Jugadores automáticos basados en árboles de búsqueda de Monte Carlo

Trabajo de Fin de Grado
Curso 2017–2018

Autor
Gabriel David Modia Pozuelo

Directores
Gonzalo Méndez Pozo
Antonio Sánchez Ruiz-Granados

Doble grado en Ingeniería Informática y Matemáticas
Facultad de Informática y Facultad de Matemáticas
Universidad Complutense de Madrid

Jugadores automáticos basados en árboles de búsqueda de Monte Carlo

Trabajo de Fin de Doble Grado Ingeniería Informática y Matemáticas
Departamento de Inteligencia Artificial

Autor
Gabriel David Modia Pozuelo

Directores
Gonzalo Méndez Pozo
Antonio Sánchez Ruiz-Granados

Convocatoria: *Septiembre 2018*

Doble grado en Ingeniería Informática y Matemáticas
Facultad de Informática y Facultad de Matemáticas
Universidad Complutense de Madrid

14 de septiembre de 2018

Dedicatoria

A Mima y Manolo.

Agradecimientos

Este trabajo no habría sido posible sin:

- Mis tutores Antonio y Gonzalo que me han guiado y dirigido, gracias por su disposición y atención.
- Mi madre, mi hermano y mi padre que, cada uno a su manera, han hecho un camino más fácil para mi.
- Alba, que incluso en la más adversa de las situaciones ha estado a mi lado.
- Alicia, que tanto en clase como en casa siempre nos hemos apoyado.
- Jose Antonio, Miguel, Marta y todos mis compañeros con los que he compartido innumerables horas de estudio, trabajo y dedicación.

A todos ellos, gracias.

Resumen

Dada la enorme cantidad de juegos existentes, una pregunta recurrente ha sido ¿Cuál es la mejor estrategia a seguir para maximizar mi beneficio? en otras palabras, ¿Qué acciones debo elegir entre las posibilidades que tengo para obtener la mayor recompensa posible? Esta cuestión, al darle una perspectiva formal resulta no ser trivial, cada configuración del tablero tiene una probabilidad asociada de ganar, definida por todos los posibles desenlaces que tiene el juego desde esa posición. Sin embargo, obtener este valor es imposible computacionalmente ya que esto implicaría revisar todas las posibles partidas dado que el número de estas es descomunal. Para dar solución a esto, existen diversos algoritmos que intentan aproximar este valor, normalmente mezclando el conocimiento previo del dominio con la búsqueda exhaustiva en el conjunto de todas las partidas, para comprobar si una acción mejora nuestra situación actual.

En este trabajo se presenta Monte Carlo Tree Search, un algoritmo que introduce una forma completamente distinta de estimar la probabilidad de ganar de cada estado de la partida. El método será generar simulaciones de partidas en las que las decisiones se toman de forma aleatoria. De esta forma, podremos calcular el porcentaje de veces que una simulación resulta victoriosa en comparación con las veces que se ha simulado, dándonos así una aproximación del valor buscado. Veremos que cuantas más simulaciones lancemos, el resultado será más ajustado a la realidad. Conoceremos 2 versiones, la versión dinámica o MCTS, y la versión plana o PMCTS. Introduciremos métodos de optimización y una perspectiva teórica del funcionamiento.

Probaremos este algoritmo en juegos, y para ello obtendremos diversas implementaciones de las 2 versiones y las adaptaremos para jugar al Reversi y al Ajedrez. Los adversarios a batir serán algoritmos varios que irán aumentando gradualmente su dificultad y nos obligarán a ir adaptando los algoritmos y mejorándolos para que sean capaces de dominar.

Por último introduciremos algunos algoritmos utilizados en el campo de la Inteligencia Artificial conocido como aprendizaje automático o Machine Learning, para que el lector posea las herramientas necesarias para entender el funcionamiento del jugador automático más potente que existe en la

actualidad: AlphaGo. Este programa ha sido capaz de vencer a los mejores jugadores del mundo en el Go, considerado hasta ahora como imposible de abarcar para la inteligencia artificial. Monte Carlo Tree Search es uno de los elementos clave en este código y veremos cuál es su papel.

Palabras clave

Monte Carlo, Teoría de Juegos, Estocástico, IA, Memoria, Ajedrez, Reversi

Abstract

Since the beginning of the last century, classic games were studied from a formal perspective adding mathematical reasoning. With the arrival of computer sciences and artificial intelligence, creating automatic players has been a recurrent problem. The part of mathematics which studies this kind of situations is called Game theory and it include classifications, representations and solutions to how to optimize the decision making. One of the typical representations in games where the players take turns to make moves is using a tree, the way to explore it is the base of how to obtain efficient players. Some classic exploration algorithms may work for simple games, but for those with more than one player, we will need approaches like Minimax.

In this work we will present Monte Carlo Tree Search, an algorithm based in avoiding to explore the whole tree because this is impossible and, instead it will try, using random game simulations, to approximate a numeric value of how useful each possible move really is. We will discover 2 versions, the dynamic version or MCTS, and the plain version or PMCTS. We will introduce methods of optimization as well as a theoretical perspective of why it works.

We will try the algorithm in game, for that we will obtain several implementations of both versions and we will adapt them to play Reversi and Chess. The opponents to beat will be diverse algorithms that will gradually increase their difficulty and will make us adapt our tools and improve them so we are able to win.

Finally, we will introduce some algorithms used in the Artificial Intelligence field known as Machine Learning, so that the reader owns the tools to understand how the most advanced player nowadays works: AlphaGo. This program has been able to beat the top Go players, considered until now as impossible to encompass by artificial intelligence. Monte Carlo Tree Search is one of the key components in this code and we will explain it's paper.

Keywords

Monte Carlo, Game Theory, Stochastic, AI, Memory, Chess, Reversi

Índice

Índice de figuras

Índice de tablas

Chapter 1

Introduction

Motivation

One of the current challenges in Artificial Intelligence is the development of algorithms to implement efficient automatic players. These must give a answer, not only correct but also as wise as possible with the finality of winning the game they were designed for. This development is not trivial at all.

The creation of these players, as the optimization for them to be better every time, reports benefits further than the world of video games or table games. After all, the idea is to optimize the decision making processes. Given this, there are applications of these algorithms in economics, politics, etc.

The main problem is to decide which of the available moves must be chosen next. For that the most direct approach is to explore the game tree and try to find the best option. This point of view is offered by some algorithms whose most remarkable representative is *Minimax*.

However, although this is an optimal algorithm, it presents a problem: the gigantic combinatorial explosion that it generates. Even though it offers good results for simple challenges, as soon as we try it in more complex systems, it is impossible to treat.

There are several alternatives to solve this issue. Throughout this work, we will explore one of them: Monte Carlo Tree Search, commonly shortened as MCTS. This algorithm's idea is, instead of generating each and every possible branch in the game tree, to use stochastic processes to choose which node to expand, moreover, the way they are expanded will also have a very strong presence of random elements. This way we get to spend much less memory than with the traditional approaches. Furthermore, the development of the tree will use simulations to generate a heuristic value for each

node, characterizing them according to if they are more or less promising, optimizing the move election process.

Objectives

We are going to define some objectives in order to explore the MCTS algorithm:

1. Obtaining a global vision of the predecessors of MCTS, as well as the tools we will use.
2. Give a formal explanation of the functioning of MCTS as well as for it's plain version known as PMCTS.
3. Obtain an implementation of MCTS for the game known as Othello, compare ir with other algorithms and check how it's behavior improves depending on the parameter chosen.
4. Implementing an MCTS based player which is capable of playing chess and check the differences with it's othello version.
5. Reviewing the most modern approaches using MCTS and the necessary tools for it.

In the conclusions section, we will review how far we accomplished them.

Memory structure

We will encompass the marked objectives throughout the memory. The first chapter will give a theoretical review of the algorithms that preceded MCTS as Minimax or Alpha-beta. In addition, it will provide a clear explanation of all the tools and situations that we will use in other chapters as game theory, game tree development, Chess, Othello, etc.

The second chapter will show the Mote Carlo Tree Search algorithm, the way it works and it's implementation. On the other hand, we will also see a detailed explanation of PMCTS, the MCTS' version without growth. Finally, we will analyze the possibility of reusing memory and how we can approximate the theoretical heuristic of each node through simulations.

In the third chapter we will obtain an implementation of MCTS and PMCTS. We will it's performance with the some others algorithms as Random, Alpha-beta and Backtracking in Othello and later in Chess. We will show and analyze the results obtained. Next we will modify the parameters so it gets better results against their adversaries.

Finally the fourth chapter will explain us how we can use machine learning to improve the results that MCTS offers, for that we will first review some of the cutting edge machine learning algorithms. Additionally we will see how this has been applied to one of the most popular artificial intelligence events of the last years: AlphaGo.

Capítulo 1

Introducción

Motivación

Uno de los desafíos actuales de la Inteligencia Artificial es el desarrollo de algoritmos para implementar jugadores automáticos eficientes. Estos deben dar una respuesta no solo correcta sino lo más acertada posible con el fin de ganar en el juego para el que se haya desarrollado. Este desarrollo no es en absoluto trivial.

La creación de estos jugadores así como la optimización para que sean cada vez mejores, reporta beneficios más allá del mundo de los videojuegos o los juegos de tablero. Al fin y al cabo, se trata de optimizar la toma de decisiones, dado esto, existen aplicaciones de estos algoritmos para temas como economía, política, etc (Brams, 2001).

El problema principal es decidir cuál de las jugadas disponibles se debe elegir a continuación. Para ello, lo más directo es intentar prever las jugadas a las que podemos llegar desde la posición en la que nos encontramos. La estructura donde se almacenan las jugadas predichas es conocida como el árbol de juego. Esta es la aproximación que ofrecen algunos algoritmos de los cuales el mayor exponente es conocido como *Minimax*, el cual exploraremos en más detalle en el siguiente capítulo.

El problema de esta aproximación es que es imposible revisar todo el árbol para saber con certeza lo bueno que es un nodo. Por tanto, para juegos complicados, se ha intentado paliar esta situación previendo únicamente un número determinado de jugadas, pero esto implica que debemos ser capaces de valorar si son buenos o malos estados intermedios, de los que a priori no tenemos información. Una forma de evaluarlo es obtener conocimiento experto del dominio, es decir, ideas o estrategias que se hayan probado y que sepamos con certeza que funcionan. Aunque puede sonar como una buena idea, el conocimiento experto es muy difícil de conseguir, además, práctica-

mente nunca está estructurado de forma que sea sencillo codificar o expresar de manera formal.

Por todo esto, vamos a explicar y a trabajar un algoritmo que ofrece una perspectiva completamente distinta: el árbol de búsqueda de Monte Carlo, comúnmente abreviado como MCTS por sus siglas en inglés. La idea de este algoritmo es, en lugar de generar todas y cada una de las posibles jugadas del árbol de juego, utilizar simulaciones de partidas estocásticas para estimar la probabilidad que tendremos de ganar desde un estado dado. De esta manera conseguimos gastar muchísimo menos tiempo computacional que con aproximaciones más tradicionales.

Objetivos

Vamos a definir objetivos a cumplir con el fin de explorar el algoritmo de MCTS:

1. Obtener una visión global de los predecesores de MCTS, así como de las diversas herramientas que utilizaremos.
2. Dar una explicación formal del funcionamiento de MCTS así como de su versión plana PMCTS.
3. Obtener una implementación de MCTS para el juego conocido como Reversi, compararlas con otros algoritmos y comprobar como mejora su rendimiento en función de los parámetros elegidos.
4. Implementar un jugador basado en MCTS que sea capaz de jugar al ajedrez y comprobar las diferencias con su versión para el Reversi.
5. Revisar las aproximaciones más modernas que utilizan MCTS y las herramientas necesarias para ello.

En las conclusiones, revisaremos en qué medida se han conseguido estos objetivos.

Estructura de la memoria

Abarcaremos los objetivos marcados a lo largo de la memoria. El capítulo 2 dará una revisión teórica de los algoritmos que han precedido a MCTS como minimax o alfa-beta. Además, proporcionará una explicación clara de todas las herramientas y situaciones que utilizaremos en los otros capítulos, como la teoría de juegos, el desarrollo de árboles de juego, el ajedrez, el Reversi, etc.

El capítulo 3 nos planteará el algoritmo del Árbol de búsqueda de Monte Carlo, su funcionamiento y su implementación. Por otra parte veremos también una explicación detallada de PMCTS, la versión de MCTS sin crecimiento. Para terminar analizaremos las posibilidades de reaprovechamiento de memoria y cómo mediante simulaciones podemos aproximar el valor heurístico teórico de cada nodo.

En el capítulo 4 obtendremos una implementación de MCTS y de PMCTS. Compararemos el rendimiento de estas con el de los algoritmos Random, Alpha-beta y Backtracking en distintos juegos. Mostraremos y analizaremos los resultados obtenidos. Después de esto, modificaremos los parámetros de nuestros algoritmos para mejorar sus resultados frente a sus adversarios.

Por último el capítulo 5 nos explicará cómo podemos utilizar el aprendizaje automático para mejorar los resultados que nos ofrece MCTS explorando cómo funcionan las redes neuronales y cómo esto se ha aplicado a uno de los eventos más mediáticos de la inteligencia artificial de los últimos años: AlphaGo.

Capítulo 2

Trabajo relacionado

Definiciones

Comenzamos este capítulo definiendo algunos conceptos básicos que se van a utilizar a lo largo de este documento, y que son indispensables a la hora de entender el contenido desarrollado a continuación.

- **Espacio de estados.** Llamaremos de esta forma al conjunto de todas las posibles configuraciones en las que se puede encontrar el juego.
- **Árbol**. Un árbol es una estructura de datos formada por los contenedores de información llamados nodos y las relaciones entre ellos llamadas vértices. En un árbol las relaciones no pueden ser cíclicas, y todos los nodos están relacionados.

  Con la definición de árbol viene cierta terminología que utilizaremos a lo largo de este documento (Pieterse y Black, 1999):

  - La **raíz** es el nodo superior del árbol.
  - Un nodo A es **hijo** de otro B cuando estos dos están relacionados por un vértice, y B está más cerca de la raíz que A.
  - La noción de **padre** es la inversa que la de hijo.
  - Diremos que un nodo es **hoja** cuando no tiene hijos.
  - Llamaremos **profundidad** de un nodo al número de vértices que lo separan de la raíz.
- **Cola.** Llamaremos así a una estructura de datos que sigue la política FIFO (*First In First Out*), es decir, el primer dato que llega a la estructura es el primero en salir. Una cola nos permitirá añadir un elemento al final y consultar o extraer el primero (Cormen et al., 2009).

- **Cola de prioridad** De forma similar a las colas normales, las colas de prioridad nos permitirán añadir y extraer el elemento inicial, sin embargo, este será siempre aquel con un valor menor (o mayor) de prioridad (Cormen et al., 2009).
- **Estado terminal.** Diremos que un estado es terminal si no podemos llegar a ningún otro desde él. En los juegos que trataremos, significará que el juego ha acabado, pues no tenemos ninguna acción que realizar.
- **Factor de ramificación.** Dado un árbol llamaremos factor de ramificación al número de hijos de cada nodo. Será usual referirnos al factor de ramificación máximo y medio ya que normalmente este varía.
- **Heurística.** Una función heurística o simplemente heurística, es una función que clasifica las alternativas en los algoritmos de búsqueda en cada paso de ramificación basado en la información disponible con el fin de decidir qué rama seguir (Pearl, 1984).

Teoría de juegos

Los juegos de mesa han funcionado siempre como entretenimiento, sin embargo, desde principios del siglo XX, con el matemático francés Émile Borel en 1921, se comenzó a dar una perspectiva formal a este tipo de pasatiempos.

Años después, en 1928, John Von Neumann, matemático húngaro, desarrolló lo que se conoce como teoría Minimax que llevaría al desarrollo del algoritmo óptimo de juego conocido como Minimax (Ross, 1997). Este algoritmo es la base de futuros desarrollos, como el objeto de este estudio.

Todos estos conocimientos se agrupan en la parte de la matemática que se conoce como Teoría de Juegos. Podemos definirla de la siguiente forma:

"Parte de la matemática aplicada que utiliza modelos para estudiar la toma de decisiones de los agentes con base en el análisis de sus interacciones estratégicas"(Gonzalez, 2015).

Dada la amplia gama de juegos y situaciones que podemos encontrar, vamos a evitar dar una definición formal de juego hasta haber entrado más en materia. Sin embargo, para ser considerado como tal, vamos a exigir la presencia de tres elementos comunes a todos los juegos (Gonzalez, 2015):

- **Jugadores.** Los agentes que intervendrán en el juego.
- **Acciones.** Las distintas estrategias entre las que puede elegir determinado agente.

- **Pagos.** El beneficio obtenido por cierto jugador dada su elección de acciones.

Para poder analizar minuciosamente cada juego, es necesario, clasificar formalmente los juegos a los que nos enfrentamos, para saber cómo abarcar la búsqueda de estrategias. Por este motivo, la teoría de juegos ofrece una clasificación completa de los juegos a abordar dependiendo de diversos factores.

- **Individuales o multijugador.** En dependencia del número de agentes interviniendo en el juego.
- **Cooperativo, competitivo o mixto.** Clasificaremos los juegos en función de si los jugadores deben colaborar, luchar unos contra otros o son libres para decidir si colaborar o no y con quién, respectivamente.
- **Simultáneos o secuenciales.** En dependencia de si los jugadores eligen las estrategias a seguir simultáneamente o, por contrario, se turnan secuencialmente para realizar acciones.
- **Información perfecta o imperfecta.** Esta clasificación se corresponde con la cantidad de información que tiene a su disposición cada jugador a la hora de tomar una decisión. El ajedrez es un ejemplo de información perfecta pues ambos jugadores conocen el tablero y las posición de las piezas, que es el conocimiento relevante del juego. Por su parte un ejemplo de información imperfecta es el póker, pues cada jugador conoce sus cartas, pero ignora las de la mano de los oponentes.
- **Longitud finita o infinita.** Distinguiremos aquellos juegos que tienen un final claro en un tiempo determinado de aquellos que en un principio no tienen por qué llegar a un final.

Juegos simultáneos

Ahora debemos encontrar una manera visual, formal y eficiente de representar el conocimiento que tenemos del juego en cuestión. Para los juegos simultáneos, en general utilizaremos lo que se conoce como forma normal. Antes de introducirla definimos un juego simultáneo como una terna:

$$(N, a, u)$$

con $N = \{1, ..., n\}$ el conjunto de naturales que representa los n jugadores que participan. Llamaremos $a_i = \{a_{ij}\}$ al conjunto de acciones disponibles para el jugador i. El conjunto total de acciones viene dado por $a = a_1 \times ... \times a_n$. Por último $u = \{u_1, u_2, ..., u_n\}$ es el conjunto de funciones $u_i : a \to \mathbb{R}$, las

1/2	a_{21}	a_{22}	...
a_{11}	$u_1((a_{11}, a_{21})), u_2((a_{11}, a_{21}))$	$u_1((a_{11}, a_{22})), u_2((a_{11}, a_{22}))$	...
a_{12}	$u_1((a_{12}, a_{21})), u_2((a_{12}, a_{21}))$	$u_1((a_{12}, a_{22})), u_2((a_{12}, a_{22}))$	...
a_{13}	$u_1((a_{13}, a_{21})), u_2((a_{13}, a_{21}))$	$u_1((a_{13}, a_{22})), u_2((a_{13}, a_{22}))$	...
$\vdots$	$\vdots$	$\vdots$	$\ddots$

Tabla 2.1: Modelo de disposición de la forma normal para dos jugadores

conocemos como funciones de utilidad que nos devuelven los pagos a cierto jugador i por el vector de acciones elegido por todos los jugadores. (Jackson, 2011). Con estos ingredientes, la forma normal nos representa una matriz donde los elementos de la misma son los pagos que reciben los jugadores en dependencia de la combinación de estrategias elegidas por cada uno. Esta representación funciona especialmente bien cuando se trata de dos jugadores, ya que se puede disponer en una matriz bidimensional. La tabla 2.1 nos da un ejemplo de cómo se disponen los datos en forma normal.

Ejemplo: El dilema del prisionero

Probablemente el ejemplo más conocido de juego simultáneo es el dilema del prisionero. La situación es la siguiente: "La policía sabe que dos individuos han cometido un crimen aunque no puede probarlo, pero sí que tiene pruebas para incriminarlos por un delito menor. Ante la alarma social levantada, se ven obligados a tener un culpable rápido, por lo que ponen a ambos sospechosos en celdas separadas y les hacen la siguiente propuesta: Si delatas al otro como autor del crimen te perdonamos tu delito menor"(Aguado, 2015).

En este supuesto nos encontramos ante dos acciones posibles, a las que llamaremos “Callar” y “Delatar”. Suponiendo una condena de 1 año por el delito menor, 3 en caso de cumplir la sentencia por el mayor individualmente y 2 por hacerlo de forma conjunta nuestro juego queda de la siguiente forma:

$$
\begin{aligned}
(N, a, u) = (\{1, 2\}, \{&Callar, Delatar\}, u) \\
&u_1((Callar, Delatar)) = u_2((Delatar, Callar)) = 3 \\
&u_1((Delatar, Callar)) = u_2((Callar, Delatar)) = 0 \\
&u_1((Callar, Callar)) = u_2((Callar, Callar)) = 1 \\
&u_1((Delatar, Delatar)) = u_2((Delatar, Delatar)) = 2
\end{aligned}
$$

Podemos representar el problema de forma extensiva como se muestra en la tabla 2.2.

Prisionero 1 \Prisionero 2	Callar	Delatar
Callar	1,1	3,0
Delatar	0,3	2,2

Tabla 2.2: Dilema del prisionero plasmado en forma normal

Juegos secuenciales

A diferencia de los juegos simultáneos, en los secuenciales, los diversos agentes se alternan a la hora de tomar decisiones, se conoce cada una de estas alternancias como turno. Después del turno de cada jugador encontramos el juego en una nueva configuración, debido a las decisiones tomadas. A estas las llamaremos estados. Por tanto, podemos definir los juegos secuenciales como una 4-tupla:

$$(N, S, a, u)$$

Donde una vez más $N = \{1, ..., n\}$ es el conjunto de jugadores. Por su parte S es el espacio de estado. En lo que concierten a las acciones, $a : N \times S \to A$ es una función que nos devuelve las acciones disponibles para un jugador en cierto estado, por tanto $A = \{\{a_i\}_{i=1...k_1}, \{a_i\}_{i=1...k_2}...\}$, es importante destacar que una acción en este caso es una función $a_i \in acc \in A \;\; a_i : S \to S$. Por último, $u : N \times S \to \mathbb{R}$ es una función que nos devuelve los pagos para un jugador en cierto estado.

Para exponer la evolución temporal inherente a la secuencialidad de los juegos no podemos valernos de la forma normal, por eso vamos a utilizar la estructura de árbol para crear lo que llamaremos forma extensiva. Utilizaremos los nodos para guardar los estados del juego y los hijos de cada nodo serán el resultado de la aplicación de cierta acción a dicho estado. A este tipo de árbol lo llamaremos árbol de juego.

Árboles de juego y ejemplos

El caso más sencillo de árbol de juego aparece cuando abarcamos juegos con $N = \{1\}$. En este caso todas las decisiones recaen sobre el mismo agente. Esto facilita la exploración del árbol, pues nos basta con indagar lo suficiente hasta llegar a un nodo terminal ganador. Después basta con ejecutar las acciones definidas por el camino que lleva al nodo en cuestión. Ejemplos de estos juegos pueden ser el solitario o el blackjack.

La exploración de los árboles de juego se complica al agregar más jugadores, pues los demás agentes tendrán, en principio, tanto poder de decisión como el primero. Los nodos en el mismo nivel en estos árboles comparten el jugador al que le toca realizar una acción y la elección del jugador asignado a cada nivel dependerá de cómo esté definida la variación de turnos.

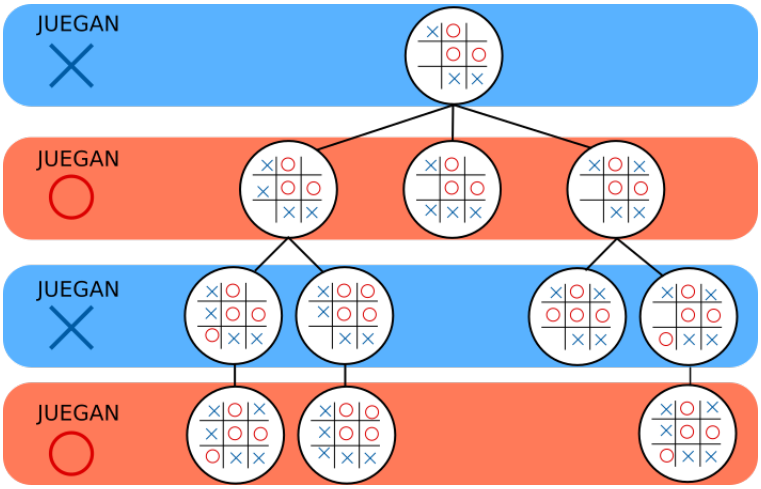


Figura 2.1: Ejemplo de árbol de decisión para un estado dado en el juego del tres en raya

La figura 2.1 muestra un ejemplo visual del árbol de juego para el tres en raya partiendo de un estado dado. La representación completa del árbol es excesivamente grande para aparecer en un folio.

En este tipo de árboles no nos basta con una exploración básica del árbol para conseguir una buena estrategia a seguir, pues aunque encontremos un camino que nos lleve a una situación de victoria, no nos corresponde decidir la totalidad del camino por lo que sería inservible. Para obtener soluciones en esta parte se han desarrollado otros algoritmos, de los cuales el más conocido es Minimax, que veremos en la siguiente sección.

Búsqueda en árboles e inducción hacia atrás

Explorar árboles para obtener cómo llegar a ciertos nodos, es un problema clásico en el campo de la inteligencia artificial, al que hay que dar solución en muchos sistemas inteligentes actuales. En esta sección exploramos soluciones clásicas y su aplicación en la teoría de juegos. De ahí, pasamos al algoritmo Minimax que nos mostrará cómo explorar un árbol de juego con múltiples agentes y veremos algunas optimizaciones.

Algoritmos tradicionales

Explorar un árbol desde la raíz hasta algún nodo que nos interese, es un problema que se presenta frecuentemente, y que podemos encontrar en infinidad de situaciones distintas. En el caso que nos atañe son la herramienta

perfecta para dar una solución a los juegos individuales. A pesar de haber numerosas formas de realizar la exploración, vamos a hablar de los métodos más conocidos: la búsqueda primero en profundidad, la búsqueda primero en anchura y la búsqueda A*. Esta última nos servirá también para introducir el concepto de heurística que nos será muy útil más adelante.

Primero en profundidad

El algoritmo de búsqueda primero en profundidad (conocido como DFS por *Depth First Search* en inglés), explorará de forma recursiva el árbol, descendiendo por el primer nodo que encuentre hasta llegar a un nodo hoja. Una vez llegados a este punto, en caso de no ser el nodo que buscamos, volvemos al nodo predecesor y elegimos la siguiente opción disponible. Podemos ver el algoritmo en pseudocódigo a continuación:

```
DFS(nodo n)
    SI n = objetivo
        DEVOLVER n
    SINO
        PARA CADA hijo EN n.hijos
            DFS(hijo)
```

Del mismo modo, existe una versión para la exploración de otras estructuras de datos diferentes a los árboles, que implementa un control de repeticiones, pero no indagaremos en ella (Rusell y Norvig, 2003).

Primero en Anchura

El algoritmo de búsqueda primero en anchura (o BFS por Breadth First Search en inglés), propone explorar primero todas los nodos que están al mismo nivel y después pasar al siguiente. Para la implementación de este algoritmo vamos a utilizar una cola. Vemos el pseudocódigo:

```
BFS(nodo raiz)
    Q = Cola
    Q.introducir(raiz)
    MIENTRAS NO Q.vacia
        n = Q.extraer()
        SI n = objetivo
            DEVOLVER n
        SINO
            PARA CADA hijo EN n.hijos
                Q.introducir(hijo)
```

Igualmente ignoramos la versión con control de repeticiones (Rusell y Norvig, 2003).

Búsqueda A*

El algoritmo A* aparece como un ejemplo de búsqueda informada. Al contrario que en las anteriores, en esta vamos a contar con información extra que nos informa de si nos estamos acercando o alejando de la situación conforme exploramos el árbol. Esta información se tratará de una heurística como se ha definido anteriormente.

El ejemplo más visual de utilización de A* es en los sistemas de guía por GPS, en los que debemos resolver el problema de ir desde un punto A a otro B en un mapa de la forma más eficiente. La heurística que se suele elegir en estos casos es la distancia en línea recta entre una posición intermedia y el destino.

Utilizaremos una cola de prioridad para que el siguiente nodo a evaluar sea aquel que sea más prometedor. Es decir, aquel que minimice la función:

$$f(n) = g(n) + h(n)$$

Dónde $h(n)$ es la heurística del nodo actual al objetivo y $g(n)$ el coste real del camino desde la raíz al nodo actual (ZENG y CHURCH, 2007).

En la figura 2.2, vemos un ejemplo de los elementos de A*. Se muestra una simplificación del funcionamiento de un software de GPS para obtener un camino. En este caso podemos ver sobre los vértices la distancia real por carretera entre las ciudades que conformarían nuestra $g(n)$ mientras que sobre los nodos (con números verdes), se ve la distancia en línea recta a nuestro nodo objetivo, estos valores conforman $h(n)$. En el ejemplo el nodo origen es Madrid, marcado en amarillo y el objetivo Castellón de la Plana, en verde.

Minimax

Tenemos ya unas cuantas herramientas para explorar árboles. Ahora, estos algoritmos no pueden ser aplicados a los árboles de juego, ya que como se ha mencionado anteriormente no corresponde a un mismo agente elegir todos los caminos. Dada esta dificultad surge Minimax.

En cada juego, debemos definir una variable de control a la que llamaremos valor de utilidad, que será diferente para cada juego y nos indicará si el nodo en cuestión proporciona un buen resultado. Este valor, vendrá definido

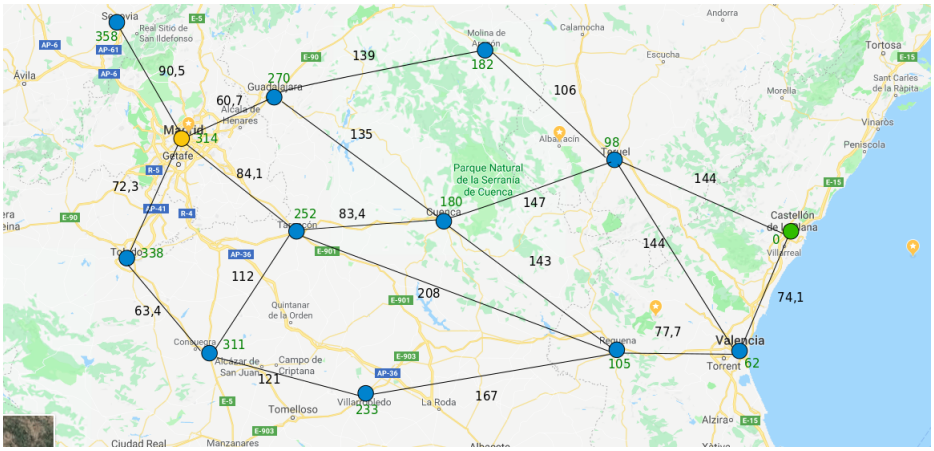


Figura 2.2: Simplificación de la búsqueda de caminos en GPS

por los nodos terminales. Debemos tener en cuenta que nuestro adversario tratará de maximizar su valor de utilidad, por lo que entre sus posibles acciones, elegirá siempre aquella que minimice nuestro valor de utilidad (van den Herik, 1999). Siendo así, el algoritmo funciona de esta manera:

1. Generar el árbol de juego completo.

2. Obtener el valor de utilidad de los nodos terminales.

3. Traspasar el valor de utilidad a los nodos superiores, eligiendo el máximo en caso de que el turno corresponda al jugador, y el mínimo en caso de que sea del oponente. Esta parte se llama inducción hacia atrás.

4. Elegir la acción correspondiente al nodo con el mayor valor de utilidad.

El problema es precisamente el primer punto, ya que en juegos complejos es impracticable generar el árbol de juego entero. En el ejemplo que mostrábamos anteriormente del *tres en raya* tenemos un factor de ramificación inicial de 9 (en la primera jugada podemos colocar nuestra ficha en nueve posiciones) que disminuye en uno en cada jugada. Saber el número exacto de partidas posibles es complicado debido a que no todas las partidas requieren los 9 movimientos, a partir de 5 puede haber un ganador y terminar la partida. A pesar de esto, podemos ver que el número de partidas distintas está acotado superiormente por $9 \cdot 8 \cdot 7 \cdot 6..,1 = 9! = 362880$ partidas.

El problema es igualmente complicado si queremos saber el número total de nodos que tiene el árbol, pero podemos hacer la misma acotación. De esta forma, en cada nivel tendremos $\frac{9!}{(9-n)!}$ nodos, por lo que generaremos

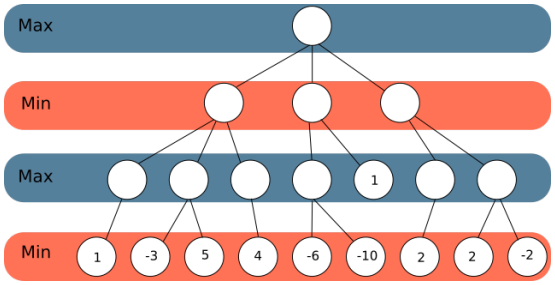


Figura 2.3: Obtención inicial de un árbol de juego

un árbol de juego de tamaño cercano a:

$$\sum_{n=1}^{9} \frac{9!}{(9-n)!} = 986409$$

Si un juego tan sencillo genera prácticamente un millón de nodos, es evidente que no resulta factible aplicarlo a juegos más complejos. El ajedrez por ejemplo tiene un factor de ramificación inicial de 20 (16 movimientos de los peones y 4 de los caballos) que puede aumentar, y durante las etapas intermedias un valor típico (Laramée, 2000) por lo que simplemente al predecir 10 jugadas obtendríamos un árbol de cerca de $2,75 \times 10^{15}$ nodos.

Una solución común a este problema es utilizar un Minimax limitado por profundidad. Con este algoritmo exploraremos hasta predecir todas las jugadas posibles en los n siguientes turnos y obtendremos igualmente el valor de utilidad de los nodos hoja. La figura 2.3 muestra cómo se genera un árbol de profundidad 4, por su parte, la 2.4 muestra cómo funciona la inducción hacia atrás de Minimax.

El problema de limitar por profundidad a Minimax es que estaremos llegando a un nodo intermedio, no tiene por qué ser un nodo ganador o perdedor. Por tanto ¿Qué valor de utilidad asignamos a estos nodos? La solución suele ser utilizar conocimiento experto del dominio, es decir, razonamientos basados en la experiencia de juegos anteriores, estrategias que se conoce que han dado un buen resultado, etc.

La gran desventaja de esta idea es que el conocimiento experto no solo es complicado de conseguir, sino que además no suele ser fácil de codificar en forma de reglas para que el algoritmo pueda trabajar. Otro problema es que este conocimiento no tiene por qué ser óptimo, puede haber estrategias mejores que simplemente no se hayan desarrollado. Ejemplos de este conocimiento se verán más adelante cuando se expliquen los juegos que se explorarán.

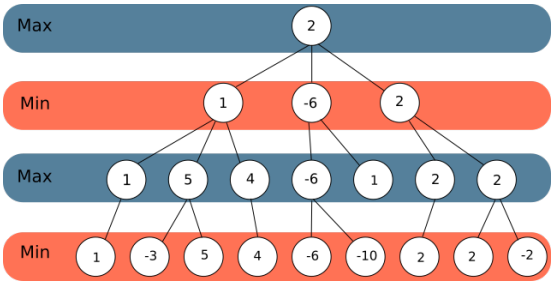


Figura 2.4: Ejecución de la inducción hacia atrás en Minimax

Poda Alpha-Beta

Vamos a presentar ahora la optimización de Minimax conocida como poda alfa-beta (más conocido por el nombre en inglés: Alpha-Beta pruning). Dado el elevado coste de Minimax, debemos intentar ahorrar tantos cálculos como sea posible. En este sentido, nos damos cuenta de que podemos evitar la exploración de ciertas ramas si sabemos de antemano que no vamos a obtener un valor mejor que el ya obtenido. Veamos el siguiente ejemplo de una operación de maximización - minimización como las que encontramos en Minimax:

$$\text{máx}\{\text{mín}\{3, 12, 8\}, \text{mín}\{2, x, y\}, \text{mín}\{14, 5, 2\}\} = \text{máx}\{3, z, 2\} \quad = 3$$

Dado que $z \leq 2$, sabemos que nunca va a ser elegido por lo que una vez obtenido el valor 2, no hace falta explorar los valores x e y ya que no van a influir (Rusell y Norvig, 2003).

Para formalizar esto vamos a definir dos variables que llamaremos α y β que definimos de la siguiente forma:

- α El valor de la mejor (es decir mayor) elección que hemos encontrado hasta ahora en cualquier punto de elección para el camino de MAX.
- β El valor de la mejor (es decir menor) elección que hemos encontrado hasta ahora en cualquier punto de elección para el camino de MIN.

Si en la exploración de un nodo de maximización encontramos un valor que supere a β, no será necesario seguir explorando las demás ramas de ese nodo. Simétricamente, si en un nodo de minimización encontramos una rama con un valor de utilidad menor que α, podaremos[1] igualmente. Podemos ver el algoritmo en pseudocódigo a continuación:

[1]Evitaremos la exploración de dicha rama

```
AlphaBeta(nodo)
    v = Maximizar(nodo, -infinito, +infinito)
    DEVOLVER v

Maximizar(nodo, alpha, beta)
    SI esEstadoTerminal(nodo) ENTONCES DEVOLVER utilidad(nodo)
    v = -infinito
    PARA CADA a EN Acciones(nodo)
        v = max(v, Minimizar(AplicarAccion(a, nodo), alpha, beta))
        SI v >= beta DEVOLVER v
        alpha = max(alpha, v)
    DEVOLVER v

Minimizar(nodo, alpha, beta)
    SI esEstadoTerminal(nodo) ENTONCES DEVOLVER utilidad(nodo)
    v = -infinito
    PARA CADA a EN Acciones(nodo)
        v = min(v, Maximizar(AplicarAccion(a, nodo), alpha, beta))
        SI alpha >= v DEVOLVER v
        beta = min(beta, v)
    DEVOLVER v
```

Juegos para experimentar

Presentamos a continuación algunos juegos que vamos a utilizar para nuestra investigación. Además de esto, profundizaremos en estrategias y conocimientos del dominio útiles para implementar jugadores eficientes.

Reversi

El *Reversi*, también conocido como *Othello*, es un juego de mesa cuyo origen proviene de Inglaterra, donde fue comercializado por primera vez en 1880 por Lewis Waterman y John W. Mollet. A día de hoy, el juego es conocido en todo el mundo, teniendo una fuerte presencia en Japón (française dÓthello, 1998).

Para jugar necesitaremos un tablero de 8x8 casillas y 64 discos de tamaño inferior a una casilla. Los discos deberán ser de colores distintos en cada cara, estos colores tradicionalmente son negro y blanco, aunque es muy típico también encontrarlos en rojo y azul. Las casillas del tablero se suelen referenciar utilizando una letra de la a a la h de izquierda a derecha para indicar la columna, y un número del 1 al 8 comenzando de arriba hacia abajo para

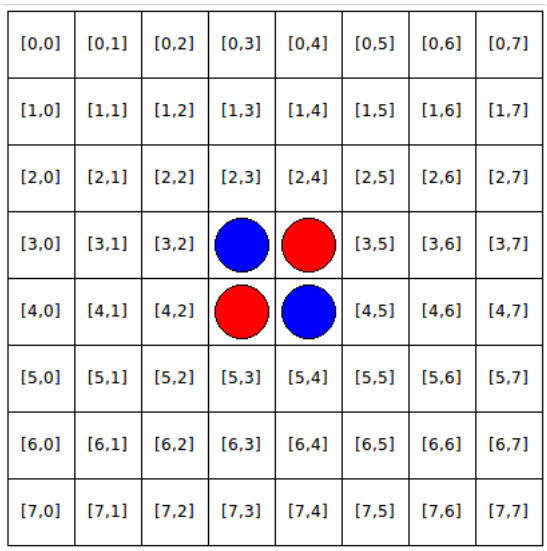


Figura 2.5: Comienzo del juego en el Reversi

señalar la fila. Al implementar el tablero la elección de representación más simple es utilizar un array bidimiensional de 8×8 posiciones, de esta forma, las casillas quedan representadas con un vector $[i, j]$ con $0 \leq i, j \leq 7$, esta será la representación que aparecerá en la mayoría de las imágenes. Jugarán dos jugadores y cada uno tendrá asignado un color.

Reglas

Según Rose (2005) las reglas del juego se definen como sigue:

1. El juego comienza con discos negros en d5 y e4, y discos blancos en d4 y e5 como se puede ver en la Figura 2.5.
2. Los jugadores alternan turnos, comenzando el jugador de fichas de color negro (o azul).
3. Un movimiento legal consiste en colocar un nuevo disco en una casilla vacía y dar la vuelta a uno o más de los discos del oponente.
4. Se dará la vuelta a cualquier disco del color del oponente, comprendido entre el disco que se acaba de jugar y cualquier otro del color del jugador preexistente en el tablero.Llamaremos a esto hacer un "Sándwich". Se pueden crear sándwiches en forma vertical, horizontal y diagonal. Para crear un sándwich, todas las casillas entre el disco nuevo y el preexistente del mismo color deben estar ocupadas por discos del oponente sin casillas vacías.

5. Los discos se pueden girar en varias direcciones en un mismo movimiento. Cualquier pieza que quede atrapada en un sándwich deberá ser volteada, implicando que el jugador al que le toca no tiene derecho a elegir si alguna pieza se gira o no.

6. Un nuevo disco no se puede jugar a no ser que gire al menos un disco del oponente. En caso de que un jugador no tenga movimientos legales para realizar, pasa el turno al siguiente jugador que prosigue jugando hasta que exista algún movimiento legal para ese jugador.

7. Si un jugador tiene al menos un movimiento legal posible, no tiene la posibilidad de pasar turno.

8. El juego continúa hasta que el tablero se queda sin casillas vacías o ninguno de los dos jugadores tiene jugadas legales posibles.

9. Una vez terminado el juego, gana aquel jugador que tenga más discos de su color en el tablero.

Clasificación

Según la clasificación presentada en la sección anterior sobre teoría de juegos, podemos ver que el Reversi se trata de un juego multijugador ya que tendremos 2 jugadores por partida, competitivo pues la cooperación está prohibida, secuencial pues los jugadores se turnan para realizar acciones, de información perfecta ya que ambos jugadores son capaces de observar todos los datos relevantes para la partida, que en este caso es el tablero y las fichas. Por último, sabemos que es de longitud finita pues en cada turno hay obligatoriamente una casilla menos vacía, acercándonos al final de la partida al completar el juego, quedando así un juego de, como mucho, 60 turnos.

Estrategias

Veamos algunas estrategias que suelen seguir los jugadores en el Reversi, estas estrategias nos darán una idea de qué jugadas son mejores y cuales son peores. Hay cientos de estrategias y libros sobre cómo obtener jugadas buenas llegando a niveles altos de sofisticación. Aquí presentamos algunas que son objetivamente eficaces y se han probado como una buena forma de crear heurísticas para jugadores automáticos.

- Las esquinas son puntos estratégicos cuyo control es prioritario. Dada su condición, es imposible crear un sándwich que incluya alguna esquina, por lo que una vez que se controla, esta no puede cambiar de color. No solo son valores seguros sino que además dan un control amplio de

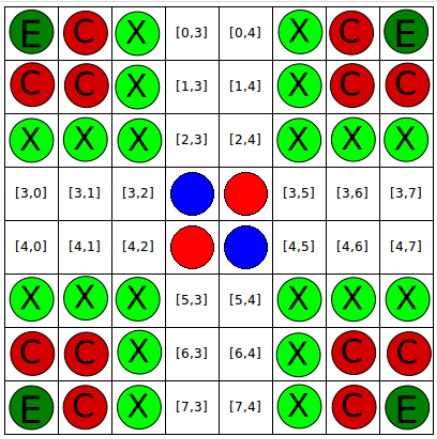


Figura 2.6: Visualización de las casillas más y menos provechosas

la zona de alrededor, pues influye en una de las diagonales principales del tablero y a dos de los cuatro lados. Llamaremos a estas casillas E.

- Por ende, las casillas contiguas a las esquinas son una mala elección en la que jugar, debido a que al dominarlas brindan a nuestro contrincante la posibilidad de controlar la esquina. Denominaremos a estas casillas C.

- Por último, una buena idea es conseguir controlar las casillas adyacentes a las casillas C, dado que estas dan la posibilidad o, en un caso en el que no tuviera otra posibilidad, obligan al oponente a jugar en una casilla C. Llamaremos a estas casillas X.

En consecuencia, una heurística eficiente debería premiar en gran medida jugar en una esquina, castigar jugar en una casilla C y un premio algo menor por jugar en las X.Podemos ver estas ideas reflejadas en la Figura 2.6.

Ajedrez

El ajedrez es uno de los juegos más conocidos mundialmente y su influencia se puede ver a día de hoy en gran parte de nuestra cultura. Sus orígenes se remontan a la India septeptrional antes del siglo VII con su predecesor el *Chaturanga*, que era inicialmente un juego para 4 personas. El juego fue navegando de civilización en civilización y fue evolucionando y adaptándose. A finales del siglo XV se definen las reglas de lo que se conoce como ajedrez moderno, presumiblemente en Valencia entre los años 1470 y 1490 manifestándose en el primer documento conocido sobre esta disciplina tal y como la conocemos hoy, el poema valenciano "Schachs dámor".

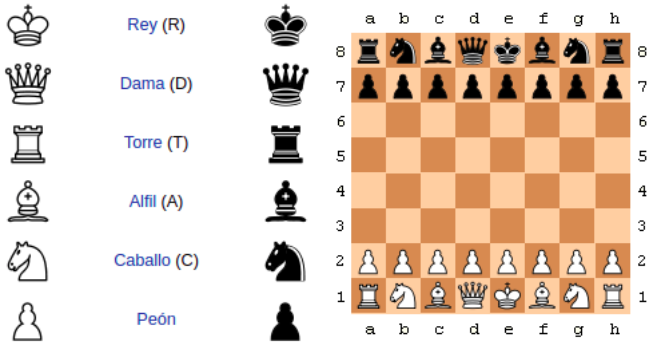


Figura 2.7: Piezas del ajedrez y colocación inicial

Necesitaremos ciertos elementos para jugarlo, comenzando por un tablero de 8x8 casillas las casillas se alternan en dos colores: blanco y negro. Deberemos contar también con 16 fichas de color blanco y sus equivalentes 16 de color negro, estas serán: 8 peones, 2 torres, 2 caballos, dos alfiles, 1 rey y 1 reina. La figura 2.7 muestra estos elementos dispuestos de la manera que da comienzo al juego.

Reglas

Presentamos los movimientos de las piezas, indispensables para entender el funcionamiento del juego:

- La Torre se mueve en vertical u horizontal tantas casillas como desee pero solo en una dirección en cada turno.
- El Alfil puede moverse tantas casillas como quiera en diagonal. De esta forma es curioso ver que cada jugador posee un alfil que se mueve en las casillas negras otro en las blancas.
- La Dama (o reina) representa las unión de ambas piezas anteriores, se puede mover tantas casillas como quiera en vertical, horizontal o diagonal.
- El Rey puede moverse en cualquier dirección pero solo una casilla a la vez.
- El caballo se mueve en forma de “L”, es decir, dos casillas de forma horizontal y luego una de forma vertical o al revés. El caballo es la única pieza que puede “saltar” a otras piezas, es decir, no importa si hay fichas entre la posición inicial y la final.

- El peón es un poco más complejo, puede moverse únicamente una casilla de forma vertical en dirección a su oponente generalmente. Cuando se encuentra en la posición inicial puede decidir entre mover una o dos casillas en su dirección habitual. Sin embargo, el peón no puede comer piezas en la misma forma en la que se mueve, necesita encontrar piezas que estén en alguna de las dos casillas adyacentes de forma vertical a la casilla en la que se movería normalmente.

Existen otras reglas sobre transformación de peones, comer al paso o enrocarse, que no comentaremos pues en la versión del ajedrez que se presentará más adelante para probar los algoritmos no se encuentran implementadas.

Una vez entendidos los movimientos, los jugadores alternan turnos decidiendo cada vez una entre todas las posibilidades de juego que se le presentan. Si una pieza se mueve a la casilla en la que se encuentra una pieza contraria se la "come" y se extrae del tablero. El objetivo del juego es llevar al rey contrario a una posición en la que se encuentre bajo un ataque directo que no pueda evadir, esta situación se conoce como Jaque Mate. Si el rey se encuentra bajo ataque, pero existe la posibilidad de escapar, nos encontraremos en jaque y esto limitará las jugadas posibles a aquellas que saquen al rey del jaque.

Utilizaremos la notación oficial en español para referirnos a los movimientos. Esta se hace mediante la inicial de cada ficha en mayúscula y la posición a la que se ha movido dada por una letra para indicar la columna y un número para la fila tal como se ha visto en la figura 2.7. Las jugadas se marcan en renglones marcando el número de turno a la izquierda seguido de la jugada de blancas y después la de negras.

Clasificación

Observamos que el Ajedrez es un juego multijugador, para 2 agentes para ser exactos, puramente competitivo, secuencial, de información perfecta.

El punto que genera mayor interés para su estudio es que se trata de un juego de longitud infinita dado que podemos encontrar series de movimientos que no lleven a la finalización del juego. Un ejemplo muy simple sería:

```
1. Cc3 Cc6
2. Cb1 Cb8
...
2n-1. Cc3 Cc6
2n. Cb1 Cb8
...
```

De esta forma hemos creado un ciclo de jugadas válidas de longitud arbitraria

que no llevará a la finalización del juego.

Estrategias

Dada la popularidad del juego y la complejidad inherente a él, existen incontables documentos que explican estrategias, ideas o formas de jugar para maximizar las posibilidades de ganar al adversario. Recorrer todos estos conceptos escapa a la finalidad de este trabajo, sin embargo, vamos a mostrar una idea básica que nos servirá para desarrollar heurísticas para el juego: el valor de las piezas.

Para esto vamos a dar un valor a cada ficha. Vemos en Wikipedia (Chess) una clasificación de los valores que se ha intentado asignar a las piezas en distintas aproximaciones de expertos en el sector, se muestran allí las referencias y se obtiene una media así que utilizaremos este valor:

- Peón: 1 punto
- Alfil: 3.3 puntos
- Caballo: 3.2 puntos
- Torre: 5.2 puntos
- Reina: 9.6 puntos

Otra idea que utilizaremos es priorizar aquellos caminos que den lugar a un jaque a nuestro favor pues suele ser, indiscutiblemente, una posición de poder, ya que limita mucho los movimientos del adversario, como ya se ha comentado.

Capítulo 3

Funcionamiento de MCTS

En este capítulo vamos a introducir el algoritmo sobre el que versa este estudio, el árbol de búsqueda Montecarlo. En primera instancia comenzaremos explorando un algoritmo clave que necesitaremos, el límite superior de confianza.

Límite superior de confianza

El algoritmo del límite superior de confianza (en adelante UCB por Upper Confidence Bound en inglés), aparece para solucionar lo que se conoce como *Multi-armed Bandit machine problem* es decir, el problema de la máquina tragaperras de muchos brazos. Lo formulamos de la siguiente forma:

"Nos enfrentamos a una máquina traga perras complicada que tiene N brazos de los que podemos tirar. Cuando tiramos de uno de estos obtendremos una recompensa, sin embargo estas recompensas no son seguras ya que cada brazo tiene un ratio de recompensa propio que desconocemos, por lo que estirar de cualquiera es arriesgado. Queremos, en la medida de lo posible maximizar nuestro beneficio."(White, 2013)

Esto genera el problema de lo que conocemos como proporción Exploración-Explotación. Llamamos explotación a indagar sobre una opción (brazo) que parece muy prometedora mientras que entendemos por exploración el variar entre las distintas opciones pues podemos encontrar algunas mejores a la que, por el momento, resulta más provechosa. UCB da solución heurística a este problema, lo definimos como:

$$UCB = arg\ max_j \left\{ \bar{X}_j + \sqrt{\frac{2 \ln n}{n_j}} \right\}$$

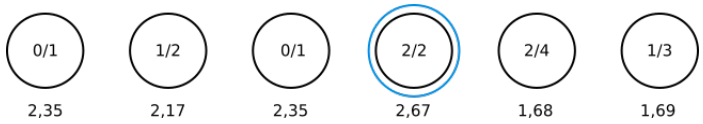


Figura 3.1: Ejemplo de selección utilizando UCB

Dónde $\bar{X}_j$ es la recompensa media obtenida en el nodo j, n el número total de ejecuciones y n_j el número de ejecuciones en el nodo j. El primer sumando favorece la explotación de nodos pues si un nodo es bueno tendrá una recompensa media alta que aumentará el valor UCB, mientras que el segundo la exploración ya que es un valor alto en aquellos que han sido poco explorados en comparación con el número total de exploraciones. La figura 3.1 nos muestra un ejemplo de aplicación de UCB, después de 13 tiradas nos encontramos con los siguiente valores para los brazos donde se marcan las veces que se ha tirado de cada brazo detrás de la barra y las veces que se ha obtenido recompensa antes. Se muestra en azul el brazo que debería ser estirado a continuación.

Monte Carlo Tree Search

El algoritmo del árbol de búsqueda de Montecarlo o *Monte Carlo Tree Search* (MCTS) busca desarrollar un árbol que explore parcialmente el espacio de estados de un problema, intentando que los nodos en los que se dedica más esfuerzo sean los que van a dar un mejor resultado.

La idea del algoritmo es precisamente evitar la exploración hasta los nodos terminales pues como se ha comentado esto resulta imposible. Por contra, vamos a estimar la probabilidad de ganar desde el estado intermedio en el que nos encontremos, es decir, calcula un valor heurístico. El medio para estimar este valor será realizar simulaciones de partidas en el que las decisiones se toman de forma aleatoria, por esto diremos que se llama de un método estocástico. La estimación del valor de cada nodo se hace únicamente mediante la simulación de partidas por lo que podemos construir jugadores que tomen buenas decisiones sin ninguna necesidad de conocimiento experto del dominio en el que se trabaja.

Existe una gran diversidad de implementaciones de MCTS, sin embargo, en esta sección vamos a explorar el funcionamiento más tradicional y en las subsiguientes veremos las variaciones que han sido necesarias para nuestros casos concretos.

El algoritmo pretende explorar un árbol y calcular un valor heurístico a cada nodo para poder elegir el que sea posiblemente el mejor. Partiremos de

un único nodo raíz y en cada iteración iremos ampliando el conocimiento que tenemos del espacio de estados. Cada una de las mentadas iteraciones del código se divide en 4 partes claramente diferenciadas: selección, expansión, simulación y retropropagación (Browne et al., 2012). Esta partes contestan respectivamente a las preguntas ¿Qué nodo expandimos?¿Cómo lo expandimos?¿Cuán bueno es el nodo resultante?¿Qué efecto tiene esto sobre el resto del árbol?

Vamos a necesitar guardar cierta información en los nodos del árbol. Cada uno almacenará:

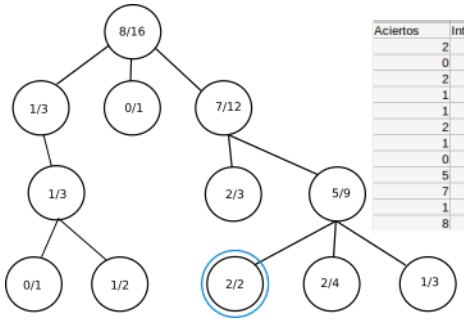
- El estado del juego.
- Un contador de veces que ha sido expandido dicho nodo.
- Un contador de veces que la expansión ha resultado en una victoria.
- La acción por la que se ha llegado a ese nodo.
- Punteros al padre y los hijos.

El objetivo es obtener una estimación de cuán buenas son las distintas acciones que tenemos a nuestra disposición. Para esto simplemente al expandir los nodos jugaremos partidas aleatorias desde cierto nodo y comprobaremos si hemos ganado o perdido, este valor se manda hacia arriba en la jerarquía de nodos, de esta forma, todos los nodos tienen la información de sus hijos.

Selección

El proceso de selección se ocupa de elegir qué nodo es el mejor exponente para ser expandido. Es aquí dónde cobra sentido el desarrollo previo sobre las máquinas tragaperras, pues utilizaremos el algoritmo UCB para seleccionar cuál es el siguiente nodo a expandir.

Sin embargo, aparece un problema y es que como acabamos de comentar (y como se explica en profundidad en la sección sobre retropropagación) la información que encontramos en los padres es la suma de la de los hijos. Por esto no podemos utilizar UCB simple, nos vemos obligados a introducir UCT (Upper Confidence bound for Trees). Simplemente sustituiremos el valor de n por el número de veces que se ha seleccionado el nodo padre en lugar del total, de esta forma, se utiliza UCB de forma local permitiendo un funcionamiento correcto del algoritmo. La figura 3.2 muestra un ejemplo de selección utilizando UCT.



Aciertos	Intentos	Intentos Padre	Recompensa media	Valor de exploración	UCT
2	2	9	1	1,4823038074	2,4823038074
0	1	16	0	2,354820045	2,354820045
2	3	12	0,6666666667	1,2870914626	1,9537581293
1	3	16	0,3333333333	1,3595559869	1,6928893202
1	2	3	0,5	1,048147074	1,548147074
2	4	9	0,5	1,048147074	1,548147074
1	3	9	0,3333333333	1,2102959906	1,5436293239
0	1	3	0	1,4823038074	1,4823038074
5	9	12	0,5555555556	0,7431026024	1,298658158
7	12	16	0,5833333333	0,6797779934	1,2631113268
1	3	3	0,3333333333	0,8558085022	1,1891418355
8	16	16	0,5	0,5887050113	1,0887050113

Figura 3.2: Ejemplo de selección utilizando UCT

Expansión

Una vez elegido el nodo a expandir, debemos dilucidar cómo lo hacemos, aquí empieza a aparecer la aleatoriedad. Hay diversas formas, veamos algunas de ellas:

Expansión simple

Es la opción más sencilla y con menor coste computacional. Consiste en que, dada la lista de acciones que tenemos disponibles para el nodo seleccionado, elegimos una de ellas, la ejecutamos y obtendremos un nuevo estado, que conformará el nuevo nodo hijo. Después continuamos hacia la siguiente fase (Chaslot et al., 2008).

Expansión múltiple

Se trata de una ampliación de la anterior, para ello, en lugar de elegir una única acción para expandir, escogemos n y pasamos a la siguiente fase. En este caso deberíamos investigar cuál es el valor óptimo para n pues un valor muy elevado significaría un gasto excesivo en esta fase que multiplicaría el gasto en la siguiente.

Simulación

Una vez elegidos los nodos a expandir, debemos comprobar cuán buenos son estos. Esa es la función de la fase de simulación. Para conocer este valor, partiremos del estado que nos muestra el nodo a simular y comenzamos a tomar decisiones aleatorias, dentro de la lista de acciones disponibles, para definir un camino hasta llegar a un nodo terminal. El estado terminal tendrá

un valor dependiendo del problema que estemos abarcando (ganar o perder, el nivel de beneficio obtenido, etc). Tomaremos entonces este como nuestro valor heurístico para el nodo.

El ejemplo más sencillo es un juego en que los posibles resultados sean ganar o perder. En este caso, desde el estado que nos proponga el nodo a simular, generamos elecciones aleatorias de las jugadas disponibles hasta llegar a un estado en el que bien habremos ganado o perdido. En caso de ganar asignaremos 1/1 al valor heurístico del nodo expandido, mientras que en caso de perder lo marcaríamos a 0/1 donde el número detrás de la barra indica el número de veces que se ha simulado.

La fase de simulación puede ser lenta si el problema que se está tratando tiene muchos posibles movimientos, si existen ciclos o si el espectro es infinito. Por eso en muchas ocasiones es más eficiente, en lugar de obligar al código a llegar a un estado terminal, elegir algún otro tipo de valor que nos indique cuán bueno es el nodo y hacer una exploración de profundidad limitada o con un tiempo limitado. Esta opción, por contra, nos hace perder uno de los puntos más importantes del algoritmo, la independencia del dominio, pues no necesita nunca conocimiento previo del juego, solamente saber si ha ganado o perdido, la capacidad de valorar un estado no terminal implica que sabemos de antemano lo bueno que es un estado.

Realizaremos simulaciones sobre los candidatos a expandirse, ahora, debemos elegir cuántas simulaciones llevar a cabo, ya que realizar una sola sobre cada candidato puede involucrar un error muy grande a la hora de valorarlo, mientras que un número de simulaciones excesivamente grande ralentiza demasiado el algoritmo. En general la toma de decisiones en las simulaciones consumen muy poco tiempo computacional ya que no es necesario detenerse prácticamente a meditar las opciones. El mayor coste de estas viene precisamente de calcular las opciones posibles en cada estado por lo que en juegos complicados se ralentiza de forma significativa.

Retropropagación

En este momento, tenemos un valor de cuán bueno es el nodo (o los nodos) nuevo(s). Nuestra tarea será trasladar esta misma información a todos los nodos y en particular a los hijos de la raíz que son, al fin y al cabo, entre los que tenemos que elegir. Este proceso se llama *retropropagación* y es tan simple como recorrer el árbol desde el nodo hoja que acabamos de crear hasta la raíz sumando los valores de resultados e intentos a los contadores de cada nodo que encontremos por medio.

Una vez llegados a este punto, hemos completado una iteración del MCTS, y con ello, obtenido un árbol con más nodos y mejor informado sobre lo buenas que son las distintas decisiones. A continuación se lleva a término la

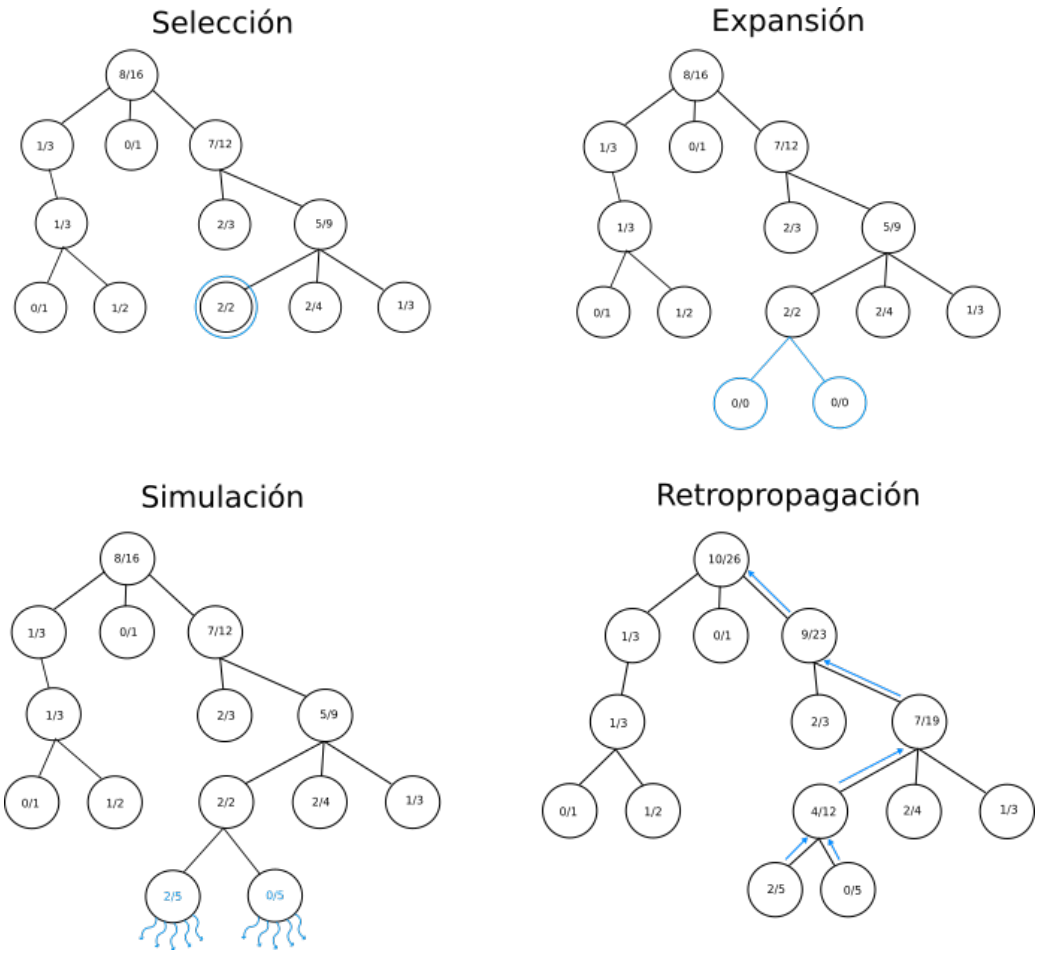


Figura 3.3: Resultado de una iteración de MCTS

siguiente iteración, y es entonces cuando aparece la pregunta: ¿Cuándo paramos? En un problema con un espacio de búsqueda virtualmente infinito, como puede ser, por ejemplo, el ajedrez (del que se hablará más adelante), podríamos explorar el árbol tanto como quisiéramos y siempre mejoraríamos la solución. Por ello, normalmente la estrategia es detenerse dada una condición que puede ser un límite de tiempo, un número máximo de nodos expandidos, una profundidad máxima, etc. Al cumplimiento de esta condición estaremos obligados a elegir el nodo que tenga mejor resultado hasta el momento.

Retomando el ejemplo expuesto en la figura 3.2, podemos ver el resultado de realizar una expansión múltiple con $n = 2$, 5 simulaciones por nodo y la retropropagación del resultado. Se muestra en la Figura 3.3

Mostramos a continuación la estructura general de MCTS en pseudocó-

digo:

```
MCTS(raiz)
    MIENTRAS hayTiempo() HACER
        seleccionado = seleccion(raiz)
        expandido = expansion(seleccionado)
        resultado = simulacion(expandido)
        retropropagacion(expandido, resultado)
    DEVOLVER maxheuristica(raiz.hijos)

seleccion(arbol)
    PARA CADA nodo EN arbol
        calcularUCT()
    DEVOLVER argmaxuct(arbol)

expansion(nodo)
    PARA i=0 HASTA numeroExpansiones
        accion = randomchoice(accionesDisponibles(nodo))
        nuevoNodo = ejecutar(accion, nodo)
        nodo.agregarHijo(nuevoNodo)

simulacion(nodo)
    aciertos = 0
    PARA i=0 HASTA numeroSimulaciones
        cnodo = copiar(nodo)
        MIENTRAS noTerminado(cnodo) HACER
            accion = randomchoice(accionesDisponibles(cnodo))
            cnodo = ejecutar(accion, cnodo)
        SI ganar(cnodo)
            aciertos = aciertos +1
    DEVOLVER aciertos

retropropagacion(nodo, aciertos)
    nodo.aciertos = nodo.aciertos+aciertos
    nodo.intentos = nodo.intentos+numeroSimulaciones
    MIENTRAS nodo.padre != NULL HACER
        nodo <= nodo.padre
        nodo.aciertos = nodo.aciertos+aciertos
        nodo.intentos = nodo.intentos+numeroSimulaciones
```

PMCTS

Vamos a implementar también la versión de MCTS conocida como *Plain Montecarlo Tree Search*. Esta versión difiere de la original en la fase de expansión que es ignorada. De esta forma el árbol de juego será estático sin ningún tipo de crecimiento. Recae sobre nosotros decidir la forma del árbol que será generado antes de comenzar las iteraciones. La generación de este árbol inicial es algo arriesgada, pues, según esté definido podría obviar opciones beneficiosas que ya no serán tomadas en consideración en ningún momento.

En la versión que hemos implementado, para dar solución a esto, generamos 2 niveles del árbol completos, todas las posibles acciones a tomar por nosotros y por nuestro contrincante. Una vez realizado este paso inicial desarrollamos las otras tres fases tantas veces como iteraciones sean posibles. Eligiendo el árbol de esta manera podemos ahorrar el cálculo de UCT como en el apartado anterior y calcular directamente UCB sobre los nodos hoja, pues la selección del nodo raíz o de algún nodo no hoja implicaría una simulación que forzosamente pasaría por alguna de las hojas, lo que es ineficiente.

Podemos apreciar entonces el pseudocódigo del cuerpo principal de este algoritmo:

```
PMCTS(raiz)
    generarArbolInicial(raiz)
    MIENTRAS hayTiempo() HACER
        seleccionado  = SeleccionPorUCB()
        Simulacion(seleccionado)
        Retropropagacion(seleccionado)
   DEVOLVER maxheuristica(raiz.hijos)
```

Reaprovechamiento de memoria

Cada vez que nos encontremos en un estado nuevo, deberemos generar un árbol de juego que parta únicamente de la raíz, para estimar una vez más la heurística de cada nodo. Una idea interesante es reaprovechar un árbol que tuviéramos creado ya de la última vez que lo generamos, para obtener mejores resultados. El problema con esto es que no podemos predecir qué movimiento va a elegir el adversario o los adversarios, con lo que es posible que este no esté entre los que nosotros hemos considerado.

Para ello lo más sencillo es guardar la estructura del árbol de la ejecución anterior y antes de comenzar las nuevas iteraciones, descendemos dos veces por la raíz del árbol imitando las acciones tomadas en el juego real. En caso de

no tener un vértice a ese nodo empezaríamos de cero, pero si tenemos suerte podemos partir de un árbol con unas cuantas heurísticas ya calculadas, lo que nos permitiría en el mismo tiempo computacional obtener los resultados equivalentes a más simulaciones.

Aproximación probabilística

En esta sección se utilizan conceptos de probabilidad que no han sido introducidos en apartados anteriores por lo que se recomienda que el lector tenga conocimientos de probabilidad y estadística. En todo caso, si el lector desea repasar las definiciones de estos conceptos, puede hacerlo en el apéndice A.

Utilizando el método de Montecarlo estamos intentando aproximar la distribución de probabilidad de una variable aleatoria discreta que nos marcará las expectativas de ganar que podemos tener dada la elección de cierta acción en un estado dado. Si llamamos X a esta VA. Tenemos dos resultados posibles:

$$X = ganar \qquad X = no\ ganar$$

Según está definido el algoritmo nuestro objetivo es aproximar $p(X = ganar|e)$ mediante simulaciones siendo e una configuración dada dentro del espacio de estados. Vamos a comprobar que efectivamente la probabilidad de que obtener una victoria desde cierto estado es igual a la de que una simulación resulte ganadora. De esta forma nos veremos en las condiciones de aplicar el teorema de Glivenko-Cantelli (que se enuncia y explica en la sección siguiente) y demostrar que efectivamente mediante este procedimiento estamos aproximando la probabilidad real de ganar desde ese estado.

Esta última probabilidad la obtenemos de forma recursiva: comenzando con el caso base, si el nodo e es un estado terminal habremos ganado o perdido por lo que:

$$p(X = ganar|e) = \begin{cases} 1 & si\ e\ es\ un\ estado\ ganador \\ 0 & ecc \end{cases}$$

Llamaremos a_i^j a ejecutar la acción a_i en el punto de toma de decisión número j y denotaremos por $a_i(e)$ al estado resultante de aplicar la acción a_i a e. En

caso de no tratarse de un estado terminal tendremos que:

$$\begin{aligned} p(X = ganar|e) &= \sum_i p(X = ganar|e, a_i^1)p(a_i^1) \\ &= \sum_{i,j} p(X = ganar|e, a_i^1, a_j^2)p(a_i^1, a_j^2) \\ &= ... \\ &= \sum_{i,...,j} p(X = ganar|e, a_i^1, ..., a_j^k)p(a_i^1, ..., a_j^k) \end{aligned}$$

Siendo k el número necesario para que $a_i^1(...(a_j^k(e)))$ sea un estado terminal. La distribución de probabilidad conjunta, viene dada por:

$$p(A, B) = P(A \cap B) = p(A|B)p(B)$$

aplicando a nuestro caso:

$$\begin{aligned} p(a_i^1, a_l^2 ..., a_j^k) &= p(a_l^2 ..., a_j^k|a_i^1)p(a_i^1) \\ &= ... \\ &= \prod_m p(a_m^r|a_j^{r-1}, ..., a_i^1) \end{aligned}$$

Estos datos son conocidos para nosotros, suponiendo que todas las acciones tienen la misma importancia, podemos suponer que $p(a_i^1) = \frac{1}{N^{(1)}}$ siendo $N^(1)$ el número de acciones entre las que elegir. Así sucesivamente $p(a_j^2|a_i^1) = \frac{1}{N_i^{(2)}}...$

Este es exactamente el comportamiento que tiene nuestra simulación, pues al elegir aleatoriamente la acción a tomar la probabilidad en nuestra simulación $p(a)$ coincide con la aproximada anteriormente. A su vez los estados terminales reportan 1 o 0 dependiendo de su valía por lo que la probabilidad de una simulación de obtener una victoria es exactamente la probabilidad de ganar que hemos obtenido teóricamente.

Las simulaciones aproximan la distribución

Ya hemos comprobado que las simulaciones tienen la misma función de distribución F que la variable aleatoria definida en el apartado anterior. Vamos ahora a ver que efectivamente al aumentar el número de simulaciones realizadas nuestro resultado heurístico se acerca a la distribución F. Este resultado está recogido en el teorema de Glivenko-Cantelli que se enuncia a continuación (Ángel Villegas, 2005).

Teorema 1 (Glivenko-Cantelli) *Si se tiene una muestra aleatoria simple de tamaño n de una población X, con función de distribución $F(x)$, para cualquier número real positivo arbitrario ε, se tiene que*

$$\lim_{n\to\infty} P\left\{\sup_{x\in\mathbb{R}} |F *_n (x) - F(x)| \geq \varepsilon\right\} = 0$$

La demostración de este teorema se puede consultar en Fisz (1963).

Este teorema nos indica que la probabilidad de que el valor supremo de la diferencia entre la función de distribución estimada con la muestra aleatoria simple (en nuestro caso los resultados de las simulaciones) y la función de distribución teórica, sea mayor que cierto valor ε es 0 cuanto el tamaño de la muestra tiende a infinito.

Con esta sección hemos demostrado que a través de las simulaciones obtendremos un valor heurístico que efectivamente aproxima el valor teórico correspondiente a cada nodo.

Capítulo 4

Aplicación de MCTS a juegos

En este capítulo revisamos la implementación de MCTS adaptado a dos juegos, el Reversi y el Ajedrez.

Aplicación al Reversi

El Reversi es un juego relativamente sencillo, ya que carece de ciclos, pues una vez colocamos una pieza, estamos un paso más cerca de acabar el juego que, como mucho, acabará en tantas jugadas como casillas haya en el juego menos las que están ocupadas inicialmente, es decir: $64 - 4 = 60$ jugadas. Dados estos datos, debemos entonces decidir cómo abarcar el problema eligiendo las componentes del algoritmo que consideremos mejores.

- **Selección**
  Elegiremos el método del UCB.

- **Expansión**
  En este aspecto vamos a probar diversas aproximaciones para comparar los resultados.

  - Vamos a comenzar probando PMCTS generando un desarrollo inicial de hasta los nietos del nodo actual. Desde ahí probaremos con simulaciones, en particular generando 70 y 140 simulaciones por turno.
  - Utilizaremos un algoritmo de Montecarlo con expansión simple

- **Simulación**
  Vamos a realizar simulaciones hasta el final del juego y en caso de ganar devolveremos el valor 1 y en caso de perder devolveremos el valor 0.

- **Retropropagación**
  Mandamos hacia arriba en el árbol los valores obtenidos de la simulación como se ha explicado anteriormente.

Random

Comenzamos enfrentando a PMCTS y a MCTS contra el algoritmo más sencillo de la colección que veremos. Se trata de *Random*, que, como bien se desprende de su nombre, se limitará a tomar una decisión aleatoria entre las diversas acciones posibles. Obviamente este algoritmo carece de cualquier tipo de conocimiento específico del dominio.

En la tabla 4.1 se muestran los resultados de la simulación de 30 partidas de Random contra PMCTS y 30 contra MCTS en el caso de que tengan limitación de 70 simulaciones por turno para cada uno, en particular el número asociado a cada jugador en cada experimento es la cantidad de fichas que ha conseguido mantener de su color al término del enfrentamiento. Como era de esperar, dada la sencillez de Random, ambas versiones obtienen un buen resultado, en el caso de PMCTS hemos obtenido un 90 % de victorias y un 10 % de derrotas, mientras que en MCTS ganamos el 83 % de las veces aunque debido a los empates, el porcentaje de derrotas sigue siendo un 10 %.

Vamos a aumentar el número de simulaciones que realizamos a 140 para comprobar si obtenemos mejores resultados, pues ya hemos visto en el apartado anterior la demostración teórica de como aumentar el número de simulaciones nos debe dar un resultado mejor. Plasmamos estos resultados en la tabla 4.2. Podemos observar que los resultados mejoran en PMCTS hasta elevar el porcentaje de victorias hasta el 96 %, por contra MCTS ha variado ligeramente perdiendo una victoria respecto al ensayo anterior aunque mantiene en un 10 % el ratio de derrotas.

Búsquedas simples en árboles de juego

Vamos ahora a aumentar un poco la dificultad. Enfrentaremos nuestros algoritmos contra una búsqueda en profundidad (DFS), en particular buscaremos dentro del árbol el primer nodo hoja que represente una partida ganadora, y nos decantaremos por ejecutar la acción que se desarrolla en esa rama. En caso de que no encontremos ninguno al recorrer el árbol de juego entero, nos decantaremos por una acción cualquiera.

Ya hemos comentado anteriormente los problemas que aparecen por los que no se suelen utilizar este tipo de algoritmos en árboles de juego, sin embargo, de esta forma aseguramos que la rama que se está eligiendo incluye, al menos, un resultado ganador, implicando así un poco más de complejidad que la simple selección aleatoria de acciones.

Experimento	Random	PMCTS		Random	MCTS
1	18	46		31	33
2	20	44		31	33
3	17	47		23	36
4	47	17		28	36
5	15	49		47	17
6	20	43		26	38
7	21	43		33	31
8	28	36		26	38
9	22	42		19	45
10	28	36		32	32
11	22	42		21	43
12	12	52		20	44
13	12	52		24	40
14	16	48		21	43
15	24	40		18	46
16	21	43		27	37
17	13	51		25	39
18	26	38		21	43
19	18	46		32	32
20	12	52		27	37
21	37	27		15	49
22	33	31		25	39
23	22	42		14	50
24	16	48		27	37
25	22	42		27	37
26	30	34		20	44
27	14	50		23	41
28	17	47		20	44
29	8	56		42	22
30	26	38		25	39
Media	21,23	42,73		25,66	38,16
Victorias	3	27		3	25

Tabla 4.1: PMCTS y MCTS (70 simulaciones) vs Random

Experimento	Random	PMCTS		Random	MCTS
1	19	45		12	52
2	23	41		30	34
3	14	50		24	40
4	14	50		13	51
5	25	39		38	26
6	20	44		18	46
7	25	39		32	32
8	13	51		25	39
9	26	38		16	48
10	29	35		25	39
11	16	48		19	45
12	23	41		24	40
13	12	52		6	57
14	18	46		43	21
15	15	49		21	43
16	24	40		23	41
17	26	38		18	46
18	16	48		6	57
19	21	43		41	23
20	12	52		32	32
21	27	37		23	41
22	34	30		23	41
23	19	45		11	53
24	20	44		30	34
25	15	46		30	34
26	21	43		32	32
27	22	42		25	39
28	20	44		31	33
29	19	45		20	44
30	24	50		31	33
Media	20,4	43,5		3	24
Victorias	1	29		3	24

Tabla 4.2: PMCTS y MCTS (140 simulaciones) vs Random

La tabla 4.3 nos muestra cómo se comportan ambas versiones de MCTS contra la búsqueda en profundidad. Vemos que seguimos obteniendo muy buenos resultados con solo 70 simulaciones. A pesar de esto vemos que DFS es capaz de ganar con más frecuencia que Random. En particular PMCTS consigue un 83 % de victorias mientras que MCTS se queda en un 76,6 %.

Dado que encontramos un margen de mejora más amplio que en el caso anterior vamos a aumentar el número de simulaciones de forma sustancial, en particular vamos a multiplicarlas por 10 hasta llegar a las 700. Plasmamos estos resultados en la tabla 4.4. Tal y como habíamos predicho de forma teórica, los resultados mejoran pues se obtiene una mejor estimación de las heurísticas de los estados. Concretamente, PMCTS sube a un 97 % de victorias mientras que MCTS aumenta a un 83 %, es decir, un 14 % y un 7 % más respectivamente.

Alpha-Beta

Vamos ahora a enfrentarnos al algoritmo más potente que hemos presentado hasta ahora: Minimax. Para ser exactos vamos a enfrentarnos a 2 versiones de Minimax, ambas limitadas por profundidad, comprobaremos entre 3 y 4 jugadas en adelante, y ambas estarán optimizadas mediante la poda Alpha-beta que se explicó en el capítulo de trabajo relacionado. El valor de utilidad que van a tener en cuenta es el número de fichas de nuestro color que tenemos en cierto estado, dado que el objetivo es intentar conseguir que al final de la partida sea lo mayor posible.

Las 2 versiones, a las que llamaremos fuerte y débil, difieren en que la primera utilizará la heurística que se expuso en la sección sobre las estrategias para el reversi, mientras que la segunda utilizará únicamente el valor de utilidad.

Para hacer que el algoritmo tenga la heurística en cuenta sumaremos al valor de utilidad 20 puntos en caso de que se vaya a conquistar una esquina y 2 en caso de que se trate de una casilla 'X'. Por contra, restaremos 5 puntos en caso de jugar en una casilla 'C'.

En la tabla 4.5 se muestran los resultados de 30 ejecuciones con la versión débil. Es notable que hemos conseguido crear dos rivales potentes frente a uno de los algoritmo más fuertes que se conocen pues PMCTS ha conseguido un 70 % de victorias, y MCTS un 33,3 %. Cabe destacar la acentuación de la diferencia entre ambas versiones, ya habíamos visto desde el principio que la versión plana daba generalmente un resultado levemente mejor pero aquí difiere en un porcentaje del 43 %.

Por otra parte, en la tabla 4.6 se muestra el enfrentamiento contra la versión reforzada con heurística de Alpha-beta. Claramente este añadido au-

Experimento	Profundidad	PMCTS		Profundidad	MCTS
1	23	41		23	41
2	17	47		16	48
3	26	38		26	38
4	34	30		11	53
5	10	54		33	31
6	50	14		15	49
7	21	43		27	37
8	23	41		26	38
9	23	41		32	32
10	14	50		30	34
11	27	37		54	10
12	25	39		26	38
13	16	48		13	21
14	13	51		9	55
15	14	50		1	62
16	22	42		8	56
17	29	35		20	44
18	17	46		0	42
19	29	35		44	20
20	26	38		25	39
21	35	29		26	38
22	23	41		16	48
23	55	9		33	31
24	25	39		28	36
25	24	40		39	25
26	47	15		10	54
27	24	40		12	52
28	17	46		21	43
29	20	44		35	29
30	23	41		19	45
Media	25,066	38,8		22,6	39,633
Victorias	5	25		6	23

Tabla 4.3: PMCTS y MCTS (70 simulaciones) vs búsqueda en profundidad

Experimento	Bactracking	PMCTS		Backtracking	MCTS
1	18	46		11	53
2	19	45		12	52
3	24	40		52	12
4	22	42		20	44
5	23	41		42	22
6	21	43		21	43
7	24	40		30	34
8	21	43		26	38
9	16	48		23	41
10	26	38		21	43
11	29	35		22	42
12	20	44		28	36
13	13	51		31	33
14	37	27		26	38
15	16	48		40	24
16	20	44		19	45
17	18	46		17	47
18	17	47		29	35
19	14	50		18	46
20	24	40		28	36
21	15	49		31	33
22	25	39		36	28
23	19	45		21	43
24	23	41		21	43
25	18	46		20	44
26	14	50		24	40
27	21	43		25	39
28	17	47		30	34
29	20	44		22	42
30	25	39		42	22
Media	20,633	43,366		26,266	37,733
Victorias	1	29		5	25

Tabla 4.4: PMCTS y MCTS (700 simulaciones) vs búsqueda en profundidad

Experimento	Alpha-Beta débil	PMCTS		Alpha-Beta débil	MCTS
1	22	42		33	0
2	26	38		48	16
3	25	39		26	38
4	38	26		56	1
5	24	40		27	37
6	19	45		25	0
7	27	37		46	18
8	16	48		32	32
9	28	36		42	22
10	25	39		31	33
11	18	46		47	17
12	19	0		21	0
13	19	45		50	14
14	19	45		31	33
15	41	23		41	23
16	39	25		31	33
17	35	29		55	9
18	22	42		54	10
19	21	43		32	32
20	30	34		29	0
21	29	35		41	23
22	23	41		38	26
23	30	34		31	33
24	26	1		23	0
25	23	0		33	31
26	19	45		36	28
27	32	32		23	41
28	30	34		28	36
29	54	10		21	43
30	30	34		34	30
Media	26,966	32,933		35,5	21,966
Victorias	8	21		18	10

Tabla 4.5: PMCTS y MCTS(140 simulaciones) vs Alpha-beta (débil)

menta sustancialmente la capacidad de Alpha-beta, pues reduce las victorias de PMCTS al 36 % y las de MCTS al 26,6 %.

Por último, con el fin de conseguir vencer con mayor frecuencia a Alpha-beta, vamos a introducir contenido específico del dominio en nuestro jugador con mejor resultado, que ha demostrado ser PMCTS. Utilizaremos la misma heurística que hemos proporcionado a Alpha-beta, para ello, una vez realizado todo el algoritmo, antes de elegir el movimiento a realizar, aumentaremos o disminuiremos el valor de bondad del nodo sumando 20 aciertos si la jugada nos haría dominar una esquina, 2 en caso de una casilla 'X'y restaremos 5 para las casillas 'C'. A la vista de esto obtenemos los resultados que se disponen en la tabla 4.7. Podemos ver una leve mejoría respecto al caso anterior, subiendo el porcentaje de victorias al 43,3 %. Vamos ahora a comprobar el comportamiento al aumentar el número de simulaciones significativamente para ver si con esto podemos definitivamente sobrepasar a Alpha-beta fuerte.

MCTS vs PMCTS

Por último, enfrentamos ambas versiones de Montecarlo Tree Search entre si y mostramos los resultados en la tabla 4.9. Ya habíamos visto en los experimentos anteriores que en general tenemos un rendimiento mejor en la versión plana que en la dinámica, aquí lo vemos claramente plasmado cuando PMCTS se alza con un 76 % de victorias.

Análisis de resultados

Podemos ver que ambas versiones del algoritmo dan muy buenos resultados contra el aleatorio, obteniendo un ratio de no derrotas siempre entre el 90 % y el 97 %. Esto nos indica que realmente el algoritmo toma buenas decisiones. Podemos ver aquí, una leve diferencia entre las dos versiones dando pistas de que PTCMS tiene un comportamiento livianamente mejor que su versión original. Vemos que al aumentar las repeticiones disminuye en una las victorias de MCTS. Podemos achacar esto a la simple probabilidad o, que el salto de simulaciones no ha sido suficiente para asegurar un aumento sustancial del rendimiento.

Una situación similar encontramos contra el algoritmo de búsqueda en profundidad, en este caso, teniendo un margen mayor de mejora, especialmente en MCTS, damos un salto categórico en la cantidad de simulaciones entre el primer experimento y el segundo. Apreciamos claramente cómo ambas versiones mejoran minimizando las opciones del adversario.

Por otra parte, el algoritmo Alpha-Beta obtiene un mayor porcentaje de victorias contra ambos. Esto era de esperar, pues como ya se ha explicado antes, se trata de un algoritmo muy sofisticado. Cabe destacar la importan-

Experimento	Alpha-Beta	PMCTS		Alpha-beta	MCTS
1	44	20		21	43
2	31	33		31	33
3	41	23		43	21
4	25	0		47	17
5	44	20		34	30
6	37	27		38	26
7	42	22		46	18
8	21	43		38	26
9	55	9		27	37
10	28	36		30	34
11	37	27		52	12
12	43	21		46	18
13	20	44		58	5
14	26	38		61	1
15	46	16		56	8
16	20	44		25	0
17	51	13		51	13
18	42	22		30	34
19	59	5		32	32
20	59	5		32	33
21	48	16		35	29
22	31	33		29	35
23	55	9		22	42
24	29	35		52	12
25	31	33		38	26
26	31	33		29	0
27	41	23		49	15
28	53	11		42	22
29	49	15		33	31
30	27	37		36	28
Media	38,86	23,76		38,76	22,7
Victorias	19	11		21	8

Tabla 4.6: PMCTS y MCTS(140 simulaciones) vs Alpha-beta (fuerte)

Experimento	Alpha - Beta	PMCTS
1	25	39
2	27	37
3	41	23
4	25	39
5	39	25
6	24	40
7	31	33
8	43	21
9	46	18
10	21	43
11	61	3
12	43	21
13	44	20
14	55	9
15	39	25
16	26	38
17	60	4
18	26	38
19	44	20
20	10	54
21	49	15
22	31	33
23	44	20
24	54	10
25	51	13
26	56	5
27	29	35
28	40	24
29	28	36
30	20	44
Media	37,733	26,166
Victorias	17	13

Tabla 4.7: PMCTS (mejorado y con 140 simulaciones) vs Alpha-beta (fuerte)

Experimento	Alpha - Beta	PMCTS (informado)		Alpha - Beta	PMCTS
1	38	26		38	26
2	51	13		51	13
3	17	47		17	47
4	30	34		30	34
5	30	34		30	34
6	48	16		48	16
7	45	19		45	19
8	28	36		28	36
9	29	35		29	35
10	30	34		30	34
11	39	0		39	0
12	27	37		27	37
13	38	26		38	26
14	14	50		14	50
15	41	23		41	23
16	31	33		31	33
17	24	40		24	40
18	31	33		31	33
19	52	12		52	12
20	17	47		17	47
21	44	20		44	20
22	31	33		31	33
23	19	45		19	45
24	46	18		46	18
25	32	32		32	32
26	31	33		31	33
27	24	40		24	40
28	30	34		30	34
29	41	23		41	23
30	43	21		43	21
Media	33,36	29,8		36,46	26,06
Victorias	12	17		14	15

Tabla 4.8: PMCTS (750 simulaciones) vs Alpha-beta (fuerte)

Experimento	MCTS	PMCTS
1	24	40
2	28	36
3	38	26
4	17	47
5	16	48
6	23	41
7	28	36
8	19	45
9	17	47
10	19	45
11	27	37
12	3	61
13	19	45
14	37	27
15	23	41
16	34	30
17	12	52
18	40	24
19	29	35
20	16	48
21	17	47
22	49	15
23	22	42
24	30	34
25	35	29
26	24	40
27	3	51
28	28	36
29	44	20
30	22	42
Media	24,76	38,9
Victorias	7	23

Tabla 4.9: MCTS (70 simulaciones) vs PMCTS (70 simulaciones)

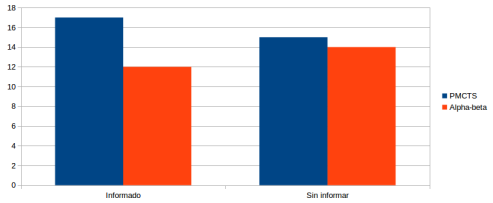


Figura 4.1: Resultados de usar o no conocimiento del dominio con 750 simulaciones

cia de la optimización heurística, pues vemos que el $\alpha - \beta$ saca más ventaja a los algoritmos de Montecarlo cuando está armada con esta. A pesar de todo, conseguimos un resultado bastante bueno contra ambas versiones, especialmente de PMCTS que gana en la mayoría de los casos contra la versión débil y es capaz de obtener más de un tercio de las victorias contra la versión fuerte. Apreciamos también en este apartado cómo el inyectar cierto conocimiento del dominio hace mejorar a PMCTS hasta estar casi igualado con el algoritmo más fuerte que hemos presentado en el caso de realizar 140 simulaciones. Una vez damos el salto a 750 vemos que PMCTS es capaz de dominar en victorias, además, al resultado es incluso mejor al inyectar conocimiento del dominio. Es notable destacar que en estos últimos experimentos se producen más victorias de PMCTS aunque este consigue una puntuación media peor.

Un punto importante a destacar es la diferencia de rendimiento entre la versión plana y la dinámica, ya que en general obtiene un resultado mucho más alto la primera. Una posible razón que explique esta diferencia es la expansión inicial del árbol que realiza PMCTS ya que generamos dos niveles completos con los que no cuenta MCTS que comienza desde la raíz, por lo que con las mismas iteraciones, realmente ha previsto más.

Para acabar esta sección, se muestran diferentes gráficas con la exposición de los resultados. En la gráfica presentada en la figura 4.2 podemos ver el resultado medio obtenido en las distintas ejecuciones, en la parte superior se muestra MCTS y en la inferior PMCTS. En esta última se señala como Alpha-beta(mejorado) el enfrentamiento entre Alpha-beta fuerte y PMCTS con conocimiento del dominio. Del mismo modo , la figura 4.1 muestra los resultados del último experimento que compara el uso de conocimiento del dominio con 750 simulaciones. Por otra parte, vemos que la figura 4.3 nos muestra dos gráficas de dispersión con las distintas partidas en las que podemos ver diferenciados los distintos experimentos y obtener una idea global del funcionamiento. Se muestra la barra divisoria en 32 pues es la puntuación mínima que hay que obtener para no perder.

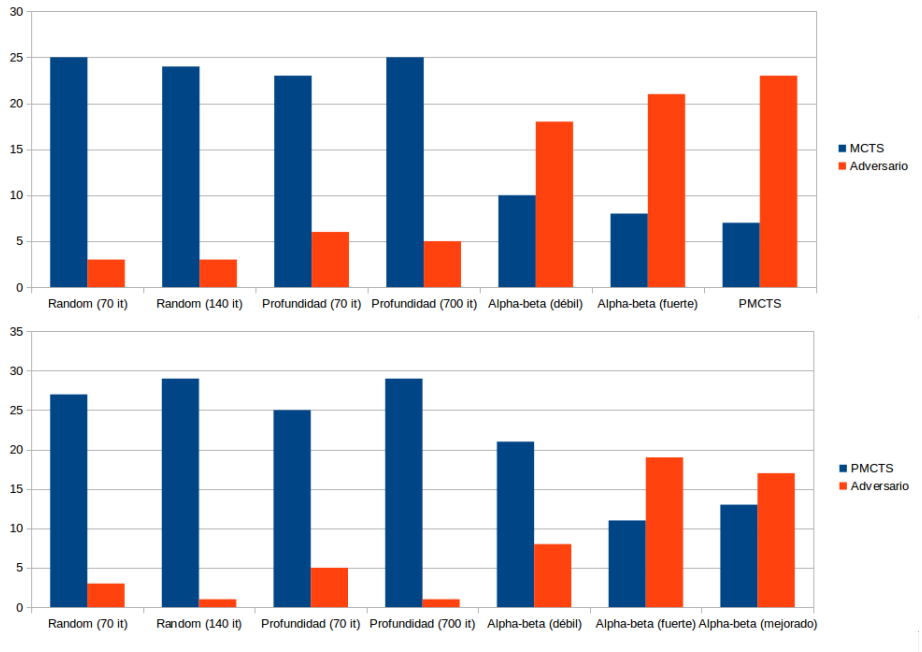


Figura 4.2: Comparación de partidas ganadas y perdidas en los diferentes enfrentamientos

Aplicación al Ajedrez

El Ajedrez de por sí es un juego bastante complicado, como ya hemos comentado anteriormente, tiene un factor de ramificación medio de 35, por ello nos es imposible abarcarlo con algoritmos más clásicos como Minimax puro. Además, a diferencia del Reversi, en el Ajedrez se pueden producir ciclos ya que todas las piezas, a excepción de los peones, pueden volver a la posición de inicio de su anterior movimiento, por lo que intentar investigar el espacio de estados completo no resulta factible. Dadas estas bases, vamos a elegir el tipo de expansión y de simulación que nos interesan.

Expansión

En este caso vamos a probar con la misma versión que utilizamos para el Reversi. Para la versión plana debemos decidir sobre qué árbol vamos a utilizar. Comenzamos generando un árbol de 3 niveles completos (la raíz y dos niveles más), al igual que hacíamos antes.

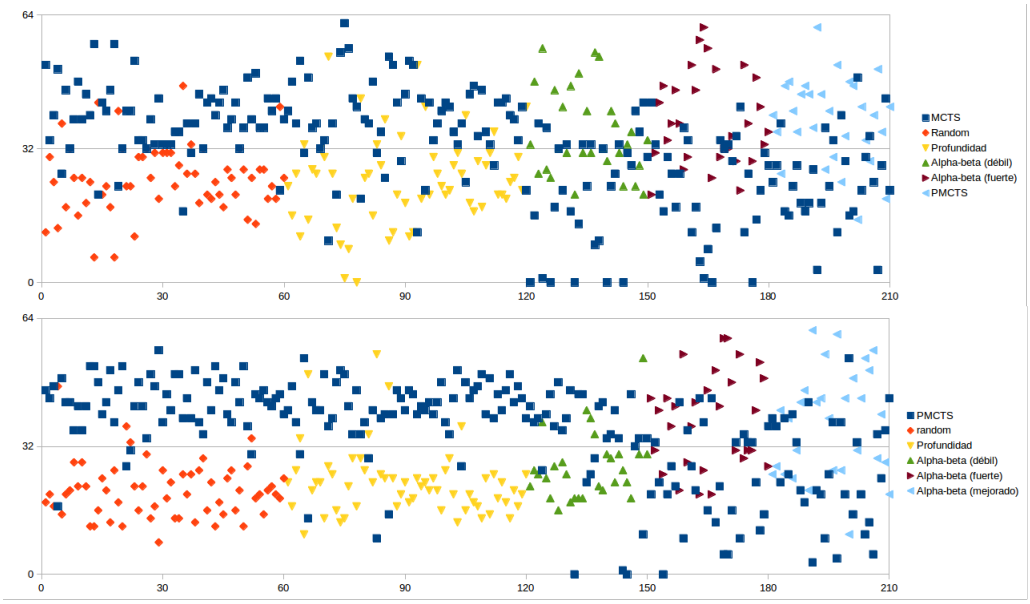


Figura 4.3: Visualización de los resultados de los experimentos

Simulación

Dada la existencia de ciclos, es mala idea intentar llegar al final del juego dando jaque mate para obtener el resultado. Es muy probable que no lleguemos a finalizar la partida, además de que en caso de llegar, el tiempo podría ser arbitrariamente largo por lo que habríamos gastado todo el tiempo computacional en un número muy reducido de simulaciones. Por esto, necesitamos ser capaces de prever un número fijo de jugadas y con ello, saber si habríamos mejorado nuestra situación respecto al adversario o no. Esta idea supone un problema, pues nos obligaría a introducir contenido específico del dominio acercando nuestro algoritmo más a una perspectiva heurística. Presentamos entonces dos alternativas:

- **Contar piezas**
  La idea sería simular n jugadas de forma aleatoria y una vez hechas comparamos si la proporción de piezas con nuestro rival ha aumentado o disminuido. Si de la comprobación resultara probado un aumento de dicho ratio, devolveríamos un valor de éxito que codificaremos como 1, en caso contrario devolveremos 0. En caso de que nos encontremos en una situación de jaque o de jaque mate consideraremos que es suficientemente buena (o mala) así que devolveremos 1 en caso de que el jaque sea a favor y 0 si es en contra. Esta aproximación, aunque incluye información específica, no incurre en un cambio radical del algoritmo,

pues resulta un conocimiento bastante superfluo.

- **Contar piezas con valor**
  En la aproximación anterior no se valora la diferencia estratégica que suponen las piezas ya que no es lo mismo perder un peón que una dama. Por ello vamos a utilizar la medida que se introdujo en la sección sobre estrategias del ajedrez.

  Esta aproximación nos costará prácticamente lo mismo a nivel computacional y promete dar valores más acertados. Por otra parte, conlleva un comportamiento altamente heurístico, pues la valoración de cada nodo necesita un conocimiento muy amplio del Ajedrez en sí mismo.

Para considerar si un nodo es bueno o malo, consideraremos la diferencia de piezas que tenemos con el adversario (con o sin ponderación). El resto del algoritmo funcionará de igual manera.

Resultados

A la hora de experimentar con el Ajedrez vemos el problema de que, en general, los jugadores automáticos que hemos probado rara vez llegan a dar mate, ya que eliminan a todas las piezas y terminan quedando únicamente los reyes vagando por el tablero, lo que equivaldría a una situación de tablas. Por tanto, en las tablas siguientes mostramos las piezas que han quedado cuando se han eliminado todas las piezas a excepción del rey de algún jugador. Se marcará también aquellas casillas en las que sí llega a jaque mate, así como las piezas que le han quedado a cada jugador al término del juego.

Podemos ver ilustrados los resultados de diversas ejecuciones en confrontación con el algoritmo *Random* definido de la misma manera que en el caso del Reversi. La tabla 4.10 nos muestra el ejemplo de 10 partidas utilizando un jugador basado en el algoritmo de Montecarlo dinámico. Por otra parte, la tabla 4.11 nos muestra 10 partidas contra la versión dinámica. Para acabar, la tabla 4.12 nos enseña 10 partidas entre PMCTS y MCTS. Todos los experimentos se han realizado con 400 simulaciones por turno y cada una de ella indaga en 15 movimientos calculando el valor de utilidad del nodo resultante basándonos en el la diferencia de piezas que se tiene.

Análisis de resultados

Podemos ver que no hemos conseguido un jugador tan eficaz como para el Reversi, incluso contra un algoritmo simple como *Random*, no consiguen dar jaque mate en la gran mayoría de situaciones. Aunque es cierto que suelen priorizar comer piezas por lo que en la gran mayoría de los casos consiguen dominar el número de piezas a su favor.

	Ganador	♙	♖	♘	♗	♕
Blancas		0	0	0	0	0
Negras	×	2	0	0	1	0
Blancas		0	0	0	0	0
Negras	×	1	0	0	0	0
Blancas		0	0	0	0	0
Negras	×	1	0	1	1	0
Blancas		0	0	0	0	0
Negras	×	0	0	1	0	0
Blancas		0	0	0	0	0
Negras	×	2	1	0	0	0
Blancas		5	0	2	1	0
Negras	Jaque mate	4	1	0	0	0
Blancas	Tablas	0	0	0	0	0
Negras	Tablas	0	0	0	0	0
Blancas		0	0	0	0	0
Negras	×	3	0	0	0	0
Blancas		0	0	0	0	0
Negras	×	2	0	1	0	0
Blancas	Jaque mate	4	1	1	2	0
Negras		1	0	0	0	0

Tabla 4.10: Random (Blancas) vs MCTS (Negras)

	Ganador	♙	♖	♘	♗	♕
Blancas	×	4	0	0	0	0
Negras		0	0	0	0	0
Blancas		0	0	0	0	0
Negras	×	0	2	1	0	0
Blancas		0	0	0	0	0
Negras	×	1	1	0	1	0
Blancas		0	0	0	0	0
Negras	×	6	0	2	0	0
Blancas		0	0	0	0	0
Negras	×	1	0	1	1	0
Blancas		0	0	0	0	0
Negras	×	1	2	1	1	1
Blancas		0	0	0	0	0
Negras	×	2	0	0	1	1
Blancas	Jaque mate	5	1	1	2	1
Negras		1	1	1	1	1
Blancas		3	1	0	0	0
Negras	Jaque mate	8	1	1	1	1
Blancas		0	0	0	0	0
Negras	×	5	0	0	0	0

Tabla 4.11: Random (Blancas) vs PMCTS (Negras)

	Ganador	♙	♖	♘	♗	♕
Blancas	×	1	0	0	0	0
Negras		0	0	0	0	0
Blancas		0	0	0	0	0
Negras	×	5	2	2	1	0
Blancas	Jaque mate	3	0	1	0	1
Negras		4	0	2	0	0
Blancas		3	0	0	0	0
Negras	Jaque mate	1	1	0	0	1
Blancas	×	2	0	0	1	0
Negras		0	0	0	0	0
Blancas		0	0	0	0	0
Negras	×	1	0	0	0	0
Blancas		0	0	0	0	0
Negras	×	4	1	1	0	0
Blancas	×	2	0	0	1	0
Negras		0	0	0	0	0
Blancas		1	0	0	1	0
Negras	Jaque mate	3	1	0	0	1
Blancas		0	0	0	0	0
Negras	×	3	1	0	1	0

Tabla 4.12: MCTS(Blancas) vs PMCTS (Negras)

Es importante destacar que aquí los experimentos funcionan mucho más despacio, la fase de simulación se siente claramente ralentizada en ambas versiones del algoritmo. Esto era de esperar pues la cantidad de movimientos posibles es muy superior. También se ve ralentizado por el cálculo de posibles acciones ya que estas necesitan calcular todos los movimientos que pueden realizar las 16 piezas (o menos si hemos ido perdiendo), y además comprobar si alguna de estas dejaría a nuestro rey en jaque, pues esto incurriría en un movimiento ilegal.

Otro punto a destacar es que son necesarias muchas más simulaciones para estimar un valor heurístico adecuado, un ejemplo que lo visualiza claramente es que en la primera jugada PMCTS calcula un árbol de juego que tiene 9 nietos en el caso del Reversi y 156 para el Ajedrez, de esta forma, es completamente inútil intentar un nivel de simulaciones parecido al que obteníamos allí, ya que con 70 simulaciones ni siquiera la mitad de los nodos habrían podido realizar una ejecución, lo que se traduce en una heurística completamente inservible.

Capítulo 5

Mejorando MCTS con redes neuronales

En este capítulo vamos a hacer un recorrido sobre los métodos más modernos implementados con el fin de perfeccionar el rendimiento de Montecarlo Tree Search. Lo dividiremos en dos secciones, en la primera expondremos los temas y conceptos pertinentes para comprender qué es el aprendizaje automático o *Machine Learning*, y en el segundo cómo lo mezclamos con MCTS para mejorar los resultados. En este tema se utilizarán diversos conceptos sobre probabilidad y álgebra que no han sido introducidos anteriormente en el documento, estos se pueden consultar íntegramente en los apéndices A y B respectivamente.

Aprendizaje automático

El campo del aprendizaje automático surge de cuando intentamos que un computador sea capaz de reconocer patrones y obtener predicciones a partir de un conjunto de datos. Destacamos dos casos principales a estudiar: regresión y clasificación.

El problema de regresión intenta afrontar el dilema de tener dos variables aleatorias correlacionadas X e Y y necesitamos determinar una aproximación a la relación existente entre ellas. Los datos nos generarán una nube de puntos y buscamos encontrar una función $y(x)$ que sea la que mejor se ajuste a las observaciones (Rouaud, 2013). La aproximación más típica es conocida como regresión lineal, que fuerza a y a definir un hiperplano del espacio, por lo que esta viene conformada de la forma:

$$y(x) = W^t x + b$$

con la matriz $W \in \mathbb{R}^{D\times K}$ y los vectores $x \in \mathbb{R}^D$ y $b \in \mathbb{R}^K$ siendo D la

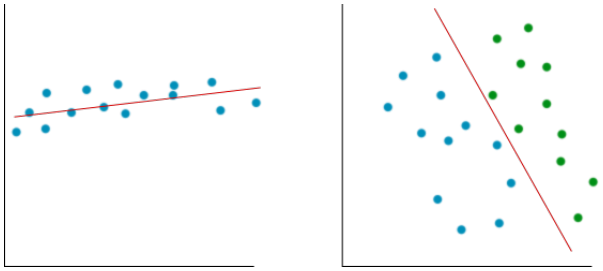


Figura 5.1: Regresión (Izquierda) y clasificación (derecha)

dimensión del espacio de los datos y K la dimensión del resultado a obtener. En la figura 5.1 se muestra un ejemplo de regresión lineal con $D = K = 1$.

Por su parte, el problema de clasificación parte de un conjunto de datos distribuidos en K clases. Nos gustaría, dado un dato de entrada, asignarlo a una clase $\mathcal{C}_k$ con $1 \leq k \leq K$. El problema que probablemente más se estudia es el conocido como clasificación binaria, que se trata del caso en el que $k = 2$ (Smola y Vishwanathan, 2008). Un ejemplo clásico son los filtros de spam, que, dado el contenido de cierto e-mail, son capaces de diferenciar si estos se tratan de correo basura o no.

En caso de que $K > 2$ conoceremos la situación como clasificación multiclase. Encontramos igualmente infinidad de ejemplos como detectar de forma automática el idioma en el que se encuentra un texto o, dadas las mediciones de los síntomas, diagnosticar la etapa del cáncer en la que se encuentra una persona, dando herramientas poderosas a los profesionales sanitarios. Resultados erróneos pueden acarrear severas consecuencias, como en el caso de diagnosticar erróneamente una enfermedad, por ello deben tener unas bases matemáticas bien fundadas para asegurar la exactitud de los resultados.

Vamos a presentar 3 algoritmos que dan solución a estos problemas, de menor a mayor complejidad: mínimos cuadrados, el perceptrón y las redes neuronales. Aunque la cantidad de algoritmos desarrollada es extensa, estos son una buena aproximación al campo de estudio, y cada uno representa una evolución del anterior. Nos aportarán herramientas poderosas para mejorar MCTS.

Mínimos cuadrados

Partimos de un conjunto de entrenamiento, compuesto por N vectores[1] de entrada y la correspondiente etiqueta que nos indicará a qué clase pertenecen, es decir $\{(x_j, t_j)\}_{j=1}^{N}$. Utilizaremos una clasificación para las etiquetas

[1] en desarrollo todos los vectores son columnas, para indicar un vector fila se indicará como traspuesta

conocida como *one hot*, en ella para todo j, t_j será un vector e_k de la base canónica[2] de $\mathbb{R}^K$.

Deseamos encontrar una función afín $y : \mathbb{R}^D \to \mathbb{R}^K$, con lo que obtendremos un vector de dimensión K. Al ser afín sabemos que nuestra función tendrá la forma: $y(x) = W^t x + b$ en las mismas condiciones que se define en la sección anterior. Asignaremos a la clase de la coordenada que haya obtenido un valor mayor, es decir:

$$y \in \mathcal{C}_k | k = \arg \max_{1 \leq \alpha \leq K} y_\alpha(x)$$

Para simplificar la notación en el desarrollo llamaremos $\tilde{x} = \begin{pmatrix} 1 \\ x \end{pmatrix}$ y $\tilde{W} = \begin{pmatrix} b^t \\ W \end{pmatrix}$. Agrupando ahora etiquetas y ejemplos de entrenamiento, llamaremos $T = (t_1, ..., t_N)$ y $\tilde{X} = (\tilde{x}_1, ..., \tilde{x}_n)$. En un caso ideal, querríamos conseguir que:

$$\forall j \; y(x_j) = W^t x + b = \tilde{W}^t \tilde{x} = t_j$$

o equivalentemente debemos resolver el sistema lineal de ecuaciones:

$$\tilde{W}^t \tilde{X} = T$$

Sin embargo en la mayoría de situaciones este sistema no tendrá solución por lo que deberemos conformarnos con el resultado que se acerque más, para ello resolveremos el problema de optimización:

$$\min_{\tilde{W}} ||\tilde{W}^t \tilde{X} - T||$$

Llamaremos $J(\tilde{W}) = ||\tilde{W}^t \tilde{X} - T||$, función de coste. Dado que la norma es no negativa, minimizar J es equivalente a minimizar $E(\tilde{W}) = \frac{1}{2}||\tilde{W}^t \tilde{X} - T||^2$, esto nos facilitará los cálculos más adelante. Presentamos a continuación 2 formas de solucionarlo, una de ellas es un método de aproximación y la otra una solución teórica.

Descenso de gradiente

El descenso de gradiente es un método iterativo que da solución a minimizar la función $E(\tilde{W})$. Para ello, calcularemos el gradiente de la función,

[2]En el espacio vectorial $\mathbb{K}^n$ el conjunto formado por los n vectores $(1, 0, 0..., 0), (0, 1, 0, ..., 0), ..., (0, 0, 0, ..., 1)$ es una base que recibe el nombre de base canónica de $\mathbb{K}^n$. Estos vectores habitualmente se denominan e_i donde $(e_i)_j = 1$ si $j = i$ y $(e_i)_j = 0$ en cualquier otro caso (Merino y Santos, 2006).

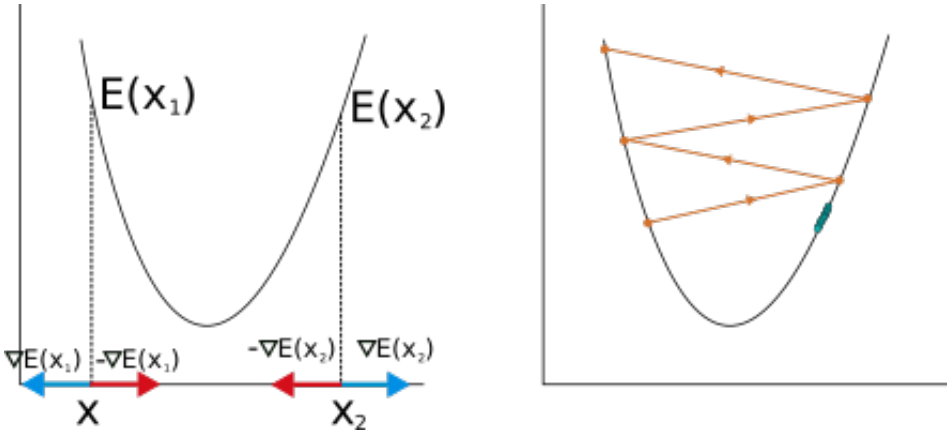


Figura 5.2: Dirección del gradiente y problemas de elegir erróneamente el parámetro α

este nos indica la dirección de ascenso de máxima pendiente, por ello, si restamos esta cantidad, estaremos decreciendo el valor de la función de coste. Tenemos:

$$\nabla E(\tilde{W}) = \tilde{X}(\tilde{W}^t\tilde{X} - T)^t$$

Este resultado se justifica en la sección siguiente. Teniendo esto, definimos la regla de actualización como:

$$\tilde{W}^{(r+1)} = \tilde{W}^{(r)} - \alpha\tilde{X}(\tilde{W}^t\tilde{X} - T)^t$$

Donde α es un parámetro conocido como *tasa de aprendizaje*. La función de este es determinar el tamaño del salto existente entre una actualización y la siguiente. Un valor muy grande puede llevar a que el método obtenga incluso un resultado opuesto al buscado. Por otra parte, un valor muy pequeño puede llevar a que el algoritmo no converja en un tiempo razonable. Se muestra una visualización de estos resultados en la figura 5.2 en la que se puede ver a la izquierda los gradientes en los distintos puntos y a la derecha el resultado de usar un α muy grande (en naranja) y el de utilizar uno muy pequeño (en azul).

Cálculo directo

Vamos a calcular el mínimo de la función $E(\tilde{W})$ de manera explícita. Comenzamos observando que:

$$E(\tilde{W}) = \frac{1}{2}||\tilde{W}^t\tilde{X} - T||^2 = \frac{1}{2}tr((\tilde{W}^t\tilde{X} - T)^t(\tilde{W}^t\tilde{X} - T))\text{??} \tag{5.1}$$

$$= \frac{1}{2}tr(\tilde{X}^t\tilde{W}\tilde{W}^t\tilde{X} - \tilde{X}^t\tilde{W}T - T^t\tilde{W}^t\tilde{X} + T^tT) \tag{5.2}$$

Nos encontramos ante una función diferenciable, podemos entonces encontrar los extremos en los puntos que anulen a $\nabla E(\tilde{W})$. Por tanto, dada una matriz cualquiera $A \in \mathbb{R}^{K\times(D+1)}$:

$$\nabla E(\tilde{W}) = \lim_{\varepsilon\to 0} \frac{E(\tilde{W} + \varepsilon A) - E(\tilde{W})}{\varepsilon}$$

substituyendo en 5.2:

$$\begin{aligned} E(\tilde{W} + \varepsilon A) = \frac{1}{2}tr(&\varepsilon^2\tilde{X}^tAA^t\tilde{X} - \varepsilon\tilde{X}^tAT - \varepsilon T^tA^T\tilde{X} \\ &+ \varepsilon\tilde{X}^t\tilde{W}A^t\tilde{X} + \varepsilon\tilde{X}^tA\tilde{W}^t\tilde{X} + \tilde{X}^t\tilde{W}\tilde{W}^t\tilde{X} \\ &- \tilde{X}^t\tilde{W}T - T^t\tilde{W}^t\tilde{X} + T^tT) \end{aligned}$$

si restamos $E(\tilde{W})$ obtenemos que:

$$\begin{aligned} E(\tilde{W} + \varepsilon A) - E(\tilde{W}) = \frac{1}{2}tr(&\varepsilon^2\tilde{X}^tAA^t\tilde{X} - \varepsilon\tilde{X}^tAT - \varepsilon T^tA^T\tilde{X} \\ &+ \varepsilon\tilde{X}^t\tilde{W}A^t\tilde{X} + \varepsilon\tilde{X}^tA\tilde{W}^t\tilde{X}) \end{aligned}$$

dividiendo por ε y teniendo en cuenta la propiedad de la traza 1:

$$\begin{aligned} \frac{E(\tilde{W} + \varepsilon A) - E(\tilde{W})}{\varepsilon} = \frac{1}{2}tr(&\varepsilon\tilde{X}^tAA^t\tilde{X} - \tilde{X}^tAT - T^tA^T\tilde{X} \\ &+ \tilde{X}^t\tilde{W}A^t\tilde{X} + \tilde{X}^tA\tilde{W}^t\tilde{X}) \end{aligned}$$

añadiendo el límite y considerando las propiedades de la traza 3 y 4:

$$\begin{aligned}\lim_{\varepsilon\to 0}\frac{E(\tilde{W}+\varepsilon A)-E(\tilde{W})}{\varepsilon} &= \frac{1}{2}tr(\tilde{X}^tAT - T^tA^T\tilde{X}\\ &+ \tilde{X}^t\tilde{W}A^t\tilde{X} + \tilde{X}^tA\tilde{W}^t\tilde{X})\\ &= \frac{1}{2}[tr((\tilde{X}^tA)(\tilde{W}^t\tilde{X}-T))\\ &+ tr(((\tilde{X}^tA)(\tilde{W}^t\tilde{X}-T))^t)]\\ &= tr((\tilde{X}^tA)(\tilde{W}^t\tilde{X}-T))\\ &= tr((\tilde{W}^t\tilde{X}-T)\tilde{X}^tA)\\ &= tr((\tilde{X}(\tilde{W}^t\tilde{X}-T)^t)^tA)\\ &= \left\langle \tilde{X}(\tilde{W}^t\tilde{X}-T)^t, A\right\rangle\end{aligned}$$

Dado que esto debe anularse $\forall A$ entonces debe ser que:

$$\begin{aligned}\tilde{X}(\tilde{W}^t\tilde{X}-T)^t &= 0\\ \tilde{X}(\tilde{X}^t\tilde{W}-T^t) &= 0\\ \tilde{X}\tilde{X}^t\tilde{W}-\tilde{X}T^t &= 0\\ \tilde{X}\tilde{X}^t\tilde{W} &= \tilde{X}T^t\\ \tilde{W} &= (\tilde{X}\tilde{X}^t)^{-t}\tilde{X}T^t\end{aligned}$$

Este método da una solución instantánea y por tanto es preferible su uso al del descenso de gradiente explicado en la sección anterior, sin embargo, no en todos los algoritmos nos va a ser posible calcular explícitamente el mínimo.

Perceptrón simple

El siguiente algoritmo a estudiar es conocido como perceptrón, el punto más importante de este es que define el elemento básico con el que construiremos nuestra red neuronal: la neurona.

Vamos a obtener un algoritmo que sea capaz de realizar una clasificación binaria. Además, vamos a utilizar una codificación distinta a la de la sección anterior para las etiquetas, en este caso $t_n \in \{1, -1\}$. Definido de esta forma, nos gustaría que nuestra función $y(x)$ devolviera 1 en caso de que el dato x pertenezca a la clase $\mathcal{C}_1$ y -1 en caso de que corresponda a $\mathcal{C}_2$ de esta forma definimos (Bishop, 2006):

$$y(x) = f(w^tx+b) = f(\tilde{w}^t\tilde{x}) = \begin{cases} 1 & si \quad w^tx+b \geq 0\\ -1 & si \quad w^tx+b < 0\end{cases}$$

teniendo $w \in \mathbb{R}^D$, $b \in \mathbb{R}$ y, con la misma abreviatura que hemos introducido antes $\tilde{x}, \tilde{w} \in \mathbb{R}^{D+1}$, siendo D la dimensión de los datos de entrada.

A f la llamaremos función de activación, su función es llevar los valores al dominio que nos interesa, en este caso ± 1. Para este algoritmo la hemos definido como una función de salto, cuando veamos redes neuronales, necesitaremos alguna función más potente que esta para obtener un resultado acertado.

Nos interesa obtener un $\tilde{w}$ que consiga $\tilde{w}^t \tilde{x_n} > 0$ si el dato x_n pertenece a la primera clase y $\tilde{w}^t \tilde{x_n} < 0$ en caso de que lo haga a la segunda. Podemos resumir esto en $\tilde{w}^t \tilde{x_n} t_n > 0$. Tenemos ya una forma de definir nuestra función de coste para cierto dato:

$$E(\tilde{w}) = \begin{cases} 0 & \text{Si el dato está bien clasificado} \\ -\tilde{w}^t \tilde{x_n} t_n & \text{En caso contrario} \end{cases}$$

la función de coste total será la suma de la función de coste asociada a cada dato de entrenamiento:

$$E(\tilde{w}) = \sum_{i=1}^{N} E_i(\tilde{w})$$

Para entrenarlo utilizaremos una versión modificada del entrenamiento expuesto para mínimos cuadrados: el descenso de gradiente estocástico. Este difiere de la versión original en que en lugar de utilizar la función de coste total actualizaremos mediante la función particular de cada dato. Es conocido como estocástico ya que en su implementación antes de comenzar a repasar los datos uno a uno y realizar el descenso, se suelen reordenar los datos de entrada de forma aleatoria. Dicho esto, basta ahora con encontrar el gradiente de la función de coste que viene definido trivialmente por $\nabla E_n(\tilde{w}) = -x_n t_n$ la regla de actualización queda entonces:

$$\tilde{w}^{(r+1)} = \begin{cases} \tilde{w}^{(r)} & \text{Si el dato está bien clasificado} \\ \tilde{w}^{(r)} + x_n t_n & \text{En caso contrario} \end{cases}$$

Aunque este algoritmo está pensado para clasificación binaria, es fácilmente generalizable al caso multiclase, basta con utilizar tantos perceptrones como clases existan y entrenar cada uno para diferenciar cierta clase $\mathcal{C}_k$ de el resto. En el caso ideal se obtendría en todos -1 excepto en 1, aunque suele ocurrir que más de uno de resultado positivo, en este caso haría falta comparar para cuál de ellos el resultado de $\tilde{w}^t \tilde{x_n}$ es mayor.

Redes neuronales

Una red neuronal, también conocida como perceptrón multicapa, generaliza el concepto presentado por el perceptrón. La idea es utilizar muchas

neuronas que compartan la información entre ellas y sean capaz de realizar las tareas de clasificación y regresión con mayor eficacia.

Dicho esto, definimos una red neuronal como una terna (N, V, w) siendo N un conjunto de neuronas y V un conjunto $\{(i,j)|i,j \in \mathbb{N}\}$ cuyos elementos son llamados conexiones entre la neurona i y la neurona j. La función $w : V \to \mathbb{R}$ define los pesos, donde $w((i,j))$, representa el peso de la conexión entre la neurona i y la j que abreviaremos como $w_{i,j}$ (Kriesel, 2005).

Ahora, queremos dar un poco más de estructura a esta definición, por ello vamos a incluir el concepto de capa. Redefinimos entonces la definición como una terna (N, V, w) solo que ahora $N = \{C_1, C_2...C_n\}$ es un conjunto de capas, cada uno de estos es entonces un conjunto de neuronas. Vamos a modificar también la definición de V para que únicamente se permitan conexiones entre una capa y la siguiente, de esta forma $V = \{(i,j,k)|i,j,k \in \mathbb{N}\}$ representa la conexión entre las neuronas i de la capa C_{k-1} y j de C_k. Por ende, el peso de esta conexión vendrá dado por $w(i,j,k)$ que abreviaremos como $w^k_{i,j}$, también utilizaremos la notación W^k para referenciar la matriz cuyos elementos son $W^k_{i,j} = w^k_{i,j}$. Obtenemos así el concepto de red neuronal multicapa.

Nuestra red define una composición de aplicaciones pues el resultado obtenido por la función y será la consecuencia de atravesar todas las capas de la red. Llamaremos $z^k : \mathbb{R}^{K_{k-1}} \to \mathbb{R}^{K_k}$ a la función cuyos datos de entrada son los resultados obtenidos en la capa anterior y genera los nuevos datos. La dimensión del espacio de entrada es $\mathbb{R}^{K_{k-1}}$ será igual al número de neuronas de la capa C_{k-1}, lo mismo ocurre con la capa siguiente. A su vez, las funciones z^k son una composición de dos funciones:

$$z^k = h^k \circ a^k$$

aquí, a^k es una función afín similar a las que hemos visto en los algoritmos anteriores:

$$a^k(x) = W^k x + b^k$$

Por su parte, la función h^k hará el papel de la función f en el algoritmo del perceptrón, pero vamos a buscar que esta sea una función continua. Una elección común suele ser la función sigmoide σ:

$$\sigma(a) = \frac{1}{1 + e^{-a}}$$

que tiene valor $0,5$ si $a = 0$, tiende rápidamente a 1 cuando $a \to \infty$, y a 0 cuando $a \to -\infty$. Aunque otras opciones pueden ser la tangente hiperbólica o la función softmax:

$$softmax(a) = \left(\frac{\exp(a_1)}{\sum_{k=1}^{K} \exp(a_i)}, ..., \frac{\exp(a_K)}{\sum_{k=1}^{K} \exp(a_i)} \right)$$

Resumiendo:

$$y(x) = z^k \circ ... \circ z^1(x) \tag{5.3}$$
$$= h^k \circ a^k \circ ... \circ h^1 \circ a^1(x) \tag{5.4}$$

Ya tenemos perfectamente delimitada nuestra herramienta, ahora queda en nuestras manos definir el número de capas, las neuronas que habrá en cada capa, y la función de coste para asegurarnos de que efectivamente nos acercamos a nuestro objetivo. Por convenio se llama capa de entrada a C_1 y C_K se denomina capa de salida, las intermedias se conocen como ocultas. Veremos en un momento que las redes con una capa oculta nos servirán para aproximar cualquier función, aunque dependiendo del dominio el aumento del número de capas puede llevar a un mejor comportamiento de la red.

El número de neuronas en las capas de entrada y salida será igual a la dimensión de los espacios en los que se encuentran los datos de entrada y de salida, aunque este último dependerá del tipo de problema que abarquemos. El número de neuronas en las capas ocultas será decisión nuestra, pueden dar resultados distintos dependiendo de este número por lo que en la mayoría de las veces la determinación viene dada por la vía experimental.

El único elemento que nos falta ahora es la función de coste. Presentamos tres posibilidades, en caso de que utilicemos la red neuronal para dar respuesta a un problema de regresión buscaremos minimizar la disparidad con los datos de entrenamiento por lo que usaremos (Bishop, 2006):

$$E(w) = \frac{1}{2} \sum_n ||y(x_n) - t_n||^2$$

Para la clasificación binaria se suele utilizar:

$$E(w) = -\sum_n t_n \ln y_n + (1 - t_n) \ln(1 - y_n)$$

por último, para la clasificación multiclase se utiliza:

$$E(w) = -\sum_n \sum_k t_{nk} \log y_{nk}$$

Entrenando las redes neuronales

Entrenaremos la redes neuronales mediante un descenso de gradiente, dada la gran cantidad de datos que estas suelen manejar, este suele ser estocástico. Nos encontramos ahora en el problema de encontrar el gradiente en una función definida como en 5.4 lo que resulta no solo complicado teóricamente sino que también muy costoso a nivel computacional, sin embargo,

vamos a aprovecharnos de la naturaleza composicional de la red para facilitar el cálculo, este método se conoce como el algoritmo de retropropagación.

Dada la naturaleza de E, utilizando la regla de la cadena:

$$\frac{\partial E}{\partial w_{i,j}^k} = \sum_{\beta=1}^{L} \sum_{\alpha=1}^{K_k} \frac{\partial E}{\partial a_\alpha^\beta} \frac{\partial a_\alpha^\beta}{\partial w_{ij}^k}$$

ahora:

$$\frac{\partial a_\alpha^\beta}{\partial w_{ij}^k} = \frac{\partial w_\alpha^\beta z^{\beta-1} + b^\beta}{\partial w_{ij}^k}$$

viendo esto, tenemos que $\frac{\partial a_\alpha^\beta}{\partial w_{ij}^k}$ se anula en todos los casos excepto para el caso en que $\beta = k$ y $\alpha = i$, en este caso:

$$\frac{\partial a_\alpha^\beta}{\partial w_{ij}^k} = z_j^{k-1}$$

esto lo podemos traducir a que:

$$\frac{\partial E}{\partial w_{i,j}^k} = \frac{\partial E}{\partial a_i^k} z_j^{k-1}$$

De igual manera, podemos comprobar que:

$$\frac{\partial E}{\partial b_i^k} = \frac{\partial E}{\partial a_i^k}$$

La única tarea que nos queda es hallar un método para calcular $\frac{\partial E}{\partial a_i^k}$ a los que denotaremos δ_i^k y llamaremos errores. Usando una vez más la regla de la cadena podemos ver que:

$$\delta^k = diag(h^k(a^k))W^{k+1}\delta^{k+1}$$

Lo que nos va a facilitar mucho la vida pues únicamente necesitamos el error siguiente. Comenzaremos entonces calculando el último de los errores de la siguiente forma:

$$\delta^L = y(x) - t$$

Ahora que tenemos todas las piezas, nos falta juntarlas y definir el algoritmo(Gurney, 2005):

1. Presentar los datos en la capa de entrada.

2. Evaluar los datos en las capas ocultas.

3. Evaluar el resultado obtenido en la capa de salida.
4. Calcular los errores δ^L
5. Retropropagar el error calculando todos los δ^k
6. Calcular el gradiente
7. Actualizar los pesos siguiendo la regla de actualización.
8. En caso de que no se hayan recorrido todos los ejemplos de entrenamiento volver al paso 1.
9. En caso de que no se hayan completado todas las *epoch* volver al paso 1 y comenzar a revisar el conjunto de entrenamiento desde el principio.

Llamamos *epoch* a cada iteración realizada sobre el conjunto de entrenamiento completo. Cuanto mayor sea el número de epochs que definamos para el algoritmo, más se acercará a la convergencia.

Por último en esta sección comentamos el llamado teorema de aproximación universal, que nos indica que una red neuronal multicapa con una sola capa oculta es capaz de aproximar cualquier función continua con la precisión que queramos (Hornik et al., 1989). Este es un resultado muy potente pues en caso de que desconozcamos una función, sabemos que existe una red neuronal que nos de un resultado muy parecido al real, siempre y cuando tengamos suficientes datos de entrenamiento. Sin embargo, no siempre es posible obtenerla pues el método de descenso de gradiente puede detenerse en mínimos locales.

Mejorando MCTS

Hemos obtenido poderosas herramientas que se utilizan en los campos más punteros de la inteligencia artificial. Vamos ahora a ver cómo se relacionan estos con MCTS y como nos ayuda a mejorar nuestros jugadores sin ayuda humana, en particular, vamos a comentar uno de los ejemplos más actuales: el conocido Alpha Go.

Alpha Go es la primera inteligencia artificial que ha sido capaz de ganar a un jugador profesional de Go consiguiendo 5 victorias a 0 contra el tres veces campeón de Go de Europa, Fan Hui, y posteriormente ganando a Lee Sedon, conocido como el mejor jugador de la década, 4 juegos a 1. Aunque el objetivo de este documento no es indagar en el juego del Go, AlphaGo define un algoritmo fácilmente adaptable a otros juegos basado en la combinación de redes neuronales con Monte Carlo Tree Search.

Tal como se expone en Silver et al. (2017c) y Silver et al. (2017a), para entrenar a AlphaGo se inicializa una red neuronal con pesos W aleatorios. Dado que la intención es obtener un algoritmo que sea capaz de aprender por sí solo sin necesidad de conocimiento humano, para generar los datos de entrenamiento se utiliza el algoritmo de MCTS modificado de forma que en su etapa de selección el valor de UCB viene dado no solo por las simulaciones ya realizadas en este, sino también por la estimación que proporcina la red neuronal, de esta forma, siendo $Q(s, a)$ la recompensa estimada por MCTS al tomar la acción a en el estado s, $N(s, a)$ el número de veces que se ha seleccionado la acción a desde el estado s para simular y $P(s, a)$ la estimación previa que genera la red neuronal en el estado actual. Ahora, podemos calcular el UCB de una decisión como:

$$U(s, a) = Q(s, a) + c_{puct} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}$$

Utilizar esta selección mejora el rendimiento de MCTS pues no parte de 0 considerando todas las acciones igual de provechosas o igual de probables, ya que cuando la red se va entrenando la estimación inicial es cada vez más certera.

Con esta nueva forma de selección, podemos realizar los pasos típicos de MCTS generando las expansiones cuando sea necesario, simulando y retropropagando los resultados. Después de las iteraciones pertinentes, habremos obtenido un valor heurístico del nodo inicial mejor que el que se había estimado mediante la red neuronal, este valor, pasa a ser un dato de entrenamiento de la red, lo que la hace mejorar y vuelta a empezar. Una vez obtenida una red neuronal actualizada, realizamos un determinado número de partidas en las que los jugadores utilizan la nueva y la antigua red neuronal, en caso de que la nueva supere a la antigua en un porcentaje superior al 55 % de las veces, actualizamos la red a la nueva y nos deshacemos de la antigua.

En Nair (2017), podemos ver un ejemplo de aplicación de este algoritmo a una versión reducida del Reversi (tablero de 6x6) y en Silver et al. (2017b) podemos ver la implementación para el ajedrez dando muy buenos resultados, superando a *stockfish* que hasta ahora había obtenido los mejores resultados en este dominio.

En comparación con nuestra implementación, que en el Reversi obtenía buenos resultados limitando las simulaciones a 70 o 140, y en el ajedrez hemos tenido que aumentar, Alpha Go utiliza en cada turno 1600 simulaciones. De acuerdo con (Silver et al., 2017c) esto corresponde con 0,4s por turno, para obtener esta velocidad, el algoritmo es ejecutado utilizando procesadores gráficos lanzando diversas simulaciones a la vez, además los sistemas de *Deep mind* empresa desarrolladora del programa, cuenta con 4 TPU's (Tensor Processor Unit), tecnología puntera desarrollada específicamente para correr este tipo de algoritmos.

Capítulo 6

Conclusiones y Trabajo Futuro

Conclusiones generales

Dada la imposibilidad de realizar una exploración entera del árbol de juego, hemos presentado dos métodos prácticos para dar solución a este problema: la abarcada por Minimax utilizando conocimiento del dominio, y la propuesta por MCTS mediante simulación de partidas. En ambos casos intentamos calcular una heurística para cada nodo que nos indique cuán bueno es. Hemos visto que una aproximación mixta que mezcle conocimiento experto y simulaciones da también resultados muy buenos consiguiendo incluso superar a algoritmos como Alpha-beta mejorado con una heurística.

Se ha demostrado en este trabajo que el algoritmo del árbol de búsqueda de MonteCarlo funciona de manera muy solvente en los entornos en los que se ha probado, dando buenos resultados tanto contra algoritmos sencillos y con algoritmos más complejos como alpha-beta. La independencia del contexto de MCTS permite adaptarlo fácilmente a cualquier juego, consiguiendo un jugador bastante resolutivo en muy poco tiempo y sin necesidad de indagar en el conocimiento propio del juego.

Se ha mostrado también los problemas principales que aparecen al aplicarlo. En particular, apreciamos una complejidad importante al intentar aplicarlo a juegos que contienen ciclos, o que no tienen una longitud definida. Se ha expuesto una solución práctica para solucionar este conflicto con el coste de prescindir de la independencia del contexto.

Hemos visto también que el mayor gasto de recursos computacionales se produce en la fase de simulación y es en ella en la que se tiene que hacer un mayor esfuerzo de optimización. Se ha visto también que un mayor número de simulaciones da una mayor fiabilidad al valor heurístico calculado para cada nodo.

Revisión de objetivos

Revisamos punto por punto los objetivos en el mismo orden en el que fueron propuestos en la introducción:

1. Una vez leído este documento, el lector, sin conocimientos previos de la materia, tiene conciencia de teoría de juegos, estructuras de datos y resoluciones varias aportadas por diversos investigadores al problema de explorar el árbol de juego. A parte tendrá conocimiento de los juegos que se han utilizado para el estudio, y, en caso de haber revisado los anexos, de probabilidad y álgebra.

2. El lector conoce las partes del algoritmo de MCTS, sus variaciones y diversas optimizaciones. Así mismo entiende estos mismos conceptos para PMCTS y conoce sus diferencias. A parte, el lector ha recibido la demostración de por qué realmente funciona la estimación del valor heurístico desde una perspectiva formal.

3. Se ha implementado un jugador para el juego del Reversi basado en MCTS y otro basado en PMCTS. Lo hemos enfrentado a *Random*, *Profundidad* y diversas versiones de *Alpha-beta*. Durante el proceso hemos ido adaptando los resultados para que se ajustarnos a cada situación y poder dar un buen resultado.

4. Hemos implementado igualmente un jugador basado en cada versión del algoritmo para el ajedrez, estos han demostrado no ser tan eficaces como los del punto anterior aunque, aún dados los bajos recursos computacionales de los que han sido dispuestos, han conseguido una buena finalización de partida teniendo en cuenta que en la mayoría de veces no se llegara al jaque mate.

5. Se han introducido tres herramientas potentes para el estudio del aprendizaje automático: mínimos cuadrados, el perceptrón y las redes neuronales. Elementos necesarios para el desarrollo del jugador más potente actualmente AlphaGo, del que se ha explicado su funcionamiento y la relación con MCTS.

Trabajo futuro

En líneas de trabajo futuro, se debería mejorar el tiempo de ejecución de las simulaciones, a día de hoy una de las ideas más comunes para generalizar cálculos sencillos es la paralelización de datos en sistemas concurrentes. La utilización de aceleradores gráficos podría llevar a un nuevo nivel al algoritmo de MCTS obteniendo resultados más certeros en menor tiempo. La

implementación en sistemas con GPU's potentes podría ser una solución a este problema.

Por otra parte, creemos que en el juego del ajedrez, el espacio de estados es tan grande que solamente con aumentar las simulaciones no bastaría, sería necesario incrementar la potencia del sistema para que las simulaciones puedan ser realmente ejecutadas hasta el final del juego así como incluir un sistema de aprendizaje que mejorará con el tiempo el funcionamiento de este. La solución propuesta por AlphaGo sería un reto interesante que probablemente daría muy buenos resultados.

El código de los experimentos se encuentra bajo licencia en github en las siguientes URL:

- Ajedrez: https://github.com/gabomodia/MCTS
- Reversi: https://github.com/gabomodia/reversi

Chapter 6

Conclusions and Future Work

General conclusions

Given the impossibility of performing a full exploration of the game tree, we have presented 2 practical methods to solve this issue: the one encompassed by Minimax using domain knowledge, and the one proposed by MCTS through simulations. In both cases we are trying to calculate a heuristic for each node which reveals how good the node is. We have seen that a mixed approach that adds up expert knowledge to the one obtained by simulations also brings good results even surpassing algorithms as Alpha-beta improved with a heuristic.

It has been shown in this work that the Monte Carlo Tree Search algorithm works in a very solvent way, in environments where it has been proven. It flaunted good results against simple algorithm as well as with expert automatic players with a wide knowledge of the domain in which they are working. The aheuristicity of MCTS allows us to adapt it easily to every game, getting a highly resolutive player in a short time and without the need to inquire in the game.

The issues that appear when applying it have also been shown. In particular, we appreciate an important complexity when using it to games which contain cycles, or that do not have a defined length. We also exposed a practical solution to this conflict by sacrificing the aheuristicity.

In addition, we have seen that the major waste of computational resources is produced in the simulation phase, and that is in it in where we must make a stronger effort in optimization. We have seen that the more simulations we perform, the higher the reliability of the heuristic calculated for each node.

Objectives review

Now we will review, one at the time, each objective in the same order that they were defined in the introduction:

1. Once read this document, the reader, without previous knowledge of the matter, now understands concepts of game theory, data structures, and several solutions given by diverse researchers to the problem of exploring a game tree. Besides, the games that have been used in this research have also been presented. If the reader took the time to review the appendixes, they will have gathered by now concepts of algebra and probability.

2. The reader currently knows the different parts of the Monte Carlo Tree Search algorithm, the way it works, variations and optimizations. This way they understand these same concepts for PMCTS, and also their differences. Moreover, the reader has received the formal proof of why the heuristic value estimation works.

3. We have developed an MCTS based Othello player as well as one PMCTS version. We faced it with *Random*, *Depth First Search* and some *Alpha-beta* versions. During the process we have adapted to each situation to be able to get a good result.

4. Following the same path, we have implemented an automatic player base in each Monte Carlo version to play chess. These have shown not to be as efficient as the ones exposed in the Othello section. Nevertheless, given the low resources they were counting with, they got to offer some good results against *Random* normally dominating the game although they almost never arrived to a check mate situation.

5. Three powerful tools have been introduce to study the machine learning field: least squares, the perceptron and neural networks. Needful elements in the development of the current most powerful player: AlphaGo. We also explained AlphaGo's functioning and the roll that MCTS plays.

Future work

In future lines of work, the simulation's execution time should be improved. Nowadays, the most common idea to generalize relatively easy calculations is to make them work in parallel in systems with this capacity. The use of graphic accelerators could bring MCTS to a new level, obtaining more

precise feedback in a lower time. The implementation in systems like CUDA could be a solution to this issue.

On the other hand, we think that in chess game, the state space is so big that only with increasing the simulations probably would not be enough. It would be necessary to enhance the power of the system so the simulations are run until the end of the game and also it would be interesting to implement a machine learning system that helps making it more accurate as the time passes.

Apéndice A

Conceptos de probabilidad

Se muestran aquí la definición de diversos conceptos que han sido utilizados a lo largo del documento.

- **Espacio medible.** Un espacio medible o probabilizable es el par $(\Omega, \mathcal{A})$ dónde Ω es un espacio muestral y $\mathcal{A}$ es un σ-algebra de conjuntos de Ω. (Gonzalez, 2016)
- **σ-álgebra.** Dado un espacio muestral Ω, se dice que una familia de subconjuntos $\mathcal{A} \subset \mathcal{P}(\Omega)$ tiene estructura de σ-álgebra si y solo si se verifican:
  1. $\Omega \in \mathcal{A}$
  2. $\forall A \in \mathcal{A}, A^c \in \mathcal{A}$
  3. $\forall \{A_n : n \geq 1\} \subset \mathcal{A}, \bigcup_{n=1}^{\infty} A_n \in \mathcal{A}$
- **Medida de probabilidad.** Una función de conjunto $P : \mathcal{A} \to [0, 1]$ es llamada medida de probabilidad si y solo si satisface:
  1. $P(A) \geq 0, \forall A \in \mathcal{A}$
  2. $P(\Omega) = 1$
  3. $\forall \{A_n : n \geq 1\} \subset \mathcal{A}$ tal que $A_i \cap A_j = \emptyset, \forall i \neq j$

  $$P\left(\bigcup_{n=1}^{\infty} A_n\right) = \sum_{n=1}^{\infty} P(A_n)$$

  (Kolmogorov, 1956)
- **Aplicación medible.** Sean $(\Omega_1, \mathcal{A}_1)$ y $(\Omega_2, \mathcal{A}_2)$ espacios medibles. Se dice que $f : \Omega_1 \to \Omega_2$ es una aplicación medible si y solo si $f^{-1}(B) \in \mathcal{A}_1 \forall B \in \mathcal{A}_2$. (Corral, 2012)

- **Función medible.** Se llama función medible a toda aplicación medible donde el espacio de destino es $(\mathbb{R}^n, \beta^n)$ para algún n. (Corral, 2012)

- **Variable aleatoria.** Una variable aleatoria es una función medible $f : (\Omega, \mathcal{A}, P) \to (\mathbb{R}, \beta)$; es decir, donde el espacio inicial es un espacio de probabilidad. (Corral, 2012)

- **Función de distribución.** La función

$$F^{(x)}(a) = P^{(x)}(-\infty, a) = Px < a$$

donde $-\infty$ y $+\infty$ son valores permitidos de a, se llama función de distribución de la variable aleatoria x.(Kolmogorov, 1956)

- **Muestra aleatoria simple.** Dada una población X se llama muestra aleatoria simple de tamaño n a la repetición de $X_1, ..., X_n$ variables aleatorias independientes con distribución igual a la de X. Es decir, la función de distribución de la muestra $(x_1, ..., x_n)$ es

$$F(x_1, ..., x_n) = \prod_{i=1}^{n} F(x_i)$$

donde $F(x)$ es la función de distribución de la población X(Ángel Villegas, 2005)

- **Función de distribución empírica.** Dada una realización particular de una muestra $(x_1, ..., x_n)$, llamamos función de distribución empírica a

$$F_n^*(x) = \begin{cases} 0 & si \quad x < x_{(1)} \\ \frac{k}{n} & si \quad x_{(k)} \leq x < x_{(k+1)} 1 \quad si \quad x \geq x_{(n)} \end{cases}$$

donde $(x_{(1)}, ..., x_{(n)})$ es la muestra ordenada de menor a mayor.(Ángel Villegas, 2005)

- **Correlación** Magnitud que mide el grado de relación existente entre dos variables aleatorias. Es medido por el coeficiente de correlación.

- **Coeficiente de correlación** El coeficiente de correlación r de dos variables aleatorias X e Y es la media aritmética de los productos de las desviaciones de los valores correspondientes de sus respectivas medias. (Kenney, 1939)

$$r = \frac{\frac{1}{N}\sum(x - \bar{x})(y - \bar{y})}{\sigma_x \sigma_y}$$

Apéndice B

Conceptos de álgebra y análisis

Presentamos a continuación definiciones y proposiciones que se han utilizado en el texto.

- **Espacio vectorial.** Sea $\mathbb{K}$ un cuerpo y V un conjunto no vacío; diremos que V es un espacio vectorial sobre $\mathbb{K}$ si:

  1. En V hay definida una operación interna, que denotaremos por $+$, de forma que $(V,+)$ es un grupo abeliano, es decir, verifica estas propiedades:
     a) $(u+v)+w = u+(v+w); \forall u,v,w \in V$
     b) $u+v = v+u; \forall u,v \in V$
     c) $\exists 0 \in V$ tal que $0+v = v+0 = v; \forall v \in V$
     d) $\forall v \in V \exists -v$ tal que $v+(-v) = (-v)+v = 0$
  2. En V hay definida una operación externa de $\mathbb{K}$, que denotaremos por yuxtaposición, verificando:
     a) $a(u+v) = au+av; \forall a \in \mathbb{K}, \forall u,v \in V$
     b) $(a+b)u = au+bu; \forall a,b \in \mathbb{K} \forall u \in V$
     c) $a(bu) = (ab)u; \forall a,b \in \mathbb{K} \forall u \in V$
     d) $1u = u; \forall u \in U$ donde 1 es la unidad para el producto en $\mathbb{K}$

  los elementos del espacio vectorial suelen llamarse vectores, mientras que a los del cuerpo $\mathbb{K}$ los llamaremos escalares. (Merino y Santos, 2006)

- **Independencia lineal.** Diremos que un conjunto de vectores $\{v_1, v_2, ..., v_n\}$ son linealmente independientes si de cada combinación lineal $a_1v_1 + ... + a_nv_n = 0$ de deduce que $a_1 = ... = a_n = 0$.(Merino y Santos, 2006)

- **Sistema de generadores.** Un conjunto de vectores S se dice que es un sistema de generadores del espacio vectorial V si todo vector de V es combinación lineal de los vectores de S.(Merino y Santos, 2006)

- **Base de un espacio.** Dado un espacio vectorial V, un subconjunto $B \subseteq V$ es una base de V si

  1. B es linealmente independiente.
  2. B es sistema de generadores de V

- **Dimensión de un espacio vectorial.** Llamamos así al número de vectores en cualquiera de las bases.(Merino y Santos, 2006)

- **Subespacio vectorial.** Sea V un espacio vectorial sobre $\mathbb{K}$ y sea U un subconjunto no vacío de V. Decimos que U es un subespacio vectorial de V si se verifican las siguientes condiciones:

  1. $\forall u, v \in U u + v \in U$
  2. $\forall u \in U \forall a \in \mathbb{K}, au \in U$

- **Espacio afín.** Dado un conjunto no vacío $\mathcal{A}$ y un espacio vectorial V diremos que $\mathcal{A}$ es un espacio afín sobre V si se tiene definida la aplicación

  $$\mathcal{A} \times \mathcal{A} \to V$$

  que a cada par de elementos de $\mathcal{A}$, (A, B), le hace corresponder un único vectores $\vec{AB}$, y que verifica las dos siguientes propiedades:

  1. Para cada $A \in \mathcal{A}$ y cada $v \in V$ existe un único elemento $B \in \mathcal{A}$ tal que $\vec{AB} = v$.
  2. Para cada terna $A, B, C \in \mathcal{A}$ ocurre $\vec{AB} + \vec{BC} = \vec{AC}$

  diremos que V es el espacio vectorial asociado y definimos la dimensión del espacio afín como la dimensión del espacio vectorial asociado. (Merino y Santos, 2006)

- **Hiperplano.** Llamaremos variedad afín a la generalización de espacio vectorial a espacio afín, y llamaremos hiperplano a una variedad afín de dimensión $n - 1$ en un espacio de dimensión n.(Merino y Santos, 2006)

- **Traza de una matriz.** Suma de los elementos de la diagonal de una matriz. Dada una matriz A denotaremos su traza como $tr(A)$ la traza cumple las siguientes propiedades:

  1. $tr(A) + tr(B) = tr(A + B)$
  2. $atr(A) = tr(aA)$ para todo a escalar
  3. $tr(A) = tr(A^t)$
  4. $tr(AB) = tr(BA)$

- **Norma de un vector.** Dado un vector u definimos su norma como:

$$||u|| = \sqrt{< u, u >}$$

(Merino y Santos, 2006)

- **Derivada.** Sea $I \subseteq \mathbb{R}$ un intervalo, $f : I \to \mathbb{R}$ una función y $c \in I$. Decimos que un número real L es la derivada de F en c si dado cualquier $\varepsilon > 0$ existe $\delta(\varepsilon) > 0$ tal que si $c \in I$ satisface $0 < |x - c| < \delta(\varepsilon)$ entonces

$$\left| \frac{f(x) - f(c)}{x - c} - L \right| < \varepsilon$$

En este caso diremos que f es diferenciable en c y escribimos $f'(c)$. Podemos calcular esto entonces como:

$$f'(c) = \lim_{h \to 0} \frac{f(c+h) - f(c)}{h}$$

. (Bartle y Shebert, 1927)

- **Derivada direccional.** En R^n generalizamos el concepto de derivada y lo llamamos derivada direccional:

$$D_v f = \lim_{h \to 0} \frac{f(x + hv) - f(x)}{h}$$

(Bombal et al., 1988)

- **Gradiente.** Llamaremos gradiente de f y lo denotaremos por ∇f al vector:

$$(D_{e_1} f, ..., D_{e_n} f)$$

Siendo $e_1, ..., e_n$ los vectores de la base canónica.(Bombal et al., 1988)

Bibliografía

Y así, del mucho leer y del poco dormir,
se le secó el celebro de manera que vino
a perder el juicio.

Miguel de Cervantes Saavedra

Aguado, J. C. Teoría de juegos. 2015.

Bartle, R. G. y Shebert, D. R. *Introduction to real analysis*. John Wiley and Sons Inc., 1927. ISBN 978-0-471-43331-6.

Bishop, C. M. *Pattern recognition and Machine Learning*. Springer, 2006. ISBN 978-0387-31073-2.

Bombal, F., Marín, L. y Vera, G. *Problemas de análisis matemático*. Ediciones Electolibris S.L, 1988. ISBN 978-8494615085.

Brams, S. J. Game theory and the missile crisis. Disponible en `https://plus.maths.org/content/game-theory-and-cuban-missile-crisis`.

Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S. y Colton, S. A survey on monte carlo tree searh methods. Disponible en `http://mcts.ai/pubs/mcts-survey-master.pdf`.

Chaslot, G. M.-B., Winands, M. H., Herik, H. J. V. D. y Uiterwijk, J. W. Progressive strategies for monte-carlo tree search. Disponible en `https://dke.maastrichtuniversity.nl/m.winands/documents/pMCTS.pdf`.

Cormen, T. H., Liserson, C. E. y Ronald L.Rivest, C. S. *Introduction to algorithms*. The MIT press, 2009.

Corral, A. G. *Apuntes de Probabilidad*. Universidad Complutense de Madrid, 2012.

FRANÇAISE DÓTHELLO, F. Une historie de 135 ans. Disponible en http://www.ffothello.org/othello/histoire/.

FISZ, M. *Probability Theory and Mathematical Statistics*. John Wiley and sons, 1963.

GONZALEZ, A. M. J. *Probabilidad*. Universidad de Almería, 2016. ISBN 978-84-16642-25-0.

GONZALEZ, D. T. Conceptos básicos de teoría de juegos. 2015.

GURNEY, K. *An Introduction to Neural Networks*. University of Sheffield, 2005. Disponible en https://www.inf.ed.ac.uk/teaching/courses/nlu/assets/reading/Gurney_et_al.pdf.

VAN DEN HERIK, H. The nature of minimax search. Disponible en https://project.dke.maastrichtuniversity.nl/games/files/phd/Beal_thesis.pdf.

HORNIK, K., STINCHCOMBE, M. y WHITE, H. Multilayer feedforward networks are universal approximators. Disponible en https://www.sciencedirect.com/science/article/pii/0893608089900208.

JACKSON, M. O. A brief introduction to the basics of game theory. Disponible en https://papers.ssrn.com/sol3/papers.cfm?abstract_id=1968579.

KENNEY, J. F. *Mathematics of Statistics part I*. Chapman and Hall, Disponible en https://archive.org/details/MathematicsOfStatisticsPartI.

KOLMOGOROV, A. *Foundations of the theory of probability*. Chelsea publishing company, Disponible en https://www.york.ac.uk/depts/maths/histstat/kolmogorov_foundations.pdf.

KRIESEL, D. *A Brief Introduction to Neural Networks*. University of Bonn, Disponible en http://www.dkriesel.com/_media/science/neuronalenetze-en-zeta2-2col-dkrieselcom.pdf.

LARAMÉE, F. D. Chess programming part iv: Basic search. Disponible en https://www.gamedev.net/articles/programming/artificial-intelligence/chess-programming-part-iv-basic-search-r1171.

MERINO, L. y SANTOS, E. *Álgebra lineal con métodos elementales*. Paraninfo, 2006. ISBN 978-84-9732-481-6.

NAIR, S. A simple alpha(go) zero tutorial. Disponible en https://web.stanford.edu/~surag/posts/alphazero.html.

Pearl, J. *Heuristics: Intelligent search strategies for computer problem solving*. Addison-Wesley Pub. Co., Inc., 1984.

Pieterse, V. y Black, P. E. *Algorithms and Theory of Computation Handbook*. CRC Press LLC, Disponible en `https://www.nist.gov/dads/HTML/tree.html`.

Rose, B. *Othello: A minute to learn, a lifetime to master*. Anjar Co, Disponible en `https://web.archive.org/web/20110512210955/http://othellogateway.com/rose/book.pdf`.

Ross, D. *Game Theory*. Stanford Encyclopedia of Philosophy, Disponible en `https://plato.stanford.edu/entries/game-theory/#Mot`.

Rouaud, M. *Probability, Statistics and Estimation*. Springer, 2013. ISBN 978-2954-93090-9.

Rusell, S. y Norvig, P. *Artificial Intelligence, A modern approach*. Prentice Hall, 2003. ISBN 9-78-0-13604259-4.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. y Hassabis, D. Mastering the game of go with deep neural networks and tree search. *Nature*, 2017a.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K. y Hassabis, D. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *Nature*, 2017b.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., Driessche, G. v. d., Graepel, T. y Hassabis, D. Mastering the game of go without human knowledge. *Nature*, vol. 550(7676), 2017c. ISSN 1476-4687.

Smola, A. y Vishwanathan, S. *Introduction to Machine Learning*. Cambridge University Press, 2008. Disponible en `http://alex.smola.org/drafts/thebook.pdf`.

Ángel Villegas, M. *Inferencia estadística*. Días de Santos, 2005. ISBN 84-0000-000-X.

White, J. *Bandit Algorithms for Website Optimization*. O'Reilly, 2013. Disponible en `http://shop.oreilly.com/product/0636920027393.do`.

WIKIPEDIA (Chess). Disponible en `https://en.wikipedia.org/wiki/Chess_piece_relative_value`.

ZENG, W. y CHURCH, R. Finding shortest paths on real road networks:the case for a*. 2007.

–¿Qué te parece desto, Sancho? – Dijo Don Quijote –
Bien podrán los encantadores quitarme la ventura,
pero el esfuerzo y el ánimo, será imposible.

Segunda parte del Ingenioso Caballero
Don Quijote de la Mancha
Miguel de Cervantes

–Buena está – dijo Sancho –; fírmela vuestra merced.
–No es menester firmarla – dijo Don Quijote–,
sino solamente poner mi rúbrica.

Primera parte del Ingenioso Caballero
Don Quijote de la Mancha
Miguel de Cervantes