ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERÍA INFORMÁTICA

JUEGOS DE ACORRALAMIENTO

Realizado por
ALBERTO ALEJANDRO RODRÍGUEZ

Dirigido por
JOSÉ RAMÓN PORTILLO FERNÁNDEZ

Departamento
MATEMÁTICA APLICADA I

Sevilla, Septiembre de 2010

Índice

1. Introducción

La *inteligencia artificial* es el nombre que se da a la disciplina informática dedicada a replicar la inteligencia humana en un dispositivo artificial.

La posibilidad de desarrollar un dispositivo capaz de percibir su entorno, procesar tales percepciones y actuar racionalmente en consecuencia, ha despertado la curiosidad del ser humano desde la antigüedad; sin embargo, no fue hasta la segunda mitad del siglo XX cuando esa posibilidad se materializó en algo tangible, gracias al influyente trabajo del matemático británico Alan M. Turing (Figura 1).

En 1950, Turing publicó un artículo en la revista *Mind*, titulado "Computing Machinery and Intelligence", en el que reflexionaba sobre el concepto de inteligencia artificial y establecía lo que luego se conocería como su famosa Prueba de Turing; que permitiría corroborar la existencia de inteligencia en una máquina, fundamentada en la hipótesis de que si una máquina se comporta en todos los aspectos como inteligente, entonces debe ser inteligente.

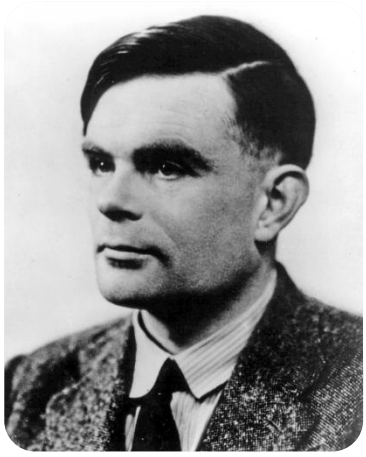


Figura 1: Alan M. Turing

En 1956, el término "inteligencia artificial" (IA) fue acuñado por John McCarthy, del Instituto de Tecnología de Massachusetts. La IA como ciencia es muy reciente; en ese año se celebró la conferencia de Dartmouth, en Hanover (Estados Unidos), donde se establecieron las bases de la inteligencia artificial como un campo independiente dentro de la informática.

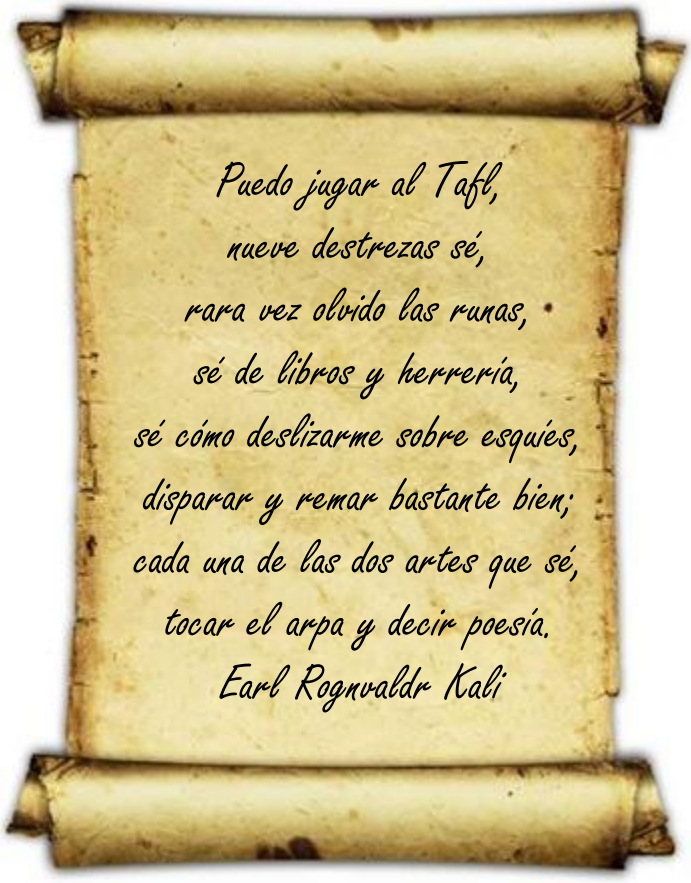
En la actualidad la IA abarca una gran variedad de subcampos, que van desde áreas de propósito general, como el aprendizaje y la percepción, a otras más específicas como la demostración de teoremas matemáticos, la escritura de poesía, el diagnóstico de enfermedades o la toma de decisiones óptimas en juegos, siendo este último el tema que nos compete en este proyecto.

El jugar a juegos fue una de las primeras tareas emprendidas en IA. Hacia 1950, casi tan pronto como los computadores se hicieron programables, el ajedrez fue abordado por Konrad Zuse (Figura 2), el inventor del primer computador programable y del primer lenguaje de programación; por Claude Shannon, el inventor de la teoría de la información; por Norbert Wiener, el creador de la teoría de control moderna; y por Alan Turing.

Figura 2: Konrad Zuse

Desde entonces hubo un progreso continuo en el nivel de juego, hasta tal punto que las máquinas han superado a las personas en las damas y en Otelo, han derrotado a campeones humanos (aunque no siempre) en ajedrez y backgammon, y son competitivos en otros muchos juegos. La excepción principal es el Go, donde los computadores funcionan a nivel aficionado.

En el presente proyecto, estudiaremos una clase específica de juegos llamada juegos de suma cero, de dos jugadores, por turnos, deterministas y de información perfecta (totalmente observables). Más concretamente se estudian tres juegos de la familia Tafl: *"El ratón y los gatos"*, *"La liebre y los perros"*, y *"El zorro y los gansos"*.



Estas son las habilidades del noble de la Edad Vikinga Escandinava. Antes de la introducción del Ajedrez (en noruego antiguo Skak-Tafl) en los siglos XI y XII, los escandinavos aguzaron su ingenio jugando un juego conocido como Tafl.

Tafl en noruego antiguo significa "tablero" y al final del período se refiere a una variedad de juegos de tablero, como el Ajedrez, Tabula (el ancestro medieval del Backgammon, Kvatru-Tafl), el Zorro y los Gansos (Hala-Tafl o Freys-Tafl), la Danza de los Tres Hombres (Hræ_-Tafl o "Tafl-rápido") y de los Nueve Hombres. Sin embargo, el término Tafl era el más usado comúnmente para referirse a un juego conocido como Hnefa-Tafl o "Tablero del Rey". Hnefatafl era conocido en Escandinavia antes del 400 d.C. y fue llevado por los vikingos a Groenlandia, Islandia, Irlanda, Gran Bretaña, Gales y hasta países del lejano oriente como Ucrania.

Los sajones desarrollaron su propia variante, derivada de un juego alemán de Tafl. Un texto latino escrito durante el reinado del Rey Athelstan (925-940) describe la forma sajona del Hnefatafl que se jugaba en Inglaterra en esa época. Los sajones jugaban así como otros europeos del norte en un tablero del mismo tamaño y que se menciona en las sagas islandesas (Mabinogion y en el Glosario de Cormac) al comienzo del siglo XIV. De hecho fue el juego más popular de la Europa del norte durante la Edad Media. Tan sólo, la aparición del Ajedrez, en el siglo XI, hizo declinar su popularidad, a pesar de lo cual, a comienzos del siglo XIV es frecuente encontrar referencias a él en muchas sagas islandesas. Las últimas referencias de haberse jugado están en Gales en 1587 y de Laponia en 1723.

Los juegos de la familia Tafl se distinguen por el tamaño desigual de las fuerzas opuestas, ya que reproducen una batalla entre dos fuerzas desiguales en número y potencia. Esa desigualdad de fuerzas entre los contrincantes constituye el rasgo definitorio de esta familia de juegos. Cada uno de los jugadores persigue objetivos diferentes: mientras que el objetivo habitual para la fuerza de menor número es llegar hasta una determinada posición en el tablero o tomar a todos los miembros de la fuerza mayor, para estos, su propósito es generalmente inmovilizar al oponente y que no pueda lograr su objetivo. Más adelante se detallarán las características que definen a cada uno de los juegos elegidos para este proyecto.

Una vez conocido el marco y el boceto del proyecto, podemos permitirnos volver al campo de la aplicación de la inteligencia artificial para conocer las herramientas que vamos a utilizar en el proyecto.

Hoy en día una buena interfaz es un requisito imprescindible para que un programa tenga éxito fuera del ámbito científico o puramente técnico. Pero además, para que el desarrollo del programa sea asumible por el programador, es preciso que el tiempo invertido en crear esa interfaz no sea excesivo. Por otra parte, debemos elegir un lenguaje de programación que nos permita explotar todo el potencial del proyecto y a su vez poder nosotros hacer uso de toda la funcionalidad que nos ofrece dicho lenguaje.

Así, la codificación se realizará en Lisp, lenguaje de programación por antonomasia de Inteligencia Artificial. Desde su especificación en 1958 hasta la actualidad, la comunidad de investigación de la inteligencia artificial y Lisp han estado estrechamente ligados, debido a que Lisp ofrece características especialmente diseñadas para manejar problemas generalmente encontrados en Inteligencia Artificial.

Y como consecuencia, para crear el *Front-end* usaremos LTK, que es la librería por excelencia para interfaces gráficas de Lisp. Es una creación de Peter Herth. Y de forma somera consiste en una tubería que transforma nuestro código lisp en código tcl que es ejecutado por el programa *wish*.

Para finalizar, es importante destacar la motivación que me ha llevado a realizar un proyecto como este, y es que la investigación de los juegos, a diferencia de otros problemas, supone un reto muy interesante porque los juegos son demasiado difíciles para poder ser resueltos. Notemos a modo de esbozo que la resolución de los juegos se produciría si pudiésemos generar un árbol de búsqueda completo, esto es, crear todos los estados posibles sucesivos al estado inicial, producidos a partir de los movimientos permitidos alternativamente de los dos jugadores. La dificultad radica en que la profundidad y el factor de ramificación de un árbol de búsqueda hacen que sea inviable obtener el árbol de movimientos *completo* en un tiempo razonable.

Por lo tanto, los juegos, como el mundo real, requieren la capacidad de tomar alguna decisión cuando no es factible calcular la decisión óptima. Así, el desafío que presenta este proyecto es como hacer uso, lo mejor posible, del tiempo.

2. Definición de objetivos

Numerosas culturas han diseñado diferentes juegos de acorralamiento a lo largo de la historia. El objetivo general de este proyecto es realizar una aplicación que juegue a varios de estos juegos de tablero de forma *inteligente*.

Pero este es el objetivo final y a más largo plazo, si descendemos en la pirámide de la abstracción (Figura 3) y destacamos los detalles relevantes del proyecto en estudio, podemos hablar de los siguientes objetivos específicos:

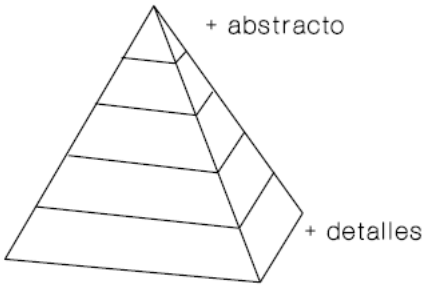


Figura 3: Pirámide de la abstracción

- Profundizar en la representación de problemas como espacio de estados.
- Profundizar en el conocimiento del algoritmo minimax con poda alfa-beta para la implementación de juegos de estrategia.
- Apreciar el papel de las funciones de evaluación estática como incorporación de información a las búsquedas, permitiendo mejorar la eficiencia práctica en la generación de árboles de juego.
- Encontrar el equilibrio óptimo entre la profundidad del árbol de búsqueda explorado y la complejidad de la función de evaluación estática, impuesto por el tiempo limitado del que se dispone para la toma de decisiones.
- Profundizar en el lenguaje de programación Lisp, con el que implementar los algoritmos de inteligencia artificial.
- Conocer la librería LTK, con la que crear interfaces gráficas en Lisp.
- Aplicar e integrar los conocimientos adquiridos, teóricos y prácticos, durante la carrera de Ingeniería Informática, así como la creatividad y originalidad, a un proyecto personal de complejidad elevada.

3. Análisis de antecedentes

La temprana historia de los juegos mecánicos se estropeó por numerosos fraudes. El más célebre de estos fue "El Turco" de Baron Wolfgang von Kempelen (1734-1804), un supuesto autómata que jugaba al ajedrez, que derrotó a Napoleón antes de ser expuesto como la caja de bromas de un mago que escondía a un humano experto en ajedrez. Jugó desde 1769 hasta 1854. En 1846, Charles Babbage (quien había sido fascinado por el Turco) parece haber contribuido a la primera discusión seria de la viabilidad del computador de ajedrez y de damas. Él también diseño, pero no construyó, una máquina con destino especial para jugar al tres en raya. La primera máquina real de juegos fue construida alrededor de 1890 por el ingeniero español Leonardo Torres y Quevedo. Se especializó en el "RTR" (rey y torre contra el rey), la fase final de ajedrez, garantizando un triunfo con el rey y torre desde cualquier posición.

El algoritmo minimax se remonta a un trabajo publicado en 1912 de Ernst Zermelo, el que desarrolló la teoría moderna de conjuntos. El trabajo, lamentablemente, tenía varios errores y no describió minimax correctamente. Un fundamento sólido, para la teoría de juegos, fue desarrollado en el trabajo seminal de Theory of Games and Economic Behaviour (von Neumann y Morgenstern, 1944), que incluyó un análisis en el que mostraba que algunos juegos requieren estrategias aleatorizadas (o imprevisibles).

Muchas figuras influyentes de los comienzos de los computadores, quedaron intrigadas por la posibilidad de jugar al ajedrez con un computador. Konrad Zuse (1945), la primera persona que diseñó un computador programable, desarrolló ideas bastante detalladas sobre cómo se podría hacer esto. El conocido libro de Norbert Wiener (1948), Cybernetics, habló de un diseño posible para un programa de ajedrez, incluso de las ideas de búsqueda minimax, límites de profundidad, y funciones de evaluación. Claude Shannon (1950) presentó los principios básicos de programas modernos de juegos con mucho más detalle que Wiener. Él introdujo la idea de la búsqueda de estabilidad y describió algunas ideas para la búsqueda del árbol de juegos selectiva (no exhaustiva). Slater (1950) y los que comentaron su artículo también exploraron las posibilidades para el juego de ajedrez por computador. En particular, I. J. Good (1950) desarrolló la noción de estabilidad independientemente de Shannon.

En 1951, Alan Turing escribió el primer programa de computador capaz de jugar una partida completa de ajedrez. Pero el programa de Turing nunca se ejecutó sobre un computador; fue probado por simulación a mano contra un jugador muy débil humano, que lo derrotó. Mientras tanto D. G. Prinz (1952) escribió, y realmente ejecutó, un programa que resolvió problemas de ajedrez, aunque no jugara un juego completo. Alex Bernstein escribió el primer programa para jugar un juego completo de ajedrez estándar.

John McCarthy concibió la idea de la búsqueda alfa-beta en 1956, aunque él no lo publicara. El programa NSS de ajedrez (Newell, 1958) usó una versión simplificada de alfa-beta; y fue el primer programa de ajedrez en hacerlo así. Según Nilsson (1971), el programa de damas de Arthur Samuel (Samuel, 1959,1967) también usó alfa-beta, aunque Samuel no lo mencionara en los informes publicados sobre el sistema. Los trabajos que describen alfa-beta fueron publicados a principios de 1960 (Hart y Edwards, 1961 ; Brudno, 1963; Slagle, 1963b). Una implementación completa de alfa-beta está descrita por Slagle y Dixon (1969) en un programa para juegos de Malah. Alfa-beta fue también utilizada por el programa "Kotok-McCarthy" de ajedrez escrito por un estudiante de John McCarthy (Kotok, 1962). Knuth y Moore (1975) proporcionan una historia de alfa-beta, junto con una demostración de su exactitud y un análisis de complejidad en tiempo. Su análisis de alfa-beta con un orden de sucesores aleatorio mostró una complejidad asintótica de $O((b/\log b)^d)$, que pareció bastante triste porque el factor de ramificación efectivo $b/\log b$ no es mucho menor que b. Ellos, entonces, se dieron cuenta que la fórmula asintótica es exacta sólo para $b > 1000$ más o menos, mientras que a menudo se aplica un $O(b^{3d/4})$ a la variedad de factores de ramificación encontrados en los juegos actuales. Pearl (1982b) muestra que alfa-beta es asintóticamente óptima entre todos los algoritmos de búsqueda de árbol de juegos de profundidad fija.

La primera partida de ajedrez de computador presentó al programa Kotok-McCarthy y al programa "ITEP" escrito a mediados de 1960 en el Instituto de Moscú de Física Teórica y Experimental (Adelson-Velsky, 1970). Esta partida intercontinental fue jugada por telégrafo. Se terminó con una victoria 3-1 para el programa ITEP en 1967. El primer programa de ajedrez que compitió con éxito contra humanos fue MacHack 6 (Greenblatt, 1967). Su grado de aproximadamente 1.400 estaba bien sobre el nivel de principiante de 1.000, pero era bajo comparado con el grado 2.800 o más que habría sido necesario para satisfacer la predicción de 1957 de Herb Simon de que un programa de computador sería el campeón mundial de ajedrez en el plazo de 10 años (Simon y Newell, 1958).

Comenzando con el primer Campeonato Norteamericano ACM de Ajedrez de computador en 1970, el concurso entre programas de ajedrez se hizo serio. Los programas a principios de 1970 se hicieron sumamente complicados, con varias clases de trucos para eliminar algunas ramas de búsqueda, para generar movimientos plausibles, etcétera.

En 1974, el primer Campeonato Mundial de Ajedrez de computador fue celebrado en Estocolmo y ganado por Kaissa (Aldeson-Velsky, 1975), otro programa de ITEP. Kaissa utilizó la aproximación mucho más directa de la búsqueda alfa-beta exhaustiva combinada con la búsqueda de estabilidad. El dominio de esta aproximación fue confirmado por la victoria convincente de CHESS 4.6 en el Campeonato Mundial de 1977 de Ajedrez de computador. CHESS 4.6 examinó hasta 400.000 posiciones por movimiento y tenía un grado de 1.900.

Una versión posterior de MacHack, de Greenblatt 6, fue el primer programa de ajedrez ejecutado sobre un *hardware* de encargo diseñado expresamente para el ajedrez (Moussouris, 1979), pero el primer programa en conseguir éxito notable por el uso del *hardware* de encargo fue Belle (Condon y Thompson, 1982). El *hardware* de generación de movimientos y de la evaluación de la posición de Belle, le permitió explorar varios millones de posiciones por movimiento. Belle consiguió un grado de 2.250, y se hizo el primer programa de nivel maestro. El sistema HITECH, también un computador con propósito especial, fue diseñado por el antiguo Campeón de Ajedrez de Correspondencia Mundial Hans Berliner y su estudiante Carl Ebeling en CMU para permitir el cálculo rápido de la función de evaluación (Ebeling, 1987; Berliner y Ebeling, 1989). Generando aproximadamente 10 millones de posiciones por movimiento, HITECH se hizo el campeón norteamericano de computador en 1985 y fue el primer programa en derrotar a un gran maestro humano en 1987. Deep Thought, que fue también desarrollado en CMU, fue más lejos en la dirección de la velocidad pura de búsqueda (Hsu, 1990). Consiguió un grado de 2.551 y fue el precursor de Deep Blue. El Premio de Fredkin, establecido en 1980, ofreció 5.000 dólares al primer programa en conseguir un grado de maestro, 10.000 dólares al primer programa en conseguir un grado FEUA (Federación de los Estados Unidos de Ajedrez) de 2.500 (cerca del nivel del gran maestro), y 100.000 dólares para el primer programa en derrotar al campeón humano mundial. El premio de 5.000 dólares fue reclamado por Belle en 1983, el premio de 10.000 dólares por Deep Thought en 1989, y el premio de 100.000 dólares por Deep Blue por su victoria sobre Garry Kasparov en 1997.

Es importante recordar que el éxito de Deep Blue fue debido a mejoras algorítmicas y de *hardware* (Hsu, 1999; Campbell, 2002). Las técnicas como la heurística de movimiento nulo (Beal, 1990) han conducido a programas que son completamente selectivos en sus búsquedas. Los tres últimos Campeonatos Mundiales de Ajedrez de computador en 1992, 1995 y 1999 fueron ganados por programas que se ejecutan sobre computadores personales. Probablemente la mayor parte de la descripción completa de un programa moderno de ajedrez la proporciona Ernst Heinz (2000), cuyo programa DARKTHOUGHT fue el programa de computador no comercial de rango más alto de los campeonatos mundiales de 1999.

La búsqueda alfa-beta es, desde muchos puntos de vista, el análogo de dos jugadores al ramificar y acotar primero en profundidad, dominada por A* en el caso de agente simple. El algoritmo SSS* (Stockman, 1979) puede verse como A* de dos jugadores y nunca expande más nodos que alfa-beta para alcanzar la misma decisión. Las exigencias de memoria y los costos indirectos computacionales de la cola hacen que SSS* sea poco práctico en su forma original, pero se ha desarrollado una versión de espacio-lineal a partir del algoritmo RBFS (Korf y chickering, 1996). Plaat (1996) desarrollaron una nueva visión de SSS* como una combinación de alfa-beta y tablas de transposiciones, mostrando cómo vencer los inconvenientes del algoritmo original y desarrollando una nueva variante llamada MTD(*f*) que ha sido adoptada por varios programas superiores.

D.F. Beal (1980) y Dana Nau (1980,1983) estudiaron las debilidades de minimax aplicado a la aproximación de las evaluaciones. Ellos mostraron que bajo ciertos axiomas de independencia sobre las distribuciones de los valores de las hojas, minimaximizar puede producir valores en la raíz que son realmente *menos* fiables que el uso directo de la función de evaluación. El libro de Pearl, *Heuristics* (1984), explica parcialmente esta paradoja aparente y analiza muchos algoritmos de juegos. Baum y Smith (1997) proponen una sustitución a base de probabilidad para minimax, y muestra que esto causa mejores opciones en ciertos juegos. Hay todavía poca teoría sobre los efectos de cortar la búsqueda en niveles diferentes y aplicar funciones de evaluación.

El algoritmo minimax esperado fue propuesto por Donald Michie (1966), aunque por supuesto sigue directamente los principios de evaluación de los árboles de juegos debido a von Neumann y Morgenstern. Bruce Ballard (1983) amplió la poda alfa-beta para cubrir árboles de nodos de posibilidad.

El primer programa de *backgammon* fue BKG (Berliner, 1977, 1980b); utilizó una función de evaluación compleja construida a mano y buscó solo a profundidad 1. Fue el primer programa que derrotó a un campeón mundial humano en un juego clásico importante (Berliner, 1980a). Berliner reconoció que éste fue un partido de exhibición muy corto (no fue un partido del campeonato mundial) y que BKG tuvo mucha suerte con los dados. El trabajo que Gerry Tesauro, primero sobre NEUROGAMMON (Tesauro, 1989) y más tarde sobre TD-GAMMON (Tesauro, 1995), mostró que se pueden obtener muchos mejores resultados mediante el aprendizaje por refuerzo.

Las damas, más que el ajedrez, fue el primer juego clásico jugado completamente por un computador. Christopher Strachey (1952) escribió el primer programa de funcionamiento para las damas. Schaeffer (1997) dio una muy legible, "con todas sus imperfecciones", cuenta del desarrollo de su programa de damas campeón del mundo Chinook.

Los primeros programas de Go fueron desarrollados algo más tarde que los de las damas y el ajedrez (Lefkovitz, 1960; Remus, 1962) y han progresado más despacio. Ryder (1971) usó una aproximación basada en la búsqueda pura con una variedad de métodos de poda selectivos para vencer el enorme factor de ramificación. Zobrist (1970) usó las reglas condición-acción para sugerir movimientos plausibles cuando aparecieran los modelos conocidos. Reitman y wilcox (1979) combinan reglas y búsqueda con efectos buenos, y los programas más modernos han seguido esta aproximación híbrida. Müller (2002) resume el estado del arte de la informatización de Go y proporciona una gran variedad de referencias.

Los trabajos sobre juegos de computador aparecen en multitud de sitios. La mal llamada conferencia *Heuristic Programming in Artificial Intelligence* hizo un informe sobre las Olimpiadas de Computador, que incluyen una amplia variedad de juegos. Hay también varias colecciones de trabajos importantes sobre la investigación en juegos (Levy, 1988a, 1988b; Marsland y Schaeffer, 1990). La Asociación Internacional de Ajedrez por Computador (ICCA), fundada en 1977, publica la *revista* trimestral *ICGA* (anteriormente la revista *ICCA*). Los trabajos importantes han sido publicados en la serie antológica *Advances in Computer Chess*, que comienza con Clarke (1977). El volumen 134 de la revista *Artificial Intelligence* (2002) contiene descripciones de programas para el ajedrez, Otelo, Hex, shogi, Go, *backgammon*, poker, Scrabble™ y otros juegos.

4. Descripción del proyecto

La teoría de juegos es un área de la matemática aplicada que utiliza modelos para estudiar interacciones en estructuras formalizadas de incentivos (los llamados *juegos*) y llevar a cabo procesos de decisión. Sus investigadores estudian las estrategias óptimas así como el comportamiento previsto y observado de individuos en juegos.

Aunque desarrollada en sus comienzos como una herramienta para entender el comportamiento de la economía, la teoría de juegos tiene la característica de ser un área en que la sustancia subyacente es principalmente una categoría de matemáticas aplicadas. Sin embargo, el ámbito de uso de la teoría de juegos es muy amplio y es investigado por especialistas en otras áreas; entre las cuales cabe destacar las ciencias económicas, la biología evolutiva, la psicología, las ciencias políticas, la investigación operativa, la informática y la estrategia militar.

Su objetivo es el análisis de los comportamientos estratégicos de los jugadores. En el mundo real, tanto en las relaciones económicas como en las políticas o sociales, son muy frecuentes las situaciones en las que, al igual que en los juegos, su resultado depende de la conjunción de decisiones de diferentes agentes o jugadores. Se dice de un comportamiento que es estratégico cuando se adopta teniendo en cuenta la influencia conjunta sobre el resultado propio y ajeno de las decisiones propias y ajenas.

La Teoría de juegos experimentó un crecimiento sustancial y se formalizó por primera vez a partir de los trabajos de John von Neumann y Oskar Morgenstern en su libro *The Theory of Games Behavior*, publicado en 1944.

Von Neumann y Morgenstern investigaron el planteamiento estratégico o no cooperativo de la Teoría de Juegos. Este planteamiento requiere especificar detalladamente lo que los jugadores pueden y no pueden hacer durante el juego, y después buscar cada jugador una estrategia óptima. Lo que es mejor para un jugador depende de lo que los otros jugadores piensan hacer, y esto a su vez depende de lo que ellos piensan del primer jugador hará.

Resolvieron este problema en el caso particular de juegos con dos jugadores cuyos intereses son diametralmente opuestos. A estos juegos se les llama estrictamente competitivos, o de suma cero, porque cualquier ganancia para un jugador siempre se equilibra exactamente por una pérdida correspondiente para el otro jugador.

Al igual que Neumann y Morgenstern, en este proyecto estudiaremos y daremos una implementación a tres ejemplos de juegos de suma cero: *"El ratón y los gatos"*, *"La liebre y los perros"*, y *"El zorro y los gansos"*.

Además de ser juegos de suma cero, los tres anteriormente citados pertenecen a una familia de juegos llamada Tafl, que les proporcionan características comunes y muy representativas:

- De tablero.
- Para dos jugadores.
- Totalmente observables.
- Deterministas.
- Tamaño desigual de las fuerzas opuestas.
- El objetivo de la fuerza menor suele ser llegar a una posición determinada en el tablero o capturar un número suficiente miembros de la fuerza mayor.
- El objetivo de la fuerza mayor generalmente es conseguir con sus piezas una posición en el tablero que le permita inmovilizar a la fuerza menor bloqueando todos los movimientos válidos de éste.

Figura 4: El juego original Tafl

4.1 El ratón y los gatos

Introducción

El juego el ratón y los gatos, también conocido como el zorro y los perros, es un antiguo juego de tablero probablemente originario de Escandinavia.

Debido a la simpleza de sus reglas y la familiaridad que nos proporciona el tablero clásico de ajedrez donde se juega; resulta un juego ideal para iniciar el estudio de los juegos de acorralamiento.

Es importante mencionar el hecho de que Elwyn R. Berlekamp, John H. Conway, y Richard K. Guy *en el libro Winning Ways for your Mathematical Plays* (Academic Press, 1982); demostraron que si se juega una partida perfecta, los gatos resultarían vencedores. Sin embargo, esta afirmación no supone problema alguno, ya que nuestro objetivo no es *resolver* el problema, sino conseguir dotar a la computadora de un sistema de toma de decisiones subóptimas en un tiempo determinado y razonablemente bajo: unos segundos.

Cómo Jugar

Figura 5: El juego el ratón y los gatos, con las piezas preparadas para comenzar el juego.

Comenzando el juego

1. El ratón y los gatos es un juego para dos personas que se juega en un tablero tradicional de ajedrez; y cinco fichas: cuatro negras, que figuran los gatos, y una blanca, que representa al ratón.
2. Uno de los jugadores toma el papel de los cuatro gatos, mientras que el otro jugador hace lo propio con el ratón. La disposición inicial de las piezas se muestra en la figura 5.
3. El ratón comienza el juego realizando un movimiento; y a continuación juegan los gatos moviendo una de sus piezas; se juega alternando los turnos de juego hasta el final de la partida.

Moviendo las piezas

4. Una casilla sólo puede contener una ficha en cada momento.
5. Solo se permiten movimientos diagonales, tanto para los gatos como para el ratón.
6. Un gato puede moverse una vez a cualquiera de las dos casillas diagonales de avance (que lo desplace arriba de su posición original) siempre que no se encuentre ya ocupada. No puede moverse diagonalmente hacia atrás a una posición que lo acerque al borde del tablero donde comenzó el juego.
7. El ratón puede moverse una vez en cualquier dirección diagonal.
8. No existen saltos ni capturas en este juego.

Fin del juego

9. El ratón gana si consigue evitar a los gatos y llegar al extremo contrario del tablero de donde comenzó la partida.
10. Los gatos ganan atrapando al ratón; esto se consigue si la liebre no tiene opción de movimiento en su turno.

4.2 La liebre y los perros

Introducción

El juego de *la liebre y los perros*, también conocido como el *juego militar francés*, es el más pequeño y simple de todos los juegos de acorralamiento. Es originario del siglo XIX en Francia, y llegó a ser muy popular entre los militares franceses durante la guerra Franco-Prusiana de 1870-1871.

En 1963, un artículo de Martin Gardner en la revista *Scientific American* generó mucho interés en el juego, y los programadores de IA empezaron a estudiarlo debido a la facilidad de implementación de sus simples reglas.

Cómo Jugar

Existen dos variantes conocidas, diferenciadas únicamente por la posición inicial de la liebre.

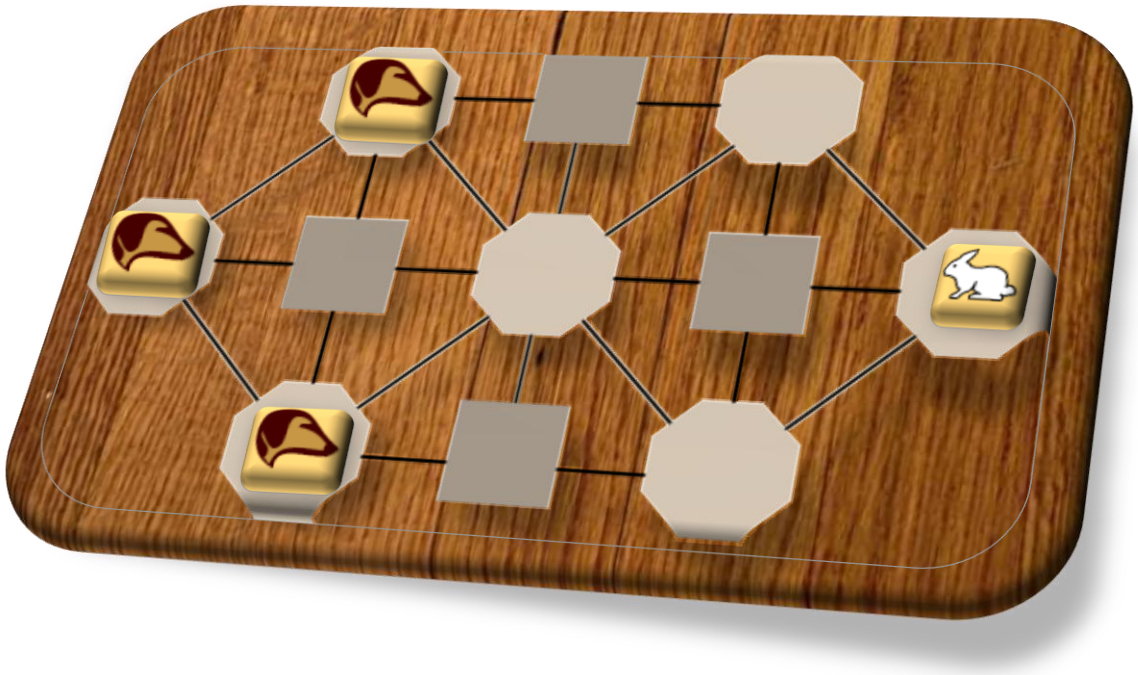


Figura 6: El juego la liebre y los perros, con las piezas preparadas para comenzar el juego.

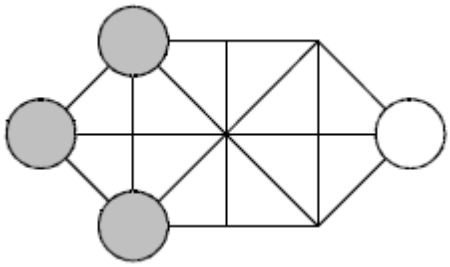


Figura 7: Una disposición inicial alternativa.

Comenzando el juego

1. La liebre y los perros es un juego para dos personas que se juega en un tablero como el mostrado en la figura 6.
2. Uno de los jugadores toma el papel de los tres perros, mientras que el otro jugador hace lo propio con la liebre. La disposición inicial de las piezas se muestra en la figura 6, o alternativamente en la figura 7.
3. Los perros comienzan el juego moviendo una de sus piezas; y a continuación mueve la liebre; se juega alternando los turnos de juego hasta el final de la partida.

Moviendo las piezas

4. Una casilla sólo puede contener una ficha en cada momento.
5. Un perro puede moverse una vez sobre una de las líneas marcadas en cualquier dirección de avance (que lo desplace a la derecha de su posición original) o dirección lateral (que lo desplace arriba o debajo de su posición original). No puede moverse hacia atrás ni diagonalmente a una posición que lo acerque a la zona del tablero donde comenzó el juego.
6. La liebre puede moverse una vez en cualquier dirección sobre una línea marcada.
7. No existen saltos ni capturas en este juego.

Fin del juego

8. La liebre gana si consigue evitar a los perros y llegar al extremo contrario del tablero de donde comenzó la partida.
9. Los perros ganan atrapando a la liebre; esto se consigue si la liebre no tiene opción de movimiento en su turno.
10. Los perros no pueden estar sin hacer ningún movimiento de avance durante diez turnos, si esto sucediera, se considera que los perros se han estancado y la liebre gana la partida.

4.3 El zorro y los gansos

Introducción

El zorro y los gansos es, con toda probabilidad, uno de los más típicos juegos de tablero en cruz. Su práctica, sobre todo en Asia y Europa, se remonta a la antigüedad. Encontramos la primera referencia en la Saga de Gretti, de la literatura islandesa, que data del año 1300. En torno a la misma época, diversos documentos han probado también su existencia en Italia e Inglaterra.

Inicialmente el juego estaba compuesto por un zorro y trece gansos. El depredador podía comerse los gansos, y uno y otras se movían en cualquier dirección. Sin embargo, a partir del siglo XVII el juego fue modificado: el número de gansos aumentó hasta diecisiete, pero sus movimientos en diagonal o hacia atrás fueron invalidados.

Con el paso del tiempo, se han testeado diversas modificaciones de las reglas del juego, intentando conseguir un equilibrio entre los dos bandos, limitando la (a priori) ventaja inicial de los gansos; que curiosamente en este juego, toman el papel del *cazador*. Es interesante destacar que en algunas de estas variantes, las líneas diagonales se han suprimido, y podrían ser jugadas perfectamente en uno de los tableros modernos del juego *Solitario.*

De esta forma se han creado diferentes variantes en multitud de lugares distintos, dando como resultado una rica familia de juegos, todos conocidos bajo el nombre: el zorro y los gansos.

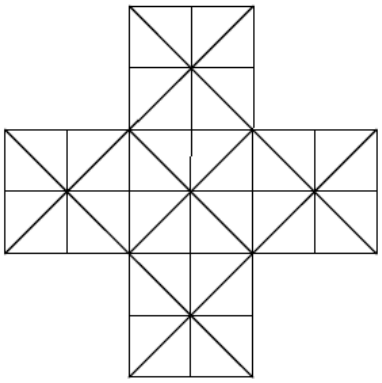


Figura 8: Tablero con treinta y tres casillas, dispuestas en forma de cruz.

Cómo Jugar

Debido a la diversificación que ha sufrido el zorro y los gansos, ninguna de las variantes puede ser considerada como las reglas estándar. Las particularidades de cada una de las variantes tampoco puede decirse que hayan tenido mucho éxito a la hora de balancear el juego. Existen versiones de 12, 13, 16, o 17 gansos; y también algunas que incluyen dos zorros y un mayor número de gansos.

En la versión que aquí se reproduce, existen quince gansos y un solo zorro, ya que la disposición inicial no está especialmente desequilibrada, en pos de una partida divertida y disputada.

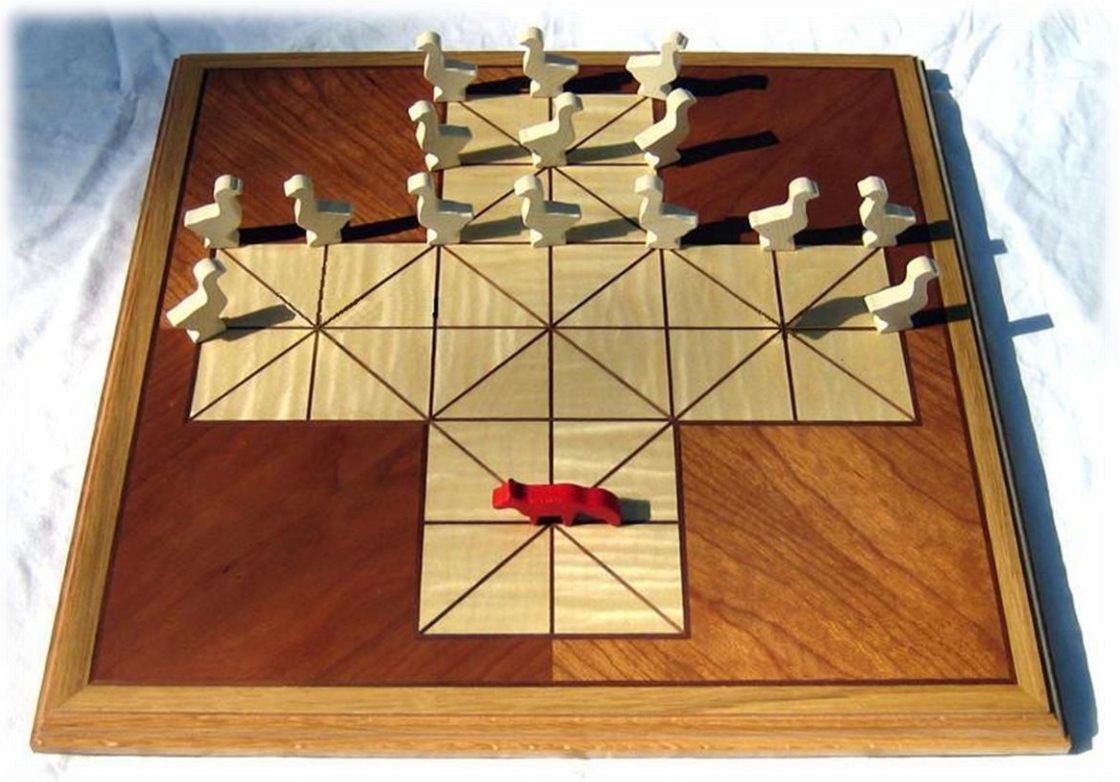


Figura 9: El juego el zorro y los gansos, con las piezas preparadas para comenzar el juego.

Comenzando el juego

1. El zorro y los gansos se juega en un tablero de 33 casillas conectadas por líneas verticales y horizontales, y por líneas diagonales en determinados lugares (ver figura 8).
2. Uno de los jugadores toma el papel de los quince gansos, mientras que el otro jugador hace lo propio con el zorro. La disposición inicial de las piezas se muestra en la figura 9.
3. Los gansos comienzan el juego moviendo una de sus piezas; y a continuación mueve el zorro; se juega alternando los turnos de juego hasta el final de la partida.

Moviendo las piezas

4. Una casilla sólo puede contener una ficha en cada momento.
5. Un ganso puede moverse una vez sobre una de las líneas marcadas en cualquier dirección de avance (que lo acerque al extremo donde comienza el zorro) o dirección lateral (que lo desplace a derecha o izquierda de su posición original). No puede moverse hacia atrás ni diagonalmente a una posición que lo acerque a la zona del tablero donde comenzó el juego.
6. El zorro puede moverse una vez en cualquier dirección sobre una línea marcada.

Capturando los gansos

7. En lugar de avanzar como ya se ha descrito, el zorro puede capturar a un ganso adyacente saltando por encima de él, si la casilla posterior se encuentra vacía y siempre que las casillas estén unidas por una línea. El ganso es entonces retirado del tablero.
8. Si el zorro, justo después de haber saltado, se encuentra en una posición donde es factible realizar un segundo salto, puede realizarlo inmediatamente. No hay límite en el número de gansos consecutivos que pueden ser capturados de esta forma durante el turno del zorro.
9. Los gansos no pueden saltar sobre el zorro.

Fin del juego

10. Los gansos ganan atrapando al zorro; esto se consigue si el zorro no tiene opción de movimiento en su turno.
11. El zorro gana si consigue llegar al extremo contrario del tablero donde comenzó o si captura suficientes gansos para evitar ser inmovilizado según la regla 9 anterior. En teoría, cuatro gansos podrían atrapar al zorro (ver figura 10).
12. Los gansos no pueden estar sin hacer ningún movimiento de avance durante diez turnos, si esto sucediera, se considera que los gansos se han estancado y el zorro gana la partida.

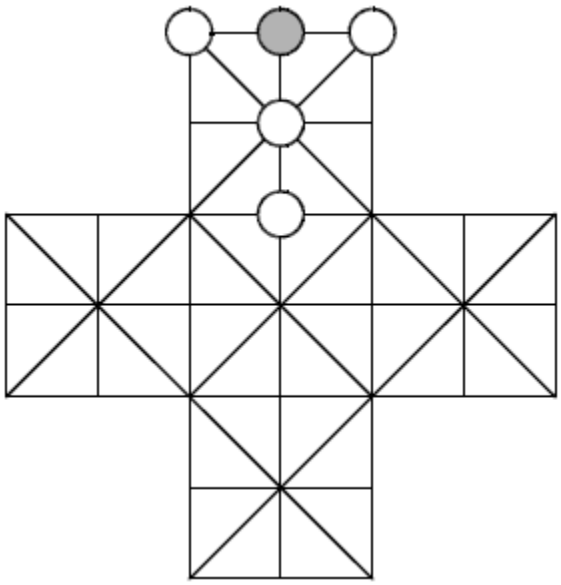


Figura 10: Cuatro gansos han conseguido atrapar al zorro

5. Aportación realizada

Desde el nacimiento de la inteligencia artificial hacia 1950, hasta la actualidad, el jugar a juegos contra un computador ha sido objeto de estudio e interés por numerosos e ilustres personajes expertos en la materia; tanto en el sector de la inteligencia artificial, como en el testeo de esta última.

A este respecto, el presente proyecto podría parecer que no tiene nada nuevo que decir, en el más amplio sentido de la palabra, pero nada más lejos de la realidad: el problema al que nos enfrentamos aporta lo suficiente para ser novedoso e interesante para cualquiera que tenga curiosidad sobre la inteligencia artificial en los juegos de mesa.

A continuación, se exponen los motivos que hacen a este trabajo original e innovador, bajo mi punto de vista, respecto a otros muchos trabajos ya existentes sobre la materia:

- **La elección de Lisp como lenguaje de programación:** Con el nacimiento de los lenguajes de más alto nivel, como C++ o Java, la implementación de aplicaciones de usuario en lenguajes más antiguos como Lisp, cada vez son menos habituales, debido a la facilidad que dan estos nuevos lenguajes de programación y los entornos de desarrollo. Por tanto, este proyecto supone un reto en cuanto a dificultad; y un soplo de aire fresco en cuanto al uso de un lenguaje de programación poco habitual en la actualidad.
- **Las funciones de evaluación estática de los tableros de juego:** Con toda seguridad la parte del proyecto que requiere de un mayor esfuerzo, imaginación y conocimiento, y por ende, también donde podemos encontrar la mayor parte de ideas y aportaciones novedosas.
- **El uso de minimax como estrategia para la toma de decisiones:** Aunque el algoritmo minimax data de 1912, lo considero como una contribución, porque la implementación del algoritmo es totalmente personal, realizada a partir del pseudocódigo del algoritmo minimax con poda alfa-beta. Podría haberse reutilizado código de terceros, y seguramente se hubiese ganado en eficiencia, pero desde un principio tenía claro que uno de los retos que presentaba este proyecto era realizar mi propia implementación del algoritmo minimax.

- **La utilización de LTK para crear interfaces gráficas en Lisp**: Sin ir más lejos, la documentación en castellano sobre cómo crear interfaces gráficas en Lisp es inexistente. Este proyecto, tiene como uno de sus objetivos sentar las bases para la creación de interfaces gráficas con LTK, con la intención de que posteriores lectores se animen a realizarlas sin dificultad. En este sentido, el proyecto si es realmente novedoso, ya que no he podido encontrar ninguna referencia respecto al uso de minimax, lisp y LTK en un mismo proyecto.

- **Los juegos elegidos para implementar:** Existen numerosos antecedentes de programación de inteligencia artificial para juegos de tablero como el ajedrez, backgammon, go, damas, etc. Sin embargo, juegos de acorralamiento con fuerzas desiguales en los dos oponentes como los que se implementan en este proyecto, no tienen estudios anteriores destacables. Apenas se encuentran un par de implementaciones con una inteligencia artificial muy básica o con movimientos predefinidos. Por lo que este proyecto aporta un valioso punto de partida en el estudio de juegos con estas características.

6. Análisis de requisitos

6.1 Objetivo

OBJ-0001	Plataforma de juegos
Versión	1.0 (08/05/2010)
Autores	Alberto Alejandro Rodríguez
Fuentes	José Ramón Portillo Fernández
Descripción	El sistema deberá *ser una plataforma donde se implementen tres juegos con una estrategia de juego común.*
Importancia	Vital
Urgencia	Inmediatamente
Estado	Validado
Estabilidad	Alta
Comentarios	Ninguno

6.2 Participantes

Organización	Departamento de matemática aplicada
Dirección	E. T. S. de Ingeniería Informática Avenida Reina Mercedes s/n 41012 – Sevilla
Teléfono	+34-954552766
Fax	+34-954557878
Comentarios	Ninguno

Participante	José Ramón Portillo Fernández
Organización	Departamento de matemática aplicada
Rol	Cliente
Es desarrollador	No
Es cliente	Sí
Es usuario	Sí
Comentarios	Ninguno

Participante	Alberto Alejandro Rodríguez
Rol	Analista/Programador
Es desarrollador	Sí
Es cliente	No
Es usuario	No
Comentarios	Ninguno

6.3 Requisitos de Usuario

FRQ-0001	**Categoría de los juegos implementados**
Versión	1.0 (08/05/2010)
Autores	Alberto Alejandro Rodríguez
Fuentes	José Ramón Portillo Fernández
Dependencias	Ninguno
Descripción	El sistema deberá *implementar tres juegos de tablero de dos jugadores, por turnos, deterministas y de información perfecta; caracterizados por el tamaño desigual de las fuerzas opuestas y donde el objetivo para la fuerza mayor sea inmovilizar al oponente.*
Importancia	Vital
Urgencia	Inmediatamente
Estado	Validado
Estabilidad	Alta
Comentarios	Ninguno

FRQ-0002	**Independencia entre Front-end y Back-end**
Versión	1.0 (08/05/2010)
Autores	Alberto Alejandro Rodríguez
Fuentes	José Ramón Portillo Fernández
Dependencias	Ninguno
Descripción	El sistema deberá *ser implementado de forma que la dependencia entre la interfaz gráfica y el desarrollo algorítmico a bajo nivel sea la menor posible, con vistas a que en un futuro pueda sustituirse el front-end.*
Importancia	Importante
Urgencia	Inmediatamente
Estado	Validado
Estabilidad	Alta
Comentarios	Ninguno

FRQ-0003	Estrategia común
Versión	1.0 (08/05/2010)
Autores	Alberto Alejandro Rodríguez
Fuentes	José Ramón Portillo Fernández
Dependencias	Ninguno
Descripción	El sistema deberá *funcionar sobre un mecanismo general algorítmico de toma de decisiones cuyo esqueleto sea común en todos los juegos.*
Importancia	Vital
Urgencia	Inmediatamente
Estado	Validado
Estabilidad	Alta
Comentarios	Ninguno

FRQ-0004	Nuevas estrategias
Versión	1.0 (08/05/2010)
Autores	Alberto Alejandro Rodríguez
Fuentes	José Ramón Portillo Fernández
Dependencias	[FRQ-0003] Estrategia común
Descripción	El sistema deberá *ser implementado de forma que sea fácilmente ampliable con nuevas estrategias de juego que se desarrollen con posterioridad.*
Importancia	Vital
Urgencia	Inmediatamente
Estado	Validado
Estabilidad	Alta
Comentarios	Ninguno

FRQ-0005	Comparativa de Estrategias
Versión	1.0 (08/05/2010)
Autores	Alberto Alejandro Rodríguez
Fuentes	José Ramón Portillo Fernández
Dependencias	[FRQ-0004] Nuevas estrategias
Descripción	El sistema deberá *permitir el juego CPU vs CPU y así enfrentar distintas estrategias de juego con objeto de encontrar puntos fuerte y debilidades de cada una de ellas; y en caso de ser posible la óptima.*
Importancia	Vital
Urgencia	Inmediatamente
Estado	Validado
Estabilidad	Alta
Comentarios	Ninguno

FRQ-0006	Niveles de dificultad
Versión	1.0 (08/05/2010)
Autores	Alberto Alejandro Rodríguez
Fuentes	José Ramón Portillo Fernández
Dependencias	Ninguno
Descripción	El sistema deberá *disponer un sistema dual de elección del nivel dificultad seleccionable por el usuario desde la interfaz gráfica y por desarrollador a bajo nivel.*
Importancia	Importante
Urgencia	Inmediatamente
Estado	Validado
Estabilidad	Alta
Comentarios	Con este requisito se persigue que gracias al crecimiento en funcionalidad que el hardware sufre con los años, el nivel de juego pueda adaptarse y mejorarse según el avance de este.

FRQ-0007	Nuevos juegos
Versión	1.0 (08/05/2010)
Autores	Alberto Alejandro Rodríguez
Fuentes	José Ramón Portillo Fernández
Dependencias	[FRQ-0003] Estrategia común
Descripción	El sistema deberá *ser implementado de forma que sea fácilmente ampliable con nuevos juegos desarrollados con posterioridad aprovechando la base de funcionamiento de los juegos originales.*
Importancia	Importante
Urgencia	Inmediatamente
Estado	Validado
Estabilidad	Alta
Comentarios	Ninguno

6.4 Requisitos Técnicos

NFR-0001	Juegos de la familia Tafl
Versión	1.0 (09/05/2010)
Autores	Alberto Alejandro Rodríguez
Fuentes	Alberto Alejandro Rodríguez
Dependencias	• [FRQ-0001] Categoría de los juegos implementados
Descripción	El sistema deberá *dar una implementación a tres juegos de la familia Tafl.* *A saber:* *- El ratón y los gatos* *- La liebre y los perros* *- El zorro y los gansos*
Importancia	Vital
Urgencia	Inmediatamente
Estado	Validado
Estabilidad	Alta
Comentarios	Ninguno

NFR-0002	Leguaje de Programación
Versión	1.0 (09/05/2010)
Autores	Alberto Alejandro Rodríguez
Fuentes	Alberto Alejandro Rodríguez
Dependencias	Ninguno
Descripción	El sistema deberá *utilizar el lenguaje de programación Lisp*
Importancia	Vital
Urgencia	Inmediatamente
Estado	Validado
Estabilidad	Alta
Comentarios	Ninguno

NFR-0003	Biblioteca gráfica
Versión	1.0 (09/05/2010)
Autores	Alberto Alejandro Rodríguez
Fuentes	Alberto Alejandro Rodríguez
Dependencias	• [FRQ-0002] Independencia entre Front-end y Back-end
Descripción	El sistema deberá *utilizar las funciones de la biblioteca LTK para crear la interfaz gráfica de usuario.*
Importancia	Vital
Urgencia	Inmediatamente
Estado	Validado
Estabilidad	Alta
Comentarios	Ninguno

NFR-0004	Estrategia minimax
Versión	1.0 (09/05/2010)
Autores	Alberto Alejandro Rodríguez
Fuentes	Alberto Alejandro Rodríguez
Dependencias	• [FRQ-0003] Estrategia común • [FRQ-0007] Nuevos juegos
Descripción	El sistema deberá *implementar el algoritmo minimax con poda alfa-beta como estrategia general de toma de decisiones.*
Importancia	Vital
Urgencia	Inmediatamente
Estado	Validado
Estabilidad	Alta
Comentarios	Ninguno

NFR-0005	Función de evaluación estática
Versión	1.0 (09/05/2010)
Autores	Alberto Alejandro Rodríguez
Fuentes	Alberto Alejandro Rodríguez
Dependencias	• [FRQ-0004] Nuevas estrategias • [FRQ-0005] Comparativa de Estrategias
Descripción	El sistema deberá *implementar una función llamada f-estatica que nos permita aproximar la utilidad verdadera de un estado sin hacer una búsqueda completa.*
Importancia	Vital
Urgencia	Inmediatamente
Estado	Validado
Estabilidad	Alta
Comentarios	Ninguno

NFR-0006	Parámetro global 'dificultad'
Versión	1.0 (09/05/2010)
Autores	Alberto Alejandro Rodríguez
Fuentes	Alberto Alejandro Rodríguez
Dependencias	• [FRQ-0006] Niveles de dificultad
Descripción	El sistema deberá *definir un parámetro global 'dificultad' de forma que con solo aumentar su valor a bajo nivel, los niveles de de dificultad 'Fácil', 'Medio' y 'Difícil' vean incrementada la IA de la computadora (a costa de un mayor tiempo de elección del movimiento).*
Importancia	Vital
Urgencia	Inmediatamente
Estado	Validado
Estabilidad	Alta
Comentarios	Ninguno

NFR-0007	**Parámetro global 'heurística'**
Versión	1.0 (09/05/2010)
Autores	Alberto Alejandro Rodríguez
Fuentes	Alberto Alejandro Rodríguez
Dependencias	• [FRQ-0004] Nuevas estrategias • [FRQ-0005] Comparativa de Estrategias
Descripción	El sistema deberá *definir un parámetro global 'heurística' que apunte a la estrategia de juego elegida para la partida de forma que nuevas estrategias desarrolladas puedan ponerse en funcionamiento con solo ser referenciadas por este parámetro.*
Importancia	Vital
Urgencia	Inmediatamente
Estado	Validado
Estabilidad	Alta
Comentarios	Ninguno

7. Análisis temporal

Id	Nombre de tarea	Trabajo	Duración	Comienzo	Fin
1	**Gestión del proyecto**	**3 horas**	**2 días**	**jue 01/10/09**	**vie 02/10/09**
2	Elaborar plan de proyecto	2 horas	1 día	jue 01/10/09	jue 01/10/09
3	Elaborar plan de calidad	0 horas	0 días	jue 01/10/09	jue 01/10/09
4	Obtención de compromiso	0 horas	0 días	jue 01/10/09	jue 01/10/09
5	Reunión inicio de proyecto	1 hora	1 día	vie 02/10/09	vie 02/10/09
6	**Proyecto Juegos de acorralamiento**	**465 horas**	**233 días**	**jue 01/10/09**	**lun 23/08/10**
7	**Fase 1: Definición**	**7 horas**	**16 días**	**jue 01/10/09**	**jue 22/10/09**
8	**Análisis de Requisitos**	**5 horas**	**14 días**	**lun 05/10/09**	**jue 22/10/09**
9	Toma de Requisitos de Usuario	2 horas	1 día	lun 05/10/09	lun 05/10/09
10	Validación de Usuario	0 horas	10 días	mar 06/10/09	lun 19/10/09
11	Estimación Global del Proyecto	1 hora	1 día	mar 20/10/09	mar 20/10/09
12	Especificación de Requisitos Técnicos	1 hora	1 día	mié 21/10/09	mié 21/10/09
13	Validación Requisitos Técnicos	0 horas	0 días	mié 21/10/09	mié 21/10/09
14	Registro de Requisitos	1 hora	1 día	jue 22/10/09	jue 22/10/09
15	Revisión fase de requisitos	0 horas	0 días	jue 22/10/09	jue 22/10/09
16	Aprobación de línea base	0 horas	0 días	jue 22/10/09	jue 22/10/09
17	**Planificación**	**2 horas**	**1 día**	**jue 01/10/09**	**jue 01/10/09**
18	Desglose de Tareas	2 horas	1 día	jue 01/10/09	jue 01/10/09
19	Fase Definición Finalizada	0 horas	0 días	jue 01/10/09	jue 01/10/09
20	**Fase 2: Análisis**	**20 horas**	**30 días**	**vie 23/10/09**	**jue 03/12/09**
21	Análisis Funcional	20 horas	30 días	vie 23/10/09	jue 03/12/09
22	Fase de Análisis Finalizada	0 horas	0 días	jue 03/12/09	jue 03/12/09
23	Revisión fase análisis funcional	0 horas	0 días	jue 01/10/09	jue 01/10/09
24	**Fase 3: Diseño**	**205 horas**	**80 días**	**vie 04/12/09**	**jue 25/03/10**
25	Análisis Orgánico El ratón y los gatos	15 horas	10 días	vie 04/12/09	jue 17/12/09
26	Análisis Orgánico La liebre y los perros	20 horas	10 días	vie 18/12/09	jue 31/12/09
27	Análisis Orgánico El zorro y los gansos	30 horas	10 días	vie 01/01/10	jue 14/01/10
28	Análisis Orgánico Front-end	60 horas	20 días	vie 15/01/10	jue 11/02/10
29	Análisis Orgánico Estrategia Minimax	80 horas	30 días	vie 12/02/10	jue 25/03/10
30	Fase de Diseño Finalizada	0 horas	0 días	jue 25/03/10	jue 25/03/10
31	Revisión fase diseño detallado	0 horas	0 días	jue 25/03/10	jue 25/03/10
32	**Fase 4: Construcción/Parametrización**	**153 horas**	**52 días**	**vie 26/03/10**	**lun 07/06/10**
33	El ratón y los gatos	21 horas	7 días	vie 26/03/10	lun 05/04/10
34	La liebre y los perros	12 horas	4 días	mar 06/04/10	vie 09/04/10
35	El zorro y los gansos	40 horas	14 días	lun 12/04/10	jue 29/04/10
36	Front-end	30 horas	10 días	vie 30/04/10	jue 13/05/10
37	Estrategia Minimax	50 horas	17 días	vie 14/05/10	lun 07/06/10

Id	Nombre de tarea	Trabajo	Duración	Comienzo	Fin
38	Fase de Construccion/Parametrización Finalizada	0 horas	0 días	lun 07/06/10	lun 07/06/10
39	Revisión fase construcción	0 horas	0 días	lun 07/06/10	lun 07/06/10
40	**Fase 5: Pruebas**	**70 horas**	**55 días**	**mar 08/06/10**	**lun 23/08/10**
41	Pruebas unitarias	50 horas	40 días	mar 08/06/10	lun 02/08/10
42	Pruebas de Integración	20 horas	15 días	mar 03/08/10	lun 23/08/10
43	Fase de Pruebas Finalizada	0 horas	0 días	lun 23/08/10	lun 23/08/10
44	Revisión fase de pruebas	0 horas	0 días	lun 23/08/10	lun 23/08/10
45	**Fase 6: Implantación**	**0 horas**	**0 días**	**jue 01/10/09**	**jue 01/10/09**
47	Fase de Implantación Finalizada	0 horas	0 días	lun 23/08/10	lun 23/08/10
48	Revisión fase de implantación	0 horas	0 días	lun 23/08/10	lun 23/08/10
49	**Fase 7: Cierre**	**10 horas**	**1 día**	**jue 01/10/09**	**jue 01/10/09**
53	Revisión fase de cierre	0 horas	0 días	lun 23/08/10	lun 23/08/10
54	**Tareas de Gestión y Control de Proyectos**	**108 horas**	**240 días**	**jue 15/10/09**	**mié 15/09/10**
55	**Mecanismos de Seguimiento y Control**	**104 horas**	**240 días**	**jue 15/10/09**	**mié 15/09/10**
56	**Análisis de avance**	**2 horas**	**127 días**	**mié 20/01/10**	**jue 15/07/10**
57	Análisis de avance 1	1 hora	1 día	mié 20/01/10	mié 20/01/10
58	Análisis de avance 2	1 hora	1 día	jue 15/07/10	jue 15/07/10
59	**Plan de Seguimiento**	**2 horas**	**127 días**	**mié 20/01/10**	**jue 15/07/10**
60	Reunión de Seguimiento 1	1 hora	1 día	mié 20/01/10	mié 20/01/10
61	Reunión de Seguimiento 2	1 hora	1 día	jue 15/07/10	jue 15/07/10
62	Gestión del Proyecto	100 horas	240 días	jue 15/10/09	mié 15/09/10
63	**Aseguramiento de Calidad**	**4 horas**	**1 día**	**mié 15/09/10**	**mié 15/09/10**
64	Actividades de Aseguramiento de Calidad (Después	2 horas	1 día	mié 15/09/10	mié 15/09/10
65	Actividades de Aseguramiento de Calidad (Después	2 horas	1 día	mié 15/09/10	mié 15/09/10
66	Finalización del Proyecto	0 horas	0 días	mié 15/09/10	mié 15/09/10

8. Análisis de costes de desarrollo

Para realizar el cómputo de costes de desarrollo del proyecto y con el objetivo de que el resultado sea lo más fiel posible a un proyecto comercial, he tomado los salarios base y el plus por convenio que perciben las distintas categorías profesionales de un Convenio vigente en la actualidad:

CATEGORÍAS	MES (x 14) Euros	AÑO Euros
GRUPO III. TÉCNICOS DE OFICINA		
Analista y Analista de sistemas	1.569,25	21.969,50
Analista-programador/a y Diseñador/a de página Web	1.539,69	21.555,66
Programador/a senior, Jefe/a de operación y Programador/a en Internet	1.103,04	15.442,56
Delineante-proyectista	1.021,11	14.295,54
Programador/a junior, Operador/a ordenador, Programador/a máquina auxiliar, Monitor/a de grabación y Técnico/a mantenimiento página Web	987,69	13.827,66
Delineante	886,70	12.413,80
Administrador/a de test	840,93	11.773,02

PLUS CONVENIO 1.1. / 31.12.2009

CATEGORÍAS	MES (x 14) Euros	AÑO Euros
GRUPO III. TÉCNICOS DE OFICINA		
Analista y Analista de sistemas	109,73	1.536,22
Analista-programador/a y Diseñador/a de página Web	102,72	1.438,08
Programador/a senior, Jefe/a de operación y Programador/a en Internet	77,80	1.089,20
Delineante-proyectista	71,32	998,48
Programador/a junior, Operador/a ordenador, Programador/a máquina auxiliar, Monitor/a de grabación y Técnico/a mantenimiento página Web	69,50	973,00
Delineante	62,39	873,46
Administrador/a de test	59,18	828,52

En una rápida ojeada a la evaluación de tiempos realizada con anterioridad, podemos ver que la duración estimada del proyecto es de 255 días de los cuáles:

1. Durante todo el ciclo de vida del proyecto tendremos contratado un Analista, que en este proyecto realizará las tareas de documentación, toma de requisitos, análisis funcional, diseño, pruebas de analista y gestión del proyecto.

   Supongamos que dicho analista dedicará de media un esfuerzo del 15%, y que el salario es de 1.569,25 + 109,73 (sin contar las pagas extras) mensuales.

   Una rápida regla de tres nos da el coste que nos supondrá tener contratado dicho analista durante todo el período del proyecto: 2.140,70 €.
2. Para la fase de desarrollo realizaremos la contratación de un programador con amplios conocimientos en lenguaje Lisp por lo que le ofrecemos un contrato como programador Senior. El contrato será por obra y servicio durante los 3 meses que se ha estimado que durará la fase de implementación y las pruebas unitarias realizadas por el programador; y el salario a percibir según la tabla adjunta anterior será de 1.103,04 + 77,80. Además, se ha pensado que la persona que se incluya en plantilla también participará en otro proyectos que estén en curso y que el esfuerzo medio dedicado al proyecto de juegos de acorralamiento será del 40%.

   Con un cálculo homólogo al anterior obtenemos que el coste de tener un programador Senior a nuestro servicio para este proyecto es de: 1.464,24 €.

Una de las primeras decisiones que se tomaron tras la asignación del proyecto fue usar software libre para su desarrollo, por lo que el coste de licencias se ve reducido a cero.

Por tanto, el *coste total* que supondría realizar este proyecto dentro del mercado laboral, en una empresa real que esté regulada por el Convenio anterior sería de 3.604,94 €.

9. Diseño

Una vez finalizada la etapa de análisis de requisitos ya debemos tener claro *qué* tiene que hacer nuestro sistema. Ahora, en la fase de diseño vamos a determinar *cómo* hacerlo.

Comencemos por imaginar la estructura que formará el esqueleto de nuestro sistema. Parece claro que podemos dividir el producto software en tres grandes bloques:

1. Interfaz gráfica.
2. Estrategia minimax.
3. Cuerpo de funciones comunes.

Si el proyecto se realizase por un grupo de trabajo en una empresa real, podríamos sin ningún problema asignar cada una de las tareas a un desarrollador diferente.

En este caso, una de las primeras decisiones de diseño tomadas es desechar la idea de utilizar distintos paquetes y archivos fuente, debido a que no estamos trabajando con un lenguaje de alto nivel orientado a objetos, no es el objetivo del proyecto y solo hay un único desarrollador.

Por tanto, el proyecto se realizará sobre un único fichero de texto plano, y con los distintos bloques claramente delimitados.

9.1 Interfaz gráfica

Debido a la naturaleza del proyecto, donde prima crear un código estable y reutilizable por distintos juegos (incluso por juegos que se puedan añadir a posteriori), la búsqueda de decisiones subóptimas, y la decisión de desarrollarlo sobre Lisp en Linux, es evidente que el objetivo no es crear una interfaz deslumbrante, sino sencilla, intuitiva y sobria, donde el usuario final no tenga más que perder 5 segundos antes de ponerse a jugar. Así, la interfaz gráfica tendrá:

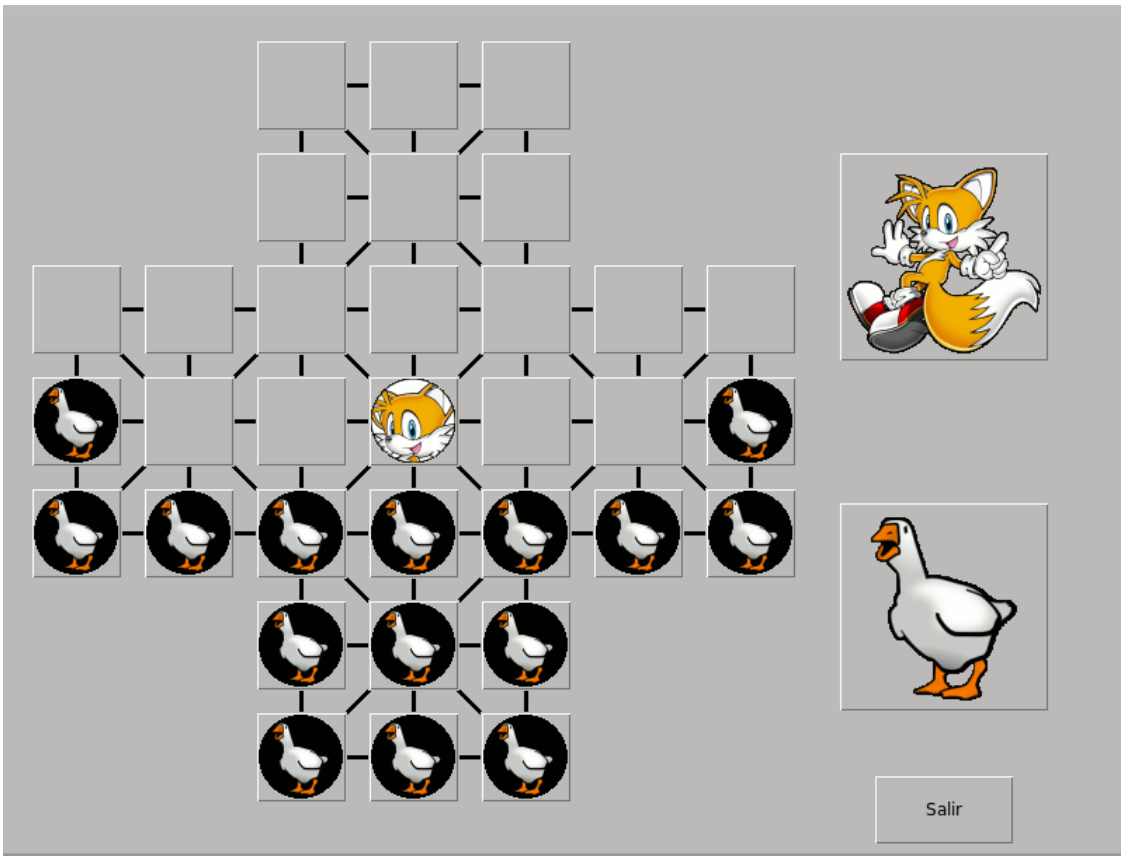
Pantalla de Bienvenida.

Selección de juego.

Selección de modo de juego

Selección de dificultad

Selección de Facción (y vista del tablero de juego)



9.2 Estrategia minimax

La estrategia minimax es sin lugar a dudas la piedra angular del proyecto, la parte más compleja y por ende la que requerirá un mayor esfuerzo de nuestra parte. Es por ello que realizar un proyecto de IA basado en la estrategia minimax con poda alfa-beta, replicando un código de esta realizado por terceros, es a mi parecer, una forma de desaprovechar toda la riqueza que puede aportarnos un proyecto de esta índole. Así, y esta afirmación es extensible al resto del proyecto, el proyecto de juegos de acorralamiento no se ha servido de ningún tipo de reutilización de código para la realización del mismo.

Aclarado este punto, el diseño del algoritmo minimax estará basado en el siguiente pseudocódigo:

```
función BÚSQUEDA-ALFA-BETA(estado) devuelve una acción
  variables de entrada: estado, estado actual del juego

  v ← MAX-VALOR(estado, −∞, +∞)
  devolver la acción de SUCESORES(estado) con valor v

---------------------------------------------------------------

función MAX-VALOR(estado,α, β) devuelve un valor utilidad
  variables de entrada: estado, estado actual del juego
            α, valor de la mejor alternativa para MAX a lo largo del camino a estado
            β, valor de la mejor alternativa para MIN a lo largo del camino a estado

  si TEST-TERMINAL(estado) entonces devolver UTILIDAD(estado)
  v ← −∞
  para a, s en SUCESORES(estado) hacer
  v ← MAX(V, MIN-VALOR(s, α, β))
       si v ≥ β entonces devolver v
       α ← MAX(α, v)
  devolver v

---------------------------------------------------------------

función MIN-VALOR(estado, α, β) devuelve un valor utilidad
  variables de entrada: estado, estado actual del juego
            α, valor de la mejor alternativa para MAX a lo largo del camino a estado
            β, valor de la mejor alternativa para MIN a lo largo del camino a estado

  si TEST-TERMINAL(estado) entonces devolver UTILIDAD(estado)
  v ← +∞
  para a, s en SUCESORES(estado) hacer
       v ← MIN(v, MAX-VALOR(s, α, β))
       si v ≤ α entonces devolver v
       β ← MIN(β, V)
  devolver v
```

Donde la función de utilidad (también llamada función de evaluación estática) devuelve una estimación de la utilidad esperada de una posición (o situación de la partida) dada, respecto a las posibilidades actuales de ganar.

El problema radica justamente en la frase *"las posibilidades reales de ganar"*, ya que después de todo, los juegos aquí implementados son carentes de azar, pues sabemos el estado actual con certeza. Sin embargo, como la búsqueda por el árbol de movimientos, como ya comentábamos, no debe utilizar demasiado tiempo y debe cortarse en estados no terminales, entonces necesariamente será incierto sobre los resultados finales de esos estados.

Este tipo de incertidumbre está inducida por limitaciones computacionales, más que de información. Si consideramos la cantidad limitada de cálculo que se permite a la función de evaluación cuando se aplica a un estado, lo mejor que se podría hacer es una conjetura sobre el resultado final.

Profundizando en esta idea, la mayoría de las funciones de evaluación trabajan calculando varias *características* del estado, para luego sumarlas según una función ponderada lineal, dando como resultado el valor estimado del estado del tablero para un determinado jugador. Así, como los tres juegos aquí presentados comparten una mecánica de juego, en este caso los juegos de acorralamiento (o de caza), ha sido un reto interesante formar un conjunto de heurísticas que funcionen para los tres juegos implementados y que compartan las características del estado que antes mencionábamos. A saber:

- Para el nodo MAX, es decir, para la facción de menor número (el ratón, la liebre y la zorra) las características que aumentan el valor de un estado son:

- La proximidad de la ficha al extremo contrario del tablero donde comenzó.
- Sobrepasar las fichas del jugador contrario. Esto es debido a que las fichas MIN no tienen permitido el movimiento hacia atrás.
- Si se divide el tablero en zonas, encontrarse en la zona que tenga un mayor desequilibrio, es decir, que sea más fácil encontrar una brecha para poder llegar al extremo contrario del tablero de juego.
- La eliminación de fichas del jugador contrario en caso de que el juego permita que el jugador MAX capture fichas del jugador MIN,

- Para el nodo MIN, es decir, para la facción de mayor número (los gatos, los perros y los gansos) las características que aumentan el valor de un estado son:

- Mantener la línea horizontalmente.
- Evitar huecos horizontalmente.
- Evitar huecos verticalmente.
- Evitar huecos diagonalmente.
- Por definición del minimax, los estados que son beneficiosos para el nodo MAX son perjudiciales para el nodo MIN y viceversa. Por tanto todas las características citadas anteriormente para el nodo MAX son aplicables a la inversa para el nodo MIN.

Ponderando adecuadamente estas características según el juego al que las apliquemos, obtendremos muy diversas heurísticas, que posteriormente podremos enfrentar en el modo de juego CPU vs CPU.

9.3 Cuerpo de funciones comunes

Debido a que los juegos implementados pertenecen a la misma familia de juegos y van a utilizar la misma estrategia de toma de decisiones, podemos definir un conjunto de funciones comunes que conformen el cuerpo de funciones de cada juego:

- *Inicializa-juego*, función que define:
  - Jugador inicial.
  - Posiciones iniciales de las fichas.
  - Tablero de juego.
- *Crea-estado-inicial*: Donde generamos la matriz que representa el tablero y las casillas que ocupan las fichas de los jugadores.
- *Pide-ayuda*: Es una función de consejo al jugador humano sobre el movimiento que debe realizar a continuación. Devuelve la casilla origen y destino del movimiento tras una llamada a minimax.
- *Selecciona-ficha*: Si el turno activo es el del jugador humano, esta función sirve para elegir la ficha que deseamos mover.

- *Selecciona-destino:* A partir de la ficha elegida, esta función nos permite finalizar el turno seleccionando la casilla destino de nuestro movimiento. En el caso de que nuestra facción esté formada por más de 1 unidad, debe permitir cambiar la ficha elegida para el movimiento del turno actual.
- *Movimiento-minimax:* Esta función realiza el turno completo de la computadora. Primero aplica el movimiento devuelto por la función minimax y después comprueba si se ha dado alguna condición de finalización de la partida.
- *Aplica-movimiento:* A partir de las casillas origen y destino del movimiento esta función se encarga de completar el movimiento modificando adecuadamente el tablero de juego y las posiciones de las fichas.
- *Comprueba-si-fin:* Si el último movimiento que se haya llevado a cabo propicia que se active alguna de las condiciones de victoria de cualquiera de los dos bandos, la partida finaliza. En caso contrario, el turno activo pasa al jugador contrario.
- *Es-estado-ganador:* Esta función define si el estado actual de la partida hace que uno de los jugadores sea el ganador, o bien porque el jugador MAX haya llegado al extremo contrario del tablero o bien porque un jugador no tiene ninguna posibilidad de realizar un movimiento permitido.
- Movimientos-validos: Dada una ficha, indica los posibles movimientos que puede realizar según la situación actual del tablero (ya que pueden producirse bloqueos con otras fichas o, si el juego lo permite, realizar capturas).
- Finaliza-juego: Esta función declara como vencedor a uno de los jugadores, según el jugador que haya actuado el último turno.
- Casilla-libre: Calcula si una casilla del tablero se encuentra vacía o contiene alguna ficha, independientemente del propietario de esta.
- Escribe-jugadas: Esta función genera un fichero de texto plano con todas las jugadas que se realizan durante la partida. Es realmente útil poder consultar a posteriori el curso que ha ido tomando la partida para poder mejorar las heurísticas de juego. Incluso podría estudiarse la forma de realizar una red neuronal de auto-aprendizaje para mejorar la inteligencia artificial del computador, tomando como entrada los propios ficheros generados con las partidas y viendo quién ha sido el vencedor al finalizar estas.

10. Implementación

Tomando como punto de partida el modelo de la fase anterior, en esta fase vamos a proceder codificar los diseños especificados en el modelo de diseño.

Como ya hemos comentado anteriormente Lisp es el lenguaje elegido para implementar el sistema. Lisp es una familia de lenguajes de programación de computadora de tipo funcional con una larga historia y una sintaxis completamente entre paréntesis. Especificado originalmente en 1958 por John McCarthy y sus colaboradores en el MIT, Lisp es el segundo más viejo lenguaje de programación de alto nivel de extenso uso hoy en día; solamente el FORTRAN es más viejo. Al igual que el FORTRAN, Lisp ha cambiado mucho desde sus comienzos, y han existido un número de dialectos en su historia. Hoy, los dialectos Lisp de propósito general más ampliamente conocidos son el Common Lisp y el Scheme.

Lisp fue creado originalmente como una notación matemática práctica para los programas de computadora, basada en el cálculo lambda de Alonzo Church. Se convirtió rápidamente en el lenguaje de programación favorito en la investigación de la inteligencia artificial. Como uno de los primeros lenguajes de programación, Lisp fue pionero en muchas ideas en ciencias de la computación, incluyendo la recursividad, las estructuras de datos de árbol, el manejo de almacenamiento automático, tipos dinámicos, y el compilador auto contenido.

El nombre LISP deriva del "**LIS**t **P**rocessing" (Proceso de LIStas). Las listas encadenadas son una de las estructuras de datos importantes del Lisp, y el código fuente del Lisp en sí mismo está compuesto de listas. Como resultado, los programas de Lisp pueden manipular el código fuente como una estructura de datos, dando lugar a los macro sistemas que permiten a los programadores crear una nueva sintaxis de lenguajes de programación de dominio específico empotrados en Lisp.

En este sentido Lisp fue el primer lenguaje de programación homoicónico: la representación primaria del código del programa es el mismo tipo de estructura de la lista que también es usada para las principales estructuras de datos. Como resultado, las funciones del Lisp pueden ser manipuladas, alteradas o aún creadas dentro de un programa Lisp sin un extensivo análisis sintáctico (parsing) o manipulación de código de máquina binario.

Sus principales ventajas son:

- Es un lenguaje funcional y simbólico, por lo que a diferencia de los estructurados y los Orientados a Objetos, no precisa declarar los tipos de las variables ni tampoco reservar memoria.
- Sencillo.
- Permite el paso de funciones como parámetros a otras funciones.
- Su interacción, al ser un lenguaje interpretado, permite programar a partir de un terminal obteniendo respuestas rápidas.
- Es un lenguaje uniforme, con un solo tipo de datos: la lista (y su variante de nivel 0, el átomo).
- Su principal forma de ejecución es la recursión, en contra de los lenguajes estructurados que se basan en la iteración.

Los inconvenientes más destacables son:

- La ejecución de los programas es más lenta, pues el intérprete tiene que gestionar la memoria y las pilas de recursión, cosa que lo hace bastante más lento que los lenguajes estructurados.
- Código poco limpio y difícil de comprender, sobretodo en programas de tamaño considerable, debido al uso de paréntesis y la notación inversa.
- Dificultad para depurarlo. Aunque haya la instrucción *trace*, que te permite trazar la ejecución, la depuración es una tarea difícil en todos los lenguajes funcionales.

Realizada la presentación de lenguaje de programación, analicemos en detalle la codificación de las dos secciones del proyecto que tienen mayor interés: cómo se implementan interfaces gráficas con Lisp y cómo se implementa el algoritmo recursivo minimax con poda alfa-beta.

10.1 Interfaces gráficas en Lisp con LTK

En este apartado daremos las nociones básicas para poder crear fácilmente una interfaz sobria y funcional.

LTK es una creación de Peter Herth. Consiste en una tubería que transforma nuestro código lisp en código tcl que es ejecutado por el programa *wish*. No es necesario entenderla para poder usarla, LTK es tremendamente sencilla de usar.

Vamos a crear un pequeño tutorial con las funciones más utilizadas y para comenzar vamos a explicar cómo dibujar una etiqueta que nos muestre "Hola, mundo”:

```
(use-package :ltk )
(with-ltk ()
(let
(( etiqueta (make-instance 'label :master nil :text "Hola, mundo" )))
(pack etiqueta)))
```

La primera línea es conocida: (use-package :ltk) simplemente importa los símbolos que necesitamos, para no tener que escribir todo el rato ltk:laFuncionQueNosInteresa.

Es en la segunda línea donde empezamos a usar ltk. La macro with-ltk inicializa la librería y ejecuta todo el código que metemos en su cuerpo. Tiene unos parámetros opcionales (en nuestro caso, la lista vacía --()--) que nos permiten controlar ciertas opciones de debug.

Cada tipo de widget en ltk se representa mediante una clase. Por eso, cuando llamamos a make-instance 'label estamos creando una etiqueta. Todos los widgets de ltk tienen unas opciones de inicialización comunes (por ejemplo *master* ; que indica el widget padre. Si lo dejamos a nil, ltk tomará a este widget como del nivel padre).

Pack es lo que se llama un "package manager", o gestor de empaquetado. Es la función que usamos cuando queremos situar un widget sobre la pantalla, y se encarga de tareas como su posición relativa a otros widgets y (en ocasiones) su tamaño.

Una vez creada nuestra primera interfaz gráfica, ya podemos comenzar a introducir nuevos widgets en nuestra ventana, y a interactuar con ellos. Para ver el manual completo de LTK podemos visitar http://www.peter-herth.de/ltk/ . Aquí expondremos a continuación varios ejemplos, pero al ser las posibilidades ilimitadas, lo mejor es explotar las facultades de la librería por uno mismo incrementando la complejidad de los widgets.

Crear la pantalla de inicio

```
(with-ltk ()
          (setf *ventana* (make-instance 'label
                                :master nil
                                :width 800
                                :height 600))
          (let ((img (make-image))
              (imgvacia (make-image)))
            (pack *ventana*)
            (image-load img "./imagenes/portada.gif")
            (configure *ventana* :image img)
            (bind *ventana* "<ButtonPress-1>"
                (lambda (evt)
                  (bind *ventana* "<ButtonPress-1>" ())
                  (configure *ventana* :image imgvacia)
                  (elige-juego))))))
```

Con el código anterior creamos una etiqueta de tamaño 800x600 que llamaremos ventana, que será la ventana de la aplicación. Además con las instrucción configure le asignamos una imagen, que será la portada de nuestro proyecto.

Mención especial merece la instrucción *bind* que será una de las más utilizadas, debido a que es la encargada de ejecutar una función cuando se produce un evento sobre el widget que especifiquemos. La estructura de utilización es la siguiente:

(bind nombre-widget "<ButtonPress-1>" (lambda (evt) ……))

Que traducido a lenguaje natural sería:

Cuando pulses el widget nombre-widget con el botón izquierdo del ratón haz…

Otras funciones y widgets de utilidad

- El widget Botón:

  *(boton1 (make-instance 'button :master *ventana* :text "Juego 1"))*

- La instrucción Place:

  (place boton1 50 200 :width 200 :height 200)

- La instrucción Destroy:

  (destroy boton1)

- La instrucción do-msg para crear una ventana emergente:

```
(setf *botonSalir* (make-instance 'button :master *ventana*
                                          :text "Salir"
                                           :command (lambda ()
                                          (do-msg "¡Hasta la próxima!")
                                          (setf *exit-mainloop* t))))
```

- La instrucción image-load:

  (image-load img "./imagenes/portada.gif")

- La instrucción Bind para desactivar algún evento:

  (bind x "<ButtonPress-1>" ())

- El parámetro :background:

  *(configure (aref *casillas* 0 4) :background :blue)*

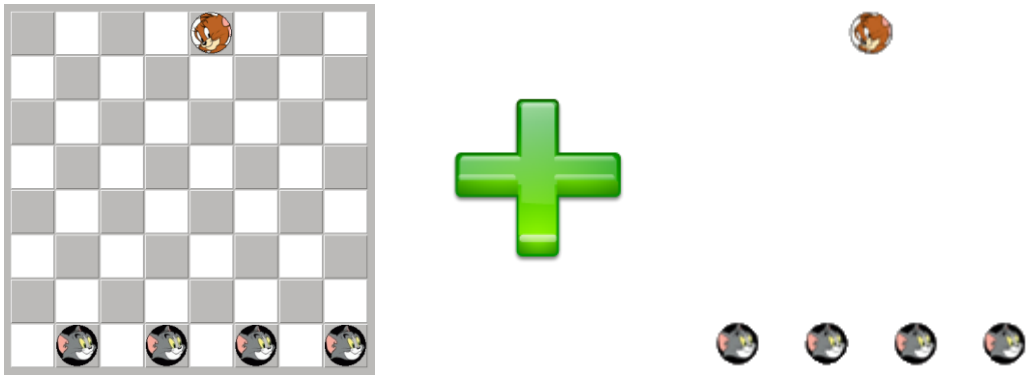
10.2 Estrategia minimax

La estrategia minimax es un esquema genérico de búsqueda para situaciones en las que el objetivo es ganar una partida en la que participan dos adversarios que realizan movimientos alternos en el juego.

El método consiste en prever el mayor número de jugadas posibles y actuar en consecuencia. Se considera siempre que el adversario va a realizar la mejor de las opciones posibles cuando a él le toque mover. Los nodos del árbol son de dos tipos: nodos MAX y nodos MIN, cada uno de los cuáles toma las decisiones de uno de los dos contrincantes. El objetivo del jugador MAX es maximizar el valor de la función f de evaluación que mide la proximidad estimada a la meta de una situación dada y, así mismo, los nodos MIN reflejan el modo de actuar del adversario con respecto al objetivo del nodo MAX; es decir, realizar aquel movimiento que haga más pequeño el valor de f.

Estructura de datos

Tan importante o más que realizar una eficiente implementación del algoritmo minimax es entender y encontrar la estructura de datos que mejor se adecue a dicha implementación, ya que será la que represente cada nodo del árbol de búsqueda.



La elección correcta de representación del nodo supone gran parte del desarrollo del algoritmo minimax ya que condicionará en número y complejidad las funciones que lo componen. Necesitamos una estructura que sea fácilmente accesible y de la que podamos extraer rápidamente toda la información del estado de la partida en cualquier momento del juego. Así, la estructura de datos más eficiente para la implementación que le he dado al algoritmo minimax es un par punteado donde:

- El primer elemento del par *(first tablero-sucesor)* punteado es la variable *tablero* que es representado por una matriz donde un 0 simboliza una casilla vacía, un 1 las fichas del jugador MAX, un 2 las fichas del jugador MIN y un -1 posiciones de la matriz que no tienen casilla asociada en el tablero.
- El segundo elemento *(rest tablero-sucesor)* del par punteado es una lista de tamaño igual al número de fichas que existan en el tablero en un determinado estado de la partida. Cada posición de la lista será a su vez un par punteado que indique las coordenadas i y j de cada una de las casillas. Además, es requisito imprescindible que el primer par punteado de esta lista se corresponda con la posición de la ficha MAX.

Así, el ejemplo de la página anterior queda representado por el siguiente par punteado:

```
(#2A((0 0 0 0 1 0 0 0)
     (0 0 0 0 0 0 0 0)
     (0 0 0 0 0 0 0 0)
     (0 0 0 0 0 0 0 0)
     (0 0 0 0 0 0 0 0)
     (0 0 0 0 0 0 0 0)
     (0 0 0 0 0 0 0 0)
     (0 2 0 2 0 2 0 2))  (0 . 4) (7 . 1) (7 . 3) (7 . 5) (7 . 7))
```

Funciones

A continuación se detallan las distintas funciones que integran la implementación concreta del algoritmo minimax que he utilizado en este proyecto.

minimax-nodoMAX (minimax-nodoMIN)

Función principal para el cálculo minimax sobre un nodo MAX (MIN), a partir de esta función comenzara la recursión, la profundidad y la poda alfa-beta. Devuelve el mejor tablero sucesor.

```
(defun minimax-nodoMAX ()
  (let* ((valor-max *minimo-valor*)
       (sucesores (calcula-sucesores *tablero* *posiciones* 1))
       (valor-actual 0)
       (mejor-sucesor (first sucesores)))
    (loop for sucesor-actual in sucesores do
        (setf valor-actual
            (minimizador-alpha-beta
                  (first sucesor-actual) (rest sucesor-actual)
                   *profundidad* valor-max *maximo-valor*))
        (cond ((= valor-actual 10000)
             (setf mejor-sucesor sucesor-actual)
             (return))
            ((> valor-actual valor-max)
             (setf mejor-sucesor sucesor-actual)
             (setf valor-max valor-actual))
            (t nil)))
    mejor-sucesor))
```

```
(defun minimax-nodoMIN ()
  (let* ((valor-min *maximo-valor*)
       (sucesores (calcula-sucesores *tablero* *posiciones* 2))
       (valor-actual 0)
       (mejor-sucesor (first sucesores)))
    (loop for sucesor-actual in sucesores do
        (setf valor-actual
            (maximizador-alpha-beta
                  (first sucesor-actual) (rest sucesor-actual)
                   *profundidad* *minimo-valor* valor-min))
        (cond ((= valor-actual -10000)
             (setf mejor-sucesor sucesor-actual)
             (return))
            ((< valor-actual valor-min)
             (setf mejor-sucesor sucesor-actual)
             (setf valor-min valor-actual))
            (t nil)))
    mejor-sucesor))
```

maximizador-alpha-beta (minimizador-alpha-beta)

Función recursiva principal con poda alfa-beta. Las condiciones de parada de la recursión son que el tablero pasado como parámetro sea un estado final o que el árbol de búsqueda haya llegado a la profundidad definida por el nivel de dificultad elegido. Si se produce el primer caso devolvemos el máximo valor (10000) indicando que ese estado nos da la victoria. En el segundo caso llamamos a la función de evaluación estática *heurística* que nos dará un valor de utilidad del tablero para el nodo correspondiente.

```
(defun maximizador-alpha-beta (sucesor posiciones profundidad alpha
beta)
  (let ((valor-max *minimo-valor*))
    (cond ((es-estado-ganador sucesor posiciones 1)
         (if (= profundidad *profundidad*)
             (setf valor-max -10000)
           (setf valor-max *minimo-valor*)))
        ((= profundidad 0)
         (setf valor-max
            (funcall *heuristica-MAX* sucesor posiciones 1)))
        (t (let ((sucesores (calcula-sucesores sucesor posiciones 1))
         (poda nil))
           (loop for sucesor-actual in sucesores while (not poda) do
               (setf valor-max (max
  (minimizador-alpha-beta (first sucesor-actual) (rest sucesor-actual)
                                          (1- profundidad) alpha beta)
                                valor-max))
               (if (>= valor-max beta)
                   (setf poda t))
               (setf alpha (max alpha valor-max))))))
    valor-max))


(defun minimizador-alpha-beta (sucesor posiciones profundidad alpha
beta)
  (let ((valor-min *maximo-valor*))
    (cond ((es-estado-ganador sucesor posiciones 2)
         (if (= profundidad *profundidad*)
             (setf valor-min 10000)
           (setf valor-min *maximo-valor*)))
        ((= profundidad 0)
         (setf valor-min
            (funcall *heuristica-MIN* sucesor posiciones 2)))
        (t (let ((sucesores (calcula-sucesores sucesor posiciones 2))
             (poda nil))
           (loop for sucesor-actual in sucesores while (not poda) do
               (setf valor-min (min
(maximizador-alpha-beta (first sucesor-actual) (rest sucesor-actual)
                                          (1- profundidad) alpha beta)
                                valor-min))
               (if (<= valor-min alpha)
                   (setf poda t))
               (setf beta (min beta valor-min))))))
    valor-min))
```

Calcula-sucesores

Función auxiliar para el algoritmo minimax. Dados un sucesor (tablero + posiciones) y un turno, calcula todos los sucesores de dicho tablero. Para ello, necesitará el uso de la ya comentada función movimientos-validos. Así, por cada movimiento válido se hará una llamada a la función *crea-sucesor*, que será la que generé el nuevo sucesor propiamente dicho.

```
(defun calcula-sucesores (tablero posiciones turno)
  (let ((sucesores nil)
      (origen nil))
    (cond ((= turno 1)
         (setf origen (first posiciones))
         (loop for posibilidad in
                  (movimientos-validos tablero origen turno) do
             (setf sucesores (append sucesores
 (list (crea-sucesor tablero posiciones origen posibilidad turno))))))
        ((= turno 2)
         (setf origen (subseq posiciones 1 5))
         (loop for orig in origen do
             (loop for posibilidad in
                  (movimientos-validos tablero orig turno) do
             (setf sucesores (append sucesores
(list (crea-sucesor tablero posiciones orig posibilidad turno)))))))))
    sucesores))
```

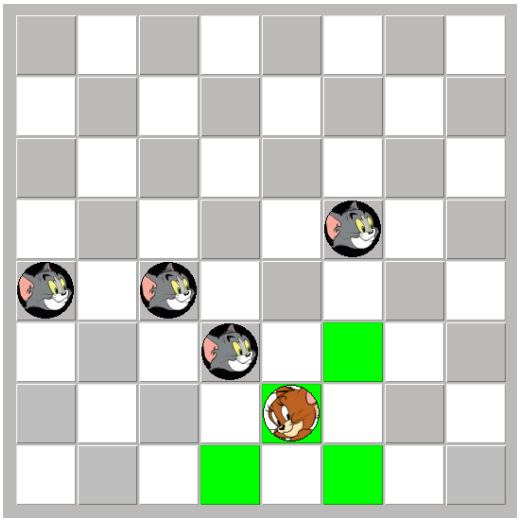
```
(defun crea-sucesor (tablero posiciones origen destino turno)
  (let ((sucesor (make-array '(8 8)))
      (posicion (loop for i in posiciones collect i)))
    (loop for i from 0 to 4 when (eq origen (nth i posiciones)) do
        (setf (nth i posicion) destino))
    (loop for fila from 0 to 7 do
        (loop for columna from 0 to 7 do
    (setf (aref sucesor fila columna) (aref tablero fila columna))))
    (setf (aref sucesor (first origen) (rest origen)) 0)
    (setf (aref sucesor (first destino) (rest destino)) turno)
    (cons sucesor posicion)))
```

11. Pruebas

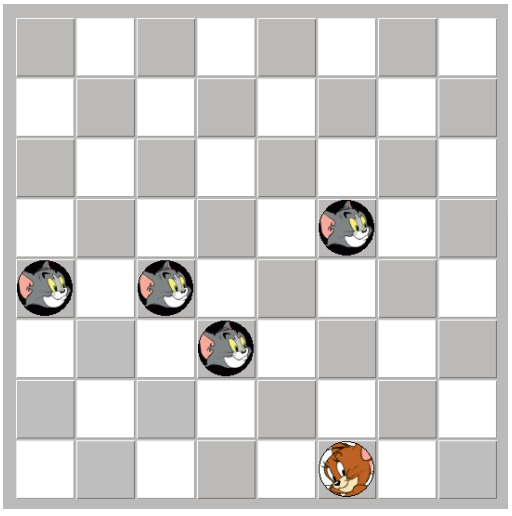
1. Caso de Prueba 1

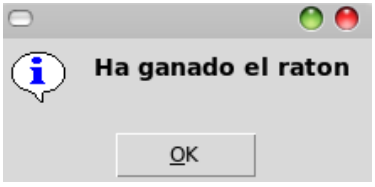
Fecha	Descripción de la prueba
12/07/2010	Realización de una partida al juego el ratón y los gatos y victoria del ratón por llegar al extremo contrario del tablero.

Datos de entrada



Resultados esperados





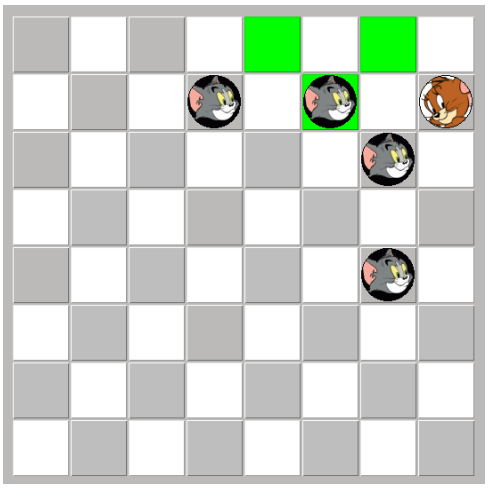
Resultados obtenidos

OK.

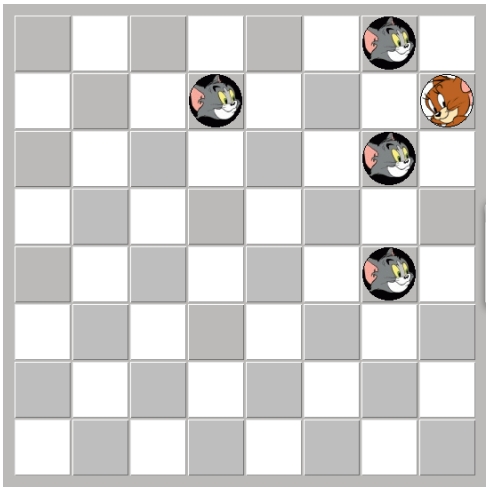
2. Caso de Prueba 2

Fecha	Descripción de la prueba
12/07/2010	Realización de una partida al juego el ratón y los gatos y victoria de los gatos por acorralar al ratón.

Datos de entrada



Resultados esperados



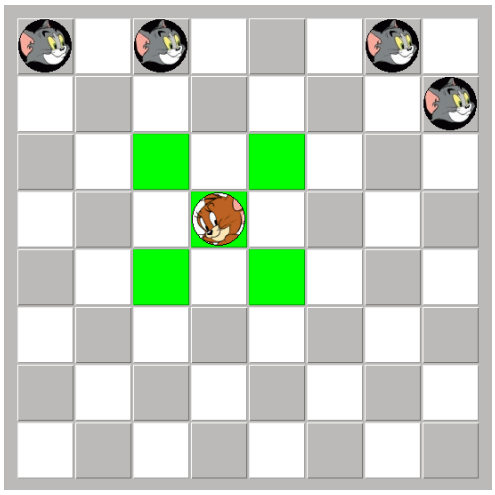
Resultados obtenidos

OK.

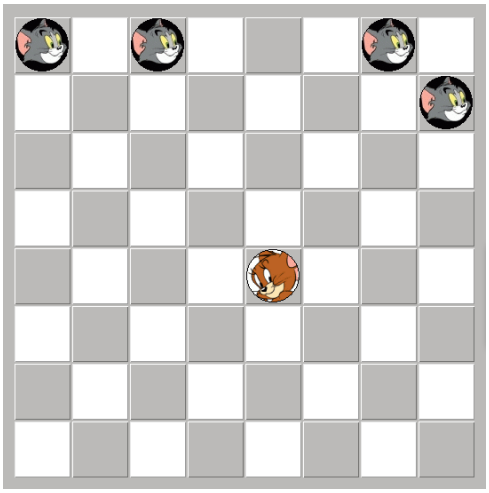
3. Caso de Prueba 3

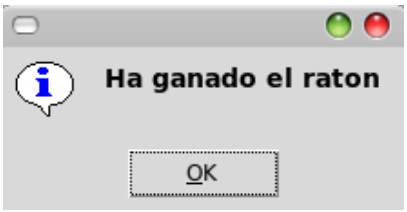
Fecha	Descripción de la prueba
12/07/2010	Realización de una partida al juego el ratón y los gatos y victoria del ratón por imposibilidad de movimiento de los gatos.

Datos de entrada



Resultados esperados





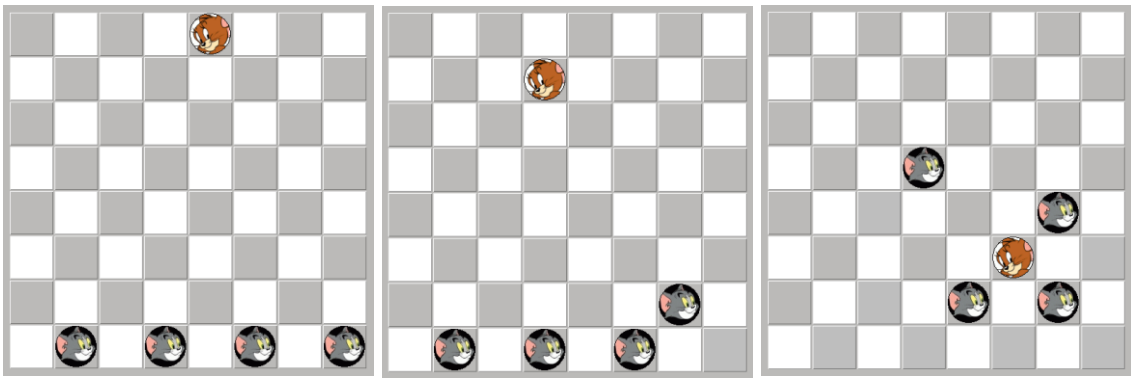
Resultados obtenidos

OK.

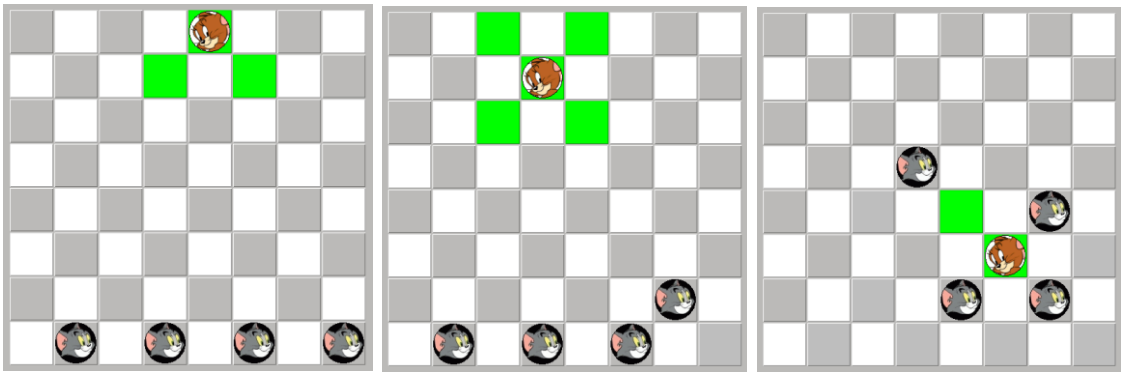
4. Caso de Prueba 4

Fecha	Descripción de la prueba
12/07/2010	Selección de movimientos válidos del ratón en diferentes situaciones de partida.

Datos de entrada



Resultados esperados



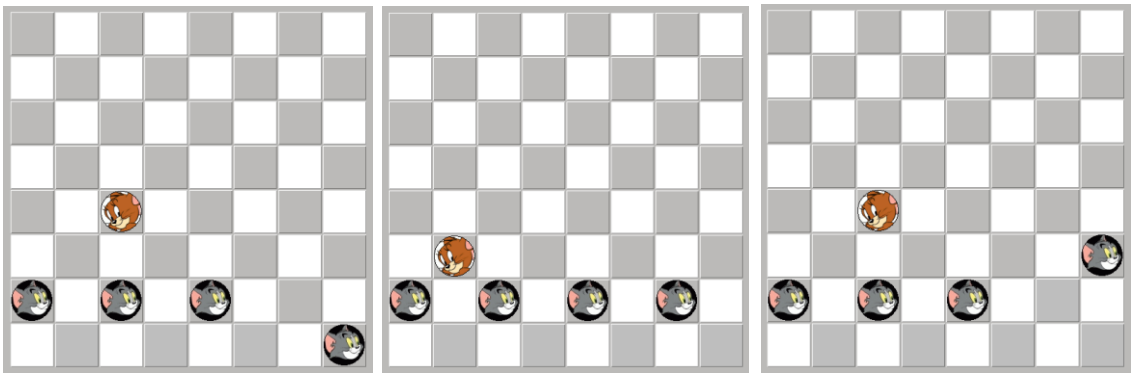
Resultados obtenidos

OK.

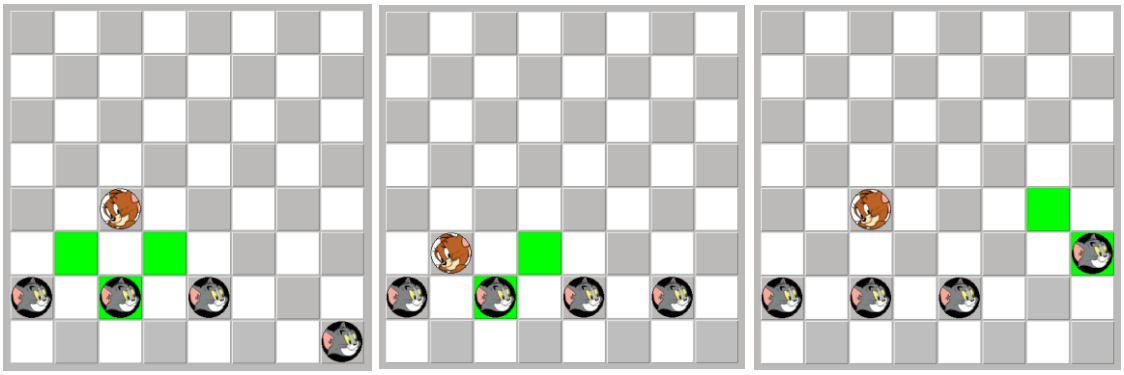
5. Caso de Prueba 5

Fecha	Descripción de la prueba
12/07/2010	Selección de movimientos válidos de los gatos en diferentes situaciones de partida.

Datos de entrada



Resultados esperados



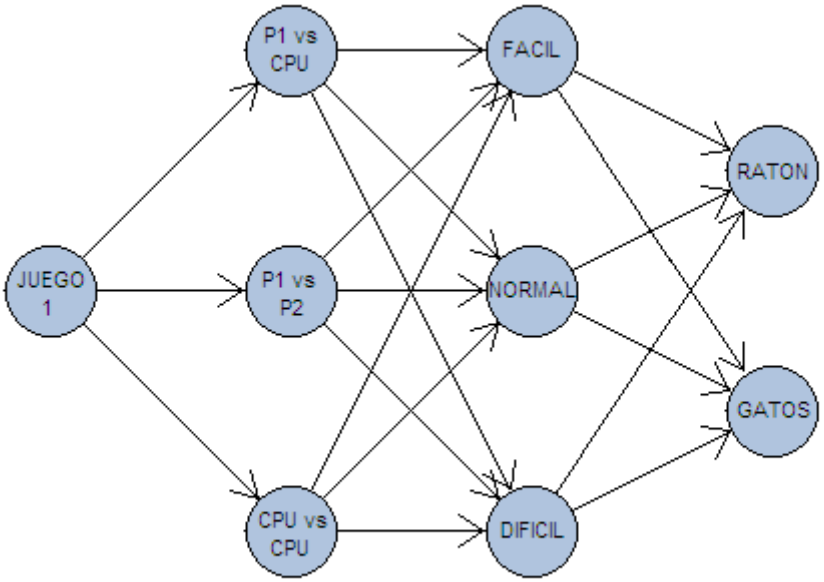
Resultados obtenidos

OK.

6. Caso de Prueba 6

Fecha	Descripción de la prueba
12/07/2010	Realización de una partida al juego el ratón y los gatos con todas las combinaciones posibles de modos de juego.

Datos de entrada



Resultados esperados

A partir del grafo dirigido anterior; todas las partidas creadas deben generarse con las características indicadas en los nodos según el camino que hayamos elegido hasta el comienzo de la partida.

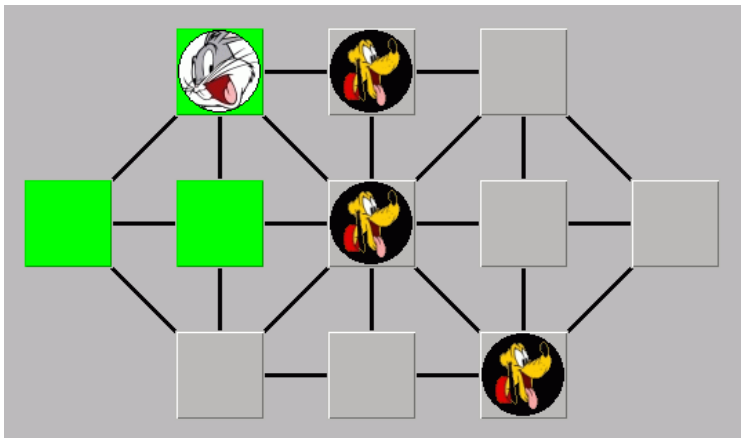
Resultados obtenidos

OK.

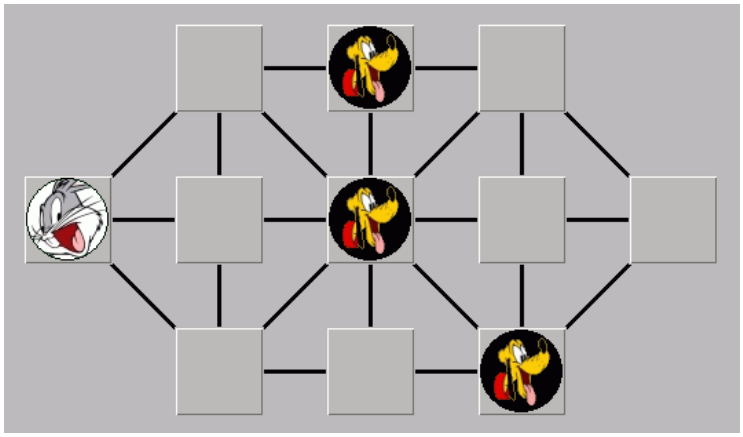
7. Caso de Prueba 7

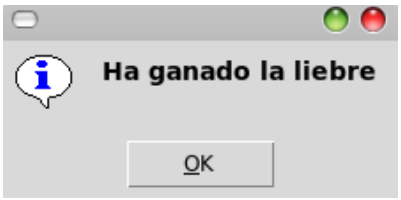
Fecha	Descripción de la prueba
12/07/2010	Realización de una partida al juego la liebre y los perros y victoria de la liebre por llegar al extremo contrario del tablero.

Datos de entrada



Resultados esperados





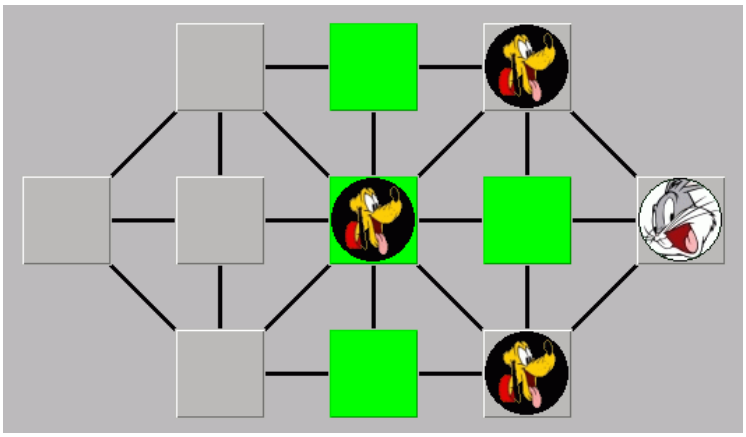
Resultados obtenidos

OK.

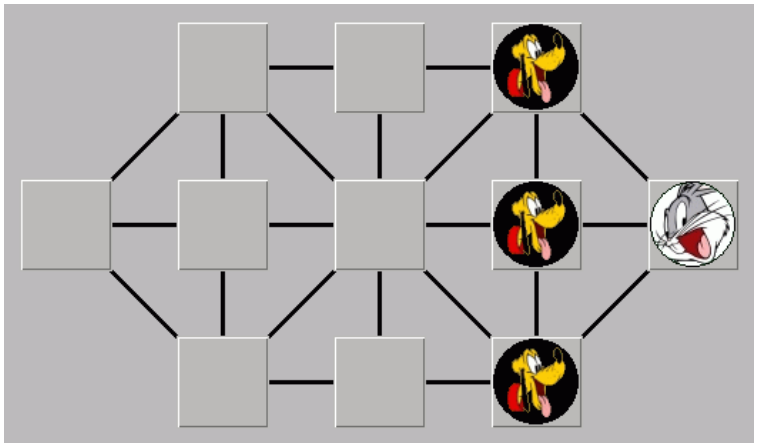
8. Caso de Prueba 8

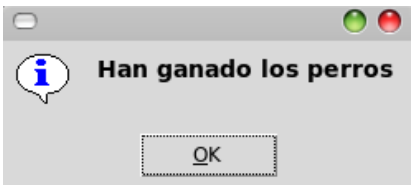
Fecha	Descripción de la prueba
12/07/2010	Realización de una partida al juego la liebre y los perros y victoria de los perros por acorralar a la liebre.

Datos de entrada



Resultados esperados





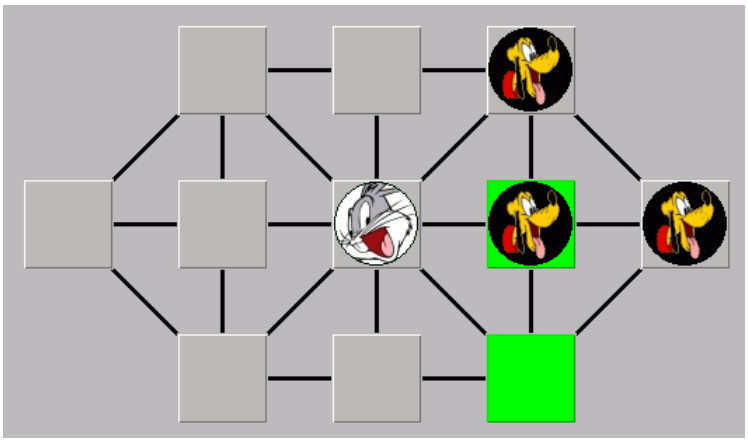
Resultados obtenidos

OK.

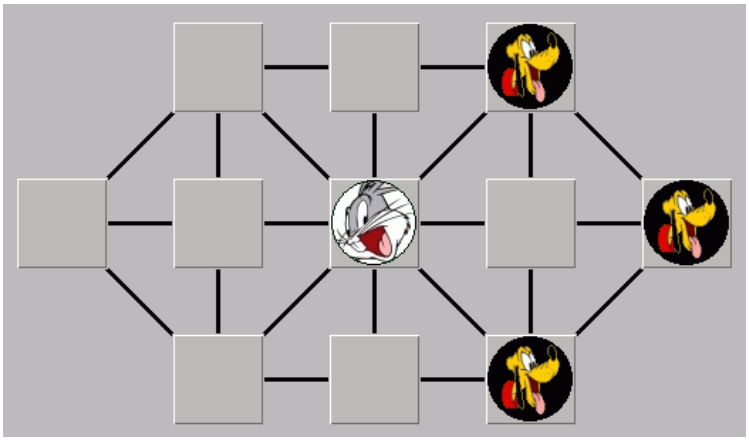
9. Caso de Prueba 9

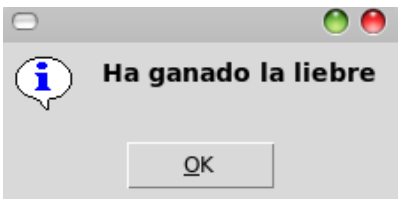
Fecha	Descripción de la prueba
12/07/2010	Realización de una partida al juego la liebre y los perros y victoria de la liebre por 10 movimientos horizontales consecutivos de los perros.

Datos de entrada



Resultados esperados





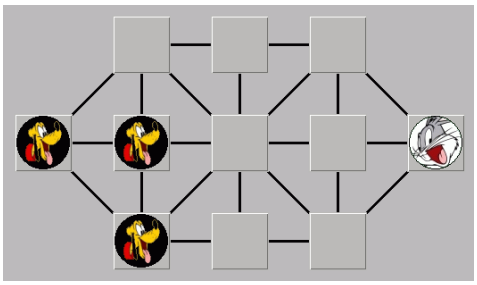
Resultados obtenidos

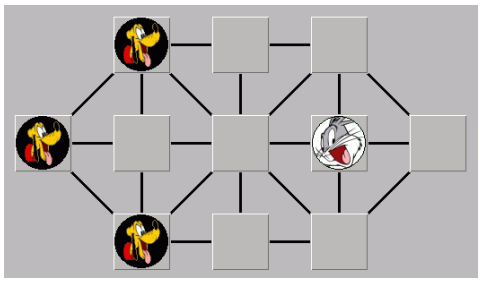
OK.

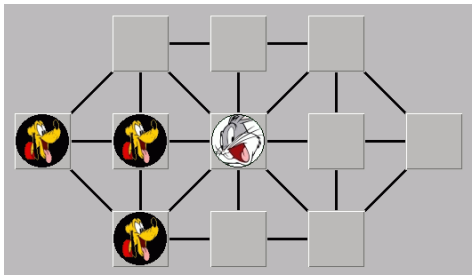
10. Caso de Prueba 10

Fecha	Descripción de la prueba
12/07/2010	Selección de movimientos válidos de la liebre en diferentes situaciones de partida.

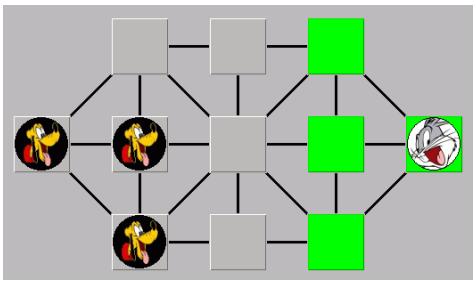
Datos de entrada

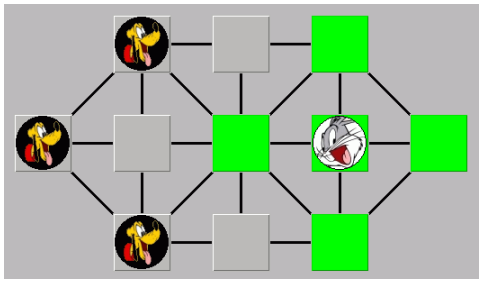


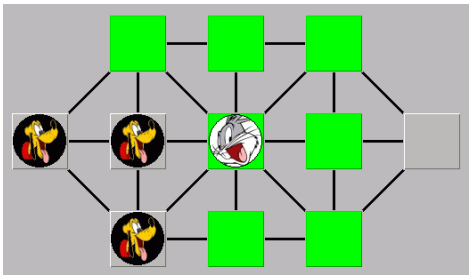




Resultados esperados







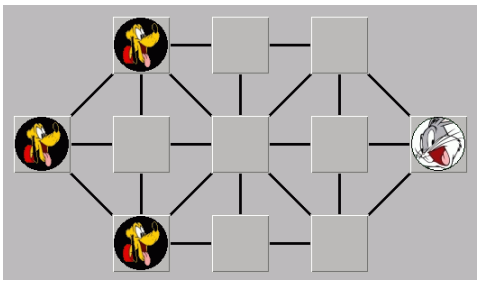
Resultados obtenidos

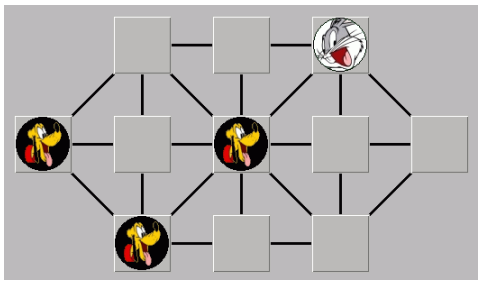
OK.

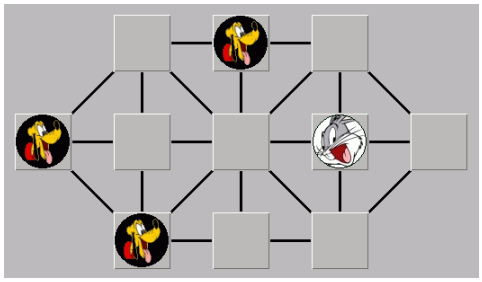
11. Caso de Prueba 11

Fecha	Descripción de la prueba
12/07/2010	Selección de movimientos válidos de los gatos en diferentes situaciones de partida.

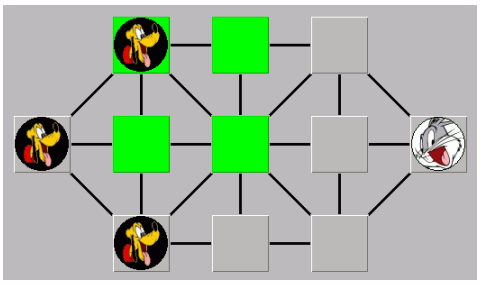
Datos de entrada

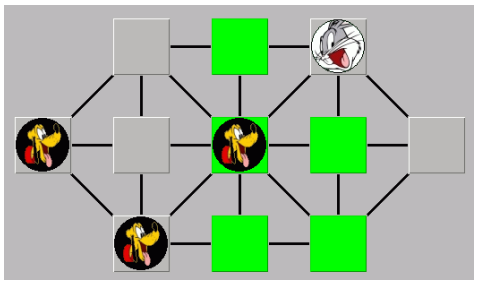


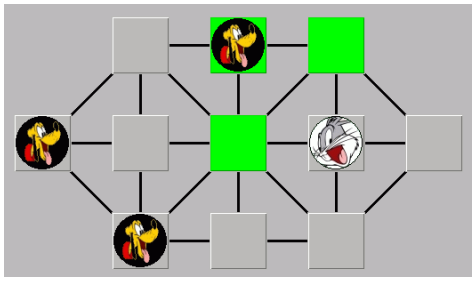




Resultados esperados







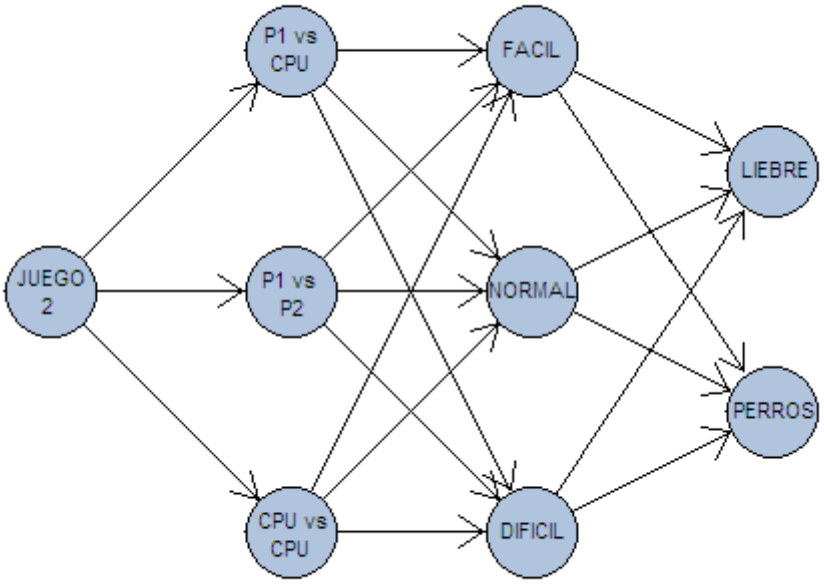
Resultados obtenidos

OK.

12. Caso de Prueba 12

Fecha	Descripción de la prueba
12/07/2010	Realización de una partida al juego la liebre y los perros con todas las combinaciones posibles de modos de juego.

Datos de entrada



Resultados esperados

A partir del grafo dirigido anterior; todas las partidas creadas deben generarse con las características indicadas en los nodos según el camino que hayamos elegido hasta el comienzo de la partida.

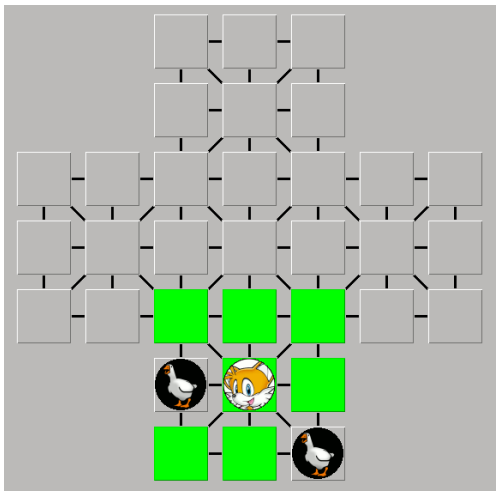
Resultados obtenidos

OK.

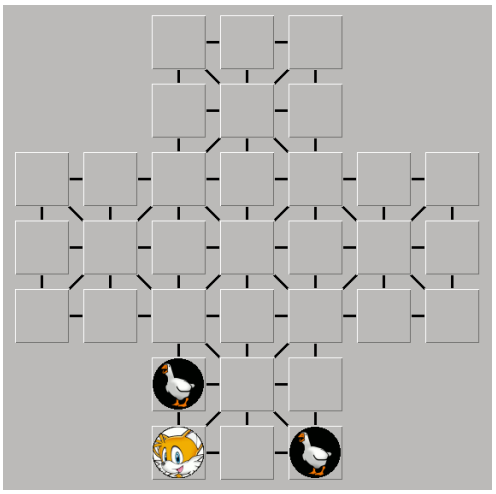
13. Caso de Prueba 13

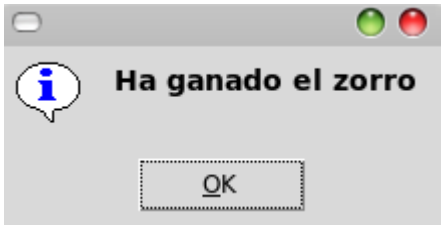
Fecha	Descripción de la prueba
12/07/2010	Realización de una partida al juego la zorra y los gansos y victoria de la zorra por llegar al extremo contrario del tablero.

Datos de entrada



Resultados esperados





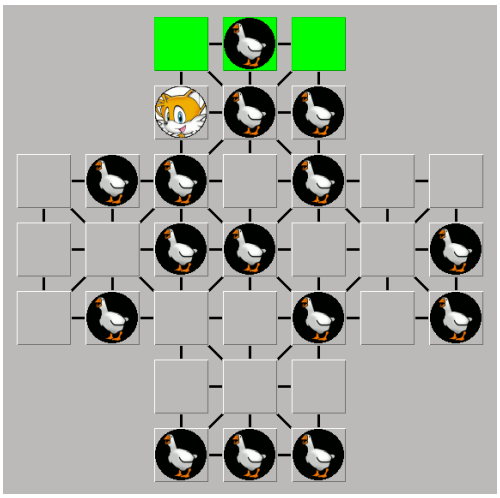
Resultados obtenidos

OK.

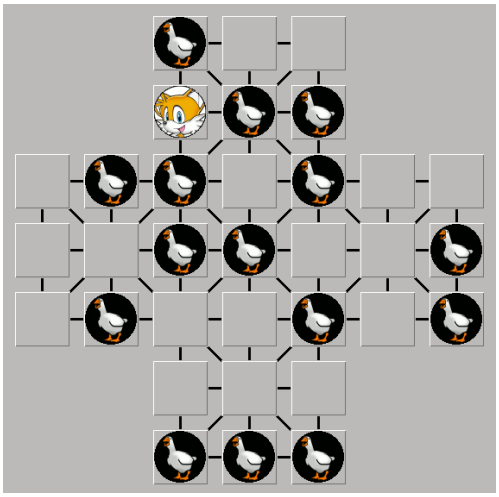
14. Caso de Prueba 14

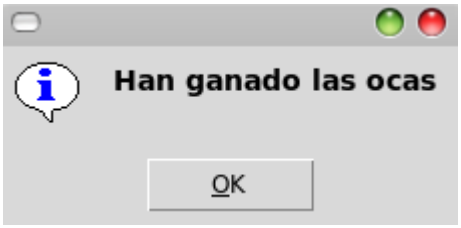
Fecha	Descripción de la prueba
12/07/2010	Realización de una partida al juego la zorra y los gansos y victoria de los gansos por acorralar a la zorra.

Datos de entrada



Resultados esperados





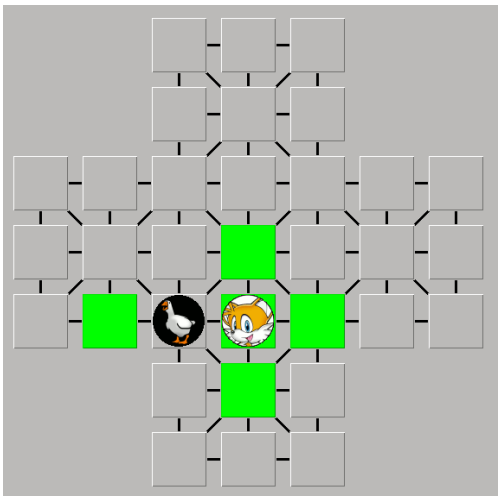
Resultados obtenidos

OK.

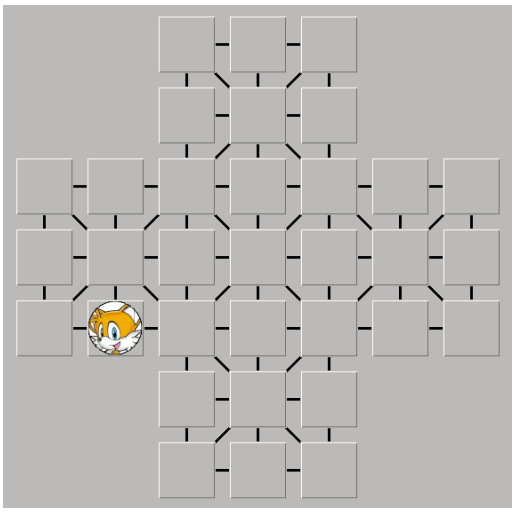
15. Caso de Prueba 15

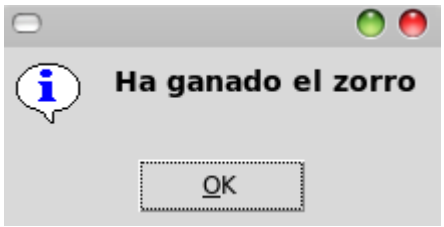
Fecha	Descripción de la prueba
12/07/2010	Realización de una partida al juego la zorra y los gansos y victoria de la zorra por haberse comido a todos los gansos (imposibilidad de movimiento).

Datos de entrada



Resultados esperados





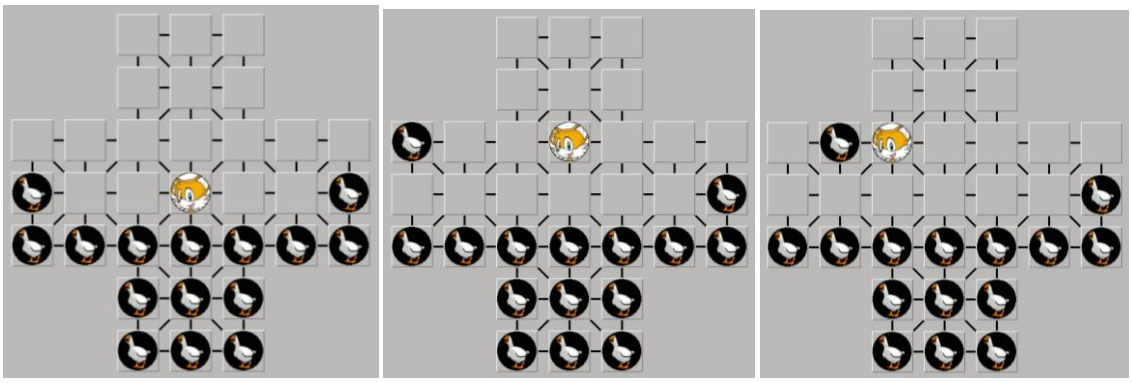
Resultados obtenidos

OK.

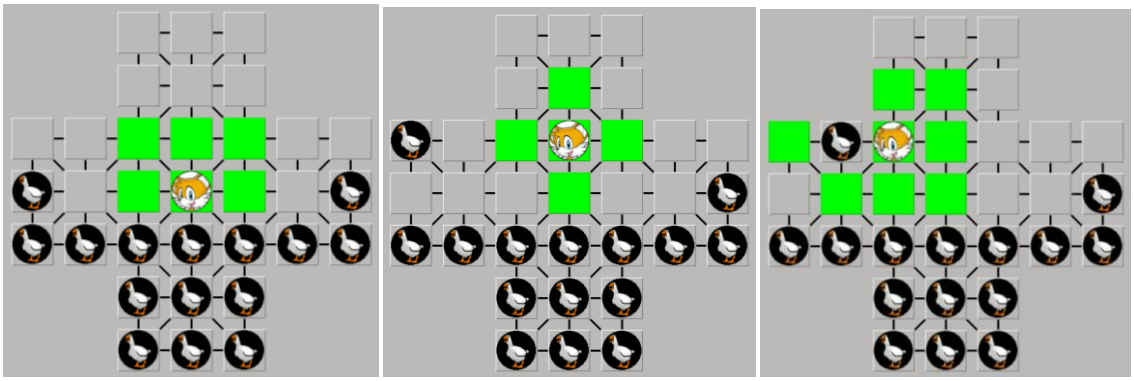
16. Caso de Prueba 16

Fecha	Descripción de la prueba
12/07/2010	Selección de movimientos válidos de la zorra en diferentes situaciones de partida.

Datos de entrada



Resultados esperados



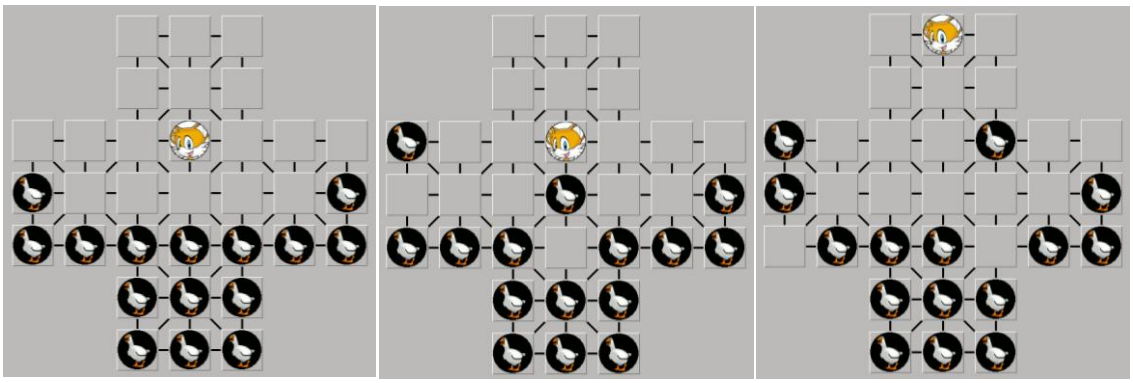
Resultados obtenidos

OK.

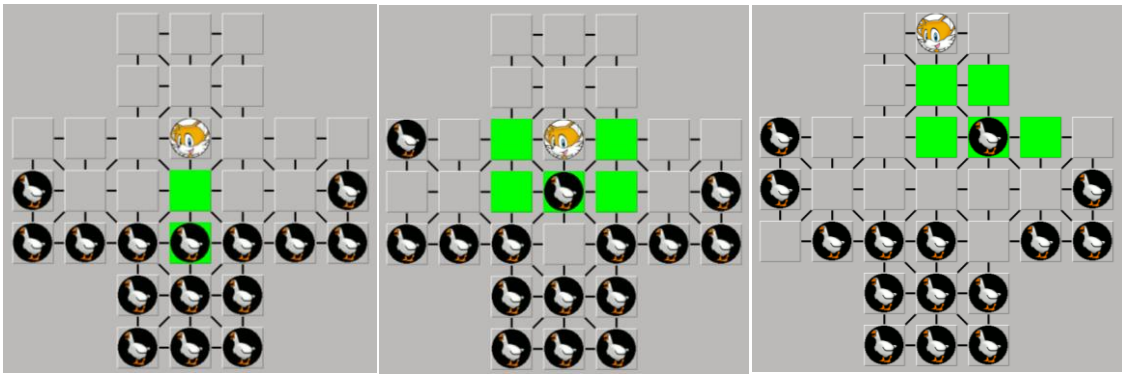
17. Caso de Prueba 17

Fecha	Descripción de la prueba
12/07/2010	Selección de movimientos válidos de los gansos en diferentes situaciones de partida.

Datos de entrada



Resultados esperados



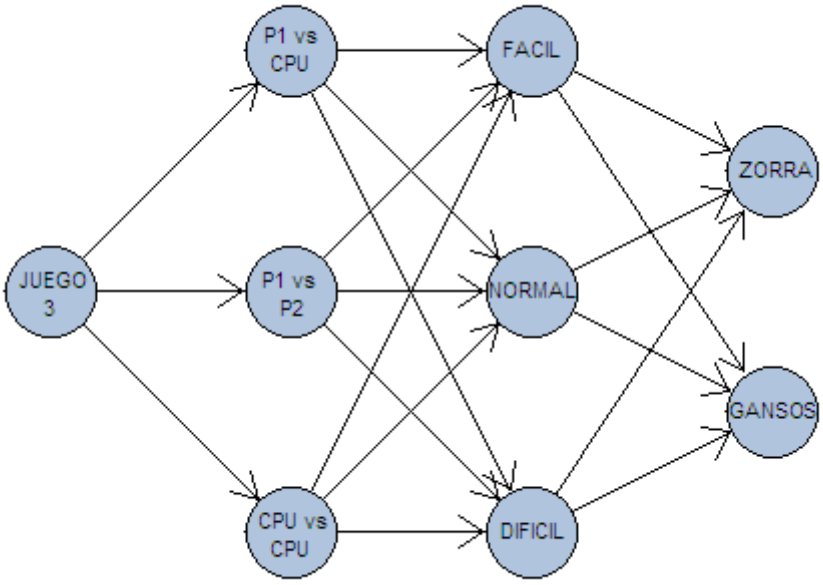
Resultados obtenidos

OK.

18. Caso de Prueba 18

Fecha	Descripción de la prueba
12/07/2010	Realización de una partida al juego la zorra y los gansos con todas las combinaciones posibles de modos de juego.

Datos de entrada



Resultados esperados

A partir del grafo dirigido anterior; todas las partidas creadas deben generarse con las características indicadas en los nodos según el camino que hayamos elegido hasta el comienzo de la partida.

Resultados obtenidos

OK.

12. Manual de usuario

Pantalla de Bienvenida

La pantalla de bienvenida no es más que la carta de presentación del software, debe llamar la atención de usuario, contener el título del proyecto y una imagen de fondo acorde con el producto software desarrollado, en este caso, los *juegos de acorralamiento*.

Con un solo clic de ratón en cualquier punto de la pantalla de bienvenida navegaremos hasta la pantalla de Selección de Juego.

Pantalla de Selección de Juego

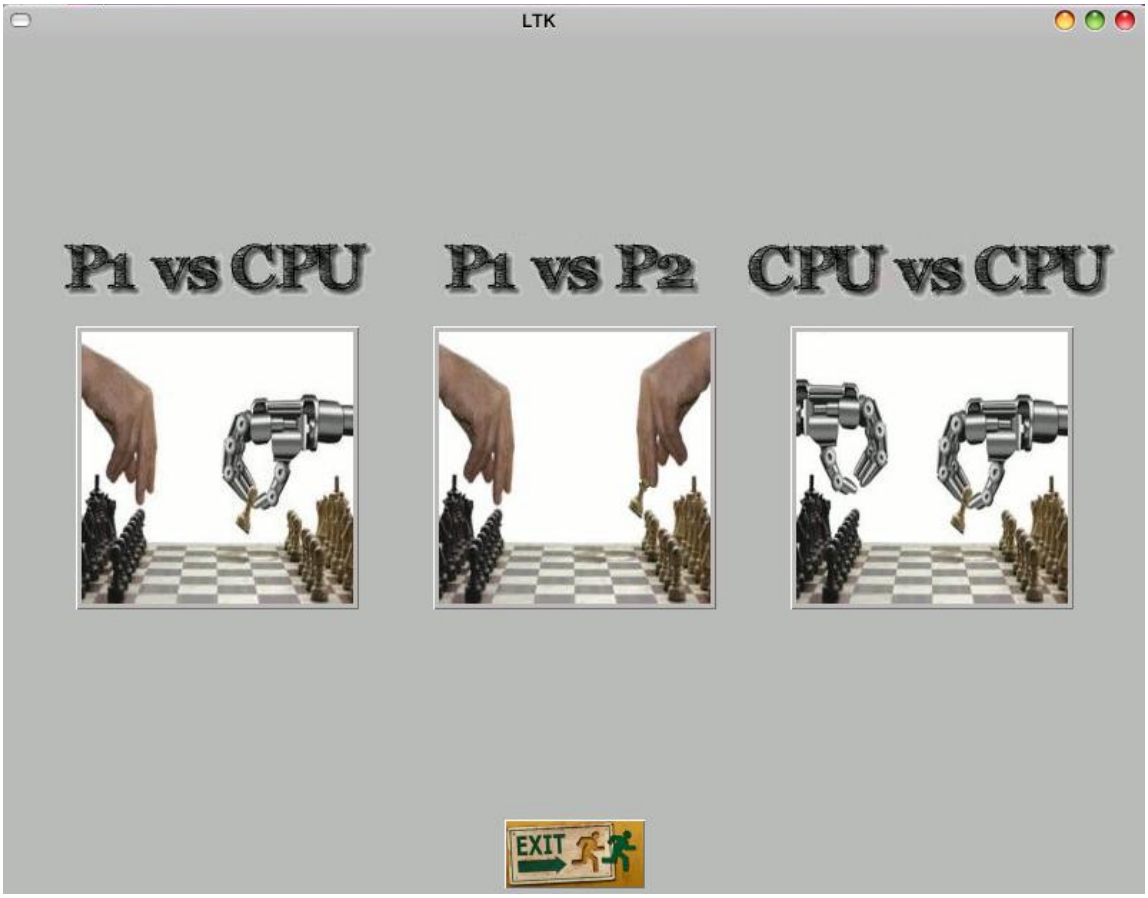
La pantalla de selección de juego muestra tres botones que se corresponden con los 3 juegos implementados: *El ratón y los gatos; la liebre y los perros; y la zorra y los gansos.*

Además, y de aquí en adelante, ya disponemos del botón Salir de la aplicación. Este botón estará siempre activo y podemos hacer uso de él en cualquier momento para abandonar la aplicación.

Una vez elegido el juego deseado para jugar la partida, debemos hacer clic en el botón correspondiente, y accederemos hasta la pantalla de Selección de Modo.

Nota: Aunque el área de pulsación de los botones es claramente identificable, cuando el puntero entra en el área de pulsación, el botón activo se ilumina y el puntero pasa de ser la flecha habitual a ser un icono representando una mano.

Pantalla de Selección de Modo

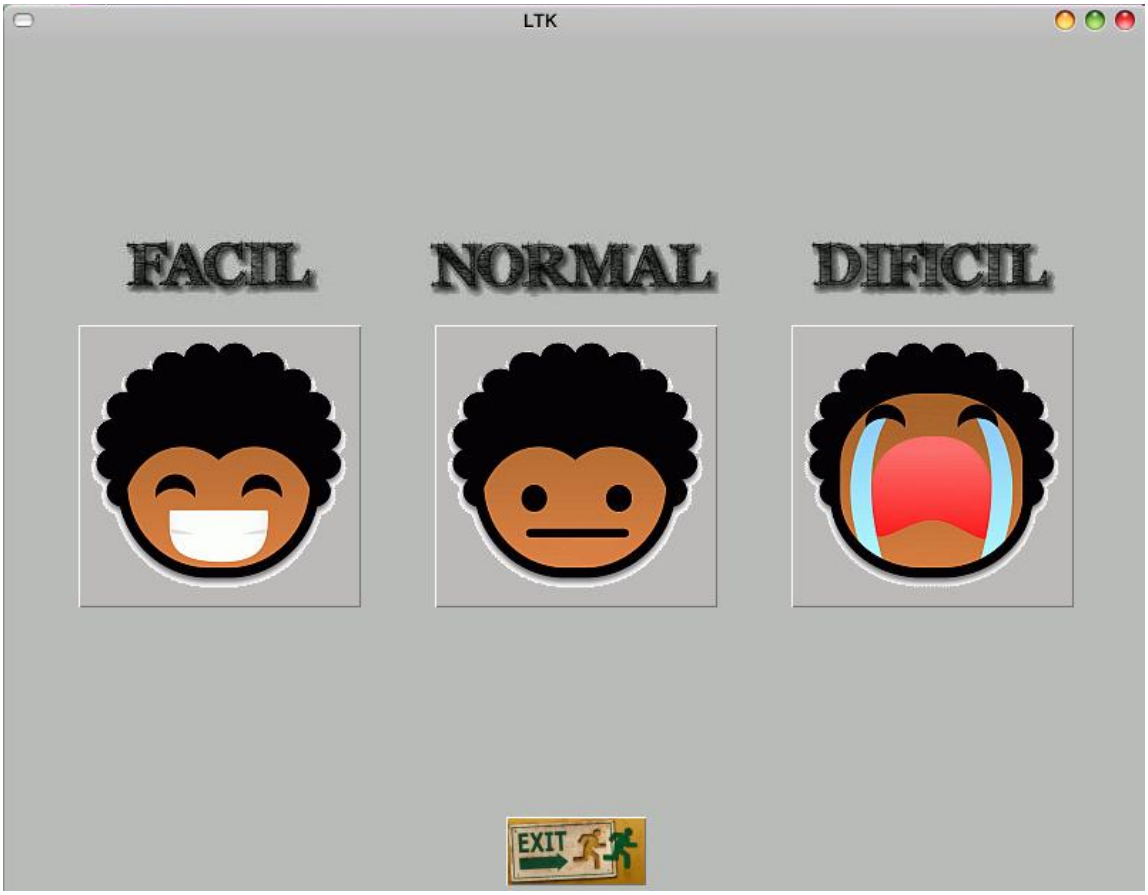


La pantalla de Selección de Modo nos da a elegir entre 3 opciones de juego:

- Juego entre un jugador humano y un jugador controlado por la computadora.
- Juego entre dos jugadores humanos.
- Juego entre dos jugadores controlados por la computadora. Este modo es especialmente útil para realizar una comparativa entre las heurísticas implementadas.

Como se ha explicado en la pantalla de Selección de Juego (y extensible a todas las pantallas con botones que nos encontraremos), una vez elegido el modo de juego, deberemos hacer clic en el área de pulsación del botón correspondiente.

Pantalla de Selección de Dificultad

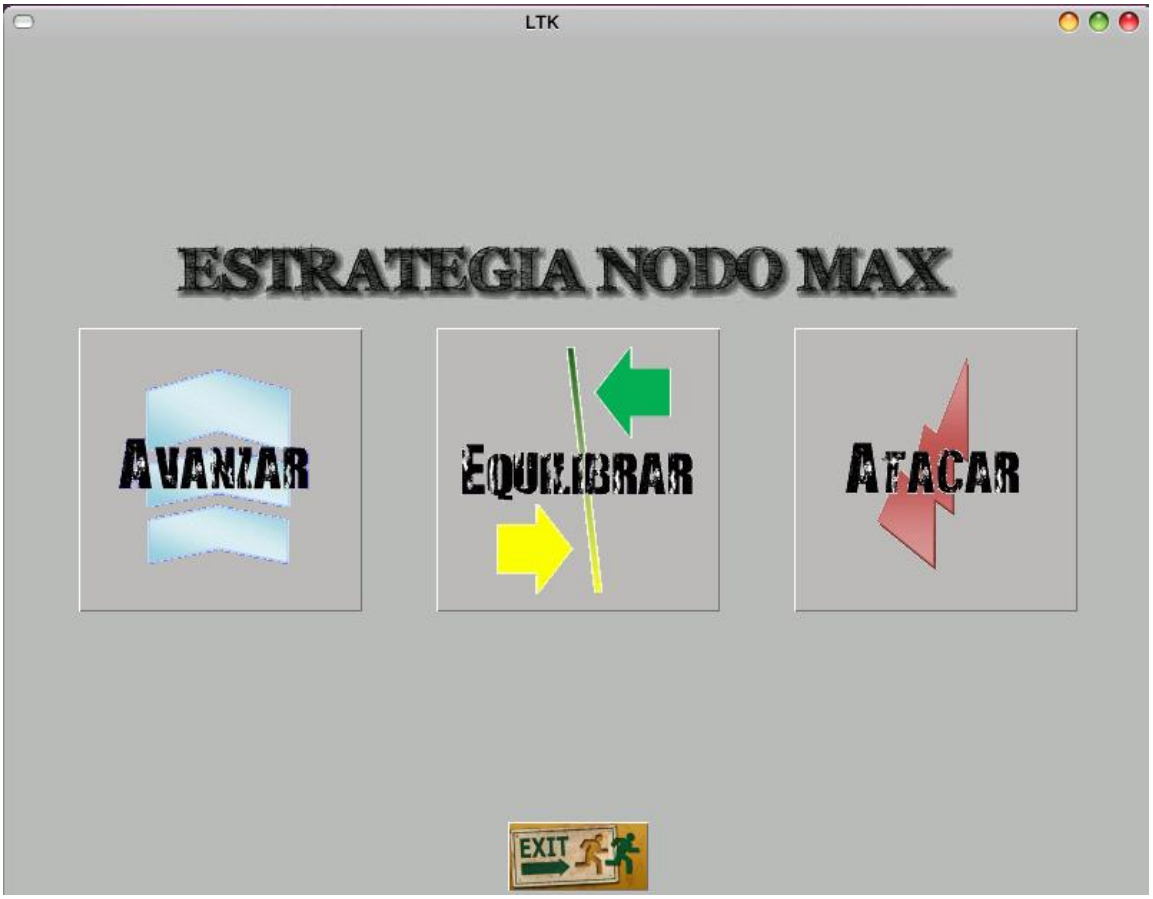


Tras la selección del modo de juego, vamos a conocer la pantalla de Selección de Dificultad. Se dispone de tres modos de dificultad para elegir: Fácil, Normal y Difícil.

Esta elección será especialmente importante en dos vertientes:

- Por una parte; a mayor dificultad, los movimientos de la computadora (en los modos de juego en los que participe) serán más óptimos, es decir, como diríamos coloquialmente, *"juega mejor"*.
- Por otra parte; cuando hagamos uso del botón Consejo (ver "Pantalla de Juego"), el movimiento que la computadora nos aconseja que hagamos en el turno actual, será más óptimo cuanto mayor dificultad hayamos elegido en esta pantalla.

Pantalla de Selección de Estrategia nodo MAX



Tras la selección el nivel de profundidad, vamos a conocer las pantallas de Selección de Estrategias. Se dispone de tres estrategias distintas para el nodo MAX, que según el modo de juego del adversario influirá en la dificultad para derrotar a la inteligencia artificial:

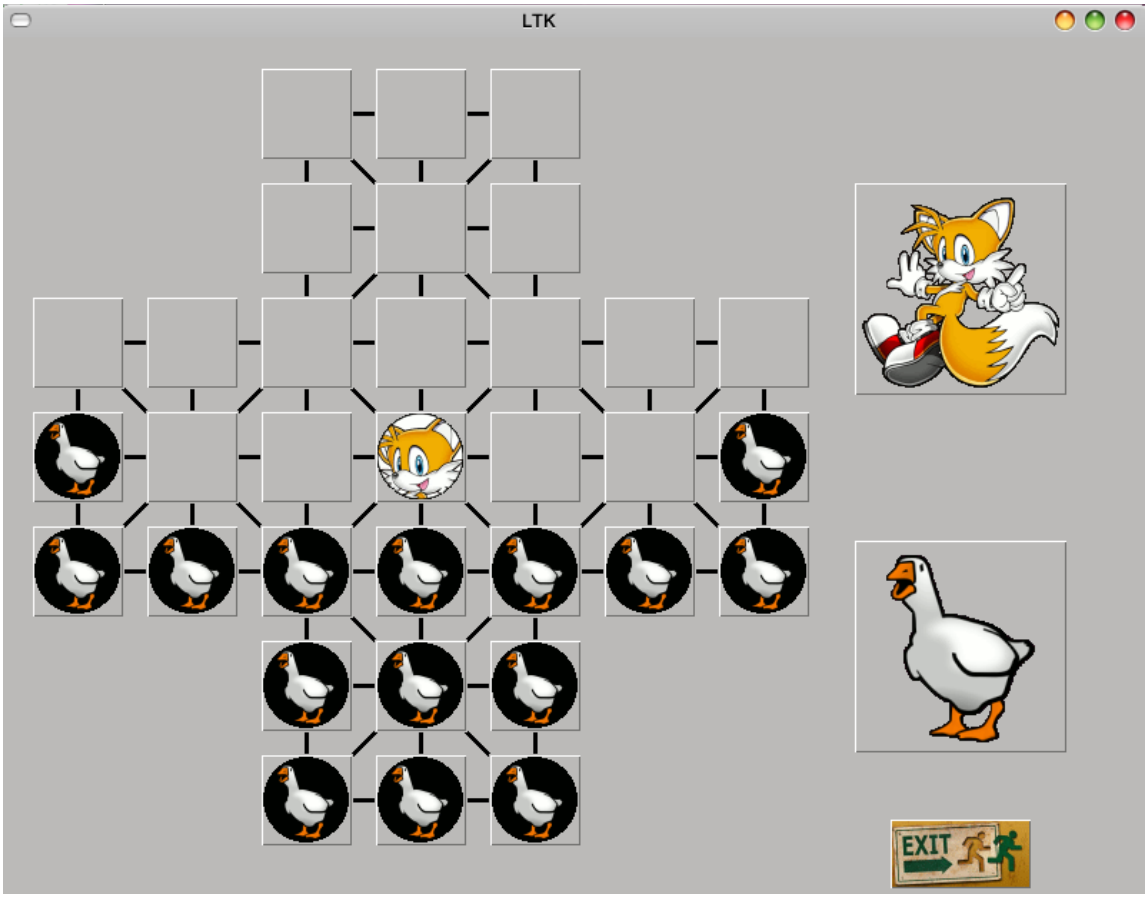
- Avanzar: Donde se ha ponderado la función lineal de la heurística para que la ficha que representa al nodo MIN busque la victoria preferentemente escogiendo los movimientos que más le aproximen al extremo contrario del tablero.
- Equilibrar: Donde los pesos de las distintas características de un estado se han ponderado de manera equilibrada.
- Atacar: Donde se ha ponderado la función lineal para que el jugador CPU ataque el flanco débil del jugador MAX. Por ejemplo, si el nodo MAX tiene una formación en 'V' intentar percutir su línea por el centro.

Pantalla de Selección de Estrategia nodo MIN

Para el nodo MIN disponemos igualmente de tres estrategias distintas:

- Avanzar: Donde se ha ponderado la función lineal de la heurística para que las fichas que representan al nodo MIN busquen la victoria preferentemente escogiendo los movimientos que más le aproximen al extremo contrario del tablero, evitando por supuesto que el jugador MAX rebase sus líneas.
- Equilibrar: Donde los pesos de las distintas características de un estado se han ponderado de manera equilibrada.
- Defender: Donde se ha ponderado la función lineal para que el jugador CPU mantenga su línea de avance cerrada, desplegando un juego más lento pero más seguro, con el propósito de evitar los huecos posibles y haciendo retroceder poco a poco a su adversario.

Pantalla de Selección de Facción

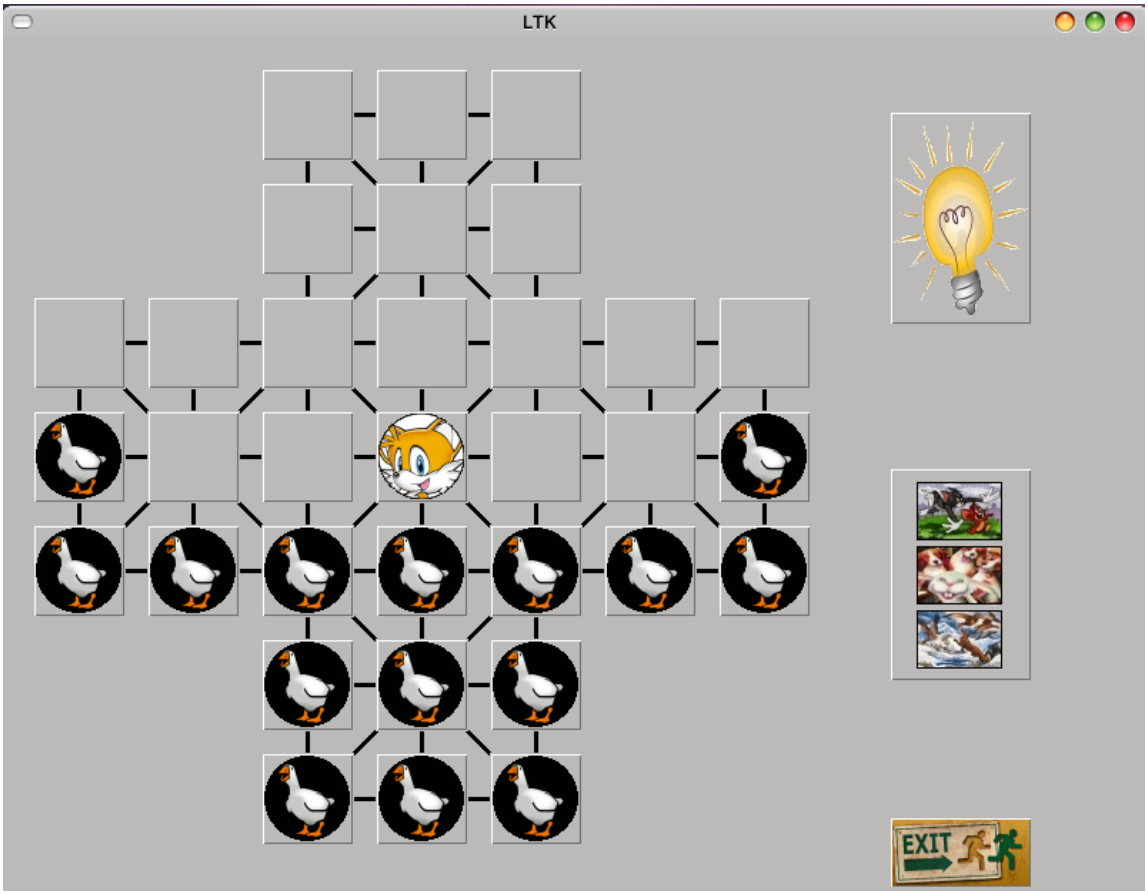


Ya elegido el nivel de dificultad, nos encontramos por primera vez con el tablero del juego elegido preparado con la colocación de fichas inicial y con la zona de Selección de Facción.

Como ya se ha comentado anteriormente los tres juegos implementados constan de dos facciones con diferente número de unidades y con distintos objetivos en la partida. En el ejemplo concreto anterior las dos facciones son la zorra y los gansos, la primera solo consta de 1 unidad mientras que los gansos tienen 15 unidades a su disposición.

En esta pantalla el usuario puede primero ver la disposición y número de las fichas de las facciones en el tablero y después debe elegir una de las dos facciones clicando en el área de pulsación del botón que representa a cada uno de ellos.

Pantalla de Juego



Una vez llegados a esta pantalla, la partida ha dado comienzo, y tenemos 4 acciones posibles:

- Pedir consejo de movimiento.
- Seleccionar la ficha que deseamos mover el turno actual.
- Volver al menú de selección de Juego.
- Abandonar la aplicación.

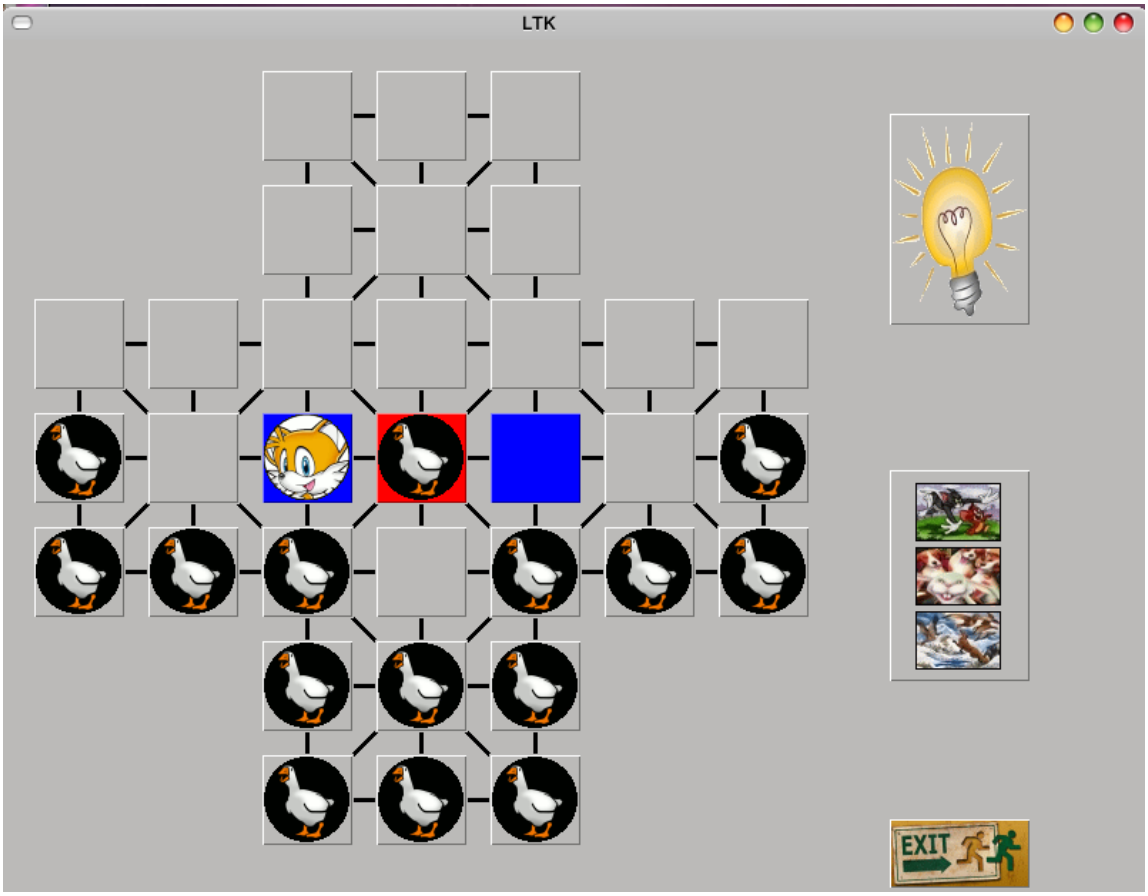
A continuación vamos a descubrir las respuestas del producto software ante la solicitud de cada una de las posibles acciones indicadas.

Pedir consejo de movimiento

El producto software tiene implementado un sistema de ayuda al jugador para cuando este se encuentre indeciso a la hora de efectuar un movimiento. Si el jugador desea utilizar este servicio, simplemente debe hacer clic en el botón de Consejo; identificable por tener una bombilla dibujada, y siempre antes de seleccionar que ficha desea mover.

La respuesta que obtendrá del sistema, será el movimiento óptimo calculado por la CPU para el estado actual de la partida y según el nivel de dificultad elegido en la pantalla de Selección de Dificultad. Durante 1 segundo se activaran las fichas que entren en juego en el movimiento sugerido según el siguiente código de colores:

- Las casillas origen y destino del movimiento de color **AZUL**.
- En el caso de que como consecuencia del movimiento una ficha sea retirada del juego, la casilla que contiene dicha ficha de color **ROJO**.

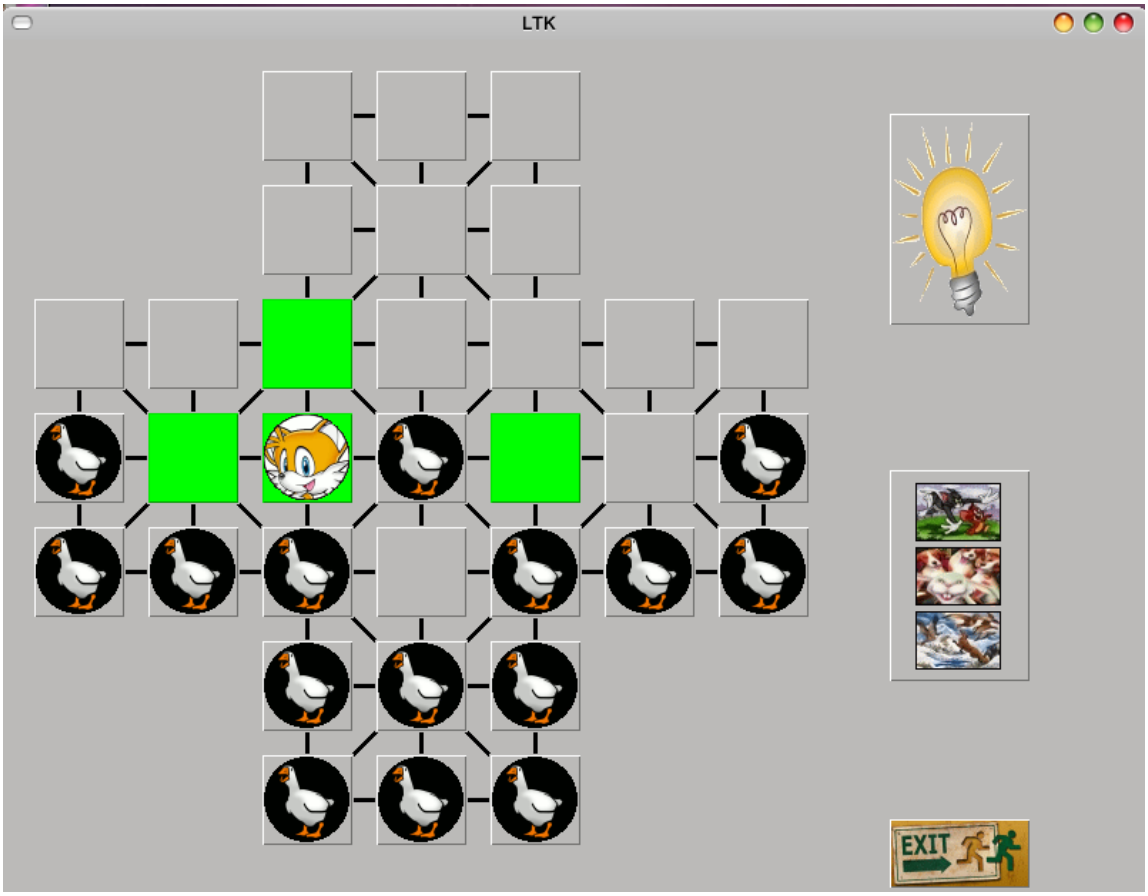


Seleccionar ficha del jugador activo para su movimiento

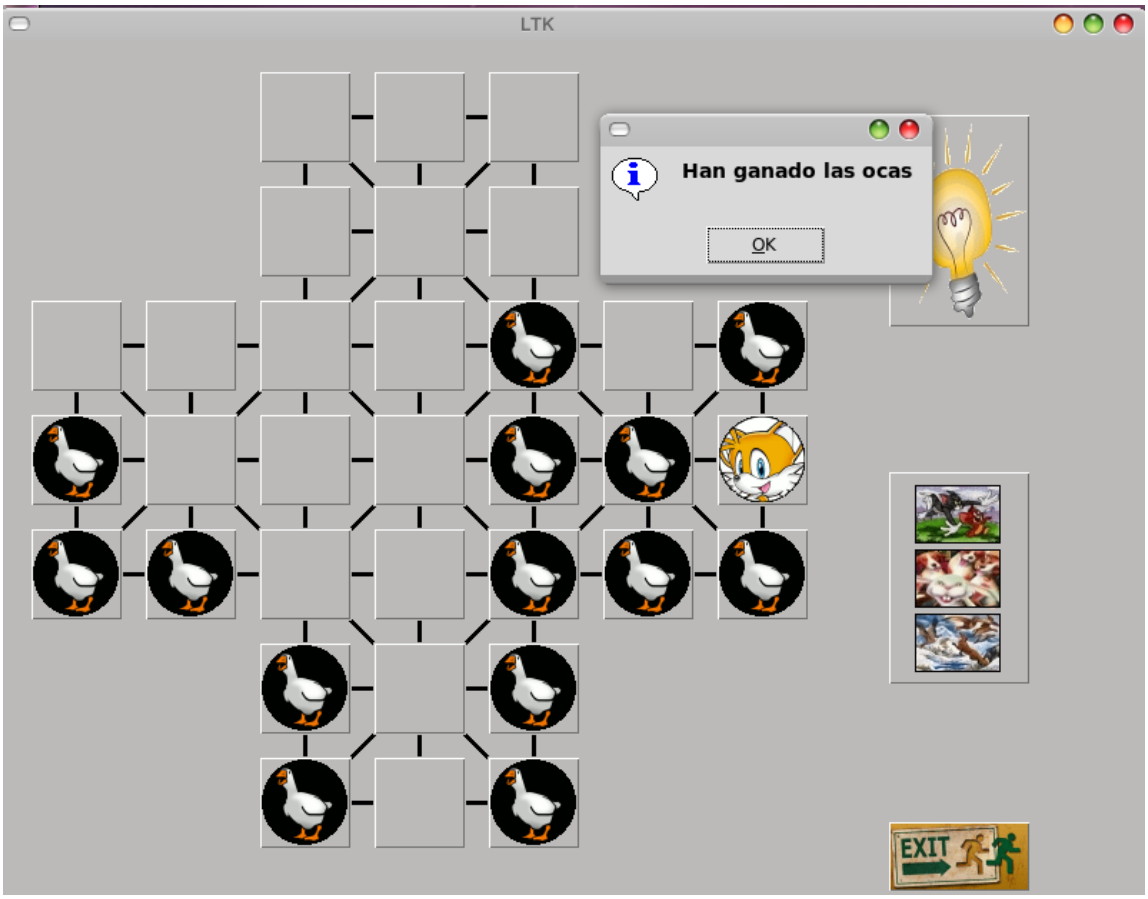
Una vez se ha decidido que ficha se desea mover, debemos hacer clic en su casilla indicando que esa es la ficha elegida para el movimiento en el turno actual. Si el jugador desea cambiar la ficha seleccionada (en el caso de que controle una de las facciones que disponen de un número de fichas igual o superior a 2), podrá realizarlo con solo clicar de nuevo en la ficha deseada.

El sistema implementa un sistema de ayuda de movimientos válidos, de forma que la respuesta a la selección de una ficha para su movimiento es la activación en color **VERDE** de la casilla origen del movimiento y las posibles casillas destino válidas.

Eligiendo cualquiera de las casillas destino activadas, el movimiento se completará (eliminando del tablero las casillas capturadas en caso de haberlas) y terminará el turno del jugador activo, dando paso al comienzo del turno del oponente.



De esta forma el juego continúa hasta que se den alguna de las condiciones de finalización de la partida, en cuyo caso se indicará en una ventana emergente el ganador de la partida y se deshabilitarán todos los botones excepto el Botón Salir para abandonar la aplicación y el botón de Menú de Selección de juego en caso de que deseemos comenzar una nueva partida.

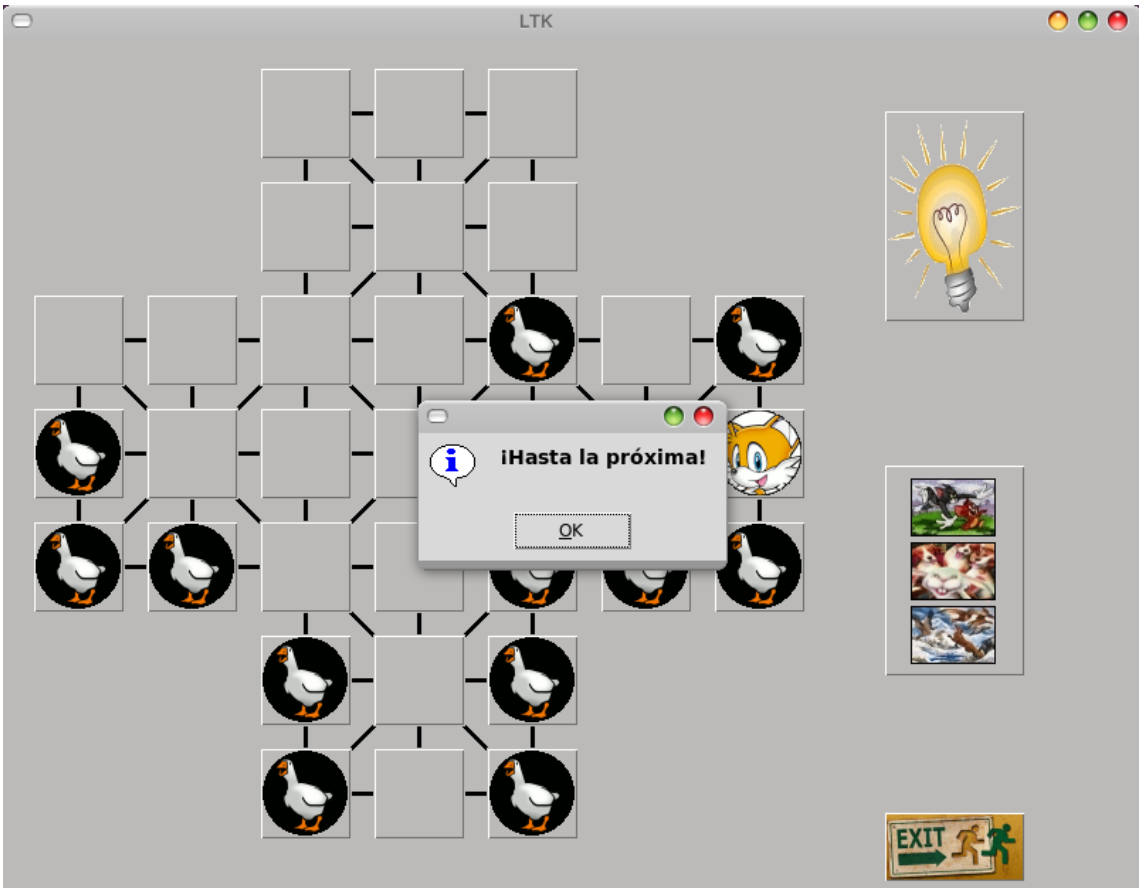


Volver al menú de Selección de Juego

Si por cualquier motivo deseamos iniciar una nueva partida durante, o una vez finalizada, la partida en curso, simplemente debemos pulsar el botón de Menú de selección de juego, que permanece activo durante toda la partida (identificable por contener los 3 iconos de los juegos implementados).

Abandonar la aplicación

Cuando hayamos terminado la partida o en cualquier otro momento podemos hacer uso del botón Salir para abandonar la aplicación (¡con la despedida y deseo de que la experiencia le haga repetir!).



13. Comparación con otras alternativas

¿Por qué elegí Lisp como lenguaje de programación?

Interpretación y compilación

Los lenguajes de programación pueden, en líneas generales, dividirse en dos categorías:

- Lenguajes interpretados
- Lenguajes compilados

Lenguaje interpretado

Un lenguaje de programación es, por definición, diferente al lenguaje máquina. Por lo tanto, debe traducirse para que el procesador pueda comprenderlo. Un programa escrito en un lenguaje interpretado requiere de un programa auxiliar (el intérprete), que traduce los comandos de los programas según sea necesario.

Lenguaje compilado

Un programa escrito en un lenguaje "compilado" se traduce a través de un programa anexo llamado compilador que, a su vez, crea un nuevo archivo independiente que no necesita ningún otro programa para ejecutarse a sí mismo. Este archivo se llama ejecutable.

Un programa escrito en un lenguaje compilado posee la ventaja de no necesitar un programa anexo para ser ejecutado una vez que ha sido compilado. Además, como sólo es necesaria una traducción, la ejecución se vuelve más rápida.

Sin embargo, no es tan flexible como un programa escrito en lenguaje interpretado, ya que cada modificación del archivo fuente (el archivo comprensible para los seres humanos: el archivo a compilar) requiere de la compilación del programa para aplicar los cambios.

Por otra parte, un programa compilado tiene la ventaja de garantizar la seguridad del código fuente. En efecto, el lenguaje interpretado, al ser directamente un lenguaje legible, hace que cualquier persona pueda conocer los secretos de fabricación de un programa y, de ese modo, copiar su código o incluso modificarlo. Por lo tanto, existe el riesgo de que los derechos de autor no sean respetados. Por otro lado, ciertas aplicaciones aseguradas necesitan confidencialidad de código para evitar las copias ilegales (transacciones bancarias, pagos en línea, comunicaciones seguras...).

Compilación vs. Interpretación

El intérprete es notablemente más lento que el compilador, ya que realiza la traducción al mismo tiempo que la ejecución. Además, esa traducción se lleva a cabo siempre que se ejecuta el programa, mientras que el compilador sólo la hace una vez. Por estos motivos, un mismo programa interpretado y compilado se ejecuta mucho más despacio en el primer caso.

La ventaja de los intérpretes es que hacen que los programas sean más portables. Así, un programa compilado en una máquina PC bajo Windows no funcionará en un Macintosh, o en un PC bajo Linux, a menos que se vuelva a compilar el programa fuente en el nuevo sistema. En cambio, un programa interpretado funcionará en todas las plataformas, siempre que dispongamos del intérprete en cada una de ellas.

Comparando su actuación con la de un ser humano, un compilador equivale a un traductor profesional que, a partir de un texto, prepara otro independiente traducido a otra lengua, mientras que un intérprete corresponde al intérprete humano, que traduce de viva voz las palabras que oye, sin dejar constancia por escrito.

Es decir: los lenguajes compilados no son mejores que los interpretados, ni al revés. Optar por uno u otro depende de la función para la que vayamos a escribir el programa y del entorno donde deba ejecutarse.

Lenguajes intermediarios

Algunos lenguajes como Lisp pertenecen a ambas categorías dado que el programa escrito en estos lenguajes puede, en ciertos casos, sufrir una fase de compilación intermediaria, en un archivo escrito en un lenguaje ininteligible (por lo tanto diferente al archivo fuente) y no ejecutable (requeriría un intérprete).

La flexibilidad que tiene Lisp, por ser un lenguaje intermediario, de aprovechar las ventajas de los lenguajes compilados e interpretados según nos convenga, junto a que Lisp es el lenguaje de programación de Inteligencia Artificial por antonomasia; fueron dos de los motivos decisorios para que este proyecto sea implementado en Lisp.

Algunos ejemplos de lenguajes ampliamente usados

A continuación, encontrará una breve lista de los lenguajes de programación actuales:

Lenguaje	Principal área de aplicación	Compilado/interpretado
ADA	Tiempo real	Lenguaje compilado
BASIC	Programación para fines educativos	Lenguaje interpretado
C	Programación de sistema	Lenguaje compilado
C++	Programación de sistema orientado a objeto	Lenguaje compilado
Cobol	Administración	Lenguaje compilado
Fortran	Cálculo	Lenguaje compilado
Java	Programación orientada a Internet	Lenguaje compilado
MATLAB	Cálculos matemáticos	Lenguaje interpretado
LISP	Inteligencia artificial	Lenguaje intermediario
Pascal	Educación	Lenguaje compilado
PHP	Desarrollo de sitios web dinámicos	Lenguaje interpretado
Perl	Procesamiento de cadenas de caracteres	Lenguaje interpretado

14. Conclusiones

El diseño y construcción de juegos clásicos de tablero no resulta ser sencillo debido a la gran variedad de posibilidades que se tienen con respecto a los movimientos. Sin embargo, es un área interesante para aplicar las metodologías de la inteligencia artificial y la programación evolutiva. Aquí se ha desarrollado una aplicación software capaz de jugar a tres juegos de forma competitiva, creando una implementación propia del algoritmo recursivo minimax para la toma de decisiones en juegos. Los resultados han sido totalmente satisfactorios, se ha conseguido dotar a la computadora de una inteligencia artificial capaz de derrotar a un jugador humano medio, llegando hasta una profundidad en el árbol de búsqueda de hasta 9 nodos en unos segundos, que teniendo en cuenta el factor de ramificación de los juegos, no hace más que confirmar la eficiente implementación que se le ha dado a la aplicación.

Así como también se han completado los objetivos inicialmente marcados, tanto a nivel personal como de conocimientos adquiridos, ha resultado ser una experiencia totalmente enriquecedora: el plasmar con éxito lo que inicialmente solo era una idea, la resolución de los problemas a los que enfrentarse durante el desarrollo, la depuración de los errores, las correcciones y mejoras que surgen a medida que se progresa en el ciclo de vida del software, no hacen sino aumentar la madurez del Ingeniero Informático que desarrolla este tipo de producto.

Además, no solo se han cumplido los requisitos sino que, en parte, se han superado, aportando al motor de búsqueda minimax que es el eje del proyecto nuevas opciones, como el sistema de escritura de estados en un fichero de texto plano o el uso de consejos de la CPU sobre el próximo movimiento a realizar.

Ya se ha comentado que este proyecto tiene un fuerte componente de investigación. Son muchos los frentes que quedan abiertos, y en los que futuros desarrollos pueden ahondar. La idea más interesante es precisamente el sistema de grabación de partidas.

Así, se puede diseñar una red neuronal de aprendizaje automático que se retroalimente con el fichero de texto que genera como salida. Podría ser capaz de detectar las situaciones que más se repiten en las partidas, con el objetivo de cambiar la distribución de pesos de la función ponderada de evaluación según el resultado final de dichas partidas. Entonces, y gracias al modo CPU vs CPU se podría crear una arquitectura que mejore la inteligencia artificial de la computadora con la ejecución de partidas sucesivas o incluso que sea capaz de hacer el reparto de pesos simultáneamente a la ejecución de una partida.

Solo este desarrollo constituiría un Proyecto Fin de Carrera por sí solo, pero no deja de ser una idea a explotar enormemente atrayente y que aportaría al campo de la investigación de inteligencia artificial un soplo de aire fresco, ya que es un campo donde hay todavía mucho que entender y comprender.

15. Bibliografía

Russell, Stuart J.; Norvig, Peter: *Inteligencia artificial: un enfoque moderno*. Prentice-Hall Hispanoamericana, 1996. Colección de Inteligencia Artificial de Prentice Hall, XXVII, 979 p., graf. ; 24 cm. ISBN 968-880-682-X.

Witten, Ian H.; Frank, Eibe: *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Series in Data Management, Second edition, 558 p. ISBN 0120884070.

Herth, Peter: LTK - a Lisp binding to the Tk toolkit. Version 0.91, 29 Enero 2006.

Alonso Jiménez, José Antonio; Martín Mateos, Francisco Jesús; Ruiz Reina, José Luis: *Una introducción al lenguaje Lisp.* Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Sevilla.