GRADO EN INGENIERÍA DEL SOFTWARE (PLAN 2009)

PROYECTO FIN DE GRADO

Curso 2016/2017

APLICACIÓN DE INTELIGENCIA ARTIFICIAL EN VIDEOJUEGOS

USO DE LA VARIANTE DEL ALGORITMO MINIMAX PODA ALPHA-BETA PARA SU DESARROLLO

ALUMNO: PABLO LUIS GUTIÉRREZ UTANDE

TUTOR: ÁNGEL ARROYO CASTILLO

ÍNDICE

RESUMEN

Actualmente, los videojuegos forman parte del día a día de un gran porcentaje de personas, con independencia de su edad, sexo, etc. Estos videojuegos, cuya complejidad puede variar desde unas pocas líneas de código a proyectos de varios años que involucran un alto número de programadores y diseñadores no habrían tenido tal éxito de no haber sido por un factor determinante: el desarrollo de la inteligencia artificial en ellos. Por ello, en este proyecto se presentará brevemente la historia de la inteligencia artificial en los videojuegos desde sus comienzos, definiendo los hitos más relevantes y desarrollando un pequeño juego que sirve como ejemplo para definir los puntos importantes a la hora de desarrollar una inteligencia artificial en un videojuego.

Palabras clave: inteligencia artificial, MiniMax, Alpha-Beta, ProjectAI, videojuegos

---

ABSTRACT

Nowadays, video games take part of the routine of a wide range of people, regardless of their age, gender, etc. These video games, whose complexity can range from a few lines of code to a several years project involving a large number of programmers and designers would not have had such success if it was not for a determining factor: the development of artificial intelligence. Therefore, in this project it will be briefly presented the history of artificial intelligence in video games from the beginning, defining the most relevant milestones and developing a small game that serves as an example to explain the most important points to take into account when the artificial intelligence of a videogame is being developed.

Keywords: artificial intelligence, MiniMax, Alpha-Beta, ProjectAI, videogames

Capítulo I. OBJETO Y METODOLOGÍA DE LA INVESTIGACIÓN

INTRODUCCIÓN. - OBJETO DEL PROYECTO. - METODOLOGÍA DE LA INVESTIGACIÓN. - FUENTES. - ESTRUCTURA DEL TRABAJO.

Introducción

La inteligencia artificial (IA) es uno de los campos de la Ingeniería del Software que más impacto está teniendo en los últimos años en la sociedad y en el cual se están invirtiendo gran cantidad de recursos[1]. El hecho de desarrollar sistemas que sean capaces de manejar situaciones de manera autónoma y decidir por sí mismos que opción tomar sin interacción de ningún humano es un reto de gran magnitud en el que, cada vez más, se ven resultados satisfactorios.

Este desarrollo tiene a su vez unas connotaciones éticas y morales —las cuales tienen su propia rama de estudio en la ética[2]—. Si se consigue desarrollar una inteligencia artificial tan potente como para razonar como un humano deben de plantearse una serie de cuestiones éticas que abarcan diferentes puntos, tanto que las máquinas no

[1] Como muestra sirven algunas noticias publicadas en los últimos tiempos, entre ellas: "Inteligencia artificial: una inversión atractiva para el capital de riesgo" (*Expansión*, 13/04/2017. Accesible desde: http://www.expansion.com/economia-digital/innovacion/2017/04/13/58ef5788ca4741445d8b45a0.html); "Inversión en robótica e inteligencia artificial" (*El blog de Paula Mercado*, *VozPópuli*, 02/02/2017. Accesible desde: http://www.vozpopuli.com/el_blog_de_paula_mercado/Inversion-robotica-inteligencia-artificial_7_995670426.html). De hecho, desde enero a junio de 2016, las más de 250 startups dedicadas a la inteligencia artificial que recoge la base de datos Crunchbase habían recaudado más de 600 millones de dólares gracias a la inversión en ésta. Fuente: https://nexo.club/la-inteligencia-artificial-en-cifras-estas-son-las-diez-startups-que-est%C3%A1n-conquistando-el-7731ebf47496 [Fecha de consulta: 01/12/2016]

[2] La ética en la inteligencia artificial se divide, a su vez, en otras dos ramas: roboética y ética de las máquinas.

lastimen a seres humanos o seres vivos en general[3], como el estatus moral propio de la máquina[4].

De hecho, los videojuegos y la inteligencia artificial son dos elementos que siempre han estado fuertemente ligados[5]. Desde el inicio de la computación, la idea de conseguir que un ordenador tenga un comportamiento similar al de una persona humana de una manera principalmente autónoma ha supuesto un gran reto.

Una de las maneras empleadas para conseguir esto es comprobar si la máquina es capaz de realizar tareas humanas, como por ejemplo, mantener una conversación sobre cualquier tema sin que la persona que participa sepa si está conversando con otro humano o con una máquina. Para ello, se realiza el test de Turing. Otro ejemplo posible es mediante el *trading*, donde mucha gente trabaja desarrollando software que sea capaz de invertir en bolsa de manera autónoma, consiguiendo beneficios[6]. En los últimos tiempos están desarrollándose de forma vertiginosa los vehículos autónomos, de manera que sean capaces de reaccionar ante cualquier situación: en este caso, no siempre se busca que tenga el comportamiento humano, ya que muchas veces estos actúan/determinan la solución de manera errónea, con consecuencias que pueden llegar a ser peligrosas para los seres humanos que los conducen.

Los juegos en los que intervienen dos jugadores como ajedrez, damas, Othello© y Backgammon —por citar algunos—, son un excelente punto de partida para

[3] HARFNOSKY, Holden. "Potential Risks from Advanced Artificial Intelligence: The Philanthropic Opportunity". En línea: http://www.openphilanthropy.org/blog/potential-risks-advanced-artificial-intelligence-philanthropic-opportunity [Fecha de consulta: 27/07/2016]

[4] BOSTROM, Nick, YUDKOWSKY, Eliezer. "The Ethics of Artificial Intelligence", en RAMSEY, William, FRANKISH, Keith [eds.], *Cambridge Handbook of Artificial Intelligence*. Londres: Cambridge University Press, 2014, pp. 316-334.

[5] SCHREINER, Tim. "Artificial Intelligence in Game Design", en *Artificial Intelligence Depot*. En línea: http://ai-depot.com/GameAI/Design.html [Fecha de consulta: 01/02/2017]

[6] Ramón López de Mántaras, director del Instituto de Inteligencia Artificial (CSIC), no obstante, cree que deberían eliminarse los robots con inteligencia artificial aplicados en Bolsa: «… Ha habido hundimientos de la Bolsa cuyos culpables son estos agentes que toman decisiones y compran o venden acciones en milisegundos. No negocian nada. […] Es muy rentable, claro. Pero también hubo un 'crash' enorme en la Bolsa por esta causa. Para mí son algo pernicioso y debería estar prohibido completamente. […] Siempre hemos de tener control sobre las máquinas, sean físicas o de software. La autonomía plena, dentro de la AI, la prohibiría completamente», según el artículo de Alfonso Plasencia para *El Mundo*. En línea:

desarrollar esta tarea. Estos juegos ofrecen entornos de información cerrados de los cuales es relativamente sencillo extraer la información relevante del juego.

Objeto del proyecto

En este caso, para el Proyecto de Fin de Grado (PFG) —dentro de la asignatura con el mismo nombre del Grado en Ingeniería de Software— se va a desarrollar una inteligencia basada principalmente en la fuerza bruta, utilizando para ello una variante del algoritmo MiniMax, que es la poda Alpha-Beta, lo que permite desarrollar la inteligencia de un contrincante complicado por el ordenador para un sencillo juego desarrollado en C++, en el que dos jugadores se enfrentan entre sí para conquistar el edificio principal del contrincante.

Tema de investigación

La motivación principal del desarrollo de este proyecto es el estudio de la inteligencia artificial aplicada al campo de los juegos de estrategia, más específicamente, las inteligencias que no se basan en scripts predefinidos. Generalmente, en el campo de los videojuegos comerciales es más conveniente el uso de este tipo de inteligencias ya que no requieren apenas de capacidad computacional y es mucho más sencillo crear distintos niveles de dificultad y comportamientos pseudoaleatorios.

Siempre nos ha llamado la atención el desarrollo de la inteligencia artificial en los videojuegos, más aún en las últimas décadas en las que se están empleando una gran cantidad de recursos de procesamiento dado el aprovechamiento de las GPU de alta capacidad para este tipo de tareas. Por ese motivo se decidió realizar una aplicación de tamaño pequeño para comprobar la dificultad que conlleva el desarrollo de un sistema de este tipo.

---

http://www.elmundo.es/economia/2016/04/04/57021c97268e3e40248b4595.html [Fecha de consulta: 11/11/2016]

Objetivos del proyecto

Los principales objetivos de este Proyecto de Fin de Grado son los siguientes:

- Realizar en C++ un videojuego sencillo, el cual sea capaz de mantener una partida contra un contrincante humano.
- Estudiar los diferentes algoritmos de inteligencia artificial desarrollados para los videojuegos.
- Utilizar la librería multiplataforma SDL para el desarrollo del videojuego.
- Desarrollar el algoritmo MiniMax con la poda Alpha-Beta de manera eficiente.
- Desarrollar tests con la librería Gtest de Google para comprobar el correcto funcionamiento de los algoritmos utilizados.
- Utilizar el lenguaje C++ de forma óptima y precisa para llevar a cabo el desarrollo del proyecto.
- Utilizar patrones de diseño como MVC, Observer y Factory para realizar una codificación con un diseño estructurado.
- Realizar la documentación pertinente en cuanto a cómo se ha desarrollado la inteligencia artificial en el videojuego del proyecto en concreto, así como en otros proyectos importantes a lo largo de la historia.

Estado de la cuestión

La inteligencia artificial aplicada a diversos ámbitos es un tema bastante tratado desde los inicios de la computación. Entre otras, han de destacarse las publicaciones de Claude Shannon en 1949, donde éste describe los posibles mecanismos que podrían usarse para construir un programa que jugase al ajedrez así como el desarrollo de Arthur Samuel relativo al Aprendizaje Automático aplicado al juego de Damas, que era capaz de mantener un nivel aceptable de juego. En las últimas décadas se han desarrollado algoritmos capaces de superar a los mejores jugadores del mundo de ajedrez (muestra de ello es la partida “Gary Kasparov vs Deep Blue”) o de Go (“Lee Sedol vs AlphaGo”). Éstos últimos han sido realmente importantes, ya que marcan un

antes y un después en la determinación de objetivos alcanzables mediante el uso de la inteligencia artificial, ya no sólo aplicada al mundo de los videojuegos sino de una manera general en otros entornos cotidianos.

Metodología de la investigación

Para la realización del presente proyecto se ha seguido una metodología de investigación genérica, a fin de rentabilizar todo el esfuerzo y lograr los objetivos propuestos según normativa[7], empleando igualmente una estructura preestablecida en la presente memoria: descripción del objeto de trabajo, con incidencia en el tema de investigación, los objetivos a desarrollar, un breve estado de la cuestión, así como la descripción metodológica seguida, fuentes empleadas en su elaboración y la descripción de la estructura de trabajo. Previo, se incluye una introducción al tema, así como el desarrollo del mismo. Finalmente, se establece un apartado de conclusiones.

Para el desarrollo general del mismo se realizó una planificación previa en la que se estimaron los posibles temas del Proyecto, así como se determinó la elección final y su comunicación al tutor. Se procedió a realizar una búsqueda de fuentes primarias de información de las que extraer contenidos mínimos y básicos sobre el tema. De igual manera, se comprobó el funcionamiento de una serie de videojuegos a fin de determinar paralelismos posibles con el que integra este proyecto —entre ellos, *Advance Wars* o la saga *Heroes of Myth and Magic*—. Se procedió a la realización del código del programa propuesto y se compilaron finalmente las conclusiones del desarrollo del mismo en la presente memoria del Proyecto.

Para las referencias bibliográficas se ha seguido la norma ISO 690-1987 ampliada en 2013 (y su equivalente UNE 50-104-94) para documentos impresos, tanto en las notas a pie de página como en la bibliografía final. Se han empleado comillas inglesas para las referencias a artículos (“”), para las notas referidas a documentos electrónicos se

[7] Trabajos Fin de Grado (actualización: curso 2014-2015, 61iw). Grado en Ingeniería del Software, plan 2009. En línea: https://www.etsisi.upm.es/estudios/grados/61iw/tfg

ha empleado la ISO 690-2, por sistematizar las transcripciones y presentación de la información, aunque con alguna salvedad: se han omitido los corchetes (<>) entre los que incluir las direcciones dado que, por defecto, los programas editores de texto los eliminan, dejando la dirección URL marcada con hipervínculo. Para la gestión de las mismas se ha empleado RefWorks[8] (citas de documentos) y la herramienta Zotero[9] (citas de páginas web).

Fuentes

Las fuentes de información empleadas —citando las más relevantes— para el desarrollo del Proyecto de Fin de Grado se pueden dividir en primarias y secundarias, creando una subdivisión propia en las primeras:

- Fuentes primarias de información:
    - Fuentes primarias bibliográficas: ha sido imprescindible el uso de material bibliográfico como el manual especializado en inteligencia artificial para videojuegos *AI for Game Developers*, de David M. Bourg y Glenn Seeman, así como *Inteligencia* Artificial, de Elaine Rich y Kevin Knight y diversos *papers* y *preprints* localizados en repositorios especializados.
    - Fuentes primarias en línea: consulta de páginas web y blogs especializados en inteligencia artificial tales como http://www.checkmarkgames.com, así como de temas generales, destacando https://stackoverflow.com
    - Fuentes primarias de código: se ha estudiado el comportamiento de videojuegos basados en estrategias por turnos como *Advance Wars*, juego para Game Boy Advance (Nintendo), o la saga de *Heroes of Myth and Magic*, para PC.
- Fuentes secundarias de información: se han empleado blogs y páginas web de grupos de investigación, así como referencias de fuentes periódicas relativas a

[8] RefWorks: http://www.upm.es/UPM/Biblioteca/ServiciosUsuario?id=32ea0649eb0f5110VgnVCM10000009c7648a____&fmt=detail

[9] Zotero: https://www.zotero.org/

la inteligencia artificial (noticias, artículos de opinión…) para apoyar los contenidos descritos de las fuentes primarias.

Estructura del trabajo

El presente trabajo se estructura, como se ha indicado someramente en el apartado anterior, como sigue:

- Capítulo I, que incluye una introducción en la que se describe la metodología empleada en su redacción así como el objeto del desarrollo del mismo.
- Capítulo II, con una breve referencia a la historia de los videojuegos de estrategia —en tiempo real y por turnos—.
- Capítulo III, donde se describe la inteligencia artificial empleada en los juegos de estrategia y se establecen las características básicas de MiniMax y la aplicación en varios juegos de mesa.
- Capítulo IV, destinado a ProjectAI y la descripción de su arquitectura y diseño principal.
- Capítulo V, centrado en el algoritmo MiniMax, empleado para el desempeño del proyecto.
- Capítulo VI, donde se detalla la recopilación de datos y los resultados obtenidos tras la realización del trabajo

Además, se incluye un apartado final donde se destacan algunas de las conclusiones obtenidas, así como la bibliografía y fuentes de información empleadas y un índice de figuras.

Capítulo II. HISTORIA DE LOS VIDEOJUEGOS DE ESTRATEGIA

ESTRATEGIA EN TIEMPO REAL. - TÁCTICA EN TIEMPO REAL. - ESTRATEGIA POR TURNOS. - TÁCTICA POR TURNOS. – EJEMPLOS.

El género de los videojuegos de estrategia hace referencia a aquellos videojuegos que ponen a prueba las habilidades de pensamiento y planeamiento del jugador. Se requiere disponer de una gran habilidad de mando para poder gestionar los diferentes recursos disponibles en cada momento y su máximo aprovechamiento para conseguir la victoria de la partida.

En este tipo de videojuegos el jugador suele tener una vista absoluta del mismo, ya sea a vista descubierta desde el principio o mediante la exploración gradual del terreno. El origen de estos videojuegos está íntimamente ligado con los juegos de mesa de estrategia.

Los diferentes subgéneros en los que se pueden categorizar los videojuegos de acuerdo a la dinámica de juego y al predominio de la táctica o estrategia son:

- Estrategia y táctica en tiempo real
- Estrategia y táctica por turnos

Estrategia en tiempo real

Los videojuegos de estrategia en tiempo real o RTS (por sus siglas en inglés real-time strategy) son videojuegos de estrategia en los que la acción transcurre de forma continuada en el tiempo.

Los RTS están pensados para ser jugados de forma muy dinámica y rápida. A diferencia de los basados en turnos, los videojuegos de estrategia en tiempo real no precisan un planteamiento tan pausado de las decisiones y se centran muy a menudo en la acción

militar. La recolección de recursos suele ser muy simple para evitar desviar la temática del juego y las batallas se representan a una escala de refriega: no obstante, hay que destacar que podemos encontrarnos con varios juegos que se centran en representar batallas multitudinarias con millares de unidades en el terreno, como sucede en las batallas de la saga *Total War*[10]. Como indica Nareyek[11], es recomendable emplear agentes autónomos[12] en el modelado del comportamiento de personajes, dado que dicha técnica permite resolver problemas como el manejo de tiempo real, dinámica, recursos y un conocimiento incompleto.

Algunos ejemplos de juegos RTS son las diferentes sagas de *Dune*[13], *Command & Conquer*[14], *Warcraft*[15], *StarCraft*[16], *Age of Empires*[17], *Empire Earth*[18], *Warlords*[19], y *The Battle For Middle Earth*[20].

[10] *Total War* (saga). Desarrollador: The Creative Assembly. Distribuidor: Electronic Arts. 2000-actualidad. https://www.totalwar.com/

[11] NAREYEK, Alexander. "Intelligent Agents for Computer Games", en MARSLAND, T.A., FRANK. I. [eds.], *Computer and Games, Second International Conference, CG 2000*, Springer LNCS 2063, pp. 414-422. En línea: http://www.ai-center.com/references/nareyek-02-gameagents.html [Fecha de consulta: 10/09/2016]

[12] Al respecto, MUÑOZ, Norman et al. "Uso de la metodología GAIA para modelar el comportamiento de personajes en un juego de estrategia en tiempo real", en *Revista de la Facultad de Ingeniería de la Universidad de Amtioquía*, n. 53, 2010, p. 216, destacan que «[…] Un buen ejemplo de la utilización de agentes en juegos es Empire Earth de Sierra en el cual la Inteligencia Artificial se compone de varios agentes llamados administradores. Juegos como Civilization, Balance of Power y Populous usan la tecnología de agentes para reaccionar a las acciones del jugador tal como lo haría un ser humano, logrando que los jugadores piensen que se trata de un oponente real».

[13] *Dune*. Desarrollador: Cryo Interactive. Distribuidor: Virgin Interactive. 1992, 1993.

[14] *Command & Conquer*. Desarrollador-distribuidor: WestWood Studios (1995-2002); Electronic Arts (2003-2010); EA Phenomic (2011); Victory Games (2011-). https://www.ea.com/es-es/games/command-and-conquer

[15] *World of Warcraft*. Diesarrollador-distribuidor: Blizzard Entertainment. 1994- . https://worldofwarcraft.com/es-es/

[16] *StarCraft*. Desarrollador: Blizzard Entertainment. Distribuidor: Blizzard Entertainment, Nintendo, Sierra. 1998- . http://eu.blizzard.com/es-es/games/sc/

[17] *Age of Empires*. Desarrollador: Ensemble Studios, Skybox Labs. Distribuidor: Microsoft Game Studios. 1997- . http://eu.blizzard.com/es-es/games/sc/

[18] *Empire Earth*. Desarrollador: Stainless Steel Studios. Distribuidor: Sierra On-Line. 2001- .

[19] *Warlords*. Desarrollador-distribuidor: Atari. 1980-.

[20] *The Battle for Middle Earth*. Desarrollador: EA Los Angeles. DIstribuidor: Electronic Arts. 2004 - . http://www2.ea.com/lotr-the-battle-for-middle-earth-2

Táctica en tiempo real

También son conocidos como RTT (por sus siglas en inglés real-time tactics). Compartiendo aspectos de los juegos de simulación y juegos de guerra, los juegos de táctica en tiempo real se enfocan en aspectos operacionales y control de guerra. A diferencia de los juegos de estrategia en tiempo real, el manejo económico y de recursos y la construcción de edificios no forman parte de las batallas.

Algunos ejemplos son las sagas *Commandos*[21], *Warhammer: Dark Omen*[22], *World in Conflict*[23] y *Close Combat*[24].

Estrategia por turnos

También conocidos por sus siglas en inglés TBS (turn-based strategy). El término "juego de estrategia por turnos" generalmente se aplica a ciertos videojuegos de estrategia para distinguirlos de los juegos de estrategia en tiempo real. Un jugador de un juego por turnos posee un período de análisis antes de realizar una acción.

Algunos ejemplos de este género son *Empire-Strike*[25], *Civilization*[26], y las sagas *Heroes of Might and Magic*[27] y *Master of Orion*[28].

Los juegos por turnos vienen en dos formas dependiendo de si, en un turno, los jugadores juegan simultáneamente o juegan sus turnos en secuencia. Los primeros son llamados juegos de estrategia por turnos simultáneos; siendo *Diplomacy* un ejemplo

[21] *Commandos*. Desarrollador: Pyro Studios. Distribuidor: Eidos Interactive. 1998-2006.
[22] *Warhammer: Dark Omen*. Desarrollador: Mindscape, Games Workshop. Distribuidor: Electronic Arts. 1998- .
[23] *World in Conflict*. Desarrollador: Massive Entertainment. Distribuidor: Sierra Entertainment. 2007- .
[24] *Close Combat*. Desarrollador: Atomic Games. Distribuidor: Microsoft. 1996.
[25] *Empire-Strike*. Desarrollador-distribuidor: Miguel González. 2004-2017. https://www.empire-strike.com/
[26] *Civilization*. Desarrollador: Sid Meier. Distribuidor: MicroProse. 1991- . https://www.civilization.com/
[27] *Heroes of Might and Magic*. Desarrollador: New World Computing... Distribuidor: Ubisoft. 1995- . https://www.ubisoft.com/es-es/game/might-and-magic-heroes-7
[28] *Master of Orion*. Desarrollador: Simtex. Distribuidor: MicroProse. 1994-2003.

notable. Los últimos caen en la categoría de los juegos de estrategia por turnos alternados, y a su vez se subdividen en:

- Posicional.
- Inicio round robin.
- Aleatorio.

La diferencia está en el orden en el que los jugadores juegan sus turnos. En posicional, los jugadores empiezan sus turnos en el mismo orden siempre. En el inicio round robin, el jugador que empieza es elegido de acuerdo a una política, cómo su propio nombre indica, round robin. En aleatorio el primer jugador es elegido al azar.

Casi todos los juegos de estrategia que no sean videojuegos (juegos de mesa) son por turnos. El mercado de los juegos de ordenador últimamente se ha inclinado más por los juegos en tiempo real.

Algunos juegos recientes han mezclado componentes de los juegos en tiempo real con componentes de los juegos por turnos. En estos juegos los jugadores disponen de 100 movimientos por día. Estos movimientos pueden ser tomados en cualquier momento de ese día independientemente de si otros jugadores hayan realizado sus movimientos o no.

Táctica por turnos

Conocidos también por sus siglas TBT (turn-based tactics). La jugabilidad táctica por turnos se caracteriza por la expectativa de los jugadores por completar sus tareas usando sólo las fuerzas de combate que se les proveen, y usualmente por la disposición de una representación realista (o por lo menos, creíble) de operaciones y tácticas militares.

Ejemplos del género son *Jagged Alliance*[29] y la saga *X-COM*[30], así como juegos de rol tácticos como *Final Fantasy Tactics*[31], la saga *Fire Emblem*[32] y los juegos de la desarrolladora Nippon Ichi.

Ejemplos

A continuación se presentan dos ejemplos de juegos de estrategia de los cuales se han extraído diferentes ideas para el desarrollo del juego de este proyecto.

Empire (Classic Empire) es el primer videojuego de estrategia por turnos. Concebido por Walter Bright en 1971, está basado en juegos de mesa como *Battle of Britain* y *Risk*.

La primera versión para ordenador fue lanzada en 1977 y estaba escrita en FORTRAN para el ordenador PDP-10 de Caltech[33]. Poco después Bright reescribió el código en ensamblador para de esta forma poder comercializarlo. Diferentes versiones fueron creadas en los siguientes años mejorando la interfaz del juego y dando pie a diferentes juegos creados después como *Civilization*.

Las reglas del juego son relativamente simples: al principio del juego se crea un mapa aleatorio basado en una cuadrícula, en el cual se reparten diferentes islas y ciudades. Los jugadores comienzan teniendo el control de una de estas ciudades, donde pueden crear diferentes tipos de unidades tácticas, como tropas, aviones y barcos, cada cual requiriendo distinto número de turnos para ser creadas.

[29] *Jagged Alliance*. Desarrollador: Madlab Software. Distribuidor: Sir-Tech. 1994.
[30] *X-COM*. Desarrollador-distribuidor: Hasbro Interactive. 1999.
[31] *Final Fantasy Tactics*. Desarrollador: Square Co. Ltd. Distribuidor: Sony. 1997-2007.
[32] *Fire Emblem*. Desarrollador: Nintendo, Intelligent Systems. Distribuidor: Nintendo. 2012- .
[33] "A Brief History of Empire". Accesible en línea: http://www.classicempire.com/history.html [Fecha de consulta: 12/02/2017]

El juego consiste en ir avanzando a lo largo del mapa, conquistando nuevas ciudades para tener mayor capacidad de creación de unidades, y buscar la conquista de las ciudades de los demás jugadores.

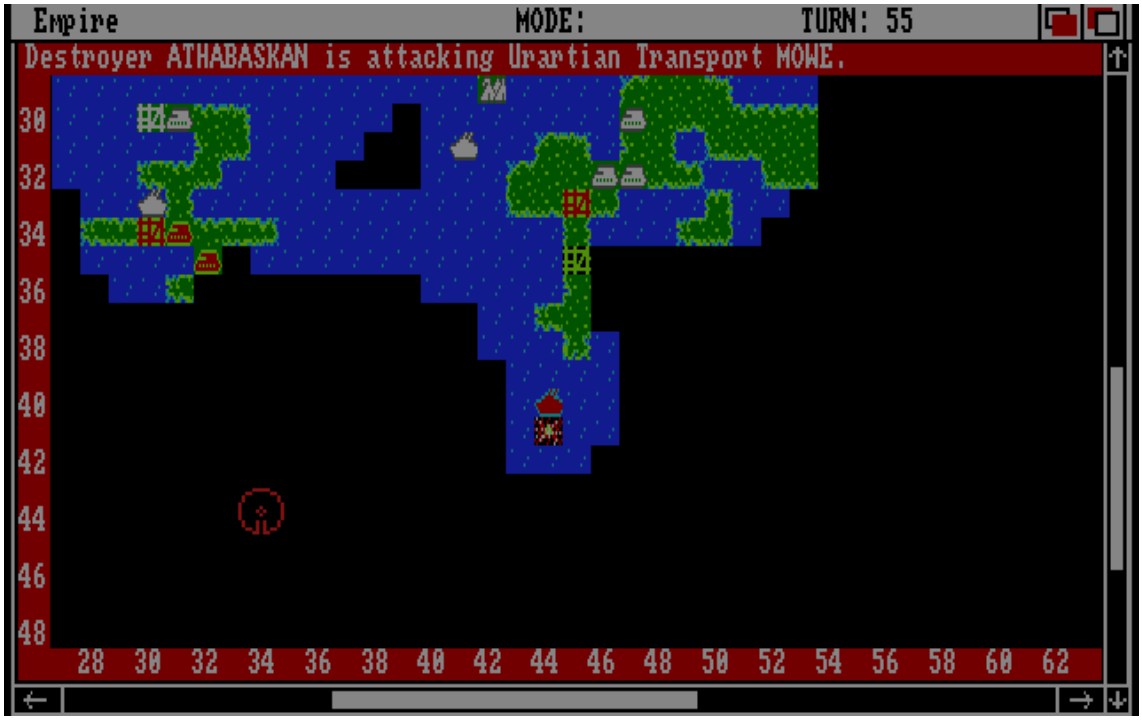


Figura 1. Pantalla de avance de juego (Empire). Fuente: elaboración propia.

Por su parte, *Advance Wars*, es un videojuego de táctica por turnos desarrollado Intelligent Systems para la Game Boy Advance y distribuido por Nintendo. Destaca por un diseño del mapa de juego sencillo e intuitivo que proporciona una gran jugabilidad.

Figura 2. Pantalla de avance de juego (Advance Wars). Fuente: elaboración propia.

El jugador y su oponente mueven sus tropas y atacan por turnos. Se obtiene la victoria eliminando las fuerzas oponentes o capturando su cuartel general. Las unidades pueden clasificarse como de combate directo, combate indirecto, infantería y transporte. Por otro lado, las unidades aéreas y navales requieren una cierta cantidad de combustible por turno para moverse. Si se quedan sin combustible, se estrellan/hunden, perdiéndose. Mientras que las unidades de infantería pueden capturar edificios, las de transporte son las encargadas de llevar otras unidades, así como generalmente suministrar combustible y munición. Las unidades indirectas pueden atacar a distancia, evitando los contraataques, pero no pueden moverse y disparar en el mismo turno. Cada unidad tiene distintos puntos fuertes y débiles respecto a otras, haciendo crucial la confección del propio ejército para enfrentarse al oponente.

Se destaca una curiosidad: los ejércitos del juego tienen ciertas similitudes con ejércitos que existen o existieron en la realidad[34]. Por ejemplo, *Orange Star* parece modelado a partir del ejército de los Estados Unidos de América[35]. *Blue Moon* se asemeja a la Unión Soviética por el tipo de boinas que llevan los soldados, que se asemejan a las que usaban los soldados soviéticos en la Segunda Guerra Mundial, *Yellow Comet* recuerda al ejército japonés de la Segunda Guerra Mundial, y *Green Earth* surge a partir de la Alemania de la Segunda Guerra Mundial.

Las unidades de *Black Hole*, sin embargo, son exactamente iguales a las de *Orange Star*, excepto por el color negro. Su OJ —oficial jefe—, Sturm, es de nacionalidad indeterminada, pues su identidad está totalmente oculta tras una máscara; a pesar de ello, aparece implícito en la secuela del juego que el ejército de Black Hole proviene de otro mundo (de hecho, la apariencia de sus unidades está alterada en la secuela para reflejar esto).

[34] Aunque no es el tema central del proyecto, ha de destacarse la labor educativa de algunos videojuegos, empleados en las aulas de educación primaria y secundaria para compartir conocimientos mediante gamificación. Son numerosos los proyectos llevados a cabo dentro y fuera de éstas —como ocurre, por ejemplo, en bibliotecas— así como la documentación escrita al respecto. Cada vez se pone más énfasis en adecuar el diseño interno —escenarios, vestuario, batallas— como externo —referencias temporales…— de los videojuegos de corte histórico. Ejemplo de ello lo encontramos en: IRIGARAY, María Victoria, LUNA, María del Rosario. "La enseñanza de la Historia a través de videojuegos de estrategia. Dos experiencias áulicas en la escuela secundaria", en *Clío & Asociados*, n. 18-19, 2014, pp. 411-437. En línea: http://sedici.unlp.edu.ar/handle/10915/47740 [Fecha de consulta: 07/01/2017]

[35] Como anécdota, se especula que Orange Star no se llamó Red Star fuera de Japón para evitar referencias a la estrella roja, utilizada frecuentemente como símbolo del comunismo.

Capítulo III. INTELIGENCIA ARTIFICIAL EN LOS JUEGOS DE ESTRATEGIA

COMIENZOS. - PRIMEROS PASOS CON MINIMAX. - EL RETO DEL AJEDREZ. - REDES NEURONALES. - PRIMER PROGRAMA PRÁCTICAMENTE IMBATIBLE: DEEP BLUE. - UN CONTRINCANTE DE GO COMPLETAMENTE IMBATIBLE.

La inteligencia artificial de la mayoría de videojuegos comerciales se produce mediante el uso de máquinas de estado y scripts los cuales actúan de cierta forma de acuerdo a la ocurrencia de un evento dado.

En muchas ocasiones se ha considerado que estos videojuegos tienen una inteligencia que se asemeja mucho al comportamiento humano y han sido premiados por ello, pero cabe preguntarse una cuestión: cuando estas acciones han sido preprogramadas por un humano, ¿qué porcentaje de inteligencia representa la aplicada por el ordenador, si es que realmente puede considerarse inteligencia como tal?

En la mayoría de estos juegos, el ordenador no es consciente de porqué ha tomado cierto camino o por qué ha llevado a cabo una determinada acción frente a otra. Tampoco aprende de sus errores o de los humanos cuando está jugando.

Otra técnica muy extendida para la aplicación de inteligencia artificial en los videojuegos comerciales es mediante el “engaño”. Por ejemplo, en cualquier juego de estrategia en el que haya un campo de batalla oculto mediante el uso de niebla que se va disipando según se explora el mapa, el jugador puede creer que el ordenador requiere del mismo procedimiento de exploración para poder tener acceso a lo que ocurre en el mapa. Sin embargo esto en la mayoría de los casos no ocurre, ya que el ordenador tiene desde el principio información completa de lo que sucede en el mapa y sabe dónde están localizadas las bases del jugador y sus unidades, para centrar así sus recursos en cómo organizar sus propios recursos y tropas y conseguir ganar la partida.

Sin embargo, el problema viene dado con el abuso de esta información. Es precisamente cuando el jugador se da cuenta de que el ordenador está haciendo trampas y siendo consciente que cualquier esfuerzo que hace o haga por ganar es inútil —ya que el ordenador siempre es capaz de saber dónde está y atacar en cualquier momento—, lo que acabará ocurriendo es que el jugador perderá interés en el juego.

Precisamente, la cuestión a la hora de crear una inteligencia artificial es encontrar ese balance en el que se proporcione un oponente digno que ofrezca una cierta dificultad pero que a la vez no sea imbatible y permita anotarse una victoria.

Buscar este equilibrio es un tema delicado y se torna una tarea difícil, ya que no todos los contrincantes humanos tienen la misma habilidad a la hora de jugar. Es importante tener una variedad de niveles de dificultad que puedan ofrecer la oportunidad de jugar al máximo número (diverso) de personas posibles.

En este caso, este tipo de videojuegos no tiene un alto interés, por lo que los esfuerzos se centran en el estudio de otros juegos como ajedrez[36] o Go, en los cuales puede haber ciertas acciones preprogramadas por una cuestión de rendimiento. Si se busca construir un programa infalible, se precisaría llegar más a fondo de la cuestión.

La principal ventaja del desarrollo de la inteligencia artificial en este tipo de juegos es que estos proporcionan una cantidad acotada de información y es relativamente sencillo medir el éxito o fracaso de las jugadas con un simple análisis del tablero de juego.

El hecho de que se trate de una tarea estructurada, de la que se puede obtener información completa, puede inducir a pensar que una "súperinteligencia" imbatible se puede conseguir examinando todos los posibles movimientos y escogiendo el mejor

[36] SHANNON, Claude E. "XXII. Programming a Computer for Playing Chess", en *Philosophical Magazine,* ser. 7, vol. 41, n. 314, marzo 1950. En línea: http://www.ee.ufpe.br/codec/Programming_a_computer_for_playing_chess.shannon.062303002.pdf [Fecha de consulta: 11/12/2016]

en función de los resultados obtenidos. El problema radica en la complejidad de cualquier videojuego actual, donde el factor de ramificación en media es sumamente elevado, y cualquier intento de comprobarlos todos sería impensable.

Comienzos

Desde el comienzo de la computación moderna, la idea de poder igualar o superar la inteligencia humana ha estado siempre presente. Puesto que la medición de la inteligencia humana es difícil, muchas personas pensaron que una buena forma de conseguirlo era con tareas que pudieran desafiar el intelecto humano, y entre ellas surgió la idea de los juegos de estrategia. Charles Babbage, considerado como «El Padre de la Computación», pensó en construir una máquina que fuese capaz de jugar a las tres en raya, e incluso programar su máquina analítica para que fuese capaz de jugar al ajedrez. Claude Shannon publicó en 1949 un artículo[37] describiendo los mecanismos que podrían usarse para escribir un programa que jugase al ajedrez, justificando la utilidad de resolver tal problema como se ve en su artículo:

> "The chess machine is an ideal one to start with, since: (1) the problem is sharply defined both in allowed operations (the moves) and in the ultimate goal (checkmate); (2) it is neither so simple as to be trivial nor too difficult for satisfactory solution; (3) chess is generally considered to require 'thinking' for skillful play; a solution of this problem will force us either to admit the possibility of a mechanized thinking or to further restrict our concept of 'thinking'; (4) the discrete structure of chess fits well into the digital nature of modern computers. … It is clear then that the problem is not that of designing a machine to play perfect chess (which is quite impractical) nor one which merely plays legal chess (which is trivial). We would like to play a skillful game, perhaps comparable to that of a good human player."

[37] SHANNON, Claude E., *op. cit.* (1950).

En dicho artículo básicamente explica que el ajedrez es ideal, ya que el problema, los movimientos y la condición de victoria están definidos de una manera muy clara; además, no es tan simple como para ser trivial ni demasiado complejo. La estructura discreta del ajedrez encaja bien en la naturaleza digital de los ordenadores modernos. Este autor finaliza diciendo que el problema no es tanto diseñar una máquina para jugar a un ajedrez perfecto ni tampoco una que juegue simplemente al ajedrez de manera simple, sino que lo ideal sería conseguir un contrincante que fuese capaz de jugar un juego hábil, tal vez comparable al de un buen jugador humano.

Considerando el caso del ajedrez, que a priori puede parecer bastante asequible y conducir a engaños como la *leyenda del Rey Sheram y el tablero de ajedrez*, donde el rey, que quería recompensar al inventor del juego, le dijo a éste que le pidiese lo que quisiese. Este ante tan generosa invitación le contestó que le bastaba con que le diese un grano con la primera casilla del tablero y por cada casilla adicional fuese sumando el número de manera exponencial. El rey ordenó que se le obsequiara inmediatamente con lo que él pedía pero, sin embargo, los matemáticos del reino, después de hacer todos los cálculos pertinentes, indicaron al rey que se tardarían miles de años en conseguir tal número de granos.

Nos encontramos ante una cifra similar cuando se intenta explorar el árbol de juego desde el primer movimiento. Este número ya fue calculado por Claude Shannon y considera que el factor de ramificación en media es más o menos 30, y el número de movimientos en media realizados por cada jugador 40. De ahí obtenemos $(30\times30)^{40}$ ó 10^{120} como el número de posibles jugadas a comprobar recorriendo el árbol de juego completo, un número que en primera instancia puede parecer no ser demasiado grande pero que, en cambio, es realmente alto teniendo en cuenta que la cantidad aproximada de átomos que se estima hay en el universo es del orden de 10^{79}.

Shannon definió en sus artículos cómo se podía usar MiniMax, del que luego se analizará más profundamente su funcionamiento, con una función de evaluación y estableció el rumbo para los futuros trabajos realizados sobre la inteligencia artificial en ajedrez proponiendo dos estrategias posibles para hacerlo:

- Mediante la búsqueda de árbol MiniMax usando la fuerza bruta con una función de evaluación.
- Usando un "generador de movimientos plausibles" en lugar de sólo las reglas del juego para encontrar un pequeño subconjunto de movimientos siguientes en cada capa durante la búsqueda de árboles.

Los futuros programas de juego de ajedrez se clasificarían a menudo como "tipo A" o "tipo B" según la estrategia en la que se basen principalmente.

Shannon especificó que la primera estrategia era la más simple pero no práctica, ya que el número de estados crece exponencialmente con cada capa adicional y el número total de posiciones posibles se vuelve intratable. Para la segunda estrategia, Shannon se inspiró en los jugadores maestros de ajedrez, que consideran selectivamente sólo movimientos prometedores. Sin embargo, un buen "generador de movimientos plausibles" no es en absoluto trivial a la hora de diseñarlo, por lo que la búsqueda a gran escala como en la estrategia "tipo A" sigue siendo de gran utilidad. Por ejemplo, *Deep Blue*[38] (el programa que venció al campeón del mundo de ajedrez Gary Kasparov) fue en esencia una combinación de ambos enfoques.

Debido a la poca capacidad computacional existente en ese momento, el primer programa de ajedrez fue ejecutado no con tubos de silicio o de vacío, ni cualquier tipo de computadora digital: Alan Turing, un matemático y pionero de la IA, pasó años trabajando en un algoritmo de Ajedrez para el Ferranti Mark 1 que completó en 1951 y al que llamó TurboChamp.

TurboChamp no era tan extenso como los sistemas propuestos por Shannon, y era muy básico comparándolo con los estándares futuros, pero aun así, era capaz de jugar al ajedrez. En 1952, Turing ejecutó manualmente el algoritmo tomando alrededor de

[38] COLOMINA, O. *et al*. "Aprendiendo mediante juegos: experiencia de una competición de juegos inteligentes", en *Actas de las X Jornadas de Enseñanza Universitaria de la Informática*, Universidad de Alicante, 2004. En línea: http://www.dccia.ua.es/~company/Otelo.pdf [Fecha de consulta: 18/02/2017].

hora y media en efectuar cada movimiento y en la que finalmente el programa perdió frente al contrincante humano. Sin embargo, Turing también publicó sus pensamientos sobre Ajedrez AI y postuló que en principio un programa que podría aprender de la experiencia y el juego en el nivel de los seres humanos debería ser completamente posible. Sólo unos años más tarde, el primer programa de ajedrez de ordenador sería ejecutado.

Todo esto sucedió antes de que realmente naciera el concepto de AI como tal —el campo de la Inteligencia Artificial—. En principio, este hecho puede datarse en la Conferencia de Dartmouth de 1956, donde hubo una sesión de *brainstorming* de un mes de duración entre las mentes más brillantes, donde el término "Inteligencia Artificial" fue acuñado. Además de los matemáticos universitarios e investigadores presentes (entre ellos Claude Shannon), también había dos ingenieros de IBM: Nathaniel Rochester y Arthur Samuel. Nathaniel Rochester encabezó un pequeño grupo que comenzó una larga tradición de gente en IBM logrando avances en IA, con Arthur Samuel siendo el primero.

Primeros pasos con MiniMax

Samuel había estado pensando en el Aprendizaje Automático (algoritmos que permiten a las computadoras resolver problemas mediante el aprendizaje en lugar de soluciones humanas codificadas a mano) desde 1949, y se centró especialmente en desarrollar una IA que pudiera aprender a jugar al juego de Damas. Damas, que tiene 10^{20} posiciones de tablero posibles, es más simple que el ajedrez (10^{47}) o Go (10^{250}) pero aún lo suficientemente complicado que no es fácil de dominar. Con las computadoras lentas e incómodas de la época, Damas fue un buen primer objetivo. Trabajando con los recursos que tenía en IBM, y particularmente su primera computadora comercial (la IBM 701), Samuel desarrolló un programa que podía jugar el juego de Damas, el primer juego dotado de inteligencia ejecutado en una

computadora. Resumió sus logros en su artículo "Some studies in machine learning using the game of Checkers[39]":

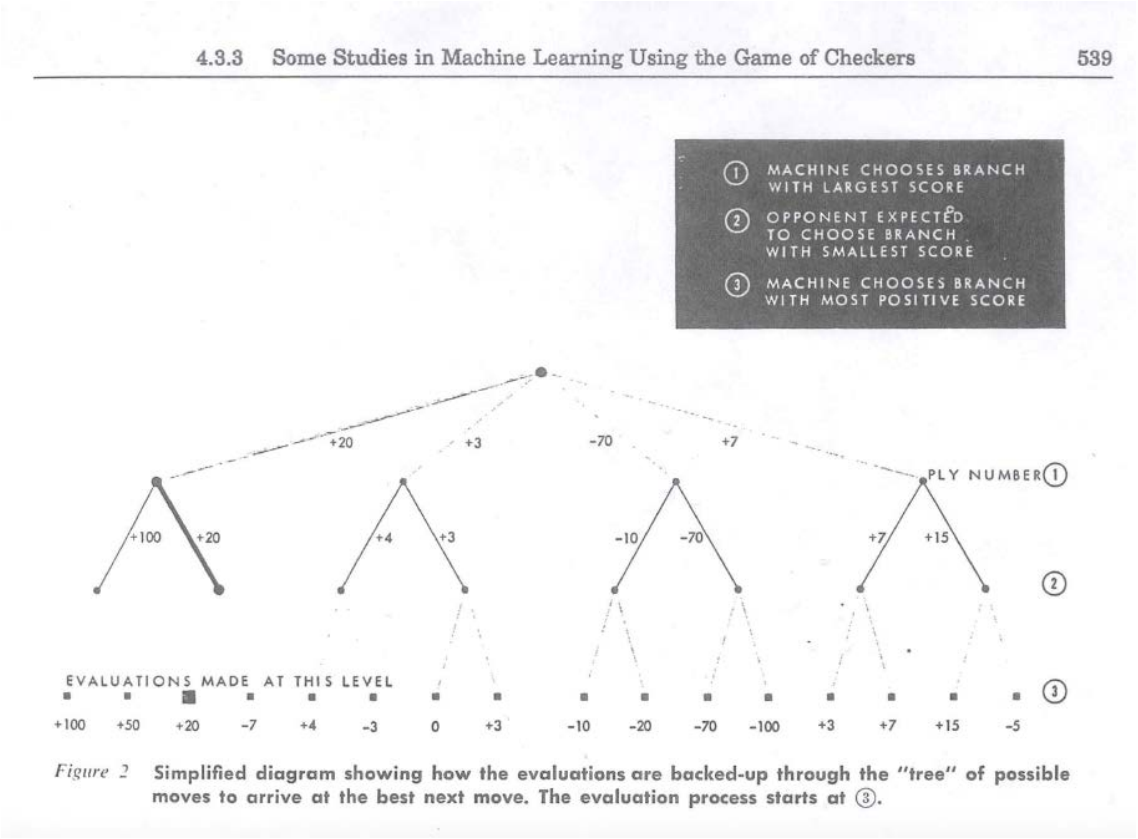


Figure 2 **Simplified diagram showing how the evaluations are backed-up through the "tree" of possible moves to arrive at the best next move. The evaluation process starts at ③.**

Figura 3. Diagrama que muestra cómo los valores son propagados desde las hojas del árbol hasta el primer nodo para encontrar el mejor movimiento siguiente. Fuente: Arthur Samuel.

Fundamentalmente, el programa se basaba en MiniMax, pero tenía un aspecto adicional muy importante que era el aprendizaje. El programa era capaz de mejorar con el tiempo sin la intervención humana directa, a través de dos métodos:

- "rote-learning", que significa memorización, proporciona la capacidad de memorizar los valores de determinadas posiciones previamente evaluadas con MiniMax, y por lo tanto no gasta recursos computacionales considerando movimientos ya comprobados anteriormente.

[39] SAMUEL, Arthur L. "Some Studies in Machine Learning Using the Game of Checkers", en *IBM Journal*, vol. 3, n. 3, julio 1969. Accesible en línea: https://www.cs.virginia.edu/~evans/greatworks/samuel1959.pdf [Fecha de consulta: 15/01/2017]

- "aprendizaje por generalización", es decir, modificando los multiplicadores para diferentes parámetros (modificando así la función de evaluación) sobre la base de los juegos anteriores jugados por el programa. Los multiplicadores fueron cambiados para reducir la diferencia entre la bondad calculada de una posición dada del tablero (según la función de la evaluación) y su valor real (calculada a través de jugar el juego hasta el final).

Rote-learning fue un método bastante bueno para hacer el programa más eficiente y capaz con el tiempo y demostró que podía funcionar bien. Pero fue el aprendizaje por generalización el que fue particularmente innovador, ya que demostró que un programa podía aprender a "intuitivamente" saber cuán buena era una posición de juego sin necesidad de simulación de movimientos futuros. Y no sólo eso, sino que el programa fue hecho para aprender jugando con versiones anteriores de sí mismo.

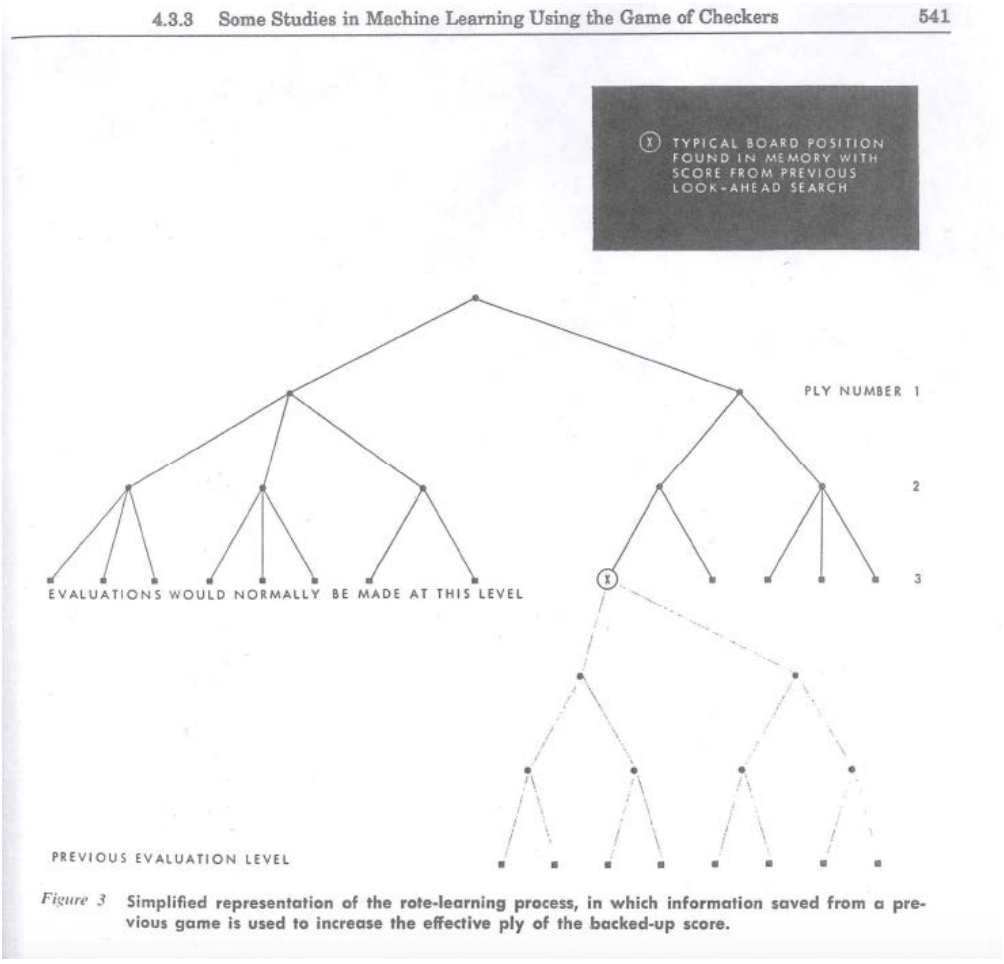


Figura 4. Representación simplificada de método rote-learning. Fuente: Arthur Samuel

Estas ideas no sólo fueron propuestas teóricamente innovadoras, sino que también se llevaron a la práctica. El programa era capaz de jugar con un nivel respetable a las Damas, lo cual no era nada fácil dada la limitada potencia de cálculo de la época. Y así, como detalla esta gran retrospectiva, cuando el programa de Samuel fue demostrado por primera vez en los primeros días de la IA (en el mismo año que la Conferencia de Dartmouth, de hecho), provocó una gran impresión:

Figura 5. Jugando a las damas con el ordenador IBM 701. Fuente: http://www-03.ibm.com

El 24 de Febrero de 1956 el programa de damas de Arthur Samuel, que fue desarrollado para jugar en el IBM 701, se probó públicamente en la televisión. En 1962, el maestro de las damas Robert Nealey jugó contra el ordenador en un IBM 7094 en una partida en la que el ordenador resultó victorioso. En otras partidas el ordenador fue derrotado, pero aun así es considerado un hito para la inteligencia artificial ofreciendo al público general una visión de la capacidad que tenían los ordenadores[40].

El reto del ajedrez

Aunque este hecho fue importante, no dejaba de ser un juego que no se consideraba de una complejidad alta: el juego realmente desafiante en ese momento era el ajedrez. Una vez más, fueron los empleados de IBM los pioneros en el primer ajedrez dotado de inteligencia artificial. El trabajo fue dirigido principalmente por Alex Bernstein, un matemático y experimentado jugador de ajedrez. Al igual que Samuel, decidió explorar el problema por interés personal y, en última instancia, llevó a la implementación de un juego de ajedrez totalmente funcional para el IBM 701, que se completó en 1957. El programa también utilizó MiniMax, pero en este caso carecía de cualquier capacidad de aprendizaje y se vio obligado a explorar únicamente los 4 movimientos por delante, y considerar sólo 7 opciones por movimiento. Hasta los años 70, la mayoría de los programas de ajedrez estarían limitados de forma similar, con alguna excepción en la que se incluía alguna lógica extra para elegir los movimientos a simular, como la estrategia de tipo B esbozada por Shannon en 1949. El programa de Bernstein tenía algunas heurísticas sencillas para seleccionar los 7 mejores movimientos para simular, lo que en sí era una nueva contribución. Sin embargo, estas limitaciones supusieron un juego muy precoz de ajedrez.

Es más, aun con todas estas limitaciones, fue el primer programa completamente funcional de ajedrez y demostró que incluso con una búsqueda MiniMax extremadamente limitada, con una función de evaluación simple y ningún aprendizaje, puede lograrse una jugabilidad de ajedrez aceptable.

A finales de la década de 1950, los ingenieros de IBM no eran los únicos que trabajaban en el desarrollo de la inteligencia artificial. Era un campo muy atractivo que trajo consigo la creación de nuevos grupos de investigación. Uno de esos grupos estaba compuesto por Allen Newell y Herbert Simon (ambos asistentes de la Conferencia de Dartmouth) de la Universidad Carnegie Mellon, y Cliff Shaw de RAND Corporation.

[40] The IBM 700 Series. http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/ibm700series/impacts/ [Fecha de consulta: 21/02/2017]

Trabajaron en el General Problem Solver (Solucionador General de Problemas), un programa de ordenador creado en 1957 con el objetivo de construir una máquina capaz de resolver problemas de carácter general. Cualquier problema simbólico formal podía ser resuelto, en principio, por el GPS. Por ejemplo: probar teoremas, resolver problemas geométricos, trabajar con lógica proposicional y jugar al ajedrez.

En esta última tarea estuvieron colaborando de 1955 a 1958, culminando en el trabajo "Chess Playing Programs and the Problem of Complexity", donde resumieron la investigación previa que había hasta el momento relacionada con el ajedrez AI y aportaron nuevas ideas que probaron con el programa de ajedrez NSS (Newell, Shaw y Simon[41]).

TABLE 1 Comparison of Current Chess Programs

	Turing	Kister, Stein, Ulam, Walden, Wells (Los Alamos)	Bernstein, Roberts, Arbuckle, Belsky (Bernstein)	Newell, Shaw, Simon (NSS)
Vital statistics				
Date	1951	1956	1957	1958
Board	8 × 8	6 × 6	8 × 8	8 × 8
Computer	Hand simulation	MANIAC-I 11,000 ops./sec	IBM 704 42,000 ops./sec	RAND JOHNNIAC 20,000 ops./sec
Chess program				
Alternatives	All moves	All moves	7 plausible moves Sequence of move generators	Variable Sequence of move generators
Depth of analysis	Until dead (exchanges only)	All moves 2 moves deep	7 plausible moves 2 moves deep	Until dead Each goal generates moves
Static evaluation	Numerical Many factors	Numerical Material, mobility	Numerical Material, mobility Area control King defense	Nonnumerical Vector of values Acceptance by goals
Integration of values	Minimax	Minimax (modified)	Minimax	Minimax
Final choice	Material dominates Otherwise, best value	Best value	Best value	1. First acceptable 2. Double function
Programming				
Language		Machine code	Machine code	IPL-IV, interpretive
Data scheme		Single board No records	Single board Centralized tables Recompute	Single board Decentralized List structure Recompute
Time	Minutes	12 min/move	8 min/move	1–10 hr/move (est.)
Space		600 words	7000 words	Now 6000 words, est. 16,000
Results				
Experience	1 game	3 games (no longer exists)	2 games	0 games Some hand simulation
Description	Loses to weak player Aimless Subtleties of evaluation lost	Beats weak player Equivalent to human with 20 games experience	Passable amateur Blind spots Positional	Good in spots (opening) No aggressive goals yet

45

Figura 6. Comparación de programas de ajedrez. Fuente: Newell, Shaw, Simon

[41] NEWELL, Allen, SHAW, Cliff, SIMON, Herbert. "Chess Playing Programs and the Problem of Complexity", en *IBM Journal of Research and Development*, Vol. 4, No. 2, 1958, pp. 320-335.

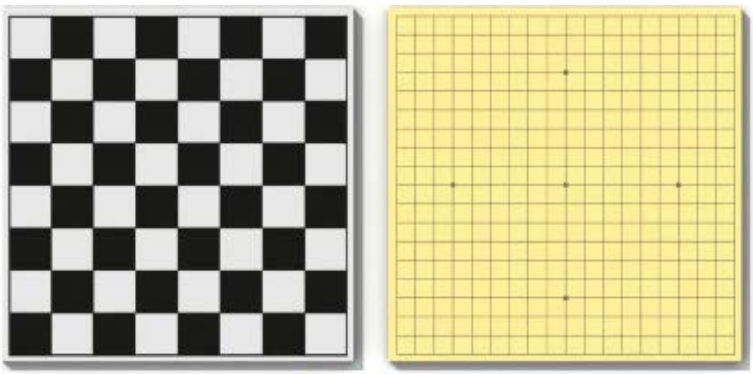
Newell, Shaw y Simon consideraron la heurística como un aspecto muy importante en su juego de ajedrez. Al igual que el programa de Bernstein, el algoritmo NSS utilizó un tipo de inteligencia simple para elegir qué movimientos explorar. La contribución más significativa del grupo a MiniMax fue la inclusión de algo que se convirtió en una parte esencial de los futuros programas que hacían uso de MiniMax: la poda alpha-beta. Ésta consiste en que el algoritmo evite la simulación de movimientos que de antemano se ve que son improductivas ("podando" las ramas del árbol que no es necesario simular), ahorrando así los preciosos recursos de computación para más prometedores movimientos permitiendo alcanzar mayor profundidad en el árbol de búsqueda.

A causa de las limitaciones del hardware y del código el resultado fue en un contrincante de ajedrez bastante malo. Sin embargo, a pesar de todo tiene constancia de que es el primer programa de ajedrez capaz de batir a un jugador casi inexperto, en el caso del NSS fue una secretaria a la que se le habían enseñado las reglas del juego una hora antes de la partida.

Para entonces el programa de Arthur Samuel seguía jugando bien a Damas y continuaba mejorando. En 1962, Samuel e IBM tenían suficiente fe en el programa para lanzarlo públicamente contra un rival de un gran nivel. El oponente elegido fue Robert Nealy, que se consideraba un maestro, pero no se llegó a clasificar como un gran jugador en ninguna competición. En parte debido a esto, y en parte porque el programa era bueno en Damas, Nealy perdió. Aunque pronto quedaría claro que el programa de Samuel no era rival para los mejores jugadores humanos en el juego ya que fue fácilmente superado por dos de ellos en el campeonato mundial de 1966, la reacción del público y los medios de comunicación a su victoria en 1962 no tuvo nada que envidiar a la de otros eventos históricos.

Mientras tanto, más equipos de investigación en la Unión Soviética y en los Estados Unidos continuaban trabajando en la implementación de un programa de Ajedrez. Un grupo de estudiantes del MIT liderado por John McCarthy desarrolló un programa de juego de ajedrez basado en MiniMax con poda alpha-beta y en 1966 se enfrentó a un programa desarrollado en el Instituto de Física Teórica y Experimental de Moscú (ITEP)

por telegrama. El programa Kotok-McCarthy perdió 3-1, y en general fue muy débil debido a que se limitó a buscar muy pocas posiciones (menos que el programa de Bernstein, incluso). Pero otro estudiante llamado Richard Greenblatt vio el programa y, siendo un hábil jugador de ajedrez, se inspiró para escribir el suyo propio —el Mac Hack—. Este programa buscó a través de muchas más posiciones y tuvo otros refinamientos, hasta el punto de vencer a un jugador humano clasificado en un torneo en 1967 y ganar o empatar varias veces más en sucesivos torneos. Pero seguía sin estar a la altura de los mejores jugadores.



GRID SIZE	
8 x 8	19 x 19

AVERAGE NUMBER OF MOVE CHOICES PER TURN	
35	200–300

LENGTH OF TYPICAL GAME	
60 moves	200 moves

NUMBER OF POSSIBLE GAME POSITIONS	
10^{44}	10^{170}

EXPLOSION OF CHOICES
(starting from average game position)

35	Move 1	200
1225	Move 2	40 000
42 875	Move 3	8 000 000
1 500 625	Move 4	1 600 000 000

Figura 7. Cracking Go. Fuente: Bryan Christie Design[42]

[42] Cracking Go. Chess vs. Go: http://spectrum.ieee.org/computing/software/cracking-go/chess-vs-go [Fecha de consulta: 15/01/2017]

Entonces, en 1968, Go alcanzó el hito que fue conquistado para el ajedrez unos diez años antes: consiguió superar a un aficionado inexperto. El programa no usaba árboles de búsqueda, sino que se basaba en emular la forma en que un jugador humano percibe la representación interna de una posición de juego en Go para reconocer patrones importantes a la hora de elegir el movimiento correcto. Esta hazaña fue realizada por Alfred Zobrist, como se describe en "A model of visual organization for the game of Go[43]".

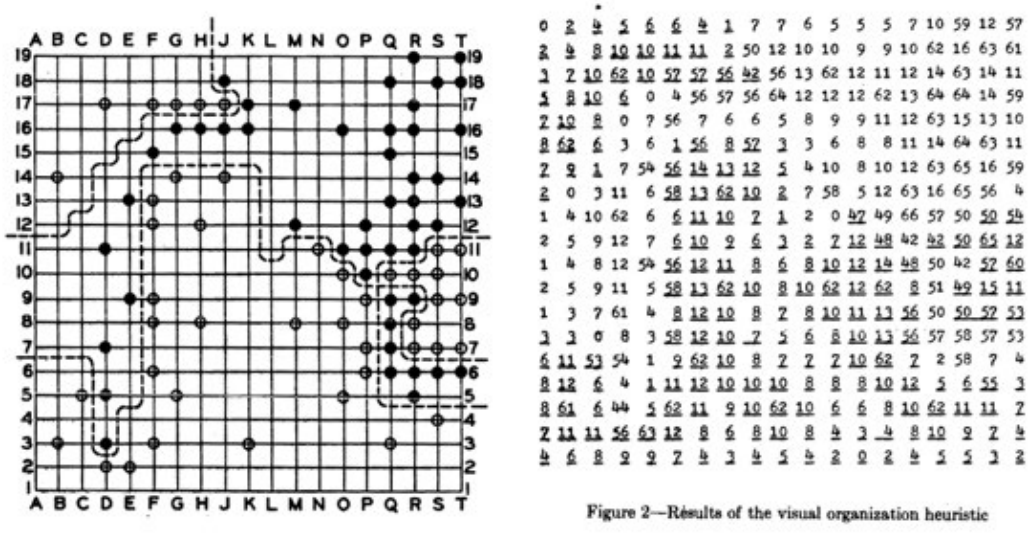


Figure 2—Results of the visual organization heuristic

Figura 8. Resultado de la organización visual propuesta por Zobrist. Fuente: Alfred Zobrist

> «The visual nature of Go
>
> The recognition and discrimination of meaningful perceptual stimuli presupposes the active formation of stable perceptual elements to be recognized and discriminated. A person lacking this process would combine all sorts of stimuli into meaningless groups.
>
> The choice of a move in Go usually involves the recognition of configurations which are meaningful to the player. This raises the question as to whether

[43] ZOBRIST, Albert L. "A model of visual organization for the game of Go", en *Sprint Joint Computer Conference*, 1969. En línea: https://www.computer.org/csdl/proceedings/afips/1969/5073/00/50730103.pdf [Fecha de consulta: 15/01/2017]

> certain basic perceptual processes are necessary for the comprehension of a Go board. The following examples might suggest that the answer is yes. First, consider the spontaneous grouping of stones of the same color which occurs during visualization of a Go board. The stones are organized into distinct groups, clusters, or armies even though they may be sparsely scattered about or somewhat intermingled. Grouping is usually the result of proximity of stones of the same color or the predominance of stones of one color in an area, but can be affected by other characteristics of the total board situation. For example, stones which fall into a line are likely to be grouped. Kohler and others have found grouping to be a basic perceptual phenomenon. Yet the recognition and discrimination of groups or armies is necessary for competent Go play».

Debido a la complejidad intrínseca del propio juego, al enorme factor de ramificación y las complicadas heurísticas, el progreso de los programas de Go fue mucho más lento que para el Ajedrez o Damas. Tendría que pasar otra década hasta que Bruce Wilcox desarrollara un programa más fuerte, una vez más sin depender de las técnicas tradicionales de AI. El enfoque era subdividir el tablero en regiones más pequeñas que eran más fáciles de tratar, lo cual se convirtió en una característica de los siguientes programas de Go. Pero incluso entonces, no estaba ni remotamente cerca de ser un contrincante aceptable.

Por otro lado los programas de ajedrez progresaban de una manera excelente. Por ejemplo, a principios de los años 70, el grupo de ajedrez AI de ITEP perfeccionó su programa en una versión mejor que nombró Kaissa, que se convirtió en el primer campeón de ajedrez de computación del mundo en 1974 después de cuadrar frente a los programas de EE.UU. El programa se benefició significativamente de las computadoras más rápidas y de una implementación eficiente que incluyó la poda alpha-beta y algunos otros trucos, mostrando por primera vez la fuerza de la estrategia de tipo AI de Shannon que se basaba más en la búsqueda rápida que la heurística inteligente o la evaluación de posición.

El primer programa informático capaz de dominar completamente a los humanos en un juego complejo no se desarrolló hasta aproximadamente 3 décadas después de que el programa de damas de Samuel ganase ese juego contra Robert Nealy, y fue el programa de damas CHINOOK. El programa fue desarrollado por un equipo de la Universidad de Alberta dirigido por Jonathan Schaeffer, a partir de 1989. En 1994 el mejor jugador de Damas en el planeta sólo logró empatar contra CHINOOK.

Figura 9. Marion Tinsley. Fuente: http://afflictor.com

El campeón invicto y matemático Marion Tinsley fue afortunado de conseguir una victoria después de perder dos veces a las damas contra Chinook, el programa del canadiense Jonathan Schaefer[44].

Ya en la década de los 90', la capacidad computacional aumentó exponencialmente y el abanico de posibilidades que se abría, en comparación con las computadoras de los años 70', era inmenso. Esto permitió dotar de innovadoras técnicas a CHINOOK:

- Una base de datos con los movimientos apertura de los grandes maestros.

[44] "Within The Decade, The Computer Will Know How The Game Will Turn Out Even Before It Begins", en Afflictor.com . http://afflictor.com/2015/12/16/within-the-decade-the-computer-will-know-how-the-game-will-turn-out-even-before-it-begins/ [Fecha de consulta: 17/02/2017]

- Un árbol de búsqueda usando alpha-beta con una función de la evaluación basada en una combinación linear de diferentes características.
- Una base de datos con todas las posibles posiciones finales del tablero con menos de ocho piezas.

Figura 10. Deep Thought. Fuente: Chessprogramming.wikispaces.com

Deep Thought[45] fue un ordenador especialmente fabricado para jugar al ajedrez, construido en la universidad de Carnegie Mellon en la década de los 80'. Fue el predecesor de Deep Blue. El proyecto fue desarrollado inicialmente por Feng-hsiung Hsu y Thomas Anantharaman. El nombre del programa fue debido al supercomputador de ficción que tenía el mismo nombre.

Volviendo al ajedrez Feng-hsiung Hsu reunió un equipo para desarrollar Deep Thought, incorporó todas estas ideas y tuvo dos puntos fuertes notables: hardware personalizado y extensiones selectivas inteligentes. De acuerdo a un estudio realizado, fue el programa de ajedrez más potente hasta ese momento en términos de cuántas posiciones era capaz de considerar por segundo.

[45] Deep Thought: https://chessprogramming.wikispaces.com/Deep+Thought [Fecha de consulta: 17/02/2017]

Esto se logró mediante la simulación y evaluación de movimientos con placas de circuito personalizadas, que trabajaban en conjunto con el software. Además de ser rápido, Deep Thought también era inteligente: realizaba una búsqueda selectiva más allá de la profundidad por defecto en posiciones prometedoras. Esto permitió que la profundidad de búsqueda se ampliara considerablemente: “El resultado fue que, en promedio, una búsqueda de profundidad N alcanzaba a lo largo de la variación principal una profundidad de entre 1,5 N y 3 N”.

Así, Deep Thought tuvo un gran éxito ya que combinaba la fuerza bruta “tipo A” (buscando todas las posiciones hasta una cierta profundidad) y la búsqueda selectiva “tipo B” (buscando más allá de esa profundidad en ciertos casos). En 1988, Deep Thought se convirtió en el campeón de ajedrez de computadoras del mundo y, más impresionantemente, venció al gran maestro de ajedrez Bent Larsen.

Otro aspecto innovador de Deep Thought fue que su función de evaluación estaba sincronizada automáticamente con una base de datos de juegos entre maestros de ajedrez, en lugar de tener todos los parámetros de la función previamente programados. Aunque los programas de Ajedrez mejoraron a lo largo de las décadas debido al aumento de la capacidad computacional e ideas como la poda alpha-beta y extensiones selectivas, casi todos los programas no tenían ningún componente de aprendizaje y recibían toda su inteligencia de sus creadores humanos. Deep Thought rompió esta tendencia.

Aun así, esta comunicación con la base de datos seguía dependiendo del conocimiento humano existente sobre el propio juego. Esto es un problema, ya que la parte más difícil de jugar un juego (cómo evaluar posiciones y movimientos selectos) al final depende del conocimiento humano y no del programa en sí mismo. Esto puede no considerarse como una inteligencia artificial auténtica, ya que lo ideal sería poder escribir programas de AI que aprendiesen estas cosas por sí mismos.

Redes neuronales

Esto se consiguió con el desarrollo de las redes neuronales. Las redes neuronales son una técnica para el aprendizaje supervisado de máquinas, que son aquellos algoritmos que pueden aprender a producir una salida deseada para algún tipo de entrada a partir del aprendizaje con muchos ejemplos de entrenamiento de pares de entrada/salida conocidos del mismo tipo. Un ejemplo de aplicación de las redes neuronales a los juegos es una versión del backgammon que incorporaba inteligencia, cuyo nombre es Neurogammon[46].

Al igual que en Go, el factor de ramificación de Backgammon es enorme y el uso de árboles de búsqueda tradicionales no resulta de lo más adecuado. Un gran factor de ramificación hace imposible alcanzar una buena profundidad en el árbol, y escribir una función de evaluación que compense esto no es trivial. Gerald Tesauro y Terrence Sejnowski exploraron un enfoque basado en el aprendizaje de una buena función de evaluación. Como se explicó en su artículo de 1989 "A parallel network that learns to play backgammon[47]", entrenaron una red neuronal que tomaba como entrada una posición de juego de backgammon y un posible movimiento potencial, y daba como salida una puntuación de la calidad de ese movimiento. Este enfoque elimina la necesidad de que los ingenieros tengan que codificar el comportamiento humano al escribir el programa, lo cual es más acorde a una inteligencia artificial.

[46] TESAURO, G. "Neurogammon: a neural-network backgammon program", en *Neural Networks*, 1990 IJCNN International Joint Conference, 1990.
[47] TESAURO, G., SEJNOWSKI, T.J. "A Parallel Network that Learns to Play Backgammon", en *Artificial Intelligence,* 39, 1989, pp. 357-390. En línea: http://papers.cnl.salk.edu/PDFs/A%20Parallel%20Network%20That%20Learns%20to%20Play%20Backgammon%201989-2965.pdf [Fecha de consulta: 18/01/17]

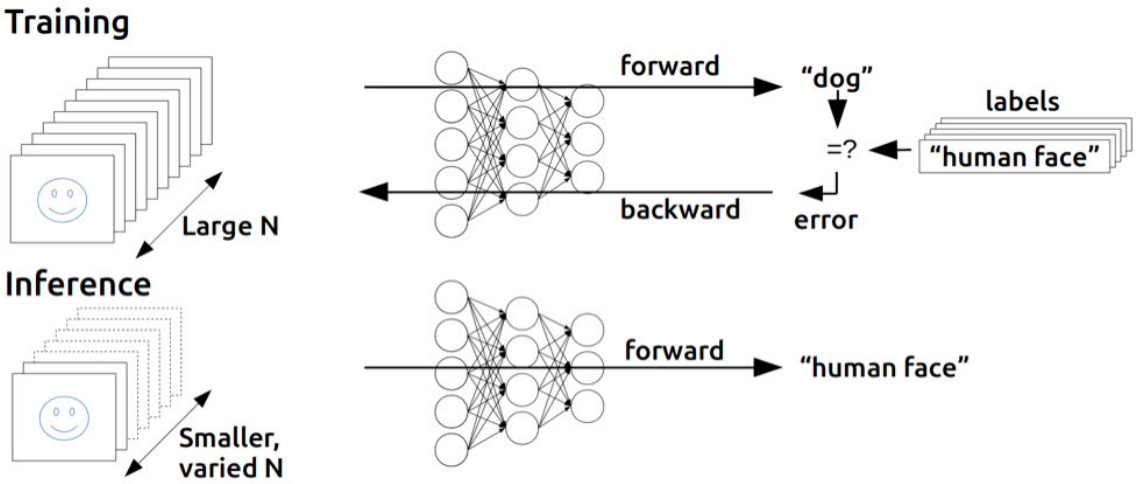


Figura 11. Entrenamiento comparado con inferencia. Fuente: nvidia.com

En entrenamiento[48] muchas entradas, a menudo en grandes remesas de datos son usados para entrenar una red neuronal. En inferencia, la red entrenada es usada para descubrir información que llega a través de pequeños lotes. Neurogammon trabajaba con posiciones del tablero como entrada y puntuaciones para cada posición como salida.

Después de diversas actualizaciones apareció la versión 1.0 de Neurogammon cuyo principal propósito era ganar a todos los demás programas de la Primera Olimpiada de Informática de 1989. Sin embargo, no era tan fuerte como los mejores jugadores humanos, algo que conseguiría otro programa basado en la red neuronal de Gerald Tesauro: TD-Gammon.

Presentado en 1992, TD-Gammon fue una aplicación muy exitosa de aprendizaje por refuerzo. A diferencia del aprendizaje supervisado, que trabaja con tipos particulares de entradas y salidas, el aprendizaje por refuerzo trata de encontrar opciones óptimas en diferentes situaciones. Más concretamente, pensamos en términos de estados (situaciones), en los que un agente (el programa) puede tomar acciones que cambien el estado del agente de una manera conocida (opciones). Cada transición entre estados resulta en una "recompensa" numérica, y se trata de determinar la acción

[48] Inference: The Next Step in GPU-Accelerated Deep Learning: https://devblogs.nvidia.com/parallelforall/inference-next-step-gpu-accelerated-deep-learning/ [Fecha de consulta: 03/03/2017]

correcta a tomar en un estado dado para obtener la recompensa más alta en el largo plazo. Mientras que el aprendizaje supervisado aprende a aproximar una función a través de ejemplos de entradas y salidas, el aprendizaje por refuerzo generalmente aprende de la "experiencia" de recibir recompensas después de intentar acciones en diferentes estados.

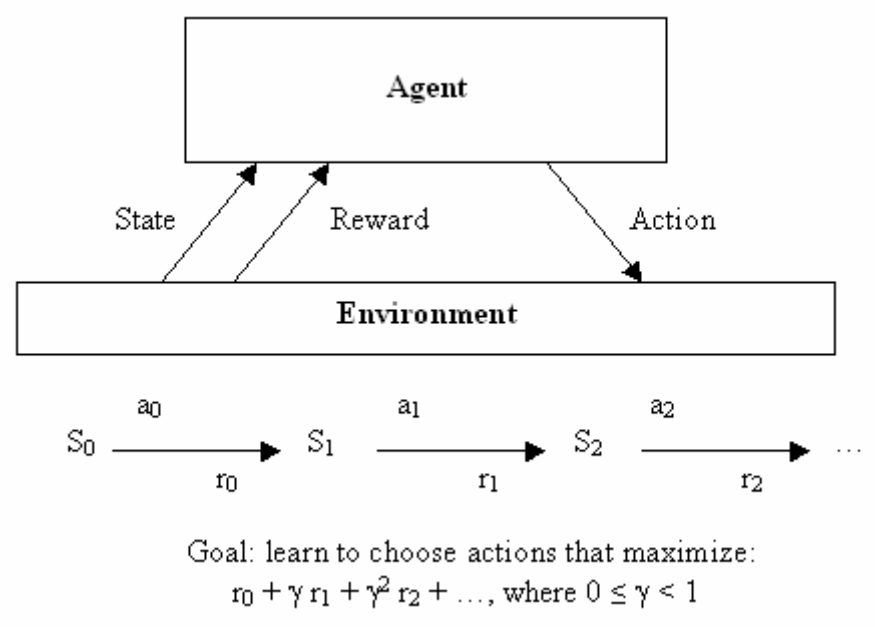


Figura 12. TD-Gammon. Fuente: G. Tesauro

Así, TD-Gammon aprendió jugando contra versiones anteriores de sí mismo, observando qué jugador ganó, y utilizando esa experiencia para afinar la red neuronal. Esto es muy diferente de como lo hacía Neurogammon, el cual requería el tratamiento de cientos de movimientos con puntuaciones asignadas por el ser humano y, por lo tanto, era mucho más engorroso que dejar que el programa jugase contra sí mismo durante horas. De hecho, el tipo de aprendizaje por refuerzo de TD-Gammon se basa en el aprendizaje por diferencia temporal desarrollado en 1986 por Richard Sutton como una formalización del aprendizaje en el trabajo de Samuel.

Con sólo las posiciones de tablero en bruto como entrada y sin aprovecharse de ningún conocimiento humano sobre el juego, TD-Gammon alcanzó un nivel de juego comparable al Neurogammon. Y con la adición de las características de Neurogammon, se hizo comparable a los mejores jugadores humanos en el mundo.

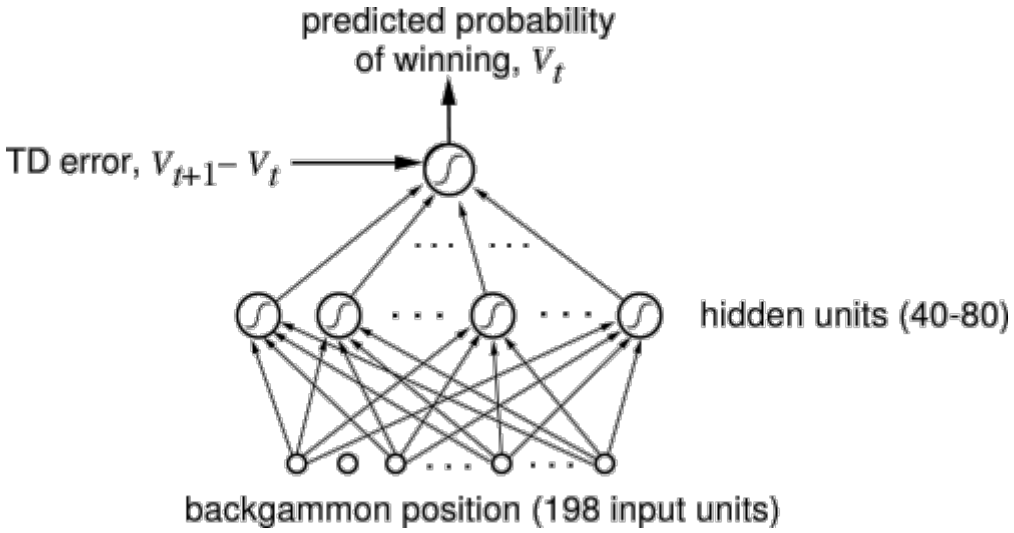


Figura 13. TD-Gammon (II). Fuente: G. Tesauro

TD-Gammon es hoy un hito en la historia de la inteligencia artificial. Pero, cuando se trató de usar el mismo enfoque para otros juegos, los resultados no fueron tan impresionantes. NeuroChess (1995) de Sebastian Thrun sólo era comparable a los programas comerciales de ajedrez en un entorno de dificultad baja, y NeuroGo (1996) de Markus Enzenberger tampoco coincidía con la habilidad de los Go existentes. En el caso de NeuroChess, se supuso que la discrepancia era en gran parte debido al gran tiempo que tomó para calcular la función de evaluación ya que el cálculo de una función de red neuronal grande toma dos órdenes de magnitud más que una función de evaluación lineal optimizada, haciendo que NeuroChess fuese incapaz de explorar tantos movimientos como el programa comercial de Ajedrez. El hecho de tener una función de evaluación mejor no supuso una ventaja frente a una función más simple que era capaz de explorar muchas más posiciones posibles…

… Lo que nos devuelve al Deep Thought. Después del éxito del primer programa, parte del mismo equipo fue contratado por IBM y se propuso la creación de Deep Thought II, que más tarde fue rebautizado como Deep Blue (Deep Thought x Big Blue = Deep Blue). En términos generales, Deep Blue era conceptualmente lo mismo que Deep Thought, pero con una capacidad computacional increíblemente superior.

Primer programa prácticamente imbatible: Deep Blue

Se trataba de un supercomputador hecho a medida para jugar al ajedrez. Sin embargo, en su primer encuentro con Gary Kasparov en 1996 Deep Blue perdió 2-4. Tras un año mejorando todo el sistema el equipo hizo historia superando a Kasparov con una puntuación de 3.5-2.5 el 11 de mayo de 1997[49].

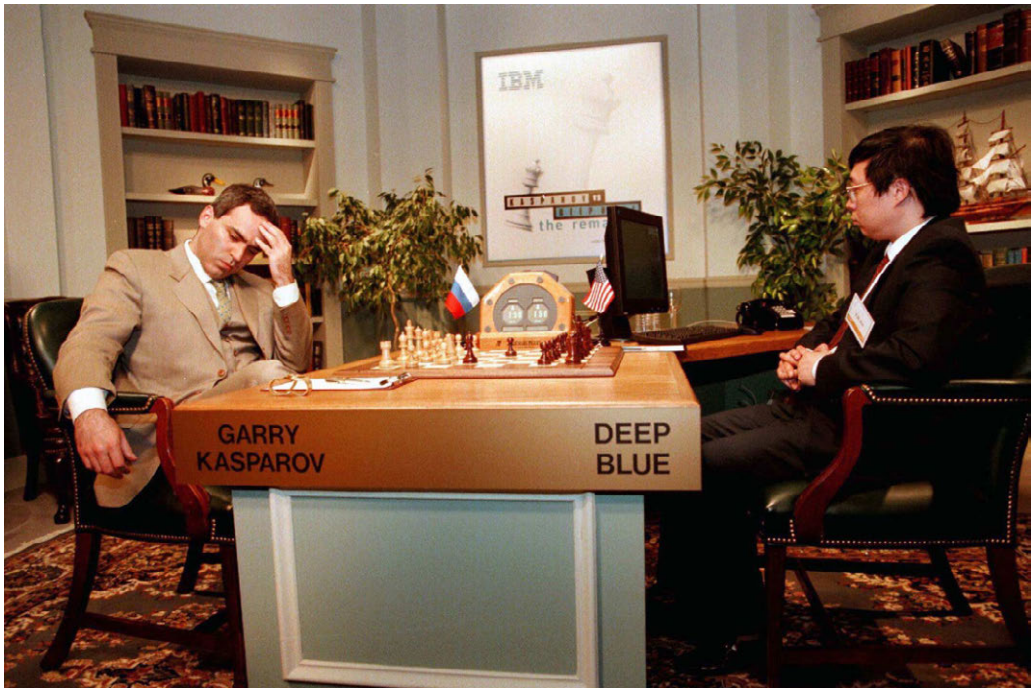


Figura 14. Gary Kasparov vs. Deep Blue. Fuente: Stanford.edu

El equipo señaló los distintos factores que contribuyeron a ese éxito, incluyendo:

1. Chips dedicados.
2. Un sistema masivo con múltiples niveles de paralelismo.
3. Un gran énfasis en las extensiones de búsqueda.
4. Una función de evaluación compleja.
5. Una base de datos con las jugadas de los grandes maestros.

Por lo tanto, no se puede afirmar que Deep Blue ganó puramente a través de la fuerza bruta, ya que incluía décadas de ideas y trabajo sobre cómo abordar el problema. Pero también es innegable que la fuerza bruta fue un factor decisivo. Deep Blue se ejecutó

con treinta procesadores dentro de una supercomputadora trabajando conjuntamente con 480 motores de búsqueda de ajedrez de un solo chip. Mientras jugaba con Kasparov observó 126 millones de posiciones por segundo en promedio, y buscó a una profundidad media de entre 6 y 12 plies y hasta un máximo de cuarenta. Todo esto le permitió ganar a duras penas, posiblemente debido a errores poco comunes por parte de Kasparov.

A finales de los 90 se podía considerar que juegos como el ajedrez y backgammon estaban dominados por la inteligencia artificial, pero no pasaba lo mismo con Go. Los mejores programas existentes no eran rival para cualquier aficionado que tuviese cierta experiencia en el juego.

Después del trabajo de los años 70 de Bruce Wilcox en los programas de Go, muchas personas continuaron dedicando su tiempo a implementar mejores programas de Go a lo largo de los años 80 y 90. Uno de los mejores programas de los años 90, *The Many Faces of Go*, alcanzó un rendimiento de 13 kyu (kyu es un vocablo japonés utilizado en las artes marciales japonesas y en otras prácticas tradicionales como son el ikebana, el juego del Go o la ceremonia del té para designar las diferentes etapas en la progresión de un debutante antes de la obtención del nivel de dan)[50].

Figure 2: Performance ranks in Go, in increasing order of strength from left to right.

Figura 15. Ranking en Go. Fuente: Sylvain Gelly et al.

[49] Deep Blue: http://stanford.edu/~cpiech/cs221/apps/deepBlue.html [Fecha de consulta: 18/03/2017]
[50] GELLY, Sylvain *et al*. "The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions", en *Communications of the ACM*, vol. 55, n. 3, marzo 2012, pp. 106-113. En línea: http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Applications_files/grand-challenge.pdf [Fecha de consulta: 23/03/2017]

David Fotland tardó más de una década en escribir las 30.000 líneas de código para implementar los componentes específicos de Go que utilizó. La combinación de un factor de ramificación más grande, juegos más extensos y posiciones de tablero más difíciles de evaluar, hicieron que los métodos usados para los programas invencibles de ajedrez no fuesen suficiente.

Afortunadamente, el uso de los algoritmos Monte Carlo supuso un cambio radical. La idea general de los algoritmos de Monte Carlo es simular un proceso, usualmente con algo de aleatoriedad, para obtener estadísticamente una buena aproximación de algún valor que es muy difícil de calcular directamente. Se reduce a comenzar con algunos valores aleatorios y semi-aleatoriamente cambiarlos gradualmente para obtener valores más óptimos. Si esto se hace durante el tiempo suficiente, se obtiene una solución cercana a la óptima que se puede encontrar con mucho menos cálculo[51].

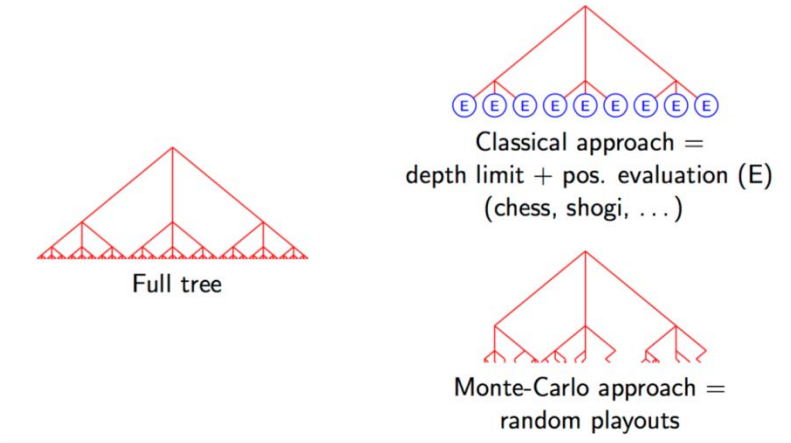


Figura 16. Algoritmos Monte Carlo. Fuente: Rémi Coulom

Las técnicas de Monte Carlo se habían aplicado desde los años 70 con los juegos de azar como el Blackjack, pero no fueron aplicadas para otros juegos como Go hasta 1993 con el *Monte Carlo Go* de Bernd Brügmann. La idea que presentó fue increíblemente simple: en vez de implementar una compleja función de evaluación y usar árboles de búsqueda, la idea es tener un programa que simule jugar muchos juegos al azar y escoja el movimiento que lleve al mejor resultado en promedio. Es una buena aproximación para el juego de Go ya que el factor de ramificación es

[51] COULOM, Rémi. "The Monte Carlo Revolution in Go". JJFoS Japanese-French Frontiers of Science Symposium. En línea: https://www.remi-coulom.fr/JFFoS/JFFoS.pdf [Fecha de consulta: 21/02/2017]

excesivamente grande y usando una función de evaluación que solo puede recorrer unos pocos movimientos hacia delante y no hasta el final del juego sería más costoso computacionalmente y los resultados no serían demasiado notables. Por lo tanto, resulta una gran alternativa utilizar ese tiempo de computación para jugar aleatoriamente un subconjunto de juegos posibles (conocidos habitualmente como *rollouts*), y luego evaluar los movimientos tomando como referencia las victorias y derrotas. A pesar de ser relativamente simple, el programa de Brügmann exhibió un buen nivel de principiante sin alejarse demasiado del mucho más complejo *Many faces of Go*.

White: Many Faces of Go
Black: Gobble

Strategy	A	A	B	B
Handicap	3	2	2	0 (no komi)
Random games	500	1000	5x400	4x500, 4x1000
1. Game	W by 4	B by 7	W by 25	W by 21
2.	W by 4	W by 41	B by 9	W by 11
3.	B by 1	W by 1	W by 9	B by 81
4.	W by 81	W by 81	B by 9	W by 11
5.	B by 22	W by 37	B by 17	W by 9

Table 1: Game results for the 9x9 board.

Figura 17. Resultados de 20 juegos con Gobble jugando como Negro con diferentes desventajas y Many Faces of Go jugando a nivel 10 (que equivaldría a un nivel medio). Fuente: Bernd Brügmann

A pesar de que fue un enfoque novedoso y que mostró una buena solución para abordar los problemas que presentaba el juego de Go, el trabajo de Brügmann[52] no se tomó en serio en un principio. Habría que esperar hasta el nuevo milenio cuando un grupo en París compuesto por Bruno Bouzy, Tristan Cazenave y Bernard Helmstetter comenzó a explorar seriamente el potencial de las técnicas de Monte Carlo. Aunque simplificaron y ampliaron el enfoque de Brügmann, sus programas aún no estaban a la altura de los más fuertes que usaban árboles de búsqueda como GNU Go.

[52] BRÜGMANN, Ben. *Monte Carlo Go*. *Technical report*, Physics Department, Syracuse University, 1993. En línea: http://www.ideanest.com/vegos/MonteCarloGo.pdf [Fecha de consulta: 22/02/2017]

- Rémi Coulom propuso el uso de las técnicas de Monte Carlo junto con los árboles de búsqueda, y acuñó el término como *Árboles de Búsqueda de Monte Carlo* (MCTS). Su programa *CrazyStone* ganó el torneo KGS computer-Go de ese año de las variantes de tablero 9x9 de Go, superando a otros programas como NeuroGo y GNU Go y demostrando así el potencial de MCTS.
- Levente Kocsis y Csaba Szepesvári desarrollaron el algoritmo UCT (Upper Confidence Bounds for Trees) cuyo propósito es marcar los límites entre la explotación (simulando movimientos que parecen prometedores para obtener una mejor estimación) y la exploración (intentando movimientos nuevos o malos para tratar de conseguir algo mejor) en MCTS.
- Por último, Sylvain Gelly junto con otros, combinaron MCTS y UCT, así como la búsqueda de patrones y poda de árboles. Resultando de su trabajo el programa MoGo que superó a CrazyStone y se convirtió en el mejor juego de Go hasta la fecha.

El rápido progreso de CrazyStone y MoGo fue bastante impresionante. En 2008, MoGo venció al jugador profesional de nivel 8 dan Kim Myungwan en un juego Go de 19x19. Esto para Myungwan suponía una gran desventaja ya que MoGo estaba funcionando en un superordenador de gran capacidad. En el mismo año, CrazyStone derrotó al profesional japonés de nivel 4 dan jugador Kaori Aoba ejecutándose en un PC normal.

Sin embargo, los programas de Go sólo estaban golpeando a los profesionales de bajo nivel. La revolucionaria idea del árbol de búsqueda de Monte Carlo permitió el uso de la fuerza bruta de una manera inteligente, y resolvió el uso de la inteligencia “tipo A”. Fue igual de decisivo que el uso de la poda alpha-beta en los programas de ajedrez, pero funcionó mejor para Go porque reemplazó las funciones de evaluación implementadas por humanos con muchos rollouts aleatorios del juego fáciles de computar paralelamente en diferentes procesadores. Pero para poder enfrentarse a los mejores seres humanos, también sería necesaria hacer uso de la inteligencia “tipo B” (emulación de instintos aprendidos de tipo humano). Esto pronto se desarrollaría, pero requeriría cambiar completamente el pensamiento y de ahí surgió el *Deep learning*.

Para tratar la inclusión del Deep learning en Go debemos remontarnos a los programas que se habían desarrollado en la década de los noventa haciendo uso de las redes neuronales. Neurogammon, por un lado, que usaba redes neuronales y aprendizaje supervisado y TD-Gammon, su sucesor que también hacía uso de redes neuronales pero basado en el aprendizaje por refuerzo. Estos programas demostraron que el aprendizaje automático era una alternativa viable al enfoque basado en la codificación manual de estrategias complejas.

De hecho, Nicol N. Schraudolph, Peter Dayan y Terrence J. Sejnowski explican en el documento "Temporal difference learning of position evaluation in the game of Go[53]" cómo aplicaron el enfoque de TD-Gammon a Go.

El grupo de Schraudolph advirtió del gran esfuerzo necesario para desarrollar un aprendizaje supervisado sobre Go (como el de Neurogammon) debido a la dificultad de generar suficientes ejemplos de tableros de Go. Por lo tanto, sugirieron usar un enfoque similar a TD-Gammon entrenando una red neuronal para evaluar una posición determinada del tablero, jugando contra sí mismo y contra otros programas. Sin embargo, el equipo descubrió que el uso de una red neuronal simple como en TD-Gammon era ineficiente, ya que no se considera el hecho de que muchas jugadas en un tablero de Go pueden girar o moverse y aun así mantenerse inalterado el valor de la jugada. Por ese motivo, decidieron utilizar una red neuronal convolucional (CNN), un tipo de red neuronal restringida para realizar el mismo cálculo para diferentes partes de la entrada. Normalmente, la restricción consiste en aplicar el mismo procesamiento a pequeñas partes de la entrada y, a continuación, tener un cierto número de capas adicionales de búsqueda de combinaciones específicas de patrones y, a continuación, utilizar esas combinaciones para calcular la salida global. La CNN en el trabajo de Schraudolph fue más simple, pero ayudó a su resultado mediante la explotación de rotación, reflexión y simetrías en la inversión de colores en Go.

[53] SCHRAUDOLPH, Nicol N., DAYAN, Peter, SEJNOWSKI, Terrence J. "Temporal Difference Learning of Position Evaluation in the Game of Go", en COWAN, J.D., TESAURO, G., ALSPECTOR, J. (eds.), *Advances in Neural Information Processing 6*, Morgan Kaufmann: San Francisco, 1994. En línea (pre-print): http://www.gatsby.ucl.ac.uk/~dayan/papers/sds94.pdf [Fecha de consulta:22/02/2017]

Las redes neuronales habían quedado en segundo plano en los años 2000, mientras que otros métodos de aprendizaje habían ganado fama y los programas de Go no lograban un buen nivel.

El paper "A fast learning algorithm for deep belief nets[54]" a menudo se toma como referencia con el resurgimiento de las redes neuronales sugiriendo un enfoque para la formación de redes neuronales "profundas" con muchas capas de unidades de cálculo. Este fue el comienzo de lo que se convertiría en el fenómeno del *deep learning*. Este documento fue publicado en 2006, el mismo año que los documentos MCTS.

Las redes neuronales profundas continuaron evolucionando y ganando adeptos en los años siguientes, y en 2009 lograron resultados récord en el reconocimiento del habla. Además de las mejoras en los algoritmos, la disponibilidad de grandes cantidades de datos para entrenamiento y la posibilidad del aprovechamiento de las capacidades computacionales masivas paralelas de las GPU fueron los factores que contribuyeron a su desarrollo. El evento que realmente marcó un punto de inflexión en la investigación y el desarrollo del deep learning fue el concurso ILSVRC (ImageNet Large Scale Visual Recognition Challenge) en 2012.

Hasta este punto, el deep learning se había aplicado a tareas de aprendizaje supervisadas no relacionadas con el juego, tales como categorizar una imagen dada o reconocer el habla humana. Esto cambió en 2013 cuando una compañía llamada DeepMind aprovechó una red neuronal para jugar a los juegos de Atari como explican en su publicación "Playing Atari with Deep Reinforcement Learning[55]". Presentaron un nuevo enfoque para el aprendizaje por refuerzo con redes neuronales profundas, que fue abandonado desde el fracaso de intentar usar el mismo enfoque de TD-Gammon para otros juegos. Con sólo la entrada de los píxeles de la pantalla, y la puntuación del juego, Deep Q-Networks aprendió a jugar Breakout, Pong y mucho otros. Este trabajo

[54] HINTON, Geoffrey E., OSINDERO, Simon, TEH, Yee-Whye. "A fast learning algorithm for deep belief nets", en *Neural Computation*, 2006. En línea: https://www.cs.toronto.edu/~hinton/absps/fastnc.pdf [Fecha de consulta: 22/02/2017]

[55] MNIH, Volodymyr *et al.* "Playing Atari with Deep Reinforcement Learning". En línea: https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf [21/02/2017]

tuvo tal impacto que en 2014 Google pagó 400 millones de dólares para adquirir DeepMind.

En 2014 dos grupos entrenaron independientemente CNNs grandes para predecir, con gran precisión, los movimientos que realizarían los jugadores expertos de Go en una posición determinada.

Los primeros en publicar fueron Christopher Clark y Amos Storkey de la Universidad de Edimburgo, el 10 de diciembre de 2014 con "Teaching Deep Convolutional Neural Networks to Play Go[56]". A diferencia de DeepMind, los investigadores desarrollaron un aprendizaje puramente supervisado: Usando dos conjuntos de datos de 16,5 millones de pares de posiciones de movimientos de juegos humanos, entrenaron una red neural para hallar la probabilidad de que un jugador haga un movimiento dado. Su CNN superó todos los resultados anteriores en la predicción de movimiento para ambos conjuntos de datos, e incluso derrotó al GNU Go.

Es importante entender que la red neuronal no tenía conocimiento previo de Go antes de ser entrenado con los datos, ni siquiera conocía las reglas del juego. Por lo tanto, se podría decir que estaba jugando puramente por "intuición", basándose en los datos que había visto durante el entrenamiento. Por ese motivo, es realmente impresionante que la red neuronal de Clark fuese capaz de vencer a un programa específicamente escrito para jugar Go, aunque tampoco es sorprendente que no fuese capaz de superar a los programas basados en MCTS, que en ese punto tenían un nivel bastante alto. Los investigadores concluyeron que combinando la predicción de movimientos aprendida por la máquina y la evaluación de movimientos de MCTS por fuerza bruta se podría conseguir un programa mucho más fuerte de lo que nunca había existido.

[56] CLARK, Christopher, STORKEY, Amos. "Teaching deep convolutional neural networks to play go". Preprint arXiv: arXiv:1412.3409, 2014. En línea: https://arxiv.org/pdf/1412.3409.pdf [Fecha de consulta: 25/02/2017]

A las dos semanas de la publicación del trabajo del grupo de Clark, se publicó un segundo artículo[57] que describía investigaciones con las mismas ambiciones, por Chris J. Maddison (Universidad de Toronto), Ilya Sutskever (Google) y Aja Huang y David Silver (DeepMind, ahora también parte de Google). Ellos entrenaron una CNN más grande y lograron resultados de predicción aún más impresionantes. Sin embargo, enfrentándose a programas MCTS seguía reflejando resultados decepcionantes. Al igual que el grupo de Clark propuso la interacción de ambos métodos llegando a implementar un prototipo.

Un contrincante de Go completamente imbatible

En enero de 2016 se marca un hito cuando se presenta al mundo un juego Go inteligente capaz de derrotar a cualquier contrincante humano, diferentes autores de entre los que se encontraban estos cuatro publicaron un artículo en la prestigiosa revista Nature anunciando el desarrollo de AlphaGo[58], el primer juego de Go capaz de derrotar a un jugador profesional de alto rango sin ninguna desventaja. Fue capaz de vencer a Fan Hui, campeón de Go de Europa, en 5 juegos, y a Lee Sedol, subcampeón internacional, por 4 a 1.

Figura 18. Caricatura de Lee Sedol a la izquierda y el campeón Fan Hui a la derecha jugando contra AlphaGo. Fuente: desconocida

[57] MADDISON, Chris J. "Move evaluation in Go using deep convolutional neural networks", *International Conference on Learning Representations*, 2015. En línea: http://www.stats.ox.ac.uk/~cmaddis/pubs/deepgo.pdf [Fecha de consulta: 25/02/2017]

[58] SILVER, David *et al*. "Mastering the Game of Go with deep neural networks and tree search", en *Nature*, n. 529, enero 2016, pp. 484-489. En línea: http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html [Fecha de consulta: 26/02/2017]

La clave del éxito de AlphaGo está en un enfoque híbrido bien diseñado ejecutado en un superordenador, basado en la combinación de redes neuronales y árboles de búsqueda[59]. Las redes usadas son conceptualmente similares a la función de evaluación de Deep Blue con la diferencia de que evolucionan mientras van aprendiendo en vez de regirse únicamente por los parámetros especificados en la programación. Creando tres redes neuronales distintas para poder trabajar conjuntamente con MCTS:

1. Una red neuronal con aprendizaje supervisado, denominada "policy network" la cual fue entrenada a partir de un conjunto de datos de 30 millones de pares de posición-movimiento de extraídas de partidas humanas y luego con aprendizaje por refuerzo (haciéndolo jugar contra versiones más antiguas de sí mismo). Esta red proporciona información sobre cuál es la siguiente acción a escoger desde una posición dada en el estado actual del juego.
2. Para evaluar los rollouts realizados en el MCTS se creó otra red mucho más rápida, la "rollout network". La policy network sólo se utiliza una vez para obtener una estimación inicial de cuán bueno es un movimiento y, a continuación, se utiliza la rollout network para elegir los muchos movimientos necesarios para llegar al final del juego en un despliegue del MCTS consiguiendo una selección de movimientos mejor que usando solo el azar pero con los beneficios de usar MCTS.
3. Desarrollaron una "value network" para que dada una jugada del tablero proporcione un valor que representa la probabilidad de victoria.

[59] "Google DeepMind's AlphaGo: How it Works": https://www.tastehit.com/blog/google-deepmind-alphago-how-it-works/ [Fecha de consulta 26/02/17]

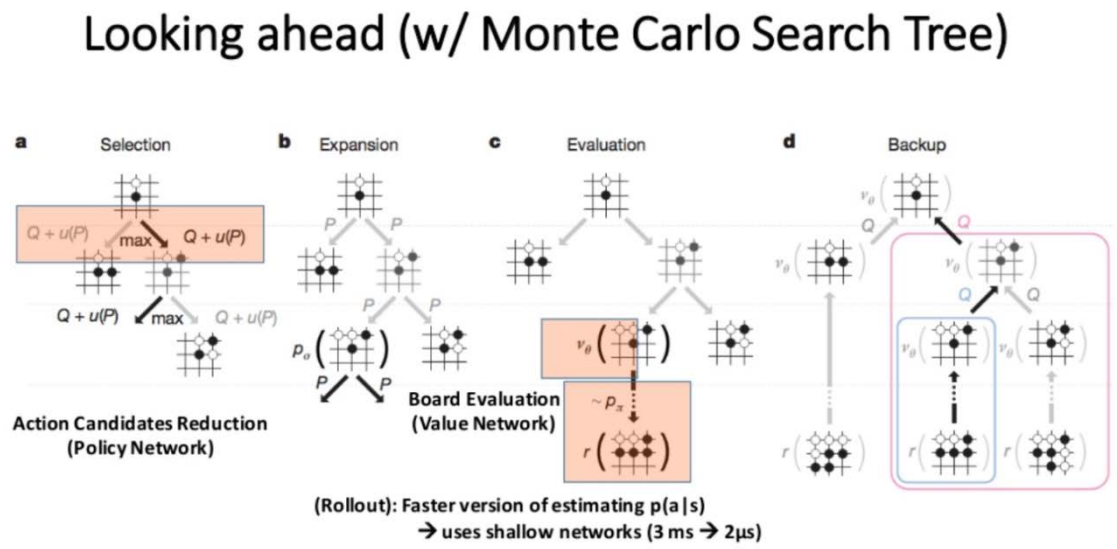


Figura 19. Red neuronal con MCST. Fuente: tastehit.com

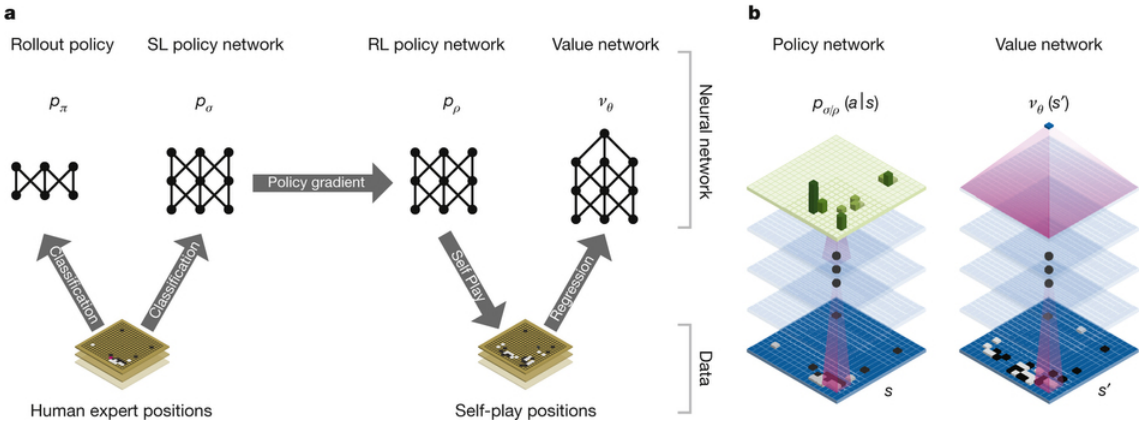


Figura 20. Redes neuronales para trabajo conjunto con MCST. Fuente: tastehit.com

Todas estas medidas se implementaron permitiendo la escalabilidad. AlphaGo utilizó 40 subprocesos de búsqueda ejecutándose en 48 CPUs, con 8 GPUs para cálculos de redes neuronales que se realizaban en paralelo. Teniendo también una versión distribuida que podría ejecutarse en múltiples máquinas, y escalar a más de mil CPUs y cerca de 200 GPUs.

Además de entrenar a un avanzado predictor de movimiento a través de una combinación de aprendizaje supervisado y de refuerzo, probablemente el aspecto más novedoso de AlphaGo es la fuerte integración de la inteligencia basada en el aprendizaje con los MCTS.

Este mismo año el equipo de Microsoft ha conseguido otro hito relacionado con los videojuegos. Han logrado desarrollar un sistema capaz de obtener la máxima puntuación en el juego de Ms. Pac-Man.

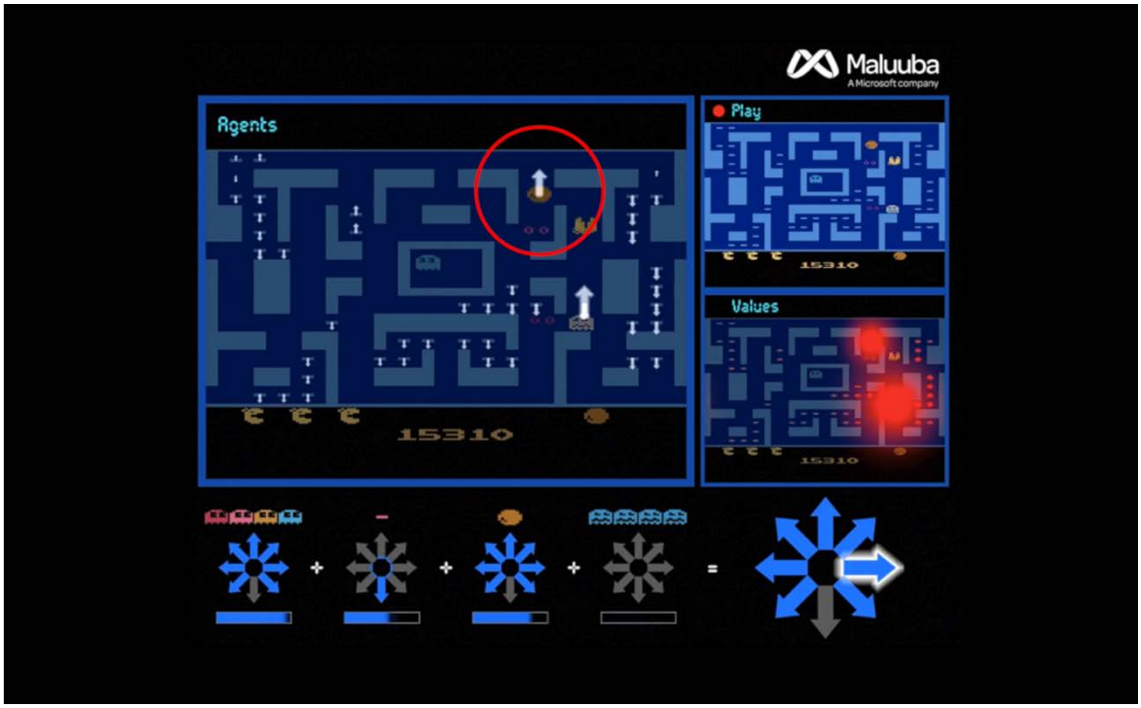


Figura 21. Sistema para lograr el máximo puntuable en Pac-Man. Fuente: blogs.microsoft.com

999.990 puntos es la máxima puntuación que se puede hacer en el videojuego Ms. Pac-Man, una versión posterior más compleja del juego original de Atari lanzado en 1981.

No hay constancia de que ningún humano haya logrado alcanzar esa puntuación en Ms. Pac-Man debido a la complejidad creciente del juego. Esta puntuación la ha logrado el sistema basado en inteligencia artificial (AI) desarrollado por Maluuba[60], una compañía canadiense de aprendizaje automático adquirida Microsoft hace unos meses.

[60] "Divide and conquer: How Microsoft researchers used AI to master Ms. Pac-Man": https://blogs.microsoft.com/next/2017/06/14/divide-conquer-microsoft-researchers-used-ai-master-ms-pac-man/#8y3fmXieWS3tMCPe.99 [Fecha de consulta: 18/06/2017]

Este sistema ha hecho uso de la técnica del divide y vencerás para evitar a los fantasmas en el videojuego dividiendo la partida en muchos problemas menores. Cada agente usado por el sistema, 163 en total, aprendió a resolver cada uno de esos problemas menores. Después el sistema aplicó el conjunto del aprendizaje para lograr la máxima puntuación.

Capítulo IV. ProjectAI

Introducción a ProjectAI. - Arquitectura general. - Diseño general

Introducción a ProjectAI

La génesis y creación de este proyecto está motivado por el desarrollo, estudio y aplicación de uno de los métodos de inteligencia artificial para juegos de estrategia antes mencionados, el algoritmo MiniMax, aplicándolo al caso práctico de un juego de estrategia. Se trata de un juego muy sencillo de 15 tiles de ancho por 10 de alto. Un terreno de juego de en el que se enfrentan dos jugadores: uno, representado por el color rojo, controlado por el humano y otro, azul, por la máquina, situados cada uno en un extremo del terreno. Cada equipo tiene un edificio principal el cual deben defender y a la vez conquistar el del equipo contrario para ganar.

Cada unidad al igual que los edificios tienen un número que representa su salud: 20 para los edificios y 10 para las unidades. Cuando esta llega a 0 en una unidad indica que esta ha muerto o si se trata de un edificio es porque ha sido conquistado. El número de soldados es inicialmente tres por equipo, cada unidad puede moverse en cualquier dirección en un radio de 3 tiles.

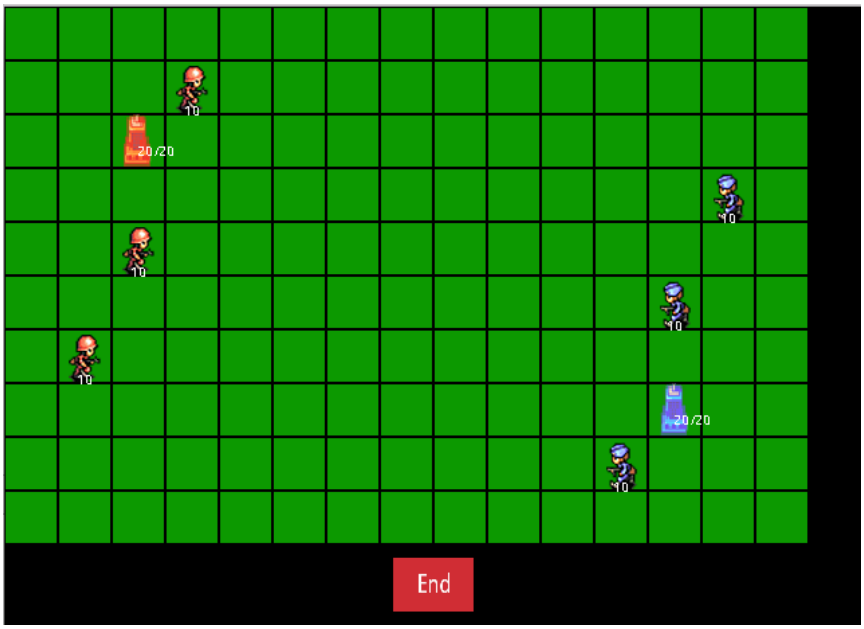


Figura 22. Interfaz de juego. Fuente: elaboración propia

La interfaz de usuario es sencilla. Para mover cualquier unidad basta con pulsar en el tile sobre el que se sitúa y entonces se pondrán en negro los posibles tiles a los que puede desplazarse. Respecto de los botones en la interfaz que aparecen en la parte inferior de la pantalla pueden ser:

- End: Finaliza el turno del jugador y comienza el del ordenador.
- Wait: Al mover una unidad indicamos que ya hemos decidido que ése es su destino y al pulsarlo tendremos control para mover el resto de unidades o para terminar el turno.
- Cancel: Al realizar un movimiento o ataque se puede retroceder si no se ha pulsado wait.
- Attack: Si estamos en el rango de ataque de una unidad enemiga podemos atacar pulsando el botón y después sobre la unidad a ser atacada (puede darse el caso de poder atacar a varias distintas).
- Capture: Esta acción se puede realizar cuando la unidad se sitúa sobre el edificio enemigo.

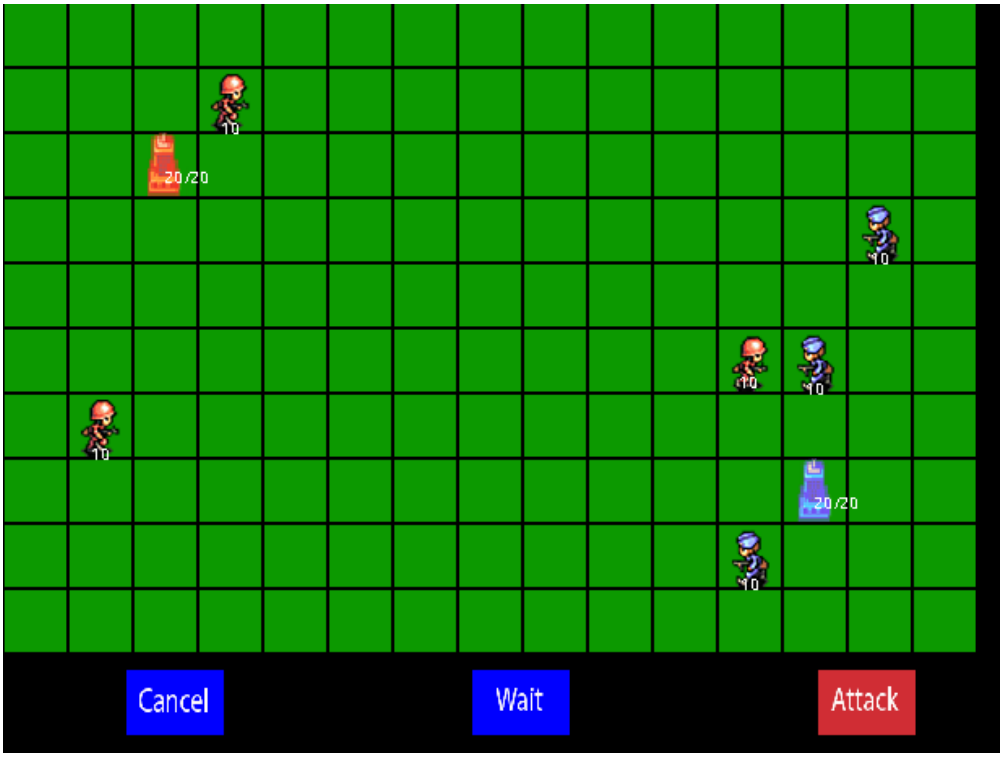


Figura 23. Interfaz de juego (II). Fuente: elaboración propia

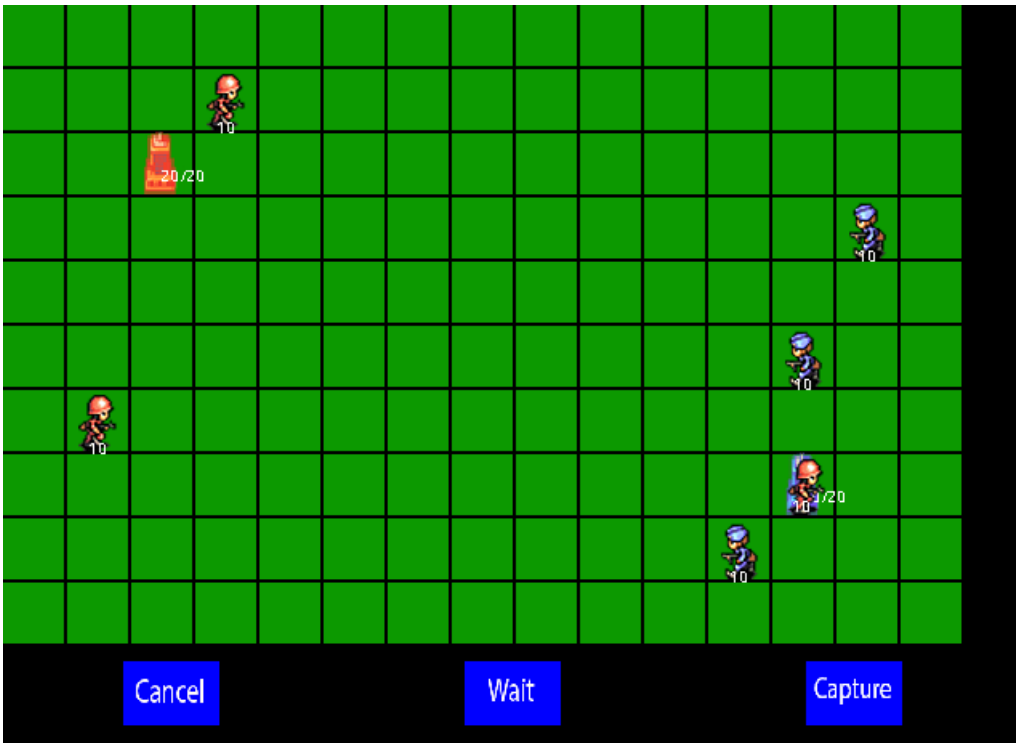


Figura 24. Interfaz de juego (III). Fuente: elaboración propia

Arquitectura general

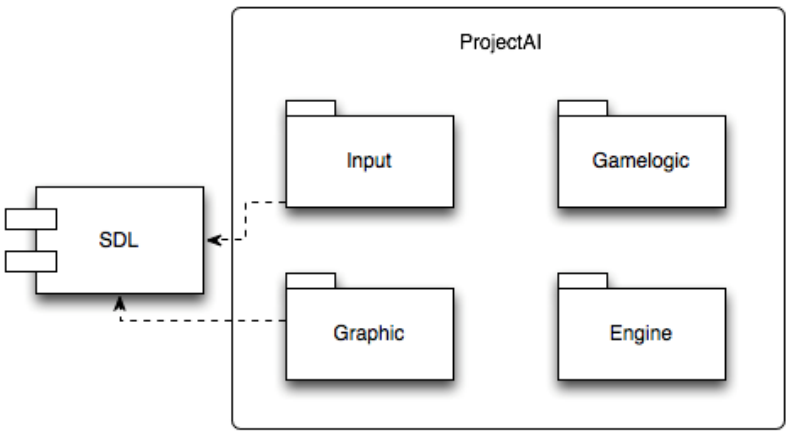


Figura 25. Arquitectura general ProjectAI. Fuente: elaboración propia

La arquitectura general de la aplicación se divide en cuatro partes. Por un lado tenemos la parte encargada de la parte gráfica y de la entrada de datos las cuales

hacen uso de la biblioteca SDL. Por el otro lado tenemos en un lado el engine del juego y por otro la parte de la lógica.

Es en esta parte dónde reside la inteligencia del juego y toda la funcionalidad relacionada con el comportamiento de este.

Diseño general

Para la implementación del juego se ha usado el lenguaje C++ en combinación con la biblioteca SDL debido al acceso a bajo nivel y versatilidad que proporcionan ambos. Además el propósito de la interfaz es que fuese sencilla para poder testear la inteligencia artificial evitando el consumo excesivo de recursos por parte de la UI. Se han usado patrones de diseño para ciertas características del juego como el modelo vista controlador y el patrón observer.

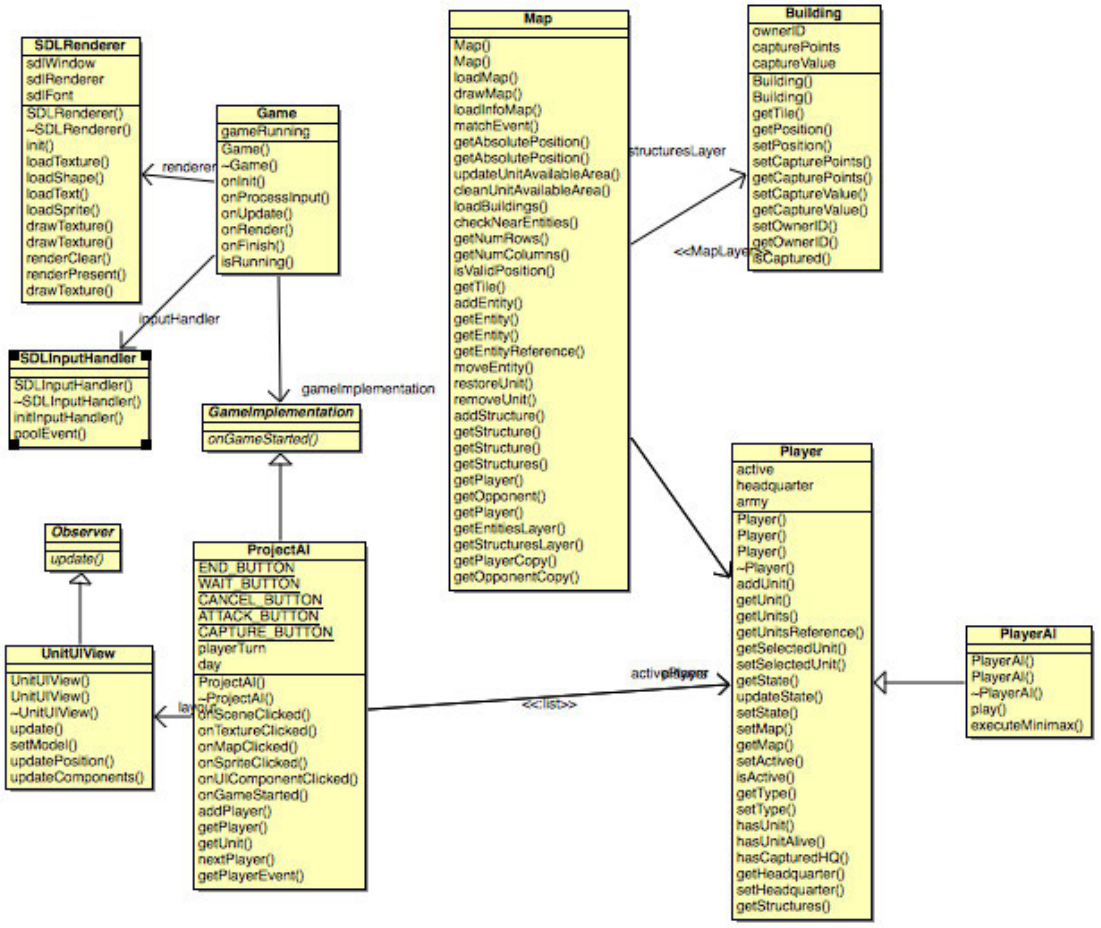


Figura 26. Diagrama de clases general ProjectAI. Fuente: elaboración propia

Al inicio de la aplicación, en el main, se crea un objeto de la clase Game que es el encargado de mantener y actualizar el bucle de renderización. Esta clase Game es la que trabaja con la biblioteca SDL.

```cpp
#include <stdio.h>

#ifdef __APPLE__
#include <SDL2/SDL.h>
#else
#include <SDL.h>
#endif
#include "Game.h"

const int FPS = 30;
const int DELAY_TIME = 1000.0f / FPS;

int main(int argc, char* arg[]) {
    Uint32 frameStart, frameTime;
    Game game = Game();

    game.onInit();

    while(game.isRunning()){
        frameStart = SDL_GetTicks();

        game.onProcessInput();
        game.onUpdate();
        game.onRender();

        frameTime = SDL_GetTicks() - frameStart;

        if(frameTime < DELAY_TIME){
           SDL_Delay(DELAY_TIME - frameTime);
        }
    }

    game.onFinish();

    return 0;
}
```

Figura 27. Código de la clase main. Fuente: elaboración propia

Éste a su vez cuando ejecuta el método onInit se encarga de crear la implementación del juego a través de la clase ProjectAI.

Esta clase se encarga de definir los parámetros de la escena del juego, como por ejemplo los distintos jugadores, sus unidades y manejar los eventos producidos por la interfaz como la gestión de los turnos, mover las unidades o controlar los eventos que se producen al pulsar los botones.

Se puede considerar como la clase principal que gestiona todo lo que ocurre a la vista del jugador.

```cpp
class ProjectAI : public GameImplementation
{
public:
    ProjectAI();
    virtual ~ProjectAI();
    void onSceneClicked(const Point position);
    void onTextureClicked(const Texture* texture);
    void onMapClicked(const Tile tile);
    void onSpriteClicked(const int id);
    void onUIComponentClicked(UIComponent* component);

    void onGameStarted(Scene* scene, Renderer* renderer);

    void addPlayer(Player* player);
```

Figura 28. Código de la clase ProjectAI. Fuente: elaboración propia

Hay tres comandos diferentes que se pueden producir en el juego. Uno es el de movimiento que implica mover una unidad de un tile a otro actualizando toda la información relativa a esta nueva posición, otro comando es el de ataque y el último es el de capturar. La ejecución de estos comandos en el juego viene dada por tres clases diferentes las cuales heredan de una clase que es Command.

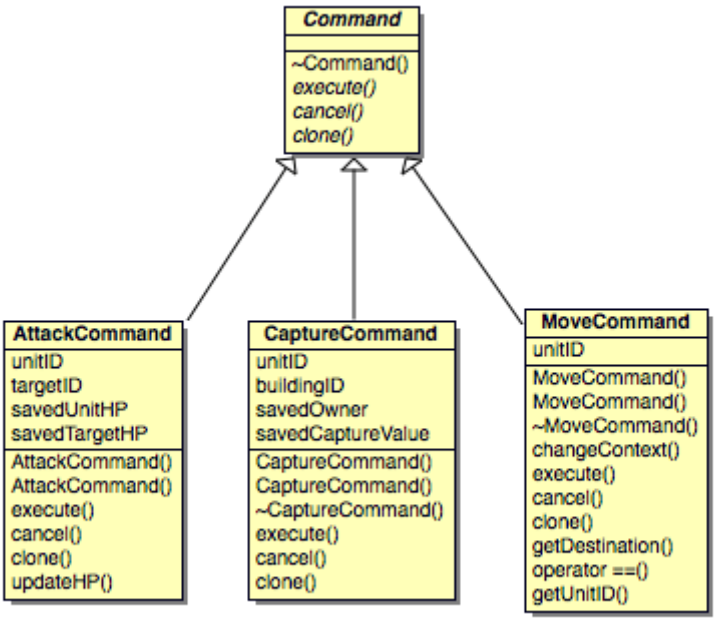


Figura 29. Diagrama de las clases involucradas en los comandos de ProjectAI. Fuente: elaboración propia

Otra de las clases importantes en el juego es la clase map, la cual que se encarga de manejar las acciones que suceden sobre el mapa en el momento del juego.

Es la principal proveedora de información para la ejecución del algoritmo de la inteligencia artificial. Proporciona la información de posición de las distintas entidades del juego además de controlar funcionalidades del juego en sí.

```
25 class Map : public MapContext {
26 public:
27     typedef std::vector< std::vector<Tile*> > TileMap;
28     Map( const Player& player, const Player& opponent );
29     Map( const Map& context );
30
31     //Width and height in tiles
32     void loadMap( Renderer* renderer, int width, int height );
33     void drawMap( Renderer* renderer );
34     void loadInfoMap( std::list<Player *> &players );
35
36     Tile* matchEvent( const Point& position );
37
38     //cast tile map position to absolute window coordinate position
39     Point getAbsolutePosition( const Point& tilePosition );
40     Point getAbsolutePosition( int x, int y );
41
42     void updateUnitAvailableArea( const Unit& unit );
43     void cleanUnitAvailableArea( const Unit& unit );
44
45     void loadBuildings( SpriteFactory* spriteFactory, Renderer* renderer,
46                         Player* player, Player* opponent );
47
48     void checkNearEntities( const Unit& unit,
49                             std::vector<UnitCommand>& commands);
50
51     int getNumRows() const;
52     int getNumColumns() const;
53
54     bool isValidPosition( const Point& position ) const;
55
56     Tile getTile( const Point& point ) const;
57
58     void addEntity( Unit& unit );
59     Unit* getEntity( const Point& reference ) const;
60     Unit* getEntity( int id ) const;
61     Point getEntityReference( int id ) const;
62
63     void moveEntity( Unit& unit, const Point& destination );
64
65     void restoreUnit( Unit& unit );
66     void removeUnit( Unit& unit );
67
68     void addStructure( Building& building );
69     Building* getStructure( const Point& reference ) const;
70     Building* getStructure( int id ) const;
71     std::vector<Building*> getStructures() const;
72
```

Figura 30. Código de la clase map. Fuente: elaboración propia

Capítulo V. MiniMax

INTRODUCCIÓN A MINIMAX. - PSEUDOCÓDIGO. - DISEÑO Y DESARROLLO

El punto más importante de este proyecto es la inteligencia artificial. Para el desempeño de esta tarea se ha escogido el algoritmo MiniMax.

Introducción a MiniMax

MiniMax es un procedimiento de búsqueda en profundidad. La idea consiste en comenzar en la posición actual y usar un generador de movimientos para generar un conjunto de movimientos plausibles. Mediante el uso de una función de evaluación estática se examinan todos los movimientos y se escoge el mejor. Después de esto, se lleva este valor hacia atrás hasta la posición de partida.

Los posibles movimientos que se pueden realizar en cada turno vienen dados por un problema de combinatoria, donde la población (m) es la cantidad de posibles movimientos que puede hacer cada unidad y la población (n) es el número de unidades de cada jugador.

Suponiendo que el número de posibles movimientos sea el mismo para todas las unidades, el número de n-tuplas viene dado por las variaciones con repetición de m elementos tomados en grupos de n.

Para escoger el movimiento de entre todos los posibles, que el ordenador va a realizar en su turno, se ha empleado el algoritmo MiniMax. Este algoritmo escoge el “mejor” movimiento para un juego de dos jugadores. El algoritmo genera un árbol de todos los posibles movimientos para ambos jugadores. Recibe el nombre de MiniMax porque busca el movimiento que el ordenador haga el movimiento que más le beneficie y maximice sus posibilidades de victoria, asumiendo que el oponente realizará el

movimiento que minimice las posibilidades de victoria del ordenador. Como los movimientos de los jugadores son alternados, en cada nivel (ply) del árbol de búsqueda recursivo se va alternando entre maximización y minimización.

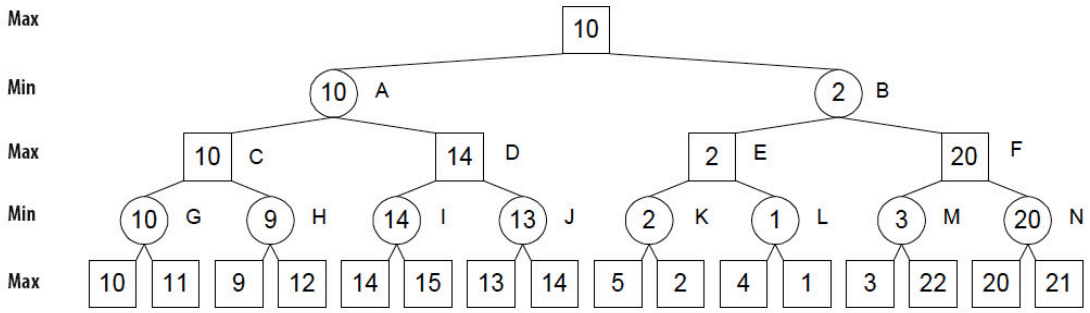


Figura 31. Ejemplo de aplicación de algoritmo MiniMax. Fuente: desconocida

Tomando como ejemplo una situación hipotética donde los posibles movimientos de cada jugador son dos distintos y la profundidad de búsqueda es cuatro. Tenemos el grafo anterior para ejemplificar el funcionamiento del algoritmo.

El nodo raíz muestra el estado actual del juego, siendo el ordenador el que tiene que realizar su movimiento. Se dan los movimientos A y B como posibilidades a ejecutar por parte del ordenador. Por cada uno de estos movimientos se irán ejecutando las acciones que corresponden con ese movimiento modificándose el estado del juego, ejecutando estos movimientos y cambiando el turno en cada llamada entre el jugador humano y el ordenador, que también tiene dos posibles movimientos. Se van realizando recursivamente las llamadas al algoritmo hasta que se alcance la profundidad establecida.

Una vez se alcanza uno de los nodos terminales se llama a la función de evaluación heurística que proporciona una puntuación dependiente de diversos factores tales como la salud de las unidades de ambos jugadores, la posición de estas respecto de los cuarteles a conquistar y otros factores que son ponderables.

Los valores más altos de esta función de evaluación son favorables para el ordenador y los más bajos son favorables para el jugador humano. Estos valores calculados al llegar a las hojas son propagados en la vuelta de las llamadas recursivas quedándose con el valor más alto o más bajo dependiendo de si el nivel del árbol en el que se encuentra es maximizador o minimizador.

El problema que conlleva la aplicación de este algoritmo en este caso concreto es, que como hemos visto antes la cantidad de movimientos posibles a evaluar en cada paso viene dada por el número de unidades y el rango de movimiento de cada una, que en es VR_{25}^{3}, que da como resultado $25^3 = 15625$si se evalúan todos los posibles tiles a los que puede moverse cada unidad. Este número se incrementa exponencialmente en cada ply del árbol por lo que si se intentasen explorar todos los posibles movimientos hasta acabar el juego el jugador humano posiblemente fallecería antes de que el ordenador realizase su primer movimiento.

Por ese motivo, se ha decidido acotar esta exploración mediante dos opciones. La primera es el uso de un filtro de acciones prometedoras que se explicará más adelante y la segunda es mediante el empleo de la poda alpha-beta.

Cabe recordar que el procedimiento MiniMax es un proceso primero en profundidad. Cada camino se explora tan lejos como lo permita el tiempo, la función de evaluación estática se aplica a las posiciones del juego en el último paso del camino, y el valor debe entonces propagarse hacia atrás en el árbol de nivel en nivel. Una de las ventajas de este tipo de procedimientos es que a menudo puede mejorarse su eficiencia usando técnicas de ramificación y acotación, en las que pueden abandonarse aquellas soluciones parciales que son claramente peores que otras soluciones conocidas.

Para poder aplicar esta poda es necesario mantener dos valores umbrales, uno que representa la cota inferior del valor que puede asignarse en último término a un nodo maximizante (que llamaremos *alpha*) y otro que represente la cota superior del valor que puede asignarse a un nodo minimizante (que llamaremos beta). Para ver cómo

funciona el procedimiento alpha-beta consideremos el ejemplo mostrado en la siguiente figura.

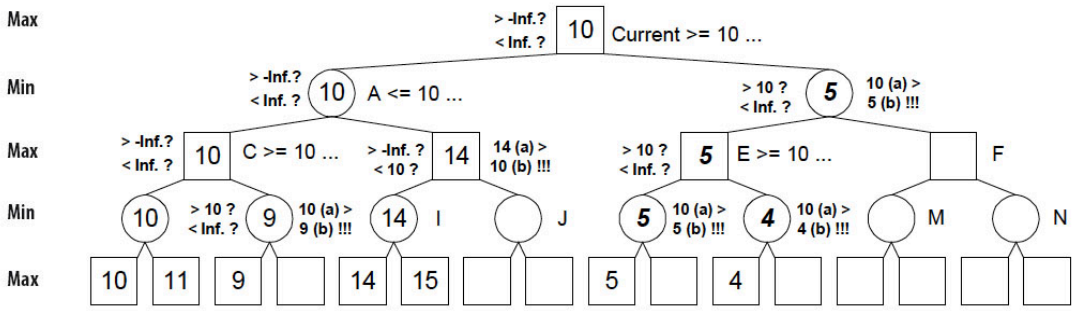


Figura 32. Árbol de algoritmo Minimax con poda alpha-beta. Fuente: desconocida

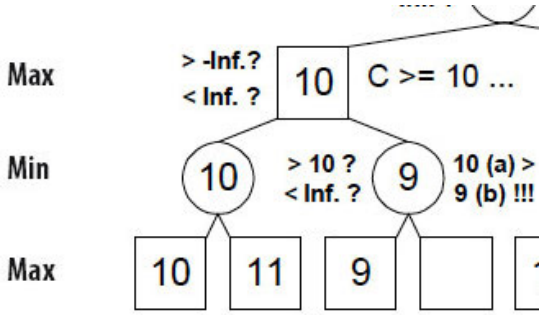


Figura 33. Detalle de algoritmo MiniMax con la poda alpha-beta. Fuente: desconocida

Si observamos la parte izquierda del árbol (Figura), cuando llegamos al cuarto elemento sabemos que en la capa anterior tenemos el valor 10, y al examinar la rama izquierda obtenemos como valor el 9. No tiene sentido examinar la rama derecha ya que si encontramos un número inferior a 9 este seguirá siendo inferior de 10 y en el siguiente nivel de maximización seguiremos escogiendo el 10, y si es un valor superior a 9 en ese mismo momento quedará descartado porque estamos en un nivel de minimización y se escogerá el 9 como valor.

En este caso se evita solo una llamada a la función de evaluación pero como se observa en la imagen se puede desechar toda una rama entera, lo que en términos de rendimiento supone una mejora sustancial.

A partir del ejemplo de la imagen completa, se ve que en los niveles maximizantes podemos excluir un movimiento tan pronto como quede claro que su valor será menor que el umbral actual, mientras que en los niveles minimizantes la búsqueda terminará cuando se descubran valores mayores que el umbral actual. Pero la exclusión de un movimiento posible del jugador maximizante significa realmente podar la búsqueda en un nivel minimizante.

La eficacia de la poda alpha-beta depende del orden en el que se examinan los sucesores, es decir, el algoritmo se comportará de forma más eficiente si examinamos primero los sucesores que tienen la probabilidad más alta de ser los mejores.

Si esto pudiera hacerse, implicaría que alpha-beta sólo tendría que examinar $O(b^{d/2})$ en lugar de los $O(b^{d})$ de MiniMax. Esto implica que el factor de ramificación eficaz será de $\sqrt{b}$ en lugar de b. En otras palabras, alpha-beta podría examinar aproximadamente dos veces más posiciones que MiniMax en la misma cantidad de tiempo.

Si se recurre a una ordenación aleatoria, que es lo más usual, el número aproximado de nodos examinados sería de $O(b^{3d/4})$.

Pseudocódigo

A continuación tenemos el pseudocódigo que representa el algoritmo MiniMax como tal y después con la poda alpha-beta.

- MiniMax

```
function minimax(node, depth, maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node

    if maximizingPlayer
        bestValue := −∞
        for each child of node
            v := minimax(child, depth − 1, FALSE)
            bestValue := max(bestValue, v)
```

```
        return bestValue

    else    (* minimizing player *)
        bestValue := +∞
        for each child of node
            v := minimax(child, depth − 1, TRUE)
            bestValue := min(bestValue, v)
        return bestValue
```

- Alpha-beta

```
function alpha-beta(node, depth, α, β, player)
  if node is a terminal node or depth = 0
      return the heuristic value of node
  if player 1
      For each child of node
          α := max(α, alpha-beta(child, depth-1, α, β, player2))
          if β≤α
              break (* pruning β *)
      return α
  else
      for each child of node
          β := min(β, alpha-beta(child, depth-1, α, β, player1))
          if β≤α
              break (* pruning α *)
      return β
```

Diseño y desarrollo

Las principales clases involucradas en la parte del algoritmo son las representadas en el siguiente diagrama:

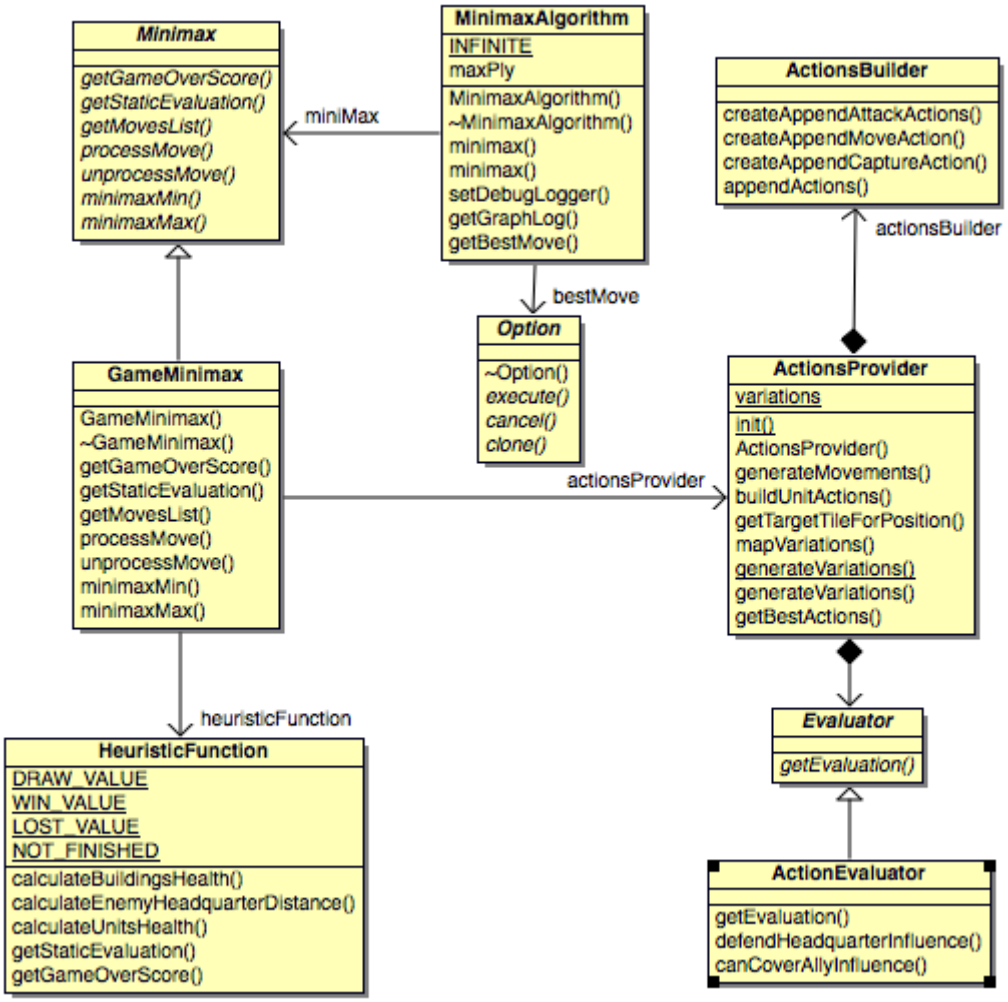


Figura 34. Diagrama de las principales clases involucradas en la parte del algoritmo de inteligencia artificial. Fuente: elaboración propia

```
int MinimaxAlgorithm::minimax( int ply, int alpha,
                               int beta, bool maximize ) {

    int gameOverScore = miniMax->getGameOverScore();
    if ( gameOverScore != GameState::NOT_FINISHED ) {
        if( graphLogger != nullptr ) {
            graphLogger->addToPath( graphLogger->getIndex() );
            graphLogger->nextPath();
        }
        return gameOverScore;
    }

    if( ply <= 0 ) {
        if( graphLogger != nullptr ) {
            graphLogger->addToPath( graphLogger->getIndex() );
            graphLogger->nextPath();
        }
        return miniMax->getStaticEvaluation();
    }

    MovementsList& movesList = miniMax->getMovesList( maximize );
    std::vector<Option*> moves = movesList.getMovementsVector();

    int bestSoFar = INFINITE;

    std::string actualNode = "";
    if( graphLogger != nullptr ){
        actualNode = graphLogger->getIndex();
        graphLogger->addToPath( actualNode );
    }

    for ( Option* option : moves ) {
        miniMax->processMove(option);
        int score = minimax( ply-1, alpha, beta, !maximize );
        miniMax->unprocessMove(option);
        bool topBranch = ply == maxPly;
        if( maximize ) {
            bestSoFar = miniMax->minimaxMax( bestSoFar, score,
                                             option ,&bestMove, topBranch );
            alpha = std::max(alpha, bestSoFar);
        } else {
            bestSoFar = miniMax->minimaxMin( bestSoFar, score );

            beta = std::min(beta, bestSoFar);
        }

        if ( beta <= alpha ) {
            if( graphLogger != nullptr ) {
                graphLogger->nextPath();
            }
            delete &movesList;
            return bestSoFar;
        } else {
            if( graphLogger != nullptr ) {
                graphLogger->addToPath( actualNode );
            }
        }
    }

    delete &movesList;
    return bestSoFar;
}
```

Figura 35. Código del algoritmo MiniMax con la poda alpha-beta. Fuente: elaboración propia

En la ejecución del algoritmo se comprueba en primer lugar si el juego se ha finalizado o si se ha alcanzado la profundidad esperada de búsqueda.

Si no se da ninguna de estas condiciones en primer lugar se llama a la función encargada de obtener la lista de movimientos a comprobar. Es en esta función donde se filtran las acciones más prometedoras, que se habían mencionado anteriormente, con el fin de reducir el tamaño del árbol de búsqueda.

```
28 MovementsList& GameMinimax::getMovesList( const bool maximize ) {
29     int numActions = Game::config.get("num_actions", 4);
30     if ( maximize ) {
31         return actionsProvider.generateMovements( mapContext.getPlayer().getId(),
32                                                   numActions );
33     } else {
34         return actionsProvider.generateMovements( mapContext.getOpponent().getId(),
35                                                   numActions );
36     }
37 }
```

Figura 36. Código de la función getMovesList. Fuente: elaboración propia

Para realizar esta tarea la función generateMovements crea en primer lugar un vector con todas las posibles acciones para cada unidad de un jugador.

```
36 MovementsList& ActionsProvider::generateMovements(
37                                     int playerID, int numActions ) const {
38     const Evaluator& evaluator = ActionEvaluator();
39     std::vector<Action*>* actionsSet = new std::vector<Action*>;
40
41     const Player* player = mapContext.getPlayer( playerID );
42
43     std::vector<Unit*> army = player->getUnits();
44     int numUnits = (int)army.size();
45     actionsSet->reserve( numUnits );
46
47     for ( const Unit* unit : army ) {
48         std::vector<Action*>& unitActions = buildUnitActions( unit->getId() );
49         getBestActions( unitActions, evaluator );
50         actionsSet->insert( actionsSet->end(),
51                             unitActions.begin(), unitActions.begin() + numActions );
52
53         std::vector<Action*> unusedActions;
54         unusedActions.insert( unusedActions.end(), unitActions.begin() + numActions,
55                               unitActions.end() );
56         unitActions.clear();
57         delete &unitActions;
58         for ( Action* action : unusedActions ) {
59             delete action;
60         }
61     }
62
63     std::vector<std::vector<int>> variations = generateVariations( numActions,
64                                                                    numUnits );
65
66     MovementsList& movements = mapVariations( numActions, variations,
67                                               *actionsSet );
68
69     return movements;
70 }
```

Figura 37. Código de la función generateMovements. Fuente: elaboración propia

Para conseguir generar las posibles acciones de cada unidad en el método buildUnitActions se crea un objeto de la clase AreaIterator.

```
88  std::vector<Action*>& ActionsProvider::buildUnitActions( int unitID ) const {
89      Unit* unit = mapContext.getEntity( unitID );
90
91      int maxAllowedActions = ( unit->getMovement() *  Game::config.get("num_actions", 4)) * 2;
92      std::vector<Action*>* actions = new std::vector<Action*>();
93      actions->reserve( maxAllowedActions );
94
95      int explorationRange = unit->getMovement() + unit->getAttackRange();
96      AreaIterator areaIterator;
97      areaIterator.buildArea( unit->getPosition(), explorationRange,
98                              mapContext.getNumColumns(), mapContext.getNumRows() );
99      while( areaIterator.hasNext() ) {
100         const Point destination = areaIterator.next();
101         if( unit->getPosition().onRange( destination, explorationRange ) ) {
102             TargetTile targetTile = getTargetTileForPosition( unit->getId(),
103                                                               destination );
104             if( targetTile != TARGET_NOT_AVAILABLE ) {
105                 actionsBuilder.appendActions( targetTile, mapContext, unitID,
106                                               destination, *actions );
107             }
108         }
109     }
110     return *actions;
111 }
```

Figura 38. Código de la función buildUnitActions. Fuente: elaboración propia

Este iterator se encarga de crear el área alrededor de una unidad comprobando la viabilidad de todas las posiciones.

Después se llama al método getBestActions el cual va a ordenar las acciones de acuerdo a la función getEvaluation que se encarga de asignar un valor a cada acción. Esta función toma como referencia la situación en el mapa para asignar un valor a esas acciones. Las acciones que incluyen un ataque o captura son las que reciben la más alta puntuación para ser evaluadas. Después se toma como referencia la posible defensa del cuartel o el avance de las tropas para ponderar las acciones.

```
8   float ActionEvaluator::getEvaluation( const Action& action,
9                                         const MapContext& context ) const {
10
11      // capture and attack actions are considered to have enough impact the be evaluated if there is one of them so we set a
            high score
12      float captureAttackModifier = 0;
13
14      if ( action.command != nullptr ) {
15          captureAttackModifier = 1;
16      }
17
18      Unit* actor = context.getEntity( action.moveCommand->getUnitID() );
19
20      //influence for position
21      const Point& destination = action.moveCommand->getDestination();
22
23      float positionInfluence = defendHeadquarterInfluence( destination, context ) +
24          canCoverAllyInfluence( *actor, destination, context );
25
26      return positionInfluence + captureAttackModifier;
27  }
28
```

Figura 39. Código de la función getEvaluation. Fuente: elaboración propia

Con esto se obtienen las mejores acciones de cada unidad en cada jugador. Pero hay otro factor a tener en cuenta a la hora de generar el abanico de posibilidades. Hay 3 unidades por cada jugador y para cada movimiento de cada unidad las otras dos se pueden combinar.

Dicho de otra forma, tomando como ejemplo una hipotética situación en la que las posibles acciones de cada unidad son Atacar (A) y Defender (D) tendríamos que las posibilidades que hay en total son:

AAA

AAD

ADA

ADD

DDD

DDA

DAD

DAA

Como ya habíamos estudiado previamente el orden es importante y se permite la repetición de los elementos por lo que nos encontramos ante una variación con repetición. Y de generar todas las posibles secuencias resultantes de la variación se encargan los métodos generateVariations y mapVariations al final de la función generateMovements.

```
173
174 std::vector<std::vector<int>>& ActionsProvider::generateVariations(
175                                                 int numActions, int numUnits ) const {
176     // look for a pre-calculated variation, if it doesn't exist, create a new one
177     auto iterator = ActionsProvider::variations.find(numUnits);
178     if ( iterator != ActionsProvider::variations.end() ) {
179         return iterator->second;
180     } else {
181         std::vector<std::vector<int>>* v = new std::vector<std::vector<int>>;
182         v->reserve(numActions);
183         std::vector<int> variation(numUnits);
184         generateVariations(v, numActions, variation, 0);
185         ActionsProvider::variations[numUnits] = *v;
186         return ActionsProvider::variations[numUnits];
187     }
188 }
189
190 void ActionsProvider::generateVariations( std::vector<std::vector<int>> *sequence,
191                         int numElements, std::vector<int> variation,
192                         int count ) {
193     if( count < variation.size() ){
194         for( int i = 0; i < numElements; i++ ) {
195             variation[count] = i;
196             generateVariations( sequence, numElements, variation, count+1 );
197         }
198     }else{
199         sequence->push_back( variation );
200     }
201 }
```

Figura 40. Código de las funciones generateVariations. Fuente: elaboración propia

```
143
144 MovementsList& ActionsProvider::mapVariations(
145                         const int numActions,
146                         std::vector<std::vector<int>>& variations,
147                         std::vector<Action*>& actions ) const {
148     MovementsList* movementsList = new MovementsList( actions );
149     movementsList->reserve( (int)variations.size() );
150
151     if( variations.empty() && actions.empty() ) {
152         return *movementsList;
153     }
154
155     if( actions.size() < variations.at( 0 ).size() * numActions ) {
156         throw IllegalStateException( "Params not valid" );
157     }
158
159     for ( int i = 0; i < variations.size(); i++ ) {
160         std::vector<int> actionIDs = variations.at( i );
161         int numUnits = (int)actionIDs.size();
162         Movement* movement = new Movement( numUnits );
163         for ( int j = 0; j < actionIDs.size(); j++ ) {
164             int actionID = actionIDs.at( j );
165             int key = actionID + j*numActions;
166             movement->addAction( *actions[key] );
167         }
168         movementsList->addMovement( *movement );
169     }
170
171     return *movementsList;
172 }
173
```

Figura 41. Código de la función mapVariations. Fuente: elaboración propia

Volviendo de nuevo a la ejecución de la función del algoritmo alpha-beta el siguiente paso es por cada posible acción que se han generado previamente llamar recursivamente a la función hasta que se llega a ply = 0, entonces en la vuelta atrás de la recursividad se va devolviendo el mejor valor utilizando los valores alpha y beta para comprobar los maximizadores y minimizadores evitando recorrer todo el árbol.

El mejor resultado se va guardando en la variable bestMove a lo largo de la ejecución del algoritmo. Este valor es recogido por la clase PlayerAI como un movimiento el cual se ejecuta después de la llamada al método del MiniMax.

El objeto graphLogger no tiene relación con el algoritmo en sí ni altera su funcionamiento. Solo se usa para construir el árbol creado en formato *.dot* con el fin de comprobar el correcto funcionamiento del algoritmo y de la preselección de los posibles movimientos.

Otra de las partes importantes del algoritmo es la heurística. Al fin y al cabo es una parte fundamental ya que se encarga de devolver el valor de la función de evaluación. Si este valor se calcula de manera errónea el algoritmo no tendrá un funcionamiento correcto. En este caso se ha buscado una función sencilla pero funcional.

Para evaluar cuán buena es una determinada situación simplemente se tiene en cuenta el cómputo de la salud de todas las unidades en el terreno, incrementando el valor cuando se trata de la salud de las tropas aliadas y disminuyéndolo con la de las enemigas. También se considera la puntuación que tienen los edificios, es decir esta comienza valiendo 20, y si alguna unidad comienza a capturarlo disminuye en función de la salud de la unidad. Al igual que con la salud de las unidades se incrementa el valor del edificio aliado y se disminuye el del enemigo.

Como el *ply* que alcanza el árbol es bajo puede ser que en las primeras evaluaciones no se lleguen a tener en cuenta estos factores. Por eso también se ha tomado la distancia al cuartel enemigo como otra variable a tener en cuenta en la función heurística.

```
int HeuristicFunction::calculateUnitsHealth( const Player& player,
                                             const Player& enemy ) {
    int result = 0;
    for (Unit* unit : player.getUnits()) {
        result = result + unit->getHP();
    }
    for (Unit* unit : enemy.getUnits()) {
        result = result - unit->getHP();
    }
    return result;
}

int HeuristicFunction::getStaticEvaluation( const Player& player,
                                            const Player& enemy ) {
    // Positive scores are good for AI
    // Negative scores are good for Human player

    int result = DRAW_VALUE;

    std::vector<Building*> playerBuildings = player.getStructures();
    std::vector<Building*> enemyBuildings = enemy.getStructures();

    result = result + calculateBuildingsHealth( player, enemy,
                                                playerBuildings, enemyBuildings );
    result = result + calculateEnemyHeadquarterDistance( player, enemy,
                                                         *player.getHeadquarter(),
                                                         *enemy.getHeadquarter() );
    result = result + calculateUnitsHealth( player, enemy );

    return result;
}
```

Figura 42. Código de las funciones encargadas de evaluar la función de evaluación estática. Fuente: elaboración propia

Capítulo VI. RECOPILACIÓN DE DATOS Y RESULTADOS

Las diferentes pruebas y ejecuciones se han realizado sobre un MacBook Pro (15-inch, Mid 2009) con procesador Intel Core 2 Duo 2,8 GHz y memoria RAM 4 GB 1067 MHz DDR3.

Alpha-beta (ply=2)	Tiempo (ms)	Nodos evaluados
1º turno	41	4711
2º turno	25	1445
3º turno	24	1323
4º turno	21	767
5º turno	13	473
6º turno	12	337

En ese turno finaliza el juego con la victoria del jugador humano, quedándose el ordenador a un turno de conquistar el cuartel humano. El jugador humano finaliza con 2 unidades con la salud al completo y el ordenador con 2 con la salud al completo y una con sólo 6. Se observa que la estrategia del ordenador ha sido buscar directamente el cuartel enemigo atacando sólo como medida defensiva.

Alpha-beta (ply=3)	Tiempo (ms)	Nodos evaluados
1º turno	554	53234
2º turno	250	19393
3º turno	359	28786
4º turno	139	13614
5º turno	38	3176
6º turno	23	1703

En este caso vuelve a ganar el jugador humano, aunque se observa que la estrategia de la inteligencia ha variado ligeramente. En lugar de buscar directamente la posición del

cuartel enemigo, ha adoptado una posición más ofensiva dando como resultado que el equipo humano sólo tiene una unidad con la salud al nivel máximo, otra unidad a mitad de salud y una con un sólo punto. De las unidades del equipo del ordenador sólo han sobrevivido dos con la salud prácticamente completa —ambas—.

Alpha-beta (ply=4)	Tiempo (ms)	Nodos evaluados
1º turno	13496	1270977
2º turno	5296	330129
3º turno	2426	118222
4º turno	1016	35547
5º turno	538	13038
6º turno	336	6709

El patrón observado por el ordenador es muy similar al obtenido con ply = 3. El ordenador toma una actitud más ofensiva hacia las tropas del jugador humano, dejándolo debilitado pero permitiendo su victoria debido a no tomar una posición más defensiva.

Alpha-beta (ply=5)	Tiempo (ms)	Nodos evaluados
1º turno	80581	4173134
2º turno	41832	3108640
3º turno	6211	610622
4º turno	366	14614
5º turno	293	12534
6º turno	189	8562

Según se aumenta el ply el comportamiento tiende a ser similar debido a que ya es lo suficientemente grande como para que alguna de las ramas del árbol llegue hasta una posición de victoria o derrota. En este caso, seguir aumentándolo carece un poco de sentido ya que el tiempo de cada turno aumenta exponencialmente. Con ply = 5 el

tiempo del ordenador para realizar su primer movimiento supera el minuto lo cual quita un poco la jugabilidad, aunque no sea ninguna prioridad en este proyecto. La mejor forma de obtener distintos resultados en un terreno de este tipo es ponderando de una manera distinta los valores que se tienen en cuenta en la heurística.

Al tratarse de un terreno pequeño y sin ningún obstáculo como es el ejemplo, la probabilidad más alta de victoria la tiene el jugador que mueve primero.

CONCLUSIONES

Las conclusiones extraídas al final de la realización de este proyecto en el cual se ha desarrollado un pequeño juego capaz de jugar contra un humano de una manera aceptable con un bajo nivel de dificultad son las siguientes:

- Se ha desarrollado con éxito una aplicación basada en C++ y SDL la cual permite a un jugador jugar contra el ordenador siendo este un contrincante con una habilidad media.

- Se ha comprobado la gran importancia que supone alcanzar una profundidad suficiente en el árbol de búsqueda para lograr un nivel de juego más alto.

- La inviabilidad del desarrollo de un juego de estrategia con un mapa de gran tamaño basado únicamente en un algoritmo del tipo búsqueda en profundidad como es MiniMax debido al amplio abanico de movimientos que hay que comprobar.

- La ventaja de usar jugadas de libro al empezar y finalizar el juego. De esta manera se puede conseguir reducir el tiempo de respuesta del algoritmo al reducir el tamaño del árbol sustancialmente ya que es al principio del juego cuando el abanico de movimientos posibles es más amplio.

- Las ventajas que tienen el uso de un sistema estadístico como el algoritmo de MonteCarlo, el cual se ha comprobado puede llegar a aumentar exponencialmente el rendimiento del algoritmo. Esto en mapas con un terreno de juego más grande puede suponer una importante diferencia.

- Para el desarrollo de un videojuego comercial es mucho más sencillo controlar el juego mediante máquinas de estado construidas con un alto número de posibilidades y un mínimo de nivel de aleatoriedad para que no parezca que el ordenador actúa siempre de la misma manera.

Después de desarrollar este proyecto la evolución natural sería desarrollar distintos tipos de unidades y mapas más grandes los cuales incluyesen distintos tipos de obstáculos como montañas y ríos para añadir una cierta variabilidad. Para esto, el siguiente paso a seguir sería conseguir una inteligencia que combinase distintos métodos dependiendo del punto en el que se encontrase la partida al igual que han hecho grandes juegos como los vistos en el capítulo 4 cómo DeepBlue, Chinook o AlphaGo.

BIBLIOGRAFÍA Y RECURSOS WEB[61]

BOSTROM, Nick, YUDKOWSKY, Eliezer. "The Ethics of Artificial Intelligence", en RAMSEY, William, FRANKISH, Keith [eds.], *Cambridge Handbook of Artificial Intelligence*. Londres: Cambridge University Press, 2014, pp. 316-334.

BRÜGMANN, Ben. *Monte Carlo Go*. *Technical report*, Physics Department, Syracuse University, 1993. En línea: http://www.ideanest.com/vegos/MonteCarloGo.pdf [Fecha de consulta: 22/02/2017]

CLARK, Christopher, STORKEY, Amos. "Teaching deep convolutional neural networks to play go". Preprint arXiv: arXiv:1412.3409, 2014. En línea: https://arxiv.org/pdf/1412.3409.pdf [Fecha de consulta: 25/02/2017]

COLOMINA, O. et al. "Aprendiendo mediante juegos: experiencia de una competición de juegos inteligentes", en *Actas de las X Jornadas de Enseñanza Universitaria de la Informática*, Universidad de Alicante, 2004.
En línea: http://www.dccia.ua.es/~company/Otelo.pdf [Fecha de consulta: 18/02/2017].

COULOM, Rémi. "The Monte Carlo Revolution in Go". *JJFoS Japanese-French Frontiers of Science Symposium*. En línea: https://www.remi-coulom.fr/JFFoS/JFFoS.pdf [Fecha de consulta: 21/02/2017]

HARFNOSKY, Holden. "Potential Risks from Advanced Artificial Intelligence: The Philanthropic Opportunity".
En línea: http://www.openphilanthropy.org/blog/potential-risks-advanced-artificial-intelligence-philanthropic-opportunity [Fecha de consulta: 27/07/2016]

[61] La presente bibliografía y recursos electrónicos empleados se organiza atendiendo a un criterio puramente nominal: se ordena alfabéticamente por autor; en el caso de los recursos web, pasan al final del listado, igualmente en orden alfabético de título.

GELLY, Sylvain *et al*. "The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions", en *Communications of the ACM*, vol. 55, n. 3, marzo 2012, pp. 106-113. En línea: http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Applications_files/grand-challenge.pdf [Fecha de consulta: 23/03/2017]

HINTON, Geoffrey E., OSINDERO, Simon, TEH, Yee-Whye. "A fast learning algorithm for deep belief nets", en *Neural Computation*, 2006. En línea: https://www.cs.toronto.edu/~hinton/absps/fastnc.pdf [Fecha de consulta: 22/02/2017]

IRIGARAY, María Victoria, LUNA, María del Rosario. "La enseñanza de la Historia a través de videojuegos de estrategia. Dos experiencias áulicas en la escuela secundaria", en *Clío & Asociados*, n. 18-19, 2014, pp. 411-437. En línea: http://sedici.unlp.edu.ar/handle/10915/47740 [Fecha de consulta: 07/01/2017]

MADDISON, Chris J. "Move evaluation in Go using deep convolutional neural networks", *International Conference on Learning Representations*, 2015. En línea: http://www.stats.ox.ac.uk/~cmaddis/pubs/deepgo.pdf [Fecha de consulta: 25/02/17]

MNIH, Volodymyr *et al.* "Playing Atari with Deep Reinforcement Learning". En línea: https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf [21/02/2017]

MUÑOZ, Norman *et al.* "Uso de la metodología GAIA para modelar el comportamiento de personajes en un juego de estrategia en tiempo real", en *Revista de la Facultad de Ingeniería de la Universidad de Amtioquía*, n. 53, junio 2010, pp. 214-224. En línea: http://jaibana.udea.edu.co/grupos/revista/revistas/nro053/Articulo%2020.pdf [Fecha de consulta: 12/01/2017]

NAREYEK, Alexander. "Intelligent Agents for Computer Games", en MARSLAND, T.A., FRANK. I. [eds.], *Computer and Games, Second International Conference*, CG 2000,

Springer LNCS 2063, pp. 414-422. En línea: http://www.ai-center.com/references/nareyek-02-gameagents.html [Fecha de consulta: 10/09/2016]

NEWELL, Allen, SHAW, Cliff, SIMON, Herbert. "Chess Playing Programs and the Problem of Complexity", en *IBM Journal of Research and Development*, Vol. 4, No. 2, 1958, pp. 320-335.

PLASENCIA, Alfonso. "«Deben prohibir los robots con inteligencia artificial en Bolsa»", en *El Mundo*. http://www.elmundo.es/economia/2016/04/04/57021c97268e3e40248b4595.html [Fecha de consulta: 11/11/2016]

SCHRAUDOLPH, Nicol N., DAYAN, Peter, SEJNOWSKI, Terrence J. "Temporal Difference Learning of Position Evaluation in the Game of Go", en COWAN, J.D., TESAURO, G., ALSPECTOR, J. (eds.), *Advances in Neural Information Processing 6*, Morgan Kaufmann: San Francisco, 1994. En línea (pre-print): http://www.gatsby.ucl.ac.uk/~dayan/papers/sds94.pdf [Fecha de consulta:22/02/17]

SHANNON, Claude E. "XXII. Programming a Computer for Playing Chess", en *Philosophical Magazine*, ser. 7, vol. 41, n. 314, marzo 1950. En línea: http://www.ee.ufpe.br/codec/Programming_a_computer_for_playing_chess.shannon.062303002.pdf [Fecha de consulta: 11/12/2016]

SILVER, David *et al*. "Mastering the Game of Go with deep neural networks and tree search", en *Nature*, n. 529, enero 2016, pp. 484-489, En línea: http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html [Fecha de consulta: 26/02/2017]

TESAURO, G. "Neurogammon: a neural-network backgammon program", en *Neural Networks, 1990 IJCNN International Joint Conference*, 1990.

TESAURO, G., SEJNOWSKI, T.J. "A Parallel Network that Learns to Play Backgammon", en *Artificial Intelligence,* 39, 1989, pp. 357-390. En línea: http://papers.cnl.salk.edu/PDFs/A%20Parallel%20Network%20That%20Learns%20to%20Play%20Backgammon%201989-2965.pdf [Fecha de consulta: 18/01/2017]

ZOBRIST, Albert L. "A model of visual organization for the game of Go", en *Sprint Joint Computer Conference*, 1969. En línea: https://www.computer.org/csdl/proceedings/afips/1969/5073/00/50730103.pdf [Fecha de consulta: 15/01/2017]

"A Brief History of Empire". En línea: http://www.classicempire.com/history.html [Fecha de consulta: 12/02/2017]

Deep Blue: http://stanford.edu/~cpiech/cs221/apps/deepBlue.html [Fecha de consulta: 18/03/2017]

Deep Thought: https://chessprogramming.wikispaces.com/Deep+Thought [Fecha de consulta: 17/02/2017]

"Divide and conquer: How Microsoft researchers used AI to master Ms. Pac-Man": https://blogs.microsoft.com/next/2017/06/14/divide-conquer-microsoft-researchers-used-ai-master-ms-pac-man/#8y3fmXieWS3tMCPe.99 [Fecha de consulta: 18/06/2017]

"Google DeepMind's AlphaGo: How it Works": https://www.tastehit.com/blog/google-deepmind-alphago-how-it-works/ [Fecha de consulta 26/02/2017]

"Inteligencia artificial: una inversión atractiva para el capital de riesgo". Expansión, 13/04/2017. En línea: http://www.expansion.com/economia-digital/innovacion/2017/04/13/58ef5788ca4741445d8b45a0.html [Fecha de revisión: 20/06/2017]

"Inversión en robótica e inteligencia artificial". *El blog de Paula Mercado, VozPópuli,* 02/02/2017. En línea: http://www.vozpopuli.com/el_blog_de_paula_mercado/Inversion-robotica-inteligencia-artificial_7_995670426.html [Fecha de revisión: 20/06/2017]

"La inteligencia artificial, en cifras: estas son las diez 'startups' que están conquistando el mundo", Nexo.club. En línea: https://nexo.club/la-inteligencia-artificial-en-cifras-estas-son-las-diez-startups-que-est%C3%A1n-conquistando-el-7731ebf47496 Fecha de consulta: 01/12/2016]

NeverStopBuilding. MiniMax: http://neverstopbuilding.com/minimax [Fecha de consulta: 01/05/2016]

The IBM 700 Series. http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/ibm700series/impacts/ [Fecha de consulta: 21/02/2017]

ÍNDICE DE FIGURAS