UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria

Informàtica

Universitat Politècnica de València

Análisis y aplicación de técnicas inteligentes a un jugador automático en videojuegos tácticos

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Iván Granados Rodrigo

Tutor: Antonio Garrido Tejero

Curso 2016-2017

Resumen

Se describe lo que es un videojuego táctico, la necesidad de que tengan jugadores automáticos y las restricciones a las que están sujetos. Se presentan algoritmos y aproximaciones posibles en la programación de uno de los jugadores; a continuación se implementa un juego como caso de estudio que replique las características de algún ejemplo actual del género. Finalmente se le añade a este varias de las técnicas explicadas anteriormente para ponerlas a prueba en simulaciones y recabar estadísticas

Palabras clave: computación, jugador automático, videojuego, inteligencia artificial, minimax, ajedrez, táctico, juego, estrategia

Abstract

It is described what a tactical video game is, the need for automated players and their restrictions. Possible algorithms and approximations in programming one of these players are discussed; next to this, as one example of the genre, a case-study game is implemented. Finally some of the previously explained techniques are added to it and tested in simulations to perform statistics

Keywords : computation, automated player, video game, artificial intelligence, minimax, chess, tactics, game, strategy

Tabla de contenidos

Sumario

1. Introducción

1.1 Justificación

El arte y la ingeniería de los videojuegos ha visto un gran crecimiento en las últimas décadas; el hecho de que estas simulaciones interactivas hayan cobrado tanta relevancia supone una oportunidad para los ingenieros informáticos – esto es, se sustituye el complicado modelado del mundo real, cuyo estado es altamente dinámico, por una simulación en su mayoría autocontenida. Se trata de una ocasión dorada para la aplicación práctica de la IA (inteligencia artificial) porque en este contexto se simplifican métodos complejos – tanto en la realización como en el coste computacional – y, por si fuera poco, resulta en productos de un mercado con muchas oportunidades.

Destacan por su sencillez – si se tiene en cuenta la taxonomía en géneros de los juegos – los de tipo táctico, como lo sería (estrictamente hablando) un ajedrez en su formato electrónico, pues **el tiempo y el espacio son discretos,** por lo que cada turno y cada casilla están perfectamente separados; y, por añadidura, el **estado** del mundo suele ser conocido para la IA (aunque a veces se pueda modelar el desconocimiento de la ubicación de las unidades enemigas, a lo que se suele llamar informalmente *niebla de guerra*)

Las **características habituales** de estos son:

• dos bandos, cada uno con una serie de unidades con distintas formas de moverse y atacar, no necesariamente las mismas en cada uno – por lo tanto puede ser asimétrico;

• un tablero discreto, separado en casillas;

• eliminar las unidades del contrario es la condición para ganar, y solo uno de los dos puede lograrlo – es, entonces, un juego de suma cero;

• las habilidades de las unidades varían según el juego o incluso dentro de estos: en unos casos es necesario que sean adyacentes para atacarse; en otros, si se cumplen ciertas restricciones, no (y no necesariamente situándose en la casilla de la unidad eliminada después, como en juegos clásicos);

• suele haber cálculos estocásticos parecidos a los que hay en el rol (probabilidades de acertar en un intento de ataque, en el uso de un objeto o habilidad limitado/a, etc.);

• a diferencia de los clásicos, como las damas, en los actuales es habitual poder mover más de una unidad en un solo turno, dependiendo de alguna métrica definida previamente; puede ser individual, por unidad – lo más común es en forma de puntos de acción –, o global entre ellas, véase [1] –;

• no siempre las unidades son eliminadas cuando sufren un ataque, sino que suelen tener *puntos de salud* que van perdiendo;

• no es extraño que haya mecánicas tridimensionales; por ejemplo, si la temática incluye armas a distancia (de fuego u otras), puede que haya elementos – coberturas – tridimensionales que ocupen casillas, o la intersección entre dos de estas, cuyo propósito es evitar la visión entre unidades (tal y como se ha llevado a cabo en el caso de prueba que presento)

Evidentemente, cada estudio y equipo personalizará lo que tiene para hacerlo único, de modo que las reglas son muy maleables. Lo más importante es que es muy usual que solo uno de los bandos sea una persona; así, la IA que lo antagoniza es crucial para el éxito del programa. El aspecto multijugador (partidas entre humanos) existe, en algunos casos, si bien el foco no suele estar puesto sobre él; y se espera que la mayoría de partidas sean contra la máquina, normalmente con una narrativa de fondo, más o menos extensa, que ligue las sesiones entre sí, justificando en aquellas, asimismo, un aumento o reducción en la dificultad o un cambio de las reglas.

La problemática aquí es: ¿cómo crear un oponente en este tipo de juego en concreto? Lo que aprendamos, ¿se puede generalizar a otros? ¿qué formalismos de la informática y las matemáticas nos podrían servir? ¿es, siquiera, práctico usarlos? Si es así, ¿cuáles de ellos son útiles? ¿qué aporta la rigurosidad de las ciencias de la computación al desarrollo de un videojuego táctico? No hay respuestas sencillas. A lo largo del trabajo abordo estas cuestiones mediante un caso de prueba (un sencillo juego y varios contrincantes) y el análisis de las aproximaciones posibles, tanto las programadas como las que no.

1.2 Objetivos

Los objetivos de este proyecto consisten en hacer:

1. **El motor de reglas de un sencillo juego que sirva como paradigma del género**

Es decir, el conjunto de normas, *el juego en sí:* los programas que rigen el comportamiento del sistema. Es un caso de estudio para analizar el problema más general de hacer jugadores automáticos para este género de juegos.

2. **Una interfaz gráfica sencilla**

Un modo en que el programador interactúe con el sistema para depurar y corregir errores, especialmente aquellos que tienen que ver con la geometría.

3. **Un análisis de los distintos modelos que se pueden utilizar**

Se trata de un vistazo a las técnicas inteligentes que se contemplan para usarse en nuestro problema y varias observaciones en cuanto a cuáles son útiles y por qué. También hay un resumen de la teoría que subyace a estas.

4. **Una implementación de algunos de estos modelos**

Una vez ya se han elegido cuáles son prácticos se eligen algunos de ellos para programarlos en código.

5. **Estadísticas de rendimiento y eficacia de estos**

Finalmente, se realizan enfrentamientos entre las técnicas, principalmente para ver si hay alguna mejor, y test estadísticos sobre los datos para inferir si las diferencias son estadísticamente significativas.

1.3 Guía de lectura

Esta memoria no tiene por qué leerse en orden por necesidad; sin embargo, para poder entender las *Conclusiones* (última parte), es preciso mirar la sección *2.4.5* para saber en qué varían las técnicas usadas en el proyecto con respecto a las originales, y la *Experimentación,* donde se obtienen los resultados principales y, muy especialmente, su justificación estadística. En *Marco teórico* hay una pequeña

introducción a la Teoría de juegos y a la teoría sobre los algoritmos minimax, amén de la sección 2.4.5 ya comentada; el resto de la sección ha de leerse solo si no se está familiarizado con esta teoría. Finalmente, la sección *Tecnología* describe las herramientas sobre las que se cimenta el código que finalmente se programó; estas dos son importantes en cuanto a cómo afectan el desarrollo de un videojuego en general y a cómo se han tomado las decisiones finales en el proyecto, pero no son totalmente cruciales para entenderlo. Aparte, **el código del proyecto en sí** se ubica en https://github.com/granates/tactical_game para cualquier consulta.

1.4 Caso de estudio elegido: el juego

Lo primero de todo es diseñar un juego como caso de prueba en el que probaremos las técnicas. Este consta de:

1. Un tablero tridimensional, implementado como tabla hash, donde para cada nodo con forma de tupla (x, y, z) existen otra serie de nodos que se consideran adyacentes y que están a una distancia (o coste) determinada.

2. Unidades que a las que les corresponde:

– **Puntos de acción**, que determinan cuánto pueden avanzar por turno

– **Salud**, si esta es menor o igual que cero esta es eliminada

– **Un objeto en un espacio** $\mathbb{R}^3$ que las representa; en este caso, un cilindro.

– **Un equipo**, 0 o 1

– Un **arma**, que tiene como parámetros:

Un rango (en distancia euclídea; es un número real)

Ataque (un entero que se refiere a los puntos de salud que resta)

Munición máxima que puede llevar

La **munición restante**

Cuántas veces dispara en una acción (valor de **ráfaga**)

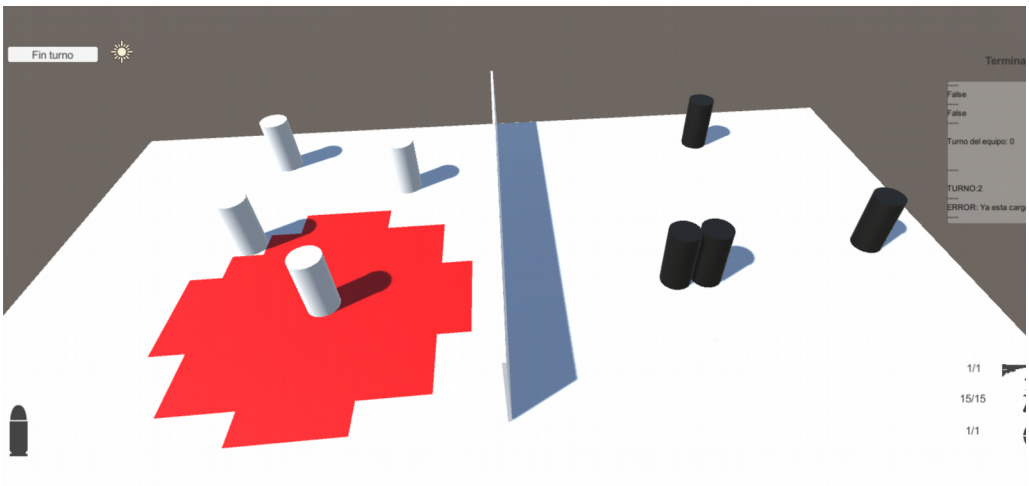

Ilustración 1: Imagen de la interfaz del juego en Unity. Se observa en rojo las casillas a las que la unidad puede llegar con sus puntos de acción. También se ve un muro como obstáculo en el medio que imposibilita que las unidades se ataquen en el estado actual de la partida.

3. Una serie de obstáculos en $\mathbb{R}^3$ que evitan, según la posición, que las unidades se vean entre sí; sin embargo, una puede ver a través de otra, es decir, las unidades son totalmente transparentes para las otras unidades.

4. También varias **operaciones**:

Una unidad puede, en un turno, realizar tres operaciones:

– **moverse por el tablero** (no termina el turno); para el cálculo del movimiento sobre el grafo se ha elegido el algoritmo de Dijkstra (véase la ilustración 1)

– **recargar** el arma (termina el turno)

– **atacar** (también termina el turno): reducir los puntos de salud de otra enemiga tantas veces como su valor de ataque con una probabilidad P que es, dadas unas constantes *máx delta rango*, *máx delta P* y *P_{máx fuera de rango}:*

Si está fuera del rango:

$$P = P_{máx\ fuera\ de\ rango} / \left| (distancia_{hasta\ oponente} - rango)^{máx\ delta\ rango} \right|$$

Si no:

$$P = P_{máx\ fuera\ de\ rango} + \left| (distancia_{hasta\ oponente} - rango) \right| / rango^{máx\ delta\ P}$$

2. Marco teórico

2.1 Teoría de juegos

El cuerpo analítico predominante en este proyecto es el de la Teoría de juegos, concebida originalmente por von Neumann en 1923 [2]; su objetivo es el análisis de ciertos objetos abstractos, llamados *juegos matemáticos,* cuyo origen y formulación parten de la noción informal de los típicos juegos de mesa como el ajedrez o las damas pero que, muy rápidamente, se generaliza para modelar el enfrentamiento (o la cooperación) entre agentes de cualquier tipo que busquen beneficio – modelado según una función de utilidad – desde situaciones como el mercado de valores hasta el dilema del prisionero.

Un juego matemático de información perfecta en forma extensiva (según [3]) es un objeto que consta de una 3-tupla: primero, un conjunto de n jugadores (de los cuales se supone que buscarán lo mejor para sí, es decir, que son racionales); segundo, un grafo de tipo árbol para el cual existe, para todo nodo hoja, una n-tupla que refleja la utilidad numérica reportada a cada jugador; y, tercero, una partición de los nodos no terminales en n subconjuntos, uno para cada jugador. Un juego, una *partida,* por lo tanto, sería un camino en el árbol desde la raíz hasta un nodo terminal. Es así que, intuitivamente, las aristas son decisiones (o movimientos/acciones) y los nodos son estados. Por otra parte, aquellos juegos que sean de información imperfecta además tienen un subconjunto de nodos, los aleatorios, para los que, para cada uno, existe una distribución de probabilidad sobre sus posibles n-tuplas de utilidad.

En esta teoría destaca, sin ninguna duda, el concepto de *Equilibrio de Nash* (véase [4]; debe su nombre a su descubridor, John F. Nash): se dice que un juego no cooperativo está en equilibrio de Nash (o solo en equilibrio) si cada jugador ha tomado una decisión para la cual no existe otra que le otorgue un aliciente para sustituirla (teniendo en cuenta las decisiones del resto); es decir, una vez elegida una estrategia ya no existen alicientes para cambiar a otra dadas las decisiones de los demás.

Otro concepto importante es la *suma* de un juego: la utilidad de un jugador J puede ser función de la de otro/s, de modo que al sumarse den un valor S, que puede ser positivo, negativo o cero; esto puede significar que, si es positiva, ambos se beneficien al terminar el juego, si es negativa, ambos pierden, y, si es cero, entonces se da un caso muy interesante: que las ganancias de uno son las pérdidas del otro, es decir, que el valor U de utilidad de J es -U para otro jugador J'.

2.2 ¿Cómo elegir estrategia?

Una vez formulada la teoría, ¿cómo elegimos el camino que más nos beneficie en un árbol? (es decir, aquellos movimientos que nosotros sí podamos elegir, claro, ya que los otros jugadores mueven en sus respectivos turnos). Efectivamente, **esto se reduce a un problema de búsqueda del camino más corto en un grafo**; es decir, dado, por ejemplo, este juego de dos jugadores (siendo cada nivel un turno distinto, alternativo para ambos):

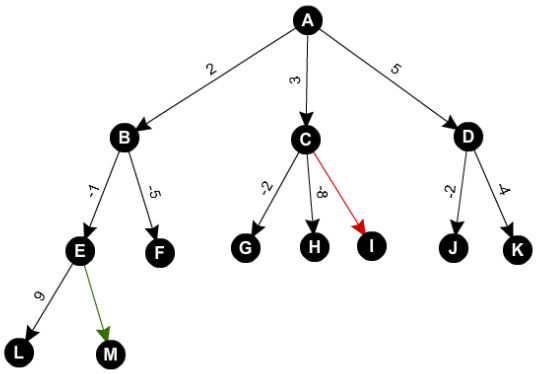

Ilustración 2: Un árbol de juego

Entonces el primer jugador (el que juega en la raíz y en el tercer nivel) querrá llegar a M, que es el nodo en que gana la partida; por contra, el adversario, cuyos movimientos son aquellos del segundo nivel, querrá llegar a I, el nodo en que él, por su parte, gana la partida. El problema para cada uno de ellos es, dado que el adversario escoja los mejores movimientos para él, ¿cómo hallo el camino de menor coste hasta un nodo final en que gane?

En el grafo anterior tenemos todos los nodos y aristas posibles del juego hasta los nodos finales más cercanos, es decir, es relativamente fácil obtener todos los costes hasta cualquier nodo sumando las diferencias de beneficio sucesivas a lo largo del árbol – una función habitualmente conocida como g(nodo) o g(vértice) – pero, ¿cómo predecimos a priori, sin conocer todo el árbol salvo hasta donde hemos llegado en el juego, la distancia de los nodos que quedan por expandir hasta un nodo final en que ganemos? Podríamos diseñar una función conocida como heurística o h(nodo) que nos prediga, aproximadamente, el coste hasta el nodo final en que ganamos la partida, con el requisito de que no lo sobreestime (a esto último se le llama admisibilidad y asegura optimalidad en varios algoritmos de búsqueda del camino más corto, es decir, se asegura que se encuentra el camino más corto siempre)

Los pesos negativos y el hecho de no poder elegir las aristas del adversario descartan gran cantidad de algoritmos; sin embargo, en la sección siguiente hablamos de uno que fue diseñado teniendo todo esto en cuenta.

2.3 Minimax

La idea del algoritmo Minimax (literalmente: minimización de la máxima pérdida) se encuentra descrita en el mismo texto que dio lugar a la Teoría de juegos [5]; en forma, primero, del Teorema Minimax (que también ha sido generalizado por varios autores [6]):

TEOREMA: En un juego matemático de suma cero con dos contrincantes (A y B) y un número finito de estrategias para ambos, existe un V tal que la mejor estrategia de A dada la de B da un valor de recompensa V igual al de la mejor

estrategia de B dada la de A cambiada de signo, de modo que: V(A|B) = -V(B|A).

Es decir, que esto coincide exactamente con la noción de *Equilibrio de Nash* en este tipo de juegos: **una vez elegida esta estrategia X** (en realidad un par *(x, y)* porque son dos agentes)**, ninguno de los dos jugadores tiene un aliciente para elegir otra, X', dadas las estrategias de los demás**; en otras palabras, se ha llegado a una situación *de equilibrio,* como si se hablara de termodinámica, donde naturalmente no tendría sentido que se cambiara de acción porque habría una especie de *resistencia:* una reducción (o cambio nulo) del beneficio o de la utilidad general; se ha llegado a una situación estática. Imaginemos que hemos elegido ir por el camino de la izquierda sabiendo que seguramente vayamos a tropezar, pero con el conocimiento de que por el derecho nos atropellaría un coche; sería, entonces, absurdo cambiar del izquierdo al derecho porque la utilidad que nos reporta el atropello es efectivamente menor que la de un eventual tropiezo, incluso si ambas son negativas.

Ahora sí, con el teorema en mano podemos describir un método, el algoritmo Minimax original, para poder encontrar ese valor V:

```
1    def Minimax(nodo, jugador_max: bool):
2        if nodo is terminal:
3            return valor(nodo)
4
5        V = -INF if jugador_max else INF
6
7        for hijo in nodo:
8            v = Minimax(hijo, not jugador_max)
9            V = max(V, v) if jugador_max else min(V, v)
10
11       return V
```

Se puede observar que al estar en un nivel de maximización se busca el mayor valor que sea, a su vez, el menor obtenido en el nivel justamente inferior del árbol, lo que quiere decir que el adversario inicial busca *maximizar el mínimo valor V;* es decir, se supone que el otro contrincante intentará minimizar el valor en su turno para evitar, en la medida de lo posible, que el jugador *MAX* gane. Es una sucesión de suposiciones en que se trata de prever las acciones futuras hasta todos los finales posibles de la partida.

Ahora bien, surge un problema: si el número de hijos promedio de un nodo (su *factor de ramificación*) es *b,* y la profundidad hasta un nodo que indique el final de la partida a favor de nuestro MAX es *d,* el tiempo esperado para encontrarlo es del orden de $O(b^d)$, lo que, a falta de un algoritmo polinómico, convierte a la búsqueda del valor Minimax en un problema NP-Completo. Esto hace que su versión básica sea poco útil fuera del ámbito teórico; así es como surgió su versión práctica:

```python
def Minimax(nodo, profundidad: int, jugador_max: bool):
    if profundidad == 0 or nodo is terminal:
        return heuristica(nodo)

    V = -INF if jugador_max else INF

    for hijo in nodo:
        v = Minimax(hijo, profundidad-1, not jugador_max)
        V = max(V, v) if jugador_max else min(V, v)

    return V
```

Observemos que se han introducido dos cosas más:

1. Una profundidad máxima de búsqueda para limitarla

2. Una operación, *heurística,* cuyo propósito es aproximar la V de un nodo sin sobreestimarla; viene a suplir el desconocimiento de la distancia real hasta un nodo final, tal y como se comentó en la sección anterior.

Ahora, si se llega al límite de profundidad, se tratará la heurística de ese nodo en lugar del verdadero valor. Con esto se mejora el algoritmo para evitar recorrer todo el árbol, siempre y cuando la función heurística esté bien diseñada, y al fin tenemos el algoritmo minimax.

2.4 Variaciones del Minimax

Desde la idea de von Neumann hasta nuestros días ha habido cambios y mejoras respecto al algoritmo clásico; en suma, los tipos de minimax barajados para el trabajo han sido:

2.4.1 Clásico (con profundidad y heurística)

El tradicional, descrito en profundidad en la sección anterior*;* tiene como problema que no solo es asintóticamente peor que su versión con poda sino que en la práctica esta no incrementa apenas las constantes de su función de tiempo. Su uso en el programa obedece a que al hacerlo con poda se ha de programar todo lo necesario de este y, aparte, por completitud en el código. **En un caso real sería ineficiente utilizarlo.**

2.4.2 Clásico, con poda alfa-beta; y Negamax

Constituyen una mejora neta del clásico sin perjudicarlo en modo alguno: la poda maneja dos variables: la alfa y la beta, que son, respectivamente, cota superior e inferior del valor minimax a lo largo del programa; el hecho de calcularlas y mantenerlas a lo largo de la ejecución permite cortar (podar) regiones enteras del árbol sin mucha complicación. Negamax, por otro lado, ofrece una mejora tan solo en forma de código: aprovechando el Teorema minimax de von Neumann (véase la sección *Marco teórico)*, deduce que no hace falta tratar a los dos jugadores de forma diferente, por lo que esta versión fija a un jugador el cálculo de la heurística y, si fuera el turno del otro, la devuelve cambiada de signo; además, esto significa que solo hay que mantener la alfa como parámetro. En la práctica se simplifica código, aunque asintóticamente no tenga impacto alguno.

Su origen es variado y ha sufrido modificaciones a lo largo del siglo pasado: hay pruebas de que fue diseñado independientemente por John McCarthy [7] y por Edwards y Hart [8] ; ha sido demostrada su optimalidad por J. Pearl (el creador de SCOUT) [9] ; y sufrido refinamientos y cambios por Knuth y Moore [10]. El estilo

del algoritmo minimax en forma de negamax ya era citado en 1980 por Fishburn [11]; sin embargo, no se habla de creador/diseñador en los artículos que he revisado, todo parece señalar a que es porque produce el mismo resultado que este y con la misma complejidad.

2.4.3 Negascout/SCOUT

Son muy similares al Negamax/Clásico con poda, y han sido descritos originalmente en [12] (Negascout) y [13] (SCOUT), respectivamente, además de que en [14] se describe un sistema muy similar; sin embargo, aun si es parecido a los clásicos, emplea un truco que, si funciona, reduce en buena medida los cálculos: presta especial atención al primero de los nodos conseguidos en la fase de generación. Si este primer hijo resulta ser el mejor candidato y el valor minimax es suyo (o de uno de sus hijos) entonces la computación termina y no hace falta revisar el resto de ramas – es una idea que resulta si la ordenación de movimientos deja en primer lugar el mejor; el problema es que, si este nodo no es el óptimo, la búsqueda tardará más que usando la poda alfa-beta, así que no se ha de entender como una mejora total de los otros en el caso general.

En el ajedrez, en la práctica, ha mostrado ser más rápido que los clásicos [15]; sin embargo, en mis pruebas, en este juego en particular, es empíricamente igual (véase la sección *Experimentación)*. Es probable que, debido a que mi programa no llega a profundidades d > 3 (téngase en cuenta que el factor de ramificación sin contar podas es mayor que b > 10000), no haya podido recabar los suficientes datos: sería necesario optimizarlo y reescribirlo en un lenguaje más eficiente para poder tener más resultados.

2.4.4 *Best-first* (SSS* [16], MTD-f [17], etc.)

Dada una jugada (estado), este tipo de procedimientos pretenden almacenar en memoria nodos anteriores cuyo valor heurístico se ha calculado – si este se encuentra– en un momento anterior de la ejecución (incluso, hipotéticamente, podría ser una base de datos completa calculada con anterioridad); sin embargo, dada la cantidad astronómica de estados posibles en los juegos tácticos es raro que un nodo se repita, en la práctica el grafo del juego es (casi) un árbol.

2.4.5 Minimax de este proyecto

Todos las técnicas minimax del trabajo tienen en común la función de coste y la función heurística que, en conjunto, suman la f(nodo), que aproxima la distancia de un nodo hasta el nodo final.

Coste:

$$g(n)=C_{salud}\cdot\sum(salud_{aliados}-salud_{enemigos})+C_{muerte}\cdot\sum(unidades_{aliadas}-unidades_{enemigas})$$

Heurística:

$$h(n)=-(C_{salud}\cdot\sum salud_{enemigos}+C_{muerte}\cdot\sum unidades_{enemigas})$$

Para unas constantes C_{salud} y C_{muerte}, convenidas previamente, cuyo propósito es que la superioridad de unidades tenga más peso que la suma de la salud de todas las unidades.

Planificación

Siguiendo con las diferencias con respecto a las implementaciones clásicas: **debido a la naturaleza probabilística de este tipo de juegos, lo que se hace siempre es barajar la situación matemáticamente esperada** (por ejemplo, reduciendo la salud de una unidad según el daño esperado del ataque: la media ponderada del daño según su probabilidad), y se continúa a partir de esta; calcular qué hacer según todas las posibilidades de fallo/acierto al disparar convertiría cualquier técnica descrita en intratable, ya que, como se observa, da lugar a un problema combinatorio.

Del párrafo anterior surge entonces una pregunta: ¿qué hacer si cuando se ejecutan las acciones los resultados difieren sustancialmente del caso esperado? **Si al ejecutar los movimientos se llega a un caso no previsto, se entra en una situación de "pánico":** donde, para esa unidad, se calcula una solución rápida con un script fijo – esto se da cuando un aliado ha elegido una misma casilla o cuando una unidad enemiga ha sido eliminada y se pretende, aun así, dispararle. Esto es, efectivamente, un problema de planificación en un entorno cambiante y, como tal, está sujeto a las restricciones de la situación real final.

Generación y poda de nodos

En cuanto a los **movimientos o aristas del árbol de juego**, antes de empezar una pasada del Minimax, es necesario **generarlos**; estos son tuplas de la forma:

$$movimiento = (unidad_{aliada}, casilla, (enemigo, h_{simple}(enemigo)))$$

donde la función h calcula el daño esperado a ese enemigo. Ahora bien, no todas las combinaciones son válidas, de modo que **la implementación elegida computa con el Algoritmo de Dijkstra las casillas a las que la unidad puede llegar dados los puntos de acción de esta**. Sin embargo, el número de movimientos promedio en el juego es b>10000 (frente a b<50 que tiene e.g.: el ajedrez), así que fue necesario buscar una solución para reducir el número de nodos a explorar en cada turno: **dado que las aristas generadas son demasiadas hay que seleccionar cuáles nos quedamos**. La función

$$h_{simple}(u_{aliada}, c, u_{enemiga}) = daño\,esperado(u_{aliada}, c, u_{enemiga}) - daño\,esperado(u_{enemiga}, c, u_{aliada})$$

está pensada para ordenarlos: aquellos cuya evaluación dé un valor mayor serán seleccionados primero; sin embargo, esta es bastante imprecisa para evitar un tiempo de cómputo elevado, así que, en su lugar, primero se ordenan según esta y después se escogen los mayores con una probabilidad mayor, la cual depende de valores extraídos de una distribución exponencial (véase, para más detalles, la función minimax_podar() en el código)

3. Tecnología

Esta se va a dividir en dos grupos: **algoritmos** y **software**:

Los **algoritmos** son las técnicas que se van a probar en el juego – *la teoría* –, y hacen referencia a los enfoques algorítmicos posibles (IA deductiva, inductiva, programación dinámica, voraz, etcétera)

El **software** engloba a las herramientas más tangibles – compiladores, *frameworks,* motores, lenguajes de programación, etc.

3.1 Algoritmos

En primer lugar, las técnicas que se pueden utilizar – lo que no quiere decir *que se deban usar,* o que sean prácticas en este caso – son muchas; voy a enumerar las que, durante el desarrollo del proyecto, he pensado que podrían llegar a implementarse:

Por un lado, las **técnicas simbólicas o deductivas**[1]:

Scripts fijos: el más sencillo y el más práctico: tan solo se evalúa el estado actual del juego para deducir qué acción llevar a cabo. Su uso en videojuegos es extremadamente común debido a que se tarda poco en programar y a que, normalmente, no precisa de una gran potencia de cálculo (suelen ser algoritmos polinómicos basados en autómatas finitos: por ejemplo: si me queda poca salud, paso al estado preprogramado de *huida;* si no, al de *persecución;* etc.) y porque, además, no suele ser obligatorio que la IA sea capaz de vencer a *cualquier* humano, hasta al mejor jugador; normalmente basta con que sea lo *suficientemente* desafiante para no sentir que es demasiado fácil y así no romper el estado psicológico de flujo (véase para esto [18])

Técnicas de búsqueda del valor minimax: son aquellos algoritmos que buscan el movimiento que garantiza minimizar la máxima pérdida; están descritos en la sección *Teoría* del presente trabajo. Son, en resumen, una familia algorítmica que, dado un juego matemático de suma cero, buscan el espacio de estados futuros posibles para, sucesivamente, analizar cuál es el mejor movimiento si el contrincante elige, a su vez, el mejor para él mismo. Su particularidad frente a otras aproximaciones es que consumen muchos recursos: el problema de decisión del valor minimax es, en el caso general, por la naturaleza combinatoria de la enumeración de las posibilidades futuras, un problema NP; una solución totalmente correcta puede incluso llegar a ser de coste factorial en el peor caso (un orden de magnitud superior al exponencial)

Sistemas expertos: los SE (sistemas expertos) son motores de inferencia que utilizan extensivamente una base de conocimiento previamente proporcionada para obtener conocimiento mediante el razonamiento lógico; su característica principal es que, en vez de utilizar el paradigma imperativo (o lo que es lo mismo, seguir una serie de pasos) se basan en reglas *si-entonces* de una forma no necesariamente determinista. Requieren mucha información del ámbito donde operan.

1 Son aquellas que, principalmente, se basan en el uso de la lógica o el tratamiento simbólico – búsqueda de patrones, etc. –; es decir, en la *deducción* de conocimiento desde este.

Por otro, **las inductivas o estadísticas**[2]:

Modelos bayesianos: se llama así a aquellos métodos que usan en gran medida la regla de Bayes para actualizar el conocimiento: informalmente consisten en que se ha de cambiar de creencia en función de la información de que se dispone. Hay muchas maneras de utilizar esta aproximación: se podrían usar las jugadas anteriores del adversario como evidencias, los turnos del mismo juego, etc. Su problema principal es que se ha de diseñar previamente con mucha rigurosidad (lo que también obliga a tener mucho conocimiento de Teoría de la probabilidad) y no puede usarse, como suele ser natural en la inducción, sin un gran volumen de datos

Otras técnicas del aprendizaje automático: En general no es trivial saber cuál es el mejor movimiento en un juego; conseguir etiquetas para el uso de **técnicas supervisadas** es una tarea muy difícil debido al gran esfuerzo humano; y tanto estas como las no supervisadas son complejas de implementar; sin embargo, ambas, en combinación, han dado fruto en juegos con un factor de ramificación muy grande [19] (algo muy habitual, por cierto, en juegos tácticos modernos), como Google mostró recientemente en *Nature:* su modelo ganó al campeón mundial de *Go*; sin embargo, esta línea de investigación es muy reciente y, de acuerdo al artículo, se precisa de una gran potencia de cálculo para lograrlo (o, lo que es lo mismo, mucho dinero), además de unas técnicas que muy fácilmente quedan fuera del ámbito de un trabajo de final de carrera.

La gran variedad de aproximaciones concretas en cada apartado es ya de por sí muy grande (en algunas es literalmente infinita), por lo que, por simplicidad, he decidido centrarme en programar un método fijo y varios de tipo minimax (ya que entre ellos comparten mucho código); queda, entonces, como ejercicio para un proyecto futuro, investigar cómo funcionarían las demás. Las razones que me han llevado a tomar esta decisión son varias:

– Un videojuego táctico es un juego matemático de suma cero con dos adversarios, los métodos minimax no solo encajan en él, sino que precisamente se diseñaron para esto

– Un script fijo no suele ser, en general y por sí mismo, difícil de programarse; depende del caso. Hay otro tipo de procedimientos que por su naturaleza ya son complicados hasta en los casos más sencillos (ya sea porque exigen muchos recursos o por el tiempo en que se llevan a cabo) Y, por si fuera poco, es muy usado en la industria; tener uno de estos algoritmos parece algo más bien necesario para un análisis certero

3.2 Software

En segundo lugar hay dos apartados en los que elegir: el lenguaje de programación y el motor gráfico para depurar visualmente los resultados. Ambos han de ser compatibles entre sí.

2 Estas aprovechan el uso de muchos resultados o evidencias para, a partir de ellos, elaborar un modelo matemático que prediga la mejor acción o el futuro más probable. Lo que es lo mismo: utilizan la *inducción* para obtener conocimiento en oposición a la *deducción.*

3.2.1 Lenguaje

Se barajan:

Interpretados y de muy alto nivel, como Python, Lua o Ruby: son muy comunes en el desarrollo de videojuegos para las tareas en tiempo de ejecución: inteligencia artificial, reglas, y demás cosas que podrían cambiar conforme se hace el juego; su contra es que son muy lentos, así que hay que vigilar qué parte del sistema se va a programar con ellos.

De medio-bajo nivel, como C y C++: se suelen usar para hacer el motor de videojuegos porque son rápidos y dejan acceso directo a memoria (punteros, etc.); son rápidos pero exigen mucho tiempo y dedicación para evitar fugas de memoria y otros problemas cercanos al hardware.

Semiinterpretados (máquinas virtuales, etc.) como Java o C#: son bastante populares, tanto para medio nivel – para, por ejemplo, tareas que no involucren la gestión de memoria – como para alto nivel, ya que siguen siendo más sencillos que C y FORTRAN y solo son del orden de dos a diez veces más lentos que ellos, en contraste con los cinco a cien veces más lentos que los lenguajes de muy alto nivel, como Python; tienen como añadido que son multiplataforma

3.2.2 Motor de juego

Se pensó en:

Un motor propio (quizá – por familiaridad – en OpenGL y C++): un sistema de geometría/trazado de rayos programado enteramente por mí mismo; es algo que, en cuestión de tiempo, podría por sí solo llevar, literalmente, cientos de horas. Incluso con los algoritmos correctos – suponiendo que no hay alternativas entre los mejores, lo que no es cierto – tampoco se prevé que el rendimiento sea el suficiente como para estar en producción sin meses, años, de esfuerzo, incluso si el modelo es el más sencillo que pueda contener este caso de prueba.

CryEngine: basado en C/C++ y con Lua como lenguaje de scripting; la documentación es algo escasa (véase [20]) y no es tan conocido como otras alternativas, dificultando su aprendizaje. Es, desde hace unos meses, gratuito; sin embargo, la licencia no contempla el uso científico (véase en la página oficial [21]) así que es probable que haya problemas legales si se usa en el proyecto.

Unity: Su núcleo está hecho en C/C++, aunque partes de él utilizan en gran medida la plataforma .Net y, especialmente, el lenguaje C#; ofrece mucha versatilidad en cuanto a lenguaje de scripting debido a .Net: se puede usar Python, Ruby, C#, Javascript, Boo, F#, Fantom, etc., etc. con interoperabilidad y memoria compartida entre todos ellos. Oficialmente ofrece documentación para Javascript y C# (antes también para Boo), aunque el acceso a la API desde cualquier otro no es, según mi experiencia, demasiado complicado. Este motor es muy popular, tiene mucha documentación y manuales no oficiales por Internet. Es gratuito siempre y cuando la empresa/particular que lo use no gane más de una cierta cantidad de dinero al año.

UnrealEngine: También escrito en C/C++, permite el uso de un lenguaje propio para el scripting, o bien C++, o bien, incluso, C++ c<on invocaciones a un intérprete de Lua; es conocido, al nivel de Unity, y ofrece documentación abundante. Pide *royalties* (un porcentaje de las ganancias) en productos comerciales; en asuntos académicos no cobra (en la portada de su web reza, de hecho: *"FREE for education"* [22]*)*

(...)

3.2.3 Elección de motor y lenguaje

Como **Unity** da lugar a mucha versatilidad y tiene tanta documentación finalmente se tomó este y la combinación de con **Python para las reglas y las técnicas inteligentes** y **C# para los asuntos gráficos de depuración** (ya que se harán muchas llamadas a la API y resulta más natural por el foco de la documentación en este lenguaje). Inicialmente se podría haber elegido CryEngine o Unreal porque dan la posibilidad de usar Lua, que es muy común en motores propios (Wikipedia, de hecho, lista 158 entradas en la página de videojuegos programados en Lua [23]); así cumpliría con el requisito de parecerse a lo usual en producción; finalmente, se descartó CryEngine por la escasez de documentación.

4. Experimentación

Para las pruebas he hecho un pequeño programa en Python, ubicado en el repositorio GitHub donde se encuentra el resto del proyecto, llamado *stats.py*, en el que se utilizan tests estadísticos Z [24] [25] que usan como parámetro de la distribución normal la proporción ganadas/totales del enfrentamiento de dos técnicas para obtener varias conclusiones. Todas las pruebas que involucren un minimax tienen una anchura b=20 y una profundidad d=2; los tests tienen un intervalo de confianza del 95% y el error muestral es menor al 5% en todos los casos (estos, aparte de todos los datos obtenidos, aparecen más adelante en esta misma sección). Además, una simplificación que se ha hecho es que, como negamax, minimax y minimax con poda alfa-beta teóricamente dan los mismos resultados, se ha elegido tomar solo uno de ellos para enfrentarse a Negascout y al script fijo; en este caso se tomó el minimax con poda.

4.1 Primera prueba: ¿Importa quién empiece la partida, si en el primer turno o el segundo?

En este experimento comparamos el ratio ganadas/partidas de los enfrentamientos cuando una técnica tiene el primer turno de juego versus cuando empieza en el segundo; la idea es averiguar qué técnicas son susceptibles de variación cuando empiezan primeras. La hipótesis nula es que las distribuciones 1) si una empieza primero y 2) si lo hace segunda son iguales. Los resultados son que las jugadas que involucran al script fijo (script fijo vs negascout y script fijo vs alfa-beta) son estadísticamente significativas con P(<0.05) – es decir, que se rechaza la hipótesis nula –, mientras que negascout vs alfa-beta da un valor z de la normal muy cercano al cero (véase la ilustración 2), es decir, que la probabilidad de que los datos se deban al azar a causa de la hipótesis nula es muy amplia – P(>>0.05) –; dado lo pequeño que es el error muestral (<4%) en todos los casos, **las evidencias sugieren que el script fijo es muy sensible al orden mientras que en el caso de los**

Ilustración 3: De izquierda a derecha: alfa-beta vs negascout, script fijo vs negascout y script fijo vs alfa-beta. ***Los valores fuera de la línea roja (límite: +-1.96) son significativos con P(<0.05)***

minimax no hay evidencias que sugieran lo contrario con un tamaño de muestra muy amplio (n>500)

Los datos son

Enfrentamiento de (X vs Y)	Partidas ganadas por X (X empieza)	Partidas jugadas (X empieza)	Partidas ganadas por X (Y empieza)	Partidas jugadas (X empieza)
Script fijo vs Alfa-beta	524	1003	818	1725
Script fijo vs Negascout	925	1790	537	1126
Alfa-beta vs Negascout	322	635	312	627

Los valores Z junto a las proporciones[3] son:

Enfrentamiento de (X vs Y)	Valor Z	Proporción victorias de X (X empieza)	Proporción victorias de X (Y empieza)
Fijo vs Alfa-beta	-0.9143	52.24% +-3.16%	47.42 %+- 2.41%
Negascout vs Alfa-beta	-0.7175	51.68% +-2.36%	47.69 %+- 2.98%
Fijo vs Negascout	-0.1483	50.71% +-3.97%	49.76 %+- 3.99%

4.2 Segunda prueba: ¿Hay alguna técnica mejor que otra?

Aquí calculamos si la diferencia en el ratio victorias/partidas entre cada par de técnicas es estadísticamente significativa en algún caso; se han sumado de entre

3 *El error muestral de la proporción se calcula según la fórmula* $e=\pm\sqrt{(1/n)}$

todos los enfrentamientos el número de victorias y el número de partidas jugadas de cada técnica para ver si hay alguna que sea mejor que otra. Con un error muestral muy pequeño (menor al 2% en todas las mediciones) podemos decir que no hay suficiente evidencia como para rechazar la hipótesis nula de que las distribuciones de cada par (X, Y) sean iguales en todos los casos. Lo que es lo mismo, tenemos un resultado interesante: **todos los algoritmos funcionan igual de bien, independientemente de su complejidad computacional, con una seguridad P(<0.05)**

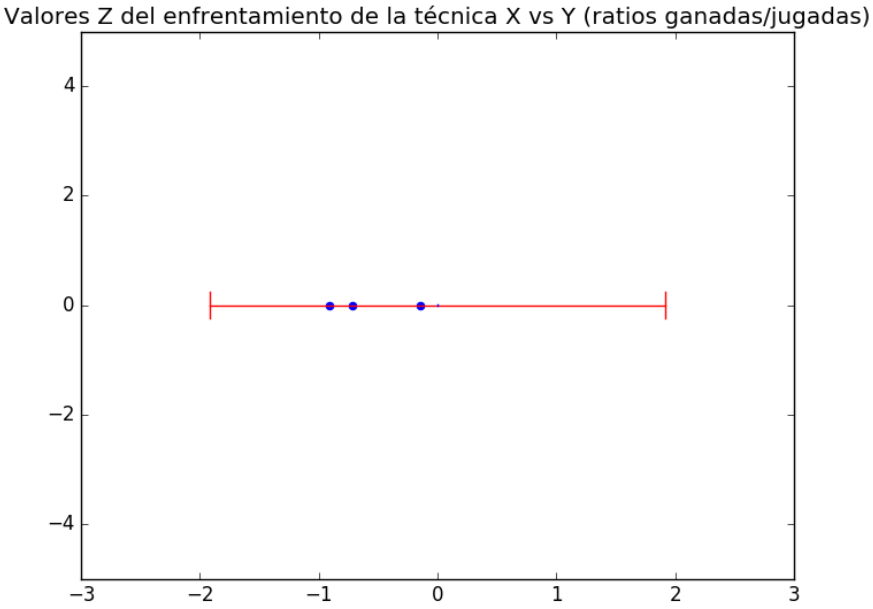

Ilustración 4: De izquierda a derecha: script fijo vs alfa-beta, negascout vs alfa-beta y fijo vs negascout. Los valores Z de los enfrentamientos en orden opuesto son los mismos cambiados de signo, lo que no cambiaría el resultado. ***Significativo si z>+1.96 o z<-1.96***

Los datos originales son:

Técnica	Partidas ganadas	Partidas jugadas	Ratio ganadas/jugadas
Script fijo	2804	5644	49.68%+-1.13%
Alfa-beta	2020	3990	50.63%+-1.58%

Negascout	2082	4178	49.83%+-1.55%

Y los valores Z junto a los ratios anteriores son:

Caso: X vs Y	**Valor Z**	**Ratio X: ganadas/jugadas**	**Ratio Y: ganadas/jugadas**
Fijo vs Alfa-beta	-0.9143	49.68% +-1.13%	50.63%+-1.58%
Negascout vs Alfa-beta	-0.7175	49.83%+-1.55%	50.63%+-1.58%
Fijo vs Negascout	-0.1483	49.68% +-1.13%	49.83%+-1.55%

5. Conclusiones

5.1 Resultados experimentales

Como hemos visto en los resultados estadísticos, no hay evidencias estadísticamente significativas de que ninguna técnica funcione mejor que las otras dados los parámetros utilizados; es decir, que, en la práctica, dados estos parámetros, se utilizaría el algoritmo de menor complejidad computacional (el script fijo). Sin embargo, es lógico pensar que los minimax probabilísticos funcionarían mejor si se usara una anchura representativa del verdadero factor de ramificación, por lo que haría falta optimizar el código/las técnicas para podernos hacer una idea mejor de cuán bien funcionan. Las consecuencias para el uso práctico de estas no son alentadoras ya que, como hemos visto, lleva mucho tiempo programarlas y este tiempo de desarrollo es preciado.

Otra conclusión interesante es que hay cierta evidencia significativa – de p-valor $P(<0.05)$ – en dos de tres casos en general (y de los que involucran al script fijo en particular) de que el orden en que empiezan a jugar los jugadores determina parcialmente el resultado de la partida, **pero no en las técnicas minimax.**

5.2 Valoración

Hay una gran cantidad de asuntos a tener en cuenta al diseñar un adversario en un videojuego táctico (y en cualquiera en general): ¿Qué ganan los desarrolladores y los jugadores al tener una IA con una fuerte carga teórica, es decir, que usa conceptos más complicados en lugar de simplemente usar un script fijo? ¿Y qué pierden…? El uso de técnicas inteligentes reportaría en:

Aspectos positivos:

1. Una gran flexibilidad para la elección de la dificultad debido a la variedad de los parámetros de los algoritmos clásicos (como los tipo minimax utilizados en este proyecto) sin necesidad de programar un método para cada caso

2. La seguridad y confianza de que son métodos probados y con gran peso en la comunidad académica

3. Un gran potencial para aumentar el reto que supone en la modalidad de un jugador, permitiendo el entrenamiento a gran nivel sin tener que recurrir a un adversario humano

Aspectos negativos:

1. En algunos casos estos métodos son muy costosos de programar (En tiempo de desarrollo; y, además, en dinero/hora – por el nivel académico necesario de las personas involucradas) y puede que no compense hacerlos en aras de tener mayor flexibilidad. Quizá con unos pocos scripts fijos el contrincante virtual sea lo *suficientemente inteligente* como para que se pueda jugar, especialmente si el videojuego tiene un fuerte componente narrativo y solo se necesita unos mínimos jugables. Esto debería ser debatido en la fase de análisis.

2. El coste exponencial (o más que exponencial) de estos métodos consume muchos recursos computacionales que, si se usa un lenguaje relativamente lento

(en comparación con C) de rápido desarrollo como Python, Javascript o Lua (bastante típicos en la industria para estos asuntos) puede fácilmente quitar una buena parte de lo necesario para que el resto de subsistemas (como la sincronización de la red, la gestión del estado del juego, etc.) funcionen; esto podría incrementar el tiempo para programar un orden de magnitud debido a tener que utilizar un lenguaje menos abstracto (como C++ o C#)

3. No siempre se utiliza el algoritmo tal y como fue concebido en un principio: como hemos visto en nuestro juego táctico ha sido necesario desarrollar una variación probabilística de los clásicos para que fuera práctica

El uso de técnicas inteligentes ha de usarse bajo el más estricto de los análisis: un juego cuyo propósito es que sea un reto en sí mismo las recibirá con los brazos abiertos; cualquier otro, probablemente, no. En el contexto del videojuego táctico, hay ciertas situaciones en las que se da (por ejemplo, ¿quién querría jugar al ajedrez contra *la máquina* si sabe que le dejará ganar? El juego *XCOM* [26] se publicitaba en esta línea como un juego "imperdonable" que deja elegir dificultades muy altas) y otros en los que no: en *Final Fantasy: Tactics* [27] se pone un gran énfasis en la construcción de una historia y en la evolución de las unidades después de cada enfrentamiento; y, por ello, quizá interese que el jugador "quiera más" simplemente por ver qué ocurre a continuación en la narración o qué armadura nueva obtendrá para su personaje preferido; es decir, tal y como se ha comentado con anterioridad, se trata de, principalmente, mantener el estado psicológico de flujo (descrito en [28]), el cual difiere de persona a persona – i.e. que cada uno llega según un grado diferente de competitividad/cooperación y esfuerzo. Esto afecta en gran medida a los aspectos de ingeniería y no solo al diseño del juego, ya que, en definitiva, establece las restricciones temporales y de corrección de las técnicas; aunque, claro está, en el plano de la investigación pura el científico puede (y debe) explorar más allá, como hizo aquel equipo de Google con el juego de Go [29].

En suma, este tema da para mucho más: queda analizar el resto de familias de algoritmos de las que se ha hablado, optimizar el código tanto en complejidad computacional como cambiando a un lenguaje más eficiente; quizá también hacer pruebas con el tiempo de desarrollo del programador de unos algoritmos frente a otros y cómo afectarían al análisis de requisitos de la aplicación. Por supuesto que, dada la naturaleza del TFG y su número de horas, es imposible llevar todo esto a cabo, y menos por una sola persona; invito al resto de alumnos y profesores a que le den una oportunidad al estudio de estos problemas, tanto en la parte algorítmica, central en este proyecto, como en la de ingeniería del software.

5.3 Trabajo futuro

Al final del desarrollo he reflexionado sobre varios asuntos que merecerían una revisión y, además, errores que necesitan corrección y que no se han subsanado por falta de tiempo:

– En mis mediciones el algoritmo de Dijkstra consume alrededor del cuarenta al cincuenta por ciento del tiempo de ejecución del programa, concretamente en el momento de la generación de las acciones posibles; una solución viable que podría conseguir una gran mejora sería precomputarlo en todos los casos y utilizar estos valores como heurística de un algoritmo de mejor búsqueda como A* (la posición de las unidades es dinámica y no se puede usar el Dijkstra ya calculado sin tratar); esto haría esa sección lineal con el número de nodos del camino más corto hasta la casilla destino, en lugar de $O(nodos + arista \cdot \log(arista))$; no es extraño este precálculo en los motores de ajedrez

– La parte de generación de movimientos que no involucra al grafo gasta alrededor del 20% ; un intento de optimización del sistema debería tener muy en cuenta esta sección de código

– En la sección gráfica en C# cada casilla individual es un objeto de la clase *Casilla* – mantener tantas instancias en el motor es tremendamente ineficiente; una mejor manera de localizar sobre qué nodo está el ratón pasaría por utilizar la parte entera de la intersección del rayo del ratón con la casilla (que es un punto); eso sí, esto dificulta colorear las casillas. Si se usara el código para un fin más práctico sería MUY importante rehacerlo de esta manera (aparte del diseño de una buena interfaz de usuario en general, al fin y al cabo el propósito de esta ahora es meramente el de informar del estado, no el de ser intuitiva)

– Una vez sean optimizados los minimax habrían de ser probados con más profundidad/más anchura; la utilizada (b=20, d=2) es claramente insuficiente para superar al script fijo – los resultados son similares entre ambos, pero el minimax tarda más. Debido a que el factor b supera el valor b=12000 en mis estimaciones, sería necesario conseguir una fracción representativa de este, quizá algo más del diez por ciento (b=1200). Más pruebas serían necesarias.

[1] *Robot Entertainment: (2012). "Hero Academy"*

[2] *von Neumann, J: (1928). "Zur Theorie der Gesellschaftsspiele"*

[3] *Hart, S.: (1992). "Games in extensive and strategic forms". En Aumann, R; Hart, S.: Handbook of Game Theory with Economic Applications 1. Elsevier. ISBN 978-0-444-88098-7*

[4] *Osborne, M. J.; Rubinstein, A.: (1994). "A course in Game Theory"*

[5] *von Neumann, J: (1928). "Zur Theorie der Gesellschaftsspiele"*

[6] *K. Fan: (1953). "Minimax theorems"*

[7] *Kotok, A.: (1962)."Artificial Intelligence Project, MIT & MIT Computation Center: Memo 41 - A Chess Playing Program"* disponible *en http://www.kotok.org/AI_Memo_41.html [Consultado por última vez 30/08/2016]*

[8] *Edwards, D. J.; Hart T. P.: (1961) "The Alfa-Beta Heuristic" disponible en http://hdl.handle.net/1721.1/6098 [Consultado por última vez 30/08/2016]*

[9] *Pearl, J.: (1982) "The Solution for the Branching Factor of the Alpha-Beta Pruning Algorithm and its Optimality", Communications of the ACM.*

[10] *Knuth, D. E.; Moore, R. W. (1975). "An Analysis of Alpha-Beta Pruning". Disponible en http://www.sciencedirect.com/science/article/pii/0004370275900193*

[11] *Fishburn, J. P.: "An Optimization of Alpha-Beta Search"*

[12] *Reinefeld, A.: Spielbaum-Suchverfahren. Informatik-Fachberich 200, Springer-Verlag, Berlin (1989), ISBN 3-540-50742-6*

[13] *Pearl, J.: (1980). "SCOUT: A Simple Game-Searching Algorithm With Proven Optimal Properties," Proceedings of the First Annual National Conference on Artificial Intelligence, Universidad de Stanford*

[14] *Fishburn, J. P.: "Analysis of Speedup in Distributed Algorithms", UMI Research Press ISBN 0-8357-1527-2, 1981, 1984*

[15] *Reinefeld, A.: Spielbaum-Suchverfahren. Informatik-Fachberich 200, Springer-Verlag, Berlin (1989), ISBN 3-540-50742-6*

[16] *Stockman, G.: (1979).*

[17] Schaeffer, A. et al.: (1996). Disponible en http://www.sciencedirect.com/science/article/pii/0004370295001263

[18]Nakamura, j.; Csikszentmihályi, M. (2001). "Flow Theory and Research" ISBN 978-0-19-803094-2

[19] D. Silver, A. Huang, C. Maddison et al.: (2016). "Mastering the game of Go with deep neural networks and tree search"

[20] http://docs.cryengine.com/display/SDKDOC1/Home Crytek. [Consultado por última vez 25/08/2016]

[21] htttp://www.cryengine.com [Consultado por última vez 25/08/2016]

[22] https://www.unrealengine.com/what-is-unreal-engine-4 [Consultado por última vez 25/08/2016]

[23] https://en.wikipedia.org/wiki/Category:Lua-scripted_video_games [Consultado por última vez 25/08/2016]

[24] Sprinthall, R. C. (2011). Basic Statistical Analysis (9th ed.). Pearson Education.

[25] Pennstate University. Probability Theory and Mathematical Statistics. Disponible en https://onlinecourses.science.psu.edu/stat414/node/268

[26] Firaxis Games: (2012). "XCOM: Enemy Unknown"

[27] Square: (1997). "Final Fantasy: Tactics"

[28] Nakamura, j.; Csikszentmihályi, M. (2001). "Flow Theory and Research" ISBN 978-0-19-803094-2

[29] D. Silver, A. Huang, C. Maddison et al.: (2016). "Mastering the game of Go with deep neural networks and tree search"