Reporte Juego Othello

Carlos Roberto Flores Luna

INTRODUCCION

Resolviendo Problemas con Agentes

Vemos como un agente puede encontrar una secuencia de acciones que logra su objetivos cuando ninguna acción individual lo hará.
Describiremos un tipo de **agente basado en objetivos llamado agente de resolución de problemas** que usa representaciones atómicas, tomando los estados del mundo sin una estructura interna visible para el problema.
La resolución de problemas comienza con definiciones precisa de los problemas y sus soluciones. Ocuparemos Algoritmos de búsqueda para resolver estos problemas, es importante diferenciar **Algoritmos de búsqueda no Informados** (algoritmos que no reciben información sobre el problema aparte de su definición) y **Algoritmo de búsqueda informados** (Se da orientación sobre dónde buscar soluciones)

Suponemos que un agente inteligente busca maximizar sus medidas de rendimiento, ¿Como un agente podría hacer esto?
Se crea una rama de posibles resultados a la resolución de un problema, se pone un peso determinado a los objetivos, los objetivos ayudan a organizar el comportamiento que el agente intenta alcanzar. Al tener múltiples objetivos el número de acciones se puede agrandar demasiado, para esto el agente determinara que acciones maximizan el conjunto de objetivos. Al tomar una acción el agente habrá analizado las acciones futuras que lo lleven a la maximización.

No olvidemos que para que un agente interactúe adecuadamente tenemos que tener bien definido un entorno, sí es observable, discreto, determinista, etc...

Bajo los supuesto anteriores la solución de cualquier problema para ser una secuencia fija de acciones. En general puede ser una ramificación que recomiende diferentes acciones en el futuro dependiendo de qué percepciones lleguen.

El proceso de búsqueda de una secuencia de acciones que alcanza la meta se llama búsqueda. Un algoritmo de búsqueda toma un problema como entrada y devuelve una solución en forma de acción secuencia. Una vez que se encuentra una solución, se pueden llevar a cabo las acciones que recomienda. Esto se llama **fase de ejecución**. Por lo tanto, tenemos un diseño simple **formular -> buscar -> ejecutar** para el agente.

Después de formular la meta y un problema para resolver, el agente llama a un procedimiento de búsqueda para resolverlo. Usando la solución para guiar sus acciones de manera secuencial, eliminando los pasos anteriores. Una vez que la solución ha sido ejecutada, el agente formulará un nuevo objetivo.

Definiendo un problema y sus soluciones, podemos definir un problema formalmente por cinco componentes. Estado inicial, Posibles Acciones, Modelo de transición de Acciones, Estado de Objetivos, Función de Costo Acciones.

Los elementos anteriores definen un problema y pueden reunirse en una única estructura de datos, eso se da como entrada a un algoritmo de resolución de problemas. Una solución a un problema son acciones secuenciales que van de un estado inicial a un estado objetivo. La calidad de la solución se mide por la función de costo de ruta (solución óptima tiene el costo de ruta más bajo entre las soluciones)

El proceso de eliminar detalles de una representación de un problema se llama abstracción. Además de abstraer las descripción de estados debemos abstraer las acciones mismas. La elección de una buena abstracción implica eliminar tanto detalle como sea posible, manteniendo la validez y asegurando que las acciones abstractas sean fáciles de llevar a cabo. Si no fuera por la capacidad de construir abstracciones útiles, los agentes inteligentes serían completamente abrumadores para el mundo real.

Estrategia de búsqueda Informada

En **computación**, dos objetivos fundamentales son encontrar **algoritmos** con buenos tiempos de ejecución y buenas soluciones, usualmente las óptimas. Una **heurística** es un algoritmo que abandona uno o ambos objetivos; por ejemplo, normalmente encuentran buenas soluciones, aunque no hay pruebas de que la solución no pueda ser arbitrariamente errónea en algunos casos; o se ejecuta razonablemente rápido, aunque no existe tampoco prueba de que siempre será así. Las heurísticas generalmente son usadas cuando no existe una solución óptima bajo las restricciones dadas (tiempo, espacio, etc.), o cuando no existe del todo.

A menudo, pueden encontrarse instancias concretas del problema donde la heurística producirá resultados muy malos o se ejecutará muy lentamente. Aun así, estas instancias concretas pueden ser ignoradas porque no deberían ocurrir nunca en la práctica por ser de origen teórico. Por tanto, el uso de heurísticas es muy común en el mundo real.

Para problemas de búsqueda del camino más corto el término tiene un significado más específico. En este caso una heurística es una **función matemática, h(n)** definida en los nodos de un **árbol de búsqueda** , que sirve como una estimación del coste del camino más económico de un nodo dado hasta el nodo objetivo. Las heurísticas se usan en los algoritmos de búsqueda informada como la búsqueda egoísta. La búsqueda egoísta escogerá el nodo que tiene el valor más bajo en la función heurística. A* expandirá los nodos que tienen el valor más bajo para g(n) + h(n), donde g(n) es el coste (exacto) del camino desde el estado inicial al nodo actual. Cuando **h(n) es admisible**, esto es si h(n) nunca sobrestima los costes de encontrar el objetivo; A* es probablemente óptimo.

Un problema clásico que usa heurísticas es el puzzle-n. Contar el número de casillas mal colocadas y encontrar la suma de la distancia Manhattan entre cada bloque y su posición al objetivo son heurísticas usadas a menudo para este problema.

Efectos de las heurísticas en el rendimiento computacional.

En cualquier problema de búsqueda donde hay *b* opciones en cada nodo y una profundidad *d* al nodo objetivo, un algoritmo de búsqueda ingenuo deberá buscar potencialmente entre b^d nodos antes de encontrar la solución. Las heurísticas mejoran la eficiencia de los algoritmos de búsqueda reduciendo el factor de ramificación de b a (idealmente) una constante b*.

Aunque cualquier heurística admisible devolverá una respuesta óptima, una heurística que devuelve un factor de ramificación más bajo es computacionalmente más eficiente para el problema en particular. Puede demostrarse que una heurística $h_2(n)$ es mejor que otra $h_1(n)$, si $h_2(n)$ domina $h_1(n)$, esto quiere decir que $h_1(n) < h_2(n)$ para todo n.

Heurísticas en la inteligencia artificial

Muchos algoritmos en la inteligencia artificial son heurísticos por naturaleza, o usan reglas heurísticas. Un ejemplo reciente es SpamAssassin que usa una amplia variedad de reglas heurísticas para determinar cuando un correo electrónico es spam. Cualquiera de las reglas usadas de forma independiente pueden llevar a errores de clasificación, pero cuando se unen múltiples reglas heurísticas, la solución es más robusta y creíble. Esto se llama alta credibilidad en el reconocimiento de patrones (extraído de las estadísticas en las que se basa). Cuando se usa la palabra heurística en el procesamiento del lenguaje basado en reglas, el reconocimiento de patrones o el procesamiento de imágenes, es usada para referirse a las reglas.

DEFINICIÓN DEL PROBLEMA

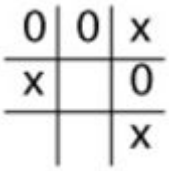
Buscando implementar un Algoritmo de Búsqueda informada al juego de Othello.
Con el fin de que la computadora sea capaz de jugar con un usuario, en distintos niveles de dificultad. Antes de detallar nuestra solución y trabajo se dará una pequeña explicación sobre el algoritmo minimax, se definirá un Tipo de Agente, REAS, Entorno de trabajo y complejidad del juego.

¿Qué es algoritmo minimax?

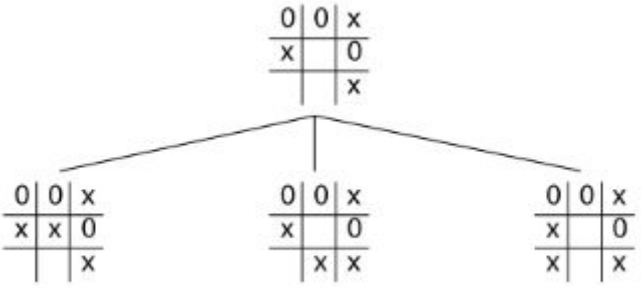
El algoritmo de minimax en simples palabras consiste en la elección del mejor movimiento para el computador, suponiendo que el contrincante escogerá uno que lo pueda perjudicar, para escoger la mejor opción este algoritmo realiza un árbol de búsqueda con todos los posibles movimientos, luego recorre todo el árbol de soluciones del juego a partir de un estado dado, es decir, según las casillas que ya han sido rellenadas. Por tanto, minimax se ejecutará cada vez que le toque mover a la IA.

En el algoritmo Minimax el espacio de búsqueda queda definido por:

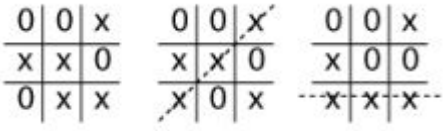
Estado inicial: Es una configuración inicial del juego, es decir, un estado en el que se encuentre el juego. Para nuestro ejemplo sería:



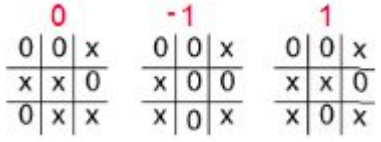
Operadores: Corresponden a las jugadas legales que se pueden hacer en el juego, en el caso del tres en raya no puedes marcar una casilla ya antes marcada.



Condición Terminal: Determina cuando el juego se acabó, en nuestro ejemplo el juego termina cuando un jugador marca tres casillas seguidas iguales, ya se horizontalmente, verticalmente o en diagonal, o se marcan todas las casillas (empate) .



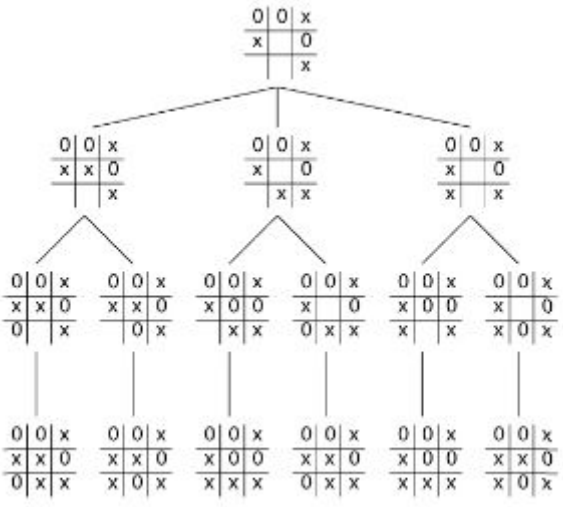
Función de Utilidad: Da un valor numérico a una configuración final de un juego. En un juego en donde se puede ganar, perder o empatar, los valores pueden ser 1, 0, o -1



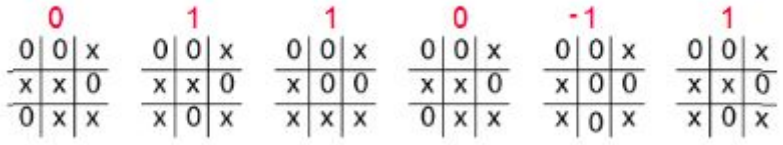
.

Implementación Minimax: Los pasos que sigue minimax pueden variar, pero lo importante es tener una idea clara de cómo es su funcionamiento, los pasos a seguir son:

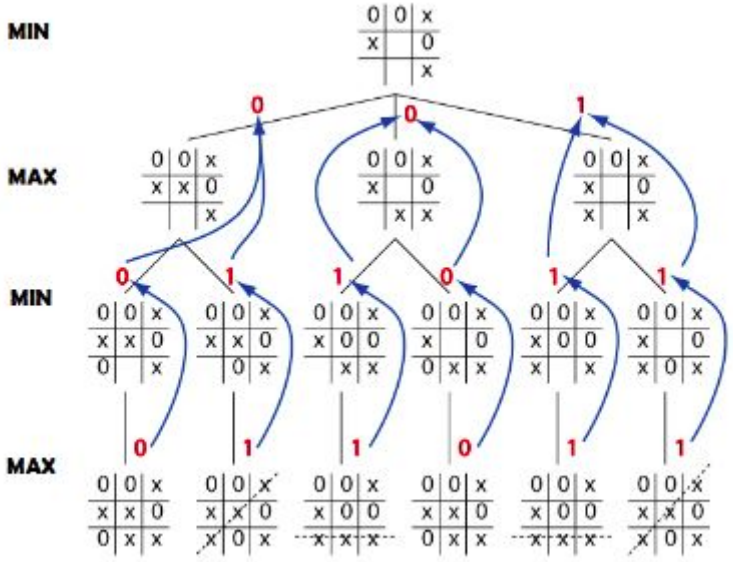
El algoritmo primero generar un árbol de soluciones completo a partir de un nodo dado. veamos el siguiente ejemplo:



Para cada nodo final, buscamos la función de utilidad de estos. En nuestro ejemplo usaremos un 0 para las partidas que terminen en empate, un 1 para las que gane la IA y un -1 para las que gane el jugador humano.



Y lo que hará el algoritmo Minimax cuando vaya regresando hacia atrás, será comunicarle a la llamada recursiva superior cuál es el mejor nodo hoja alcanzado hasta el momento. Cada llamada recursiva tiene que saber a quién le toca jugar, para analizar si el movimiento realizado pertenece a la IA o al otro jugador, ya que cuando sea el turno de la IA nos interesa MAXIMIZAR el resultado, y cuando sea el turno del rival MINIMIZAR su resultado.



Al final el algoritmo nos devolverá la jugada que debe realizar la máquina para maximizar sus posibilidades y bloquear las posibilidades del rival.

¿Cuáles son sus limitaciones?

Complejidad exponencial, ¡no se puede reducir a polinómica!

Esto lleva a que el algoritmo sea muy lento y en la práctica el método es impracticable. (A menos que sean supuestos sencillos). Realizar la búsqueda completa del árbol requiere tiempo y memoria.

Para optimizar minimax puede limitarse la búsqueda por nivel de profundidad o por tiempo de ejecución. Otra posible técnica es el uso de la poda alfa-beta. Esta optimización se basa en evitar el cálculo de ramas cuya evaluación final no va a poder superar los valores previamente obtenidos.

Tipo de Agente, REAS, Entorno de Trabajo

Agente basado en objetivos (Observa como es el estado actual y analiza las acciones eligiendo la que mejor le convenga)

REAS Othello	
Rendimiento	Identificación de jugadas válidas (pistas para el usuario, jugadas válidas).
	Ejecuta heurística implementada (Algoritmo Minimax, poda alfa-beta).
Entorno	Tablero de juego.
	Reglas del juego.
	Elegir dificultad.
Actuadores	Colocar fichas adecuadamente en el tablero.
	Voltear fichas mías y oponente al terminar una jugada.
Sensores	Estado del tablero (matrices con información del juego).
	Identificación de jugada del usuario.

Propiedades del Entorno Othello	
Observable	Totalmente observable, los sensores del agente proporcionan acceso al estado completo del entorno.
Deterministico	El siguiente estado del entorno está determinado por el estado actual.
Episodico	Se divide en episodios atómicos independientes, donde cada episodio consiste en la percepción del agente y la realización de una única acción posterior.
Estatico	El entorno no cambia mientras el agente delibera.
Discreto	EL entorno tiene un número finito de estados distintos.
Individual	Un solo agente resolviendo el problema.

Complejidad en el Juego

Los programas de Otelo buscan los posibles movimientos legales en un **Árbol de Juego**. En teoría examina todas las posiciones/nodos, donde cada movimiento de un jugador es llamado una jugada ("ply"). Esta búsqueda continúa hasta que se alcance cierta profundidad máxima o hasta que el programa determine que una posición final “hoja” ha sido alcanzada. Una implementación muy simple de esta aproximación, conocida como Minimax o Negamax, solo puede buscar a una profundidad pequeña en un tiempo razonable, por tanto se han elaborado varios métodos para acelerar la búsqueda de buenos movimientos. Estos se basan en Poda alfa-beta. Para reducir el tamaño del árbol de búsqueda también se utilizan varias heurísticas como: ordenamiento de los tableros hijos, tablas de transposición, y búsqueda selectiva. Para acelerar la búsqueda en máquinas con múltiples procesadores o núcleos, se puede implementar una “búsqueda en paralelo”.

La complejidad del juego othello es **PSPACE-completo.** Los problemas en PSPACE-completo pueden verse como los problemas más difíciles de la clase PSPACE. Se sospecha fuertemente que estos problemas están fuera de las clases de complejidad P y NP, pero no hay prueba de ello. Se sabe que no están contenidos en la clase NC. Matemáticamente tienen una complejidad exponencial por el número de ramificaciones que el propio juego va creando.

Definiendo el Estado-Espacio y el tamaño del árbol de decisiones se tienen aproximaciones, con la función **O(b^d)** donde **b = Factor de Ramificación, d = Profundidad del Árbol.**

El número de posibles partidas se puede calcular aproximadamente teniendo en cuenta la movilidad media 8 y el número de movimientos totales 60.

Suponiendo una movilidad media de 8, esto quiere decir que en un momento la partida podemos tomar 8 diferentes caminos para continuar, en el siguiente turno hay otras 8 ramificaciones para cada una de las 8 anteriores (lo que daría 8x8 = 64 partidas diferentes). Las ramificaciones siguen en cada turno, y el número de partidas posibles se obtienen multiplicando por 8 tantas veces como turnos hay (60) es decir **8^60 = 10^54** posibles partidas.

Con esto podemos decir que la dificultad de nuestra IA para jugar con una persona estará dada por el número de jugadas (ramificaciones) que puede analizar nuestro programa. Con b = 8.

- Novato d = 20 implica 8^20 posibles partidas (Minimax)
- Intermedio d = 40 implica 8^40 (Poda alfa beta)
- Avanzado d = 60 implica 8^60 (Poda alfa beta)

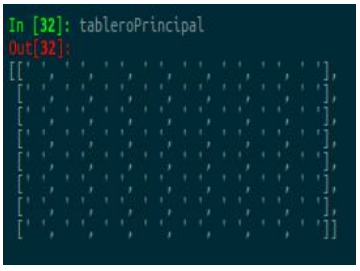
DESCRIPCIÓN DE LA PROPUESTA E IMPLEMENTACIÓN

IMPLEMENTACIÓN DEL JUEGO

El juego se desarrollo en python 3.7 en un mismo script.
Su interfaz gráfica es la terminal.
La estructura general del código es la siguiente:

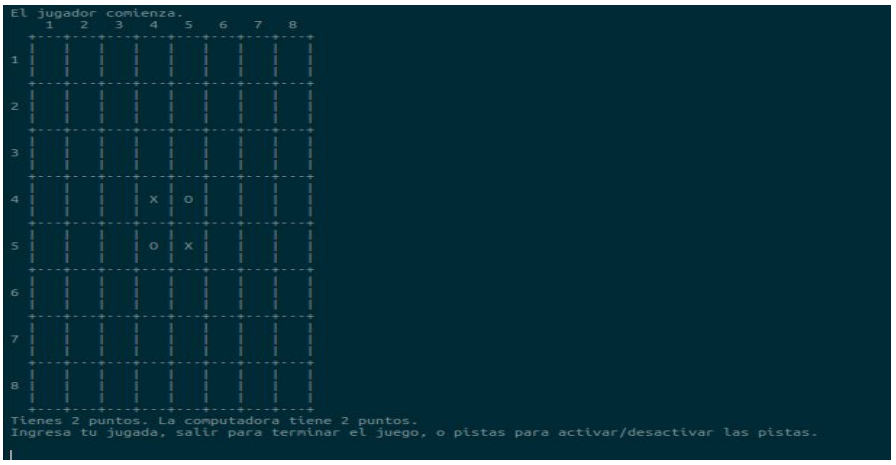
- Estructura del tablero de juegos (dibujar,reiniciar,nuevo tablero)
- Funciones de Interacción (Obtención del puntaje, obtención de baldosas del jugador, determinación de quien comienza el juego,preguntar al jugador si desea jugar de nuevo, mostrar puntajes actuales)
- Funciones para el desarrollo del juego (hacer Jugadas, obtener copias del tablero, obtener jugadas del jugador)
- Funciones para validar jugadas y obtener jugadas disponibles para el siguiente estado del juego
- IA del juego (búsqueda informada), análisis de las mejores jugadas para la máquina en diferentes niveles

El motor del tablero es una lista de listas de 8x8.



Aquí se van guardando las jugadas y se aplican las diferentes funciones definidas para ir avanzando en el juego

La visualización del juego nos arroja el puntaje, nos permite colocar nuestra jugada primero la coordenada x(columnas) y luego la y(filas).



Profundizaremos sobre la heurística aplicada.

Tomando la idea del análisis de las posibles jugadas a realizar con el juego gato, el análisis es el mismo para el juego othello con la diferencia que trabajamos sobre un tablero de 8x8 y con una mayor complejidad en el juego.

Con un punto podemos ver las posibles jugadas que se pueden realizar en el tablero.

¿Cómo funciona la ejecución de las jugadas?

Primero validamos que la jugada sea válida (función esJugadaValida()):
1.- Que pertenezca al tablero.
2.- Al Menos convierta una pieza a favor del jugador en turno.

Segundo, obtenemos las posibles jugadas que se pueden realizar en el estado actual (función obtenerJugadasVálidas()):
1.- Obtenemos una lista de jugadas válidas.

2.- Obtenemos el tablero con las jugadas válidas.

Tercero, Volteamos fichas (función hacerJugada()):
1.- Para cada jugada calculamos el número de fichas que voltea.
2.- Se arroja una matriz con todas las posibles jugadas.

Cuarto, jugadas jugador o computadora (funciones obtenerJugadaJugador() y obtenerJugadaComputadora()):
1.- Si es turno del jugador puede habilitar el modo “pistas”, para conocer las jugadas disponibles
2.- Ejecuta la jugada dependiendo del nivel de dificultad que esté indicado

La idea era determinar 3 niveles de dificultad, solo se encuentran implementados los 2 primero niveles.

Novato: Toma una jugada válida de forma aleatoria
Intermedio: Itera sobre las jugadas válidas, contando el número de fichas que cada jugada voltea, al final toma la jugada que mayor número de fichas haya volteado.
Experto: Implementa el algoritmo Minimax con poda alfa beta, crea una ramificación de las jugadas que permitan ganar y hace una poda alfa - beta.

La máquina juega a ganar, ya que siempre está buscando la mejor jugada

CONCLUSIONES

La solución de un problema por medio de heurísticas es bastante importante en nuestros días, ya que permiten modelar y dar solución a múltiples problemas.
Una ventaja importante al implementar este tipo de algoritmos es que un humano puede hacer que una máquina simule el comportamiento lógico de una persona.
El trabajo matemático sobre gráficas es muy interesante, tenemos distintas formas de modelar y optimizar las entradas y salidas. Por ejemplo Metodo Simplex, Ruta más Corta, Ruta Optima, Random Forest. Me gusto conocer otro tipo de algoritmo que permite simular una IA.
Siempre es importante cuidar la complejidad computacional ya que esto permite la eficiencia de nuestros ordenadores para resolver problemas.

REFERENCIAS BIBLIOGRÁFICAS

-Paradigmas de programación de inteligencia artificial de Peter Norvig.
-Stuart Rusell and Peter Norvig, Artificial Intelligence: A Modern Approach', 3rd Edition, Prentice Hall, 2009.
-Ernest Davis, Representations of Commonsense Knowledge, Morgan Kaufmann Pub, 1990.