, por ello hay que entender el funcionamiento a un alto nivel.

Todo lo descrito a continuación representa la base existente en *Fuego*[2] para el desarrollo del proyecto. Por tanto, en este apartado se estudia la funcionalidad base de la que dispone *Fuego*.

Fuego cuenta con cinco librerías y dos aplicaciones:

- Librerías
    - GtpEngine: Implementación de Go Text Protocol (GTP); independiente al juego. Es un protocolo de texto para comandos relacionados al Go.
    - SmartGame: Clases de utilidad para juegos de dos jugadores. Es independiente del Go.
    - Go: Clases específicas del Go.
    - SimplePlayers: Jugadores con algoritmos de juego simples
    - GoUct: Jugador UCT (Upper Confidence bounds applied to Trees). Es una extensión de la búsqueda Monte Carlo.
- Aplicaciones
    - FuegoTest: Interfaz GTP con comandos de prueba.
    - FuegoMain: Interfaz GTP con GoUctPlayer (Jugador UCT).

De todos los comandos existentes el que merece ser analizado es el comando GenMove<Color>. Este comando es el que genera un nuevo movimiento a partir del algoritmo seleccionado. (En este caso UCT)

A continuación se muestran diagramas que representan el funcionamiento de un método. Se ven las llamadas y bucles entre ellos. Su interpretación consiste en:

- Los métodos se interpretan desde la parte superior del diagrama.
- Una flecha indica una llamada a un método. Una llamada puede ser condicional.
- Las llamadas se interpretan desde la parte izquierda del diagrama.

En la figura 4.1 se representa el proceso que sigue el comando para lanzar la búsqueda. *Fuego* tratará de buscar un movimiento en un libro de movimientos antes de generar un movimiento. Este libro contiene jugadas iniciales y son movimientos designados por un

jugador humano. Los jugadores de Go no hacen correctamente las primeras jugadas, por ello se utiliza este libro de movimientos. Si no existe ninguna coincidencia en dicho libro, no existe ningún movimiento que hacer de manera automática y por tanto se pasará a utilizar uno de los siguientes métodos:

- Utilizando una política
- Búsqueda UCT (La que utilizaremos)
- Búsqueda Monte-Carlo (Búsqueda Clásica sin mejoras)

En este caso seguiremos la búsqueda UCT.

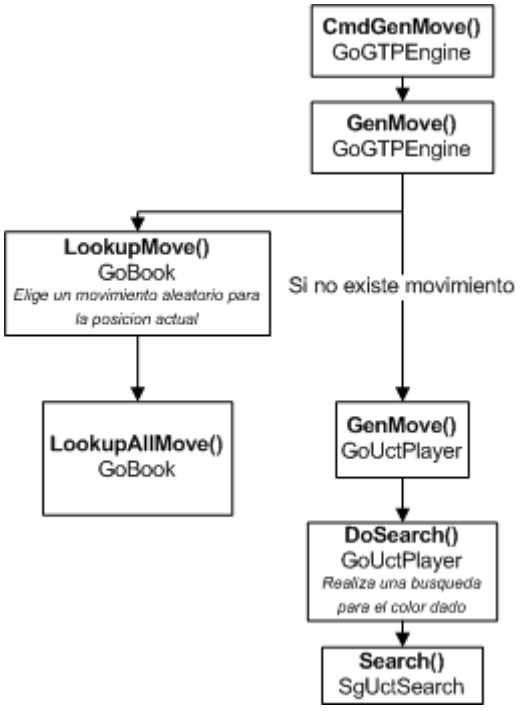


Ilustración 15: GenMove()

El método *Search()* empieza iniciando el juego y creando hilos para procesar las jugadas. Una vez un hilo ha comenzado, fuego entra en un bucle de búsqueda para construir un árbol de manera iterativa jugando partidas. El bucle se termina cuando es imposible expandir el árbol. Una vez se finaliza la búsqueda, se podan los nodos con cuenta baja y se procede a buscar la mejor secuencia del árbol.

La secuencia se halla encontrando el mejor nodo hijo del nodo actual del árbol. La selección tiene cuatro elecciones posibles:

- Elegir movimiento con el mayor promedio de victorias/derrotas
- Elegir movimiento con mayor cuenta
- Usar el límite UCT
- Usar suma ponderada de los valores UCT y RAVE (valor métrico para la elección del mejor nodo)

En la figura 4.2 se representa el proceso seguido por *Search()*.

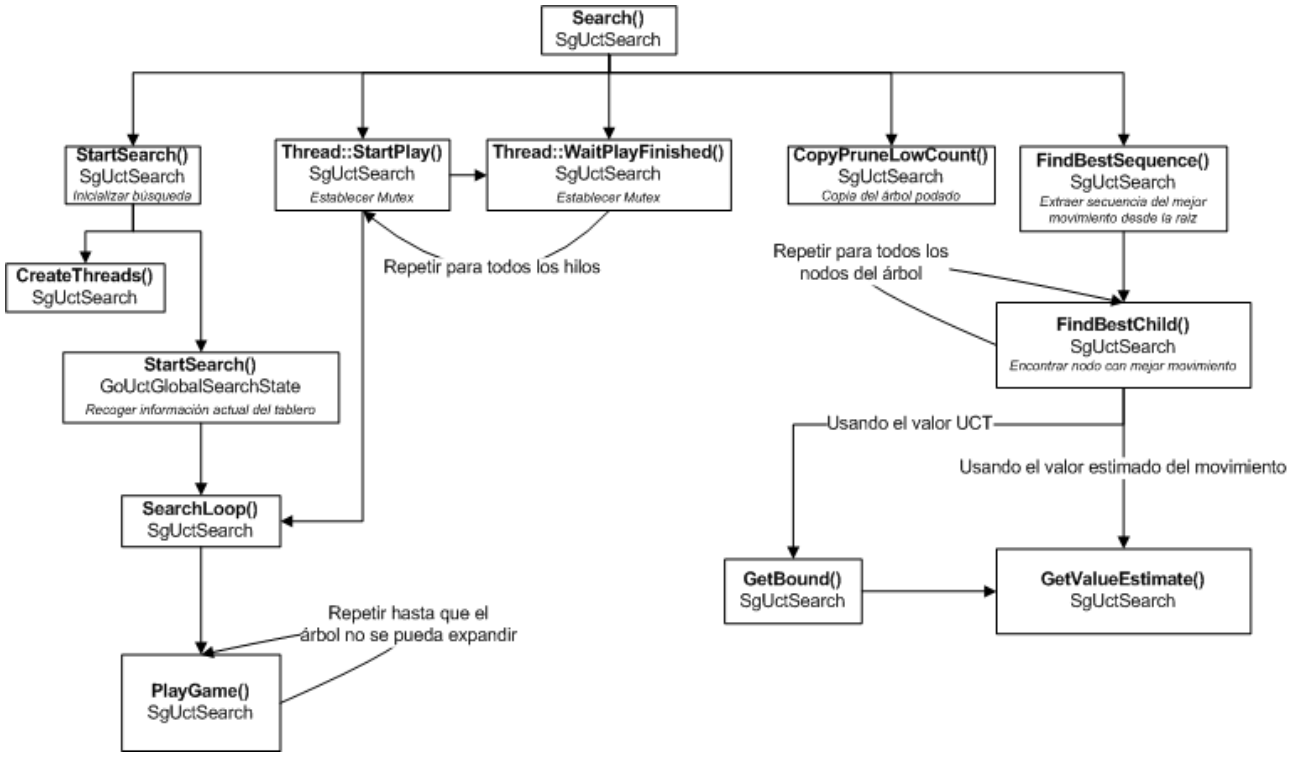


Ilustración 16: Search()

PlayGame() se divide en dos partes, una fase dentro del árbol (*In-treephase*) y una fase de jugadas (*Playoutphase*). Todos los movimientos realizados dentro de estas fases son simulaciones de partidas y serán deshechos al final, manteniendo toda la información aprendida. La condición de parada de un juego es que ambos jugadores pasen dos veces (dos movimientos "Pasar" seguidos).

Con toda la información recogida, cuando se acaba el juego, el árbol y los valores estadísticos son actualizados.

Lo que el programa hace en estas fases es, primero (*In-tree*) crea nodos en el árbol hasta llegar a nodos hoja (se prueba una victoria o derrota).Fuera del árbol (*Playout*) se hacen jugadas hasta llegar a estados finales de la partida.

En la figura 4.3 se representa el proceso que sigue *PlayGame().*

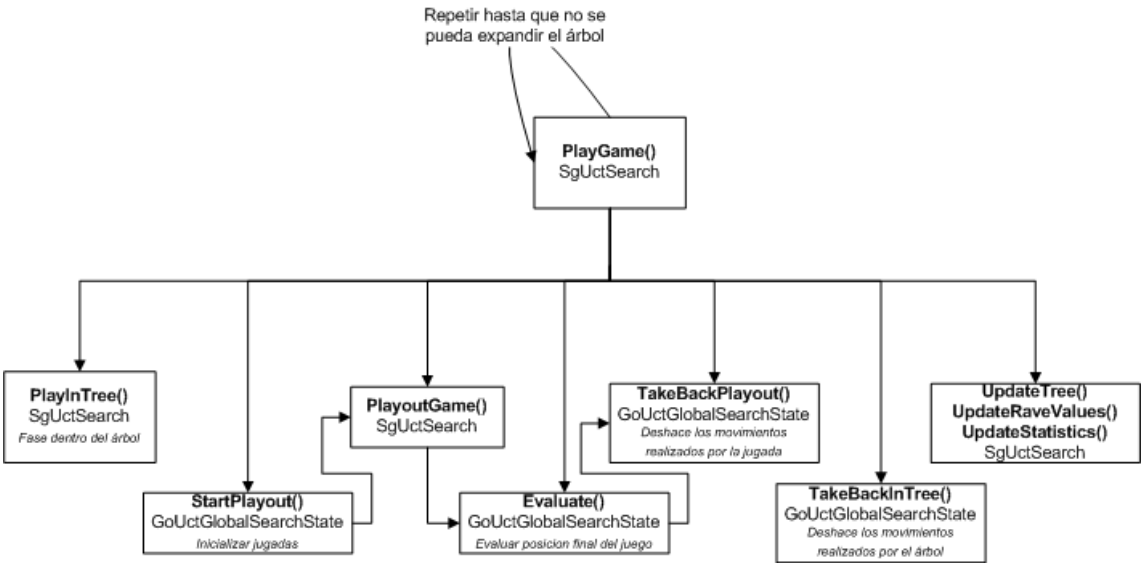


Ilustración 17: PlayGame()

La fase dentro del árbol expande nodos hasta que hay una victoria o derrota. En primer lugar genera movimientos legales sin explorar, luego genera los nodos hijos y elige el mejor hijo en función al límite UTC y, finalmente, ejecuta el movimiento. El bucle continúa hasta que se produce un estado de victoria o de derrota.

En la figura 4.4 se muestra el proceso que sigue *PlayInTree()*

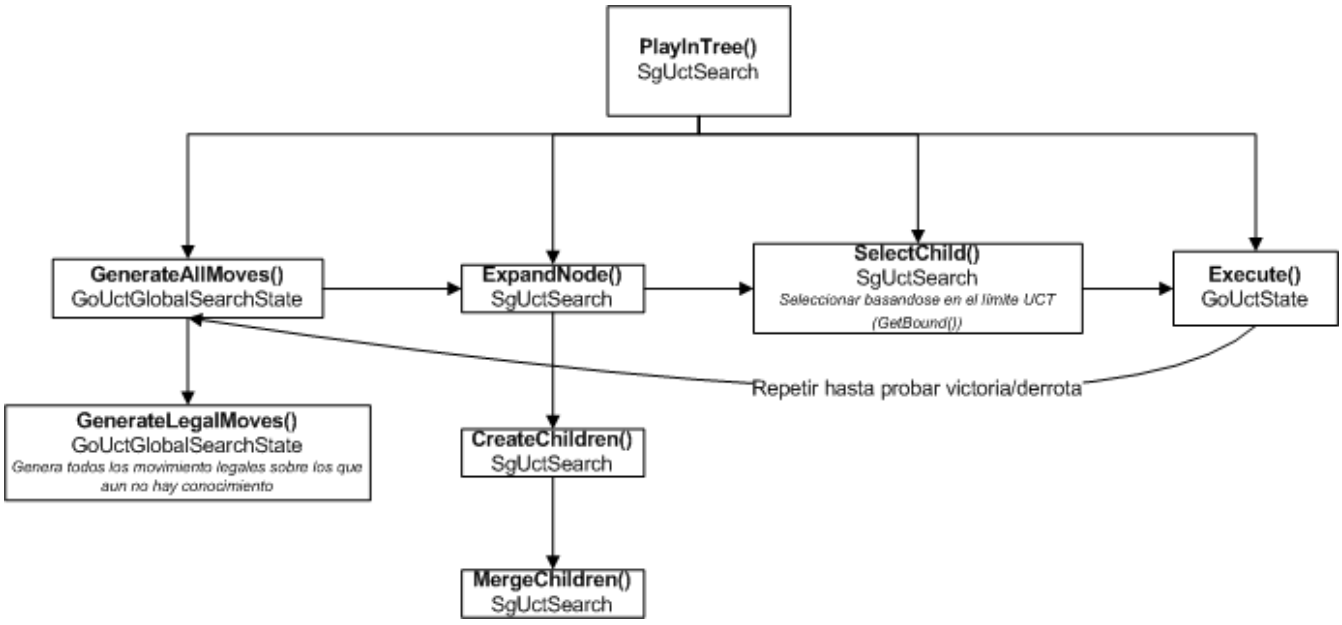


Ilustración 18: PlayInTree()

Fuera del árbol se produce la fase de jugadas. Durante esta fase se trata de generar movimientos en la jugada basándose en la política existente para ello, no es aleatorio como en el Monte-Carlo habitual. Los movimientos se generan hasta llegar a un movimiento "Pasar".

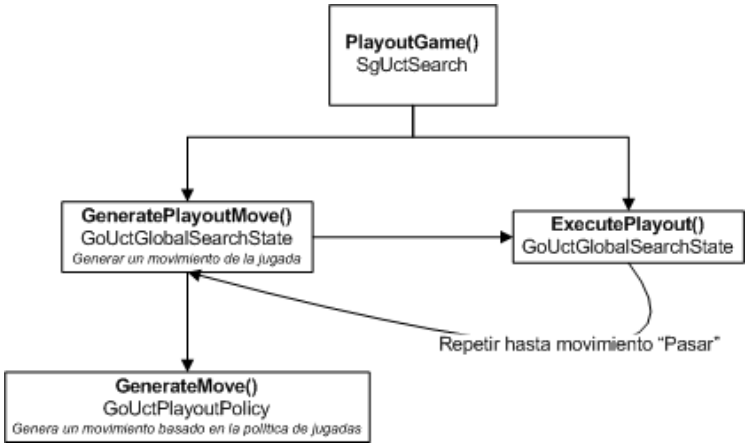


Ilustración 19: PlayoutGame()

Estos son los pasos que requiere seguir el comando para generar un movimiento. Esto cubre la primera parte de la funcionalidad, generar el árbol Monte-Carlo. Para cubrir la funcionalidad referente a guardar árboles en ficheros es necesario analizar otro comando distinto. Este es mucho más sencillo que el comando de generar un nuevo movimiento (no requiere hacer una búsqueda). El comando responsable de esta funcionalidad es *uct_savetree*.

Sencillamente llama a *SaveTree()* crea la cabecera del fichero de salida y, recursivamente, almacena todos los nodos del árbol generado por un movimiento

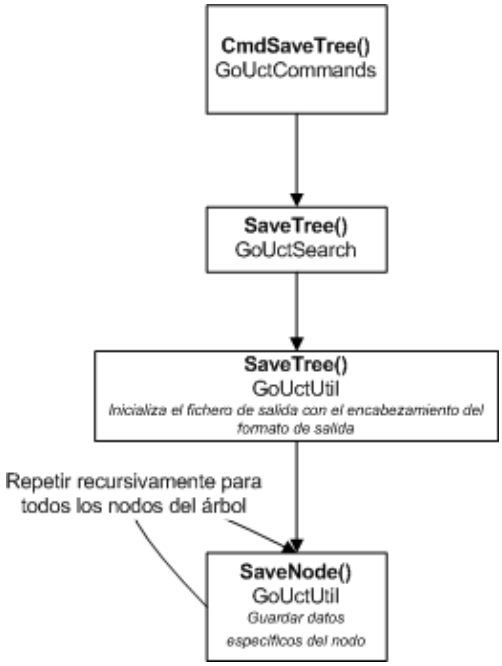


Ilustración 20: CmdSaveTree()

4.5.2. Diseño de alto nivel.

La aplicación final contará con 2 tipos de funcionalidades:

- Tipo 1: Funcionalidades internas. Estas deben ser tratadas en el interior del árbol y forman parte del árbol.
- Tipo 2: Funcionalidades externas. Estas pueden ser tratadas en el momento en el que se está generando la salida.

Por parte del Tipo 1 encontramos la funcionalidad principal de generar árboles de Monte-Carlo con mapas de influencia (Requisitos F-01 y F-02). Según la arquitectura de la aplicación es posible colocar todas las funcionalidades de análisis de datos (Requisitos F-08, F-10, F-12, F-13, F-17) en el momento de imprimir la salida junto al resto de funcionalidades de salida de datos (Requisitos F-04, F-05, F-06, F-09, F-11, F-14, F-15 y F-16)

Con todo esto en cuenta se parte del diseño del tipo 1 de funcionalidades. Estos son los componentes que será necesario modificar según han sido identificados con la información del punto 4.1.1.

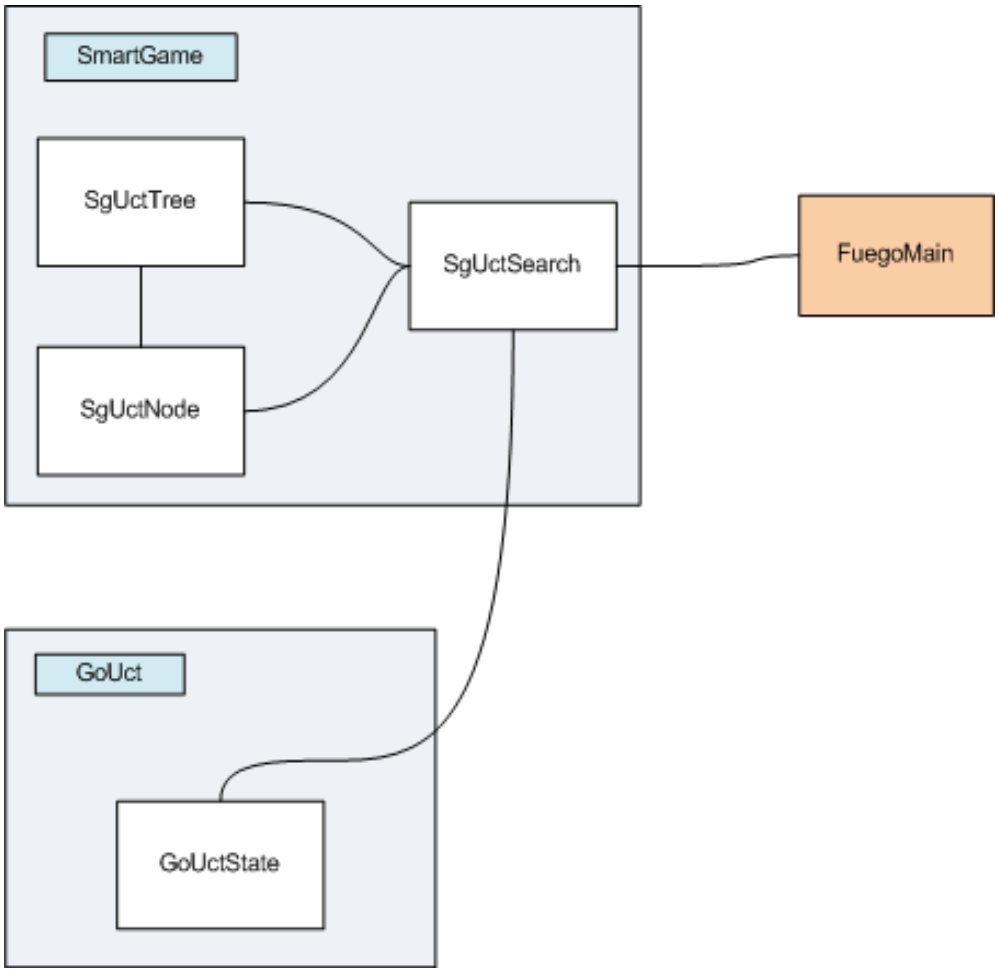


Ilustración 21: Componentes y relaciones de la búsqueda Monte-Carlo

Este es el espacio en el que se introducirán los cambios en la aplicación para incorporar la funcionalidad principal.

Las modificaciones a introducir en cada clase son las siguientes:

- *SgUctNode*:
  - Estructuras de datos de almacenamiento de mapas de influencia.
  - Operaciones de manejo de mapas de influencia:
    - Inicialización
    - Acceso
    - Copia
    - Representación
    - Unión
    - Diferencial
  - Actualizar operaciones de copia de nodos.
- *SgUctTree*:
  - Actualizar operaciones de actualización del árbol:
    - Creación de hijos.
    - Copia de ramas.
    - Añadir resultados de una partida.
- *GoUctState*:
  - Creación de mapas de influencia a partir de un estado.
- *SgUctSearch*:
  - Actualización de los mapas de influencia.

Para los cambios del tipo 2 de funcionalidades será necesario introducir cambios en el guardado de datos. Se utilizará el comando uct_savetree, que hace lo que necesitamos. En este caso se requiere adaptar este comando para que disponga de las funcionalidades que necesitamos.

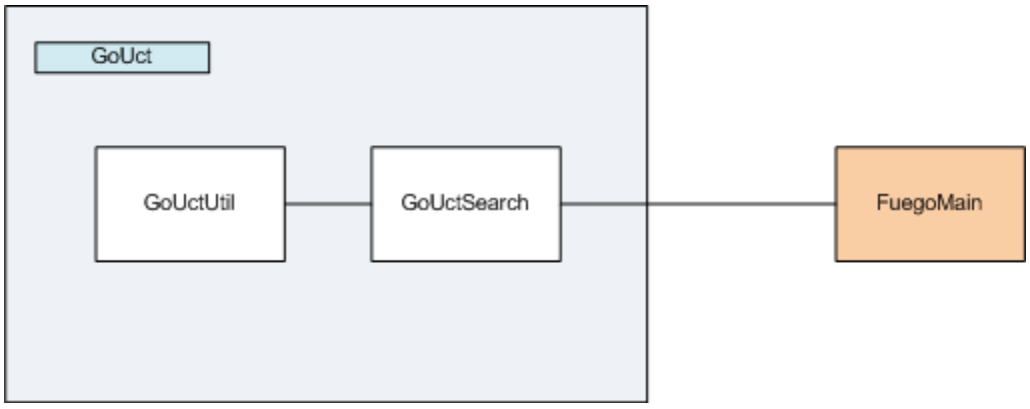


Ilustración 22: Componentes y relaciones del guardado de árboles

Las modificaciones a introducir en cada clase son las siguientes:

- GoUctSearch:
  - Modificar el paso de datos a GoUctUtil.
- GoUctUtil:
  - Crear cabecera de Weka.
  - Añadir funciones de análisis.
    - Mapa de libertades.
    - Analizador de patrones.
    - Contador de patrones.
    - Reconocedor de patrones.
  - Añadir guardado de nodos en formato Weka.
  - Añadir representación de mapas.
    - Mapas de influencia.
    - Diferenciales de mapas de influencia.

Con estas modificaciones se cubrirá la parte de salida.

En adición a todas las modificaciones es necesario crear nuevos comandos en GoUctCommands y GoGtpEngine para satisfacer los requisitos funcionales que faltan (Requisitos F-19, F-20 y F-21).

- GoUctCommands:
  - Uct_savetree2: Savetree modificado, incluye nueva información y omite el nodo raíz.
  - Uct_savetree_arff: Savetree modificado, devuelve el árbol en formato Weka.
  - Uct_analysis_results: Devuelve un fichero sgf, arff y un modelo.
- GoGtpEngine
  - Analyze: Carga una partida y genera un árbol de Monte-Carlo a partir de la partida

4.5.3. Diseño de bajo nivel

Como ya hemos diferenciado anteriormente, existen dos partes en la aplicación, el análisis y la representación de la salida. Atendiendo a las necesidades de modificación expuestas en el diseño de alto nivel se ha concretado un diseño de clases UML y la descripción de los métodos necesarios.

En primer lugar, se encuentra la parte de análisis de un tablero. Para mayor simplicidad se han omitido la mayoría de métodos que no tienen relación con el proyecto. Los métodos nuevos se marcarán con un asterisco.

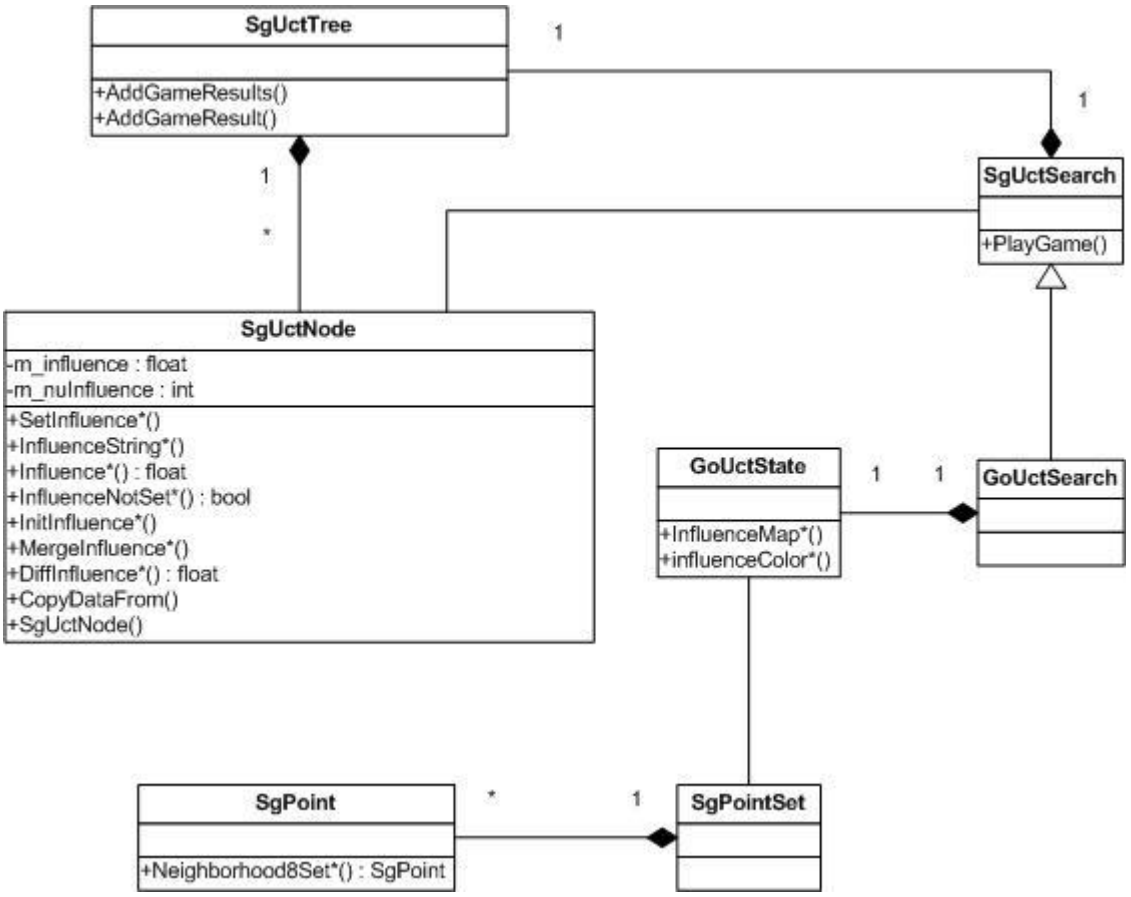


Ilustración 23: Diagrama UML: Análisis

A continuación se describen los métodos nuevos (Marcados con un *). Los métodos preexistentes no es necesario describirlos, solo se requiere tener en cuenta que tienen que ser modificados.

- *SetInfluence*:
    - Requisito: F-01 y F-02
    - Descripción: Establecer la influencia de un nodo.
    - Entradas: Mapa de influencia
    - Salidas: Sin salida
- *InfluenceString*:
    - Requisito: F-06 y F-09
    - Descripción: Representar el mapa de influencia de un nodo
    - Entradas: Mapa de influencia/ Sin entrada
    - Salidas: Representación del mapa de influencia
- *Influence*:
    - Requisito: F-01 y F-02
    - Descripción: Recuperar la influencia de un nodo
    - Entradas: Sin entrada
    - Salidas: Mapa de influencia
- *InfluenceNotSet*:
    - Requisito: F-01 y F-02
    - Descripción: Comprobar si la influencia está establecida
    - Entradas: Sin entrada
    - Salidas: True/False
- *MergeInfluence*:
    - Requisito: F-01 y F-02
    - Descripción: Unir la influencia del nodo actual y un mapa
    - Entradas: Mapa de influencia/Nodo
    - Salidas: Mapa de influencia
- *DiffInfluence*:
    - Requisito: F-08
    - Descripción: Diferencial de la influencia del nodo actual y un mapa
    - Entradas: Mapa de influencia/Nodo
    - Salidas: Mapa de influencia
- *InfluenceMap*:
    - Requisito: F-01
    - Descripción: Generar un mapa de influencia
    - Entradas: Nodo
    - Salidas: Sin salida
- *InfluenceMapColor*:
    - Requisito: F-01
    - Descripción: Generar un mapa de influencia para el color
    - Entradas: Mapa de influencia, Conjunto de movimientos, Color
    - Salidas: Sin salida

Por otra parte, la salida requiere también ciertos cambios. Al igual que con el primer diagrama, en la explicación se han omitido los métodos que no tienen que ver con el proyecto y los métodos nuevos se han marcado con un asterisco.

Ilustración 24: Diagrama UML: Salida

A continuación se describen los nuevos métodos

- *GoUctSearch::SaveTree2*:
  - Requisito: F-16
  - Descripción: Pasar los datos hacia las utilidades
  - Entradas: Profundidad máxima, Fichero de salida
  - Salidas: Sin salida
- *GoUctSearch::SaveTreeArff*:
  - Requisito: F-15
  - Descripción: Pasar los datos hacia las utilidades
  - Entradas: Profundidad máxima, Fichero de salida, Existe
  - Salidas: Sin salida
- *GoUctUtil::SaveTree2*:
  - Requisito: F-16
  - Descripción: Crear una cabecera sgf
  - Entradas: Árbol, Tamaño de tablero, Piedras, Jugador, Fichero de salida, Profundidad máxima
  - Salidas: Sin salida
- *GoUctUtil::SaveTreeArff*:
  - Requisito: F-15
  - Descripción: Crear una cabecera arff
  - Entradas: Árbol, Tamaño de tablero, Piedras, Jugador, Fichero de salida, Profundidad máxima, Existe
  - Salidas: Sin salida
- *SaveNode2*:
  - Requisito: F-04, F-05, F-06, F-07, F-09, F-11, F-14 y F-16
  - Descripción: Grabar un nodo sgf
  - Entradas: Fichero de salida, Árbol, Nodo, Jugador, Tamaño del tablero, Profundidad máxima, Profundidad
  - Salidas: Sin salida

- *SaveNodeArff:*
    - Requisito: F-04, F-05, F-06, F-07, F-09, F-11, F-14 y F-15
    - Descripción: Grabar un nodo arff
    - Entradas: Fichero de salida, Árbol, Nodo, Jugador, Tamaño del tablero, Profundidad máxima, Profundidad, Tablero, Patrones, Mapa diferencia.
    - Salidas: Sin salida
- *Include:*
    - Requisito: F-03
    - Descripción: Incluir un movimiento en un tablero
    - Entradas: Fichero de salida, Árbol, Nodo, Jugador, Tamaño del tablero, Profundidad máxima, Profundidad, Tablero, Patrones, Mapa diferencia.
    - Salidas: Sin salida
- *Exclude:*
    - Requisito: F-03
    - Descripción: Excluir un punto de un tablero
    - Entradas: Tablero, Tamaño de tablero, Movimiento, Jugador.
    - Salidas: Sin salida
- *RotateByNinetyToLeft:*
    - Requisito: F-13
    - Descripción: Rotar un tablero
    - Entradas: Tablero, Tamaño de tablero.
    - Salidas: Sin salida
- *ExistsPatternInPos:*
    - Requisito: F-12 y F-13
    - Descripción: Comprobar si existe un patrón en una posición dada.
    - Entradas: Tablero, Tamaño de tablero, Patrón, Coordenadas.
    - Salidas: True/False
- *ExistsPatternInPos:*
    - Requisito: F-12 y F-13
    - Descripción: Comprobar si existe un patrón en un tablero.
    - Entradas: Tablero, Tamaño de tablero, Patrón.
    - Salidas: True/False
- *CountPatternInBoard:*
    - Requisito: F-12 y F-13
    - Descripción: Contar un patrón en un tablero
    - Entradas: Tablero, Tamaño de tablero, Patrón
    - Salidas: Integer
- *LibertiesPoint:*
    - Requisito: F-10
    - Descripción: Contar las libertades alrededor de un punto.
    - Entradas: Tablero, Tamaño de tablero, Posición
    - Salidas: Integer

Para finalizar, se muestra el diagrama UML del diseño completo para las modificaciones de la herramienta Fuego. Se muestran ambas partes unidas de la siguiente manera:

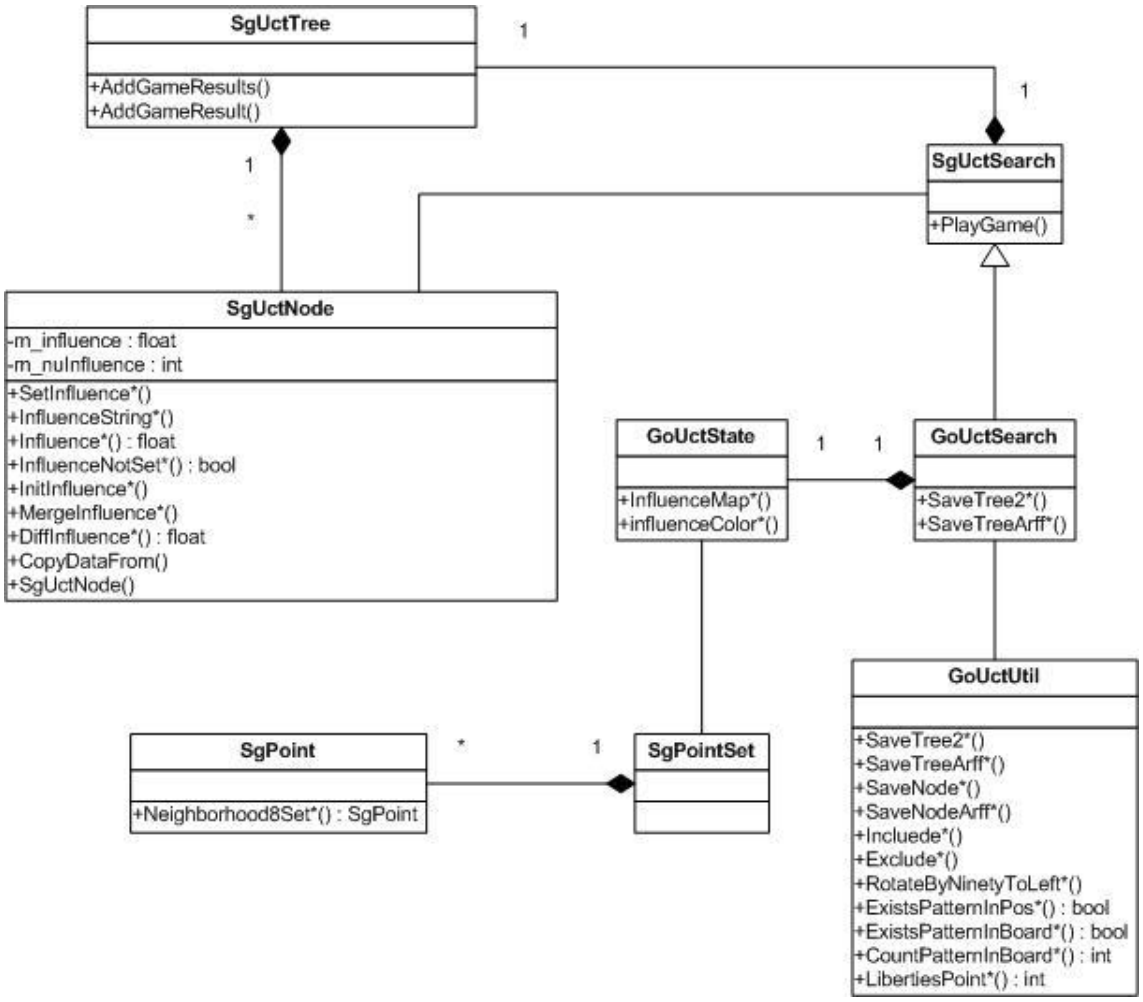


Ilustración 25: Diagrama UML: Completo

4.6. Trazabilidad de requisitos

A continuación se muestra una tabla para trazar los requisitos y comprobar que se han cumplimentado todos con el diseño. Se tendrán en cuenta los siguientes factores para la tabla:

- Métodos nuevos.
- Comandos nuevos.
- Funcionalidad Pre-existente en la herramienta.

	SetInfluence	*InfluenceString*	*Influence*	*InfluenceNotSet*	*MergeInfluence*	*DiffInfluence*	*InfluenceMap*	*InfluenceColor*
F-01	X		X	X	X		X	X
F-02	X		X	X	X			
F-03								
F-04								
F-05								
F-06		X						
F-07								
F-08						X		
F-09		X						

	GoUctSearch ::SaveTree2	*GoUctSearch ::SaveTreeArff*	*GoUctUtil ::SaveTree2*	*GoUctUtil ::SaveTreeArff*	*SaveNode2*	*SaveNodeArff*	*Include*	*Exclude*
F-01								
F-02								
F-03							X	X
F-04					X	X		
F-05					X	X		
F-06					X	X		
F-07					X	X		
F-08								
F-09					X	X		
F-10								
F-11					X	X		
F-12								
F-13								
F-14					X	X		
F-15		X		X		X		
F-16	X		X		X			

	RotateBy NinetyToLeft	*ExistPattern InPos*	*ExistPattern InBoard*	*CountPattern InBoard*	*LibertiesPoint*	***Existente en el sistema***	***Nuevos comandos***
F-01							
F-02							
F-03						X	
F-04						X	
F-05							
F-06						X	
F-07						X	
F-08							
F-09							
F-10					X		
F-11							

F-12		X	X	X			
F-13	X	X	X	X			
F-14							
F-15							
F-16						X	
F-17							X
F-18						X	
F-19						X	
F-20						X	
F-21							X
NF-01						X	
NF-02							
D-01						X	

Tabla 1: Tabla de trazabilidad de requisitos

5. Analizador de posiciones

5.1. Introducción

En este apartado del proyecto se describe el trabajo realizado a bajo nivel en la aplicación diseñada en el punto 4. Se aproximará la situación desde lo general a lo más específico. Se comienza explicando las distintas clases modificadas y, después se describen los distintos métodos y estructuras de datos que se han utilizado para conseguir la funcionalidad descrita.

A diferencia del diseño, es imprescindible comentar todos los cambios realizados sobre la herramienta, no solo los nuevos métodos. Esto se debe a que en este apartado se busca describir en detalle cómo funciona el sistema. También se buscará diferenciar el sistema actual, con sus nuevas funcionalidades, del sistema anterior, comentando y explicando en qué han cambiado con respecto a la versión base o inicial.

5.2. Clases

5.2.1. SgUctNode

En primer lugar y como base para el resto del proyecto se encuentra la clase *SgUctNode*. Esta clase tiene como función principal representar un nodo de un árbol de Monte-Carlo. Todo el manejo de Mapas de influencia se realiza en esta parte, la representación, unión, creación…

Este tipo de nodo es independiente del Go. Por tanto todos sus métodos son independientes del Go. Esto supone que es un nodo para realizar búsquedas Monte-Carlo para cualquier juego de dos jugadores.

Esta clase contenía inicialmente información del número de hijos, un puntero a su primer hijo, el movimiento realizado en el nodo actual, el valor RAVE, un contador del conocimiento, un contador de las veces que se ha llegado al nodo, si es una victoria o derrota probada y una cuenta de las derrotas virtuales. A excepción del movimiento, el contador de pasadas por el nodo y el valor RAVE, no se ha utilizado ninguna información más.

Añadiendo a la información existente se ha incluido una representación de un mapa de influencia y un contador de los mapas de influencia que se han añadido.

5.2.2. SgUctTree

Partiendo de la anterior clase *SgUctNode* se ha creado un árbol modificado en la clase *SgUctTree*. Como tal esta clase es un conjunto de nodos relacionados a partir de un nodo inicial o raíz. Este árbol además cuenta con un número de nodos limitados para generar búsquedas.

En esta clase se operan las actualizaciones de árbol y gestiona los enlaces entre los nodos. Aquí se ven representadas las operaciones de crear hijos del árbol o de añadir resultados de una partida.

Los nodos dentro del árbol se recorren a partir de un objeto *SgUctTreeIterator*, que es un iterador de nodos. Sin embargo, para recorrer el árbol completo se utilizarán las relaciones entre nodos de forma recursiva.

5.2.3. SgUctSearch

La clase *SgUctSearch* hace uso de un árbol de búsqueda del tipo *SgUctTree* para realizar los cálculos necesarios. Por tanto esta clase se encarga de realizar la búsqueda del mejor movimiento posible a partir de las partidas aleatorias que realiza el algoritmo de búsqueda.

Para la búsqueda, contiene información de la partida, el hilo al que pertenece y varias variables de control.

Entre sus funciones más relevantes se encuentra encontrar el mejor nodo hijo, encontrar la mejor secuencia, recoger valores y crear hijos. Las funciones más importantes son las que juegan la partida, la generación de jugadas, las jugadas dentro del árbol y el bucle de búsqueda.

Para incluir la nueva información se incluye el cálculo de mapas de influencia en estados finales en el momento en el que se llega a un estado final en las jugadas. Las jugadas se crean a partir de nodos que han sido probados como victoria o derrota. Que un nodo sea una victoria o derrota probada no implica que sea un estado final de la partida. Es posible que haya sido probado porque alguno de sus hijos haya propagado su estado hacia atrás. En este caso hay que calcular el estado final a partir de las jugadas y no de los nodos probados.

5.2.4. GoUctState

La clase *GoUctState* representa un estado de una búsqueda. A diferencia de las clases *Sg*, que son independientes del juego, la clase *Go* son específicas del juego del Go, por lo que pueden incluir operaciones específicas del juego en concreto.

Esta clase contiene información referente al estado de la partida: el tablero y la longitud de la partida. Debido a que contiene esta información, es el lugar en el que se calculan los mapas de influencia del estado final.

Una instancia de esta clase se almacena en las búsquedas para llevar cuenta del progreso del tablero.

5.2.5. GoUctSearch

La clase *GoUctSearch* hereda sus propiedades de *SgUctSearch* y aporta funciones específicas del juego del Go.

El principal interés de modificación de esta clase es que desde ella se pueden almacenar árboles en un fichero. Todas las referencias para guardar el árbol en un fichero se contienen en esta clase.

En realidad, el uso que se le da es de puente con *GoUctUtil.* A partir de *GoUctSearch* se conecta con *GoUctUtil* y se aportan todos los datos necesarios de la búsqueda en árbol.

5.2.6. GoUctUtil

La clase *GoUctUtil* ofrece un conjunto de utilidades específicas del juego del Go y la búsqueda en árbol de Monte-Carlo.

Entre todas las utilidades que ofrece se encuentra guardar un árbol en un fichero de salida.

5.2.7. SgPoint

La clase *SgPoint* contiene una representación de un punto para un juego de tablero. La representación del tablero se hace a través de enteros. Todas las posiciones posibles en un tablero de 19x19 se pueden representar mediante una estructura de datos de tipo *SgPoint*. La representación muestra los puntos en el formato de SgPoint. Cada entero es la posición que indica, por ejemplo, la posición B2 se representa como 42, la posición G4 se representa como 87 y T19 se representa como 399. Estas direcciones se obtienen a través del siguiente diagrama:

```
19|381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399
18|361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379
17|341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359
16|321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339
15|301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319
14|281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299
13|261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279
12|241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259
11|221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239
10|201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219
 9|181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199
 8|161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179
 7|141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
 6|121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139
 5|101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119
 4| 81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99
 3| 61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79
 2| 41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59
 1| 21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39
   [A] [B] [C] [D] [E] [F] [G] [H] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T]
```

Ilustración 26:Tablero representado mediante SgPoint

En el caso de que el tablero fuera de menor tamaño o no fuese cuadrado, únicamente sería necesario quitar columnas hasta tener el número adecuado. Se utiliza esta representación fija del tablero debido a que es lo más general posible y no requiere de hacer transformaciones matemáticas para navegar entre puntos del tablero.

Para moverse entre puntos del tablero se usa el siguiente gráfico:

```
                p + SG_NS
p - SG_WE         p        p + SG_WE
                p - SG_NS
```

Ecuación 4: Movimientos entre puntos

SG_XX es un número que indica a cuanto está la casilla adyacente en la dirección deseada. En este caso los número laterales (SG_WE) son 1 y los horizontales (SG_NS) 20. La dirección en la que moverse viene determinado por los números indicados y si es una resta o una suma.

5.3. Funciones y estructuras de datos

5.3.1. Modificados

5.3.1.1. SgUctNode

- ***SgUctNode()***

Como el constructor de la clase es necesario inicializar los valores de todos los atributos de clase. En este caso se han añadido dos atributos nuevos *m_influence* y *m_nuInfluence*, los cuales es necesario tener en cuenta en la representación.

m_influence y *m_nuInfluence* se inicializan a "null" y a 0 respectivamente.

- ***CopyDataFrom()***

De manera similar al caso del constructor, es necesario añadir la copia de los nuevos valores al método para que, al copiar los nodos, se mantengan los mapas de influencia ya creados.

Los nodos se copian cuando se quiere reutilizar un árbol, se reutilizan ramas ya exploradas para hacer una búsqueda más rápida. Copiando los mapas de influencia además del resto de información se incorporan los cambios al proceso de reutilización.

5.3.1.2. SgUctTree

- ***AddGameResutls()***

Añade los resultados de una partida una cantidad determinada de veces. Aparte de añadir los resultados antiguos de la partida, ha sido modificado para añadir el mapa de influencia a los resultados de la partida.

Esta función es llamada por *SgUctSearch::UpdateTree()* para mantener actualizados todos los nodos del árbol.

- ***AddGameResult()***

Añade un resultado individual de una partida. Al igual que *AddGameResutls()* se ha incorporado la adición del mapa de influencia.

Ambas funciones se han modificado de la misma manera, para que, si existe un nodo padre añadir los resultados de la influencia al nodo en cuestión y mezclarlos con la información anterior de la influencia.

5.3.1.3. SgUctSearch

- ***PlayGame()***

PlayGame() es el método encargado de jugar una partida. Es una función compleja con muchas operaciones y su estructura es la siguiente.

En primer lugar se inicializan una serie de variables y seguidamente se hace la fase dentro del árbol (In-tree). Una vez se ha acabado se comprueba si la lista de movimientos no está vacía y el nodo al que se llega es terminal, en este caso se evalúa la partida de la fase interior del árbol.

Si la lista de movimientos no está vacía y el último nodo de dicha lista está probado (es victoria o derrota) se evalúan las distintas secuencias de la partida de la fase interior de árbol. Por el contrario si esto no ocurre, se ejecuta la fase de jugadas. Cuando acaba esta fase se revierten los cambios en el árbol y se actualiza el árbol con la información recogida.

La modificación que se ha introducido es en la fase de jugadas. En esta fase es donde se recogen más resultados finales de partidas porque son jugadas rápidas sin repercusión en el árbol y se generan más que nodos finales, de lo que se obtiene más información precisa sobre el reparto de territorios.

La recogida del mapa de influencia se hace en el lugar en el que se evalúa una jugada. Previamente a evaluar se genera el mapa de influencia. El mapa de influencia es independiente de la evaluación realizada por *Fuego* y no utiliza ninguno de sus datos, por lo que funciona como un ente independiente.

Una vez generado el mapa de influencia *PlayGame()* hace una llamada a *UpdateTree()* para actualizar todos los nodos del árbol con la información recogida. A su vez *UpdateTree()* hace uso del método *AddGameResults()* para actualizar por los nodos, de esta manera se logran propagar los mapas de influencia creados por los nodos del árbol que lleguen.

5.3.2. Nuevos

5.3.2.1. SgUctNode

- ***M_influence***

M_influence es un vector lineal que representa un mapa de influencia. En este caso la representación es mediante un entero de 0 a 360, para indicar las 361 posiciones de un tablero de 19x19. Se interpreta desde la esquina inferior izquierda que es el punto A1, luego B1 y así sucesivamente hasta completar la fila 1 en T1. Por ejemplo, el punto A1 se representa como 0, C3 se representa como 41 y T19 como 360. Cada dirección se obtiene a partir del siguiente diagrama:

```
19|  342  343  344  345  346  347  348  349  350  351  352  353  354  355  356  357  358  359  360
18|  323  324  325  326  327  328  329  330  331  332  333  334  335  336  337  338  339  340  341
17|  304  305  306  307  308  309  310  311  312  313  314  315  316  317  318  319  320  321  322
16|  285  286  287  288  289  290  291  292  293  294  295  296  297  298  299  300  301  302  303
15|  266  267  268  269  270  271  272  273  274  275  276  277  278  279  280  281  282  283  284
14|  247  248  249  250  251  252  253  254  255  256  257  258  259  260  261  262  263  264  265
13|  228  229  230  231  232  233  234  235  236  237  238  239  240  241  242  243  244  245  246
12|  209  210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  225  226  227
11|  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208
10|  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189
 9|  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170
 8|  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151
 7|  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132
 6|   95   96   97   98   99  100  101  102  103  104  105  106  107  108  109  110  111  112  113
 5|   76   77   78   79   80   81   82   83   84   85   86   87   88   89   90   91   92   93   94
 4|   57   58   59   60   61   62   63   64   65   66   67   68   69   70   71   72   73   74   75
 3|   38   39   40   41   42   43   44   45   46   47   48   49   50   51   52   53   54   55   56
 2|   19   20   21   22   23   24   25   26   27   28   29   30   31   32   33   34   35   36   37
 1|    0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16   17   18
     [A]  [B]  [C]  [D]  [E]  [F]  [G]  [H]  [J]  [K]  [L]  [M]  [N]  [O]  [P]  [Q]  [R]  [S]  [T]
```

Ilustración 27: Representación interna del mapa de influencia

Como se puede apreciar no coincide con la representación de *SgPoint*. No se ha seguido la representación estándar para: hacer coincidir la representación de la posición en el vector y el punto que representa; independizar la representación del tablero y el mapa de influencia; facilitar la transformación a una cadena de texto; e incluir valores de coma flotante en la representación.

La representación de cada punto de influencia se mide con un número en coma flotante en una posición determinada del tablero. El número que representa la influencia puede tomar valores entre 0 y 1. Cuanto más se aproxime a 0 la influencia de la casilla pertenecerá al jugador que tenga el color negro; mientras que si se aproxima más a 1 la influencia de la casilla pertenecerá al jugador blanco; y en el caso que sea una casilla en disputa el número que representa la influencia se aproximará a 0.5.

- ***M_nuInfluence***

*M_nuInfluence*es un valor Integer positivo que indica cuántos mapas de influencia han llegado al nodo. Un nodo puede haber recibido múltiples mapas de influencia por la creación inicial del mapa de influencia (que se hace mediante jugadas y hace varios mapas) o por la propagación de los mapas de influencia de niveles inferiores del árbol de búsqueda.

En cualquier caso, se utiliza para poder calcular la media aritmética y mezclar mapas de influencia entrantes.

- ***SetInfluence()***

Este método sustituye el mapa de influencia actual con un nuevo mapa de influencia. Se utiliza para la primera asignación del mapa de influencia.

- ***InfluenceString()***

Este método recoge un mapa de influencia, o el mapa de influencia del nodo y lo transforma en una representación visual del mismo. El número representativo de la influencia que puede tomar valores entre 0 y 1 tiene unos valores determinados para asignarles un símbolo en la representación gráfica.

Existen los siguientes símbolos dependiendo de la influencia:

- Valores mayores de 0.8: O
- Valores entre 0.8 y 0.6: o
- Valores entre 0.6 y 0.4: .
- Valores entre 0.4 y 0.2: x
- Valores menores de 0.2: X

Los distintos valores representan distintos grados de dominación de cierto color, las mayúsculas (O y X) representan una fuerte dominación de la casilla del color correspondiente (blanco y negro respectivamente); las minúsculas (o y x) representan un control parcial sobre la casilla del color correspondiente; y finalmente el punto (.) representa una casilla en disputa.

Para la representación se sigue toda una línea del tablero y cuando se llega al final de la línea se coloca un salto de línea y se ubica al principio del flujo de salida. Se hace de esta manera para que sea posible imprimir el tablero desde la fila inferior a la primera fila.

InfluenceString() se aplica para representar los mapas de influencia en el formato sgf, que no requiere análisis de datos. Para el formato arff no se utiliza ya que es preferible usar atributos numéricos.

- ***Influence()***

Método que devuelve el mapa de influencia del nodo. Se utiliza para hacer accesos al mapa desde clases externas.

- ***InfluenceNotSet()***

Método que comprueba si la influencia no ha sido todavía inicializada. El método comprueba si *m_influence*es de tamaño 0. Si el vector es de tamaño 0 o nulo implica que todavía no se ha inicializado el mapa de influencia. Cualquier otro tamaño implica que ya se ha inicializado. Se utiliza para comprobar cuándo inicializar la influencia.

- ***InitInfluence()***

Método que inicializa la influencia. Da tamaño y valores 0.5 a todas las posiciones del vector. Como la influencia no ha sido definida todavía en este punto se considera que todas las posiciones están en disputa. A la hora de crear la mezcla de dos influencias esto simplifica la comprobación de si se ha definido por primera vez la influencia o solo se ha inicializado.

- ***MergeInfluence()***

Método que mezcla dos mapas de influencia. El proceso de asignación es el siguiente. Primero se comprueba si la influencia esta inicializada, en caso de que no lo esté se inicializa y se establece el mapa de influencia según ha sido indicado en los parámetros del método. Si la influencia ya ha sido definida se calcula la media aritmética del mapa de influencia y se mezclan los resultados. Para la mezcla de los resultados, se aplica la siguiente fórmula a cada una de las posiciones del vector:

$$\boldsymbol{Inf_i = \frac{Inf_i \cdot nInf + New_i}{nInf + 1}}$$

Ecuación 5: Unión de influencias

- ***Inf***: Influencia actual
- ***nInf***: Numero de mapas de influencia
- ***New***: Nueva entrada de influencia
- ***i***: Posición en el vector.

Aplicando la fórmula descrita se calcula la media de todos los mapas existentes de manera incremental, solo se necesita el número de mapas de influencia que han sido calculados y el mapa de influencia actual. Gracias a que se almacena *m_nuInfluence* es posible hacer este cálculo. Al final de

cada adición de un mapa de influencia se incrementa el número de mapas, independientemente de si es la primera adición o de que sean las adiciones sucesivas de distintos mapas.

- ***DiffInfluence()***

Método que calcula la diferencia entre un mapa de influencia entrante y el mapa de influencia del nodo actual. La diferencia que se realiza está normalizada para que se mantenga en el mismo rango que la influencia es decir, que la diferencia siempre estará entre 0 y 1. La diferencia se normaliza para que sea fácilmente cuantificable, en caso de no estar normalizada los valores variarían entre 1 y -1, lo cual complica la legibilidad de la diferencia.

Para la normalización y la resta simultáneas se ha utilizado la siguiente fórmula:

$$\boldsymbol{Diff_{norm} = \frac{Diff - Diff_{min}}{Diff_{max} - Diff_{min}}}$$

Ecuación 6: Diferencia de influencias

- $\boldsymbol{Diff_{norm}}$: Diferencia normalizada.
- $\boldsymbol{Diff_{min}}$: Resultado mínimo de la diferencia.
- $\boldsymbol{Diff_{max}}$: Resultado máximo de la diferencia.
- $\boldsymbol{Diff}$: Diferencia original.

La diferencia entre dos números de influencia es la diferencia entre dos números entre 0 y 1. Evaluando los casos extremos:

$\boldsymbol{Inf_1 - Inf_2}$	$\boldsymbol{Inf_1 = 0}$	$\boldsymbol{Inf_1 = 1}$
$\boldsymbol{Inf_2 = 0}$	0	1
$\boldsymbol{Inf_2 = 1}$	-1	0

De lo que se obtiene que, el mínimo obtenible de la diferencia es -1 y el máximo es 1. Por ello se puede sustituir con $\boldsymbol{Diff_{min}} = -1$, $\boldsymbol{Diff_{max}} = 1$ y $\boldsymbol{Diff} = Inf_1 - Inf_2$. Se obtiene la siguiente ecuación:

$$\boldsymbol{Diff_{norm} = \frac{Inf_1 - Inf_2 + 1}{2}}$$

Ecuación 7: Diferencia de influencias simplificada

Siempre que se utiliza este método se calcula para estimar la diferencia entre un nodo padre y un nodo hijo. Cifras cercanas a 0.5 implican que el mapa se ha mantenido constante en esa posición, si se aproximan a 1 implican que el blanco ha perdido control sobre esa posición de manera abrupta, por el contrario si se aproximan a 0 implica que el negro ha perdido esa influencia.

5.3.2.2. GoUctState

- ***InfluenceMap()***

InfluenceMap() e *InfluenceMapColor()* son los métodos más importantes de la aplicación. Suponen la generación del mapa de influencia en una posición final de un tablero. Este cálculo es más sencillo que el cálculo de la influencia en cualquier otro punto de la partida.

El proceso que sigue *InfluenceMap()* es el siguiente. Primero, se recupera el tablero *m_uctBd*(Tablero utilizado para la fase de jugadas) y se transforma a dos conjuntos de piedras, un conjunto de piedras blancas y un conjunto de piedras negras. A continuación se inicializa un vector de influencia con todos los valores en 0.5 y se calculan los mapas de influencia para cada color por separado con *InfluenceMapColor().*

Antes de finalizar se truncan los valores a sus valores posibles más cercanos (0, 0.5 ó 1). Finalmente, si la influencia del nodo de entrada no está establecida se inicializa, de lo contrario se mezcla con la influencia existente.

- ***InfluenceMapColor()***

*InfluenceMapColor()*realiza el cálculo interno de un mapa de influencia para un conjunto de piedras de un color. Para asignar influencia al mapa se sigue el siguiente proceso. Se comienza un bucle que recorre todos los movimientos, excluyendo una piedra cada vez que se procesa hasta que el conjunto de todas las piedras está completamente vacío.

Procesar una piedra implica lo siguiente: transformar el punto a una posición del vector de influencia. Si el color es negro se resta 1 a la influencia de esa posición. Si el color es blanco se suma 1 a la influencia de la posición. De esta forma los puntos en disputa serán 0.5, y el resto estarán completamente dominados en todo caso porque es el final de la partida. Cuando se ha hecho la adición o la sustracción se recoge el conjunto de puntos adyacentes a la posición que se está calculando mediante *Neighborhood8Set()*.

De nuevo, con el conjunto de posiciones adyacentes, se vuelven a procesar las posiciones obtenidas de la misma manera sin generar un conjunto de casillas adyacentes.

La siguiente figura muestra dos piedras, una blanca y otra negra. La blanca expande su influencia en sus ocho casillas adyacentes y a la casilla sobre la que está. La piedra negra hace exactamente lo mismo, pero sustrae 1 a la hora de expandir su influencia. El resto de casillas que no quedan adyacentes quedan neutrales.

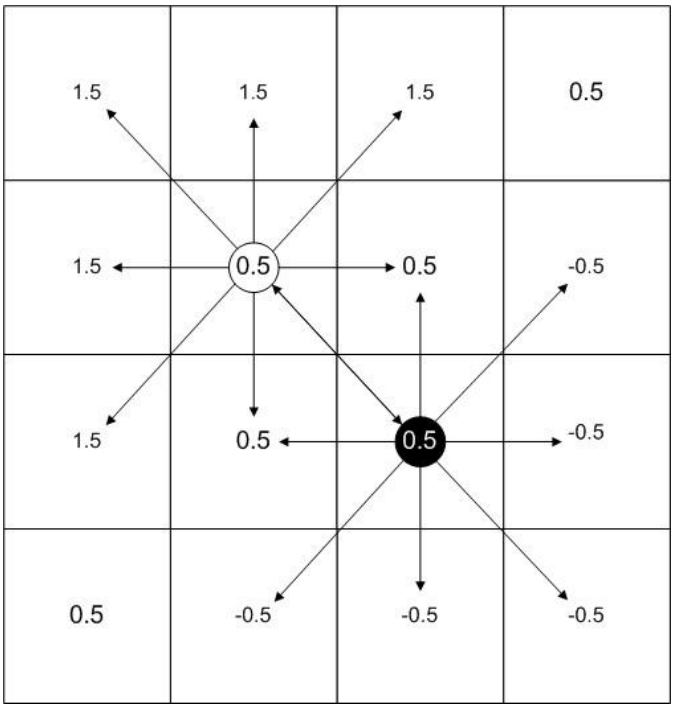


Ilustración 28: Ejemplo de generación de un mapa de influencia

5.3.2.3. SgPoint

- ***Neighborhood8Set()***

Neighborhood8Set() usa un número del 0 al 7 asignado a cada casilla en este orden:

```
3|  0 1 2
2|  3 X 4
1|  5 6 7
   [A][B][C]
```

Ilustración 29: Vecindario

Se utiliza en un bucle para recoger un conjunto de puntos por lo que se utilizan consecutivamente. Para hallar cada punto se hacen operaciones en referencia a la representación del tablero mediante las siguientes:

```
p + SG_NS - SG_WE     p + SG_NS      p + SG_NS + SG_WE
    p - SG_WE             p              p + SG_WE
p - SG_NS - SG_WE     p - SG_NS      p - SG_NS + SG_WE
```

Ecuación 8: Movimiento expandido entre posiciones

Uniendo ambos resultados obtenemos el siguiente conjunto de ecuaciones para obtener los puntos:

$$\begin{aligned} p_0 &= p + SG_{NS} - SG_{WE} \\ p_1 &= p + SG_{NS} \\ p_2 &= p + SG_{NS} + SG_{WE} \\ p_3 &= p - SG_{WE} \end{aligned} \qquad \begin{aligned} p_4 &= p + SG_{WE} \\ p_5 &= p - SG_{NS} - SG_{WE} \\ p_6 &= p - SG_{NS} \\ p_7 &= p - SG_{NS} + SG_{WE} \end{aligned}$$

Ecuación 9: Cálculo de los puntos

5.3.2.4. GoUctSearch

- ***SaveTree2()***

SaveTree2() es una copia de *SaveTree()* que redirige al *GoUctUtil::SaveTree2()*.

- ***SaveTreeArff()***

SaveTreeArff() es una copia de *SaveTree()*ampliada con información de si existe el fichero o no que redirige al *GoUctUtil::SaveTree2()*.

5.3.2.5. GoUctUtil

- ***SaveTree2()***

SaveTree2() es una copia de *SaveTree()* que redirige a *SaveNode2()*

- ***SaveTreeArff()***

SaveTreeArff() incluye una cabecera *arff*, genera algunos datos necesarios para analizar posteriormente y redirige a *SaveNodeArff()*. Aquí se incluyen los patrones que se utilizarán en el guardado de nodos. Los patrones utilizados son los siguientes:

- Ojo:

```
   3|  11 1
  2|  1  0.51
   1|  11 1
     [A][B][C]
```

- Cuadrado:

```
   3|  1  1
   2|  1  1
      [A][B]
```

- Doble ojo:

```
5|   1   1   1  0.5 0.5
4|   1  0.5  1  0.5 0.5
3|   1   1   1  0.5 0.5
2|   1  0.5  1  0.5 0.5
1|   1   1   1  0.5 0.5
    [A] [B] [C] [D] [E]
```

- Ojo invadido:

```
3|  0   1   1
2|  1  0.5  1
1|  1   1   0
[A] [B] [C]
```

- Ojo incompleto:

```
3| 0.5  1  0.5
2|  1  0.5  1
1| 0.5  1  0.5
   [A] [B] [C]
```

Los patrones siguen una representación concreta, 1 y 0 son colores opuestos, 1 no es necesariamente blanco, pero si 1 es blanco 0 es obligatoriamente negro y si 0 es blanco 1 es obligatoriamente negro. Esto permite hacer comprobaciones independientes del color.

Después de generar los patrones descritos anteriormente se leen las piedras que entran por parámetros y lo transforman en una representación de un tablero.

Cuando se genera la representación del tablero se añaden las cabeceras del fichero arff que son:

- Jugador que tiene que mover
- Movimiento
- Mapa de influencia
- Diferencia de influencia
- Tablero
- Mapa de libertades
- Cuenta de movimientos
- Cuenta de posiciones
- Cuenta de ojos
- Mapas de patrones por cada patrón

    - Valor RAVE
    - Proporción de victorias.

- ***SaveNode2()***

SaveNode2() guarda un nodo y se llama recursivamente para recorrer todo el árbol de búsqueda.

Se almacena la información mediante el siguiente proceso. Primero se almacena el mapa de influencia, la cuenta de movimientos, la cuenta de posiciones y la proporción de victorias.

Para cada hijo del árbol se imprime la diferencia con el nodo padre, imprimiendo el movimiento, el jugador que mueve y el mapa de diferencia en sí. El mapa diferencia representa qué jugador ha perdido influencia en qué medida como se ha explicado en *DiffInfluence()*

Cuando se ha imprimido toda la información se pasa a almacenar el siguiente nodo hijo en profundidad.

Existe una limitación de profundidad y, en el caso de que se exceda, se aborta la expansión en esa rama.

- ***SaveNodeArff()***

De la misma manera en la que *SaveNode2()* y SaveNode() guardan nodos *sgf*, *SaveNodeArff()* guarda un nodo *arff* y se llama recursivamente para recorrer todo el árbol de búsqueda.

A diferencia del archivo *sgf* que se guarda como un árbol de juego, esta función genera una instancia por cada instante de partida.

Se utiliza el siguiente proceso para guardar una instancia *arff*. Primero se omite el nodo raíz. Esto sucede para almacenar los mapas diferencia en los hijos con respecto al padre. En *SaveNode2()* los mapas diferencia se almacenaban todos con el nodo padre. En esta función se almacena cada mapa diferencia en el nodo hijo. Por tanto el nodo razón que no es hijo de nadie no contendría mapa diferencia y dejaría información incompleta.

Tras omitir el primer nodo se comienzan a imprimir los atributos en el orden descrito en *SaveTreeArff()*.

Cuando se han imprimido los atributos se añade una nueva línea con el nodo hijo recursivamente. Entre el padre y el hijo se actualiza el tablero, incluyendo y excluyendo la piedra del movimiento del hijo.

- ***Include()***

Include() incluye un punto en la representación del mapa, añade un 1 o un 0 (dependiendo del color blanco o negro respectivamente) en la posición indicada de la representación.

- ***Exclude()***

*Exclude()*excluye un punto en la representación del mapa, añade un 0.5 en la posición indicada de la representación.

- ***RotateByNinetyToLeft()***

RotateByNinetyToLeft() rota una matriz 90 grados. El método opera por anillos, primero el anillo exterior y profundizando sucesivamente cuando un anillo ha sido rotado por completo.

En la siguiente figura se representa como se opera sobre la matriz:

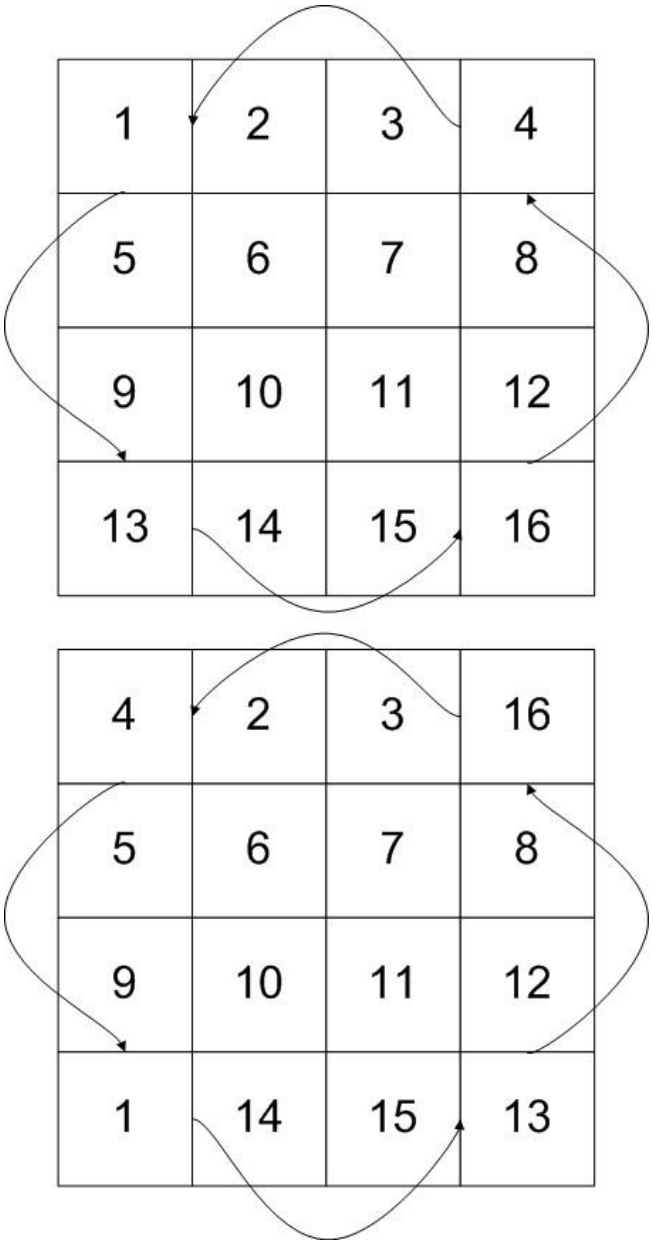


Ilustración 30: Primera iteración del algoritmo de rotado

- ***ExistsPatternInPos()***

El método *ExistsPatternInPos()* comprueba si en una posición existe un patrón a partir de la posición se examinan las casillas inferiores a la derecha con una extensión dependiente del tamaño del patrón.

Además también se comprueban las distintas rotaciones del patrón, esto ayuda a no excluir los patrones que no son exactamente de la misma forma que el original, pero están rotados.

Se hacen 2 comprobaciones simultáneas, una para el caso que el número 1 del patrón signifique blanco y otra para el caso en el que el número 0 del patrón signifique negro. Es decir, se mira si coincide con el tablero y luego se mira si es directamente opuesto al tablero.

- ***ExistsPatternInBoard()***

Para comprobar si existe un patrón en el tablero con *ExistPatternInBoard()*sencillamente se hace un bucle de todo el tablero (ajustado al tamaño del patrón) y se comprueba cada posición del tablero. Si en algún momento se da el caso de que se reconoce un patrón, se para y sale de la función.

- ***CountPatternInBoard()***

Este método solo se aplica para el patrón de ojo. *CountPatternInBoard()* recorre todo el tablero en busca de un patrón. Cada vez que hay una ocurrencia de un patrón se añade 1 a un contador de patrones. Es independiente que el patrón sea negro o blanco, solo importa que exista para que entre en la cuenta.

- ***LibertiesPoint()***

LibertiesPoint() calcula en un punto, el número de libertades de una casilla. Una libertad implica una casilla no ocupada adyacente a una piedra. Toda casilla desocupada se cataloga como -1 pues no afectan a las libertades.

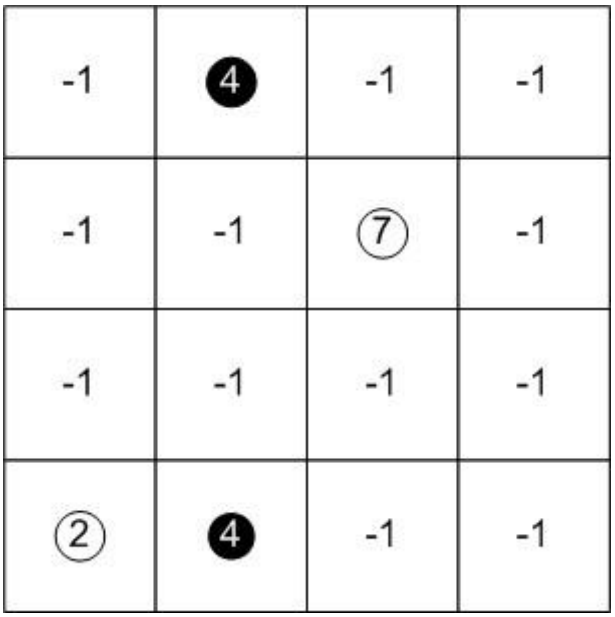


Ilustración 31: Ejemplo de mapa de libertades

5.4. Comandos

5.4.1. GoUctCommands

- ***Uct_savetree2:***

Comando que llama al método *SaveTree2()* para guardar un árbol de tipo en un fichero de tipo *sgf*.

- ***Uct_savetree_arff:***

Comando que llama al método SaveTreeArff() para guardar un árbol en un fichero de tipo *arff*

- ***Uct_analysis_results***

Comando que llama a SaveTree2() para guardar un árbol en un fichero de tipo sgf. Después llama a SaveTreeArff() para guardar un árbol en un fichero de tipo arff.

Con el último fichero se hace una llamada a Weka en dos partes. Primero se comienza con la transformación del conjunto de datos a un nuevo conjunto realizando selección de atributos. En segundo lugar se pasa el análisis sobre los datos mediante un árbol de regresión M5P.

5.4.2. GoGtpEngine

- ***Analyze:***

Comando que a partir de un tablero genera un árbol de Monte-Carlo. En realidad es una expansión del comando *GenMove* pero incorpora la carga de un fichero de manera automática y supone que el movimiento es el del jugador que tiene el turno.

6. Experimentación

6.1.Metodología

Para la experimentación se utilizará la herramienta descrita en el punto 5. Para la realización de los experimentos se subdividirá la experimentación en 3 fases diferenciadas. La estructura de la experimentación es la siguiente:

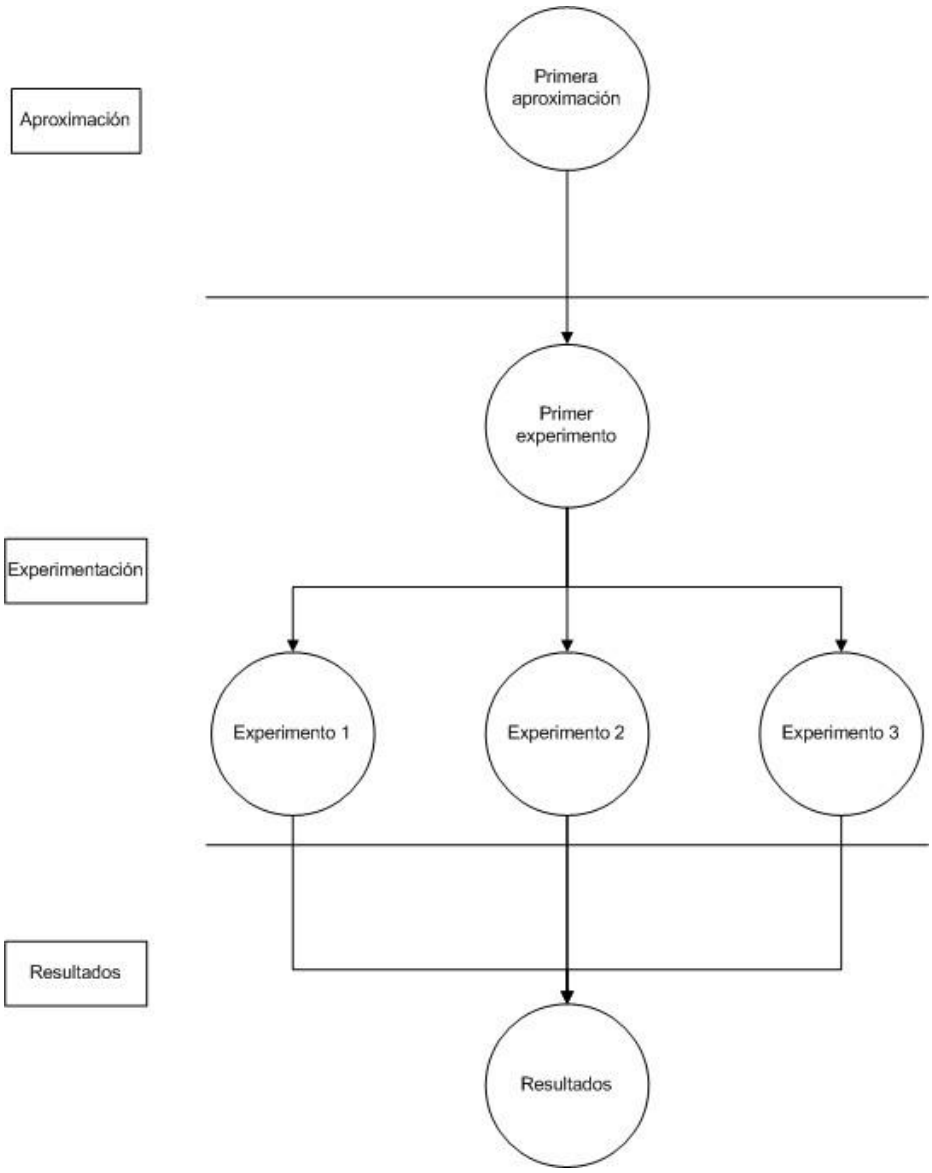


Ilustración 32: Diagrama de la experimentación

6.1.1. Fase de aproximación

La fase de aproximación consiste en tener una primera idea de cómo rinden exactamente los algoritmos de análisis de datos, previamente a utilizarlos en un experimento real.

Aquí es donde se deben empezar a descartar algoritmos y transformaciones para mejorar la efectividad y eficiencia de los futuros experimentos.

Para esta fase se utilizará un tablero generado de manera arbitraria para que:

- Existan algunos patrones sobre el mapa.
- Haya un claro jugador que domine la partida.

Mediante este conjunto podemos establecer un punto de referencia para las siguientes fases e iterar a partir de este conjunto.

6.1.2. Fase de experimentación

La fase de aproximación es la fase central del proceso, donde se generarán resultados para su futuro análisis. En este caso se subdivide en dos partes, un primer experimento y la batería de experimentos.

Para los distintos experimentos se definirán:

- Tablero utilizado
- Filtros aplicados
- Algoritmos aplicados

De lo resultados se obtendrá:

- Coeficiente de correlación

6.1.2.1. Fase de experimentación: Primer experimento

Para obtener un punto de referencia a partir del que comparar se generará un primer experimento con todo lo aprendido en la aproximación y con el mapa generado en dicha fase.

Con estos resultados se pueden discernir con mayor precisión que algoritmos y transformaciones funcionan sobre el conjunto de datos del Go.

6.1.2.2. Fase de experimentación: Batería de experimentos

En la batería de experimentos se llevarán a cabo una serie de experimentos sobre tableros generados a partir de casos de juego reales y problemas difíciles [3]. A diferencia de la primera aproximación y el primer experimento, no se utilizará el tablero de referencia. Esto se debe a que en este punto se quiere investigar la efectividad real del análisis de datos.

Realizar 3 pruebas sobre tableros distintos asegura que los resultados no estén sesgados por algún tipo de irregularidad en el tablero y contrastar procesos iguales sobre casos específicos distintos.

6.1.3. Fase de resultados

La fase de resultados consiste en recoger todos los resultados en la fase de experimentación y analizarlos. Analizar los resultados implica comprobar si ha sido efectiva la experimentación y si los resultados obtenidos son significativos.

6.2. Primera aproximación

En la primera aproximación se ha utilizado el mapa de prueba generado de manera arbitraria para hacer las primeras pruebas. El tablero es el siguiente:

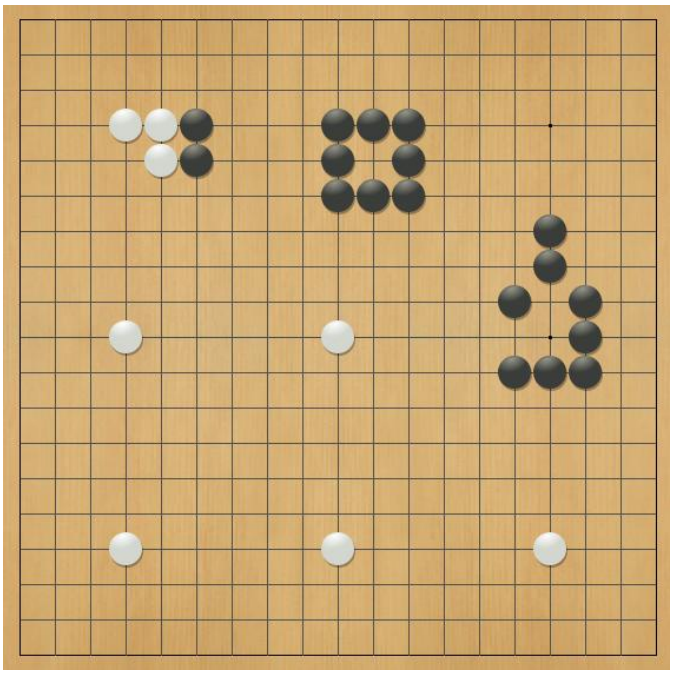


Ilustración 33: Tablero referencia

Se han utilizado los siguientes atributos. N representa el lado del tablero y la cifra entre paréntesis el número de atributos.

- Jugador que tiene que mover *(1)*
- Movimiento *(1)*
- Mapa de influencia *(N^2)*
- Diferencia de influencia *(N^2)*
- Tablero *(N^2)*
- Mapa de libertades *(N^2)*
- Cuenta de movimientos *(1)*
- Cuenta de posiciones *(1)*
- Cuenta de ojos *(1)*
- Mapas de patrones por cada patrón *($5 \cdot N^2$)*
- Valor RAVE *(1)*

- Clase: Proporción de victorias. *(1)*

Por tanto la suma total de atributos es de: $9 \cdot N^2 + 7$. Suponiendo un tablero con $N = 19$, el número de atributos utilizados es de 3256.

Para este conjunto se han utilizado 941 instancias.

Para crear el conjunto inicial de datos se ha empleado la herramienta *Fuego* con las modificaciones descritas. En primer lugar es necesario cargar el tablero descrito anteriormente a *Fuego* con el comando *loadsgf* y después generar un movimiento para que se haga la búsqueda mediante *genmove*. Una vez el árbol ha sido generado se utiliza el comando *uct_savetree_arff* para generar un fichero arff con la información de los nodos del árbol de Monte-Carlo.

Estos datos se han introducido a *Weka* y utilizando el *explorer* se ha analizado el fichero. Primero, el atributo "movimiento" no permite hacer regresión, por lo cual se retira del conjunto de datos. Una vez retirado se comienza a probar el conjunto con distintos algoritmos como:

- Perceptrón multicapa: Este algoritmo utiliza capas de nodos totalmente conectadas hacia la capa siguiente hasta alcanzar la salida. Se mapean los atributos a los nodos que sirven de entrada y, en regresión de una sola clase, existe un nodo salida que recoge el resultado de los cálculos de la red neuronal.
- Regresión lineal: Este algoritmo utiliza los atributos para explicar linealmente la clase.
- M5P[15][16]: Este algoritmo genera árboles de decisión que contienen dos tipos de nodos. El primero son nodos de decisión que subdividen el árbol con una comprobación binaria. El segundo tipo son los nodos terminales que contienen modelos lineales de decisión.
- M5Rules[14][15][16]: Este algoritmo es similar a M5P. Genera un conjunto de reglas para realizar la decisión. Hay reglas que subdividen el conjunto de reglas en dos mediante una comprobación binaria y reglas que contienen modelos lineales.
- IBk[17]: Este algoritmo almacena todas las instancias en un espacio. Cuando se predice devuelve la media de los valores de los k vecinos más cercanos del nuevo punto.

Sin embargo, sin aplicar ninguna transformación la dimensionalidad del conjunto de datos es demasiado alta y se tarda demasiado tiempo en obtener unos resultados aceptables. Dado que hay un número reducido de instancias hay demasiados atributos. Muchos de ellos son redundantes al referirse a la misma casilla o por tratar la influencia de casillas adyacentes. Para hacer un análisis efectivo es necesario reducir las dimensiones del problema de aprendizaje. Se consideran las siguientes técnicas para reducir la dimensionalidad del conjunto de atributos:

- Remove Useless[10]: Esta transformación retira atributos que presentan una variabilidad baja o nula.
- Principal Component Analysis (PCA)[11]: Esta transformación recoge un conjunto de atributos y, a partir de los valores de estos, los transforma en un conjunto de atributos linealmente independientes intentando explicar la variabilidad.
- Selección de Atributos[12]: Esta transformación elige los atributos más importantes en relación con la clase a partir de un evaluador y un algoritmo de búsqueda.
- Proyecciones Aleatorias[13]: Esta transformación proyecta el conjunto de atributos sobre un espacio de dimensión menor usando una matriz aleatoria con columnas de longitud unidad.

Como existen muchos atributos repetidos, el primer paso para hacer un mejor análisis es retirar atributos inútiles que se puede hacer mediante el filtro *removeUseless*. Con esto se vuelven a repetir las distintas pruebas con Perceptrón multicapa, Regresión lineal, M5P, M5Rules e IBk. El Perceptrón multicapa sigue presentando en este punto tiempos muy altos, a diferencia de Regresión lineal, M5P, M5Rules e IBk, que sus tiempos empiezan a ser manejables.

En este momento se retira el Perceptrón multicapa de los posibles algoritmos y se mantienen el resto de algoritmos. Para mejorar los tiempos se transforma el conjunto de datos mediante distintas técnicas: selección de atributos, PCA y *Proyecciones aleatorias.* Una vez las transformaciones están hechas se repiten las pruebas con Regresión lineal, M5P, M5Rules e IBk.

Finalmente los tiempos son cortos y se puede analizar mediante estos algoritmos, por lo que en este punto acaba la aproximación y es posible avanzar al primer experimento.

En resumen, de la primera aproximación experimental se han concluido los siguientes puntos:

- No es viable aplicar el análisis al fichero sin aplicar ningún tipo de transformación, el tiempo de análisis es demasiado alto para los algoritmos de análisis más importantes.
- No es viable aplicar una red de neuronas, el tiempo de análisis es demasiado alto incluso tras transformar el conjunto de datos.
- Transformar el conjunto de datos es una opción viable, se tarda menos de 2 minutos para las transformaciones más importantes.
- El atributo "movimiento" no se puede aplicar con regresión debido a que es una cadena de caracteres.

Para el resto de la experimentación se concluye experimentar con:

- Regresión lineal
- Árboles de regresión (M5P)

- Reglas (M5Rules)
- Aprendizaje vago o basado en instancias (IBk)
- Transformaciones
    - PCA
    - Selección de atributos
    - Proyecciones aleatorias

6.3. Primer experimento

Para este experimento se aplicarán los algoritmos:

- Regresión lineal
- M5P
- IBk
- RulesM5P

Se utilizarán las siguientes transformaciones al conjunto

- PCA
- Selección de atributos

El tablero utilizado es el mismo que el aplicado en la primera aproximación, es el tablero de referencia.

Una vez se genera el conjunto inicial con *Fuego*, se crean dos transformaciones distintas del conjunto original a partir de *Weka*, el conjunto PCA y el conjunto selección de atributos. Ambos comparten 3 transformaciones y se diferencian en una transformación. Las transformaciones comunes son: Normalizar, Retirar inútiles y Retirar "Movimiento". Normalizar lleva todos los datos al rango entre 0 y 1. Retirar inútiles quita los atributos irrelevantes para la clase. Retirar "Movimiento", elimina el atributo para el análisis por regresión.

Se diferencian en que la última transformación aplicada es la que define al conjunto. La última transformación del conjunto PCA es la transformación PCA y, sobre el conjunto de selección de atributos, se aplica selección de atributos.

Finalmente se obtienen 3 conjuntos con las siguientes características:

	Atributos	Instancias
Original	3256	941
RemoveUseless	1230	
PCA	382	
Selección de atributos	39	

Tabla 2: Propiedades de los conjuntos del primer experimento

En esta tabla se ve como el conjunto inicial de datos tiene 3256 atributos para 941 instancias. La dimensionalidad es demasiado alta, por ello se reduce a PCA y selección

de atributos que tienen 382 y 39 atributos respectivamente, que son dimensiones más adecuadas.

De los conjuntos disponibles se han cogido PCA y Selección de atributos. Utilizando el *experimenter* de *Weka* se introducen ambos conjuntos y los algoritmos deseados. El último paso es poner en funcionamiento el *experimenter* con la información introducida. Para hacer la validación, se utiliza validación cruzada. Los resultados obtenidos de los conjuntos son los siguientes coeficientes de correlación:

	Regresión	M5P	IBk	RulesM5P
PCA	0.43	0.36	0.06	0.36
Selección de atributos	0.85	0.86	0.77	0.80

Tabla 3: Resultados primer experimento

En esta tabla se ven los resultados de PCA y Selección de atributos. Selección de atributos muestra coeficientes de correlación mucho más altos que PCA. Dado que hay pocas instancias es posible que se haya cometido sobreaprendizaje al utilizar el conjunto de PCA, por tanto se valora usar más instancias en la muestra inicial.

Todos los resultados serán analizados en conjunto al final de la experimentación en la fase de "resultados".

Además de lo descrito y los resultados propuestos se muestra un ejemplo del modelo generado con el mejor modelo. Como en este caso es M5P se muestra un árbol de regresión y un nodo de ejemplo de dicho árbol. En algunos casos el árbol es demasiado grande y solo se proporciona un extracto. En este primer experimento se proporcionan los dos primeros niveles del árbol.

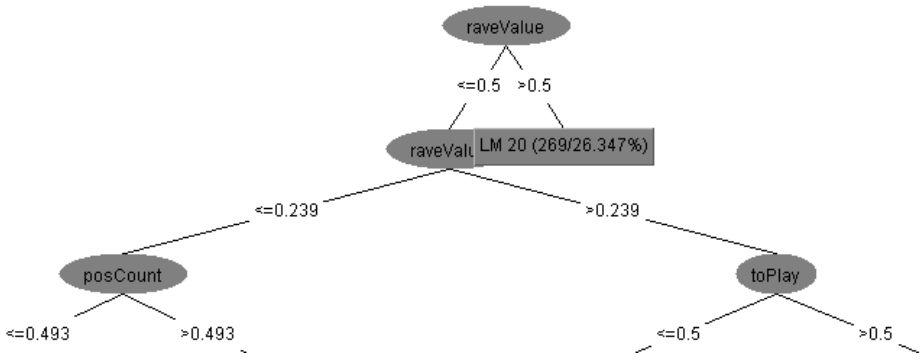


Ilustración 34: Árbol de regresión M5P del primer experimento.

Uno de los nodos que contiene el árbol es:

```
LM 18 =>0.014 * toPlay
      - 0.0291 * infDiff4z15
      - 0.2816 * infDiff9z18
      - 0.07 * infDiff12z4
```

```
+ 0.1112 * infDiff12z9
- 0.09 * infDiff12z17
- 0.0701 * infDiff14z13
- 0.0024 * board7z13
- 0.0069 * board17z7
- 0.0007 * lib6z9
- 0.0009 * lib11z9
+ 0.1576 * lib17z6
- 0.2452 * posCount
- 0.1607 * raveValue
+ 0.8507
```

En el árbol se muestra que *raveValue* está en el nodo raíz, apuntando a que sea un atributo relacionado con la clase. Después se ven *posCount* y *toPlay* en los dos primeros niveles, por lo que mantienen importancia en este árbol. Todos estos valores son globales del tablero por lo que es normal que estén correlados con la clase.

En el nodo se ve la importancia de ciertas casillas en relación con la clase. Por ejemplo las posiciones (4,15), (2,9) y (17,6) con coeficientes superiores a 0.1. No todas las casillas tienen la misma relevancia, de hecho hay algunas casillas que no aparecen en el nodo hoja.

Por último se incluye un mapa de influencia del tablero siguiendo la siguiente notación.

- O: Fuertemente dominado por blancas
- o: Moderadamente dominado por blancas
- .: En disputa
- x: Moderadamente dominado por negras
- X: Fuertemente dominado por negras

```
19|   o   o   .   .   .   x   x   x   x   x   x   x   x   x   x   x   x   x   x
18|   o   o   o   .   .   .   x   x   x   x   x   x   x   x   x   x   x   x   x
17|   o   o   o   o   .   x   x   x   X   X   X   X   X   x   x   x   x   x   x
16|   o   o   o   O   o   x   X   x   X   X   X   X   X   X   x   x   x   x   x
15|   o   o   o   O   o   x   X   x   X   X   X   X   X   X   X   x   x   x   x
14|   o   o   o   o   o   .   x   x   X   X   X   X   X   X   X   X   X   x   x
13|   o   o   o   o   o   .   .   x   x   X   X   X   X   X   X   X   X   X   X
12|   o   o   o   o   o   .   .   .   x   x   x   x   X   X   X   X   X   X   X
11|   o   o   o   o   o   o   .   .   .   .   .   x   x   X   X   X   X   X   X
10|   o   o   o   O   o   o   .   .   o   .   .   x   x   X   X   X   X   X   X
 9|   o   o   o   o   o   o   o   .   .   .   .   .   x   X   X   X   X   X   X
 8|   o   o   o   o   o   o   o   o   .   .   .   .   .   x   x   X   X   X   x   x
 7|   o   o   o   o   o   o   o   o   o   .   .   .   .   .   .   x   x   x   x   x
 6|   o   o   o   o   o   o   o   o   o   o   o   .   .   .   .   .   .   .   .   .
 5|   .   o   o   o   o   o   o   o   o   o   o   o   .   .   .   o   o   .   .   .
 4|   o   o   o   O   o   o   o   o   o   o   o   o   .   o   o   o   o   .   .
 3|   o   o   o   o   o   o   o   o   o   o   o   o   o   o   o   o   o   o   .
 2|   o   o   o   o   o   o   .   o   o   o   o   o   o   o   o   o   o   o   .
 1|   o   o   o   o   o   o   o   o   o   o   o   o   o   o   o   o   o   o   o
    [A] [B] [C] [D] [E] [F] [G] [H] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T]
```

Ilustración 35: Representación gráfica de la influencia del tablero referencia

Este mapa de influencia se calcula mediante el método de análisis descrito en el proyecto. Resumidamente, se genera una búsqueda en árbol Monte-Carlo, en sus nodos hoja se calcula el mapa de influencia para la posición final y se propaga hacia atrás por todo el árbol hasta llegar al nodo raíz. El mapa de influencia del nodo raíz es el mapa de influencia de la posición analizada.

El mapa generado muestra cómo, a pesar de que el tablero esté prácticamente vacío, el programa muestra cómo las zonas están ya decididas. El jugador negro controlará casi siempre la parte superior izquierda y el jugador blanco controlará la zona inferior derecha. El control del jugador negro además es mucho más fuerte que el del jugador blanco. Esto principalmente se debe a que el jugador blanco tiene más extensión del tablero dominada, pero sin estructuras, pero el negro tiene un ojo en su posición, que le otorga mucha ventaja.

6.4. Batería de experimentos

Para la batería de experimentos se aplicarán los algoritmos:

- Regresión lineal
- M5P
- IBk
- RulesM5P

Se utilizarán las siguientes transformaciones al conjunto

- PCA
- Selección de atributos
- Proyecciones aleatorias

Todas las pruebas comparten las mismas propiedades. Se utilizarán los algoritmos y transformaciones descritas. Cada tablero ha sido extraído de un libro de problemas[3] y el número del tablero viene indicado en el pie de la ilustración de dicho tablero.

6.4.1. Experimento 1

Se utilizan los algoritmos y transformaciones descritos en la batería de experimentos. El mapa utilizado es el que se muestra a continuación:

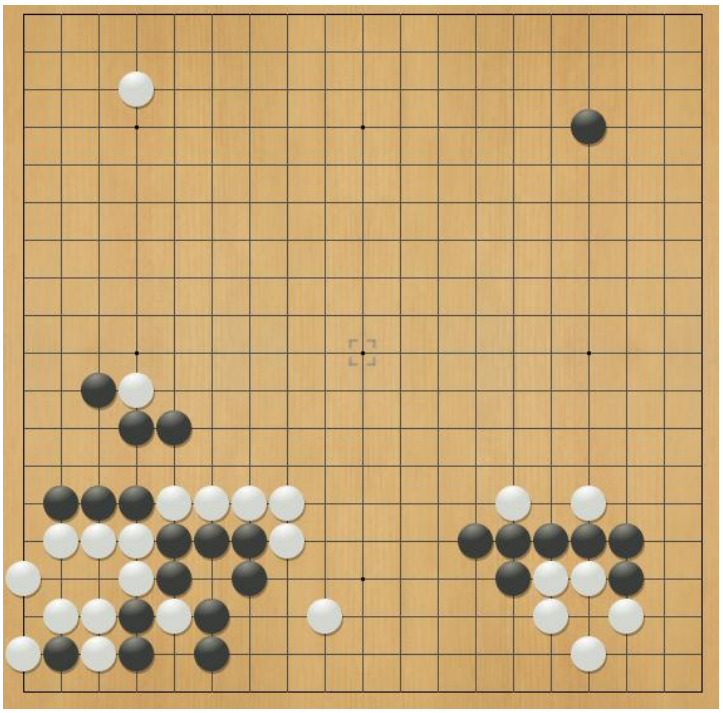


Ilustración 36: Tablero del experimento 1 nº 222

Estas son las propiedades del tablero del experimento 1:

	Atributos	Instancias
Original	3256	1676
RemoveUseless	1190	
PCA	376	
Selección de atributos	30	
Proyecciones aleatorias	11	

Tabla 4: Propiedades de los conjuntos del experimento 1

Del conjunto original de 3256 atributos se reducen dos órdenes de magnitud con selección de atributos y un orden de magnitud con PCA. De esta manera las dimensiones son adecuadas para el análisis.

Estos son los coeficientes de correlación del experimento 1:

	Regresión	M5P	IBk	RulesM5P
PCA	0.51	0.56	0.12	0.51
Selección de atributos	0.92	0.93	0.85	0.93
Proyecciones aleatorias	0.10	0.23	0.11	0.21

Tabla 5: Resultados experimento 1

Se han mejorado los resultados de PCA y Selección de atributos en una décima aproximadamente. Los resultados de PCA siguen siendo inferiores a selección de

atributos. Proyecciones aleatorias proporciona resultados inferiores a PCA y selección de atributos. Recoger muchas más instancias ha mejorado ligeramente los resultados de PCA por lo que el problema del sobreaprendizaje no es tan acentuado. Proyecciones aleatorias funciona muy negativamente, esto es probable que se deba a que extrae muy pocos atributos en comparación a los atributos.

Un ejemplo del árbol generado a partir de M5P y el conjunto de selección de atributos es el siguiente:

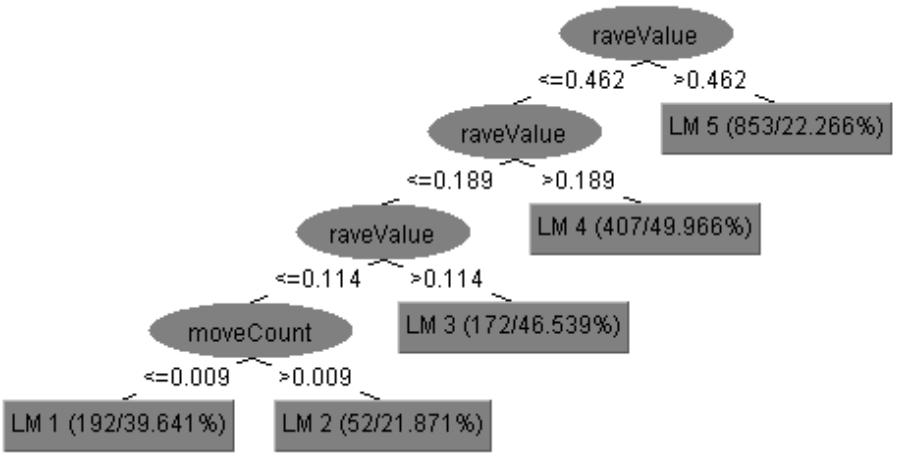


Ilustración 37: Árbol de regresión M5P experimento 1

Uno de los nodos del árbol contiene la siguiente ecuación de regresión:

```
LM 4 => 0.0455 * infDiff12z1
      + 0.0018 * infDiff15z16
      + 0.0002 * board3z16
      + 0.0033 * board12z15
      - 0.0572 * lib2z5
      + 0.0301 * lib3z1
      - 0.0004 * lib5z12
      - 0.0005 * lib9z2
      + 0.0112 * moveCount
      - 0.2669 * raveValue
      + 0.434
```

En el árbol aparecen de nuevo *raveValue* y *moveCount*. En comparación al primer experimento el árbol es mucho menor, con pocos niveles y con menor número de modelos lineales. Esta vez no aparece *toPlay*, es posible que su importancia sea solo circunstancial.

En este nodo hay tres posiciones con importancia superior al resto. Estas son las posiciones (12,1), (2,5) y (3,1). Todas tienen coeficientes superiores a 0.03. Esta vez estos coeficientes son menores que en el primer experimento.

El mapa de influencia del tablero es:

```
19|   o   o   o   o   o   o   .   .   .   .   .   .   .   .   .   x   X   X   X
18|   o   o   o   o   o   o   .   .   .   .   .   .   .   .   x   x   x   x   x
17|   o   o   o   o   o   o   .   .   .   .   .   .   .   x   x   x   x   x   x
16|   .   .   o   o   o   .   .   .   .   .   .   .   .   x   x   x   x   x   x
15|   .   .   .   .   .   .   .   .   .   .   .   .   .   x   x   x   x   x   x
14|   .   .   .   .   .   .   .   .   .   .   .   .   x   x   x   x   x   x   x
13|   .   .   .   .   .   .   .   .   .   .   .   .   .   x   x   x   x   x   x
12|   x   x   .   .   .   .   .   .   .   .   .   .   .   x   x   x   x   x   x
11|   x   x   x   .   .   .   .   .   .   .   .   x   .   .   x   x   x   x   x
10|   x   x   x   x   .   x   .   .   .   .   x   x   x   .   x   .   x   x   x
 9|   X   X   X   X   x   x   x   .   .   .   x   x   x   x   x   x   x   x   x
 8|   X   X   X   X   x   x   .   .   .   .   x   x   x   x   x   x   x   x   x
 7|   X   X   X   X   x   o   o   o   .   .   x   x   x   x   x   x   x   x   x
 6|   x   x   X   x   x   .   o   o   o   .   .   x   x   x   x   X   X   X   X
 5|   O   O   O   o   .   X   .   o   o   .   .   x   X   X   x   X   X   X   x
 4|   O   O   O   O   X   X   X   .   o   .   .   x   X   X   x   .   x   x   x
 3|   O   O   O   x   X   X   X   .   o   o   .   .   x   .   O   O   o   .   .
 2|   O   O   O   x   X   X   X   .   .   .   .   .   .   o   O   O   O   o   o
 1|   O   O   O   x   X   X   X   .   .   .   .   .   o   o   o   O   O   o   o
    [A] [B] [C] [D] [E] [F] [G] [H] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T]
```

Ilustración 38: Representación gráfica de la influencia del tablero del experimento 1

Este mapa de influencia es más complejo que el anterior. Para empezar se ve que en el momento actual el jugador negro tiene mucho más tablero bajo su control y el blanco está en un mal lugar en comparación. Sin embargo, el blanco puede aprovechar que la zona del medio está sin decidir para cambiar el rumbo de la partida a su favor. Además, el blanco tiene pequeñas formaciones controladas en los bordes del tablero, las cuales son muy seguras.

6.4.2. Experimento 2

Se utilizan los algoritmos y transformaciones descritos en la batería de experimentos. El mapa utilizado es el que se muestra a continuación:

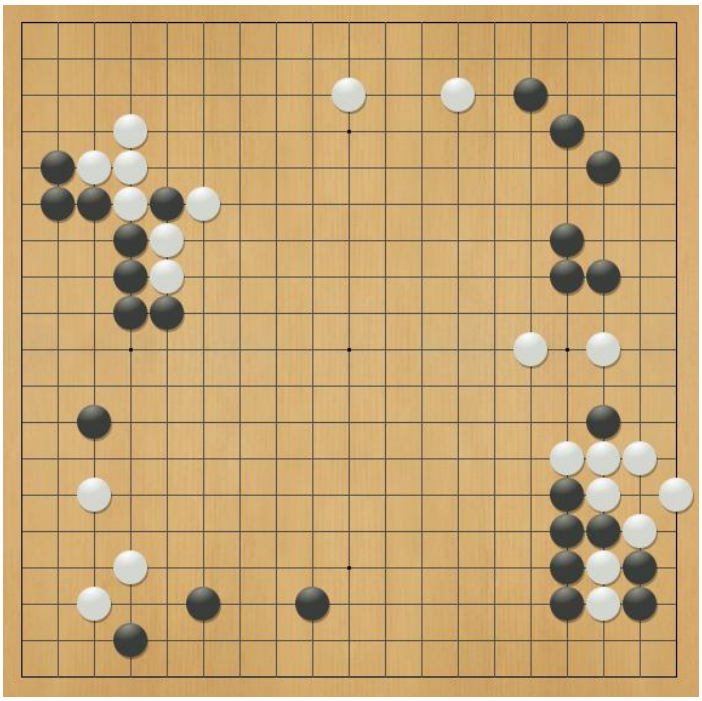


Ilustración 39: Tablero del experimento 2 nº 384

Estas son las propiedades del tablero del experimento 2:

	Atributos	Instancias
Original	3256	1614
RemoveUseless	1263	
PCA	376	
Selección de atributos	27	
Proyecciones aleatorias	11	

Tabla 6: Propiedades de los conjuntos del experimento 2

Del conjunto original de 3256 atributos se reducen dos órdenes de magnitud con selección de atributos y un orden de magnitud con PCA. Siempre se ofrecen cifras de atributos similares, PCA oscila siempre entre los 400-300 atributos, selección de atributos entre 25-35 atributos y proyecciones aleatorias siempre tiene 11.

Estos son los coeficientes de correlación del experimento 2:

	Regresión	M5P	IBk	RulesM5P
PCA	0.61	0.58	-0.01	0.51
Selección de atributos	0.92	0.93	0.86	0.93
Proyecciones aleatorias	0.18	0.24	0.08	0.23

Tabla 7: Resultados experimento 2

PCA y proyecciones aleatorias ofrecen mejores resultados que en experimentos anteriores, pero no se acercan a los resultados de selección de atributos. Hay una anomalía en los resultados por parte de IBk que da -0.01 de coeficiente de correlación.

Un ejemplo del árbol generado a partir de M5P y el conjunto de selección de atributos es el siguiente:

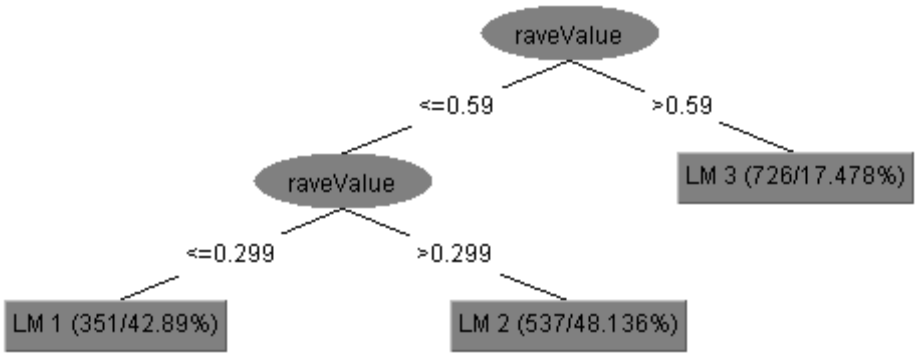


Ilustración 40: Árbol de regresión M5P experimento 2

Uno de los nodos del árbol contiene la siguiente ecuación de regresión:

```
LM 1 =>-0.0031 * infDiff4z18
       + 0.0016 * infDiff5z5
       - 0.0025 * infDiff16z1
       + 0.0578 * infDiff19z15
       - 0.0004 * lib2z19
       - 0.1249 * lib9z2
       - 0.0359 * lib12z6
       + 0.165 * moveCount
       - 0.2025 * raveValue
       + 0.5211
```

El árbol es muy reducido y muestra *raveValue* en todos los nodos de decisión. Sólo hay tres modelos lineales. En el nodo se puede ver como la posición (9,2) mantiene la mayor importancia. De nuevo es un coeficiente superior a 0.1.

El mapa de influencia del tablero es:

```
19|   .   .   .   o   o   o   o   o   o   o   o   o   o   o   .   x   x   x   x   x
18|   .   .   o   o   o   o   o   o   O   O   o   o   o   .   x   x   x   x   x   x
17|   .   .   o   o   o   o   o   o   O   O   o   o   o   .   X   X   X   x   x   x
16|   x   .   o   O   O   O   o   o   o   O   o   o   o   .   X   X   X   X   X   X
15|   X   X   o   O   O   O   o   o   o   o   o   o   .   .   x   X   X   X   X   X
14|   X   X   X   O   O   O   O   o   o   o   o   .   .   x   x   X   X   X   X   x
13|   X   X   X   X   O   O   o   o   o   o   .   .   .   .   X   X   X   X   x   x
12|   X   X   X   X   x   .   .   .   .   .   .   .   .   .   x   X   X   X   x   x
11|   X   X   X   X   X   x   .   .   .   .   .   .   .   o   .   .   .   .   .   .
10|   x   X   X   X   X   x   x   .   .   .   o   o   o   o   O   O   O   o   o   o
 9|   x   x   X   x   x   x   x   x   .   .   .   o   o   o   O   O   o   o   o   o
 8|   x   x   x   x   x   x   x   .   .   .   .   .   o   o   o   o   O   O   O   O
 7|   .   .   .   .   .   x   .   .   .   .   .   .   .   .   o   O   O   O   O   O
 6|   o   o   o   o   .   .   .   .   .   .   .   .   .   .   x   .   O   O   O   O
 5|   o   o   o   o   .   .   x   x   x   x   .   .   x   x   X   X   X   o   O   O
 4|   o   o   o   o   .   x   x   x   x   x   x   x   x   x   X   X   X   X   X   X
 3|   o   o   o   o   x   x   X   X   X   x   x   x   x   x   X   X   X   X   X   X
 2|   o   o   .   x   x   x   x   x   X   x   x   x   x   x   X   X   X   X   X   X
 1|   o   o   .   x   x   x   x   x   x   x   x   x   x   x   X   X   X   X   X   X
     [A] [B] [C] [D] [E] [F] [G] [H] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T]
```

Ilustración 41: Representación gráfica de la influencia del tablero del experimento 2

El juego en este punto de la partida está bastante igualado, con cierta ventaja para el jugador negro. Hay muy poco territorio en disputa. La mayor parte de los bordes ya están dominados por cualquiera de los dos jugadores. El jugador blanco puede unir dos de sus zonas para conseguir mayor dominio de la partida. El jugador negro puede hacer la misma jugada uniendo dos de sus zonas.

6.4.3. Experimento 3

Se utilizan los algoritmos y transformaciones descritos en la batería de experimentos. El mapa utilizado es el que se muestra a continuación:

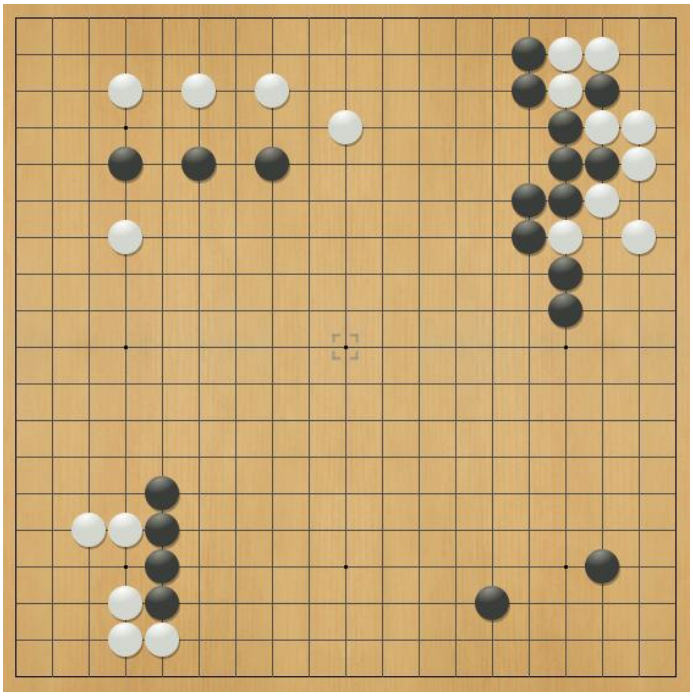


Ilustración 42: Tablero del experimento 3 nº 175

Estas son las propiedades del tablero del experimento 3:

	Atributos	Instancias
Original	3256	1741
PCA	440	
Selección de atributos	32	
Proyecciones aleatorias	11	

Tabla 8: Propiedades de los conjuntos del experimento 3

Este último experimento reafirma que los atributos siempre son similares, el que más varía es PCA por su orden de magnitud superior.

Estos son los coeficientes de correlación del experimento 3:

	Regresión	M5P	IBk	RulesM5P
PCA	0.59	0.53	0.10	0.57
Selección de atributos	0.91	0.92	0.83	0.92
Proyecciones aleatorias	0.14	0.24	0.16	0.24

Tabla 9: Resultados experimento 3

Definitivamente selección de atributos siempre será superior al resto de transformaciones para nuestro proceso de aprendizaje. PCA y proyecciones aleatorias son peores en comparación a selección de atributos en todos los casos.

Un extracto de los dos primeros niveles del árbol generado a partir de M5P y el conjunto de selección de atributos es el siguiente:

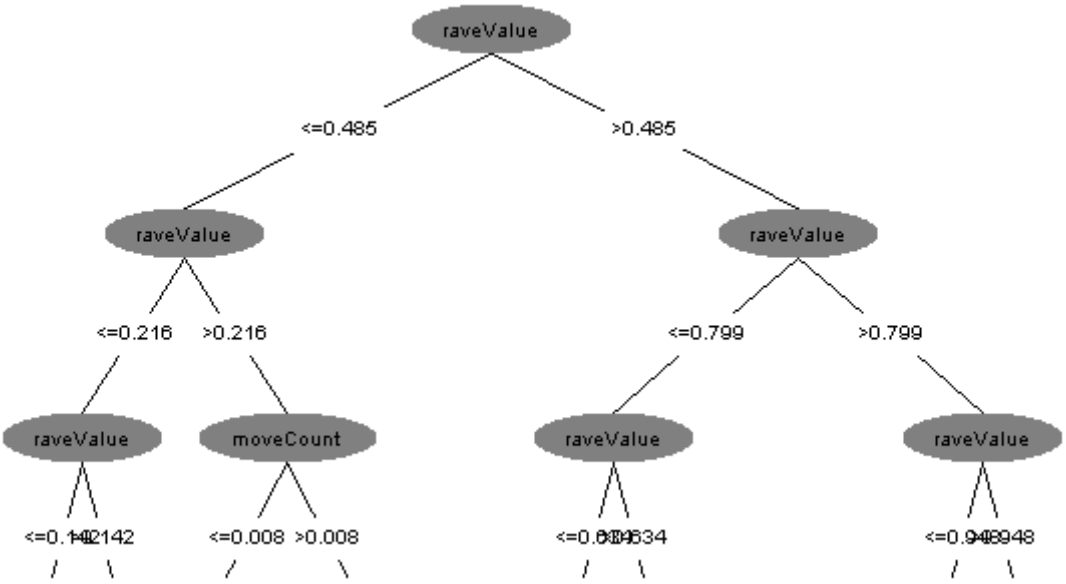


Ilustración 43: Árbol de regresión M5P experimento 3

Uno de los nodos del árbol contiene la siguiente ecuación de regresión:

```
LM 7 => 0.0046 * inf7z7
      - 0.0121 * infDiff11z14
      + 0.0924 * infDiff14z9
      - 0.2243 * infDiff17z16
      - 0.0592 * board3z7
      + 0.024 * board16z11
      + 0.0062 * lib9z6
      + 0.0045 * lib18z15
      + 0.0178 * moveCount
      - 0.2331 * raveValue
      + 0.539
```

En el árbol los valores *raveValue* y *moveCount* son los atributos dominantes en el árbol. El atributo *raveValue* es el más frecuente. En el nodo mostrado hay un coeficiente del tablero superior a 0.2. Esto implica que es una posición muy importante en el tablero.

El mapa de influencia del tablero es:

```
19|   o   o   o   o   O   O   O   O   O   .   .   .   x   x   o   O   O   O   O
18|   o   o   o   O   O   O   O   O   o   .   .   .   x   X   x   O   O   O   O
17|   o   o   o   O   O   O   O   O   o   o   .   .   x   X   X   o   O   O   O
16|   .   .   .   .   .   .   .   .   o   o   .   .   x   X   X   X   O   O   O
15|   .   .   .   x   x   x   x   x   .   .   .   x   x   X   X   X   O   O   O
14|   .   .   .   .   .   x   x   x   x   .   .   x   x   X   X   X   .   O   O
13|   .   .   o   o   .   .   .   .   .   .   .   x   x   X   X   X   x   O   O
12|   .   .   o   o   o   .   .   .   .   .   .   x   x   X   X   X   X   .   .
11|   .   .   .   .   .   .   .   .   .   .   .   x   x   x   X   X   X   x   x
10|   o   o   .   .   .   .   .   .   .   .   .   x   x   x   X   X   X   x   x
 9|   o   o   .   .   .   .   .   .   .   .   x   x   x   x   x   x   x   x   x
 8|   o   o   .   .   .   .   x   x   .   .   x   x   x   x   x   x   x   x   x
 7|   o   o   o   .   x   x   x   x   x   .   x   x   x   x   x   x   x   x   x
 6|   o   o   O   .   x   x   x   x   x   .   .   x   x   x   x   x   x   x   x
 5|   O   O   O   o   X   X   x   x   x   x   x   x   x   x   x   X   X   x   x
 4|   O   O   O   O   x   X   x   x   x   x   x   x   x   X   X   X   X   x   x
 3|   O   O   O   O   .   x   x   x   x   x   x   x   X   X   X   X   X   x   x
 2|   O   O   O   O   O   o   .   .   x   x   x   x   X   X   X   X   x   x   x
 1|   O   O   O   O   O   o   .   .   .   x   x   x   x   X   X   x   x   x   x
     [A] [B] [C] [D] [E] [F] [G] [H] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T]
```

Ilustración 44: Representación gráfica de la influencia del tablero del experimento 3

En este tablero hay poca decisión por parte del jugador blanco. El jugador negro tiene el mapa dominado en su gran mayoría con puntos bastante fuertes y hay muy poca extensión del tablero en disputa. El jugador blanco tiene oportunidad de neutralizar parte de la influencia negra que está aislada, aunque de poco sirva para el resultado final de la partida.

6.5. Resultados

En primer lugar y la parte más importante son los resultados obtenidos mediante selección de atributos. Consistentemente en todos los algoritmos selección de atributos presenta los coeficientes de correlación más altos, siempre superiores a 0.8. En el conjunto de datos existen muchos atributos que no se utilizan para nada o que no presentan ninguna relación con la clase. Sin embargo los datos que se retiran son distintos para cada tablero. Por tanto la información es irrelevante de manera circunstancial.

Luego, en cada experimento, el algoritmo IBk presenta siempre los peores resultados. Con selección de atributos los resultados de IBk son los más bajos aunque dentro del estándar que se ha indicado previamente.

Tanto las transformaciones de PCA y Proyecciones aleatorias no se acercan a la precisión ofrecida por Selección de atributos. En concreto PCA funciona mejor que Proyecciones aleatorias. En todos los casos los resultados son similares, manteniéndose PCA y Proyecciones aleatorias en los mismos rangos de resultados, por lo que se pueden descartar como transformaciones viables.

Como se han obtenido coeficientes de correlación altos de los árboles de regresión, se pueden analizar en detalle. En primer lugar en todos los árboles *raveValue* siempre toma valores altos en los modelos de regresión lineal de los nodos hoja y toma posiciones

importantes en el árbol en los primeros niveles. Esto implica que el valor RAVE es un indicador muy potente para predecir la clase.

Luego en los modelos lineales se muestran una serie de atributos junto a sus coeficientes. En estos modelos se puede observar como hay dos valores que se repiten siempre: *moveCount* y *raveValue*. Como ya se ha comentado raveValue tiene gran importancia sobre la clase, y *moveCount* comparte esa importancia. Pero, sin duda alguna la parte más interesante son las distintas posiciones del tablero que aparecen en los modelos de regresión de los nodos hoja.

Se puede comprobar las posiciones importantes en esta partida. Se utiliza como ejemplo para el nodo de ejemplo del experimento 3. De este nodo se puede ver información importante sobre el transcurso de la partida. Teniendo en cuenta que cuando un jugador va ganando la clase se aproxima a 1 si es el blanco o a 0 si el jugador es el negro:

```
LM 7 => 0.0046 * inf7z7          - 0.0121 * infDiff11z14
      + 0.0924 * infDiff14z9     - 0.2243 * infDiff17z16
      - 0.0592 * board3z7        + 0.024 * board16z11
      + 0.0062 * lib9z6          + 0.0045 * lib18z15
      + 0.0178 * moveCount       - 0.2331 * raveValue
      + 0.539
```

- La posición (7,7) tiene importancia estratégica para el jugador blanco.
- Al jugador negro le interesa el control de la posición (17,16)
- Al jugador blanco le interesa el con control de la posición (4,9)
- El jugador blanco se beneficia de mantener libres las posiciones (9,6) y (18,15)

Este ejemplo de análisis se puede realizar para cada nodo hoja de cualquier árbol de decisión realizado.

Por tanto, tras los experimentos realizados, se concluye que:

- Es posible predecir quién va ganando en la partida a partir de los distintos mapas e información estadística al 90% aproximadamente.
- El algoritmo que mejor se comporta ante este conjunto de datos es M5P para un conjunto de datos filtrado con selección de atributos.
- Los atributos *raveValue* y *moveCount* tienen especial peso en el árbol de decisión.
- A partir de los modelos de regresión de los nodos hoja se puede realizar un análisis de posiciones importantes y de decisiones estratégicas.
- Las transformaciones realizadas por PCA y *Proyecciones aleatorias* ofrecen resultados muy por debajo de los deseados, además el algoritmo IBk ofrece los peores resultados del conjunto de algoritmos.

7. Gestión de proyecto

En este punto se examina la planificación, presupuestos y regulación que rodea el proyecto. Para cada uno de los aspectos de la gestión se destina un apartado propio donde se expandirá lo que ha sido desarrollado para dicho aspecto de la gestión.

7.1. Planificación

El proyecto consta de varias partes diferenciadas, pero en general se pueden diferenciar dos partes separadas y paralelizables: La realización y la documentación. La realización es la parte conceptual del proyecto, donde se examinan alternativas y se toman decisiones de diseño, la documentación es donde se plasma la parte de la realización sobre el papel. La división entre realización y documentación se ha trazado para independizar partes para paralelizar trabajo.

Ambas partes tienen a su vez 4 fases:

- Análisis: Fase donde se aproxima el problema propuesto, examinando su viabilidad, requisitos y estado del arte.
- Diseño: Fase donde se conceptualiza una solución para crear la herramienta, examinando alternativas y buscando la solución más apropiada.
- Desarrollo: Fase donde se aplica lo aprendido en el análisis y diseño para generar la herramienta deseada.
- Experimentación: Fase en la que, con la herramienta perfectamente funciona, se hace una serie de pruebas para tratar de resolver y dar una solución al problema propuesto.

En realidad la documentación no es más que una extensión y de cualquiera de estas fases. Las tareas de documentación son la parte final de cada fase, donde se cierra y esta parte concreta es paralela con la siguiente parte del proyecto.

Cada fase tiene su faceta de realización, donde se trabaja sobre ese aspecto del desarrollo. La representación del flujo de trabajo es la siguiente:

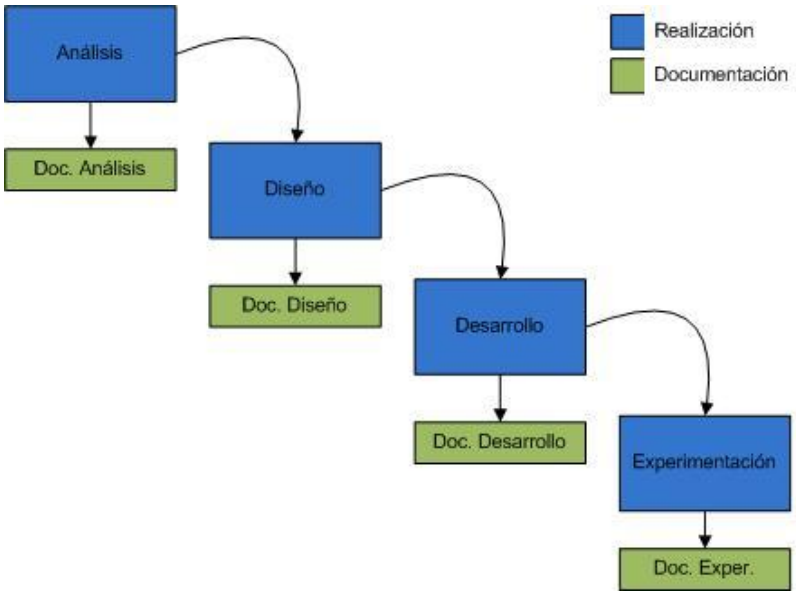


Ilustración 45: Proceso del proyecto

ID	Tarea	Comienzo	Final	Duración
1	Análisis	02/12/2013	31/12/2013	22d
2	Diseño	01/01/2014	30/01/2014	22d
3	Desarrollo	03/02/2014	30/05/2014	85d
4	Experimentación	02/06/2014	30/06/2014	21d
5	Documentación Análisis	16/01/2014	30/01/2014	11d
6	Documentación Diseño	03/02/2014	17/02/2014	11d
7	Documentación Desarrollo	02/06/2014	15/07/2014	32d
8	Documentación Experimentación	17/07/2014	15/08/2014	22d

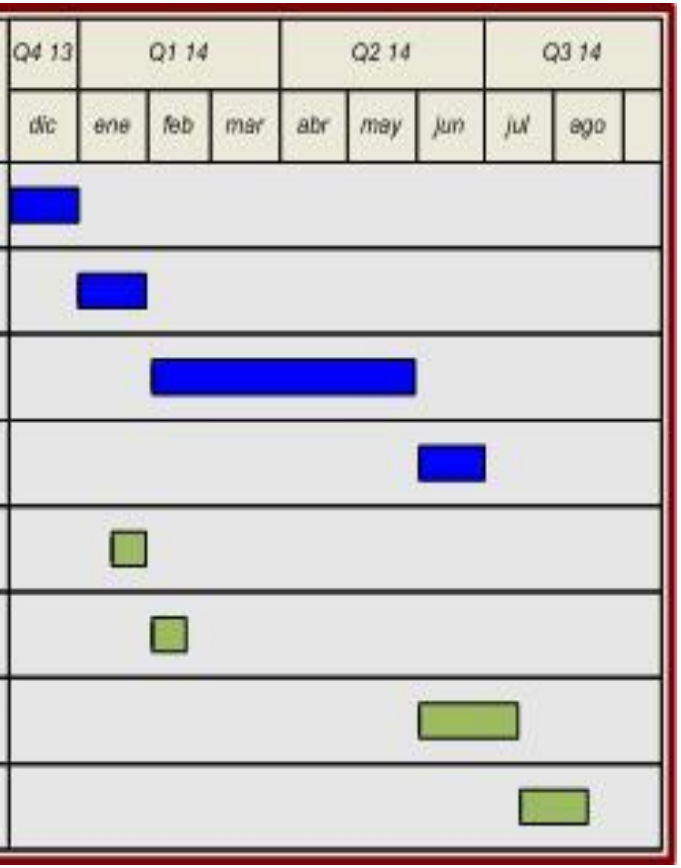


Tabla 10: Resolución de la planificación

En total para cada fase, según el diagrama de Gannt realizado, se dedicarán los siguientes tiempos al proyecto:

- Por Categoría:
  - Realización completa: 150 días / 66,3% del proyecto
  - Documentación completa: 76 días / 33,6% del proyecto
- Por Fase:
  - Análisis: 33 días / 14,6% del proyecto
  - Diseño: 33 días / 14,6% del proyecto
  - Desarrollo: 117 días / 51,8% del proyecto
  - Experimentación: 43 días / 19% del proyecto
- Duración estimada del proyecto (Sin paralelizar): 226 días
- Duración estimada del proyecto (Paralelizado): 184 días

Estos cálculos tienen en cuenta la información del diagrama Gannt, el cual omite días no lectivos. Este cálculo representa la cantidad de días dedicados al proyecto.

7.2. Presupuesto

En este presupuesto se calculará una estimación del coste estimado del proyecto.

El coste del proyecto se estimará en función del material requerido, ya sea hardware o software; y en función del personal del proyecto, calculado a partir de la planificación y las horas estimadas.

En primer lugar, el proyecto se estima que dure 184 días, pero la duración en horas es de entre 300 y 360 horas, por tanto se estiman 2 horas de trabajo al día. Esta cifra puede variar, pero es una media del cálculo de tiempo.

A continuación se detalla el presupuesto destinado a cada elemento del material.

Categoría	Ítem	Ud.	Precio/Ud.	Total
Hardware	Ordenador	x1	500€	500€
	Impresora	x1	40€	40€
Material fungible	Hojas Din A4, pack 500	x1	4€	4€
	Tinta negra impresión	x2	15€	30€
Total				**574€**

Tabla 11: Presupuesto del material

A continuación se detallan los costes indirectos.

Ítem	Precio/Mes.	Total
Luz	50€	450€
Alquiler	400€	3600€
Transporte	40€	360€
Total		**4410€**

Tabla 12: Presupuesto para costes indirectos

A continuación se detalla el presupuesto destinado al personal.

Función	Nombre	Horas	Precio/Hora	Total
Desarrollador	Javier Huertas	360	15	5400€
Supervisor 1	Álvaro Torralba	20	25	500€
Supervisor 2	Daniel Borrajo	10	25	250€
Total				**6150€**

Tabla 13: Presupuesto del personal

Finalmente, una tabla resumen del presupuesto.

Función	Total
Materiales	574€
Costes indirectos	4410€
Personal	6150€
Total	**11134€**

Tabla 14: Resolución del presupuesto

El coste total del proyecto se calcula que sea de 11134€

7.3. Regulación

Todo el software utilizado se encuentra bajo la licencia GNU GPL[4]. Esto se ha hecho así para ahorrar en costes de software a la vez que apoyar el software público.

Estar bajo la licencia GNU GPL implica:

- Que el contenido está bajo la protección del Copyright
- Que el contenido ofrece permiso legal para copiar, distribuir y modificar.

Todo el trabajo realizado está, por tanto, dentro de los términos de la licencia debido a:

- No son modificaciones de uso comercial.
- Las modificaciones están explícitamente indicadas
- Se respeta la licencia GNU GPL

8. Conclusiones

El Go es un juego tan simple como complejo a la vez. Una gran cantidad de matices ocultos y diferentes estrategias lo convierten en uno de los problemas menos aproximables de la computación. A lo largo del proyecto se ha desarrollado una herramienta para poder aproximar el problema de una manera diferente. Con esto cabía el riesgo de que, en realidad, fuese imposible aplicar la propagación hacia atrás de manera eficaz para conseguir resultados.

Al final y tras la experimentación se han conseguido resultados positivos. En primer lugar la herramienta presenta una visión muy útil de la influencia del tablero, lo cual ayuda a conseguir ver cómo se ha desarrollado la partida hasta ahora y cómo se puede explotar un territorio para maximizar la puntuación. A la hora de analizar un tablero se han llegado a resultados muy buenos. Es posible recoger un mapa de influencia con una precisión aceptable y, aunque existe un gran componente aleatorio, es posible recuperar mapas de influencia consistentes. A través de este análisis se puede, visualmente, distinguir grupos de fichas que viven, mueren, zonas relevantes para un jugador y zonas neutrales en un punto de la partida.

Los resultados de la experimentación también han sido satisfactorios, con coeficientes de correlación altos, lo que indica que los valores de quién tiene ventaja en la partida son predecibles y la influencia tiene relación con ellos. Además gracias a la selección de atributos, se puede ver exactamente qué casillas del tablero son las más relevantes y por qué. A partir de esto se pueden responder a preguntas como ¿Es la casilla B16 importante en este tablero?¿Por su grado de libertad?¿Tiene un ojo?¿Posee mucha influencia del jugador blanco?... Gracias a que se analiza con un árbol los resultados pueden analizarse y explicarse.

Examinando los objetivos obtenemos:

- *Crear una herramienta capaz de extraer información a partir de un tablero de Go.*
  - *Encontrar una herramienta que genere el árbol de Monte-Carlo.*
    - Se ha encontrado la herramienta *Fuego*, que era la más apropiada para el desarrollo. El resto de alternativas no eran válidas o sencillamente Fuego era una opción mucho mejor.
  - *Analizar la herramienta para su posterior modificación.*
    - A lo largo del análisis del problema se ha examinado la herramienta en sus partes más relacionadas con el proyecto.
  - *Adaptar la herramienta para que extraiga la información que se desee.*
    - En el apartado de la descripción del analizador de posiciones se describe todos los cambios realizados para la extracción, en especial, toda la parte relativa a la generación y propagación de influencia.

- *Crear una herramienta capaz de representar información en un formato analizable.*
    - *Organizar los datos*
        - Ambos métodos *SaveNode* cumplen esta función: organizar datos e imprimirlos, tal y como se describe en el apartado de analizador de posiciones.
    - *Crear un formato de salida para la interpretación directa*
        - En este caso no ha sido necesario crear un formato nuevo, dado que existe *sgf*, que es interpretable por programas de visualización de tableros.
    - *Crear un formato de salida para el análisis mediante aprendizaje automático*
        - De nuevo, solo ha sido necesario utilizar el formato *arff*, que es interpretable por *Weka*
- *Experimentar con los datos obtenidos mediante aprendizaje automático.*
    - *Buscar atributos clave de un tablero de Go.*
        - A través de investigar distintas fuentes[5][6] de Go se ha generado un conjunto de atributos clave de un tablero de Go, así como la influencia.
    - *Encontrar información relevante de los experimentos realizados.*
        - Se han descubierto algunos detalles interesantes, en concreto hay dos destacables. El primero es que es posible predecir quién es el jugador con ventaja a partir de las técnicas utilizadas, el segundo es que es posible encontrar posiciones relevantes a partir de la selección de atributos.
- *Conocer más profundamente el Go y su funcionamiento.*
    - Descubrir nuevas técnicas de análisis para el Go.
        - Con este proyecto se pretendía aplicar y unir técnicas antiguas que se descubrieron que no eran viables y la aproximación moderna con árboles de búsqueda Monte-Carlo. A través del uso conjunto del análisis de datos y los árboles de Monte-Carlo se han conseguido resultados interesantes, por lo que puede ser viable aplicar esta técnica a la hora de intentar vencer a un jugador real.

9. Líneas futuras

A partir de aquí el proyecto es perfectamente expandible y hay muchas posibles mejoras que podrían realizarse. En primer lugar y más importante, es la posibilidad de incorporar todo lo aprendido en un jugador automático de Go. Esto supondría un avance importante, porque sería un jugador que conjuga una técnica de búsqueda con una técnica de aprendizaje automático. Por tanto, incorporar la influencia y el análisis de datos a la toma de decisiones de un jugador sería el siguiente paso más lógico de este proyecto y sería su principal motivo de expansión.

Con la herramienta realizada sería posible estudiar tableros de una manera más analítica, para estudiar casos concretos de difícil decisión o problemas de Go. El programa no proporciona posiciones exactas, pero si una buena idea de qué está pasando exactamente en el tablero y quién es el jugador que va ganando en un determinado instante de tiempo.

A raíz del punto anterior se contempla que sería posible utilizar la herramienta y adaptarla para conseguir una herramienta de aprendizaje para nuevos jugadores de Go. Debido a que es un juego fácil de aprender y difícil de dominar sería muy útil crear una herramienta capaz de ayudar a comprender la partida de una manera más profesional. La herramienta nunca se aproximaría a la ayuda que podría ofrecer un jugador experto, pero si podría dar una buena idea del estado de la partida y proporcionar la información relevante al jugador novato para que empiece a comprender por sí mismo el funcionamiento del Go.

Bibliografía y Referencias

[1]Hearn, (Fecha de acceso: 2014, 9 de agosto). *Computational Complexity of Games and Puzzles*. (Fecha de trabajo original: 2006). Disponible en: http://www.ics.uci.edu/~eppstein/cgt/hard.html

[2] Grace I. Lin, (2009). *Fuego Go: The Missing Manual*, http://users.soe.ucsc.edu/~glin/docs/Fuego_Fall09Report.pdf

[3] Bozulich, R. (2002). *501 Opening Problems*. Tokyo, Japón: Kiseido

[4] Anónimo, (Fecha de acceso: 2014, 4 de Septiembre) *GNU General Public Licence* (Fecha del trabajo original: 29 de Junio 2007). Disponible en: http://www.gnu.org/copyleft/gpl.html

[5] Wu, L., Baldi, P. (n.d.). A Scalable Machine Learning Approach to Go, http://papers.nips.cc/paper/3094-a-scalable-machine-learning-approach-to-go.pdf

[6] Silver, D., Sutton, R., Müller, M. (n.d.). *Reinforcement Learning of Local Shape in the Game of Go,* http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Applications_files/local-shape.pdf

[7] Kocsis, L. , Szepesvári, C. (n.d.). *Bandit based Monte-Carlo Planning,* https://web.engr.oregonstate.edu/~afern/classes/cs533/notes/uct.pdf

[8] Brügmann, B. (1993). *Monte Carlo Go,* http://www.ideanest.com/vegos/MonteCarloGo.pdf

[9] M. Enzenberger, M. Müller, B. Arneson and R. Segal. Fuego - An Open-Source Framework for Board Games and Go Engine Based on Monte Carlo Tree Search IEEE Transactions on Computational Intelligence and AI in Games, 2(4), 259-270. Special issue on Monte Carlo Techniques and Computer Go, 2010.

[10] Anónimo, (Fecha de acceso: 2014, 28 de Agosto). *Remove Useless* (n.d). Disponible en: http://wiki.pentaho.com/display/DATAMINING/RemoveUseless

[11] Jolliffe, I. (2005). *Principal component analysis*. John Wiley & Sons, Ltd.

[12] Kohavi, R., & John, G. H. (1997). *Wrappers for feature subset selection*. Artificial intelligence, 97(1), 273-324.

[13] D. Fradkin, D. Madigan. *Experiments with random projections for machine learning*. En: KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, New York, NY, USA, 517-522, 003.

[14] Geoffrey Holmes, Mark Hall, Eibe Frank: *Generating Rule Sets from Model Trees*. En: Twelfth Australian Joint Conference on Artificial Intelligence, 1-12, 1999.

[15] Ross J. Quinlan: *Learning with Continuous Classes*. En: 5th Australian Joint Conference on Artificial Intelligence, Singapore, 343-348, 1992.

[16] Y. Wang, I. H. Witten: *Induction of model trees for predicting continuous classes*. En: Poster papers of the 9th European Conference on Machine Learning, 1997.

[17] D. Aha, D. Kibler (1991). *Instance-based learning algorithms*. Machine Learning. 6:37-66.