Sistemas Informáticos

Curso 2004 - 2005

Implementación De Juegos de Estrategia Con Programación Evolutiva

Javier Murillo Blanco

Javier Laureano Collado

Iván García-Magariño García

Dirigido por:

Lourdes Araujo Serna

Departamento de S.I.P.

Índice

Palabras clave

Se usarán con motivos de futuras búsquedas bibliográficas

1. Juegos de estrategia
2. Programación Evolutiva
3. Aprendizaje adaptativo
4. Inteligencia Artificial
5. Algoritmos genéticos
6. Algoritmos evolutivos
7. Evolución

Resumen inicial

Castellano

El objetivo de este trabajo es implementar juegos de contrincante, que tengan la suficiente complejidad para que la selección de la jugada a realizar no pueda abordarse mediante una búsqueda exhaustiva.

Entre estos juegos están los de estrategia del tipo ``civilización", en los que cada jugador hace el papel de regente de una civilización empezando con unos pocos pobladores y trata de construir un imperio compitiendo con otras civilizaciones. El objetivo del juego es dirigir esta civilización desde su inicio hasta conquistar el sistema.

Los algoritmos evolutivos constituyen un método de búsqueda alternativo para abordar problemas complejos de búsqueda a través de modelos computacionales de procesos evolutivos. El propósito genérico de estos algoritmos consiste en guiar una búsqueda estocástica haciendo evolucionar un conjunto de estructuras y seleccionando de modo iterativo las más adecuadas. La principal aportación de la Computación Evolutiva a la metodología de resolución de problemas consiste en el uso de mecanismos de selección de soluciones potenciales y de construcción de nuevos candidatos por recombinación de características de otros ya presentes, de modo parecido a como ocurre en la evolución de los organismos naturales.

En este trabajo utilizaremos este tipo de algoritmos para la selección de las jugadas de la máquina, definiendo previamente una serie de parámetros a optimizar en el juego de que se trate.

Por otra parte, los juegos de estrategia constituyen un excelente banco de pruebas para el estudio de procesos al nivel colectivo. El objetivo es determinar los mecanismos de interacción entre los individuos de una colectividad que hacen emerger comportamientos adaptativos o inteligentes al nivel de toda la colectividad de organismos. La simulación de hormigueros es un ejemplo, y sugiere múltiples aplicaciones a la Informática, como la computación distribuida. Otro campo relacionado es la evolución filogenética, que investiga las leyes que rigen la evolución de las poblaciones, los mecanismos de transmisión genética, selección natural y adaptación de las especies.

En el trabajo también aplicaremos estas ideas, permitiendo que la máquina realice su propia evolución, lo que dará lugar a estrategias a comparar con las obtenidas con métodos de búsqueda sobre un conjunto de criterios predefinidos.

Inglés

Summary:

The goal of this work is to implement games against other opponents, that have the neccesary level of complexity in order to can not find the best action with a thoroughly search. Among this games there are strategy ones, such as "Civilization", in which every player governs a different civilization beginning with some inhabitants and tries to build up and empire competing with the others players. The goal of this game is to lead the civilization from its begin until you conquer the system.

Evolutionary algorithms constitute an alternative method to tackle complex search problems using computational models of evolutionary processes. The generic objective of these algorithms is to guide a stochastic search to evolve a set of structures and to select, in an iterative way, the best ones. The main contribution of the Evolutionary Computation to the problem resolution methodology is to use selection mechanisms for potential solutions and to build new candidates by means of mixing characteristics of others that already exist, in the same way in which evolution takes place in natural organisms. In this work we will use this kind of algorithms for selecting the machine moves, but previously we will define a set of parameters that must be optimized in the corresponding game. Also, strategy games constitute an excellent bench for the study of processes at a collective level. The goal is to determine the interaction mechanisms between members of a group that generate adaptative or intelligent performances at the whole group level. Anthill simulation is an example, and suggests different applications to computer science, like distributed computing.

Another related field is phylogeny evolution, that research the laws that govern the population evolution, the genetic transmission mechanisms, the natural selection and the species adaptation.

In this work we will also apply this ideas, letting the machine to do its own evolution, which will give some strategies that will be compared with those obtained with search methods over a set of predefined criterions.

1. Introducción

1.1 Explicación General

El proyecto que abordamos consiste en un estudio de la Programación Evolutiva aplicada a los juegos de estrategia. Para ello, no sólo hacemos un estudio detallado de los temas que rodean a nuestro trabajo (como se ve en el apartado correspondiente), sino que además implementamos nuestro propio juego. Nuestro juego se parecerá a los juegos de estrategia comerciales en apariencia; pero, a diferencia de ellos, incorpora un sistema de Inteligencia Artificial combinado con Aprendizaje bastante sofisticado.

Dada una situación del juego, en un instante dado, en una partida del sistema contra el jugador humano, el sistema intentará explorar el mayor número de casos posibles. El sistema decidirá la jugada que más le conviene, suponiendo que su oponente elegirá la jugada más conveniente para él y peor para el sistema. Esta es una técnica bastante habitual en los juegos que incorporan IA. Sin embargo, para explorar todos los casos hasta los “finales de partida”, se necesitaría un tiempo totalmente inabordable. Por ello, a cada posible situación del juego intermedia se le asignará una calidad. Esta calidad de cada estado de la partida indicará en qué medida es ventajosa para uno u otro jugador. El anterior algoritmo se conoce como “minimax”, y es explicado con detalle más adelante.

El problema subyacente es estimar esa calidad de los distintos estados de la partida. Para esto, podremos usar muchísimos estimadores, por ejemplo el número de fichas de cada jugador o qué jugador tiene ocupada la posición central del tablero. Lo más normal es tomar una combinación ponderada de distintas estimaciones o calidades básicas.

Pues bien, dadas las distintas estimaciones básicas, los pesos en la ponderación de la estimación compuesta determinarán el tipo de estrategia de juego, ya que de esta manera da más prioridad a unas características o calidades frente a otras.

El mayor esfuerzo y atención de nuestro proyecto es determinar los anteriores pesos de estimación. Para ello utilizamos la Programación Evolutiva (PE). Esta nueva área introducida es explicada con detalle en el apartado correspondiente. La Programación Evolutiva se basa en que, dada un conjunto de soluciones (individuos) escogidas inicialmente de forma aleatoria, éstas se irán cruzando, mutando y seleccionando a lo largo de distintas generaciones, imitando un proceso natural de selección de Darwin. Fruto de dicha evolución / selección se obtendrá un individuo o solución cuasi-óptima. En nuestro proyecto dicha soluciones o individuos son los conjuntos de pesos de

ponderación. Para determinar la calidad o aptitud de unos pesos (o estrategias de juego) frente a otros se realizarán torneos de partidas máquina contra máquina.

1.2 Objetivos.

Los objetivos de este proyecto son:

1) Abordar la implementación de un juego de estrategia usando la programación evolutiva.

2) En consecuencia al anterior, surge la idea de aprendizaje. El aprendizaje se abordará en dos sentidos:

2.1) Dadas unas estimaciones atómicas, el sistema aprende la manera y medida en que combinar dichas estimaciones, por medio de unos pesos de ponderación (usa programación evolutiva)

2.2) El sistema aprenderá de manera individualizada contra cada oponente, por medio de almacenamiento de estrategias. Esto es utilizará en sus procesos de aprendizajes la última estrategia ganadora que ganó al oponente contra el que está jugando, (usa algoritmo de escalada sencillo).

3) Proporcionar un trabajo de investigación de alternativas posibles y temas relacionados.

4) Proporcionar una interfaz gráfica amigable, que haga amena la interacción del humano con el sistema.

5) Crear nuestro propio juego de estrategia, proporcionando así un rasgo de originalidad.

6) Hacer un juego eficiente en cuanto a tiempo de ejecución.

1.3 Símil

Dado que la programación evolutiva tiene una importancia enorme en nuestro proyecto, y dado que el lector puede que inicialmente no se encuentre familiarizado con la misma, creemos que es una buena idea explicarlo con un pequeño símil, en esta introducción.

En el modo de uso de la programación evolutiva explicado, el sistema se asemeja a un aprendiz que se dirige a una escuela de ajedrez. En dicha escuela le enseñan las estrategias básicas:

Intentar tener en todo momento más piezas que el oponente

Cada pieza tiene un valor, reina = 10, torre = 5, alfil = 3...

Se debe intentar que nuestras piezas ocupen las piezas centrales

etc.

Las anteriores estrategias se asemejan a las funciones de estimación atómica que nosotros (la escuela) mostramos al sistema (aprendiz). Sin embargo, nuestro aprendiz todavía no sabe jugar bien ajedrez, le falta la experiencia. Ante una nueva partida no sabe qué prioridad dar a cada estrategia. A lo largo del tiempo nuestro aprendiz adquiere de manera intuitiva qué prioridad dar a cada estrategia y cuáles son claramente más importantes, o le llevan con más facilidad o rapidez a la victoria.

La importancia de cada estrategia se asemeja a los pesos de la función de estimación ponderada.

La experiencia del aprendiz se asemeja a los torneos llevados a cabo por la programación evolutiva, que le hicieron aprender.

Otra manera de verlo sería que cada individuo de la población de PE es un aprendiz de la escuela. El Algoritmo Evolutivo (AE) consistiría en que todos los aprendices de la misma escuela jugaran entre sí, y se fueran clasificando los mejores. Así mismo los jugadores compararían ideas (cruces) y experimentarían ciertos cambios para probar (mutación). De esta manera el aprendiz ganador sería el resultado de todo el esfuerzo de la Escuela de Ajedrez y que se asemejaría al resultado del AE.

2. Introducción y Estudio Preliminar.

2.1 Introducción a Áreas Relacionadas

2.1.1 Introducción a la IA

La inteligencia artificial (IA abreviado) es la ciencia que se encarga de estudiar las maneras para simular o hacer que un ordenador “piense” en la medida que lo hace un ser humano. Se dedica al estudio de los campos que todavía poseen problemas no resueltos. Cuando se resuelven, pasan a salir del estudio de la IA.

2.1.1.1 Áreas que abarca la IA

Los campos abarcados por la IA son muchísimos, por ejemplo aquí mostramos unos cuántos:

- Tratamiento de Lenguaje Natural: Por ejemplo pueden resumir, traducir, extraer cierta información, filtrado de Spam (correo basura indiscriminado)...

- Agentes software, que simulan agentes humanos con ciertas capacidades de decisión. Por ejemplo, compra de bolsa, búsqueda del viaje más adecuado,...

- Predicciones del tiempo

- Por último, **los juegos**. Dentro de los juegos se pueden incluir los de tiempo real (por ejemplo los juegos de fútbol), y juegos por turnos, que a su vez se pueden clasificar por sus números de jugadores. De múltiples jugadores (ejemplo el parchís) y de dos jugadores (por ejemplo ajedrez, damas, etc.)

La mayoría de estas áreas son bastante extensas, con gran cantidad de reglas, casos imprevistos, y factores aleatorios que influyen a la hora de querer obtener el comportamiento deseado. Es por ello, que hay problemas que todavía no se consideran resueltos con total éxito. Ejemplo de esto podría ser el problema de traducir un texto de un lenguaje natural (p.e el español) a otro lenguaje natural (p.e. el inglés), de tal forma que el significado global, nexos y coherencia sean perfectos. Otro ejemplo, sería resumir un texto plano sin estructura alguna obteniendo el significado más importante.

Sin embargo, los juegos son un simplificación de la realidad, en la cual todas las cualidades de la supuesta realidad se convierten en un conjunto finito de reglas, perfectamente definidas, y sin ninguna clase de incertidumbre (quizás convenga aclarar que un elemento aleatorio de probabilidad conocida no es incierto sino aleatorio, p.e “lanzar un dado”). Un ejemplo de juego, de los más clásicos, es el ajedrez. En éste se simula una batalla o guerra entre dos ejércitos, dirigidos ambos por un único rey. Aunque en la realidad una batalla tenga muchísimos factores inciertos, en el ajedrez, se determina el número de unidades de cada ejercito, (por ejemplo caballos, soldados de a pie o “peones” etc.), así como sus movimientos y capacidades de ataque, con unas reglas claramente definidas.

Por tanto la IA puede abordar los juegos de manera sencilla, y con gran éxito. Por ello, en nuestro proyecto hemos elegido crear un nuevo juego, y desarrollar en este una “inteligencia artificial” que desarrolle un buen comportamiento en el jugador simulado por la máquina.

2.1.1.2 Métodos usados por la IA

Para llevar a cabo el comportamiento “inteligente” se han desarrollado gran variedad de métodos o mecanismos. En la mayoría de los casos, dada una entrada (situación de la realidad, equivalente a lo que el humano recibiría por sus sentidos) el sistema debe devolver una salida (acción que desempeñaría el humano).

En concreto, se suele buscar la solución en un conjunto de estados solución posibles. Para encontrar dicha solución hay infinidad de métodos entre los que están:

- Búsqueda en un espacio de estados clásica: Búsqueda primero en profundidad, búsqueda en anchura, búsqueda voraz (greedy), algoritmo A*,...

- Búsqueda en situaciones con dos adversarios: minimax, alfa-beta. Son muy usadas en juegos de adversarios (ajedrez, damas, etc.).

- El uso de paradigma basado en reglas: Se parte de un estado inicial, al cual se le aplican reglas hasta obtener el estado de salida (solución).

- Paradigma de razonamiento basado en casos: Se parte de un conjunto de problemas ya resueltos. Dado un nuevo problema se busca el problema más similar del conjunto de problemas resueltos (etapa de recuperación) y se devuelve la adaptación de la solución del problema más similar (adaptación). Si el nuevo caso o problema resuelto es correcto y representativo (etapa de revisión), se aprende (etapa de aprendizaje).

- Programación Evolutiva: Explicada anteriormente.

... y muchos más.

Nosotros en nuestro proyecto hemos usado la búsqueda en juegos de dos adversarios (en concreto minimax), combinado con la programación evolutiva según explicaremos más adelante.

2.1.1.3 IA en juegos

La IA en juegos se suele basar en el siguiente concepto: Cada estado es una situación de un juego en un instante. Por ejemplo, en el caso del ajedrez, un estado determinaría la situación de todas las fichas del tablero, así como quién tiene el turno (“blancas” o “negras”).

Así mismo las posibles jugadas que puede realizar cada jugador, las llamaremos operadores. Estos, en el ajedrez, serían del estilo “mover peón en rey 1 a rey 3” (rey 1 y rey 3 identifican las posiciones de origen y destino del movimiento).

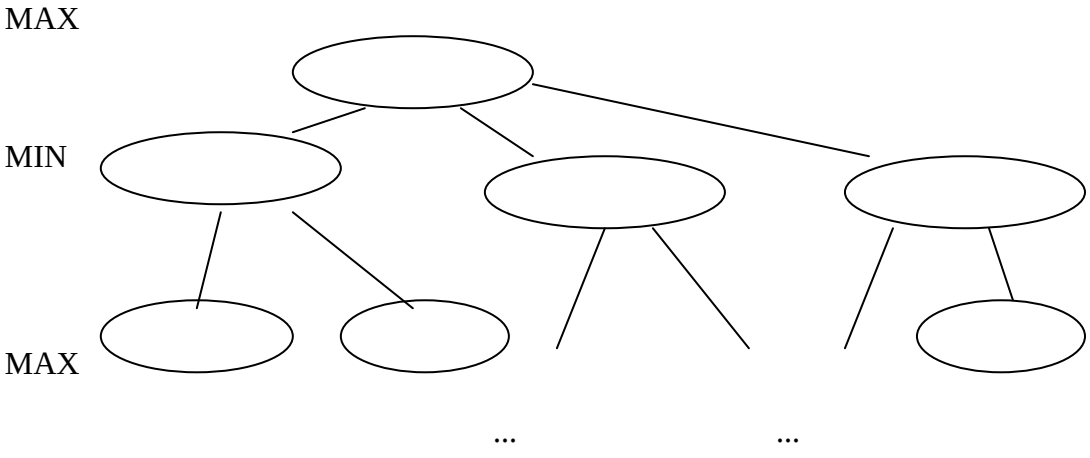
En concreto, en los juegos por turnos, cuando la máquina tenga el turno, deberá decidir, dado un estado, qué operador aplicar (dentro de los operadores que sean aplicables en ese momento, lo cual será otro tema importante a decidir).

Además, en juegos de dos oponentes y por turnos, suponemos que para cada final de partida, se le puede asignar un valor que llamaremos resultado. A los dos jugadores les llamaremos MAX y MIN (el objetivo de MAX es maximizar el valor de una partida, lo cual hace que él gane y su oponente pierda, y MIN lo contrario). Por convenio, si el resultado es positivo, diremos que MAX ha resultado ganador, en caso de ser cero diremos que ha habido un empate, y en caso contrario diremos que ha ganado MIN. Cuanto mayor sea el valor absoluto del resultado, mayor será la victoria para uno u otro jugador. De esta manera, a MAX le convendrá maximizar el resultado, mientras que a MIN le conviene minimizarlo.

En estos juegos el mecanismo que se suele usar es minimax. La máquina explora todos los estados, en forma de árbol. Cada nodo será un estado, y cada arista entre nodos representará un operador que se puede aplicar al estado “padre” de la arista. Los nodos hoja serán los finales de partida, y se valorarán con su resultado. Estos valores serán ascendidos hasta la raíz de la siguiente forma. Para cada nodo interno se supondrá que el jugador que tiene el turno realizará la acción que más le conviene, esto es si el turno es de MAX, el valor del nodo será el máximo de los valores de sus hijos. Por el contrario, si el turno es de MIN el valor ser el mínimo.

Por tanto, el valor de un estado del juego, dado por el algoritmo minimax representa el resultado de la partida, suponiendo que en cada jugada, el jugador haga la mejor jugada posible. El sistema, dado un estado, verá de entre todos los operadores que puede aplicar, qué operador le lleva a un estado cuya valoración sea más favorable para el jugador correspondiente. Dicho operador será la jugada elegida por el sistema en un punto dado.

Minimax:



Cabe destacar que el algoritmo “minimax con poda alfa-beta” se comporta igual que el algoritmo minimax, pero es más eficiente. Éste se basa en la poda de las zonas del árbol que no influyen, debido a la exploración que se lleva hasta el momento.

Sin embargo, en la mayoría de juegos, dado el gran factor de ramificación y profundidad de estos supuestos árboles, aplicar el algoritmo minimax como lo hemos explicado anteriormente es inabordable. Para ello usaremos lo que, en la traducción del libro *“Inteligencia Artificial” de Russel & Norving*, se denomina decisiones imperfectas. Estas decisiones imperfectas se basan en limitar la profundidad del árbol minimax, de tal forma que las hojas que no representen un final de partida se les asigne un valor estimado. Esto lo determina la función de evaluación o función de estimación.

Esta función de estimación se basa en ciertas heurísticas que determinan la calidad de un estado para uno u otro jugador, e intentan asemejarse al valor que tendría el árbol de valoración para ese estado si se realizara la exploración completa. Esta estimación es imprecisa, ya que en ningún momento se puede saber con certeza la precisión de dicha estimación.

Esta función de estimación suele tener en cuenta diferentes factores (por ejemplo, en el ajedrez sería suma de valores de todas las fichas, quién ocupa las casillas centrales del tablero, etc.). La función de estimación total suele ser una función de ponderación que asigna pesos (wi) a los distintos factores (fi):

estimación(estado) = w1 * f1(estado) + ... + wn*fn(estado)

Nuestro proyecto usa minimax con decisiones imperfectas, y que será la programación evolutiva la que calcule dichos pesos “wi”.

2.1.2 Introducción a la programación evolutiva.

ORÍGENES

La principal característica de la programación evolutiva es que está inspirada en la naturaleza, y en como resuelve ésta los problemas básicos para la supervivencia y evolución de una especie de individuos. Es por tanto que es una forma de programar basada en la evolución natural de los organismos vivos.

En la naturaleza todos los seres vivos se enfrentan a problemas que deben resolver con éxito, como conseguir más luz del sol, o cazar una mosca. La Computación Evolutiva interpreta la naturaleza como una inmensa máquina de resolver problemas y trata de encontrar el origen de dicha potencialidad para utilizarla en nuestros programas.

Bajo condiciones ambientales ventajosas, la población tiende a crecer a ritmo exponencial, y dado que los alimentos, espacio físico, etc. no lo hacen en la misma proporción, nacerán muchos más individuos de los que es posible que sobrevivan. En consecuencia, se desprende que un individuo, si actúa de un modo provechoso para él, tendrá una mayor probabilidad de sobrevivir, y será seleccionado naturalmente.

La teoría de la selección de las especies (Darwin) sostiene que aquellos individuos de una población que posean los caracteres más ventajosos dejarán proporcionalmente más descendencia en la siguiente generación; y si tales caracteres se deben a diferencias genéticas, que pueden transmitirse a los descendientes, tenderá a cambiar la composición genética de la población, aumentando el número de individuos con dichas características. De esta forma, la población completa de seres vivos se adapta a las circunstancias variables de su entorno. El resultado final es que los seres vivos tienden a perfeccionarse en relación con las circunstancias que los envuelven.

Podemos decir que la Computación Evolutiva surge a finales de los años 60 de mano de John Holland. Fue él quien planteó la posibilidad de incorporar estos mecanismos de selección y adaptación que muestra la naturaleza a programas que simulasen una Inteligencia Artificial. Como es de suponer, dada la poca capacidad computacional de los ordenadores de aquella época, este estudio fue meramente académico, y no fue hasta mediados de los 80 cuando se pudieron empezar a aplicar estos algoritmos en la resolución de ciertos problemas hasta entonces inabordables.

En la actualidad podemos encontrar estas técnicas funcionando con éxito en aplicaciones tales como:

- Decisión y Estrategia:
    - Toma de decisiones financieras (presupuestos)
    - Búsqueda de reglas en juegos
    - Experimentación de alternativas de marketing
    - Gestión de franquicias

- Diseño y parametrización:
  - Diseño de pistas de circuitos integrados VLSI
  - Parametrización de Sistemas (configuraciones de equipos hardware)
  - Diseño de redes (telecomunicaciones, carreteras) Ubicación de nodos
- Planificación y asignación de recursos:
  - Asignación del orden de N procesos en M CPU
  - Ordenación (ordenar cajas en un almacén)
  - Asignación de horarios de clase o turnos de trabajo en un hotel
  - Resolución de sistemas de ecuaciones no lineales
  - Enrutamiento
  - El problema del viajante de comercio (TSP, Travel Salesman Problem)
  - Predicción
  - Selección de combinación de métodos de realización de series temporales

ESTRUCTURA

"Tenemos un problema ¡Qué fastidio! Bueno, vamos a intentar solucionarlo. ¿Sabemos cómo hacerlo? No. Entonces, a probar. ¿Podemos generar varias soluciones válidas a ese problema? Ya, claro que podemos, pero van a ser todas muy malas soluciones. Bueno, no importa. Las generamos: 40, 100, las que sean. ¿Alguna es mejor que otra? Todas son malas, pero unas son menos malas que otras. Cogemos las mejores, las otras las eliminamos. ¿Y ahora qué? Bueno, las podemos coger de dos en dos y mezclarlas a ver si sale algo bueno ¿Sí? Bueno, a veces funciona. Vamos a hacerlo otra vez. Y otra. También podemos mezclarlas de otra forma. ¡Oye esto se estanca, ahora son todas iguales! Entonces, vamos a coger de vez en cuando alguna de las malas, no sólo de las buenas. ¿Y si hacemos pequeños cambios al azar en pequeñas zonas de alguna solución, a ver si alguna da en el clavo? Vale... ¡Oye, esto está mejorando!" [REF 1]

En un Algoritmo Evolutivo tenemos que definir una estructura de datos que admita todas las posibles soluciones a un problema. Cada elemento perteneciente a una de estas estructuras será una solución al problema, que podrá ser mejor o peor (tendremos que analizarlas).

Solucionar el problema consistirá en encontrar la solución óptima, y por tanto, los Algoritmos Evolutivos son en realidad un método de búsqueda. Pero no solo son un método de búsqueda, sino algo más, ya que las soluciones al problema son capaces de reproducirse entre sí, combinando sus características y generando nuevas soluciones.

En cada ciclo se seleccionan las soluciones que más se acercan al objetivo buscado, "eliminando" el resto de soluciones (ya que a veces no se eliminan, sino que se mantienen en la población pero con muy baja probabilidad de ser seleccionadas). Las soluciones seleccionadas se reproducirán entre sí, permitiendo de vez en cuando alguna mutación o modificación al azar durante la reproducción.

En definitiva, los pasos que realiza un algoritmo genético son:

Se genera un conjunto de N soluciones válidas al problema. Valores típicos de N son desde 1 hasta 200. Cada una de estas entidades representa una solución distinta a un mismo problema. Estas entidades se pueden generar al azar. También se pueden generar a partir de soluciones ya conocidas del problema, que se pretendan mejorar, o mediante posibles "trozos de soluciones" (más conocidos como bloques constructores), es decir, con lo que creemos que pueden ser elementos componentes de la solución final aunque no sepamos cómo combinarlos.

Se evalúan las soluciones existentes y se decide, en función de esta evaluación, dos cosas. Por una parte, qué soluciones van a sobrevivir y cuáles no; y por otra, cuáles se van a reproducir y cuáles no. En el caso de reproducirse, se especifica la potencia reproductora de la solución, de forma que es posible decidir que unas soluciones se reproduzcan más que otras.

Tal como se ha establecido en el paso anterior, se eliminan ciertas soluciones y se mantienen otras, y se efectúa la reproducción o recombinación de genes (normalmente por parejas) de las entidades existentes. Por ejemplo, se realizan cruzamientos de patrones a partir de cierto punto elegido al azar, de forma que los nuevos patrones posean un segmento de cada uno de los progenitores.

Se efectúan mutaciones (cambios al azar en los genes) de los nuevos patrones, según una tasa determinada. Algunos estudios aconsejan realizar mutaciones también sobre los padres.

Se vuelve al paso 2 hasta que se cumpla el criterio de parada, que puede ser por ejemplo, que el peso de la mejor entidad supere cierto valor, o que se halla alcanzado el límite máximo de generaciones.

BASES DE LA PROGRAMACIÓN EVOLUTIVA

Tal y como dijimos anteriormente, los Algoritmos Genéticos (AG) se pueden entender como una estrategia más de búsqueda, al igual que las conocidas "a lo Ancho", "en Profundidad", Escalar-Colinas, Mejor-Primero, y A*. El algoritmo selecciona una serie de elementos tratando de encontrar la combinación óptima de estos, utilizando para ello la experiencia adquirida en anteriores combinaciones, almacenada en unos patrones genéticos. Sin embargo, en este caso, las posibilidades de esta metodología transcienden de la búsqueda heurística, debido a una razón de peso: el algoritmo no trabaja Arriba-Abajo, sino Abajo-Arriba: es decir, no es necesario disponer de un conocimiento profundo del problema a resolver, para que éste pueda ser descompuesto en sub-problemas, sino que se parte de estructuras simples que interactúan, dejando que sea la evolución quien haga el trabajo.

Gracias a esto sólo es necesario ser capaces de identificar en qué casos las soluciones intermedias se acercan o se alejan de la solución buscada, para que el sistema se perfeccione automáticamente, desarrollando sus propios comportamientos, funcionalidades y soluciones. Ciertamente, se puede decir con humildad que esta técnica ha demostrado ya su relativa utilidad, en la vida biológica, creándonos a nosotros mismos.

REPRODUCCIÓN: La reproducción sexual tiene la ventaja de que combina dos (o más) bloques de genes que ya han probado su efectividad, de manera que es posible que el patrón resultante herede las características deseables en ambos progenitores, en cuyo caso será seleccionado, aumentando la existencia de dichas características en la población. Precisamente, más del 95% de las especies vivas en el planeta poseen reproducción sexual.

MUTACIÓN: Aunque es poco probable que un cambio cualquiera en un gen sea beneficioso, las mutaciones producen varias ventajas: dificultan el estancamiento de la población aumentando la variedad, de manera que ante un cambio brusco del entorno, la comunidad pueda adaptarse a la nueva situación; además, permiten la generación de cualquier segmento de patrón, de manera que si en el conjunto inicial de patrones no se encuentran todos los elementos necesarios para formar el óptimo, sea posible llegar a ella.

SELECCIÓN: Aquí existe una gran diferencia entre la selección que se presenta en la naturaleza y la que nosotros establecemos en nuestras granjas o programas. Nosotros podemos seleccionar para la reproducción los seres que más nos interesan, ya sean las

vacas que más leche dan o los agentes software que mejor resuelven un problema. En cambio, en la naturaleza no existe -en principio- una inteligencia exterior que determine la dirección de la evolución. Y sin embargo, la evolución en la naturaleza sí se produce. Todo esto nos lleva a pensar que, aparentemente, es mejor guiar nosotros la selección y acelerar la mejora de una población de individuos de esta forma. Sin embargo, es algo que no está demostrado, ya que uno de los principales problemas de los algoritmos genéticos es el que se puede expresar con la siguiente pregunta, ¿Cómo han podido generarse gradualmente órganos complejos, como las plumas para el vuelo, si sólo el órgano completamente funcional es útil al individuo? No siempre es fácil explicar todos los rasgos que presenta un organismo con un mecanismo adaptativo.

PRINCIPALES CARACTERISTICAS

¿CUÁNDO PODEMOS APLICAR UN ALGORITMO EVOLUTIVO?

En primer lugar, en el problema a resolver, la meta ha de poder ser observada en grados cualitativamente comparables. Por otra parte, las principales dificultades a la hora de implementar un algoritmo genético son:

- Definir una estructura de datos que pueda contener patrones que representen, la solución óptima buscada (desconocida) y todas las posibles alternativas de aproximaciones a la solución.

- Definir un tipo de patrón tal que si un patrón es seleccionado positivamente, que esto no sea debido a la interacción de los distintos segmentos del patrón, sino que existan segmentos que por sí solos provocan una selección positiva.

- Definir una función de evaluación que seleccione los mejores individuos.

Las condiciones que debe cumplir un problema para ser abordable por algoritmos evolutivos son:

- El espacio de búsqueda deber ser acotado.

- Debe existir un procedimiento relativamente rápido que asigne un grado de utilidad a cada solución propuesta, de forma que este grado de utilidad asignado corresponda, o bien directamente con la calidad de la solución en cuanto al problema a resolver, o bien con un valor de calidad relativo al resto de la población que permita obtener en el futuro mejores soluciones.

- Debe existir un método de codificación de soluciones que admita la posibilidad de que los cruzamientos combinen las características positivas de ambos progenitores. Este método debe permitir también aplicar algún mecanismo de mutación que sea capaz de conseguir tanto soluciones muy dispares respecto de la solución sin mutar, como muy parecidas a ésta.

¿QUÉ PODEMOS VARIAR PARA REFINAR?

- Número de entidades inicial
- Método de generación de estas entidades (al azar o según una función)
- Tasa de reproducción (constante, variable según una distribución estadística determinada)
- Tasa de defunciones
- Método de reproducción: asexual, sexual con 2 progenitores, 3, etc.
- Tasa de mutación (constante, variable)
- Tipos de mutaciones producidas
- Permitir o no la transmisión genética de la experiencia
- Todo tipo de restricciones que hagan que el funcionamiento de cada uno de los aspectos anteriores dependa de ciertas condiciones.

OPCIONES DE UN ALGORITMO EVOLUTIVO

Opciones Generales:

- Número de entidades.
- Número de elementos (genes, reglas) por cada agente.

Método de Evaluación: Asignar un peso

- Desordenar las entidades antes de evaluarlas
- Diferentes formas de modificación de los pesos después de la evaluación. Por ejemplo, el peso de una entidad se puede calcular independientemente de las

demás entidades, o se puede modificar posteriormente este valor, disminuyendo el peso si existe otra entidad muy parecida, analizando para ello un cierto subconjunto de la población vecina.

Método de Selección: ¿Quién muere? ¿Quién se reproduce?

- Con o sin reemplazamiento
- Método de la ruleta
- Método de los torneos
- Seleccionar el n mejor y el m peor

Método de Reproducción: Generar y mutar nuevos hijos

- Los padres pueden tomarse por parejas o en grupos más numerosos, elegidos al azar o en orden de pesos
- En el caso de detectar que los progenitores son muy parecidos, se puede realizar una acción especial, como incrementar la probabilidad de mutación
- Las entidades pueden comunicar a otras su conocimiento, ya sea a toda o a una parte de la población, directamente o a través de una pizarra, (una especie de tablón de anuncios)

Método de recombinación de genes: se pueden tomar genes de uno u otro progenitor al azar, en un cierto orden, con uno, dos o más puntos de corte, etc.

- Tasa de mutación variable
- Fijar una tasa de mutación diferente para cada individuo o incluso para cada gen

Hacer que sea más probable que se produzca una mutación en un gen si en su vecino ya se ha producido

- Sustituir por mutaciones genes sin utilidad, como reglas incorrectas o repetidas
- Tipos de mutaciones

2.1.3 Introducción a los Juegos de Estrategia

Los juegos de estrategia son uno de los estilos de juego más completos e importantes dentro del mundo de los videojuegos. Hoy en día la mayoría permiten muchas posibilidades tales como el juego online, múltiples bandos aliados o enemigos combatiendo a la vez, vista isométrica, gráficos espectaculares,...

Repasaremos algunos de los más conocidos juegos que hay en el mercado a día de hoy:

2.1.3.1 Age of Empires

Creado por los estudios Ensemble, actualmente está en una fase de desarrollo bastante avanzada su tercera entrega (la salida se espera para la segunda mitad del 2005).

Una captura de pantalla del programa funcionando es:

En la segunda parte, el objetivo era que los jugadores comandaran hasta la victoria una de las 13 civilizaciones más poderosas. Estas incluyen: los Francos, Japoneses, Bizantinos, Vikingos, Mongoles, Celtas,...

Cada civilización tiene atributos únicos, construcciones y tecnologías también así como una unidad de combate basada en su antecedente histórico. Esta saga, apadrinada por uno de los grandes de la informática como es Microsoft, es una de las sagas más importantes dentro de los juegos de estrategia. Su forma de ver el terreno la han hecho uno de sus elementos más característicos. Ofrece bastante información, imágenes, música,... en la web para consultar.

Los requerimientos necesarios, para poder hacernos una idea de lo que deberíamos poseer, son:

- PC multimedia con Pentium 166MHz o un procesador más rápido.
- Sistema operativo Microsoft Windows 95, Windows 98, Windows NT 4.0 con Service Pack 5.
- 32 MB de RAM
- 200 MB de espacio de disco duro; 100 MB adicionales de espacio de disco duro para archivos de intercambio.
- Monitor Super VGA con soporte de resolución de 800x600.
- Tarjeta local de video bus que soporte 800x600, resolución de 256 colores y 2 MB de VRAM
- Unidad de CD-ROM drive de dos bits.

- Microsoft Mouse o un dispositivo apuntador compatible.

- Módem 28.8Kbps (o más veloz) para Internet o juego de uno contra otro.

- Tarjeta de audio con audífonos o bocinas.

- Para acceder a MSN Gaming Zone necesita Internet Explorer 3.02 o posterior o Netscape Communicator 4.0 o posterior.

- Se requiere acceso a Internet para juego en Internet. Puede requerir el pago de una cuota por separado a un proveedor de Servicios de Internet. Pueden aplicar cargos por tiempo de conexión y telefonía local.

El tema del idioma es algo a lo que debemos los hispanohablantes irnos acostumbrando, al menos de momento. La mayoría de las empresas tanto desarrolladoras como distribuidoras son o pertenecen al mercado anglosajón. Por otro lado, la mayoría de los equipos de programación (salvo honrosas excepciones como puede ser Pyro Studios), vienen de países de habla inglesa. Por si ello fuese poco, el inglés es el idioma internacional en Internet, foco masivo de difusión para estos programas.

Casi todos los programas que vamos a ver suelen destacar las mejoras en la IA que van añadiendo siendo de los aspectos a los que más importancia se da.

Ej. “Las unidades son más inteligentes con nueva inteligencia artificial (AI) opciones como Guardia, Patrulla y Seguimiento así como múltiples niveles de agresión”.

Habitualmente, estos juegos se valen de campañas de “tutorial” para explicar a los jugadores novatos el funcionamiento de los controles, menús,... que posteriormente deberán manejar con soltura para desenvolverse sobre el terreno de batalla.

El listado de premios que ha obtenido este programa es bastante largo. Puede verse en su web:

http://www.microsoft.com/latam/juegos/age2/premios.asp

2.1.3.2 Caesar 3

Destaca sobremanera la tercera entrega de este programa. Una primera noción nos la da Internet sobre este programa. Gran juego en el que nos pondremos al servicio del Imperio Romano en misiones tan dispares como construir una ciudad para abastecer de alimentos a Roma o para defender el avance de los enemigos del Imperio. En esta misma página se destaca:

VENTAJAS:

Gran adicción, que nos hará pasar numerosas horas delante del ordenador

El nivel de dificultad, que está muy bien logrado y nos hace engancharnos aún más...

INCONVENIENTES:

Se le coge el truco y al final siempre haces lo mismo... (aunque sea después de 1000 horas de juego, bueno, no tantas!)

La lucha de los ejércitos es un poco simple así que si queréis un juego de verdadera estrategia jugar a otro...

Posteriormente, esta misma compañía descubrió un filón con este estilo de juego, y creó sucesivas sagas pero ambientadas en otros imperios como por ejemplo Faraón (Egipto), Zeus (Grecia),...

Como bien dice un usuario de un foro dedicado al tema:

“Siguiendo la estela de la saga de juegos de Sierra llamados Caesar, Faraón hace la versión egipcia de este popular juego de construcción, dirección, organización y administración de ciudades ambientado en la Antigua Roma trasladándolo esta vez al Antiguo Egipto, a tierras de los faraones.”

Un problema al que se han tenido que enfrentar continuamente estos desarrolladores ha sido a que es complicado intentar explotar la misma idea continuamente. De hecho, como dice otro usuario del foro:

"He jugado a varios juegos de Sierra, como el Caesar en todas sus versiones... Este juego, esta en la línea de los anteriores, no aporta nada nuevo, si no has jugado a versiones anteriores si te lo recomiendo..." [REF 2]

Aquí arriba tenemos una captura de pantalla a todo gas del programa del que estamos hablando. El próximo juego que vamos a comentar debemos destacar que es el más parecido al nuestro en concreto, tanto en reglas como en objetivos.

2.1.3.3 HEROES OF MIGHT & MAGIC

La historia principal de Héroes IV se divide en siete campañas para un solo jugador. La heroína protagonista responde al nombre de Emilia Nighthayen que ha salido de su reino a causa de un cataclismo y llega al vecino territorio de Axeoth. Allí reina con mano férrea el malvado Gavin Magnus que se ha aliado con las fuerzas del mal. La eterna lucha entre el bien y el mal, pero nosotros decidiremos el bando ganador según escojamos a quién ayudar. Actualmente tenemos cuatro versiones de este popular juego de estrategia. Desde la primera siempre ha tenido un estilo muy peculiar y definido, que además a título personal nos ha enganchado bastante. Es por ello que el juego que más ha influenciado en el nuestro.

La idea es básicamente la misma que en la mayoría de estos juegos, eliminar al resto de oponentes. Esta idea es un clásico en todos los juegos de estrategia que hemos visto hasta ahora, y es algo que no ha evolucionado demasiado.

Actualmente han surgido ya lo que se podrían considerar primeras imágenes de la quinta parte de este maravilloso programa. Un ejemplo es el siguiente:

Está ambientado en una especie de edad media habitada por todo tipo de seres extraños: desde dragones a hidras pasando por górgonas, enanos, cíclopes, zombies... y una serie de "héroes" que son en torno a los que gira el juego. Cada uno de esos héroes se mueve por el escenario, buscando artefactos, recogiendo recursos, reclutando otras criaturas, y en general haciendo de brazo armado de los jugadores. Los héroes adquieren habilidades, aprenden, suben de nivel…

Es un juego lleno de colorido, basado en mundos fantasiosos, en el que turno a turno vas explorando enormes mapas con montones de monstruos para encontrar tesoros y objetos de diferentes tipos, y vas mejorando tu héroe a lo largo de las diferentes luchas para hacerle más poderoso. Esta creado por la empresa 3D0. Se supone que se puede jugar en red, pero parece que aún no está todo resuelto para Mac. Se juega en tercera persona (del plural) sobre mapas de perspectiva isométrica y muy alta calidad gráfica. En ese sentido es un poco como un juego de tablero.

El otro elemento fundamental son las ciudades. Aquí, Héroes recordaba a Civilization antes... pero cada vez menos. Las ciudades se conquistan, pierden, administran... En cada una (y hay decenas de tipos) se pueden construir distintos tipos de edificios que proporcionan servicios, recursos, o directamente criaturas para tus ejércitos. Hasta ahí normal, administración de recursos. Pero es que las ciudades de Héroes III tienen tal nivel gráfico que se convierten en una de las atracciones del juego. Puedes ver cada edificio que construyes y gestionarlo directamente.

Finalmente, están los escenarios. El juego se puede tomar por partes aisladas (escenarios) o bien mediante historias que forman parte de un argumento coherente en el que los "buenos" reconquistan un territorio que ha caído en las manos de fuerzas oscuras.

Los escenarios pueden ser de varios tipos, desde los tradicionales que exigen eliminar a los demás contendientes pasando por los que permiten ganar con un objetivo alternativo, hasta los que exigen condiciones concretas (captura tal ciudad) o plazos definidos (elimina a tal criatura en tantos turnos). Los mapas varían en extensión, desde pequeños y rápidos hasta XL o casi eternos, y están plagados de objetos y de obstáculos.

Para aumentar la variedad, están los distintos niveles de dificultad. El efecto principal de variarlo es que aumenta o disminuye la abundancia de recursos y la agresividad de los contrincantes. El nivel fácil no es relajado, el nivel normal está muy bien... y hay hasta "imposible" para el que de verdad quiera retos difíciles.

2.1.3.4 CIVILIZATION

Hace tiempo ya de la aparición de este juego, uno de los más aclamados y más jugados por todo tipo de usuarios del PC. El realizador de este juego Sid Meier's, es uno de los mejores productores de juegos para el ordenador, títulos posteriores como Colonization, Civilization II (juegos que también comentaremos aquí), le han dado el prestigio y el puesto que se merece.

La temática de este juego de Microprose (compañía importantísima en este tipo de juegos), es dominar el mundo. A grandes rasgos el juego discurre desde la edad de piedra, año 3300 antes de Jesucristo, hasta la época actual.

Para comenzar se dispone de una unidad de colono, con la que debemos fundar nuestra primera ciudad y empezar a desarrollarnos como civilización. Hay muchos tipos de unidades, éstas varían según la época en la que estemos y los progresos técnicos y conocimientos que vayamos adquiriendo. Pueden ser desde unos simples guerreros con lanzas, hasta unos temibles tanques, pasando por caballeros, milicianos, guerreros con lanzas y espadas, catapultas y además de tipos marinos, como pueden ser galeones, cruceros, portaaviones, submarinos, etc.

El terreno por el que discurre el juego puede ser un mundo aleatorio, uno creado por el propio usuario o el mapa terrestre. Este está dividido en celdas o, más precisamente, las unidades están condicionadas por sus movimientos, lo cual hace que parezca el terreno dividido en celdas invisibles

Las ciudades que vamos construyendo, o mejor dicho fundando, deben ser acondicionadas por elementos comunes, como pueden ser bancos, granero, barracones, etc... Estos elementos a la vez de apaciguar el ánimo de los ciudadanos nos benefician de una serie de factores sobre nuestra economía y sobre capacidades de nuestras unidades. Cabe destacar también que es posible construir determinados monumentos históricos o de envergadura, los cuales son elogiados y nos regalan modificadores extras también sobre las unidades y ciudades.

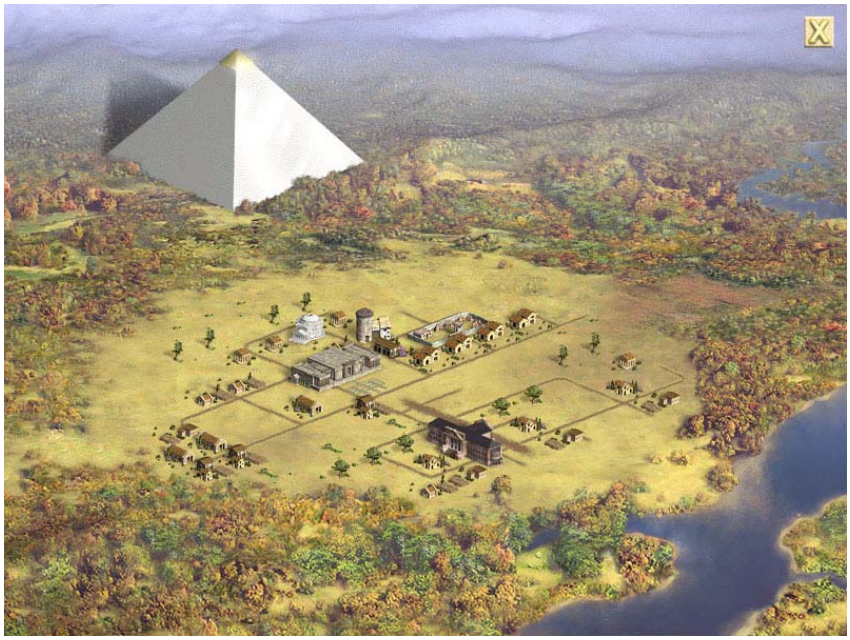
En el juego hay que controlar los descubrimientos que vamos haciendo, las relaciones con otras civilizaciones, las cuales nos pueden aportar a su vez nuevos conocimientos, tributos o en su defecto plantearnos una guerra a muerte.

Como se puede ver este juego, es un sinfín de actividades, llegando en algunos momentos a no saber como dominar todos los factores que se nos presentan, la capacidad del jugador de afrontar las continuas revoluciones en las ciudades, el golpeo incesante de unidades enemigas viene muy condicionado por el nivel de dificultad que se elija, entre 5, de caudillo a emperador, y del numero de civilizaciones junto con el que se quiera jugar, llegando a poder ser estas un total de 10.

Ya para finalizar conviene decir que este juego es uno de los pioneros dentro del género, sentó muchas bases para futuros juegos, y ha sido y seguirá siendo, rey entre reyes. Destacar también en el aspecto gráfico y sonoro no esta muy logrado, aunque ello no reste emoción y jugabilidad al mismo. Este es un juego echo por y para jugar, no para deleitarse mirando gráficos, ya que estos son los justos. Cabe decir que hay que jugar a él para saber lo que engancha y decir también que todo estratega que se precie debe haber jugado en alguna ocasión a este juego, porque si no es así, no sabe lo que se pierde.

Es del año 1991, así que lleva ya un tiempecito el abuelete ☺.

En la última versión que tenemos del juego, las cosas han cambiado un tanto:



2.1.3.5 WARHAMMER 40000

Cada una de las razas de este juego viene bastante bien detallada, así como toda la historia y el guión que lo rodea, demostrando que hay mucho más que el videojuego detrás de este universo. Como ejemplo, podemos tomar la raza Marines Espaciales

[REF 3] :

Es una fuerza creada por el Emperador y curtida en la forja de la guerra, los MARINES ESPACIALES son monjes guerreros dedicados a servir a su creador y hay un fuerte lazo de honor que les une a su sagrada tarea. Aunque son relativamente pocos en número, los MARINES ESPACIALES están siempre en el frente luchando ferozmente por la supervivencia de la Humanidad contra los ataques viciosos y constantes de los enemigos de la Humanidad.

La superioridad sobrehumana de los Marines Espaciales es el resultado de numerosas modificaciones genéticas, de acondicionamiento psíquico, de entrenamiento físico riguroso y de una fe incuestionable. Un Marine luchando en solitario es por sí solo un guerrero formidable, pero cuando lucha al lado de un millar de hermanos (en organizaciones conocidas por el nombre de CAPÍTULOS), estos entonces se convierten en el arma férrea de la justicia - son las espadas vivas del Emperador, que realizan su voluntad y protegen a su gente a lo largo y ancho del Imperio.

Se podría decir que un Marine Espacial sirve al Emperador hasta su último aliento, pero eso sería infravalorar su devoción, ya que algunos le continúan sirviendo incluyo más allá de la muerte - se trata de los mortíferos Dreadnoughts. Un Dreadnought es un vehículo inteligente que lucha, se desplaza y es particularmente sobresaliente en la destrucción de objetivos difíciles, como edificios y vehículos. Los Dreadnoughts son pilotados por un Marine Espacial que ha sido gravemente herido en el campo de batalla, y que ha cambiado la certeza de una muerte segura por un entierro permanente en el interior de un sarcófago blindado.

Y los Dreadnoughts son sólo una de las muchas unidades que componen un Capítulo de Marines Espaciales.

Cada Capítulo tiene un líder que es el Señor del Capítulo, cuyo heroísmo y extraordinaria habilidad hace que sobresalga incluso entre sus excepcionales compañeros. Veterano, aguerrido en mil batallas, el Señor del Capítulo siempre está en el frente más avanzado del campo de batalla, dirigiendo cada carga y liderando a sus tropas con total autoridad.

El BIBLIOTECARIO es el recipiente de la sabiduría antigua del Capítulo. Puede canalizar su potente energía psíquica para hacer que desaparezcan demonios, para someter enemigos, para proteger a sus tropas de ataques psíquicos o para aumentar su resistencia física en el fragor de la batalla.

El APOTECARIO es uno de los especialistas clave en el Capítulo de Marines Espaciales. Aunque empuña una espada de sierra con habilidad y precisión, la misión principal del Apotecario es sanar las heridas de sus hermanos Marines en el fragor de la batalla. Cuando un hermano sucumbe en combate, la misión sagrada del Apotecario es recoger su material genético. Las semillas genéticas, del caído son un material único y común a todos los miembros del Capítulo. Estas semillas pueden usarse de nuevo para crear nuevos Marines Espaciales y así continuar la historia del Capítulo. Las escuadras DE INFANTERÍA constituyen el núcleo de los Marines Espaciales. Otras tropas importantes son los EXPLORADORES, que se desplazan con velocidad y sigilo; y los Exterminadores, que van armados hasta los dientes y poseen una gran experiencia y que pueden ser teletransportados directamente al combate. Cada escuadra desempeña una función vital dentro del Capítulo.

Todos los Marines, ya sean eruditos o soldados de infantería, se encuentran bajo el mando de un Comandante, conocen el propósito de su misión y sirven con heroísmo y devoción absoluta a su Capítulo y a la gloria del Emperador.

Son los Marines Espaciales…

…!No conocen el miedo!

Después de toda esta larga parrafada, podemos decir que poco aporta nuevo a los anteriores juegos de estrategia. De hecho, es un individuo bastante menos conocido salvo por sus admiradores jeje, que los anteriormente citados.

Otro modo muy interesante de ver las características de cada programa, y que evidentemente deberíamos de soportar como desarrolladores en el caso de que nuestro programa estuviera incluido en el mercado, es la comparativa mediante tablas de atributos de varios de ellos. Como ejemplo, el siguiente:

2.1.3.6 COMPARATIVA DE 5 JUEGOS DE ESTRATEGIA

Tzar, Faraón, Age of Empires II, Alpha Centauri, Home World, Pizza $yndicate.

NOMBRE	Tzar	Faraón	Age of Empires II	Alpha Centauri
FABRICANT.	FXInteractive	Sierra	Microsoft	Firaxis
DISTRIBUID.	FXInteractive	Havas Inter.	Microsoft	EA
Teléfono info	91 799 01 10	91 383 26 23	902 197 198	91 304 70
Fax	91 799 12 25	91 383 24 37	91 803 83 10	91 754 52 65
Edad recom.	No especif.	No especif.	No especif.	No especif.
SERVICIO	3 % NOTA	3 % NOTA	3 % NOTA	3 % NOTA
Telf. Ayuda	91 799 01 10	91 383 27 60	902 197 198	91 304 70 91
Serv. Online	www.fxplanet.com	www.havasinteractive.com	www.microsoft.es	www.ea.com
INSTALAC.	11 % NOTA	11 % NOTA	11 % NOTA	11 %

				NOTA
Autoarran.	Si	Si	Si	Si
¿Indica espacio necesario?	Si	Si	Si	No
¿Indica espacio libre?	Si	No	No	No
Manual	Muy comple-to/muy claro	Muy comple- to/muy claro	Muy comple-to/muy claro	Muy comple-to/muy claro
Idioma	Español	Español	Español	Español
DirectX	DirectX 7	Direct X 6.2	DirectX 6.1	DirectX 6
MANEJO	10 % NOTA	10 % NOTA	10 % NOTA	10 % NOTA
Periféricos	Tecl., ratón	Tecl., ratón	Tecl., ratón	Tecl., ratón
Opc. Conf.	Muchas	Muchas	Muchas	Muchas

Idioma	Español	Español	Español	Español
CALIDAD	26 % NOTA	26 % NOTA	26 % NOTA	26 % NOTA
Gráf./video	Muy buenos	Buenos	Buenos	Muy buenos
Sonido/voz	Buenos	Muy buenos	Bueno/No	Muy buenos
Rendimiento	Muy bueno	Muy bueno	Muy bueno	Muy bueno
Jugabilidad	Muy fácil	Muy fácil	Muy fácil	Fácil
Amplitud	Amplio	Amplio	Muy amplio	Muy amplio
DIVERSIÓN	Redondo en todo	Muy adictivo y completo	Un clásico renovado	Calidad ante todo
Nota parcial	9’00	8’90	8’64	8’32
CALIDAD	SOBRESA-LIENTE	NOTABLE	NOTABLE	NOTABLE

PRECIO/CAL	SOBRESA-LIENTE	BIEN	BIEN	BIEN
PRECIO	2.995 PTAS	6.995 PTAS	7.990 PTAS	6.990 PTAS

Ganador calidad: Tzar

Ganador precio/calidad: Tzar

2.2 Estudio: Alternativas de IA y Programación Evolutiva (PE), y el motivo de las opciones elegidas.

En este apartado, entre otras cosas, consideramos las alternativas de incluir IA y PE que estudiamos y posteriormente explicaremos en profundidad la alternativa que hemos elegido para desarrollar nuestro proyecto.

2.2.1 ¿Por qué usar PE en lugar de Algoritmos deterministas?

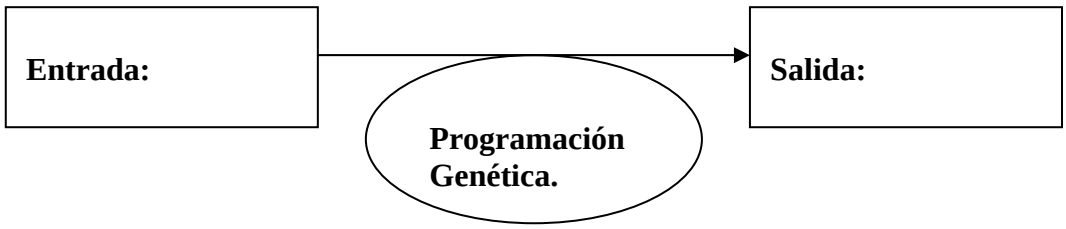
Los algoritmos deterministas siempre devuelven el mismo resultado, salvando los comportamientos pseudo aleatorios que le demos, por ejemplo con las funciones random en los lenguajes de programación de alto nivel (pueden incluir ciertos rasgos de aprendizaje que les haga cambiar su comportamiento). La PE usa algoritmos no deterministas. De esta manera, el sistema en el juego podrá adoptar infinidad de formas diferentes de jugar, de tal forma que el jugador humano perciba en su oponente simulado por el sistema una forma de jugar tan rica y variada como podría ser la de un oponente humano.

2.2.2 Alternativas que consideramos en el uso de PE

Como objetivo inicial, decidimos que nuestro proyecto usara la programación evolutiva. También hemos determinado ya que nuestro juego es de dos jugadores y por turno. Teniendo en cuenta estos dos factores existen, entre otras, las siguientes alternativas:

2.2.2.1 Uso de la “Programación Genética”

Se denomina “programación genética” al uso de la programación evolutiva para obtener como resultados programas. Esto es, el programa, que usa la PE y es programado por humanos, genera como salida un programa que resuelve un problema dado. Esto es, el segundo programa no ha sido programado directamente por los programadores humanos, sino que ha sido generado por otro programa. Quizás gráficamente se entienda mejor:



En el caso de nuestro juego el problema sería obtener un comportamiento en un juego, de tal forma que maximizase las partidas “ganadas” contra jugadores humanos. La salida del sistema sería un programa que llevase a cabo el jugar contra humanos. Dicho programa se debería parecer a “minimax” o “minimax con poda alfa-beta”, aunque con distintos matices que lo hicieran más variado.

Es importante destacar que para poder llevar a cabo de manera práctica la programación genética, el programa resultante no estará escrito en un lenguaje de programación comercial como puede ser C, Java o Pascal, sino que dicho programa estará escrito en un lenguaje con un repertorio muy reducido de instrucciones, y especialmente pensado para el tipo de problema que estamos intentando resolver. Obsérvese por ejemplo *la práctica tres de la asignatura de PE en la Facultad de Informática de la UCM, en el año 2003-2004*. En esta práctica se buscaba ordenar una pila de letras de tal forma que formase la palabra UNIVERSAL, cambiando las letras desde la mesa a la pila y viceversa. El lenguaje tenía un conjunto de tan sólo siete instrucciones.

En el conjunto de instrucciones que se usaran como lenguaje para el programa para juegos de dos adversarios, para un buen funcionamiento deberíamos incluir:

Una instrucción que dado un estado del juego devuelva una estimación aceptable. Esta instrucción sería importante implementarla bien, y ya supondría una gran ventaja. De la misma forma tendríamos que incluir una instrucción que determine los operadores

aplicables para un estado, funciones que devuelvan el máximo y el mínimo de una lista, y otras instrucciones de control muy sencillitas.

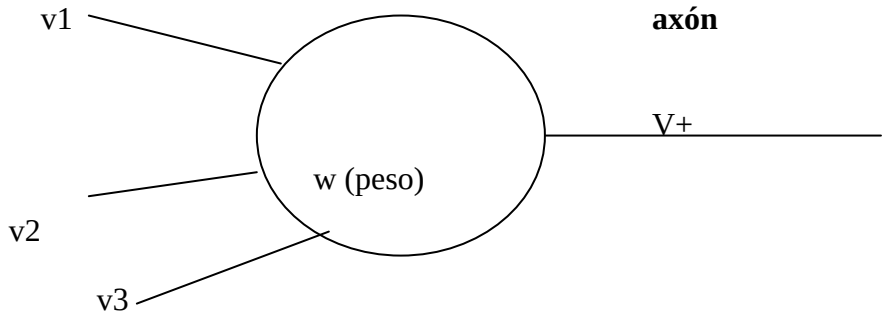
Los autores *R. GroB, k, Albrecht, W.Kantschic y W. Banzhaf de la universidad Dortmund en Alemania* [REF 4], usaron la Programación Genética, de forma muy similar a lo anteriormente descrito, para desarrollar un programa que jugará al ajedrez. En su trabajo muestran los excelentes resultados que obtuvieron.

2.2.2.2 Uso de PE para el cálculo de pesos de una Red Neuronal

Para entender esta alternativa, lo más conveniente es que expliquemos brevemente en que consiste una red neuronal. Cómo para todo programa o problema básico, dada una entrada debemos obtener una salida. Pues bien las redes neuronales software imitan el comportamiento de las redes neuronales humanas, por ejemplo. Cada neurona o unidad básica tiene la siguiente función:

NEURONA:

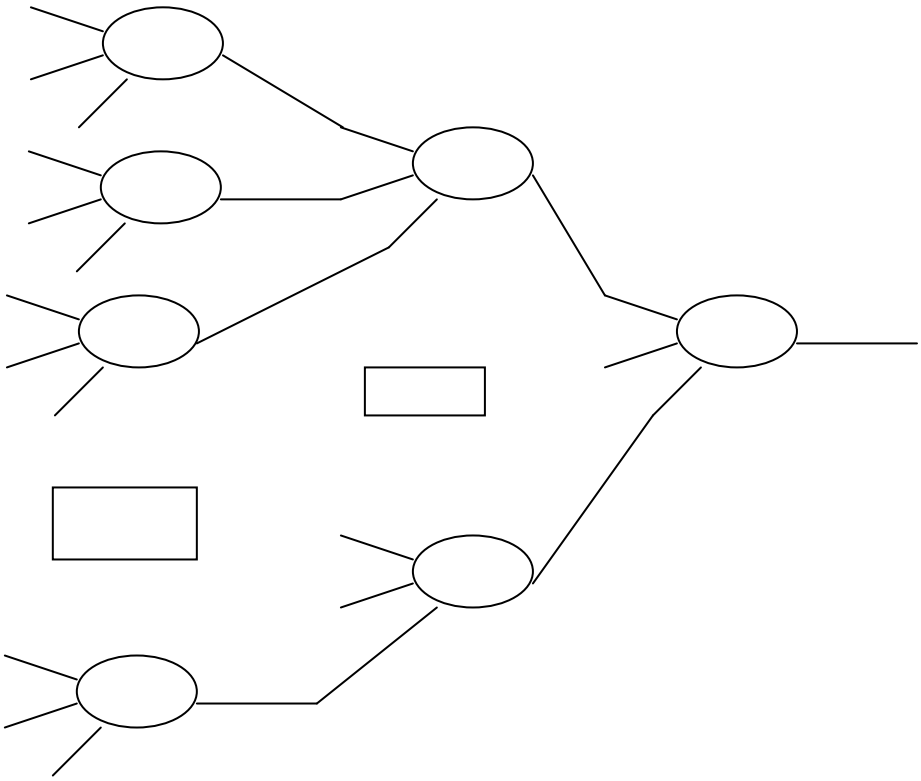
DENDRITAS



Cada dendrita recibe un valor de entrada, si dichos valores de entrada (o potenciales) suman un valor mayor o igual que el peso "w" de la neurona, emiten un valor (o potencial) constante de V+. En caso contrario el axón emite un valor (o potencial) nulo.

Las neuronas pueden ser dispuestas de muchas formas, para formar diferentes tipos de redes neuronales. Una de las formas más comunes es:

RED NEURONAL EN FORMA DE ARBOL: (Tree Neuronal Network)



Las funciones que pueden abarcar las redes neuronales son completas, esto es, pueden calcular cualquier función que podamos expresar formalmente. Una rápida demostración de ello, es que, para el caso particular de dos dendritas por neurona, en la que todos las valores de entrada y salida de cada neurona sean '0' o'1', para valor de w = 0,5 la neurona se comporta como una puerta OR (la salida del axón es '1' con que cualquiera de las dos dendritas valga '1'), para valor w = 1,5 se comporta como una puerta AND (es necesario que las dos dendritas de entrada tengan su valor a uno). Por tanto, un caso concreto de red neuronal serían aquellas que simulan puertas lógicas AND y OR. Como todos los informáticos sabemos, estas dos puertas lógicas son un conjunto universal. Luego queda demostrado que una red neuronal puede expresar cualquier función. Sin embargo, las redes neuronales son muy ricas, ya que se pueden tomar infinidades de valores w, que las hagan actuar de muy diversas formas.

Pues bien, un algoritmo evolutivo podría calcular los pesos de las neuronas que proporcionan una función de mayor calidad. Es decir cada individuo de la población individual sería un conjunto de pesos para la red neuronal.

En nuestro caso de juegos de estrategia, la función que calcularíamos por este método sería la función de estimación para determinar la calidad de un estado del juego. Para la función de adaptación, usaríamos un algoritmo minimax clásico usando los pesos anteriores. Se realizarían torneos entre las distintas "redes neuronales" (en realidad entre los pesos de las neuronas, ya que la red neuronal sería la misma) para ir obteniendo por programación evolutiva el mejor conjunto de pesos que haga a la red neuronal comportarse mejor.

Lo realmente glorioso y fascinante de éste método y distinto a la mayoría de posibles alternativas es que no se necesitaría conocimiento experto humano acerca de estimaciones del estado posible, sino que la propia red neuronal entrenada con Programación Evolutiva es la que determina dicha estimación.

Las redes neuronales, en general no han tenido mucho éxito (en Estados Unidos ha sido una de las áreas de investigación en que más se ha invertido y de las que menos resultados productivos ha obtenido en proporción). Sin embargo, los *autores K. Chellapilla, B. Fogel* [REF 5] muestran un trabajo sobre damas, usando redes neuronales de la forma anteriormente descrita, y con un gran éxito. En su trabajo, conectaron las dendritas correspondientes de la red neuronal a las posiciones del tablero. Los valores de entrada variaban entre tres dependiendo si había una ficha de uno u otro color y si estaba vacía. Los resultados fueron más que exitosos. Aunque también cabe destacar la gran cantidad de tiempo necesario. Se necesitó un Pentium de 400Hz durante unos seis meses (con unas 840 generaciones).

Este método fue descartado para nuestro proyecto, debido a la magnitud, y al excesivo tiempo de entrenamiento, del que dada la estructura y calendario del proyecto, en ningún momento creímos disponer.

2.2.2.3 Uso de PE para el cálculo de pesos de la Función de Estimación

Una tercera alternativa consistiría en usar el algoritmo minimax clásico y una función de estimación ponderada esto es, que dicha función de ponderación se calculase como ponderación de otras, según se explicó en el apartado de IA:

estimación(estado) = w1 * f1(estado) + ... + wn*fn(estado)

En este caso, la programación evolutiva se utilizaría para calcular los pesos "wi" de ponderación. Para ello, cada individuo de la población de PE sería un vector de pesos. La adaptación (o fitness) de los individuos se calcularía por medio de torneos. Es similar al entrenamiento de redes neuronales, en la medida en que diferentes conjuntos de pesos que influyen en la función de estimación compiten entre sí.

Sin embargo, observamos que las funciones f1,..., fn son funciones que parten de conocimiento humano. Luego dicha "búsqueda de la función de estimación" está mucho más "guiada" que en el caso de las redes neuronales (en nuestro proyecto apenas varios segundos bastarán para reproducir una función de estimación aceptable). El incluir conocimiento heurístico humano supone una ventaja y un inconveniente en sí. La desventaja es que es necesario disponer de dicho conocimiento (en el ajedrez por ejemplo cantidad de reglas tácticas, como es positivo ocupar posiciones centrales, y negativo tener dos peones en la misma columna, etc.). La ventaja es que el proceso de aprendizaje se acelera enormemente.

Un ejemplo de esta tercera alternativa de uso de la programación evolutiva es *el trabajo de k. Chisholm y P. Bradbeer en la Universidad de Napier en Edinburgo, Escocia* [REF 6]. En este trabajo se muestra la técnica anteriormente descrita para desarrollar un juego de damas.

2.2.3 Alternativas Consideradas en el aprendizaje contra humano:

Una de las características más interesantes y solicitadas en un juego que incluya aprendizaje es que lo aprendido quede almacenado de una vez para otra. Pero es más, es bastante solicitado (entre otras personas por nuestra directora de proyecto Lourdes Araujo, profesora en la UCM, Madrid) que la aplicación pueda aprender de manera individualizada, de jugar contra un jugador humano en concreto. De esta, manera se pretende que la aplicación desarrolle una "táctica" ganadora contra la forma de jugar contra ese individuo. Esta cualidad, se puede abordar, entre otras, con las siguientes alternativas:

2.2.3.1 Razonamiento Basado en Casos

Usando el paradigma de razonamiento basado en casos, se podría hacer lo siguiente: Inicialmente el sistema desarrollaría una táctica con cualquier tipo de algoritmo. En nuestro caso, con Programación Evolutiva. A posteriori, se jugarían diferentes partidas contra el humano que se quiere aprender a ganar. En cada partida, se almacenarían las

parejas (estado, operador) más significativas tanto del jugador humano como del sistema. Además se completaría a posteriori el resultado final al que se llego en la partida. Resultando cada caso aprendido o almacenado en la base de caso como:

* Estado o situación de partida (problema)

* Operador o jugada (solución)

* Resultado de la partida (comprobación)

En partidas sucesivas, ante una situación o estado de juego contra dicho humano, se buscaría en la base de casos el caso que tuviera el estado más similar al estado actual y hubiera llevado a una victoria al jugador que tenía el turno en ese instante. El operador o jugada del caso se adaptará para el nuevo estado o situación y será el que se aplique.

Lo más difícil de esto es determinar cuando dos estados se parecen (cálculo de la función de similitud). Algunas opciones por ejemplo serían:

* Dos estados son similares simétricamente iguales (esta es muy restrictiva)

* Dos estados son similares si el número de piezas son similares

* etc.

También tendrá cierta dificultad adaptar el operador. Por ejemplo, si los estados originales son movimientos, dichos movimientos deberán ser transformados.

Observamos que, de esta forma, el sistema aprende, en parte, de las jugadas de su oponente. Esta forma de aprendizaje es asombrosa, y suele tener resultados bastantes buenos si se escogen bien la función de similitud y la de adaptación del operador (no debe confundir el lector esta función de adaptación con la usada en PE)

Sin embargo, esta forma de aprendizaje es muy diferente al resto de aprendizaje utilizado hasta el momento por lo que fue descartada.

2.2.3.2 PE calculando la adaptación mediante partidas contra humano

Una segunda alternativa sería desarrollar un algoritmo evolutivo para el cálculo de pesos en la función de estimación al igual que el método que hemos usado nosotros,

explicado en el apartado de diseño. Pero la diferencia radicaría en que la adaptación de cada individuo (o conjunto de pesos) se calcularía mediante un número fijo de partidas contra el humano. La adaptación de dicho individuo sería el número de partidas ganadas contra el humano. El algoritmo evolutivo continuaría de manera habitual generación contra generación.

En el caso hipotético de implementar esta desproporcionada opción, se permitiría al usuario descansar al final de cualquier partida. En cuyo caso habría que almacenar en un fichero la situación actual del algoritmo evolutivo (tales como número de generación, número de individuo que esta calculando la adaptación, y la población de indiviudos actual). Dicha información se recuperaría desde el archivo cuando el usuario quisiera seguir continuando el algoritmo evolutivo, partida tras partida.

El resultado final sería un conjunto de pesos que mejor juega y más veces gana sin duda, contra ese humano en concreto.

Sin embargo, esta opción tiene grandes desventajas. En primer lugar, se necesitaría que el humano realizase muchísimas partidas para que el sistema aprenda (por ejemplo, imaginemos que el número fijado de partidas es diez, el de individuos en la población es diez y el numero de generaciones es diez. El jugador humano debería realizar 10 x 10 x 10 = 1000 partidas, intratable).

Esta opción tiene otra desventaja incluso mayor. Nuestro sistema necesitaría 1000 partidas para aprender. Pero con lo que no se cuenta, es que con que el humano también aprende y a una velocidad mucho mayor. Por lo tanto, cuando el sistema haya aprendido, el humano habrá aprendido mucho más. En mi opinión, no conviene desafiar a la inteligencia humana, o al menos de esta manera tan "tonta" o desequilibrada.

Una buena idea sería que dichas partidas las jugaran humanos distintos, encontrados en la red. Quizás se podría poner un nombre o titulo atractivo para aumentar el número de usuarios que entren en "nuestra página" y contribuyesen al entrenamiento de nuestro sistema.

2.2.3.3 Algoritmo de Escalada en combinación con la PE ya vista

Como introducción cabe explicar que el algoritmo de escalada no hace vuelta atrás o "backtracking". Esto es parte de una solución, y sólo cambia su solución por una solución si la nueva es mejor que la anterior.

De la misma forma nuestro sistema empezaría con un conjunto de pesos calculados por la PE, y se observaría resultado. Contra un cierto jugador humano, cada nueva partida o

cada cierto número de jugadas se evaluarían los resultados. En caso de ser cada partida se miraría el resultado final de la partida y en caso de ser cada cierto número de jugadas se observaría si ha sacado desventaja o desventaja y se almacenaría.

En cada recálculo de los pesos por programación evolutiva se introduciría en la población inicial, un individuo inicial con el conjunto de pesos que mejor resultado haya tenido hasta el momento.

Cada vez que se evaluasen los resultados se observaría si dichos resultados son mejores o peores que los resultados obtenidos hasta el momento. En caso afirmativo, se guardarán como mejores resultados el conjunto de pesos nuevo. En caso negativo continuaremos con el conjunto de pesos antiguos.

El conjunto de pesos "mejor" así como dicho "mejor resultado" que obtuvo, se guardarán en un fichero y serán recordados cada vez.

2.2.3.4 Alternativas Elegida en el aprendizaje contra humano:

En conclusión hemos estimado que la alternativa más adecuada en nuestro proyecto para el aprendizaje contra el ser humano es algoritmo de escalada.

El razonamiento basado en casos es de complejidad razonable y están fuera de los objetivos de este proyecto, el estudio de PE en juegos de estrategia. El razonamiento basado en casos tiene entidad como para formar otro proyecto entero.

El uso del humano como herramienta para calcular la adaptación (fitness) en la PE, se haría muy pesado para el jugador humano, y lo consideramos innecesario.

Por el contrario, el uso del algoritmo de escalada es sencillo y cómodo; además se adapta a las necesidades de nuestro proyecto sin un gran coste extra, como el que tendrían las otras dos opciones. También se esperan unos resultados satisfactorios, aunque eso son ya conclusiones que deberemos sacar en el correspondiente apartado de validación.

3. Especificación del Proyecto

3.1 Especificación del Sistema

3.1.1 Requisitos Funcionales

El sistema cumplirá los objetivos expuestos en el apartado 1.2, en lo referente a implementación, aprendizaje e interfaz gráfica.

3.1.2 Requisitos No Funcionales

El sistema, en la interacción con el usuario, deberá tener unos tiempos de respuesta aceptables, incluso en los pasos en los que intervenga un proceso de aprendizaje, al iniciar una partida, por ejemplo, no más de varios segundos. Asimismo, deberían mantenerse por la calidad del juego unos tiempos de respuesta durante la acción casi inmediatos. No debería asimismo requerir un supercomputador para poder funcionar fluidamente, ya que la idea es que sea un juego que cualquier o casi cualquier persona pueda hacer correr en la máquina de su casa.

3.2 Especificación del Juego

El juego de estrategia usado en nuestro proyecto es un juego nuevo inventado por nosotros mismos. Esto da un rasgo de originalidad al proyecto. El juego constará de dos modos: el modo general, y el modo batalla, que será un modo que complemente al primero.

3.2.1 Modo General

Habrá dos jugadores (generalmente uno será dirigido por el humano y otro por la máquina) .Cada jugador dirige un ejército. Dicho ejército tendrá:

- Ciudades (pueden tener en su interior un líder del mismo jugador)

- Áreas de Recursos Apropiados

- Líderes (cada uno de los cuales con un número de soldados dados)

- Número de recursos (representan alimento, dinero y otras cosas)

Asimismo, habrá recursos y ciudades que no sean de nadie y que se puedan conquistar por cualquiera de los líderes.

El juego será por turnos, que se irán alternando entre los dos jugadores. El jugador que tenga el turno podrá realizar alguna de las siguientes operaciones:

- Mover un líder (en cualquiera de las direcciones: arriba, abajo, izquierda y derecha):

- Si un líder llega un área vacía se situará en ella.

- Si llega a un recurso o ciudad de su propiedad sólo podrá entrar si no hay ningún líder en ella.

- Si el líder entra en una ciudad del oponente, el líder se situará en dicha casilla, y la ciudad quedará conquistada.

- Si el líder entra en un área de recursos del oponente, dicha área de recursos pasará a pertenecer al jugador que dirige dicho líder.

En cualquier caso en el que un líder llegue a la casilla donde este otro líder (el otro líder podrá estar en una casilla sin nada más o podrá estar en dentro de una ciudad o recurso), se libra una batalla entre los dos lideres. El líder ganador se situará en la casilla de la batalla, sometiéndose a las reglas de movimiento anteriores (véase el subapartado de “modo batalla”).

- Comprar un líder en una ciudad vacía de su propiedad. En cuyo caso el nuevo líder tendrá cero soldados y se encontrará en la ciudad correspondiente. Para ello es necesario tener y gastar cierto número de recursos.

- Comprar un soldado en una casilla que contenga un líder de su propiedad. En dicho caso el líder incrementará su número de soldados. Para ello, es necesario tener y gastar cierto número de recursos.

Después de cada turno, cada ejército verá incrementado su número de recursos de forma proporcional a las áreas de recursos apropiadas en dicho instante.

La disposición inicial del juego, tanto el tamaño del campo de batalla, como disposición inicial de los ejércitos, será variada.

3.2.2 Modo Batalla

Cada vez que se libre una batalla entre dos líderes, se abrirá una nueva subpartida. En dicha partida de batalla, sólo habrá un tipo de unidad, los soldados.

Dichos soldados podrán moverse en las cuatro direcciones. Si un soldado se mueve sobre la posición del soldado enemigo será un ataque con éxito, en el cual el soldado enemigo desaparece y el soldado atacante pasa a ocupar su posición.

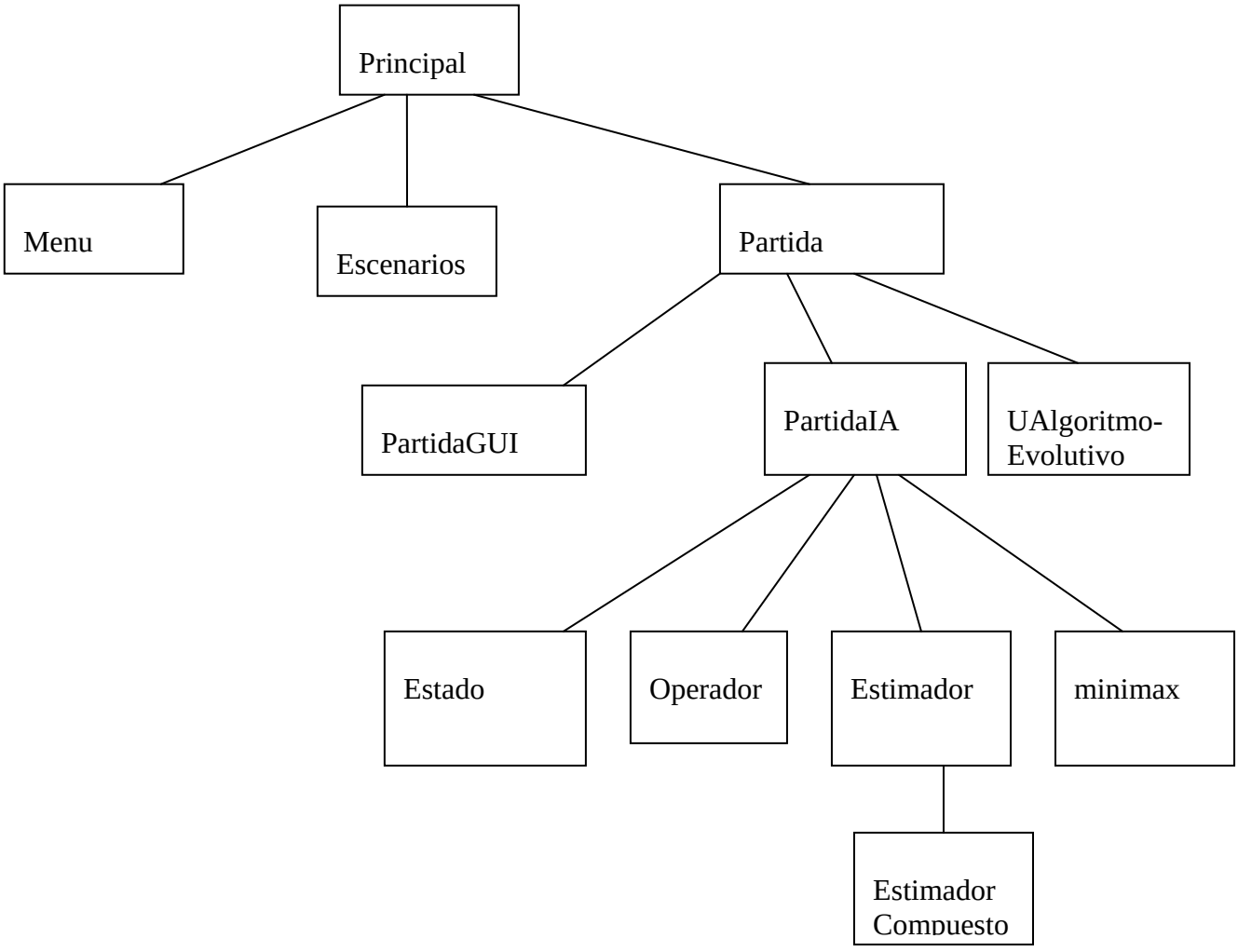
El numero de soldados de cada jugador será el numero de soldados de cada líder enfrentado en la batalla más uno (suponemos que el propio líder juega como un soldado más en la batalla). La batalla se acabará cuando finalice el tiempo o alguno de los dos jugadores se quede sin soldados.

El líder ganador será aquel que al final tenga más soldados, el número de soldados de ese líder en el juego general, tendrá en cuenta las pérdidas en la batalla. En concreto, será la diferencia de soldados de un jugador con otro (sin tener en cuenta el soldado que representa al líder).

4. Diseño e Implementación

4.1 Diseño General

Nuestro proyecto consta de los siguientes módulos o clases:



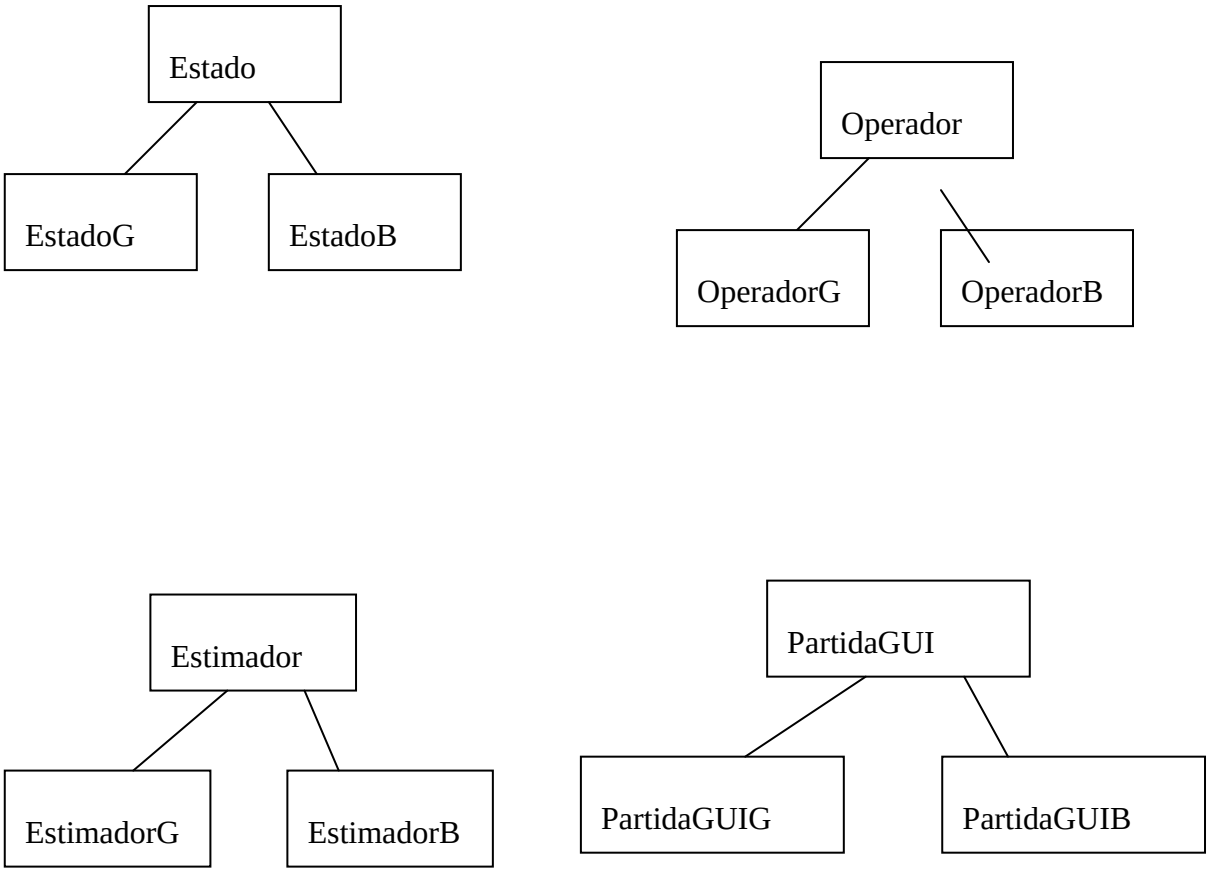
Los anteriores módulos son usados indistintamente por los dos modos de juego, el modo general (con recursos, ciudades y líderes) y el modo batalla (sólo con soldados). El conseguir que tengan tantos módulos o clases compartidos entre dos modos tan diferentes de juego lo consideramos una cualidad positiva, en especial los referentes a "partida" , "minimax" y "UAlgoritmoEvolutivo", esto se consigue por medio de interfaces "clases abstractas en C++" como "Estado", "Estimador" y "Operador"

Sin embargo hay métodos que son específicos de cada módulo, y en los cuales se distinguen con el siguiente convenio:

* Terminan con "G" aquellos específicos del modo General

* Terminan con "B" los del modo Batalla

Obsérvese el siguiente diagrama:



Hemos intentado desde un principio crear un diseño para nuestro programa lo más genérico posible, de modo que tanto el mantenimiento como posibles ampliaciones fuesen posibles de un modo rápido y sencillo.

Durante la primera fase de creación del videojuego, todo el aspecto relacionado con el diseño tuvo una importancia capital, ya que en buena medida el diseño (aunque es posible cambiarlo) determina el proyecto en gran parte.

Posteriormente, se llegó a un punto donde la implementación fue ganando terreno, lo cual era una evolución lógica en el programa.

Sin embargo, y pese a echarle dedicación, como en casi todos los proyectos grandes de programación, el diseño fue cambiando sucesivamente pasando por diversas versiones por varios motivos: a veces porque se encontraban formas mas elegantes o sencillas de hacer las cosas, a veces porque llegábamos a un punto donde nuestro diseño no era consistente, o en parte no cuadraba con las ideas de POO (programación orientada a objetos), como por ejemplo, la independencia, y la encapsulación...

Al estar trabajando en C++, podíamos optar por usar clases, módulos con funciones, combinados... Al final , elegimos este último aspecto, y según creíamos conveniente usábamos un modo de trabajo u otro.

- El primer módulo a citar es **Principal**, que contiene las funciones que inicializan ratón, teclado, sonido, el modo gráfico...

  Además, es quien lleva el control sobre Allegro (librería gráfica utilizada para la creación del juego), al ser quien le dice cuándo debe inicializarse y finalizarse.

  Otro detalle es que en él se ubica la función ‘main’ de nuestra aplicación. En ella se crea el menú principal, se visualiza, se recoge la opción que se haya marcado, y se pone en marcha la partida si es necesario. Aunque no lleva carga lógica en el programa de aquí en adelante, hasta que la partida no finalice, es una clase muy importante, ya que también la utilizamos para poder enlazar varias partidas sin tener que salir. Es importante decir que para evitar la inclusión múltiple, en los módulos se han incluido las siguientes líneas:

  -- al comienzo del módulo --

  #ifndef (Nombre_Módulo)H // Evitar inclusiones múltiples

  #define (Nombre_Módulo)H

--al final del módulo --

#endif

Asimismo, es un sitio muy útil, al ser tan general, para poder realizar pruebas en los largos periodos de depuración por los que ha tenido inevitablemente que pasar la aplicación.

- La clase que se encarga de mostrar el menú principal , e íntimamente ligada a la anterior es la clase **Menú**, que lleva buena parte de interactividad con el usuario, y devuelve al programa lo que el usuario va eligiendo para que este lo pueda procesar. En lugar de crear y destruir cada visualización del menú, creamos una sola instancia de esta clase, para usarla todas las veces que sea menester. El método principal de esta clase es el método 'Mostrar' que se encarga de mostrar la pantalla principal de selección del menú. A partir de ahí, tenemos métodos para poder ver las pantallas especificas según la opción. Como podemos ver, nuestro diseño también conlleva una externa organización de directorios en Música, Imágenes... dado que si no fuera así, encontrar posteriormente un archivo entre el total podía ser un caos, y no quedaría un sistema de archivos para nada legible.

Por otro lado, y aunque parezcan módulos algo desconectados del resto, tenemos dos bastante curiosos: Sonido y Archivos.

- **Sonido** se encarga de llevar, a excepción de la inicialización y finalización de drivers de sonido, lo cual es misión de Principal, de todo lo relacionado con el sonido. En principio, tiene funciones para poder reproducir un archivo WAV que tengamos en nuestro disco duro (la edición de estos ficheros debería hacerse externamente, desde nuestro programa lo que hacemos es simplemente reproducirlos), parar la música que esté actualmente reproduciéndose y controlar los parámetros de reproducción (frecuencia de reproducción, volumen, panoramización...). En principio, no es muy recomendable que alguien manipule sin saberlo estos parámetros, ya que puede tener efectos curiosos sobre las melodías. No obstante, el volumen es perfectamente modificable desde el exterior (altavoces...). Nosotros nos encargaremos de rellenar estos parámetros. En este apartado nos valemos del tipo SAMPLE incluido en Allegro. En este caso, podemos considerar que SAMPLE es una estructura de alto nivel, ya que es posible modificar los sonidos a más bajo nivel. Para ello se emplean las voces. Se debe reservar y liberar las voces explícitamente, ya que éstas no se

liberan solas al acabar la reproducción, pero esto le permite especificar precisamente lo que quiere hacer con el sonido. Incluso puede modificar algunos campos de la estructura sample.

- El otro módulo del que queremos hablar en este momento es **Archivos**. Aquí se encuentra la información relacionada con los archivos en nuestro caso, y también con el aprendizaje. En nuestro caso, tenemos un aprendizaje personalizado consistente en almacenar en un archivo nombrado con el nombre del jugador que está actualmente jugando, con una extensión conocida (txt) para que pueda ser observado y estudiado tanto por nosotros como por otras personas. Dentro almacenamos el individuo que le venció anteriormente y del cual extraeremos información sobre como jugar. Para el caso del diseño, lo importante es que necesitamos funciones para poder llevar a cabo estas acciones:

  'CargarInd' y 'GuardarInd' cargan y guardan respectivamente, usando como parámetros el individuo donde queremos cargar o guardar la información, y la ruta del archivo. Uno de los grandes problemas con el que nos hemos encontrado aquí ha sido transformar objetos de tipo cadena de caracteres a tipo real. Para ello, es por lo que se han creado funciones del tipo strToFloat (cadena a real) o charToDig (carácter a digito). Estas funciones están predefinidas para C++, pero debido a problemas con el compilador tuvimos que crearlas nosotros mismos.

- Otro de los grandes pilares de nuestro juego es la clase **Partida**. Se puede decir que esta clase es posiblemente la que lleva una mayor carga lógica de nuestro programa. Por una parte, lleva la parte inteligente tanto de lo que podríamos considerar juego general como la del apartado de las batallas. Por otro lado, lleva la información de todos los jugadores que tenemos en la partida, una indicación o flag de finalización para saber cuando se ha acabado, el mejor individuo obtenido por el algoritmo evolutivo (del cual obtenemos los pesos para jugar), y además un puntero a la parte gráfica, donde realizaremos toda la interacción de esta parte con el usuario. Como métodos fundamentales podemos reseñar todos: 'Jugar', 'JugarHumano', 'JugarMaquina', 'JugarBatalla'... En esta clase, problemas constantes han sido que atributos necesitaba, que usos y formato les íbamos a dar (dudas entre usar listas, vectores...), cuantos y que parámetros iban a tener los constructores...

  También estuvo en los primeros momentos de creación la duda de cuantos jugadores simultáneos íbamos a permitir en una misma partida, pero al final decidimos que fuesen dos. Desde Partida nos encargamos de inicializar tanto la parte gráfica como inteligente del juego, y posteriormente de controlarlas.

- La parte inteligente de la partida es un objeto de la clase **PartidaIA.** Esta clase esta preparada para, teniendo un atributo de tipo Estado, extraer toda la

información posible del estado actual de la partida. También tiene información de la profundidad con la que estamos realizando minimax, e información sobre los jugadores y los estimadores. Su método estrella es jugar, que tal como reza su comentario:

/*

METODO: jugar()

DESCRIPCION: Se juega una partida completa y se devuelve el resultado.

Esta operación sólo se podrá invocar en partidas de maquina contra maquina.

*/

De aquí deducimos que esta clase es usada tanto para el entrenamiento de nuestro programa a través de la PE en los torneos, como posteriormente para realizar jugadas durante el juego por la máquina. Para ello hace uso de los métodos 'JugarMaquina' y 'JugarHumano'. Sus cambios los va aplicando sobre el estado, de modo que a nivel lógico esta cambiando la partida de situación. En esta clase usamos vectores de C++, extremadamente útiles por sus posibilidades de acceso. En función de nuestras necesidades, en otros puntos del código lo que usamos es la clase **Lista**, clase implementada por nosotros, que implementa una lista enlazada (sus atributos y métodos son los clásicos para este TAD).Usamos plantillas para generar listas genéricas.

- Módulos que nos han resultado también de utilidad, aunque no participan de modo fundamental en la aplicación, han sido por ejemplo **Mensajes,** muy usado en tareas de comunicación con el usuario, y sobre todo depuración, ya que hemos tenido algunos problemas con el debugger del propio Dev-Cpp, o **Matemático**, donde implementamos pequeñas funciones exclusivamente matemáticas para usar en los algoritmos.

- Otra clase que nos servirá enormemente a posteriori es la clase **Posición**. En ella, aunque de un contenido muy trivial, implementamos una posición en 2D, pero con métodos para acceder y modificar sus parámetros.

- Las clases **Operador** (implementada a través de **OperadorG** y **OperadorB**), nos permiten crear diferentes operadores para la batalla y para el juego general. Evidentemente, debíamos tener separados estos detalles en el diseño, ya que el hecho de comprar un soldado por poner un ejemplo no tiene mucho sentido una vez ha comenzado una batalla al menos en nuestra aplicación.

- Algo parecido ocurre con los **estimadores**. En nuestro diseño hemos procurado que el hecho de añadir un estimador sea lo más sencillo posible, ya que pensamos que puede ser de los sitios donde más cambios tengamos que realizar en un breve periodo de tiempo.

- Por último, comentar la estructura que le otorgamos a cada una de las posiciones del tablero. Cada una de ellas es un **ObjetoMapa**, pero más en concreto puede ser una especificación de esto último, como puede ser **Líder**, **Ciudad** o **Recurso**.

- Como bien se dijo en la especificación, se trata de un juego de ejércitos, de ahí que existan las clases **Unidad** y **GrupoUnidad** (la cual es un conjunto numerado de la anterior clase), para que, por ejemplo, en nuestro programa los líderes puedan llevar batallones.

4.2 Diseño de la Inteligencia Artificial

Hemos usado el algoritmo "minimax" clásico. Obsérvese el apartado Inteligencia Artificial en juegos. Además hemos usado clases que encapsulen los comportamientos de una partida.

La clase Estado

La clase Estado es abstracta y representa un estado de un juego (mírese el apartado sobre IA). Esta clase contiene atributos comunes a los dos juegos

* turno: Enumerado cuál de los jugadores tiene el turno en instante dado.

* tiempo: Tiempo límite restante

Los métodos abstractos principales son:

```
/* Devuelve los operadores (jugadas) aplicadas a un Estado */
Lista<Operador>* operadores();
/* Aplica un operador al estado */
virtual void aplicar(Operador*)=0;
/* Indica si se encuentra en un final de partida */
virtual bool final()=0;
/* Evalua un estado que se encuentre en final de partida. Esto es quien ha
ganado y por cuánto. Si el valor es positivo ha ganado MAX, si es negativo MIN,
y cero si es empate */
virtual float evaluar()=0;
/* Devuelve una copia del estado */
virtual Estado* clone()=0;
/* Dado un operador, indica si es posible aplicarlo al estado actual */
virtual bool operadorAplicable(Operador* op)=0;
```

Implementaciones de la clase Estado: EstadoG e EstadoB

Quizás las mayores alternativas planteadas se encuentran en torno a las implementaciones de esta clase abstracta. Las alternativas principales eran:

1) Utilizar una lista para almacenar las fichas y sus posiciones

2) Utilizar una matriz y en cada posición almacenar la supuesta ficha en caso de haberla.

Pues bien, para tomar esta decisión se tiene que tener en cuenta la eficiencia y las operaciones aplicadas sobre Estado, así como sus respectivos costes y frecuencias.

Dichas operaciones son por ejemplo:

*) Aplicar un operador sobre un soldado. Para mover un líder o un soldado hay que tener en cuenta a la posición origen y destino. El coste depende de la opción elegida

1) Hay que recorrer toda la lista para ver si se encuentra dicha posición

(lineal frente al número de fichas)

2) Basta con mirar en la posición de la matriz dada

*) Estimar un estado. Algunos de los estimadores complejos, necesitan recorrer todos los líderes y sus posiciones. Obtener todas las posiciones tiene un coste diferente según la opción de implementación elegida:

1) Basta con devolver la lista

2) Habrá que recorrer toda la matriz (coste lineal con respectos al número de casillas)

Cabe destacar que dependiendo del tamaño y número de fichas usadas en media beneficiará más una u otra representación:

1) Para tableros grandes (con muchas casillas) con generalmente pocas "fichas", la implementación de la lista es más conveniente. Así tiene información de forma compacta.

2) Para tableros pequeños y con muchas fichas, usar una matriz es más conveniente. No ganamos apenas en memoria, sin embargo tenemos la información más organizada, cada posición está en un sitio fijo de la memoria

Después de estas y otras consideraciones llegamos a la conclusión que lo mejor sería:

* Para EstadoB: Usar lista, ya que se iban a desarrollar estimadores complejos que necesitaban acceder a todas las posiciones.

* Para EstadoG: Usamos las matrices, dado que el gran número de elementos y sus diferentes tipo (lideres, ciudades y áreas de Recurso) requerían por claridad y robustez estar dispuestos y organizados en una matriz.

4.3 Diseño de la Programación Evolutiva

En nuestro proyecto, hemos usado la tercera alternativa, de las tres que se propone en el apartado de Introducción y Estudio Preliminar. Esta es, usar la programación evolutiva para obtener los mejores pesos de ponderación para una función de estimación compuesta por otras.

*estimación(estado) = w1 * f1(estado) + ... + wn*fn(estado)*

El algoritmo de búsqueda de estados será el algoritmo "minimax" clásico. El cual usará la función de estimación con el vector de pesos calculados por la programación evolutiva.

Es importante remarcar que la programación evolutiva se aplica al inicio de cada partida. Esto es, se obtienen los pesos de ponderación del estimador compuesto. Durante el desarrollo de la partida se utilizará el estimador compuesto con los pesos calculados inicialmente. Por tanto, durante la partida no se ejecutará ningún algoritmo evolutivo. El algoritmo utilizado por excelencia es “minimax”. Esto es similar en ambos modos de juego (“general” y “batalla”), donde se aplicará la programación evolutiva exclusivamente al inicio de cada partida o batalla.

Los motivos para usar esta alternativa han sido su sencillez y rápido aprendizaje frente a otros métodos como por ejemplo el que usa redes neuronales (necesitaba seis meses de entrenamiento). El motivo de no escoger la programación genética, es que requiere muchos recursos, tiempo y grandes poblaciones. Dado nuestro juego y sus reglas han sido inventadas por nosotros, no disponíamos de esa función de estimación "optima". Hemos preferido investigar y desarrollar el uso de la PE para obtener los pesos de una ponderación, pudiendo así obtener nuestra propia función de estimación "óptima" en el transcurso de nuestro proyecto.

A continuación, mostramos en mayor detalle, las características de nuestro algoritmo evolutivo.

Representación de los Individuos

Cada individuo de la población de la población (pool) contendrá un vector de pesos. Cada peso sería un número real comprendido entre 0 y 1. Elegimos esta representación

frente a representaciones binarias, por mejorar el cruce, al poder usar así funciones matemáticas como medias.

w1 = ...	w2 = ...	...	wi = ...	...	wn = ...

Algoritmo Evolutivo General

El algoritmo tendría el esquema clásico de un algoritmo evolutivo. Tendría el siguiente aspecto.

```
// Algoritmo Genético
 pob=poblacion_inicial( ... );
 // Calculo de la calidad de los individuos
 adaptación(pob, ... );
// Calculo de los valores para ser seleccionados
 evaluación( .... );
 for(int generacion=0;generacion<num_max_gen;generacion++){
        seleccion(pob,pos_mejor);
        //guardo a el mejor individuo de la población (elitismo)
        indiv_mejor=pob[p.tam_pob-1];
        pob.pop_back();
        reproduccion(pob,p.prob_cruce);
        if((generacion%p.periodo_mut)==0)
              mutacion(pob,p.prob_mut);
        //coloco en la nueva población al que antes era el mejor individuo
```

```
        pob.push_back(indiv_mejor);

        adaptacion(pob,.. );

        evaluacion(pob, ...);

    }
```

NOTA: En el anterior esquema de algoritmo, y siguientes usamos como sintaxis la de C++ y como convenio usamos los nombres de funciones utilizados en la asignatura de Programación Evolutiva de la UCM (Madrid).

Adaptación (fitness)

Lo más característico de la alternativa usada en nuestro proyecto es la forma de calcular la adaptación (o fitness) o calidad de nuestros individuos. Para calcular dicha calidad, se hará un torneo entre los conjuntos de pesos correspondientes, en la que todos jugarán contra todos ida y vuelta. Lo de jugar dos partidas, es para eliminar las posibles ventajas de empezar primero o posibles características de estar situado en una posición u otra del tablero. Dicha adaptación se calcula con el siguiente algoritmo:

```
// Puesta a cero de la aptitud. Inicialmente cero partidas ganadas por
// cada individuo
 for(int i = 0; i< size; i++)
     pob[i].aptitud =0;


 // Torneo: La aptitud de cada individuo será el numero de partidas ganadas
 for(int i=0;i<pob_size;i++)
 {
     for(int j=0; j<pob_size;j++)
     {
```

```
        if ( i!=j)
        {
          // Copiamos los pesos en los estimadores
            estim1->setPesos(pob[i].pesos);
            estim2->setPesos(pob[j].pesos);

         // Jugamos la partida
            partidaAux = new PartidaIA(estado,estim1,estim2);
            while(!partidaAux->final())
            {
               op = partidaAux->jugarMaquina();
               delete op;
            }
           // Obtenemos el resultado
            result = partidaAux->getResultado();

    // Dependiendo del resultado incrementaremos la aptitud de uno u otro individuo.
En caso de
   //   empate no incrementaremos la aptitud de ningún individuo
            if (result > 0)
               pob[i].aptitud++;
            if (result < 0)
               pob[j].aptitud++;
        }
```

```
    }
}
```

Cruce (crossover)

Para el cruce usamos una reproducción sexual (en la que intervienen dos individuos) después de mucho probar usamos medias ponderadas para entre los vectores de pesos. Inicialmente usamos medias aritméticas. Esto es:

Sean dos vectores de pesos:

wa1, wa2, ... , waN

wb1, wb2, ... , wbN

de dos individuos a y b, los pesos nuevos serian, para todo i en (1..N)

wai = 1/2*wai ' +1/2* wbi '

wbi = 1/2*wai ' +1/2* wbi '

Sin embargo, experimentalmente observamos que dicho cálculo no se comportaba del todo bien. Se comportaba mejor el siguiente cálculo ponderado:

wai = 2/3*wai ' +1/3* wbi '

wbi = 1/2*wai ' +2/3* wbi '

A posteriori, vimos que esto era muy coherente. Esto es, en el primero caso los dos individuos cruzados pasaban a tener exactamente los mismos pesos y por tanto pasaban a ser el "mismo" individuo duplicado. Esto empeoraba ligeramente la riqueza y variedad de los individuos en la población, frente a la segunda opción en la cual el resultado del cruce de los individuos eran dos individuos distintos.

Elitismo

Otro rasgo característico es el uso de elitismo en cada generación, esto es conservar siempre de una generación el individuo con mayor calidad o adaptación (fitness).

Población Inicial y Mutación

La población inicial se escogerá dando valores aleatorios en el rango permitido [0..1] a cada peso.

En el caso de la mutación, cada peso de cada individuo, con una probabilidad de mutación (muy baja), se volverá a escoger aleatoriamente en el rango permitido.

Uso de juegos reducidos para el entrenamiento

Aunque el juego contra el humano podría tener tamaño y tiempos límites grandes; Por motivos de eficiencia, al igual que en *el trabajo de Chisholm y Bradbeer de la Universidad de Napier* [REF 6] anteriormente mencionado, en los torneos desarrollados en el algoritmo evolutivo usamos juegos de tamaño y tiempo límites reducidos. El motivo de esto es el gran número de partidas que se deben realizar en dicho algoritmo evolutivo. Por tanto, para mantener una eficiencia aceptable (varios segundos), fue necesario usar estos juegos "reducidos".

Esto tiene la ventaja de la mejora notable de la eficiencia. Sin embargo, en juegos grandes el comportamiento es ligeramente peor en algunos casos, ya que la función de estimación se ha "entrenado y aprendido" en tableros pequeños.

Por ejemplo, entrenándose el modo batalla en tableros cuatro por cuatro, la función de estimación da un peso enorme a las cuatro casillas centrales:

	+	+	
	+	+	

Observamos que dichas posiciones centrales permitían acorralar a cualquier contrincante sin escapatoria alguna. Por ejemplo, el soldado "-" se encuentra sin escapatoria ante los soldados "+"

			-
	+	+	
	+	+	

Sin embargo, la partida real se jugaba en seis por seis, la estimación seguía asignando un gran peso a estas casillas centrales, sin embargo en este tipo de batallas las cuatro posiciones centrales no permiten acorralar al soldado "-". Obsérvese como el soldado "-" puede escaparse:

					-
		+	+		
		+	+		

Al final, decidimos que era mejor entrenar el juego en un tamaño más parecido al tamaño de la partida real. Entendemos por tamaño de la partida real al tamaño de la partida contra la que jugará el usuario. Con esto, evitamos problemas parecidos al que acabamos de mencionar.

4.4 Diseño parte gráfica

4.4.1 Eligiendo librería

Lo primero que hay que decir es que para esta parte del proyecto hemos utilizado una librería gráfica que se utiliza con C/C++, y que se llama “Allegro”. Esta librería está especialmente diseñada para el desarrollo de videojuegos en dos dimensiones, y presenta una cierta comodidad a la hora de elaborar el entorno gráfico de nuestra aplicación. Aunque, como veremos más adelante, también presenta sus dificultades y problemas.

Cabe decir que hemos empleado esta librería y no otra, ya que es la que más garantías de estabilidad y funcionamiento presentaba, y de la que existía más información, tutoriales, y publicaciones en los foros que nos ha ayudado bastante. De hecho, inicialmente, se ha elaborado un CD de formación recopilando información de cursos de C++, de la herramienta Allegro,... que nos ha sido de mucha utilidad y adjuntamos en formato digital como apéndice a la memoria. Además, en la documentación de esta librería existe información de calidad acerca de cómo se maneja cada función, los parámetros que utilizan y varios ejemplos que nos fueron muy útiles en un principio.

Con todo esto, decidimos usar Allegro. Y nada más hacerlo nos dimos cuenta del primer problema que se nos planteaba: allegro no es compatible con C++ Builder. Para solucionarlo encontramos una herramienta, Dev-C++, que sí que era compatible, y que resultaba ser un entorno de edición bastante amigable. Con lo cual nos pusimos manos a la obra y comenzó nuestra tarea.

4.4.2 Los primeros pasos

Para empezar a familiarizarnos con Allegro lo que hicimos fue cargar algunos de los ejemplos que traía en el Dev-C++ y probarlos a ver que pasaba. Aparte de esto, estudiamos el código de cada uno y analizamos lo que podía sernos útil para nuestro juego.

A partir de aquí, hicimos nuestras primeras pruebas, que consistían en pequeños módulos independientes unos de otros y que realizaban tareas sencillas. Primero mostrábamos alguna imagen por pantalla, luego introdujimos el ratón y el teclado para que reaccionase a nuestras órdenes, y más tarde intentamos juntar todo.

Una de las primeras pruebas grandes que realizamos consistía en hacer un menú, cuyo aspecto era prácticamente igual que el de la versión final, en el que teníamos varios botones que se veían de distinto color según el ratón estuviese encima o no. También, logramos hacer que al pinchar en algún botón pasásemos a otra pantalla, como podía ser la pantalla de créditos. En definitiva, varias pantallas para pasar de una a otra y ser capaces de controlar el ratón y el teclado.

Pantalla inicial:

Pantalla de créditos:

Gracias a esto conseguimos aprender a utilizar algunas funciones básicas de Allegro, que supondrían la base para implementar el resto de la parte gráfica. No entraremos en un nivel de detalle demasiado elevado, pero si comentaremos algunas funciones básicas.

Estas se encargan de la inicialización de Allegro:

- 'allegro_init(…)'. Inicializa la librería de Allegro.
- 'install_mouse(…)'. Para reconocer el ratón.
- 'install_timer(…)'. Para iniciar los temporizadores.
- 'install_sound(…)'. Para instalar el sonido.
- 'set_color_depth(…)'. Fija el color de fondo.
- 'set_gfx_mode(…)'. Fija la resolución de la pantalla.

Otras se encargan de la interacción con el usuario:

- mouse_b &1. Variable que indica si esta pulsado el botón izquierdo del ratón.
- mouse_b &2. Variable que indica si esta pulsado el botón derecho del ratón.
- mouse_x. Variable que indica en que posición del eje X se encuentra el ratón.
- mouse_y. Variable que indica en que posición del eje Y se encuentra el ratón.
- 'keypressed()'. Función que indica si hay alguna tecla pulsada.

Y otras de dibujar en la pantalla:

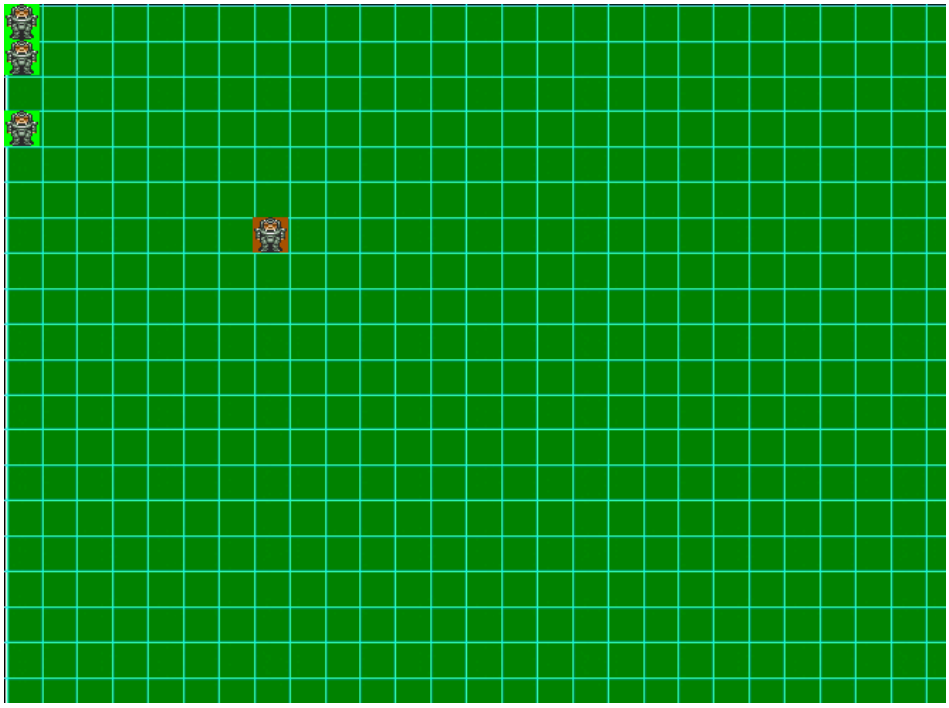
- 'blit(…)'. Dibuja un BITMAP en la posición indicada.
- 'stretch_blit(…)'. Dibuja un BITMAP en la posición indicada, y encajándolo en unas dimensiones dadas. (Esta función no la encontramos hasta muy tarde, con lo que tuvimos algunos problemas que ya relataremos más tarde)

- ‘textout_ex(…)’. Dibuja el texto indicado en la posición deseada.
- ‘load_tga(…)’. Para cargar imágenes .tga
- ‘destroy_bitmap(…)’. Para destruir BITMAPS.

Estas son algunas de las principales funciones de Allegro, y que hemos usado bastante a menudo. Como se puede observar, hemos trabajado con BITMAPS en formato TGA, que son todas nuestras imágenes (la gran mayoría las hemos creado nosotros mismos). Básicamente, el proceso para mostrar una de estas imágenes por pantalla consiste en cargarla, elegir dónde la queremos colocar y cuánto queremos que ocupe, y pintarla.

Las funciones descritas anteriormente pueden parecer algo rudimentarias, y en verdad lo son. Esto por una parte es ventajoso, ya que te permite trabajar a un nivel de abstracción bastante bajo y realizar prácticamente lo que uno desee sin ningún impedimento. Ahora bien, trabajar a ese nivel es realmente complicado, y realizar cualquier operación, por sencilla que pueda parecer, lleva muchísimo tiempo y quebraderos de cabeza.

Otra de las grandes pruebas que realizamos consistió en un escenario de juego básico. Principalmente era un mapa de color verde y dividido en casillas. Sobre él dibujamos varios soldados, y conseguíamos seleccionar alguno de ellos pinchándolo con el ratón, y moverlo a otra casilla del mapa pinchando también con el ratón.



Visto así no parece gran cosa, pero para realizar esto hay que tener en cuenta varios aspectos:

Los BITMAPS se dibujan con la función 'blit()', pero la posición se le pasa en píxeles. Esto supone un problema ya que nuestras casillas vienen dadas por dos números enteros correspondientes a la fila y la columna. Por lo tanto, ¿cómo saber en qué píxel empieza cada casilla? ¿cuántos píxeles ocupa cada casilla? Para solucionar esto tuvimos que crear una matriz del tamaño del mapa (filas por columnas), y en cada posición guardar el valor en píxeles en que empezaba esa casilla.

Teníamos que distinguir entre el soldado que estaba seleccionado y los que no. Para ello creamos otro dibujo del mismo soldado, pero con distinto color.

Teníamos que tener en cuenta lo que ocurría si teníamos un soldado seleccionado y pinchábamos sobre otro, en vez de sobre una casilla vacía.

La variable mencionada antes 'mouse_b &1' nos indica si está pulsado el botón izquierdo del ratón, pero no es capaz de reconocer un solo click. Es decir, nosotros solo queríamos

Nosotros solo queríamos permitir un movimiento por soldado. Es decir, que lo moviésemos a otra casilla y dejase de estar seleccionado, de forma que si lo queríamos mover de nuevo teníamos que volver a seleccionarlo. Ahora bien, la variable mencionada antes 'mouse_b &1' nos indica si está pulsado el botón izquierdo del ratón, pero no es capaz de reconocer un solo click. Con lo que al mover un soldado, no solo lo movía, sino que lo volvía a seleccionar de nuevo. Para arreglar esto tuvimos que recurrir a variables booleanas de control para saber si el ratón había estado anteriormente pulsado o no. En fin, esta es un ejemplo de los problemas de trabajar a tan bajo nivel.

La función 'blit(…)' dibuja un BITMAP entero en la posición que le digas, pero no te lo ajusta a un tamaño determinado. Es decir, que dibuja el BITMAP entero del tamaño que sea. Por ello, tuvimos que hacer el BITMAP de los soldados justo del tamaño adecuado para que encajasen en cada casilla del mapa (concretamente 25x25 píxeles).

Aparte de todo esto, otro problema a resolver era evitar que el puntero de nuestro ratón parpadease, debido a que repintábamos una y otra vez los BITMAPS en la pantalla.

Como se ve, para una tarea aparentemente sencilla, se encontraron multitud de problemas. Pero aun así, el resultado fue bueno y nos sirvió para aprender todavía mas cosas sobre Allegro.

4.4.3 Ampliando y estructurando

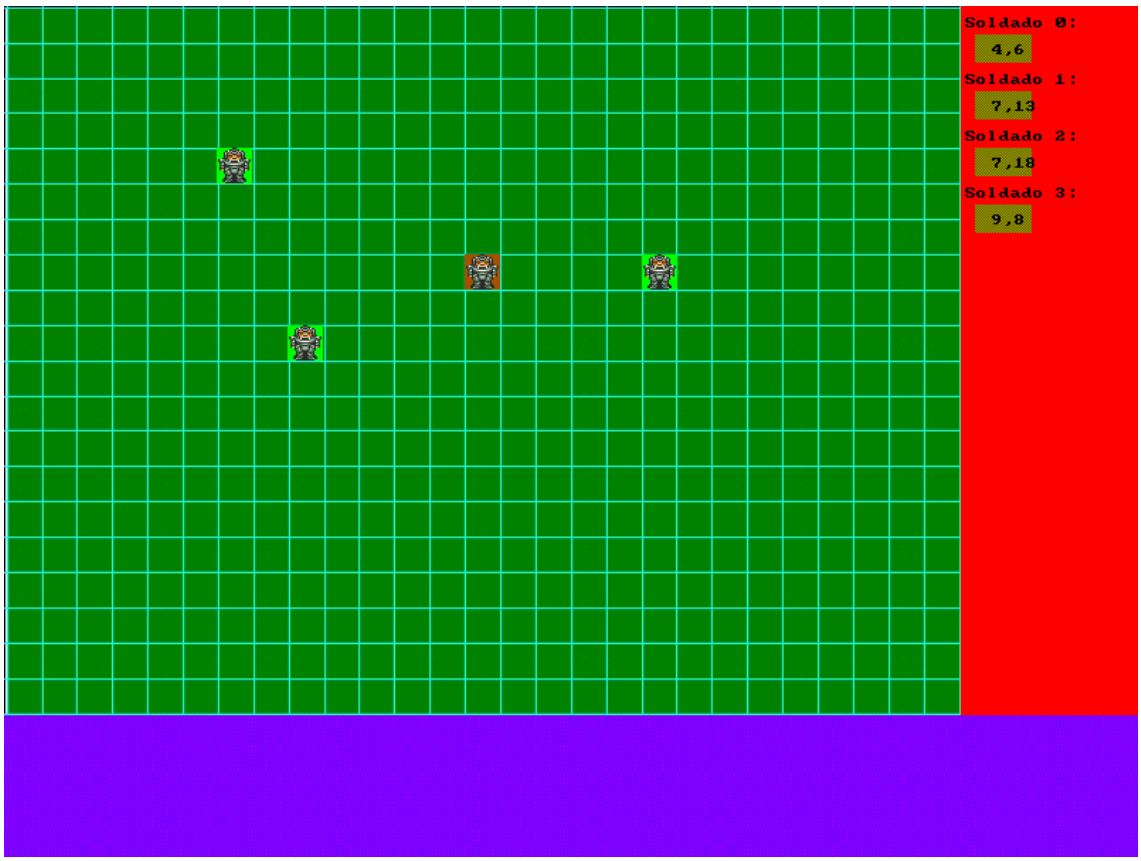
Nuestro siguiente objetivo consistía en: añadir un menú lateral a la derecha que mostrase algo de información, añadir un menú en la parte inferior, y mostrar en medio la parte que habíamos realizado anteriormente.

En una primera versión lo que hicimos fue ir ampliando poco a poco la función que realizaba la prueba anterior. Una vez dominada la función 'blit(...)' no resultó muy complicado hacer que se viesen los dos menús con el mapa en medio. Lo que trajo más quebraderos de cabeza fue cómo meter texto para mostrar información que variase en tiempo de ejecución, ya que hasta el momento solo dibujábamos BITMAPS "estáticos".

Investigamos la función 'textout_ex(…)' y, a base de prueba y error, conseguimos dar con las posiciones para escribir, y con los colores que queríamos. No entraré en mucho más detalle ya que casi todos los problemas eran debidos a entender para qué servía cada parámetro de la función, y a investigar como se fijaban los colores.

Una cosa que si que tuvimos que tener en cuenta fue que si seleccionábamos un soldado y lo intentábamos mover fuera del mapa (a algún menú), no nos diese ningún error y simplemente dejase de estar seleccionado ese soldado.

El resultado que obtuvimos fue el siguiente:



(Sí, es cierto… se podía mejorar bastante el diseño y los colores…)

Se puede ver como en este momento mostrábamos la información sobre en qué casilla se encuentra cada soldado. Nótese que aquí la fila es la primera coordenada y la columna la segunda. Esto lo cambiaremos mas tarde, a la hora de integrar con la parte lógica, para estandarizar los convenios de notación. En ese momento las columnas pasarán a ser la primera coordenada, y las filas la segunda.

Esta vista principal requería estructuración o modularización, que describimos a continuación.

Dividimos el diseño en diversas funciones de forma que la dependencia entre ellas fuese solo la necesaria. De esta forma se pretende que, ante un cambio o retoque en el aspecto gráfico, el código que se deba cambiar no sea mucho y se localice fácilmente. Así mismo, tenemos la posibilidad de dibujar los botones internamente dentro de cada menú y usando medidas relativas a éste y no a toda la pantalla.

Las funciones que implementamos son:

- 'dibujaVistaPrincipal(…)'. Se encarga de dibujar todo lo que mostramos por pantalla. La vista general.
- 'dibujaMenuDcho(…)'. Dibuja el menú de la derecha. Desde esta función se llamará las veces que sea necesario a la de dibujar los botones.
- 'dibujaMenuBajo(…)'. Igual que el anterior pero ahora para el menú inferior.
- 'dibujaMapa(…)'. Dibuja el mapa principal de juego, en el que se desarrolla la acción. También es el método encargado de calcular el ancho y alto del escenario.
- 'dibujaBotonConEtiqueta(…)'. Dibuja un botón con un texto en su interior. Además añade una etiqueta justo encima del botón mostrando el mensaje pasado por parámetro.

Cuando integramos el dibujo de los menús y el mapa con la parte del movimiento de los soldados observamos que el juego se ralentizaba muchísimo. Al principio pensamos que podía ser debido a que en el bucle principal ejecutábamos demasiadas cosas o asignábamos memoria y se producía un fallo en el rendimiento del sistema. Pero después de observar el código descartamos esta opción ya que no podía ser debido a eso.

Al final el problema era que estaba cargando los BITMAPS necesarios una y otra vez en cada vuelta del bucle, con lo que según iba pasando el tiempo nos íbamos quedando sin recursos. Para solucionar este problema creamos una nueva función 'cargarBITMAPS()' que se encarga de cargar los BITMAPS necesarios una única vez al principio del programa.

Como último apunte para esta sección, resaltar que también conseguimos integrar el menú principal con esta versión, de forma que al pulsar sobre el botón 'Empezar' del menú, cambiábamos de pantalla y aparecía todo lo explicado anteriormente.

4.4.4 Crecemos y nos juntamos

La siguiente etapa importante en el desarrollo de la parte gráfica fue el intento de unirla con la parte lógica (Inteligencia Artificial) del juego.

Teníamos dos posibilidades a la hora de afrontar esta tarea:

- Realizar una clase que englobase todo.

- Dejarlo en un módulo aparte con las funciones que fuesen necesarias.

Tras debatir y analizar los pros y los contras de cada posibilidad decidimos dejarlo como un módulo, ya que hacerlo como una clase iba a consumir más recursos, y supondría tener que realizar algunos cambios en el diseño que llevábamos hasta el momento. Tal vez estos motivos no eran demasiados concluyentes, pero como hasta ese momento lo habíamos estructurado todo como un módulo, decidimos que era mejor seguir haciéndolo así.

Primero se integró todo lo que estaba hecho de la parte gráfica en un módulo llamado 'PartidaGUI'. Aquí es donde se encontraban todas las funciones que tenían que ver con Allegro y con los gráficos de la aplicación. Lo que pretendíamos hacer era separar lo máximo posible los gráficos de la lógica de la aplicación, para que los posibles cambios en un sitio no afectasen al resto. Es por ello por lo que decidimos que la interfaz únicamente necesitaría dos funciones para relacionarse con el exterior: 'refrescarEstado(Estado e)' y 'consigueOperador(Estado e)'. La primera se encarga de representar en pantalla el estado que se le pasa por parámetro. La segunda se encarga de analizar el operador que realiza el usuario (comprar soldado, mover a la izquierda, derecha…) y devolverlo.

Realizar la función 'refrescarEstado()' fue relativamente sencillo, ya que es prácticamente lo mismo que ya teníamos realizado. Solo hacía falta cambiarle el nombre a la función 'DibujaVistaPrincipal()' (la cual poseía todo el código necesario) de antes, y hacer algunos retoques. Si acaso, lo más complicado fue ponernos de acuerdo en qué datos teníamos que guardar en el estado para poder representarlo.

Nos encontramos con algunos problemas, como por ejemplo:

- Ahora el número de soldados y su posición venía dado por el estado, con lo que hubo que cambiar la forma en que los dibujábamos en sus casillas iniciales hasta ese momento.

- Al ser los BITMAPS de un tamaño fijo, el mapa no podía tener un número de filas y columnas cualquiera, sino que tenía que ser el exacto para que los

soldados se dibujasen bien en sus casillas. Este es un problema debido al cual teníamos que fijar el ancho y alto del escenario desde el módulo 'PartidaGUI'. Nosotros no queríamos que esto fuera así, ya que lo lógico era que se fijase el tamaño del mapa al construirlo en el algoritmo principal del juego. Pero de momento este problema no lo pudimos solucionar.

- Debido al problema anterior, ya no podíamos tener solo dos métodos que se relacionasen con el exterior, sino que tuvimos que hacer otros dos más para averiguar el ancho y el alto del tablero: 'dameAncho()' y 'dameAlto()'.

- Queríamos que en el menú de la derecha se mostrase información acerca del mejor individuo obtenido en la programación evolutiva. Esta información no saldrá en la versión definitiva del juego, pero la queríamos tener para poder depurar mejor, y ver como jugaba realmente el ordenador. Para hacer esto teníamos dos posibilidades: meter esa información en el estado, o pasarle un parámetro más a la función 'refrescarEstado()': Optamos por esta última forma de hacerlo, ya que considerábamos que alterar la clase Estado era mas crítico y a la larga podría traernos más problemas. De modo que hicimos otra función 'refrescarEstado2(Estado e, TIndividuo i)' para no alterar la que ya teníamos hecha y que funcionaba correctamente. Esta nueva función sería eliminada más tarde, una vez que ya no necesitásemos conocer esa información.

- En cuanto a la función 'consigueOperador()', tuvimos la suerte de poder aprovechar bastante código del que ya teníamos hecho, aunque nos encontramos con algunas dificultades. Por ejemplo, antes seleccionábamos un soldado y lo movíamos a cualquier otra casilla. Ahora esto no podía ser así, ya que un soldado solo puede mover una casilla hacia la izquierda, derecha, arriba o abajo. Para controlar esto tuvimos que tratar cada pulsación del ratón y comprobar que si había un soldado seleccionado y habíamos pinchado en una casilla vacía, solo moveríamos el soldado allí en caso de que la casilla fuese adyacente a donde se encontraba el soldado. Y después de controlar esto todavía teníamos que generar el operador correspondiente y devolverlo.

Con todo lo anterior conseguíamos resolver solo algunos de los problemas, ya que otro muy importante era el que se derivaba de que ahora existían dos jugadores distintos que se podían atacar unos a otros. Para realizar todo lo necesario tuvimos que tener en cuenta varias cosas:

- ¿Cómo distinguir los soldados de un jugador y los de la máquina? Creamos nuevos BITMAPS, con el mismo dibujo, pero de distinto color. Así distinguíamos entre unos y otros. Esta solución se convirtió, a partir de este momento, en una tónica a seguir. Como curiosidad, diremos que para cada

dibujo que aparece en la versión final (soldados, castillos, recursos…) tuvimos que realizar entre 3 y 4 distintos: uno para el jugador humano, otro para el jugador CPU, otro para cuando estaba seleccionado un jugador humano, y otro para cuando un recurso o ciudad no eran de nadie.

- Decidimos dibujar de un color distinto las casillas a las que te puedes mover, para que sea más sencillo para el usuario a la hora de jugar.

- A la hora de mover un soldado, teníamos que distinguir si lo movíamos a una casilla vacía o a una ocupada por un soldado del jugador contrario. También había que tener en cuenta que nosotros no podíamos seleccionar ni mover los soldados del enemigo.

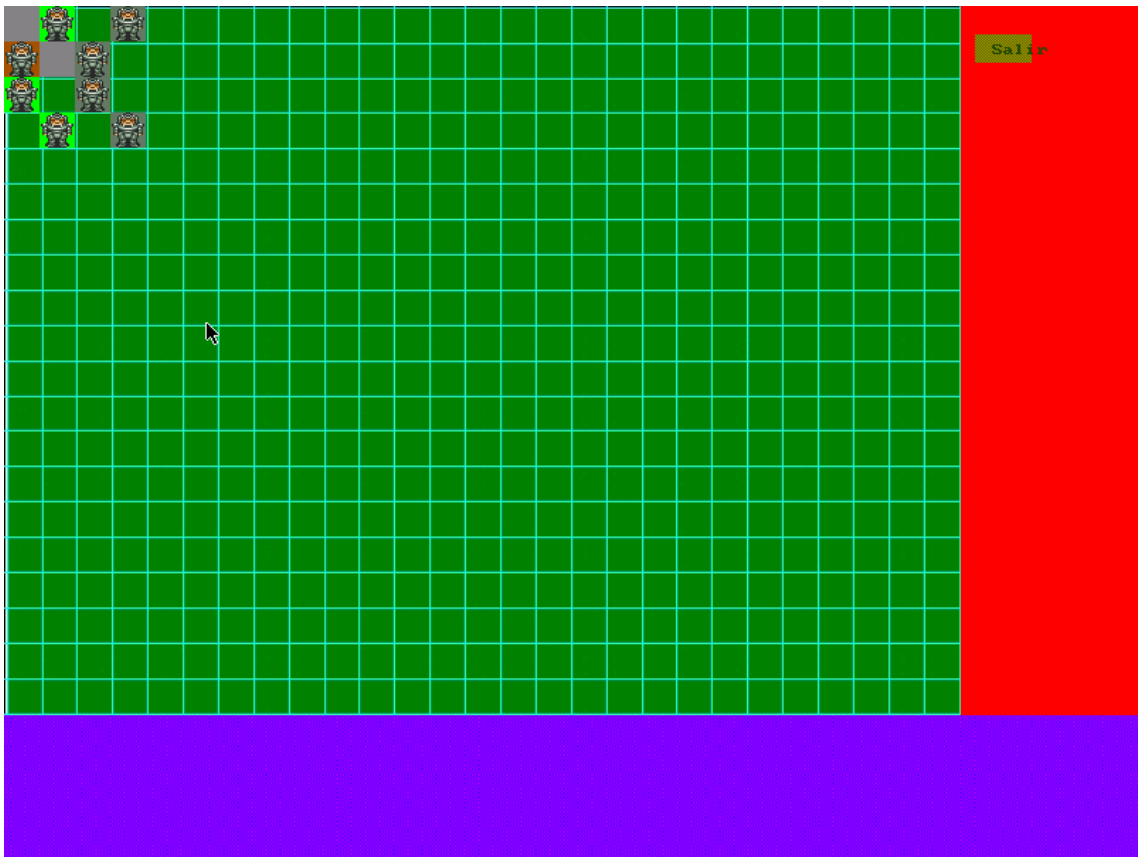
- Añadimos un botón para salir, de forma que al pulsarlo se acabase la partida y se volviese al menú principal. Debido a este botón nos encontramos con los dos problemas de a continuación.

- Al pulsar el botón de salir no volvíamos al menú principal, sino que la aplicación se quedaba colgada y no podíamos continuar. Esto fue arreglado más tarde. El fallo estaba en que al dejar de mostrar el menú principal destruíamos sus BITMPAS (con la función ‘destroy(…)’ explicada anteriormente), pero al ir a mostrarlo otra vez no los recargábamos.

- A veces, al pulsar en el botón de salir no te detectaba la pulsación y por lo tanto no hacía nada. Lo mismo pasaba al pulsar sobre algún soldado. Esto era debido a un problema de diseño, ya que en el hilo de ejecución de nuestro programa existían dos puntos en los que se detectaba una pulsación del ratón. En uno de ellos se detectaba si se había pulsado el botón de salir, y en el otro se detectaban las pulsaciones dentro del mapa. Por ello, si por ejemplo pulsábamos el botón de salir, pero el algoritmo se metía en la detección de una pulsación en el mapa, no se realizaba ninguna acción.

- Otra consecuencia de añadir un botón para salir, es que esa información la teníamos que devolver al exterior, con lo que creamos una nueva función ‘dameSalir()’, que devuelve un booleano indicando si se ha pulsado el botón de salir o no. Al contrario de lo que nos ocurría con las funciones ‘dameAlto()’ y ‘dameAncho()’, esta nueva creada de ‘dameSalir()’ sí que era necesaria e importante para el funcionamiento del juego y queríamos que estuviese así diseñado.

¡Y al final, después de lograr vencer tantas adversidades, conseguimos jugar nuestra primera partida con la interfaz gráfica funcionando! Vemos un instante de una partida en la siguiente captura de pantalla:



(Sí, es cierto… seguíamos pudiendo mejorar bastante el diseño y los colores… y también podíamos hacer el botón de salir un poco más grande para que no se saliese el texto)

Nota: obsérvese que en esta captura de pantalla se ve el puntero del ratón, pero sin embargo en las realizadas anteriormente no se apreciaba. Esto es debido al problema que comentábamos de que el ratón parpadeaba.

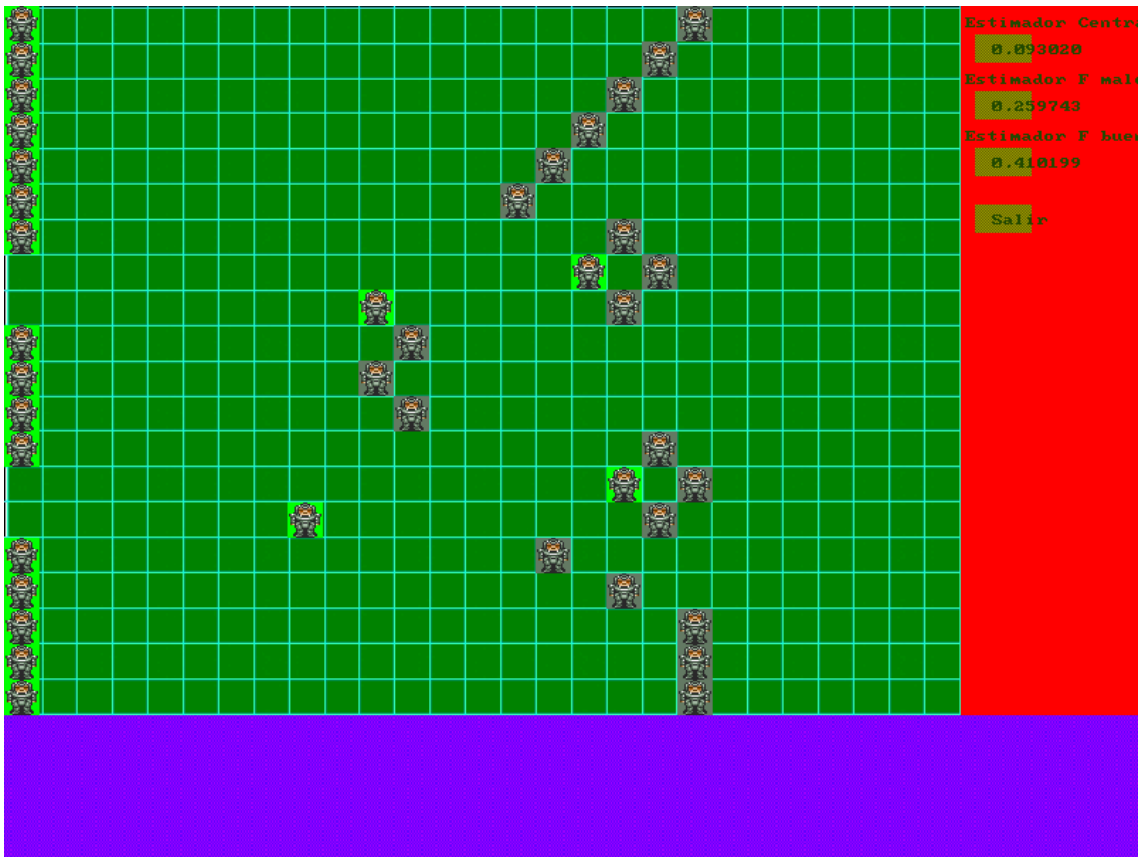
También se aprecia que hay un soldado seleccionado (ubicado en la segunda fila y primera columna), y sus casillas adyacentes (a las que se puede mover) están en gris. También se ve como los jugadores contrarios en vez de tener el fondo verde(claro) lo tienen gris (oscuro).

Vemos que al principio, aunque el mapa fuese mucho más grande, se dibujaban los soldados en las 4 primeras filas y columnas. Esto es porque la parte lógica de momento

solo estaba implementada para jugar en tableros de 4x4. Más tarde se cambiará para jugar en tableros de cualquier dimensión.

Un problema que se nos planteará, y que no tendrá fácil solución, es que solo podíamos jugar poniendo los soldados en tableros cuadrados (del mismo ancho que alto). Aunque el mapa (tablero verde cuadriculado) es rectangular, la alineación de los soldados ha de ser cuadrada para que funcione la aplicación. Este problema tardaremos bastante en arreglarlo. Lo veremos más adelante.

Mostramos aquí una versión más avanzada que la anterior, en la que se ve un estado intermedio de una partida. Se puede ver como en el menú de la derecha aparecen datos sobre los estimadores que utiliza la Inteligencia Artificial para jugar. Son los estimadores (estimador central, estimador fichas malo y estimador fichas bueno).



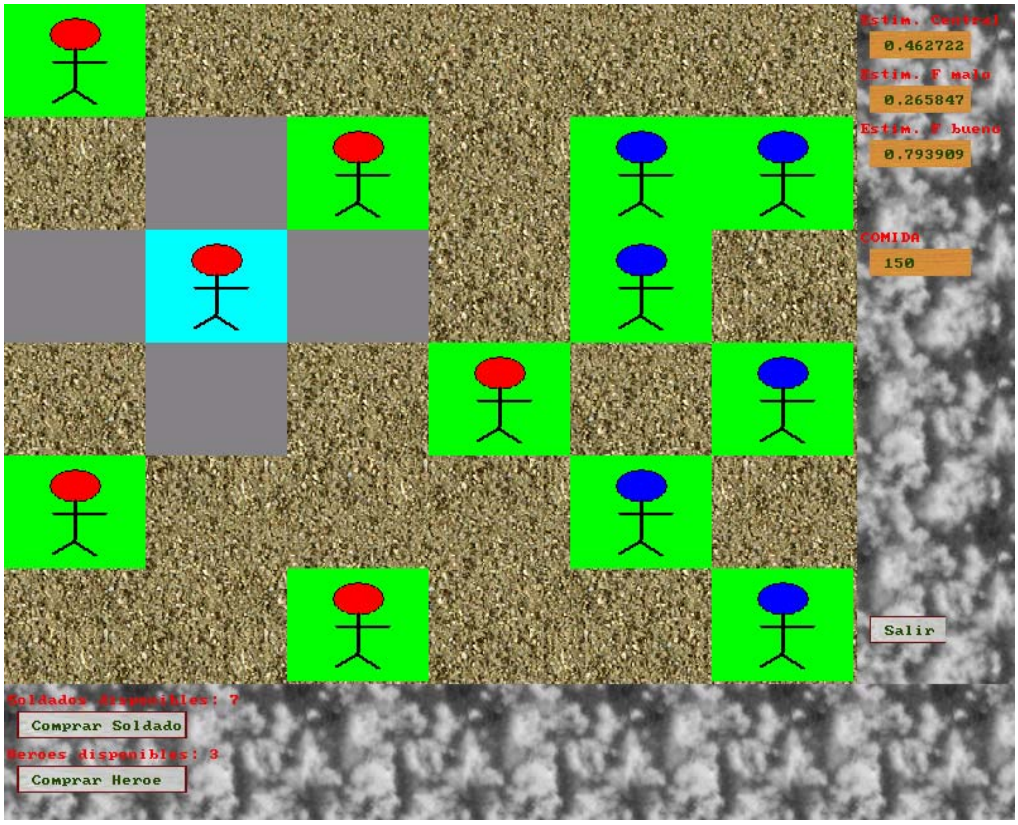
Volvemos a recalcar que, aunque ahora hay más soldados y juegan en un tablero más grande, éste sigue siendo cuadrado.

4.4.5 Mejoramos diseño y nos convertimos en clase

Nuestro siguiente reto consistía en: mejorar el diseño (aspecto gráfico), y convertir el módulo PartidaGUI en una clase.

Hablaremos primero de la mejora del diseño. Lo que queríamos conseguir es muy fácil de explicar. Simple y llanamente, queríamos que el juego fuese más bonito y atractivo. Este es un tema que hasta el final estuvimos mejorando constantemente, como se podrá comprobar en las distintas capturas de pantalla que mostraremos.

Como este paso solo implicaba el crear nuevos BITMAPS y añadirlos, y ningún cambio en el código, no nos supuso un gran problema. Vemos el resultado final:



(Sí, los soldados necesitaban todavía algún que otro retoque… pero el resto ha mejorado bastante)

Desde este momento empezamos a tener dos tipos de botones: los marrones, que son sólo de información, y los grises, que realizan acciones. De momento, las acciones de Comprar Soldado y Comprar Héroe (tenía que cambiarse a Líder) solo estaban a modo de ejemplo, ya que la parte lógica no las tenía en cuenta.

También es importante destacar que ya para esta versión estaban solucionados los dos problemas relacionados con el botón salir.

Por una parte volvíamos al menú principal cuando lo pinchábamos, y ya no daba ese error. Como dijimos anteriormente, lo solucionamos volviendo a cargar los BITMAPS necesarios, y realizando algunos cambios en cuanto a la clase Menú, tales como cambiar el destructor, y retocando el método mostrarMenu().

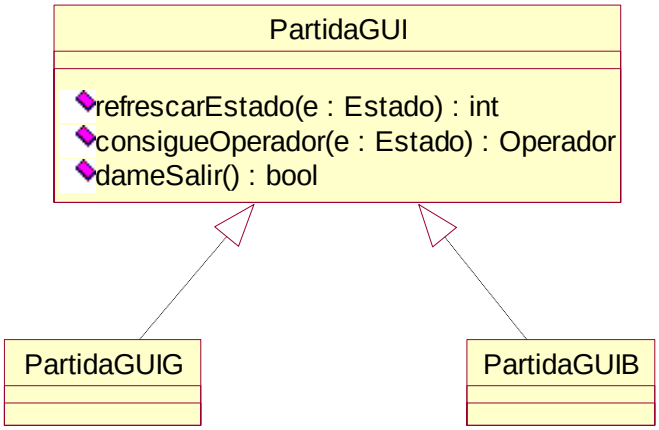
Y por otro lado, solucionamos el problema de que al pinchar en él no nos hiciese caso. Esto conseguimos arreglarlo metiendo todo el código que detecta una pulsación en la parte que antes teníamos únicamente para detectar pulsaciones en el mapa. Esto era algo lógico y sencillo como idea, ya que al tener una única zona de detección de pulsación no se nos escaparía nunca una pulsación del ratón. En un principio era fácil distinguir entre una pulsación dentro del mapa o fuera, pero claro, ¿cómo saber qué botón hemos pulsado? Para ello creamos una estructura que guardase información acerca de la posición inicial en píxeles del botón (x,y), el ancho y el alto del mismo, y un código al cual le asociaríamos más adelante una acción, para saber si lo que queremos es salir, comprar un soldado…

```
struct TBoton

{

   int x,y,ancho,alto,codigo;

};
```

Otro objetivo a realizar era construir una clase para la parte gráfica en vez de dejarlo como un modulo. Desde la parte lógica estábamos empezando a trabajar en el otro tipo de juego de que íbamos a disponer. Es decir, estábamos empezando a distinguir entre la partida general (la que juega con líderes, recursos y ciudades) y la partida batalla (en la que se pelean los soldados de dos líderes contrarios). Hasta el momento, lo que

teníamos realizado nos servía bastante bien para la parte de la batalla, pero teníamos que implementar y diseñar toda la parte de la partida general.

Para seguir el mismo modelo que estábamos realizando en el resto de la aplicación, decidimos construir una clase abstracta PartidaGUI, que fuese extendida por las clases PartidaGUIG, y partidaGUIB, para la partida general y la partida batalla respectivamente.



Vemos en el anterior diagrama como solo necesitamos 3 funciones para relacionar la parte gráfica con el exterior, gracias al cuidadoso diseño para evitar dependencias entre distintas partes.

Con lo cual, primero creamos la clase abstracta PartidaGUI, y a continuación reestructuramos el módulo que teníamos para convertirlo en la clase PartidaGUIB. Este cambio no resultó demasiado complicado, ya que con el tiempo habíamos conseguido tenerlo bastante bien organizado.

Lo que sí que tuvimos que hacer fue, por ejemplo, quitar las opciones para comprar soldados y líderes, ya que en la partida batalla estas acciones no se pueden realizar.

Al final, las funciones que componen la clase PartidaGUIB son las siguientes:

- Las necesarias para extender PartidaGUI:
    - 'int refrescarEstado(Estado *e)'. Dibuja en pantalla el estado pasado por parámetro. Devuelve 0 en caso de éxito y -1 en caso de error.
    - 'Operador* consigueOperador(Estado* e)'. Analiza las acciones que realiza el usuario humano (a partir del estado pasado por parámetro) y devuelve el operador correspondiente.
    - 'bool dameSalir()'. Devuelve cierto si se ha pulsado el botón de salir. Falso en caso contrario.
- Resto de métodos públicos:
    - 'PartidaGUIB(Estado* e)'. Constructor de la clase a partir de un estado dado.
    - '~PartidaGUIB()'. Destructor de la clase.
- Métodos privados:
    - 'int dibujaMenuDcho(int x,int y,int ancho,int alto,EstadoB *estado)'. Dibuja el menú derecho a partir de la posición (x,y), con el ancho y alto especificados. El estado es necesario para mostrar la información en los botones. Devuelve 0 si no ha habido error.
    - 'int dibujaMenuBajo(int x,int y,int ancho,int alto,EstadoB *estado)'. Similar al anterior. Devuelve 0 si no ha habido error.
    - 'int dibujaMapa(int x,int y,int ancho,int alto,EstadoB *estado)'. Como los dos anteriores, pero este se encarga de dibujar el mapa central. Devuelve 0 si no ha habido error.
    - 'int dibujaBotonConEtiqueta(int x,int y,int ancho,int alto,char* etiq,char* bot,int codigo,int modoBot)'. Dibuja un botón con el texto bot, y una etiqueta encima con el texto etiq. El ancho y el alto que queremos para el botón se lo pasamos por parámetro. El parámetro modoBot indica si se trata de un botón de acción o de información. Devuelve 0 si no ha habido error.
    - 'int estaRatonEnRect(int Xini,int Yini,int despX,int despY)'. Método que comprueba si el ratón está dentro del rectángulo pasado por parámetro. Devuelve cero si no ha habido error.

- ‘Posicion* soldadoSeleccionado(EstadoB* estado,TTurno turno,int posX,int posY)’. Si existe un soldado en (posX,posY) devolvemos su posición. En caso contrario devolvemos (-1,-1).

- ‘int cargarBITMAPS()’. Carga todos los BITMAPS necesarios. Devuelve 0 si no ha habido error.

- ‘BITMAP* cargaImagenTGA(char *ruta,PALETTE *paleta)’. Carga el BITMAP que se encuentra en la ruta especificada.

4.4.6 Creando la partida general

El principal objetivo de esta era implementar la clase PartidaGUIG para ser capaces de representar gráficamente el escenario de una partida general.

Una primera idea consistía en partir de la base que teníamos en PartidaGUIB y adaptarlo a la nueva partida. Pero esto, que en principio parecía que iba a ser una labor fácil, se complicó bastante debido a que la forma en que almacenábamos los datos en la clase EstadoB (para PartidaGUIB) y en EstadoG (para PartidaGUIG) eran muy distintos. Mientras que en EstadoB disponíamos de dos listas, una para las fichas del jugador humano y otras para las fichas del ordenador, en EstadoG teníamos una matriz rellena de objetos de tipo ObjetoMapa. Esto suponía que las ideas y estructura que habíamos desarrollado para PartidaGUIB las podíamos reutilizar, pero toda la implementación interna había que cambiarla.

Sin embargo, ahora resultaba un poco más “fácil” dibujar el mapa, ya que lo que hacemos son dos bucles for anidados para recorrer toda la matriz que representa el mapa, e ir dibujando el ObjetoMapa que haya en cada posición, en caso de que exista.

Cabe destacar que en este momento fue cuando solucionamos dos problemas importantes que veníamos arrastrando desde tiempo atrás:

Al fin conseguimos dibujar los BITMAPS del tamaño que nosotros queramos sin necesidad de crearlos de un tamaño específico. Esto lo conseguimos gracias a la función de la librería Allegro ‘stretch_blit(…)’, que dibuja un BITMAP igual que la función ‘blit(…)’ pero pudiendo especificar el tamaño del ancho y el alto en donde queremos que se ajuste la imagen. Con esto conseguimos que el alto y el ancho del tablero se especificase desde la clase Estado, que era lo pretendíamos hacer desde un principio. Ahora ya podíamos dibujar los mapas de forma genérica.

Conseguimos solucionar también el problema de que los soldados tuviesen que estar en un mapa cuadrado en vez de en uno rectangular. Ahora ya también podíamos jugar en tableros rectangulares.

En cuanto a las funciones que hemos necesitado, están todas las que ya teníamos en la clase PartidaGUIB y además hemos añadido alguna más, ya que ahora es necesario tener en cuenta muchos más aspectos:

- Tenemos que dibujar ciudades y recursos, tanto del jugador humano, como los del ordenador, y como los que no son de nadie.

- Los líderes deben tener información acerca del número de soldados que tiene cada uno. Esto lo hacemos mostrando ese número en la esquina superior izquierda del dibujo de cada líder. Habíamos barajado la posibilidad de que esta información apareciese en un botón en el menú inferior para el líder que tuviésemos seleccionado, pero la descartamos ya que para el usuario era mucho más difícil ver la información ya que tendría que ir seleccionando todos los líderes para ver cuantos soldados tiene cada uno.

- Teníamos que representar de alguna forma un estado en el que hubiese algún líder dentro de una ciudad. Para dibujarlo no era demasiado complicado, ya que lo que hicimos fue dibujar el BITMAP de la ciudad o recurso, y en la esquina superior izquierda de éste el BITMAP del líder, pero más pequeño (esto último fue posible gracias a que habíamos conseguido dibujar BITMAPS del tamaño que nosotros quisiésemos). Lo realmente complicado surgió a la hora de implementar el método 'consigueOperador(…)'. ¿Cómo saber si teníamos seleccionada una ciudad o el líder que había dentro? Es más, ¿cómo saber si dentro de una ciudad hay un líder realmente? Después de hablar mucho sobre el tema, llegamos a la conclusión de que lo más lógico y eficiente era meter en un nuevo atributo en las clases Recurso y Ciudad que guardase el Líder que había dentro ('NULL' en caso de no existir). De esta forma, al pinchar con el ratón en una ciudad, la seleccionamos, pero sabemos si dentro hay un líder o no, y podremos realizar las acciones oportunas. Una situación que nos facilitó bastante las cosas es que los operadores que se pueden realizar en una ciudad son distintos de los que puede realizar un líder (en una ciudad solo podemos comprar líderes, mientras que un líder puede comprar soldados o desplazarse en cualquier dirección).

Los métodos públicos son los mismos que en la clase PartidaGUIB (salvo que hemos cambiado el constructor y destructor, obviamente), y en cuanto a los nuevos métodos privados que hemos implementado tenemos:

- 'void dibujaLider(líder* l,int x,int y,int tamX,int tamY)'. Dibuja un líder a partir de la posición x,y del tamaño especificado. Pasamos el líder por parámetro para poder saber cuántos soldados tiene. Decidimos separar esta función del resto del código para facilitar las cosas a la hora de dibujar un líder que se encontrase dentro de una ciudad o recurso.

- 'bool liderSeleccionado(EstadoG* estado,TTurno turno,int posX,int posY)'. Devuelve cierto si en la casilla definida por posX y posY existe un líder perteneciente al jugador definido por turno.

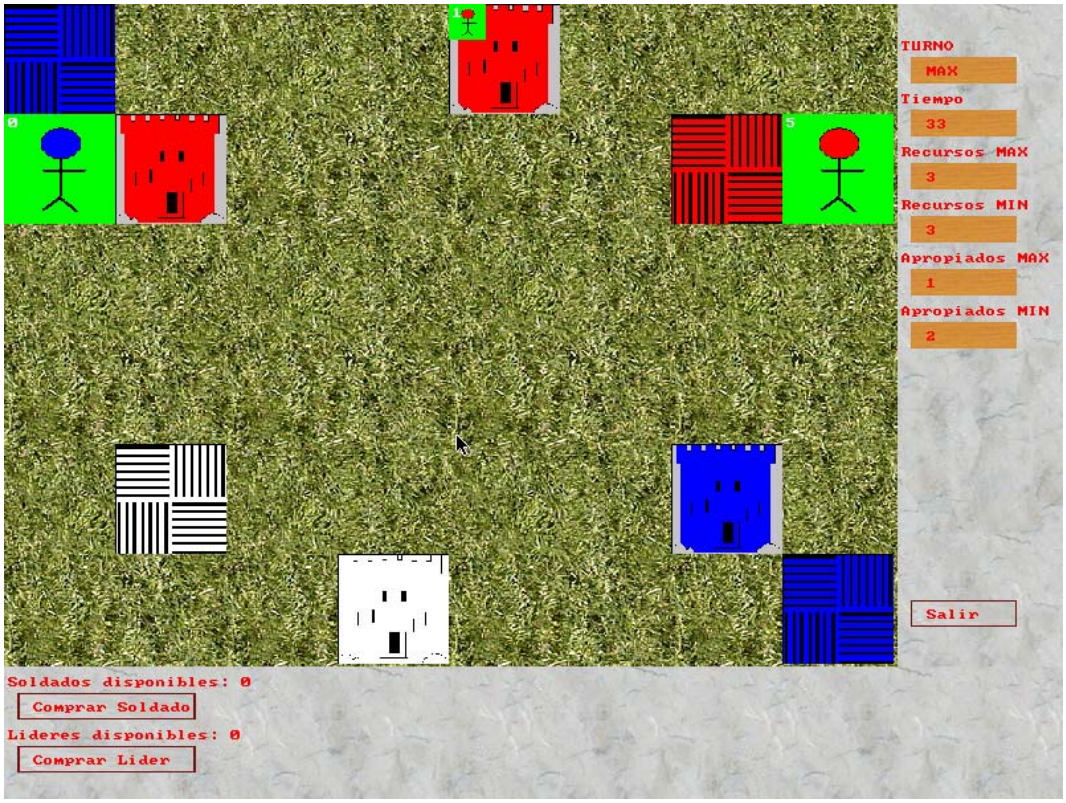
- 'bool ciudadSeleccionada(EstadoG* estado,TTurno turno,int posX,int posY)'. Devuelve cierto si en la casilla definida por posX y posY existe una ciudad perteneciente al jugador definido por turno.

- 'bool granjaSeleccionada(EstadoG* estado,TTurno turno,int posX,int posY)'. Devuelve cierto si en la casilla definida por posX y posY existe un recurso perteneciente al jugador definido por turno.

- 'Posicion* objetoSeleccionado(EstadoG* estado,TTurno turno,int posX,int posY)'. Devuelve Posicion(posX,posY) si en la casilla definida por posX y posY existe un objeto perteneciente al jugador definido por turno. En caso contrario devuelve Posicion(-1,-1).

- 'bool tengoLider(EstadoG* estado,int posX,int posY)'. Devuelve cierto si tengo un líder en la casilla indicada por posX y posY, aunque este dentro de una ciudad o recurso.

Al final, la representación de un estado general nos quedó así:



(Sí, tiene mejor pinta… aunque los soldados son un poco ‘cabezones’…)

Vemos como se ha mejorado el aspecto del menú lateral y el inferior, así como el fondo del mapa. También podemos apreciar que hay diferentes botones para mostrar información, como el número de recursos para comprar de que dispone cada jugador, el número de recursos que cada jugador posee (granjas), el tiempo que queda para acabar la partida, o el turno del que le toca jugar.

También, aunque no se ve demasiado bien en la captura de pantalla, cada líder tiene un número en la esquina superior izquierda indicando el número de soldados que posee. Así mismo, se puede ver como queda representado el que un líder esté dentro de una ciudad.

Hablando de otro aspecto de la parte gráfica, quizá un poco más ligado con la parte lógica, quisimos añadir la posibilidad de que el usuario introdujese su nombre para personalizar las partidas, y que nuestro juego fuese capaz de aprender a jugar contra un

jugador humano en concreto. Para ello creamos una nueva pantalla (de aspecto bastante sencillo) que se muestra cuando el jugador le da al botón de empezar una nueva partida:

También pretendíamos que durante la partida se mostrase el nombre del jugador en el botón de información correspondiente al turno, pero de momento no lo incluimos, sino que lo dejamos para un poco más adelante.

4.4.7 Alcanzamos el final

Por fin llegamos al final. En la última etapa del desarrollo de nuestro juego, pretendíamos hacerlo algo más atractivo de lo que ya era, y para ello quisimos mejorar los gráficos del juego creando nuevos BITMAPS más logrados.

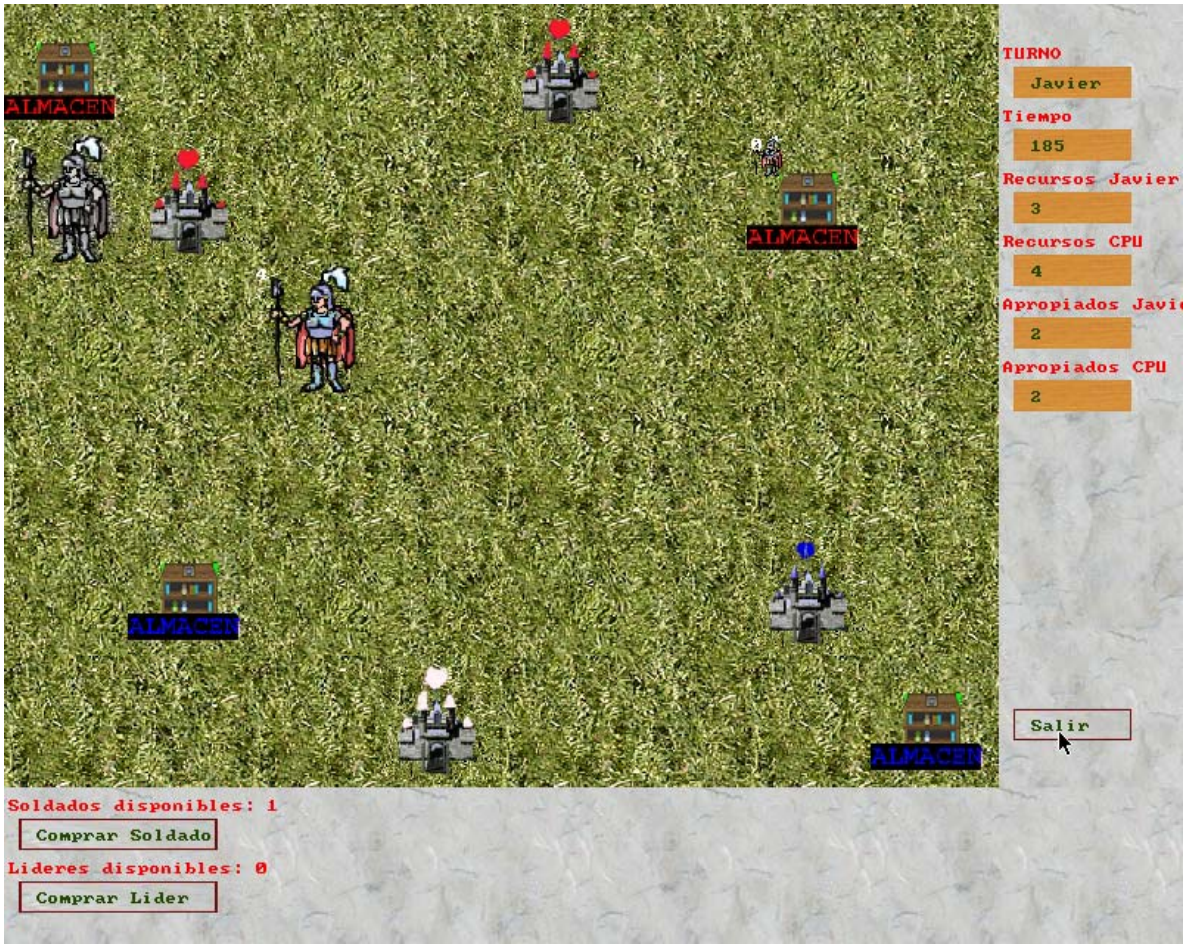
Otro paso importante era el de conseguir unir en una misma partida los dos modos de juego existentes, el modo General y el modo Batalla. Esta parte quizás no era tan crítica para la parte gráfica, ya que si podíamos representar ambos estados (EstadoG y

EstadoB) por separado, no debería haber problemas a la hora de cambiar de uno a otro. Principalmente los problemas que encontramos fueron a la hora de guardar la información oportuna del EstadoG para que a la hora de acabar la batalla entre dos líderes pudiésemos seguir con la partida general.

También investigamos la posibilidad de añadir alguna animación para que el juego resultase todavía más atractivo. Consultamos la documentación de Allegro para ver como se podía realizar y vimos que solo era compatible con animaciones en formato FLI o FLC, de los cuales ninguno de nosotros había oído hablar nunca. En seguida buscamos en Internet información y encontramos algún programa que se suponía era capaz de realizar tales animaciones, pero eran bastante viejos e inestables, y además, en el mejor de los casos, suponía una tarea demasiado tediosa realizar una animación completa, ya que había que dibujar todos los BITMAPS uno por uno. Para acelerar las pruebas, buscamos algún ejemplo que estuviese ya hecho en Internet, y después de ver cientos de páginas al fin encontramos algunos (no demasiado buenos, por cierto, y que difícilmente podíamos aprovechar para nuestro juego). Ya que los habíamos encontrado decidimos probarlos, para ver si quedaban bien y de verdad merecía la pena dedicar tanto tiempo a esto. Pero Allegro no era capaz de reproducir ninguna de las animaciones conseguidas, unas veces por que no habían sido realizadas con un programa compatible, y otras porque simplemente no se cargaban. Así que, después de esta experiencia, decidimos que era mejor aprovechar nuestro tiempo en realizar tareas más útiles.

También conseguimos introducir la información del nombre del jugador para mostrarla por pantalla y hacer que el juego aprendiese contra un humano determinado. Después de barajar varias posibilidades, como por ejemplo pasar el nombre del jugador por parámetro a la función ‘refrescarEstado(…)’, llegamos a la conclusión de que lo más inteligente era añadir un nuevo atributo a la clase Estado, y así disponer de la información en todo momento.

Después de trabajar en todo esto, obtuvimos el siguiente resultado para la partida en modo general:

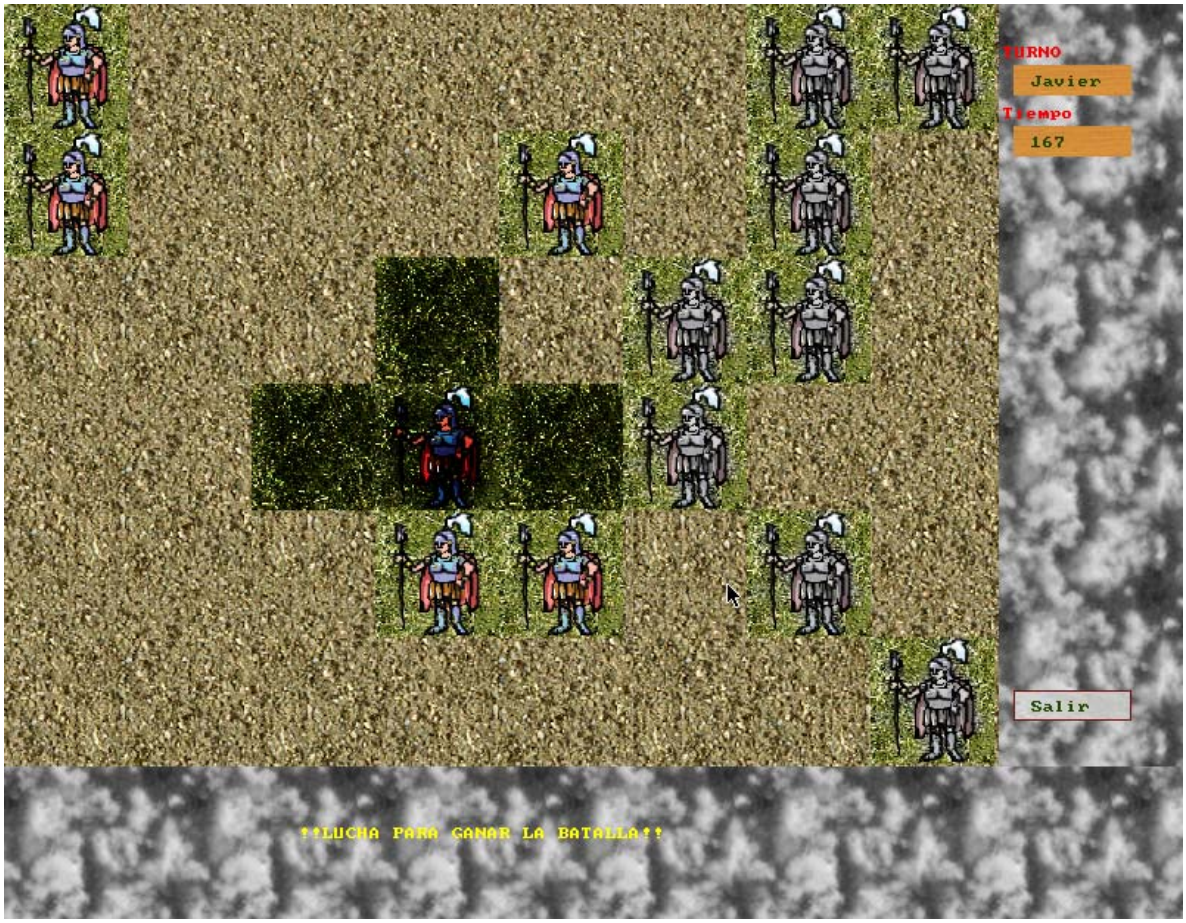


(Ahora sí… ahora estábamos realmente satisfechos con el resultado…)

Podemos observar como a la derecha aparece el nombre del jugador que habíamos introducido previamente. También vemos como ha mejorado el aspecto de los soldados, los castillos y los almacenes (antes los llamábamos granjas). Si se observa el almacén de arriba a la derecha, se ve como hay un soldado en su interior (tal y como lo representábamos en versiones anteriores).

Cabe destacar que los objetos que pertenecen al jugador humano están destacados en rojo, los del ordenador en azul (excepto los líderes, que están en gris), y los que no pertenecen a nadie en blanco.

La siguiente captura de pantalla muestra una partida en modo batalla:



Aquí tenemos que decir que también cambiamos la implementación de esta clase, ya que ahora no dibujábamos un número de soldados por defecto, sino el número concreto de soldados de que dispusiese cada líder, más uno que es el propio líder. Y como es de suponer, al acabar la batalla, el líder vencedor aparecía con el número de soldados que hubiesen sobrevivido a la batalla (menos uno que es el propio líder).

Y por último, y como simple curiosidad, pedimos al lector de estas líneas que compraré el resultado final con aquellos primeros prototipos que consistían en un mapa verde cuadriculado con 4 soldados que se movían por la pantalla.

4.5 Aprendizaje Individualizado

Para un aprendizaje individualizado contra cada oponente usamos el Algoritmo de Escalada en combinación con la PE ya vista. Esta será una de las diferentes alternativas consideradas en el apartado de Introducción y Estudio Preliminar.

Como introducción cabe explicar que el algoritmo de escalada no hace vuelta atrás o "backtracking". Esto es parte de una solución, y sólo cambia su solución por una solución si la nueva es mejor que la anterior.

De la misma forma nuestro sistema empieza con un conjunto de pesos calculados por la PE, y se observaría resultado. Contra un cierto jugador humano, cada nueva partida o cada cierto número de jugadas se evalúan los resultados. En caso de ser cada partida se miraría el resultado final de la partida y en caso de ser cada cierto número de jugadas se observaría si ha sacado desventaja o ventaja y se almacenaría.

En cada recálculo de los pesos por programación evolutiva se introduce en la población inicial, un individuo inicial con el conjunto de pesos que mejor resultado haya tenido hasta el momento.

Cada vez que se evaluasen los resultados se observaría si dichos resultados son mejores o peores que los resultados obtenidos hasta el momento. En caso afirmativo, se guardarán como mejores resultados el conjunto de pesos nuevo. En caso negativo continuaremos con el conjunto de pesos antiguos.

El conjunto de pesos "mejor" así como dicho "mejor resultado" que obtuvo, se guardarán en un fichero y serán recordados cada vez.

4.6 Estimadores

4.6.1 Explicación General de Los Estimadores

Los estimadores evalúan la calidad de un estado o situación de la partida. Las partidas tienen dos modos de juego, como hemos explicado en el apartado de especificación. Por tanto, los estimadores estarán clasificados para cada uno de los modos de juego: "batalla" y "general".

Los estimadores son los que guían la manera de jugar de la máquina. También es muy importante el peso que se le asigne a cada uno de los estimadores con la programación evolutiva.

Todos los estimadores devuelven un numero entre -1.0 y 1.0. Esta cualidad es necesaria para que las combinaciones de los estimadores compuestos, se hagan de forma equilibrada y coherente. Además permite un equilibrio con los finales de partida.

El valor 1.0 indica que la situación es totalmente favorable al jugador MAX, mientras que el valor -1.0 indica que el valor es totalmente favorable al jugador MIN. Por lo general, los valores que devuelven los estimadores son un valor intermedio.

A primera vista, puede parecer que mantener el valor de la estimación entre los límites de -1.0 y 1.0 es algo muy sencillo. Sin embargo, esto no es así para estimadores de complejidad media.

NOTA 1: En los siguientes apartados cuando mencionemos "MAX" o "MIN", nos referiremos al jugador MAX y al jugador MIN respectivamente.

NOTA 2: Además, cuando estemos hablando del modo "batalla", si usamos la palabra "fichas", nos referiremos a los soldados. Mientras que si usamos la palabra "fichas", cuando estemos hablando del modo "general", nos referiremos a los líderes. El término "fichas" por tanto engloba a las unidades que tienen capacidad de movimiento en uno y otro modo de juego.

4.6.2 Estimadores de Modo Batalla

Todos los estimadores expuestos a continuación son sobre el modo de juego "batalla". Este modo de juego ha sido detallado en el apartado de especificación.

NOTA: Por norma, todos los estimadores del modo de juego de "batalla" acabarán con la palabra "batalla". Sin embargo, se omitirá esta última palabra si por el contexto no queda duda. Otra opción es abreviar dicha palabra con "B.". Un convenio similar usaremos más adelante con los estimadores del modo de juego "general".

4.6.2.1 Estimador Fichas Batalla

Este estimador evalúa la partida en modo "batalla" según el número de fichas (soldados) que tiene cada jugador. Para ello, tiene en cuenta la diferencia de fichas entre uno y otro jugador. Ese valor es dividido por el número de fichas totales, para mantenerse en el rango de valores -1.0 y 1.0. De esta manera se tiene en cuenta la proporción de fichas de ventaja que saca un jugador a otro.

Sea

numMax = numero de fichas de MAX

numMin = numero de fichas de MIN

entonces

estimacion = (numMax-numMin) / (numMax+numMin)

En las ejecuciones hemos observado que , al tener en cuenta una proporción de los soldados de uno y otro jugador, el sistema que use este estimador hará "cambios" (atacar aunque luego sea atacado) cuando se encuentra en superioridad de piezas sobre su oponente. Por el contrario, cuando se encuentra en inferioridad de piezas, evitará estos "cambios".

4.6.2.2 Estimador Central Batalla

Este estimador favorece al jugador cuyas fichas estén situadas en posiciones del tablero más centrales. Este estimador es más complejo que el estimador de fichas.

En este estimador, cada posición del tablero tendrá un peso, dependiendo de lo central que esté situada. La fórmula que determina el peso de una casilla es la siguiente (más adelante hay un ejemplo):

pesoCasilla = min((min(x,n-1-x)), min(y,m-1-y))

donde

x,y es la posición de la casilla.

n,m son las dimensiones del tablero.

Para entenderlo mejor, mostramos gráficamente los pesos que tendría las casillas de un tablero de 6x6:

0	0	0	0	0	0
0	1	1	1	1	0
0	1	2	2	1	0
0	1	2	2	1	0
0	1	1	1	1	0
0	0	0	0	0	0

Claramente, observamos que se asigna un peso mayor a las casillas cuanto más centrales están.

En este estimador, cada ficha de cada jugador es multiplicado el "peso" de la casilla. La estimación será la diferencia de la suma de puntuaciones de cada jugador. Para que la estimación se encuentre en el rango -1.0 y 1.0 será conveniente dividir el anterior valor por el máximo posible de puntuación.

Sea

sumaMax = sumatorio de los pesos de las casillas de la fichas de MAX

sumaMin = sumatorio de los pesos de las casillas de la fichas de MIN

entonces

estimacion = (sumaMax - sumaMin) / maximoPuntuacion

El máximo de puntuación posible se dará en una situación del tablero como se describe a continuación. El jugador con más fichas las tiene todas localizadas en las posiciones más centrales (con peso máximo). Su oponente tiene todas las fichas en las posiciones del borde (con peso igual a cero). Por tanto, esta máxima puntuación se calcula de la siguiente forma:

*maximoPuntuacion= maximoPeso * max (numMax, numMin)*

donde

maximoPeso: peso de las casillas más centrales (en el ejemplo de tablero 6x6, el valor sería "2").

numMax, numMin: número de fichas de los jugadores MAX y MIN respectivamente.

Como conclusión, observamos que la estimación favorecerá al jugador con sus fichas colocadas en posiciones más centrales, y además se mantendrá siempre en el rango -1.0 y 1.0. La máquina , al usar este estimador, moverá sus fichas hacia posiciones centrales.

4.6.2.3 Estimador Agresivo Arriesgado Batalla

Este estimador valorará mejor a aquellos estados en el que las fichas (soldados) de un jugador estén cercas de sus oponentes. El sistema, cuando está guiado por este estimador, tiende a aproximar sus fichas a las de su contrincante.

Para cada pareja de fichas, una de cada jugador, se estimará una puntuación. Esta puntuación será mayor cuanto más cerca se encuentren las fichas. Esta puntuación será la siguiente:

$$puntuacion(f1,f2) = (maximaDist - dist)^2$$

donde

f1, f2: cada una de las fichas

dist: distancia entre las dos fichas. Esto es igual a:

$$dist = abs(x1-x2)+abs(y1-y2)$$

donde

(x1,y1),(x2,y2) son las posiciones de las fichas f1,f2.

maximaDist: = distancia máxima entre dos fichas.

$$maximaDist = (n-1) + (m-1)$$

donde

n = número de filas

m = número de columnas

NOTA: En la práctica hemos tomado un valor para la máxima distancia ligeramente superior (n + m)

La puntuación de cada par de fichas tiene en cuenta la distancia. Como queríamos que fuera mayor cuanto más cerca estuviera,la distancia se encuentra **multiplicada por (-1)**.

Sin embargo, **queremos que la puntuación sea positiva**. Por eso, le hemos sumado la mínima constante que lo asegura. Esto es, la máxima distancia.

Además observamos que el valor de proximidad (puntuación) anteriormente descrito se encuentra **elevado al cuadrado**. Esto rasgo lo hemos elegido para dar mayor puntuación cuanto más cerca se encuentre una ficha de una ficha cercana, independientemente que se aleje de una ficha lejana. Si usamos una puntuación "lineal" respecto a la distancia, sucede que las dos siguientes situaciones se valoran con la misma suma de puntuaciones:

Estado 1 (tablero 10 x 1)

M1				*m1*					*M2*

Estado 2 (tablero 10 x 1)

M1	*m1*								*M2*

donde

M1,M2: fichas del jugador MAX

m1 : ficha del jugador MIN

Efectivamente, si usamos un puntuación "lineal", los dos anteriores estados tendrían la misma estimación dada que la suma (puntuación(M1,m1) + puntuación(M2,m1)) permanecería constante. Sin embargo, si usamos un peso "cuadratico", el "Estado 2" tiene una estimación mayor. Como conclusión observamos que el sistema que use una puntuación "cuadrática" tiende a acercar la ficha "m1" a la ficha rival M1, esto es la ficha rival más cercana. Por el contrario, usando la puntuación "lineal", no ocurre lo mismo. Es por esto por lo que hemos escogido la puntuación "cuadrática".

La estimación considera el sumatorio del peso para cada par de fichas. Además es dividido por la máxima puntuación para ser un valor comprendido entre -1.0 y 1.0. Por último, es importante destacar que, **para que tanto el jugador MAX como el jugador MIN lo consideren una cualidad "buena", es necesario determinar el signo según el jugador que esté haciendo la estimación.**

estimacion = + suma / maximaPunt si la estimación la hace MAX

- suma / maximaPunt si la estimación la hace MIN.

donde

$suma = \Sigma_{\text{cada ficha fi de MAX}} \Sigma_{\text{cada ficha fj de MIN}}$ *peso (fi, fj) (sumatorio)*

*maximaPunt = numMax * numMin * maximaDist (máxima puntuación)*

donde

numMax : número de fichas de MAX

numMin : número de fichas de MIN

maximaDist: la máxima distancia posible entre dos fichas (igual que en la definición de peso)

El estimador ha sido nombrado "arriesgado" ya que busca estar cerca de su oponente, sin tener en cuenta que pueda ser atacado por el mismo. En cierta forma, se puede decir que se "arriesga". Este problema es tenido en cuenta en el siguiente estimador.

Este factor de "arriesgarse" que produce este estimador, permite que dos jugadores dirigidos por el sistema, se arriesguen de vez en cuando, permitiendo que se produzcan pérdidas por ambos bandos de manera natural. Cuando no se usa este estimador, en la mayoría de partidas, los jugadores dirigidos por la máquina, suelen tomar una actitud más "precavida". De esta forma, no se producen apenas pérdidas por ninguno de los dos bandos, y las partidas se suelen acabar por consumición del tiempo. Por tanto, podría

decirse que este estimador sirve de "catalizador" en partidas máquina contra máquina. En química, se llama "catalizador" a la sustancia que permite o provoca que se de una reacción química entre varias sustancias. En partidas de máquina contra humano, este estimador permite que la máquina posea ciertos rasgos "humanos", al decidir arriesgarse.

4.6.2.4 Estimador Agresivo Batalla

Este estimador buscará estar cerca de su oponente, pero además dará mucha más importancia a tener más fichas que su oponente. En realidad, este estimador perfeccionará la técnica "agresiva" o atacante del anterior estimador. No sólo se colocará cerca de su oponente, sino que además no depende de la combinación con otros estimadores para atacar cuando ya esté lo suficientemente cerca.

Este estimador tiene en cuenta los posibles ataques y la repercusión del cambio del número de fichas. De esta manera, con este estimador, se favorece el ataque directo en caso de ser posible. También considera los posibles ataques del oponente. Por tanto, tomará una actitud más "prudente" o "precavida", a la hora de acercarse en su último paso.

Por tanto, la estimación se produce por la combinación de dos estimaciones, que ya han sido detalladas anteriormente.

*estimacion = ratio * estimFichas + (1-ratio)*estimAgresArr*

donde

estimFich : similar a estimación del estimador "Fichas"

estimAgresArr : similar a estimación del estimador "Agresivo Arriesgado"

ratio : proporción de combinación (hemos usado 0.9)

En realidad, esta tarea de combinación de estimaciones y elección del peso es más propia de la PE. Sin embargo, dado que este estimación engloba un único comportamiento en sí misma (comportamiento atacante) , hemos decidido mantenerla para un único estimador. Como más adelante veremos, en la validación, el número de estimadores para el que se puede aprender pesos de manera razonable está limitado por

el tiempo de entrenamiento del que dispongamos. Por ello, hemos decidido realizar esta combinación en un único estimador, y dejar al Algoritmo Evolutivo "tomar decisiones" (elegir pesos) de más "alto nivel".

4.6.2.5 Estimador Acorralar Batalla

Este estimador favorecerá al jugador que acorrale a las fichas del rival (normalmente en una esquina). De esta forma se consigue privar de tanta libertad de movimientos al jugador rival y forzarle a llegar a una situación en la cual, al mover una ficha, no le quede más remedio que situarse junto a una ficha rival y exponerse a que le ataquen.

Un ejemplo de esta situación sería el siguiente:

	B				
A					

Si el jugador B consiguiese dejar al jugador A en esa posición, ahora cualquier movimiento de A le llevaría a perder su ficha, ya que al moverla quedaría situado junto a B y éste podría atacarle.

Para conseguir un estimador que haga esto, hemos empleado una técnica de “abarcar el mapa”. Para ello, creamos un mapa del mismo tamaño que el original, pero vacío, y a cada casilla se la asignamos a uno u otro jugador dependiendo de quién esté más cerca de ella (según la distancia de Manhatan).

Para ver como sería uno de estos mapas, mostramos un ejemplo sencillo en el cual solo hay una ficha de cada jugador. Las casillas en subrayadas son las posiciones donde realmente están las fichas, mientras que las marcadas en cursiva son las que nosotros hemos asignado a uno u otro jugador. En caso de que una casilla esté a la misma distancia de ambos jugadores, se marcara con una N indicando que no es de nadie.

A	*A*	*A*	*A*	*N*	*N*
A	*A*	*A*	*A*	*N*	*N*
A	*A*	**A**	A	*N*	*N*
A	*A*	*A*	*N*	*B*	*B*
N	*N*	*N*	*B*	B	*B*
N	*N*	*N*	*B*	*B*	*B*

En la situación anterior vemos como A tiene ventaja sobre B, ya que posee mas casillas marcadas como suyas. Para calcular la puntuación de cada jugador y devolver un valor entre –1.0 y 1.0 empleamos la siguiente fórmula:

casillasMAX-casillasMIN)/casillasTotales

De este modo, será ventajoso que un jugador tenga más casillas marcadas como suyas que el jugador contrario, y tenderá a acercarse a él y a dejarle en una esquina, ya que de

esa forma consigue marcar el mayor número de posible de casillas. Así conseguiríamos dejar al rival en la posición mostrada en el primer tablero.

4.6.3 Estimadores de Modo General

Todos los estimadores expuestos a continuación son sobre el modo de juego "general". Este modo de juego ha sido detallado en el apartado de especificación.

4.6.3.1 Estimador Número de Lideres General

Este estimador favorece al jugador que tiene mayor número de lideres. En el modo "general", cada líder representa una ficha con capacidad de movimiento.

Para favorecer al jugador con más lideres, se tiene en cuenta la diferencia de líderes entre uno y otro jugador. Ese valor es dividido por el número total de líderes, para mantenerse en el rango de valores -1.0 y 1.0. De esta manera se tiene en cuenta la proporción de líderes de ventaja que saca un jugador a otro.

Sea

lideresMax = numero de líderes de MAX

lideresMin = numero de líderes de MIN

entonces

estimacion = (líderesMax-líderesMin) / (líderesMax+líderesMin)

4.6.3.2 Estimador Recursos Apropiados General

Este estimador favorece al jugador que tiene mayor número de recursos apropiados. En el modo general, los recursos apropiados (almacenes) determina qué número de recursos

en bruto (por ejemplo, alimento) se incrementa cada turno a cada jugador. La situación será más favorable, cuanto más recursos apropiados se tenga

Para ello, se tiene en cuenta la diferencia de recursos apropiados entre uno y otro jugador. Ese valor es dividido por el número total de recursos apropiados, para mantenerse en el rango de valores -1.0 y 1.0. De esta manera, se tiene en cuenta la proporción de recursos apropiados de ventaja que saca un jugador a otro.

Sea

recMax = numero de recursos apropiados de MAX

recMin = numero de recursos apropiados de MIN

entonces

estimacion = (recMax-recMin) / (recMax+recMin)

4.6.3.3 Estimador Ciudades Apropiadas General

Este estimador favorecerá al jugador que tenga mayor número de ciudades apropiadas. En el modo general, en las ciudades que el jugador posea podrá crear nuevos líderes, lo cual favorecerá su situación.

Para ello, se tiene en cuenta la diferencia de ciudades apropiadas entre uno y otro jugador. Ese valor es dividido por el número total de ciudades apropiadas, para mantenerse en el rango de valores -1.0 y 1.0. De esta manera, se tiene en cuenta la proporción de ciudades apropiadas de ventaja que saca un jugador a otro.

Sea

ciudMax = numero de ciudades apropiadas de MAX

ciudMin = numero de ciudades apropiadas de MIN

entonces

estimacion = (ciudMax-ciudMin) / (ciudMax+ciudMin)

4.6.3.4 Estimador Número de Soldados General

Para entender bien este estimador, quizás sea necesario recordar que éste es aplicado en el modo "general". En este modo, las unidades con capacidad de movimiento o fichas son los líderes. Cada líder a su vez tiene un número de soldados.

Teniendo clara la diferencia entre líderes y soldados de cada líder, ya podemos comprender que este estimador calcula la proporción de soldados de un jugador frente a otro. Además se considera que cada líder por si sólo tiene el valor de un soldado. Por tanto, la estimación sería la siguiente:

Sea

solMax = número de soldados de MAX, contando cada líder como un soldado

solMin = número de soldados de MIN , contando cada líder como un soldado

entonces

estimación = (solMax - solMin)/(solMax + solMin)

4.6.3.5 Estimador Próximo Recursos General

Este estimador valora en favor de aquel jugador que tiene los lideres más próximos a los recursos. Por tanto, un sistema que use este estimador tiende a acercar sus lideres hacia los recursos, sin ser necesario un árbol de búsqueda de gran profundidad. Como un estado simplemente mejora su calidad para un jugador si éste mueve uno de sus líderes hacia un recurso, el sistema en cada paso tiende a mover el líder hacia el recurso más cercano.

El algoritmo, que seguiremos en este caso para calcular la estimación, es el siguiente.

para cada *recurso R* ***hacer***

fMax <- ficha más acercada de MAX a R

puntMax <- punt(R, fMax)

suma <- suma + puntMax

fMin <- ficha más acercada de MIN a R

puntMax <- punt(R, fMin)

suma <- suma - puntMax

fin para

estimacion <- suma / maximaPunt

donde

punt: puntuación.

punt (R, f) = (maximaDist - dist) ^{2}

donde

R: recurso,

f: ficha

dist: distancia entre la ficha y el recurso (el cálculo es similar a otras distancias explicadas anteriormente).

maximaDist: = distancia máxima entre dos casillas cualquiera (la misma que la definida en el estimador "Agresivo Arriesgado Batalla").

maximaPunt: máxima puntuación

*maximaPunt = numRec * max(numMax, numMin)*

donde

numRec: número de recursos del mapa

numMax: número de fichas (líderes) de MAX

numMin: número de fichas (líderes) de MIN

Como primera observación, la variable "suma" del algoritmo almacena el sumatorio de puntuaciones para MAX menos el sumatorio de puntuaciones para MIN, lo que favorece a uno u otro jugador dependiendo de la posición relativa de sus líderes respecto a los recursos:

$$suma = \Sigma puntuaciones\ para\ MAX\ -\ \Sigma puntuaciones\ para\ MIN$$

En el peso de cada par de fichas está en función de la distancia. Como queríamos que la puntuación fuera mayor cuanto más cerca estuviera (distancia menor), la distancia se encuentra **multiplicada por (-1)** . Sin embargo, **queremos que el peso sea positivo**, por eso le hemos sumado la mínima constante que lo asegura. Esto es, la máxima distancia.

Además observamos que el valor de proximidad (puntuación) anteriormente descrito se encuentra **elevado al cuadrado**. Esto lo hemos escogido para dar mayor puntuación cuanto más cerca se encuentre a un recurso cercano, independientemente de que se aleje de un recurso lejano. Si usamos un peso "lineal", no elevado al cuadrado, sucede que las dos siguientes situaciones se valoran con la misma puntuación:

Estado 1 (tablero 10 x 1)

R1				m1					R2

Estado 2 (tablero 10 x 1)

R1	m1								R2

donde

R1,R2: recursos

m1 : ficha de un jugador

Efectivamente, si usamos una puntuación "lineal", (supongamos que es el único líder del jugador) los dos anteriores estados tendrían la misma estimación dada que la suma (puntuación (R1,m1) + puntuación (R2,m1)) permanecería constante. Sin embargo, si usamos una puntuación "cuadratico", el Westado 2" tiene una estimación mayor. Como conclusión. observamos que el sistema que use un peso "cuadrático" tiende a acercar la ficha "m1" al recurso R1, el recurso más cercano. Por el contrario, usando la puntuación "lineal" no ocurre lo mismo. Este es el motivo por el que hemos escogido la puntuación "cuadrática".

Es curioso destacar que, para cada recurso, se harán las valoraciones sólo para el líder más cercano de cada jugador. En este aspecto se difiere a los estimadores "Agresivos". El efecto de esto será que, para cada recurso, se acercará de cada jugador como mucho un líder. Esto lo hemos hecho así ya que hemos creído que con un líder por cada recurso sería suficiente; y así dejaríamos libre al resto de líderes para otras tareas. Por el contrario, en los estimadores "agresivos" decidimos acercar todos líderes disponibles, ya que pueden producirse bajas en una posible batalla.

4.6.3.6 Estimador Próximo Ciudades General

Este estimador tiene en cuenta la proximidad que hay entre cada ciudad y el líder más cercano de cada jugador. El significado y los cálculos necesarios son muy parecidos al "Estimador Próximo Recursos General" (el estimador anterior), por lo que no los repetiremos. La única diferencia es que se sustituye en el razonamiento y en los cálculos los "recursos" por las "ciudades".

4.6.3.7 Estimador Agresivo General

Para ambos jugadores, este estimador valora mejor los estados en el cual los líderes de un jugador se encuentren cerca de los líderes de su oponente. Además no sólo se tiene en cuenta dicha proximidad, sino que además se tiene en cuenta el número de fichas (líderes) de cada jugador. De este manera, se tinen en cuenta los posibles ataques.

Los cálculos son idénticos a los del estimador "Agresivo Batalla", por lo que los omitiremos. La única diferencia se encuentra en que se tiene en cuenta a los líderes en lugar de a los soldados. Sin embargo, esto es más que razonable ya que lo líderes y soldados desempeñan papeles muy similares en el modo "general" y "batalla" respectivamente, el papel de "ficha".

La evaluación de los resultados de los posibles ataques son algo más complejas que los del modo "batalla", ya que se considera una predicción de la batalla que se produciría entre los dos líderes. Sin embargo, la predicción de los resultados ficticios de la supuesta batalla es independiente totalmente de este estimador en sí. En concreto dicha predicción se hace al aplicar un operador a un estado en el árbol de búsqueda "minimax".

4.6.3.8 Agresivo Inteligente General

El estimador "Agresivo Inteligente General" valorará los estados en favor del jugador que tenga sus líderes fuertes (con muchos soldados) cerca de líderes débiles (con menos soldados) de su oponente. De esta manera, el sistema que use este estimador buscará acercar sus líderes fuertes a los líderes débiles de su oponente. En la misma medida, intentará alejar sus soldados débiles de los soldados fuertes de su oponente.

El algoritmo que seguiremos en este caso para calcular esta estimación es el siguiente.

sumaMax <- 0;

sumaMin <-0;

para cada *lider lMax de MAX* ***hacer***

para cada *lider lMin de MIN* ***hacer***

punt <- punt (lMax, lMin)

si *(ns(lMax) > ns(lMin))* ***entonces***

sumaMax <- sumaMax + peso

sino si *(ns (lMin) > ns (lMin)* ***entonces***

sumaMin <- sumaMin + peso

sino

si *(turnoEstim=MAX)* ***entonces***

sumaMax<-sumaMax+punt

sino

sumaMin<- sumMin + punt

fsi

fsi

fin para

fin para

estimacion <- (sumaMax -sumaMin) / maximaPunt

donde

ns (líder): función que devuelve el número de soldados de un líder. De este manera, determina lo "fuerte" que es.

turnoEstim: turno del jugador que está realizando la estimación. Es necesario para beneficiar a ambos jugadores. Para ello, en cada caso se favorecerá al jugador que esté haciendo la estimación

punt: puntuación.

punt (lider1, lider2) = (maximaDist - dist) ^{2}

donde

dist: distancia entre los dos líderes (el cálculo es similar a otras distancias explicadas anteriormente).

maximaDist: = distancia máxima entre dos casillas cualquiera (suma del número de filas y número de columnas).

maximaPunt: máxima puntuación

*maximaPunt = numMax*numMin*maximaDist (máxima puntuación)*

donde

numMax: número de fichas (líderes) de MAX

numMin: número de fichas (líderes) de MIN

Como primera observación, la variable "sumaMax" y "sumaMin" llevan la puntuación de cada jugador.

Para cada par de líderes, uno de cada jugador, se estimará una puntuación (variable "punt") según la proximidad de cada líder. Este puntuación será asignada o sumada al jugador propietario del líder más "fuerte" (con mayor número de soldados). En caso de que ambos líderes sean igual de "fuertes" , la puntuación será asignada al jugador que esté haciendo la estimación. De esta manera, el sistema que use este estimador tiende también a mover sus líderes hacia otros líderes rivales de igual "fuerza".

De nuevo, al igual que otros estimadores mencionados anteriormente, la puntuación para cada par de líderes depende "cuadráticamente" de la distancia, dando más importancia a las parejas de líderes que se encuentran próximos, frente a los que se encuentran lejanos.

Además, cabe recordar que la estimación se divide entre el máximo de puntuación posible, para proporcionar un valor comprendido entre -1.0 y 1.0 .

Por último, mencionamos que el resultado de ejecutar partidas máquina contra máquina con este estimador en exclusiva, provoca el efecto de que los líderes se persiguen unos a otros. Cada líder fuerte intenta acercarse a un líder débil rival, y cada líder débil huye del líder fuerte del rival más cercano.

4.6.3.9 Estimador Acorralar General

Estimador que da mejor puntuación al jugador que tenga acorralados a los líderes del rival. Este estimador es similar al estimador acorralar explicado en el modo batalla, con lo que aquella explicación sirve también para este.

4.7 Lenguaje y Herramientas usadas

Hemos usado C++, dado que el resultado según las herramientas existentes es que la ejecución de sus programas son de las más eficientes, necesarios para procesos costosos como pueden ser los utilizados en Programación Evolutiva. De hecho, C++ es el lenguaje estándar para la creación de grandes proyectos, entre los que podemos incluir los videojuegos de un determinado tamaño.

En concreto hemos usado Dev C++ como entorno de desarrollo junto con la librería "Allegro". Esta decisión ya ha sido explicada en el apartado de diseño de la parte gráfica.

4.8 Problemas usando Dev-C++

A lo largo de la realización de todo el proyecto, nos hemos ido encontrando con varios problemas con esta herramienta. Unos fueron fáciles de solucionar, pero otros requirieron muchos esfuerzos, a la vez que nos tuvimos que leer prácticamente todos los foros que existen en Internet acerca de Dev-C++.

- Al crear las primeras pruebas, que eran módulos sueltos, todo iba bien, pero a la hora de crear el primer proyecto empezaron a surgir los problemas. Para que funcionase la librería Allegro, teníamos que introducir unos parámetros de configuración en las opciones del compilador. En principio esto era lo único necesario para que todo funcionase, pero al crear un proyecto y haber dependencias entre varias clases situadas en diferentes archivos, nos surgió nuestro primer Linker error. Como en todos los problemas que tuvimos con esta herramienta, buscamos posibles soluciones en foros de Internet.

  Una observación importante para el correcto funcionamiento es que los archivos de cabecera .h de las clases requieren:

  ```
  #ifndef xxxxx_H

  #define xxxxx_H

  .

  · //cuerpo de la cabecera

  .

  #endif
  ```

  Al no ponerlas, existían conflictos ya que existían múltiples declaraciones de una misma clase, siempre que fuese incluida desde dos o más clases externas.

- Después de solucionar el problema de las cabeceras de las clases explicado en el primer punto, todavía seguíamos teniendo errores de linkado. Otra vez buscamos en Internet y, después de probar todas las sugerencias que encontramos, llegamos a la conclusión de que era necesario añadir una librería exclusiva de Allegro a las opciones del proyecto que queríamos compilar, ya que sino no funcionaba.

- Al fin habíamos conseguido eliminar los linker error, pero nuestros problemas no habían acabado. Ahora nos aparecía un Build error, que supusimos que era producido ya en el último paso de la compilación, al crear el fichero ejecutable. De nuevo vuelta a los foros de Internet y a los manuales que encontrábamos en la red. Y de nuevo, acabamos descubriendo que para solucionar el error teníamos que cambiar un parámetro en las opciones del compilador de Dev-C++. En vez de generar los ejecutables con un archivo llamado make.exe, debíamos generarlos con otro llamado mingw32-make.exe.

- Para trabajar el los laboratorios de la facultad, disponíamos de laboratorios que tenían instalado Dev-C++, pero no la librería Allegro. Al no disponer de permisos de instalación tuvimos que idear nuestros propios 'truquillos' para poder fijar las variables de entorno necesarias y ejecutar una versión de Dev-C++ que teníamos nosotros con Allegro ya incluido, desde el directorio de local, que es la única carpeta donde podemos copiar archivos. Para conseguir todo esto tuvimos que crear unos scripts especiales.

- Todavía, de vez en cuando, nos volvía a surgir algún linker error o build error sin que tuviese ningún motivo aparente. Investigábamos a ver si se nos había olvidado alguna opción del compilador o del proyecto, pero todo estaba bien. En este tipo de situaciones, lo que hacíamos era recurrir a una versión anterior del proyecto y volver a rehacer lo que hubiésemos hecho nuevo. Debido a esto, creábamos nuevas versiones cada muy poco tiempo, para no perder demasiada parte del trabajo, y es por ello por lo que al final acabamos teniendo más de 150 versiones distintas.

- A la hora de usar el depurador que traía incorporado Dev-C++ (estaba bien hecho, no tenemos ninguna queja en su contra), teníamos el problema de que una vez que inicializábamos Allegro no podíamos usarlo porque el compilador no lo aceptaba. Con lo que depurar la parte gráfica fue bastante complicado, y tuvimos que recurrir a funciones que mostraban mensajes por pantalla (de hecho, hasta creamos un módulo llamado 'Mensajes.h' para tal función).

- A veces, al modificar las cabeceras de las clases (los archivos .h), Dev-C++ no los recompilaba si no habías cambiado el correspondiente .CPP. Debido a esto, a veces realizábamos cambios en el código que en principio no daban errores, pero que más tarde sí que se mostraban, llevando a confusiones ya que creíamos que el error estaba en lo último que habíamos cambiado, cuando no tenía por que ser así. Al final, cuando nos dimos cuenta de que esto sucedía así, lo que hacíamos era recompilar todo el proyecto cada vez que cambiábamos un archivo de cabecera.

5. Validación y Resultados Obtenidos

En este apartado, exponemos los resultados de nuestro proyecto lo más objetivamente posible, de manera que puede ser útil por futuros trabajos que tengan relación con el tema que se aborda en nuestro proyecto.

La parte principal de nuestro proyecto es la incorporación de la Programación Evolutiva (PE) a los juegos de estrategia. Por este motivo, hemos dedicado parte de nuestros esfuerzos a determinar en qué medida la PE añade un factor de aprendizaje. Si bien es cierto que nuestro proyecto incluye información "experta" o heurística (estimadores); también es cierto que la manera de combinar estos distintos estimadores se deja en mano de la PE.

En las sucesivas pruebas expuestas, simulamos partidas completas (con modo "batalla" y modo "general" entrelazado) entre dos jugadores dirigidos por el sistema (máquina). Ambos jugadores usan exactamente el mismo conjunto de estimadores. El primero de ellos usa unos pesos de ponderación para los estimadores obtenidos como fruto de un aprendizaje por PE. Sin embargo, el segundo individuo usa unos pesos aleatorios de ponderación. De esta forma, sabemos cual es la calidad del aprendizaje llevado a cabo por la PE.

En las últimas pruebas, para validar el aprendizaje individualizado, usaremos otro método que explicaremos posteriormente.

NOTA: En lo sucesivo nos referiremos con las palabras "PE" y "aleatorio" para determinar los jugadores con pesos calculados por PE y pesos aleatorios respectivamente.

En todas las pruebas sucesivas ganará el jugador que, en conjunto, tenga más líderes y soldados cuando se acabe la partida. Una partida podrá finalizar porque algún jugador se quede sin líderes o por tiempo.

5.1 Mejor Prueba o Resultado (prueba 1)

Primero, mostramos la prueba con la mejor combinación de valores de los parámetros. Más adelante, intentamos dar una explicación de por qué esos valores son mejores que otros.

Las características usadas en esta primera prueba expuesta son:

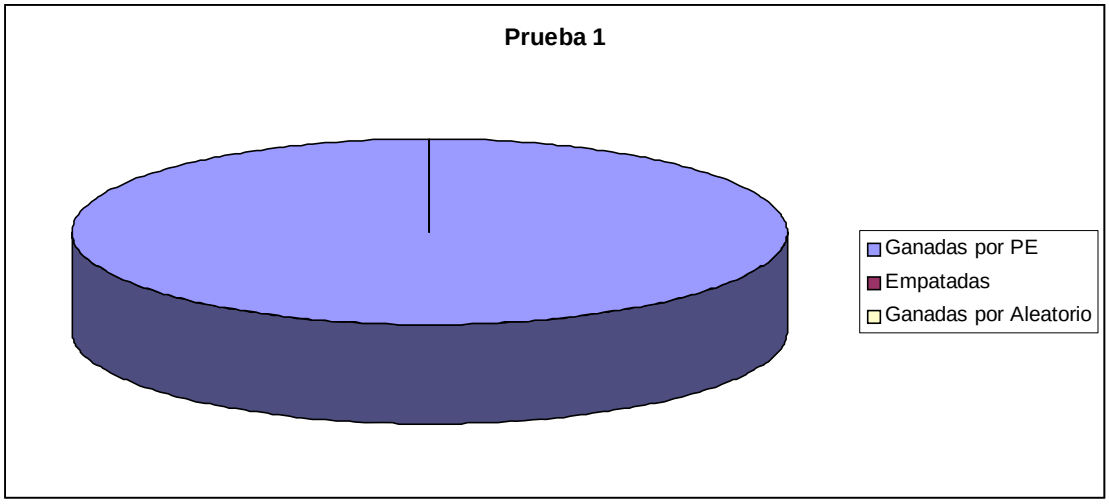
Carácterísticas	*Modo "General"*	*Modo "Batalla"*
Estimadores	*- Número de Soldados* *- Próximo a Recursos* *- Agresivo*	*- Número de Fichas* *- Agresivo Arriesgado*
Tamaño del tablero en la partida.	*8 x 6*	*8 x 6*
Tamaño del tablero en el entrenamiento.	*5 x 4*	*8 x 6*
Tiempo Límite en la partida	*200 movimientos*	*100 movimientos*
Tiempo Límite en el entrenamiento.	*100 movimientos*	*100 movimientos*

Los parámetros propios de la PE son:

Parámetros	*Modo "General"*	*Modo "Batalla"*
tamaño de la población	*3*	*3*
probabilidad de cruce	*0.5*	*0.5*
probabilidad de mutación	*0.1*	*0.3*
periodo de mutación *(cada cuantos ciclos se hace la mutación)*	*1*	*1*
número de generaciones	*5*	*2*

Las prueba se hizo sobre un conjunto de **14 partidas** y los resultados fueron:

Ganadas por "PE"	Empatadas	Ganadas por "Aleatorio"
14	0	0



Como observamos, el jugador entrenado por PE ganó el 100% de las partidas. Estos resultados son más que favorables.

Las partidas de las pruebas eran visualizadas, lo que nos permitía observar los diferentes matices de cada partida.

NOTA: Los apartados siguientes, cuyo número empieza por “5.1” son referentes a esta primera prueba, “prueba 1”. Cuando cambiemos a otra prueba lo diremos explícitamente. También es cierto que los comentarios sobre los parámetros de la “prueba 1”, nos introducirán al resto de las pruebas.

5.1.1 Variedad de las partidas

En primer lugar, cabe destacar que las partidas se desarrollaban de manera totalmente diferentes:

- En algunas partidas, el "jugador PE", tomaba una estrategia agresiva, e eliminaba a su oponente en pocas jugadas;

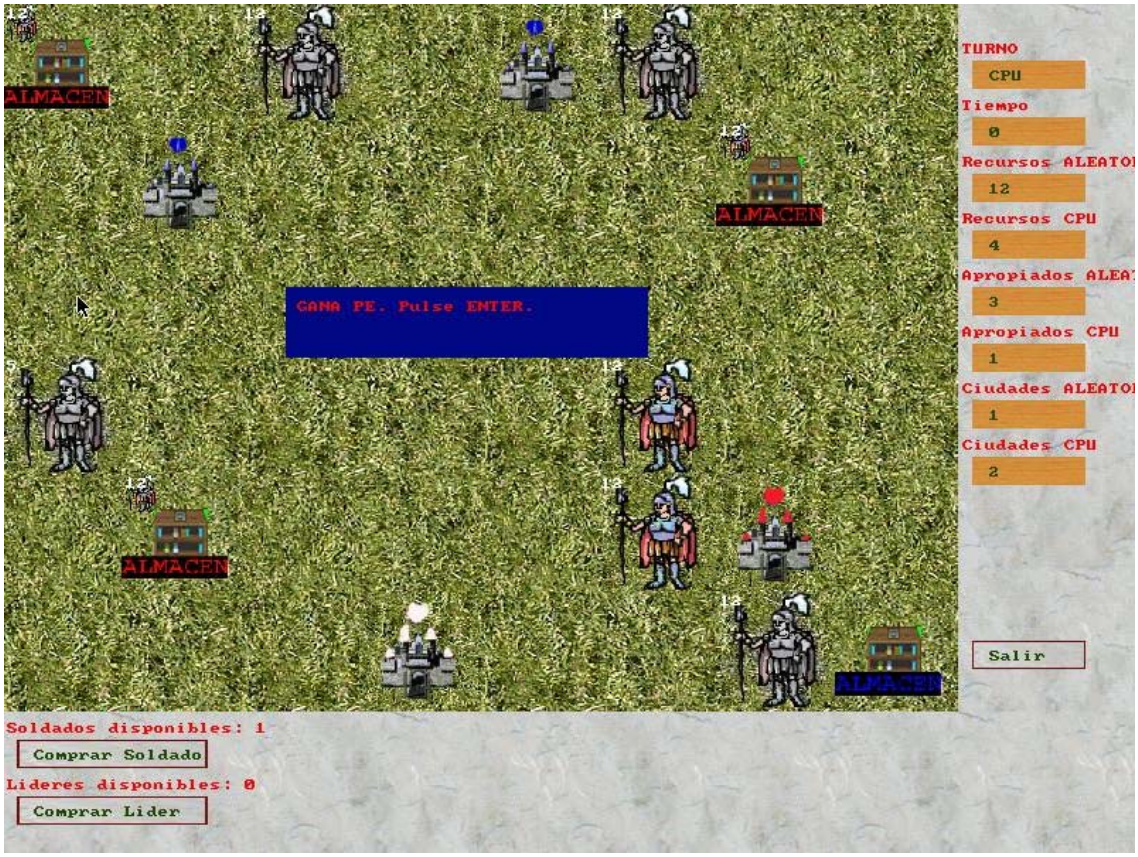
- En otras, el "jugador PE" seguía una estrategia más conservadora, apropiándose de casi todos los recursos del mapa(o todos , dependiendo la partida), pudiendo así aumentar su número de lideres y soldados a mayor velocidad que su oponente y ganarle por tiempo.

Cabe destacar que, aunque las partidas se puedan clasificar en esas estrategias, la secuencia de movimientos de una a otra partida, dentro de una misma categoría, eran también diferentes.

Como categorías se han expuesto los dos extremos, aunque en realidad se podría decir que ,en cada caso, usaba una estrategia peculiar, en un punto intermedio de cada uno de los dos extremos.

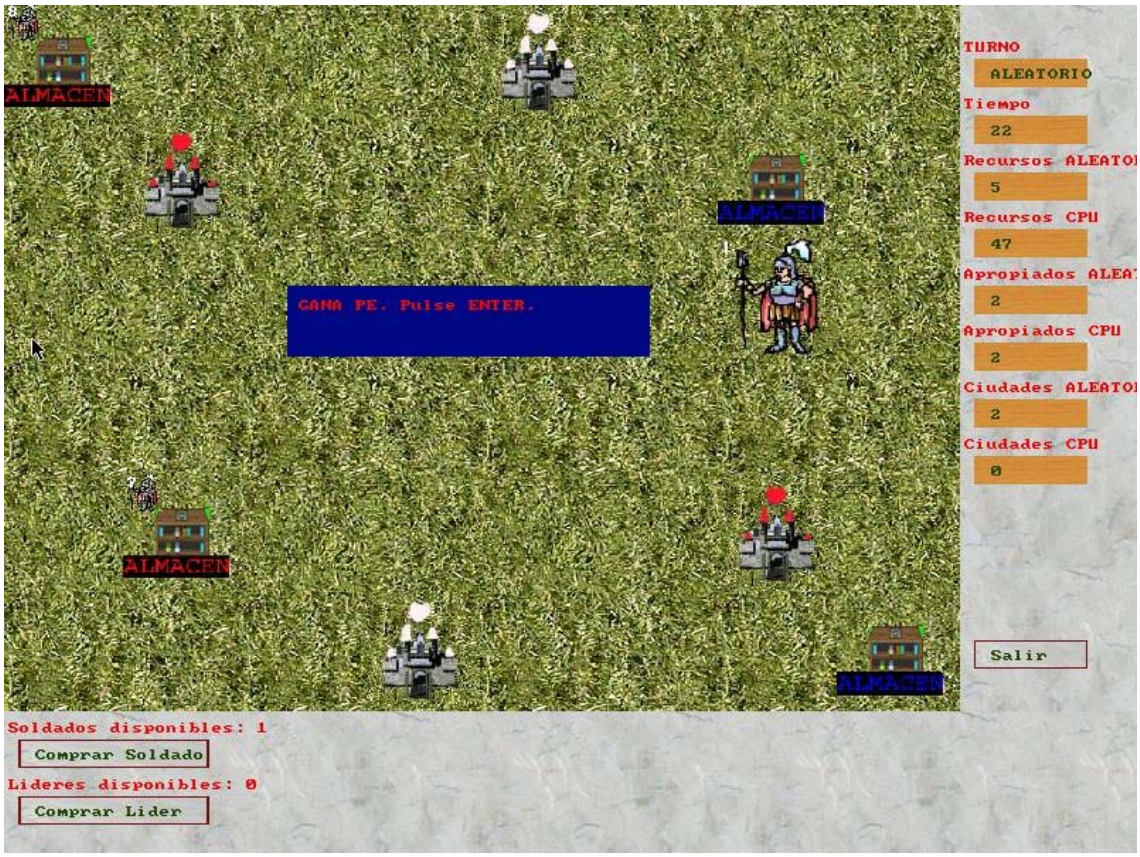
A continuación mostramos dos ejemplos:

Ejemplo 1 de ejecución de la “prueba 1”



El “jugador PE” es representadas por el jugador con color rojo, mientras que el jugador gris o azul es el “jugador aleatorio”. Observamos que el individuo "PE" tiene apropiados tres de los cuatro recursos disponibles en el mapa. Además se ve cómo los recursos apropiados por "PE" están siendo custodiados (en cada "almacen", se puede observar si hay un líder, fijándonos en la parte superior izquierda de dicho "almacen") cada uno de ellos por un líder fuerte (con el máximo número de soldados, 12). Al tener más recursos apropiados, ha podido comprar líderes y soldados más rápido que su oponente. Ha ganado por tiempo (si el tiempo se excede gana aquél que tenga mayor suma de líderes y soldados). En este ejemplo, se puede decir que la "PE" ha seguido una estrategia conservadora.

Ejemplo 2 de ejecución de la prueba 1



En este otro ejemplo, el “jugador PE" ha ganado eliminando a todos los líderes de su oponente. Obsérvese que no hay ningún líder del “jugador aleatorio" (color gris). Se sigue el mismo convenio de colores que para el anterior ejemplo. Se puede decir que en esta partida "PE" ha seguido una estrategia más agresiva.

También cabe decir que, probablemente, la variedad de finales de partida, proviene en gran medida de las diferentes formas de jugar tanto del "jugador PE" como del “jugador aleatorio". Recordemos una vez más que las dos anteriores ejecuciones usan los mismos estimadores, ya que pertenecen a la misma prueba, “la prueba 1”. Lo que varía son los pesos que se dan a estos estimadores.

5.1.2 El modo "general" es más decisivo

Al observar, las partidas de la prueba 1, hemos podido ver cómo la PE es más eficaz en juegos complejos. Me explico; si bien todas las partidas completas(modo general + modo batalla) eran ganadas por “PE”; sin embargo, dentro del modo general, había batallas entre líderes, en las que el resultado no era tan favorable para el “jugador PE”. Cómo los pesos son recalculados por cada individuo, cada vez que se inicia una nueva batalla dentro del modo general, en media, “PE” ganaba más batallas y siempre ganaba la partida general.

NOTA: Hemos hecho pruebas cambiando los estimadores del modo “batalla”, y hemos llegado a la anterior conclusión. Sin embargo, hemos preferido centrarnos en este documento a las pruebas que tienen en cuenta el modo “general”.

De este hecho, deducimos dos conclusiones:

- Es mucho más decisivo, que un contrincante juegue bien en modo "general", ya que batallas hay muchas, y en realidad son más simples, y fáciles de forzar un empate, por ejemplo.

- El entrenamiento por PE actúa mejor en juegos más complejos, como por ejemplo el modo "general", en el que hay recursos y ciudades, y en el que los líderes tiene más o menos fuerza según su numero de soldados y donde se pueden comprar nuevos lideres o soldados. Por contrario, el modo "batalla" es más simple, sólo se pueden mover los soldados.

A continuación haremos ciertos comentarios acerca de la influencia de los distintos parámetros y características.

5.1.3 Tamaños del Tablero

Recordamos que los tamaños de tablero usado son:

Carácterísticas	Modo "General"	Modo "Batalla"
Tamaño del tablero en la partida.	8 x 6	8 x 6
Tamaño del tablero en el entrenamiento.	5 x 4	8 x 6

Después de las pruebas hechas, hemos observado que lo más favorable, para un buen entrenamiento, es que el tamaño del tablero usado para el entrenamiento sea idéntico al tamaño del tablero de las partidas “reales”. Sin embargo, por motivos de eficiencia, esto no es siempre posible, en cuyo caso lo más aconsejable es usar tamaños de tablero lo más parecido posible al tamaño “real”.

En nuestro caso, el tamaño de tablero en modo "batalla" se ha podido usar el mismo que el tamaño del tablero de entrenamiento. Esto se debe a que el modo "batalla" es un juego simple y tiene un factor de ramificación (número de jugadas posibles en cada turno) bajo.

Por lo contrario, el juego en modo "general" es más complejo y tiene un factor de ramificación más alto, y además el factor de ramificación tiende a crecer según avanza la partida. Es normal que según avanza la partida, cada vez hay más líderes y por tanto

más posibilidades y mayor factor de ramificación. Es lo contrario que lo que ocurre en el modo "batalla", en el que el número de soldados disminuye debido a los ataques de uno u otro contrincante. Por todo esto, hemos observado empíricamente, que el tiempo necesario para entrenarse en modo “general” en un tablero de 8 x 6 (tamaño real), con los mismos valores para el resto de parámetros, era inabordable. Hemos usado tableros más reducidos para el entrenamiento, aunque no tan diferente (5 x 4).

Obsérvese los dos tamaños posibles para una partida en modo "general"

Tablero 8 x 6 de la partida real:

Tablero 5 x 4 usado para el entrenamiento:

Se observan las casillas más grandes, ya que hay menos casillas (sólo 5 x 4).

5.1.4 Número de Movimientos Límite

El tiempo límite es mayor en las partidas de las pruebas de validación que en las partidas de entrenamiento. Esto se debe a que las partidas de las partidas de validación simulan partidas reales; mientras que las partidas de entrenamiento se interesa que sean cortas por motivos de eficiencia.

5.1.5 Los Parámetros propios de PE

Brevemente, recordamos los parámetros usados en esta “prueba 1”:

Parámetros	Modo "General"	Modo "Batalla"
Tamaño de la población	3	3
probabilidad de cruce	0.5	0.5
probabilidad de mutación	0.1	0.3
periodo de mutación (cada cuantos ciclos se hace la mutación)	1	1
Número de generaciones	5	2

En general, observamos unos **valores relativamente pequeños de tamaño de población**. Una población de sólo tres individuos (número de vectores de pesos, no confundir con número de estimadores) es una población bastante pequeña comparada con la que suele usar en otros problemas resueltos por PE. Esto se debe a que para calcular la aptitud o calidad de los individuos realizamos un torneo de todos contra todos. A su vez simular cada partida, tiene un coste alto. Por motivos de eficiencia, hemos usado este valor tan pequeño.

Lo mismo ocurre con el **número de generaciones**. También es bastante pequeño, pero aun así hemos obtenido buenos resultados.

La probabilidad de mutación en el modo "Batalla" es bastante alta a lo que acostumbramos a ver en otros problemas resueltos por PE. Este valor, es más grande que de costumbres, ya que el tamaño de la población y el número de generaciones para

ese modo son muy pequeños, y si no, la probabilidad de que ocurriera al menos una mutación en todo el algoritmo evolutivo era baja. En realidad, este valor poco usual ni mejora ni empeora considerablemente los resultados. Recordemos una vez más que en este documento nos hemos centrado en la validación del modo “general”.

5.1.6 Tiempos de Entrenamiento

Los entrenamientos de las pruebas de tres estimadores duraban aproximadamente de unos **7 a 8 segundos**

NOTA: Brevemente, recalcamos que en estas pruebas se juegan partidas completas, es decir, que tienen entrelazada el modo "general" y el modo "batalla"; y que los resultados de uno interfiere en otro, y viceversa, según se determinó en la especificación. Sin embargo, el entrenamiento, por PE, se realiza por separado para cada uno de los dos modos de juego. Esto entrenamiento por separado se hace por motivos de eficiencia. Además el entrenamiento es más efectivo.

5.1.7 Los Estimadores

Una de las cosas más importante a remarcar, es que la calidad del entrenamiento depende muchísimo de los estimadores que se usen.

- En primer lugar, la calidad del entrenamiento depende del número de estimadores que se usen.

- En segundo lugar, la calidad del entrenamiento depende de la combinación de estimadores que se use.

Las anteriores afirmaciones las demostraremos en las siguientes pruebas.

5.2 Otras Pruebas de Tres Estimadores

En concreto , en estas pruebas haremos hincapié que, incluso manteniendo constante el número de estimadores, dependerá mucho en el resultado de las pruebas, qué estimadores usemos.

Mantendremos los mismos parámetros. Lo único que cambiaremos son qué estimadores usar, aunque mantendremos el número de estimadores. Los resultados son:

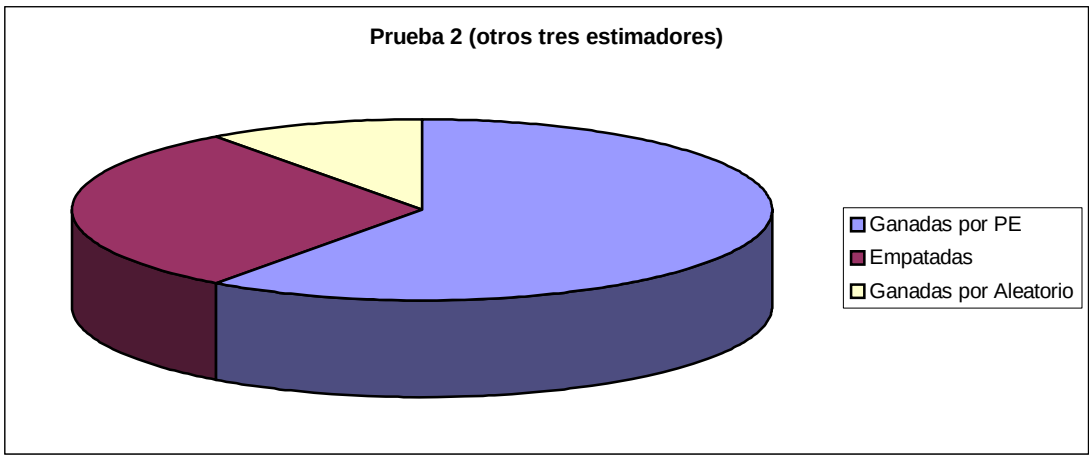
5.2.1 Prueba 2

Mostramos los estimadores usados:

Carácterísticas	Modo "General"	Modo "Batalla"
Estimadores	- Próximo Recursos - Agresivo Inteligente - Agresivo	- Número de Fichas - Acorralar - Agresivo

Los resultados hechos sobre **10 partidas** fueron:

Ganadas por PE	Empatadas	Ganadas por Aleatorio
6	3	1



Observamos que en esta prueba el **porcentaje de victorias del jugador "PE" son menor(60%)**, aunque también es cierto que hay bastantes empates (30%), y **casi ninguna pérdida del jugador "PE".**

Probablemente, esto se deba a que los estimadores usados no son tan diferentes como los usados en la "prueba 1". En este caso los estimadores "Agresivo General" y "Agresivo Inteligente General" se parecen, y el hecho de dar más peso a uno a otro no influye tanto en la manera de jugar. Por ello, la PE tiene menos variedad donde elegir, y por tanto donde poder aprender.

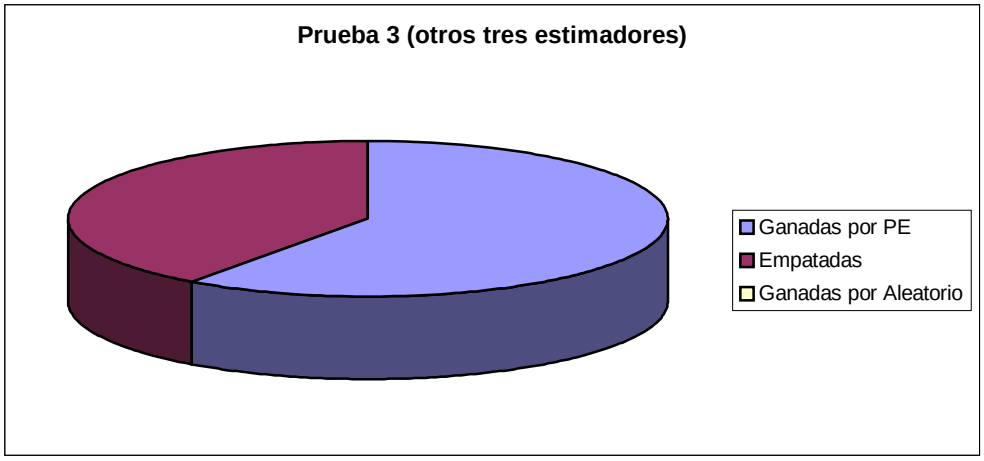
5.2.2 Prueba 3

Mostramos los estimadores usados

Carácterísticas	Modo "General"	Modo "Batalla"
Estimadores	- Número de Líderes - Próximo Recursos - Agresivo	- Número de Fichas - Acorralar - Agresivo

Los resultados de entre **cinco partidas** son:

Ganadas por PE	Empatadas	Ganadas por Aleatorio
3	2	0



Lo destacable de esta prueba es que todas las partidas acaban en empates (40%) o victorias del “jugador PE" (60%).

En realidad, el único cambio frente a los estimadores del modo "general" escogidos en la "prueba 1" (100% de victorias) ha sido el cambio del estimador "Número de Soldados G. " por el estimador "Número de Líderes G." Este cambio ha producido que un 40% de las victorias pasen a ser empates. Esto es razonable, si tenemos en cuenta que el estimador "Número de Soldados" es más "completo" que el estimador "Número Líderes", ya que "Número de Soldados" cuenta el número de soldados y de líderes. Cada líder por si sólo cuenta como un soldado en este último estimador (véase el apartado de estimadores). Por tanto, al ser "Número de Soldados" (usado en Prueba 1) más completo, la “PE” disponía de una "herramienta" (estimador) más potente a su disposición y por eso los resultados fueron mejores.

NOTA: Ahora y en adelante, nos fijaremos en los estimadores usados en el modo "general", prescindiendo de los comentarios acerca de los de modo "batalla", debido a que es el modo "general" el más "decisivo", como hemos ya mencionado anteriormente. En ocasiones mostramos los estimadores usados en el modo “batalla”, pero simplemente como un dato más sin centrarnos en él.

5.3 Pruebas con Diferentes Números de Estimadores.

En esta pruebas intentamos destacar que el número de estimadores usados para el entrenamiento es decisivo. En concreto, debemos tener en cuenta que los individuos de la población del Algoritmo Evolutivo son vectores de "pesos". Por tanto, el espacio de búsqueda tendrá una dimensión igual al número de estimadores. Hasta ahora, las pruebas han sido hecho con tres estimadores, luego el espacio de búsqueda era tridimensional. Para entenderlo bien exponemos los siguientes ejemplos.

* dos dimensiones: Encontrar un punto en un cuadrado (dimensión plana)

* tres dimensiones : encontrar un punto en un cubo (dimensión espacial)

* cuatro dimensiones: encontrar un punto en un cubo en un instante de tiempo (dimensión espacial con tiempo).

* ... y así, que se imagine el lector para más dimensiones.

Con los anteriores ejemplos, intentamos mostrar cómo de "rápido" crece el espacio de búsqueda, según crece el número de dimensiones. Concretamente, este crecimiento tiene orden "exponencial".

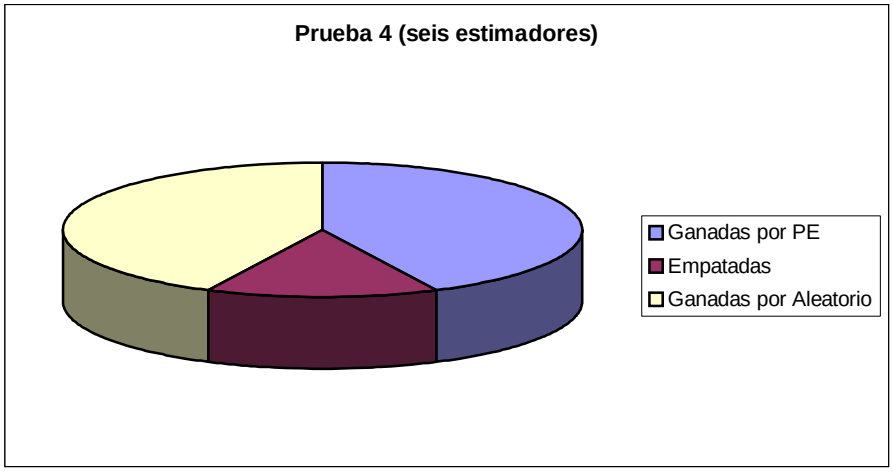
5.3.1 Seis Estimadores (Prueba 4)

Para entender este concepto, si mantenemos el mismo tamaño de población y mismo número de generaciones que en la "prueba 1", e intentamos que el aprendizaje se produzca, sobre un número de estimadores mayor (por ejemplo seis) el aprendizaje es nulo o prácticamente nulo. A continuación mostramos los estimadores usados:

Carácterísticas	Modo "General"	Modo "Batalla"
Estimadores	- Número de Soldados - Próximo Recursos - Agresivo - Próximo Ciudades - Agresivo Inteligente - Acorralar	- Número de Fichas - Agresivo Arriesgado

De **siete partidas** jugadas los resultados fueron:

Ganadas por PE	Empatadas	Ganadas por Aleatorio
3	1	3



En este ejemplo, en concreto, el aprendizaje se podría decir que es nulo o prácticamente nulo, dado que el porcentaje de victorias del jugador "aleatorio" es igual al porcentaje por el jugador por "PE".

Sin embargo, **estos resultados probablemente se podrían mejorar si cambiamos los parámetros propios del algoritmo evolutivo**, tales como el tamaño de la población y el número de recursos. **Al aumentar estos parámetros, también aumentaría considerablemente el tiempo de entrenamiento.** Dado que el espacio de búsqueda crece de forma exponencial en cuanto al número de estimadores, y dado que con tres estimadores se necesitaban unos siete u ocho segundos, puede que este entrenamiento durara del orden de días o semanas, para obtener resultados aceptables. En realidad la estimación del tiempo necesario es muy difícil de hacer, con los datos de los que disponemos . Lo que si podemos asegurar es que el entrenamiento no se podría hacer al empezar cada partida como lo hemos hecho hasta ahora.

5.3.2 Dos Estimadores (Prueba 5)

También es curioso plantearse que ocurre si reducimos aun más el número de estimadores. Por ejemplo, si introducimos dos estimadores, la variedad de estrategias ya es menor.

De nuevo, el aprendizaje depende los estimadores que elijamos. Se puede obtener muy buenos resultados si se usa un ejemplo de “juguete”. Al principio de nuestras pruebas del modo batalla, usamos dos estimadores "Número de Fichas", y "Número de Fichas Malo". El segundo estimador, favorecía justo al jugador que tuviera menos fichas, justo al contrario de lo natural. Por tanto el jugador que diera más peso al segundo estimador buscaría tener menos fichas, o lo que es lo mismo. perder. Pues bien, en estas pruebas, hechas ya hace tiempo, el algoritmo evolutivo siempre asignaba más pesos al estimador "bueno" que al "malo". Por tanto, se podía decir que el aprendizaje era "óptimo".

Aun en el anterior ejemplo de juguete podría ocurrir que un individuo totalmente "aleatorio" produjera una combinación "óptima" de pesos, esto es que asignara mayor peso al estimador "bueno" que al "malo". Luego no podríamos garantizar que el jugador "PE" ganara el 100% de las veces al jugador "aleatorio". Sin embargo, no podríamos afirmar que el aprendizaje no hubiera sido "óptimo"

Algo similar, creemos que ocurre en el siguiente ejemplo con estimadores reales, (y no escogidos a drede como en el ejemplo de “juguete”) :

Prueba 5

Los estimadores usados son:

Carácterísticas	Modo "General"
Estimadores	- Número de Líderes - Próximo Recursos

De **cinco partidas**, se obtuvieron los siguientes resultados:

Ganadas por PE	Empatadas	Ganadas por Aleatorio
3	2	0

Dado que los dos estimadores elegidos no son uno el contrario del otro (como en el caso del ejemplo de "juguete", no se puede afirmar que una combinación de pesos es óptima. Por tanto, las soluciones obtenidas por el Algoritmo Evolutivo probablemente son "casi óptimas" (cuasióptimas), sin llegar a serlo. Probablemente el 40% de empates se deba a las soluciones no sean óptimas totales, y al hecho de que el individuo aleatorio pueda tener una combinación "buena" de pesos por azar (al ser un espacio de búsqueda reducido, dimensión plana).

Sin embargo, cabe destacar que, en esta prueba, el jugador "PE" no pierde ni una sola partida contra el individuo "aleatorio".

5.4 Validación del aprendizaje individualizado.

En este último apartado, intentamos medir la calidad del aprendizaje individualizado. Para ello, **tenemos dos jugadores. Ambos usan los mismo estimadores y calculan sus pesos mediante PE. La diferencia es que el primero de ellos usará el algoritmo de escalada para el aprendizaje contra su oponente, mientras que el segundo mantendrá fijos sus pesos,** guardándolos en un archivo. De esta manera, ambos individuos usan estrategias razonables (no aleatorias), pero sólo uno de ellos aprenderá a lo largo de diferentes partidas.

En todas las pruebas mostramos las partidas jugadas en orden secuencial para ver la evolución del aprendizaje individualizado.

NOTA: En lo sucesivo usaremos la palabra “aprendizaje” para referirnos al jugador que usa aprendizaje, y “participante” para el jugador que mantiene fijos los pesos.

A continuación mostramos los resultados obtenidos.

5.4.1 Aprendizaje óptimo (Prueba 6)

Mostramos los estimadores usados en esta prueba:

Características	*Modo "General"*	*Modo "Batalla"*
Estimadores	*- Próximo Recursos* *- Agresivo* *- Agresivo Inteligente*	*- Número de Fichas* *- Agresivo* *- Acorralar*

Los resultados de las partidas jugadas fueron:

Número de Partida	*"Aprendizaje"*	*Empate*	*"Participante"*
1ª	*X*		
2ª	*X*		
3ª	*X*		
4ª	*X*		
5ª	*X*		
6ª	*X*		

En esta prueba observamos como el "participante", desde un principio, empezó perdiendo. Según esto, ya en la primera partida, era mejor la estrategia de "aprendizaje", con lo cual recordaría la estrategia para las siguientes partidas. Al no poder evolucionar "participante", siempre tenía una estrategia peor que la de "aprendizaje".

El desarrollo de la partida observado era muy parecido, e incluso idéntico en la mayoría de las veces. Esto probablemente se deba a que, en la mayoría de los casos, la estrategia aprendida resultaba ganadora tras las diferentes ejecuciones del algoritmo evolutivo.

Recordamos que aunque "aprendizaje" tenga una buena estrategia, siempre intenta mejorarla con una nueva ejecución del algoritmo evolutivo.

5.4.2 Resultados heterogéneos (Prueba 7)

En esta prueba el aprendizaje sólo se hará en el modo "general". En el modo "batalla" se introducirá el estimador "catalizador", estimador "agresivo arriesgado". Este estimador provocará que las batallas tengan resultados variados. Esto provoca que las

partidas “generales” tengan resultados tan heterogéneos, como los que se muestran en esta prueba.

Los estimadores usados fueron los siguientes:

Características	*Modo "General"*	*Modo "Batalla"*
Estimadores	*- Próximo Recursos* *- Agresivo* *- Agresivo Inteligente*	*- Agresivo Arriesgado* *- Agresivo* *- Acorralar*

Los resultados fueron los siguientes:

Número de Partida	*“Aprendizaje”*	*Empate*	*“Participante”*
1ª			*X*
2ª			*X*
3ª		*X*	
4ª			*X*
5ª	*X*		
6ª	*X*		

7ª	*X*		
8ª		*X*	
9ª			*X*
10ª			*X*

Para ver en que medida aprende el jugador “aprendizaje” mostraremos los porcentajes de los resultados de los siguientes conjuntos de partidas:

- Las partidas hasta que “aprendizaje” ganó la primera partida, con esta incluida (se corresponde con las primeras cinco partidas de nuestra prueba).
- El resto de las partidas (se corresponde con las cinco últimas partidas).

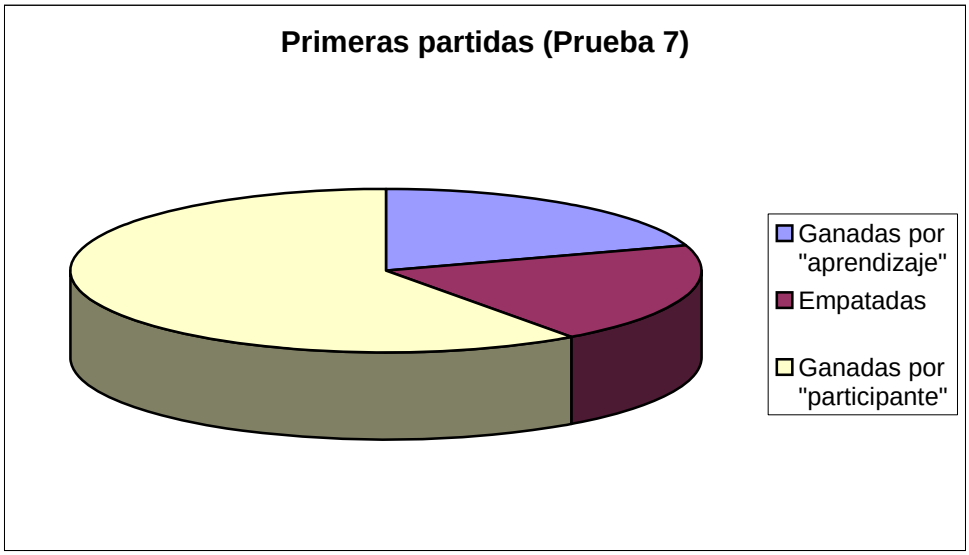
Los datos referentes a los porcentajes y los gráficos son:

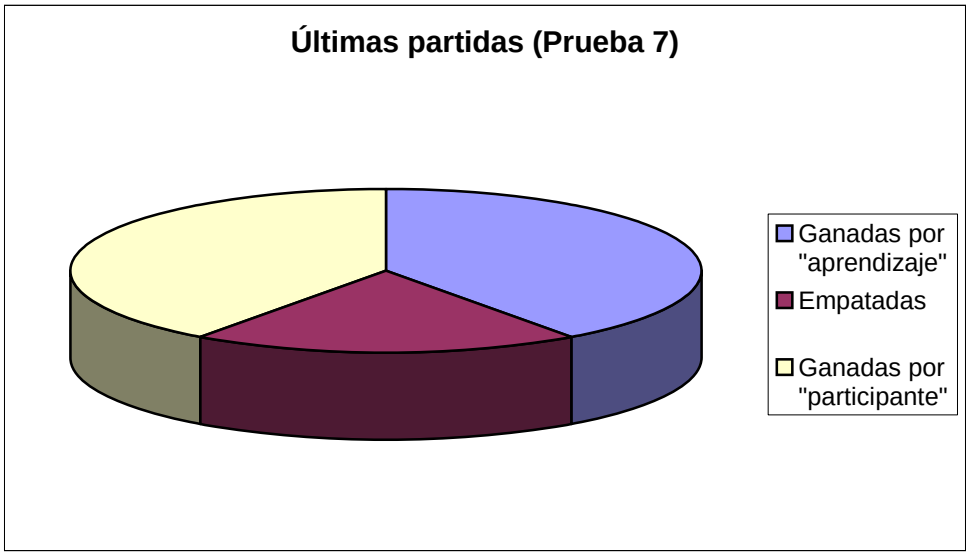
Primeras partidas

Ganadas por "aprendizaje"	*Empatadas*	*Ganadas por "participante"*
1	*1*	*3*

Últimas partidas

Ganadas por "aprendizaje"	*Empatadas*	*Ganadas por "participante"*
2	*1*	*2*





Observamos como el jugador “aprendizaje” ha experimentado una evolución en su juego contra el jugador “participante”. Inicialmente sólo ganaba el 20% de las partidas, mientras que su oponente ganaba el 60%. Después de ganar la primera partida, vemos como el porcentaje de partidas ganadas por “aprendizaje” aumenta hasta el 40%, igualándose con el porcentaje de partidas que gana su oponente.

A partir de estos resultados, deducimos que el jugador “participante” empezó con una estrategia “bastante buena”, que era capaz de ganar a la del jugador “aprendizaje”. Sin embargo, con el tiempo, el jugador “aprendizaje” ha ido experimentando con distintas estrategias hasta conseguir vencer a su oponente. En el momento en que gana la primera partida, recuerda esa estrategia para usarla en sucesivas ocasiones. Una vez ganada esta primera partida se ve que los resultados se igualan entre los dos jugadores.

Recordemos que el jugador “aprendizaje” seguirá aplicando el algoritmo evolutivo para mejorar su estrategia contra su oponente. Sin embargo, probablemente el jugador “participante” tiene una estrategia “bastante buena” como para ser mejorada.

6. Manual de Usuario

Se pretende dar en este documento una guía de usuario para saber cómo jugar a nuestro juego. Se detallan las instrucciones paso a paso.

Para empezar debemos arrancar el ejecutable: archivo Eras.exe.

Acto seguido aparece el Menú Principal del juego:

En esta pantalla, podemos seleccionar alguno de los botones de la izquierda.

El botón de *Créditos* muestra la siguiente pantalla:

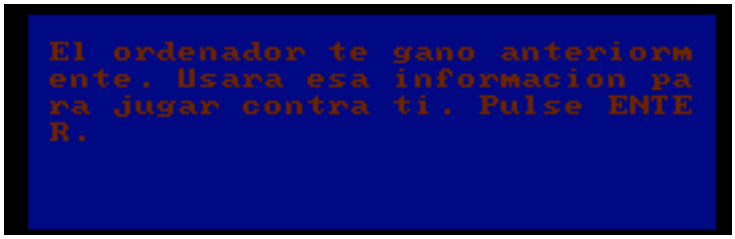
Si pulsamos el botón de *Volver* regresamos al Menú Principal.

Desde el Menú Principal, si pulsamos el botón de *Salir*, saldremos de la aplicación.

Para empezar una nueva partida, pulsaremos el botón *Empezar*. A continuación nos aparece la siguiente pantalla:

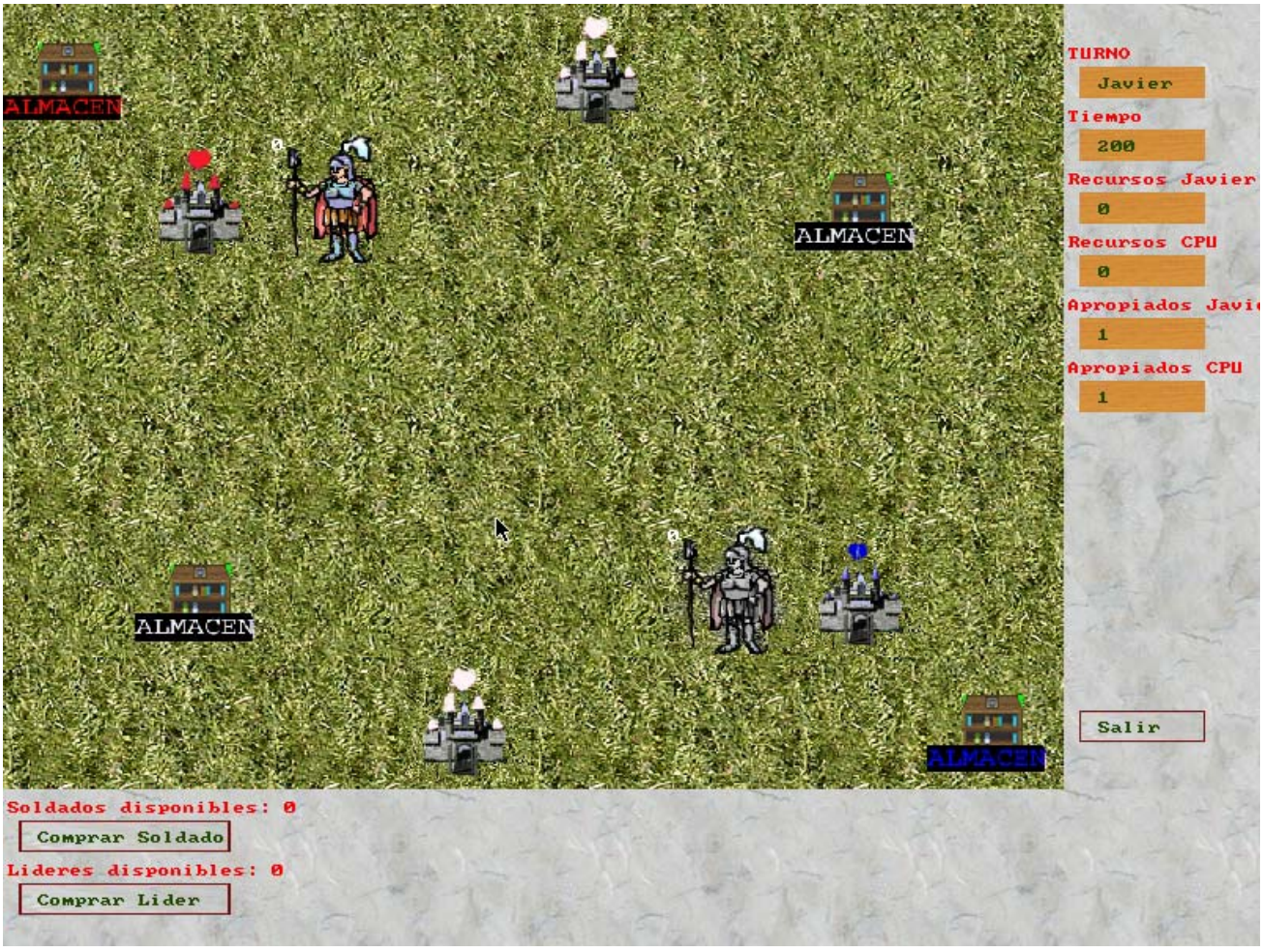
En ella, debemos introducir nuestro nombre en el mismo sitio donde aparece en la imagen el nombre de Javier (de ahora en adelante usaremos este nombre como ejemplo de un posible nombre de usuario, pero se puede introducir el que se desee), y después pulsaremos el botón OK con el ratón.

Si ya habíamos jugado antes con ese nombre y el ordenador nos había ganado, usará esa información aprendida, mejorándola más todavía, para jugar contra nosotros. Nos mostrará el siguiente mensaje, y tendremos que pulsar la tecla Enter.



Después aparecerá el siguiente mensaje informativo, que nos indica que el ordenador está realizando el proceso de aprendizaje. Cuando se nos indique pulsaremos la tecla Enter.

A continuación aparece el escenario principal del juego, que tiene el siguiente aspecto:



Pasamos a describir la escena:

Los soldados son Líderes. El objetivo del juego es acabar con todos los líderes del jugador contrario. Nuestros líderes tienen la capa de color rojo, y los del ordenador de un color más grisáceo.

Los castillos representan ciudades. Las rojas son las nuestras, las azules las del ordenador, y las blancas no son de nadie.

Los almacenes representan recursos. Los rojos son los nuestros, los azules los del ordenador, y los blancos no son de nadie.

Cada líder posee un número de soldados que viene indicado como un número en la esquina superior izquierda del dibujo que lo representa. En la siguiente imagen se aprecia que el líder tiene 0 soldados.

El menú de la derecha muestra varios botones:

TURNO: nombre del jugador del cual es el turno. Puede ser el nombre del jugador o CPU indicando que juega el ordenador.

TIEMPO: tiempo restante para la finalización de la partida, si es que ningún jugador acaba antes con todos los líderes rivales.

RECURSOS JAVIER: muestra el número de recursos de que dispone el jugador humano.

RECURSOS CPU: muestra el número de recursos de que dispone el ordenador (CPU).

APROPIADOS JAVIER: muestra el número de almacenes que tiene en posesión el jugador humano (los que ha conquistado durante la partida).

APROPIADOS CPU: muestra el número de almacenes que tiene en posesión el ordenador (los que ha conquistado durante la partida).

SALIR: al pulsarlo sale de la partida actual y vuelve al Menú Principal.

El menú inferior muestra los siguientes botones:

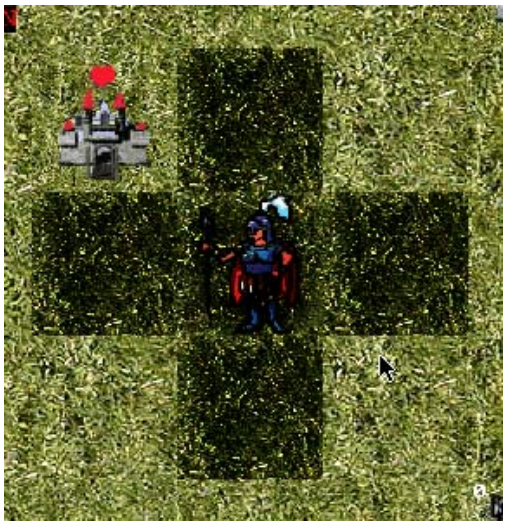
COMPRAR SOLDADO: Si tenemos soldados disponibles (para comprarlos) y un líder seleccionado, al pulsar este botón se incrementará en uno el número de soldados de que dispone el líder que teníamos seleccionado.

COMPRAR LÍDER: Si tenemos líderes disponibles (para comprarlos) y una ciudad seleccionada, al pulsar este botón aparecerá un nuevo líder (inicialmente con 0 soldados) en la ciudad seleccionada. Es requisito que en la ciudad no haya en ese momento ningún líder.

Ahora describiremos cómo se juega una partida. Como ya hemos dicho anteriormente, el objetivo del juego es acabar con todos los líderes del jugador contrario antes de que se acabe el tiempo. Cada jugador (humano y CPU) puede ejecutar una sola acción por turno.

Las acciones que podemos realizar para alcanzar tal fin son las siguientes:

Desplazar un líder a una casilla adyacente. Podemos seleccionar un líder pinchándolo con el ratón y moverlo una casilla hacia arriba, abajo, derecha o izquierda, pulsándola también con el ratón. Las casillas a las que nos podemos desplazar aparecen sombreadas.



Comprar un líder en una ciudad. En caso de que tengamos líderes disponibles, podemos seleccionar una ciudad y pulsar el botón *Comprar Líder*. Esto hará que se cree un nuevo líder en esa ciudad, siempre y cuando no existiese ya un líder dentro.

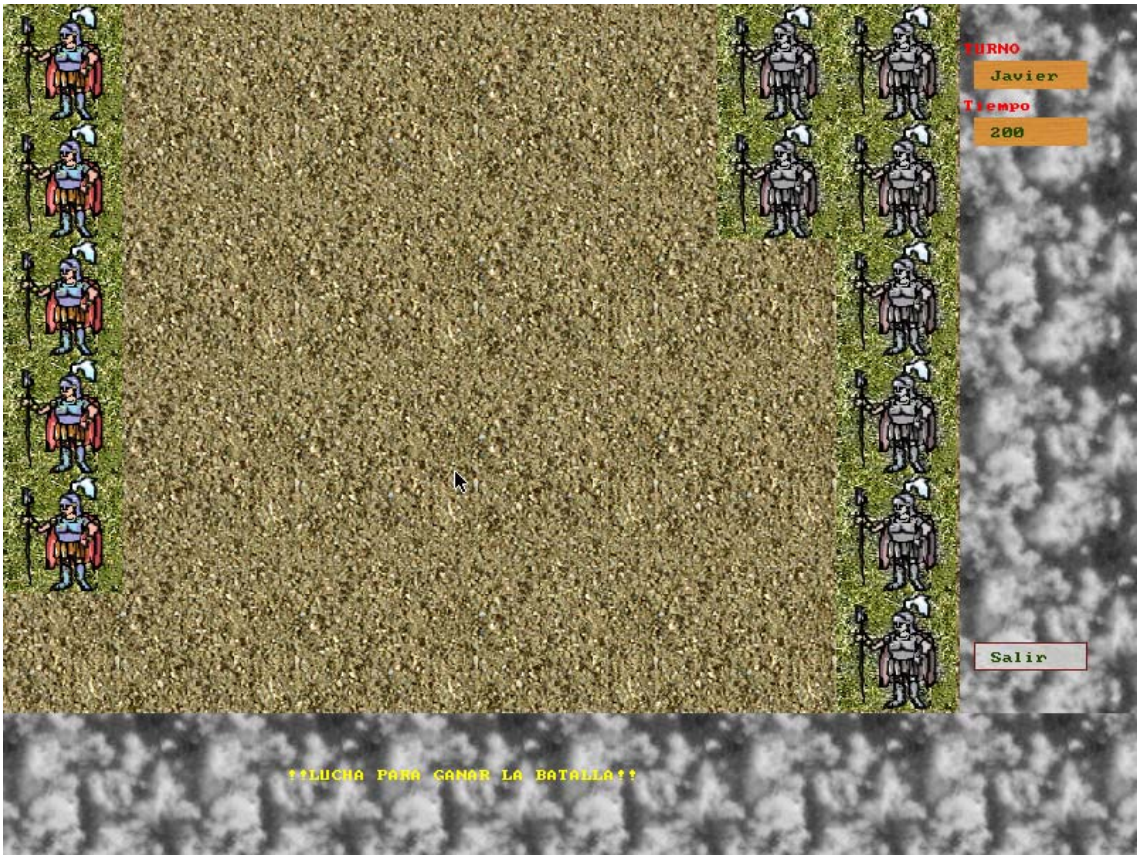
Comprar un soldado para un líder. En caso de que tengamos soldados disponibles, podemos seleccionar un soldado (en caso de que esté dentro de una ciudad o recurso, seleccionaremos la ciudad o recurso correspondiente) y pulsar el botón *Comprar Soldado*. Esto hará que se incremente en uno el número de soldados de ese líder. Cuantos más soldados tenga un líder, más posibilidades tendremos de ganar una batalla contra un líder rival.

Una vez que hemos definido cada botón y cada acción, pasamos a explicar las reglas del juego.

- Los líderes se pueden mover por todo el mapa, y cada uno siempre irá con el número de soldados de que disponga. Si hacemos que un soldado entre en una ciudad o recurso, pueden ocurrir dos cosas:
    - Que no haya ningún líder rival en el interior, en cuyo caso la ciudad o recurso pasará a ser nuestro automáticamente.
    - Que haya un líder rival en su interior, en cuyo caso se librará una batalla para ver quien gana (este modo de juego se explicará más adelante).
- Cada almacén que tengamos apropiado nos generará una unidad de recurso en cada turno. Cuantos más almacenes tengamos, más rápido incrementaremos nuestra cantidad de recursos y más soldados y líderes podremos comprar. También es bueno conquistar ciudades, ya que en ellas es donde podremos crear líderes.
- Cada soldado cuesta 3 unidades de recurso y cada líder 10, aunque en principio estos valores son modificables.

- Para librar una batalla contra un líder rival tendremos que situarnos en una casilla adyacente a donde este él y desplazarnos encima suyo (da igual que esté en terreno abierto como dentro de una ciudad o almacén).

Una vez que realicemos lo anterior, cambiará el escenario de juego, pasándose a disputar la batalla. El escenario de una batalla tiene el siguiente aspecto:



A la izquierda aparecen los soldados del jugador humano, cuyo número será igual al número de soldados que tenía el líder implicado en la batalla, más uno que es el propio líder. En el lado derecho aparecen los soldados del ordenador de forma similar.

Vemos como aquí también aparece un botón de salir para abandonar la partida, y también se muestra la información del jugador al que le toca el turno y del tiempo de batalla que queda (este tiempo es independiente del tiempo de la partida general).

Aquí las posibles acciones son solo desplazar un soldado a una casilla adyacente a la que se encuentra. Si en la casilla a la que se desplaza se encuentra un soldado rival, se lo 'comerá', haciendo así que desaparezca el soldado contrincante. El objetivo es acabar de esta forma con todos los soldados rivales. Es aquí donde se aprecia lo que comentamos anteriormente de que es importante tener más soldados que el rival, ya que se facilitan bastantes las cosas de esta forma.

Cuando un jugador se queda sin soldados, se vuelve a la pantalla de la partida general, tal y como se dejo, pero en la casilla donde se ha librado la batalla aparece el líder vencedor, con el número de soldados que le hayan quedado 'vivos' tras librar la batalla (menos uno que es el propio líder). En caso de empate, gana el jugador que haya atacado primero.

De vuelta en la partida principal, seguiríamos jugando del mismo modo a como lo hemos hecho hasta ahora, hasta conseguir eliminar al rival.

Una vez acabada la partida, aparecerá un mensaje indicando si hemos ganado o no. Pulsaremos Enter (tal y como se nos indica), y volveremos al Menú Principal, desde donde podremos volver a elegir la opción que queramos.

Cabe señalar aquí que en caso de que el ordenador nos venza guardará la información que ha utilizado para ganarnos y en las siguientes partidas que juguemos contra él usará esa información para aprender de una manera más óptima.

7 Reflexiones y Conclusiones

7.1 ¿Pueden los ordenadores pensar?

El problema de si los ordenadores pueden pensar es un problema que inquieta a los filósofos de grande estima, y en el que nosotros intentaremos dar nuestra visión de manera breve.

Al igual que John Searle en su libro “Mentes, Cerebros y Ciencia”, para resolver el problema, pensamos que es necesario concretar que entendemos por “ordenadores” y qué entendemos por “pensar”.

Por “ordenador” entenderemos computador digital. Un computador digital trata la información con números binarios, y hace cálculos lógicos. Un computador digital ejecuta un programa software, el cual consta de una secuencia determinada de instrucciones. Observamos que el problema no tendría sentido si extendiésemos el significado de “ordenador” a cualquier artilugio creado por el ser humano; ya que en este caso una clonación humana entraría en el concepto de ordenador, al igual que cualquier tipo de imitación exacta del cerebro humano (biológicamente hablando). Obviamente, no sería este el problema que queremos resolver.

La definición de la palabra “pensar” revoca a un problema mucho más antiguo, el “problema mente-cuerpo”. Al intentar ver el significado de “pensar”, vemos que lo primero que tenemos que resolver es si “pensar” es una cualidad del cerebro material del ser humano; o por el contrario si existe una sustancia no material, a la que podríamos llamar “alma”, que es la que realmente piensa. Acerca del problema mente-cuerpo hay muchos libros escritos determinando cantidad de matices. Simplemente, mencionamos que hay dos vertientes en las que se clasifican las diferentes opiniones:

- La vertiente monista, que defiende que sólo hay un tipo de sustancia.
- La vertiente dualista, que dice que hay dos tipos de sustancias: la material (o extensa), y la espiritual (o inextensa o alma).

Dados los grandes avances del área de la Inteligencia Artificial (IA), se han conseguido programas software que consiguen que los computadores puedan desempeñar tareas propias de los humanos, que piensan. De hecho los defensores de la Inteligencia Artificial Fuerte (IA fuerte), comparan el cerebro con el hardware y el alma con el programa software perfecto. De esta manera, opinan que la unión del software perfecto y el hardware apropiado puede pensar en la misma medida que lo hace un ser humano.

Un conocido ejemplo para rebatir la IA fuerte es el de la "habitación china" propuesto por Searle. Imaginemos que conseguimos hacer un programa que pueda mantener una conversación en chino, al igual que lo pudiera hacer un nativo chino. Esto sería equivalente a que nosotros, hablantes españoles, que no conocemos el chino, nos introdujéramos en una habitación, en el que se nos dieran símbolos chinos, y nosotros debiéramos seguir una secuencia de pasos (programa), por el cuál manipuláramos los símbolos chinos de unas cestas a otras, dando como salida una lista de símbolos chinos. Estos símbolos chinos simularían la respuesta de un hablante chino. Sin embargo, nosotros no sabemos chino. Es decir, por muy bien que simulemos ser capaces de hablar chino, no conocemos el significado de los símbolos chinos que hemos usado.

Según esto, hay quien opina que, al igual que en el ejemplo de la "habitación china", el computador trabajará de manera sintáctica con los símbolos, siguiendo un programa; pero el computador jamás conocerá la semántica de los símbolos; ya que la semántica será la que los humanos le asignemos.

Sin embargo existen otras opiniones, como la de A. M. Turing, que defiende la postura contraria. Al igual que los conductistas, piensa que la mejor forma de ver que dos cosas son iguales es ver si se comportan de manera similar. Turing defiende que si en el ejemplo de la "habitación china" el computador es indistinguible con un hablante chino, entonces no podremos afirmar que el computador no sepa hablar chino. De la misma forma, cuando decimos que un hablante chino sabe hablar chino y pensar, nos basamos únicamente en lo que él dice y afirma al exterior. Por tanto, Turing afirma que si el comportamiento del computador es indistinguible con el de un hablante chino, entonces podemos decir que el computador sabe hablar chino, en la misma medida que podemos decir que un hablante chino sabe hablar chino. Además, Turing piensa que el hecho de programar el programa software que se comporte igual que un humano es sólo cuestión de tiempo, dado la gran velocidad del avance de la informática e IA.

Independientemente de si un computador que se comporte de manera similar a un humano se pueda decir que sea una simulación perfecta, o de afirmar que realmente

piense, creemos que hay que valorar el inteligente y preciso trabajo de dichos informáticos.

Nosotros tres tenemos distintas opiniones en relación a todo lo expuesto. Hay quien piensa más en la línea que define Turing, y defiende que en un futuro se conseguirá realizar ordenadores que realmente piensen igual que un ser humano. Mientras que hay quien cree más en la vertiente dualista de Descartes y opina que un ordenador jamás podrá igualar el pensamiento de un ser humano. Esta es probablemente una de las características que ha contribuido al resultado final de nuestro proyecto (excelente en nuestra opinión), ya que veíamos posibles soluciones desde puntos de vista muy distintos unos de otros, y el abanico de ideas era mucho más amplio.

Concluimos diciendo que, dado el gran interés que ha despertado en nosotros el hecho de conseguir que un programa “piense”, hemos llevado a cabo nuestro proyecto, que ha intentado, en todo momento, simular el pensamiento y el aprendizaje propio de los humanos.

8. División del trabajo entre miembros del grupo

En nuestro caso, hemos llevado la mayor parte del trabajo de modo conjunto, reuniéndonos habitualmente algún día de la semana en concreto, aunque siempre con un horario relativamente flexible según las necesidades y tiempo de cada uno.

Casi todos hemos tenido por necesidades en algún momento que releer todo el programa para entender la parte de los demás, y así poder trabajar mejorando esas partes.

Específicamente, Iván ha centrado más su trabajo en la parte evolutiva inicialmente, llevando a cabo también buena parte de la IA. Posteriormente, ha tenido que involucrarse también en el desarrollo de código lógico a medida que el diseño se complicó. A él además se deben algunas de las ideas que han sido posteriormente pilares en nuestro programa, como la separación entre el modo batalla y el general.

Javier Laureano ha sido desde un primer momento del proyecto “el ingeniero gráfico”, el encargado de tratar desde nuestro programa con el usuario y el que era encargado de dotar de parte visual a nuestra aplicación. Por otro lado, y al haber trabajado bastante en el código del programa, ha sido quien más se ha peleado con los programas como el Dev-Cpp, que ha requerido tratar numerosos problemas.

Javier Murillo se dedicó inicialmente a la formación del resto del grupo creando un CD con varias aplicaciones. Posteriormente, ha trabajado tocando partes muy diversas del programa, un poco a modo de “comodín”. Pasó de elaborar el diseño de la aplicación en Rational Rose, a la parte gráfica con Javier Laureano creando imágenes, pasando por Sonido, Archivos (aprendizaje), estimadores...

En general, podemos decir que el trabajo que hemos desarrollado ha sido bastante grato y hemos tenido buenas sensaciones trabajando juntos. Creemos que en el propio programa es donde pueden verse los frutos de lo que hemos creado.

9. Referencias y Bibliografía

- [REF 1] http://redcientifica.com/gaia/ce/ceno_c.htm#quees
- [REF 2] http://www.dooyoo.es/archivos-juegos-y-consolas
- [REF 3] http://www.dawnofwargame.com/es
- [REF 4] “Evolving chess playing programs”, R.Grob, K. Albrecht, W.Kantschik, W.Banzhaf
- [REF 5] “Evolving an expert checkers playing program without using human expertise”, K.Chellapilla, David B. Fogel
- [REF 6] “Machine learning using a genetic algorithm to optimise a draughts program board evaluation function”, Kenneth J.Chisholm and Peter V.G.Bradbeer
- Libros también utilizados :
    - "Genetic Algotihms + Data Structures = Evolution Programs ", Michalewicz.
    - "Inteligencia Artificial" Russell & Nurving

Autorización

Autorizamos a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Los abajo firmantes,

Javier Murillo Blanco

Javier Laureano Collado

Iván García-Magariño García