Duccio Profeti

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
Facultad de Ingeniería
IIC2613 Inteligencia Artificial
2024-1

Tarea 3

Problemas de Búsqueda

Fecha de entrega: Lunes 20 de mayo

Teorıa (1.75 pts.)

El algoritmo Rushed-A* es una variabte de A*. Su pseudocodigo contiene una lınea adicional a A* y se presenta a continuacı´on, donde la funcı´on Insertar est´a implementada de la misma manera que el tradicional A*.
Algoritmo Rushed-A*

Input: Un problema de busqueda (S, A, s0, G)
Output: Un nodo objetivo
for each s ∈ S do g(s) ← ∞ g(s0) ← 0; f(s0) ← h(s0); Open ← {s0}
while Open/= ∅
Extrae un u desde Open con menor valor-f if u es objetivo return u
for each v ∈ Succ(u) do
if v es objetivo y f(v) = f(u) return v Insertar v
// retorno tradicional // retorno adicional

Actividad 1.1. (0.75 ptos.) Haz un argumento a favor de usar Rushed-A* en vez de la implementación típica de A*

El algoritmo Rushed-A* representa una variante modificada del A* tradicional, diseñada para optimizar ciertos aspectos de rendimiento mediante la terminación anticipada de la búsqueda bajo condiciones específicas. Esta característica distintiva del Rushed-A* lo hace particularmente adecuado para entornos donde la velocidad de respuesta es más crítica que la obtención de la solución óptima absoluta.

Argumento a favor del uso de Rushed-A* sobre A*
Situación Hipotética: Control de Tráfico en Tiempo Real

Contexto: Imaginemos un sistema de control de tráfico en tiempo real que utiliza datos en vivo para gestionar y dirigir flujos de vehículos en una ciudad grande. El sistema debe calcular rápidamente rutas alternativas para múltiples vehículos en respuesta a condiciones cambiantes, como accidentes de tráfico, congestión o cierres de carreteras.
Desafío: En un escenario típico de gestión de tráfico, las rutas deben recalculares frecuentemente a medida que cambian las condiciones. Usar un algoritmo que siempre busca la ruta óptima, como el A* tradicional, podría resultar en una sobrecarga computacional significativa y retardos en la respuesta, lo que a su vez podría empeorar la congestión.

Ventajas del Uso de Rushed-A*:

1. **Eficiencia Temporal**: Rushed-A* puede interrumpir la búsqueda tan pronto como encuentra un nodo objetivo cuya función de evaluación f es igual a la del nodo actualmente en expansión. Esto permite que el sistema responda más rápidamente a situaciones cambiantes, recalculando rutas en tiempo real sin el requisito de alcanzar siempre la solución más óptima.
2. **Reducción de la Carga Computacional**: Al terminar las búsquedas antes, Rushed-A* reduce el número total de nodos que se exploran, disminuyendo así la carga computacional y el uso de la memoria. Esto es crucial en sistemas de control de tráfico donde múltiples rutas deben ser calculadas simultáneamente y la eficiencia del procesamiento afecta directamente el rendimiento del sistema.
3. **Adaptabilidad**: Rushed-A* ofrece un balance entre velocidad y precisión que es adecuado para entornos dinámicos. Al proporcionar soluciones que son "suficientemente buenas", permite que los controladores de tráfico implementen rápidamente medidas que pueden aliviar la congestión y responder a emergencias sin demora.

En aplicaciones como el control de tráfico en tiempo real, donde las condiciones son dinámicas y la demanda de respuestas rápidas es alta, Rushed-A* ofrece una ventaja significativa sobre A* en términos de eficiencia computacional y temporal. Su capacidad para adaptarse rápidamente a nuevas informaciones y su menor demanda de recursos lo hacen ideal para escenarios donde la velocidad y la adaptabilidad son más importantes que la precisión absoluta de la solución.

1.2. (1 pto.) Encuentra y describe una condición no trivial de h que garantiza que Rushed-A* solo retorna soluciones óptimas. Presenta una completa demostración de que Rushed-A* es óptimo en estas condiciones.

Para garantizar que el algoritmo Rushed-A* solo retorne soluciones óptimas, es fundamental emplear una heurística que sea admisible y consistente. Estas condiciones son cruciales para mantener las propiedades de optimalidad inherentes al A* tradicional, incluso cuando Rushed-A* permite terminaciones anticipadas bajo ciertos criterios.

Definiciones Clave

Definición (Admisibilidad)

Una función heurística h se dice *admisible*, si para todo s:

$$h(s) \leq h^*(s)$$

Definición (Heurísticas Consistentes)

Una heurística se dice *consistente* ssi

- $h(s) = 0$, para todo $s \in G$.
- $h(s) \leq c(s, s') + h(s')$, para todo vecino s' de s.

Demostración de la Optimalidad de Rushed-A*

Cuando Rushed-A* inicia, se establece que (g(s_0) = 0) y (f(s_0) = h(s_0)), siendo s_0 el nodo inicial. Debido a la admisibilidad de h , (f(s_0)) es una cota inferior del costo real mínimo del inicio al objetivo.

A medida que Rushed-A* procede:

1. **Expansión de Nodos**: Rushed-A* expande nodos basándose en el valor mínimo de f. Si se alcanza un nodo objetivo u, el algoritmo termina retornando u , garantizando así que(f(u) = g(u) + h(u)) es el costo mínimo debido a la heurística admisible.

2. **Terminación Anticipada:** Si un sucesor v de u también es un objetivo y (f(v) = f(u)), entonces Rushed-A* puede retornar(v) directamente. Por la consistencia de (h), el valor
f(v) = g(u) + c(u, v) + h(v) no excede g(v) , asegurando que v también representa una solución óptima.

3. **Validación por Inducción:** Similar a la demostración de A*, si aplicamos inducción podemos argumentar que en cada expansión, g(s_j) refleja el costo del camino óptimo desde el inicio hasta s_j. Si Rushed-A* decide retornar un nodo v donde f(v) = f(u), el costo g(v) también debe ser óptimo, validando así la decisión del algoritmo de seleccionar v como la solución final.

En conclusión, utilizando una heurística que es tanto admisible como consistente, Rushed-A* no solo ofrece la posibilidad de terminar la búsqueda de manera prematura sino que también garantiza la optimalidad de las soluciones encontradas. Esto lo convierte en una herramienta valiosa en entornos donde la eficiencia y la rapidez son esenciales, sin comprometer la exactitud del resultado final.

DCComilon

2.1. Actividad 1: Implementación BFS y DFS invertido.

Test 1				
Algoritmo	**Tiempo (ms)**	**Expansiones**	**Largo del camino**	**Costo del camino**
DFS	0.09	41	37	37
Inverted DFS	0.27	56	23	23
BFS	0.23	54	21	21

Test 2				
Algoritmo	**Tiempo (ms)**	**Expansiones**	**Largo del camino**	**Costo del camino**
DFS	0.05	40	38	47.5
Inverted DFS	0.37	52	25	20.5
BFS	0.068	56	18	23

Test 3				
Algoritmo	**Tiempo (ms)**	**Expansiones**	**Largo del camino**	**Costo del camino**
DFS	0.40	198	156	156
Inverted DFS	0.97	331	108	108
BFS	1.17	382	94	94

Test 4				
Algoritmo	**Tiempo (ms)**	**Expansiones**	**Largo del camino**	**Costo del camino**
DFS	0.40	198	156	164
Inverted DFS	1.14	331	108	108
BFS	1.27	382	94	107

Test 5				
Algoritmo	**Tiempo (ms)**	**Expansiones**	**Largo del camino**	**Costo del camino**
DFS	1.18	361	125	125
Inverted DFS	1.05	331	175	175
BFS	2.88	629	95	95

Test 6				
Algoritmo	**Tiempo (ms)**	**Expansiones**	**Largo del camino**	**Costo del camino**
DFS	1.20	361	125	145
Inverted DFS	1.25	331	175	197
BFS	3.25	629	95	110

Test 7				
Algoritmo	**Tiempo (ms)**	**Expansiones**	**Largo del camino**	**Costo del camino**
DFS	19.30	1697	434	482
Inverted DFS	11.79	1246	662	780
BFS	22.63	1793	262	354

Test 8				
Algoritmo	**Tiempo (ms)**	**Expansiones**	**Largo del camino**	**Costo del camino**
DFS	18.70	1669	344	415
Inverted DFS	7.42	915	392	416
BFS	18.65	1598	224	292

2.2. Actividad 2.2: Implementación de heurísticas.

Test 1				
Algoritmo	**Tiempo (ms)**	**Expansiones**	**Largo del camino**	**Costo del camino**
Manhattan	0.80	56	20	20
Euclidian	0.61	59	20	20

Test 2				
Algoritmo	**Tiempo (ms)**	**Expansiones**	**Largo del camino**	**Costo del camino**
Manhattan	0.29	44	26	21
Euclidian	0.71	51	26	21

Test 3				
Algoritmo	**Tiempo (ms)**	**Expansiones**	**Largo del camino**	**Costo del camino**
Manhattan	3.43	380	98	103
Euclidian	3.31	379	98	103

Test 4				
Algoritmo	**Tiempo (ms)**	**Expansiones**	**Largo del camino**	**Costo del camino**
Manhattan	5.32	566	95	95
Euclidian	6.12	583	95	95

Test 5				
Algoritmo	**Tiempo (ms)**	**Expansiones**	**Largo del camino**	**Costo del camino**
Manhattan	5.32	566	95	95
Euclidian	6.12	583	95	95

Test 6				
Algoritmo	**Tiempo (ms)**	**Expansiones**	**Largo del camino**	**Costo del camino**
Manhattan	4.73	539	110	103.5
Euclidian	4.59	554	110	103.5

Test 7				
Algoritmo	**Tiempo (ms)**	**Expansiones**	**Largo del camino**	**Costo del camino**
Manhattan	13.50	1604	322	285.5
Euclidian	14.90	1612	322	285.5

Test 8				
Algoritmo	**Tiempo (ms)**	**Expansiones**	**Largo del camino**	**Costo del camino**
Manhattan	13.01	1803	348	261
Euclidian	19.29	1813	348	261

¿Por qué utilizamos la mitad del valor original de estas heurísticas?
Utilizamos la mitad del valor original de las heurísticas para asegurar que la heurística siga siendo admisible y consistente. Una heurística es admisible si nunca sobreestima el costo real para llegar al objetivo, y es consistente si para cualquier nodo n y sucesor n'el valor de h(n) es menor o igual que el costo de alcanzar n'desde n más $h(n')$

Reducir las heurísticas a la mitad garantiza que estas propiedades se mantengan, lo cual es crucial para que el algoritmo A* encuentre una solución óptima.

¿Qué ocurriría si utilizamos el valor completo?
Si utilizamos el valor completo de las heurísticas, podríamos perder la admisibilidad y la consistencia. Esto significa que la heurística podría sobreestimar el costo para llegar al objetivo, lo que podría llevar al algoritmo A* a encontrar soluciones no óptimas. En otras palabras, A* podría encontrar un camino que no sea el más corto o el menos costoso, ya que confiaría demasiado en la heurística y podría pasar por alto soluciones mejores.

Conclusión
La implementación de las heurísticas y la ejecución del algoritmo A* con estas heurísticas proporciona una forma de evaluar la eficiencia del algoritmo en diferentes mapas. Al reducir las heurísticas a la mitad, garantizamos que A* sigue siendo óptimo, aunque podríamos sacrificar algo de eficiencia en términos de expansiones y tiempo de búsqueda.

Actividad 3: Implementación de Recursive Best-First Search (RBFS).

Tuve dificultades con esta actividad, que intenté solucionar sin éxito. Aunque la implementé en el código, no funcionó correctamente. A pesar de realizar varios cambios en el main, el problema persistió

Actividad 4: Comparación

Test 1				
Algoritmo	**Tiempo (ms)**	**Expansiones**	**Largo del camino**	**Costo del camino**
DFS	0.09	41	37	37
Inverted DFS	0.27	56	23	23
BFS	0.23	54	21	21
Manhattan	0.80	56	20	20
Euclidian	0.61	59	20	20

Test 2				
Algoritmo	**Tiempo (ms)**	**Expansiones**	**Largo del camino**	**Costo del camino**
DFS	0.05	40	38	47.5
Inverted DFS	0.37	52	25	20.5
BFS	0.068	56	18	23
Manhattan	0.29	44	26	21
Euclidian	0.71	51	26	21

Test 3				
Algoritmo	**Tiempo (ms)**	**Expansiones**	**Largo del camino**	**Costo del camino**
DFS	0.40	198	156	156
Inverted DFS	0.97	331	108	108
BFS	1.17	382	94	94
Manhattan	3.43	380	98	103
Euclidian	3.31	379	98	103

Test 4				
Algoritmo	**Tiempo (ms)**	**Expansiones**	**Largo del camino**	**Costo del camino**
DFS	0.40	198	156	164
Inverted DFS	1.14	331	108	108
BFS	1.27	382	94	107
Manhattan	5.32	566	95	95
Euclidian	6.12	583	95	95

Test 5				
Algoritmo	**Tiempo (ms)**	**Expansiones**	**Largo del camino**	**Costo del camino**
DFS	1.18	361	125	125
Inverted DFS	1.05	331	175	175
BFS	2.88	629	95	95
Manhattan	5.32	566	95	95
Euclidian	6.12	583	95	95

Test 6				
Algoritmo	**Tiempo (ms)**	**Expansiones**	**Largo del camino**	**Costo del camino**
DFS	1.20	361	125	145
Inverted DFS	1.25	331	175	197
BFS	3.25	629	95	110
Manhattan	4.73	539	110	103.5
Euclidian	4.59	554	110	103.5

Test 7				
Algoritmo	**Tiempo (ms)**	**Expansiones**	**Largo del camino**	**Costo del camino**
DFS	19.30	1697	434	482
Inverted DFS	11.79	1246	662	780
BFS	22.63	1793	262	354
Manhattan	13.50	1604	322	285.5
Euclidian	14.90	1612	322	285.5

Test 8				
Algoritmo	**Tiempo (ms)**	**Expansiones**	**Largo del camino**	**Costo del camino**
DFS	18.70	1669	344	415
Inverted DFS	7.42	915	392	416
BFS	18.65	1598	224	292
Manhattan	13.01	1803	348	261
Euclidian	19.29	1813	348	261

En la evaluación de los algoritmos DFS, Inverted DFS, BFS, y las heurísticas de A* (Manhattan y Euclidian) a través de una serie de tests, es crucial analizar meticulosamente los tiempos de búsqueda, el número de expansiones, el largo del camino, y el costo del camino para cada uno. La comparación muestra que BFS, aunque no siempre es el más rápido en términos de tiempo de ejecución, ofrece consistentemente los caminos más cortos y de menor costo. Esto lo hace particularmente valioso en aplicaciones donde el equilibrio entre eficiencia y economía es prioritario.

Por otro lado, las heurísticas de Manhattan y Euclidian, implementaciones de A*, son generalmente más rápidas en entornos simples debido a su capacidad para estimar mejor la distancia al objetivo sin restricciones de movimiento ortogonal estricto. Sin embargo, pueden no ser tan efectivas en términos de minimización de costos en mapas más complejos. En comparación, DFS y su variante Inverted DFS, aunque tienden a ejecutarse rápidamente, frecuentemente resultan en mayores costos y caminos más largos, lo que puede no ser ideal en escenarios donde la eficiencia del camino es crítica.

Desde una perspectiva teórica y cualitativa, y observando su comportamiento en el visualizador, **BFS emerge como el algoritmo más equilibrado,** proporcionando un compromiso robusto entre tiempo de ejecución y optimización de ruta. Este equilibrio lo convierte en la elección preferida para escenarios donde se valoran tanto la precisión del camino como el rendimiento económico.

3. DCCheckers

3.1. Actividad 1: Monte Carlo Tree Search

Profundidad 1			
Nº	Winner	Tiempo promedio MCTS (s)	Tiempo Minimax (s)
1	Montecarlo	4.410	0.000683
2	Montecarlo	3.960	0.001320
3	Montecarlo	2.380	0.000810
4	Montecarlo	2.465	0.000960
5	Minimax	2.510	0.000990
6	Montecarlo	2.360	0.000780
7	Montecarlo	2.340	0.000610
8	Montecarlo	2.430	0.000750
9	Montecarlo	2.470	0.000970
10	Montecarlo	2.380	0.000740

Profundidad 2			
Nº	Winner	Tiempo promedio MCTS (s)	Tiempo Minimax (s)
1	Montecarlo	3.460	0.013780
2	Montecarlo	2.680	0.014980
3	Montecarlo	2.860	0.013840
4	Montecarlo	3.160	0.016030
5	Montecarlo	2.920	0.010010
6	Montecarlo	3.380	0.015040
7	Montecarlo	2.880	0.008310
8	Montecarlo	2.420	0.008710
9	Montecarlo	2.490	0.007110
10	Montecarlo	2.400	0.008770

Profundidad 3			
Nº	Winner	Tiempo promedio MCTS (s)	Tiempo Minimax (s)
1	Minimax	2.250	0.003960
2	Minimax	2.550	0.010250
3	Minimax	2.440	0.009940
4	Montecarlo	2.400	0.006100
5	Minimax	2.810	0.006970
6	Montecarlo	2.290	0.007170
7	Minimax	2.450	0.010910
8	Minimax	2.410	0.008240
9	Minimax	2.320	0.007730
10	Montecarlo	2.310	0.009140

En esta segunda actividad, comparamos el algoritmo Minimax con Monte Carlo Tree Search (MCTS) utilizando tres profundidades distintas (1, 2 y 3) en 10 partidas diferentes para cada profundidad. El rendimiento de ambos algoritmos varía según la profundidad de búsqueda. A una profundidad de 1, MCTS generalmente gana más partidas debido a su capacidad para explorar rápidamente un gran número de movimientos posibles. Esta ventaja se debe a que MCTS utiliza una exploración amplia y aleatoria, lo que es beneficioso en búsquedas

superficiales. Sin embargo, a medida que la profundidad aumenta a 2 y 3, Minimax comienza a ganar más partidas, especialmente a profundidad 3, donde obtiene la mayoría de las victorias. Este cambio se debe a la capacidad de Minimax para evaluar exhaustivamente las posiciones del tablero y anticipar movimientos futuros, proporcionando decisiones más estratégicas y bien informadas.

Teóricamente, este comportamiento se puede explicar por la naturaleza de los algoritmos. MCTS se beneficia en profundidades menores debido a su exploración rápida y amplia, que descubre movimientos prometedores de manera eficiente. En contraste, a mayores profundidades, la evaluación detallada de Minimax y su capacidad para considerar múltiples niveles de movimientos y contramovimientos le dan una ventaja significativa en la toma de decisiones estratégicas. Sin embargo, el costo computacional de Minimax aumenta con la profundidad, lo que puede ser una limitación en escenarios donde el tiempo de respuesta es crítico.

3.2. Actividad 2: Profundidades distintas.

Minimax White 1v3 Black			
N°	**Winner**	**Tiempo Minimax White (s)**	**Tiempo Minimax Black(s)**
1	White Min	0,001218	0,010638
Minimax White 3v5 Black			
N°	**Winner**	**Tiempo Minimax White (s)**	**Tiempo Minimax Black(s)**
1	White Min	0,008144	0,009057
Minimax White 5v1 Black			
N°	**Winner**	**Tiempo Minimax White (s)**	**Tiempo Minimax Black(s)**
1	White Min	0,009002	0,000968

Al comparar el desempeño de agentes que utilizan Minimax con profundidades de 1 y 3 (1v3), teóricamente, el incremento en la profundidad permite al algoritmo evaluar más movimientos futuros, mejorando la calidad de las decisiones estratégicas. Experimentalmente, esto se refleja en tiempos de decisión más largos pero con una clara ventaja en la calidad de las jugadas, ya que el agente con mayor profundidad puede anticipar mejor las respuestas del oponente. En este caso, los resultados muestran que White, utilizando una profundidad mayor, gana debido a su capacidad para planificar jugadas más efectivas.

En la comparación entre profundidades de 3 y 5 (3v5), aumentar la profundidad de 3 a 5 incrementa la capacidad del algoritmo para prever y reaccionar a un mayor número de posibles escenarios futuros. Teóricamente, esto se traduce en decisiones aún más precisas y estratégicas, aunque a costa de un tiempo de procesamiento significativamente mayor. Los resultados experimentales indican que, aunque el tiempo de decisión aumenta considerablemente, la precisión y la efectividad de las jugadas también mejoran,

permitiendo al agente que utiliza profundidad 5 (White) superar a su contraparte con profundidad 3 (Black) debido a una mejor anticipación y planificación.

En el enfrentamiento entre profundidades de 5 y 1 (5v1), el jugador con una profundidad de 5 tiene una ventaja teórica significativa sobre el jugador con una profundidad de 1. El incremento en la profundidad permite una evaluación más exhaustiva de las posibles movidas y contramovidas, resultando en un juego más estratégico y controlado. Experimentalmente, esto se refleja en victorias consistentes del jugador que utiliza profundidad 5, ya que puede prever y evitar trampas inmediatas, optimizando su estrategia a largo plazo, a pesar del mayor tiempo requerido para cada decisión.

El incremento en la profundidad del algoritmo Minimax mejora significativamente la capacidad de anticipación y planificación, resultando en jugadas más estratégicas y efectivas. Aunque esto conlleva un mayor tiempo de procesamiento, los beneficios en términos de precisión y calidad de las decisiones justifican el costo adicional, especialmente en entornos competitivos o complejos.

3.3. Actividad 3: Poda Alfa-Beta.

Poda Alfa Beta

Minimax White 1v3 Black			
N°	**Winner**	**Tiempo Minimax White (s)**	**Tiempo Minimax Black(s)**
1	White Min	0.002713	0.016729
Minimax White 3v5 Black			
N°	**Winner**	**Tiempo Minimax White (s)**	**Tiempo Minimax Black(s)**
1	White Min	0.011953	0.012981
Minimax White 5v1 Black			
N°	**Winner**	**Tiempo Minimax White (s)**	**Tiempo Minimax Black(s)**
1	White Min	0.0143212	0.001589

La poda alfa-beta es una optimización del algoritmo Minimax que reduce el número de nodos evaluados en el árbol de búsqueda. Implementando esta técnica en el código, cuando el argumento de la función alphabeta es True, se observa una mejora significativa en la eficiencia del algoritmo. Al repetir el experimento de la Actividad 3 utilizando la poda alfa-beta, los resultados muestran una reducción considerable en el tiempo promedio de toma de decisiones para ambos jugadores (White y Black).

En los enfrentamientos Minimax White 1v3 Black, el tiempo promedio para White se reduce a 0.002713 segundos en comparación con 0.016729 segundos para Black. En Minimax White 3v5 Black, el tiempo para White es de 0.011953 segundos frente a 0.012981 segundos para Black. Finalmente, en Minimax White 5v1 Black, los tiempos son 0.0143212 segundos para White y 0.001589 segundos para Black. Estos resultados indican que la poda alfa-beta permite al algoritmo descartar ramas del árbol de búsqueda que no afectarán la decisión final, reduciendo así la cantidad de cálculos necesarios.

La implementación de la poda alfa-beta mejora el desempeño y la eficiencia del algoritmo Minimax al disminuir el número de nodos evaluados, lo que resulta en una toma de decisiones más rápida sin comprometer la calidad de las jugadas. Esto es especialmente útil en juegos complejos donde la profundidad de búsqueda puede ser significativa. En resumen, la poda alfa-beta contribuye a un juego más ágil y eficiente, permitiendo a los agentes tomar decisiones estratégicas de manera más rápida y efectiva.

Actividad 4 (BONUS): Función de valor.

La función de valor actual para evaluar el estado del juego se calcula como:

$$(N_{fichasblancas} - N_{fichasnegras}) + \frac{1}{2}(N_{reyesblancos} - N_{reyesnegros})$$

Beneficios:

La función es simple de calcular, lo que permite evaluaciones rápidas y eficientes durante la búsqueda Minimax. Esto es especialmente importante cuando se deben evaluar muchos estados posibles del juego. Ademas la fonction enfoque en la cantidad de Piezas: Considera la cantidad de fichas y reyes, lo que generalmente es un buen indicador del estado del juego. Más piezas suelen indicar una posición más fuerte.

Problemas:

La función no considera la posición de las fichas en el tablero, lo que puede ser crucial para determinar la fuerza real de una posición; por ejemplo, una ficha en una posición central puede ser más valiosa que una en una esquina. Además, la diferencia ponderada de reyes frente a fichas puede no reflejar adecuadamente su valor real en el juego, donde los reyes pueden tener un impacto mucho mayor en ciertas situaciones.

Ejemplo: En una situación donde un jugador tiene muchas fichas pero están atrapadas o en posiciones desfavorables, esta función podría sobrevalorar su posición. De manera similar, no se valoran adecuadamente las posiciones estratégicas de las fichas que podrían tener un impacto significativo en el resultado del juego.

Nueva Función de Evaluación Propuesta

Propongo una nueva función de evaluación que tenga en cuenta no solo la cantidad de fichas y reyes, sino también su posición en el tablero. La nueva función podría ser:

$$\text{Valor} = (N_{\text{fichas\_blancas}} - N_{\text{fichas\_negras}}) + 2 \times (N_{\text{reyes\_blancos}} - N_{\text{reyes\_negros}}) + \sum_{\text{ficha blanca}} \text{pos\_valor} - \sum_{\text{ficha negra}} \text{pos\_valor}$$

Donde pos_valor es un valor asignado a cada posición en el tablero basado en su importancia estratégica.

Beneficios:

Al incluir la valoración de la posición, la función puede reflejar mejor la realidad del juego, donde la ubicación de las piezas es crucial, y al dar un peso mayor a los reyes, la función refleja mejor su verdadero impacto en el juego.

Problemas:

Esta función es más compleja de calcular, lo que puede ralentizar el proceso de evaluación en el algoritmo Minimax, y requiere una afinación cuidadosa de los valores posicionales y los pesos asignados a los reyes para equilibrar adecuadamente la evaluación. En resumen, aunque la nueva función de evaluación es más compleja, ofrece una representación más precisa del estado del juego al considerar tanto la cantidad como la posición de las fichas y los reyes, lo que puede resultar en una mejor toma de decisiones estratégicas durante el juego.

Conclusión

La nueva función de valor mejora la precisión de la evaluación del estado del juego al incorporar la estructura, movilidad y control del tablero. Aunque más compleja, proporciona una representación más fiel de la realidad del juego, lo que puede llevar a decisiones más estratégicas y efectivas en el algoritmo Minimax.