Sistemas de Inteligencia Artificial
Métodos de búsqueda no informados e informados
Eternity 2

Esteban Pintos (51048) - Cristian Pereyra (51190) - Matías De Santi (51051)

25 de Marzo de 2013

1. Introducción

Para el primer trabajo práctico de la materia se nos pidió resolver el juego Eternity 2 utilizando métodos de búsqueda no informados e informados. Este informe explica cómo se representó el problema para ser resuelto y muestra los resultados obtenidos a partir de las diferentes resoluciones.

2. Breve explicación del juego Eternity 2

El juego Eternity 2 consta de un tablero cuadrado de nxn con n^2 piezas a ser colocadas. Cada pieza tiene la particularidad de que tiene 4 colores, uno en cada lado. El objetivo del juego es colocar todas las piezas de manera tal que las piezas con color gris queden en los bordes (con el color gris tocando el borde) y las piezas adyacentes tengan el mismo color en el borde que comparten. Las piezas pueden ser rotadas antes de ser colocadas. Un ejemplo de este juego se encuentra en la siguente URL: http://www.juegosrox.com/logica/juego-eternity-2.html.

3. Definición del problema

Antes de realizar una representación del problema, se optó por definir el problema de manera clara. Para esto, definimos lo siguiente:

- **Estado inicial**: tablero vacío
- **Conjunto de posibles acciones**: dado un estado, se puede colocar sobre el tablero cualquier ficha siempre y cuando esta no haya sido colocada antes y no haya una ficha en la posición que se quiere colocar.
- **Modelo de transición**: colocar una pieza en una posición del tablero da como resultado un nuevo tablero con todas las fichas que tenía el anterior más la que ha sido colocada
- **Condición de solución**: un tablero es solución del problema si las fichas con color gris han sido colocadas con el color gris del lado del borde y para toda ficha adyacente en el tablero, los colores de los bordes que comparten son iguales.

4. Representación del problema

Una vez definido el problema, se realizó la representación del mismo.

4.1. Representación de estados

Para representar los estados, se optó por que cada uno tuviera un tablero, la pieza que se colocó sobre el mismo y una referencia al estado a partir del cual se generó.

El tablero se representó con un HashMap cuya clave es un punto del tablero y el valor es la pieza colocada. Al crear un estado, el tablero únicamente contiene la pieza que diferencia al estado actual de su padre. Esto se dedidió que sea así para poder ahorrar el uso de memoria hasta que realmente sea necesario tener guardadas todas las piezas. Cuando al tablero se le pide la pieza en una posición que no tiene guardada, este se la pide al estado padre y así sucesivamente. Si alguno de los padres tiene guardada una pieza en dicha posición, la retorna y todos los hijos almacenan dicha información para no tener que volver a pedirla. En caso de que la posición solicitada esté vacía, se retorna una pieza vacía y todos los estados hijos también la almacenan. Es por esto que un estado cuenta con una referencia al estado padre.

Es importante también la forma en que se representó una pieza. En este caso, se optó por representar una pieza como un objeto con 4 variables: *up*, *right*, *down* y $left$. Cada una de estas variables indica el color que tiene la pieza en cada uno de los bordes. Una pieza vacía tiene todas estas variables con valor -1. El color que debe estar contra los bordes del tablero se indica con el número 0. Esto es una abstracción respecto del problema original ya que el mismo cuenta con colores y formas. En esta representación, se asigna un número a cada combinación posible de color y forma.

Una vez que se tuvo representado un estado cualquiera y las piezas, se definió el estado inicial. El estado inicial se representó con un tablero vacío, la pieza colocada en *null* al igual que la referencia al estado padre.

4.2. Acciones posibles

Si bien existen dos posibles acciones en el juego real (colocar una pieza en una posición del tablero y rotar una pieza), se decidió reducir a una sola las acciones posibles en la representación del problema: colocar una pieza en una posición del tablero. El poder rotar las piezas se resolvió de la siguiente manera: cuando se inicia el programa, se crean todas las posibles reglas, siendo cada una de la siguiente forma: colocar la pieza p en la posición x, y. Al generar las reglas para todas las piezas, también se generan reglas para las piezas rotadas. Esto hace que todas las reglas sean iguales y por lo tanto la aplicación de la misma es sumamente sencilla: se debe verificar que la pieza no haya sido colocada en el tablero (ya sea rotada o no) y que la posición en la cual se quiere colocar no haya otra pieza.

4.3. Modelo de transición

Al aplicar una regla sobre un tablero, se retorna un tablero nuevo con la pieza que se agregó en la posición indicada.

4.4. Condición de terminación

El tablero debe estar completo y las piezas deben cumplir las siguientes condiciones:

- Si la pieza contiene color 0, entonces este color debe estar pegado a un borde del tablero.

- Las piezas adyacentes deberan tener los mismos colores en los bordes que tienen en común.

5. Algoritmos implementados

5.1. Algoritmos no informados

Se implementaron tres algoritmos no informados: **Depth First Search** (DFS), **Breadth First Search** (BFS) y **Iterative Deepening** (ID).

5.2. Algoritmos informados

Se implementaron dos algoritmos informados: **Greedy Search** y **A***. Para estos algoritmos se utilizaron 3 funciones distintas:

- $g(n)$: costo de haber llegado del nodo inicial al nodo n, utilizado sólo por **A***.

- $h(n)$: estimación del costo de llegar del nodo n al nodo *goal*

6. Heurísticas

Para este trabajo se implementaron tres heurísticas distintas. La característica en común de las mismas es que parten del hecho que el tablero es válido. Si el estado es inválido, es decir que el tablero no está bien formado, entonces la heurística devuelve el máximo valor posible para evitar que se explore dicho estado.

6.1. Heurística 1: distancias Manhattan

Esta heurística surgió luego de jugar el juego repetidas veces. Se vió que si se colocan las piezas de los bordes primero, y luego se continúa con las piezas que son adyacentes a las del borde, entonces el problema se puede resolver de manera más rápida. Para poder llevar esto a código, se decidió realizarlo de la siguiente manera.

Se tomó un valor fijo para un tablero de dimensión nxn que es la suma de las distancias manhattan de cada posición del tablero. Por ejemplo, para un tablero de dimensión $5x5$ esta suma da 60.

Por otra parte, para un estado dado, se calcula la suma de las distancias manhattan de las piezas que estan colocadas. Como las piezas que estan en los bordes tienen una distancia mayor que las que estan más cerca del borde, cuantas mas piezas estén colocadas en los bordes esta suma será mayor.

Finalmente, la $h(n)$ se calcula como la resta entre el valor calculado para el tablero entero y el valor calculado para el estado dado.

6.2. Heurística 2: orden de colocación de las piezas

Esta heurística es más simple que la mencionada anteriormente. En este caso, el objetivo de la heurística es colocar las piezas en orden de espiral, empezando del $(0, 0)$, luego el $(0, 1)$ y así hasta completar la primer fila. Luego desciende por la ultima columna y continúa de esa manera.

6.3. Heurística 3: color de piezas

Esta heurística penaliza aquellos tableros en los cuales ninguna ficha puede ser colocada ya que no existe ninguna ficha en el tablero con el

color correspondiente para que alguna de las que falta colocar sea colocada. Es decir, ese tablero no conducirá a una solución válida.

Es imporante aclarar que ninguna de las heurísticas presentadas es admisible.

7. Funciones de costo

Se desarrollaron 3 funciones de costo

7.1. Función 1: Costo basado en rotaciones

Se desarrolló una función de costo cuyo objetivo es desalentar a los caminos que tengan piezas rotadas, para evitar tener que analizar simetrías.

7.2. Función 2: Costo uniforme

Retorna el mismo valor parar todos los estados.

7.3. Función 3: Costo basado en distancia al centro

Le da menor valor de heurística a aquellos tableros que tiene mayor cantidad de piezas puestas en las afueras.

8. Resultados

Los resultados que se comentan a continuación fueron extraidos de pruebas realizadas sobre tableros de diferentes dimensiones, pero todos los tableros contaban con 6 colores distintos más el color gris. A los algoritmos se les dio un tiempo máximo de ejecución de 60 segundos. Las pruebas fueron realizadas en una computadora con un procesador Intel Core i7 de 2 GHz y una memoria RAM exclusiva para el programa de 12GB. Aquellos resultados cuyo Tiempo de Sim. sea s"quiere decir que dicho algoritmo no encontró solución antes del tiempo máximo.

En la tabla 1 se pueden ver los resultados de los distintos algoritmos para un tablero de 2x2. Si bien los tiempos de ejecución son relativamente chicos, es interesante destacar la cantidad de nodos expandidos en cada algoritmo. Los algoritmos informados expanden una cantidad considerablemente menor que los no informados (exceptuando a A* con Manhattan distance y costo basado en colres).

Para un tablero de 3x3, los resultados pueden verse en la tabla 2. Como puede observarse, para este tablero los métodos no informados tardan considerablemente más que los informados. Si se toma el algoritmo Greedy con la heurística de Manhattan distance para ambos tableros, el incremento de tiempos fue del 722 %. Esto se explica teniendo en cuenta el factor de ramificación.

8.1. Factor de ramificación

Al iniciar el programa se crean las reglas de la siguiente manera: para cada pieza del tablero se crea una regla para cada posición del tablero. Es decir, por cada pieza se crean n^2 reglas. Además, se deben crear las reglas para la misma pieza rotada, por lo cual por cada pieza se crean $4n^2$ reglas. Dado que existen n^2 piezas en el juego, hay $4n^4$ reglas.

Todas estas reglas son aplicables en el primer estado, por lo cual el estado inicial tendrá $4n^4$ hijos. Cada uno de estos hijos a su vez tendrá $4n^4 - 4$ reglas para aplicar ya que 4 reglas no son aplicables porque una pieza ya se colocó. En resumen, dado un nodo del árbol de búsqueda cuya profundidad es d, entonces el factor de ramificación es $b = 4n^4 - 4 \times d$.

A continuación se muestra una tabla donde, para cada tamaño de tablero, se muestran la cantidad de nodos que tiene el árbol de exploración:

- **2x2**: 11182080
- **3x3**: 24817428680914501632000
- **4x4**: 905703814554358516996854857467489926355353600000

Sin embargo, estos números tienen en cuenta el orden de aplicación de las reglas y por lo tanto existen un gran número de simetrías que deben ser detectadas para evitar analizar dos veces estados equivalentes. Además, como las reglas no validan que el estado que se genera sea válido, muchos de estos estados son inválidos y por lo tanto no llevarán a una solución correcta. Sin embargo, deben ser analizados hasta que esté completo y por eso es que los algoritmos no informados son sumamente lentos.

El rápido crecimiento de la cantidad de estados a analizar trajo como consecuencia que la memoria RAM fuera una limitación a la hora de correr los algoritmos. Aquellos algoritmos que no contaban con información acerca del problema se quedaban rápidamente sin memoria RAM. Lo mismo le sucedió a los algoritmos informados, pero a diferencia de los no informados, llegaron a resolver tableros de 4x4 y con 6 colores. Sin embargo, a la hora de resolver tableros de 5x5 y 6 colores, sufrieron la falta de memoria RAM.

8.2. Tableros de dimensión superior a 3x3

Considerando que para un tablero de 3x3 se hallaban soluciones solamente con heurísticas, se decidió probar con un tablero de 4x4 y utilizar dichas heurísticas. Los resultados pueden verse en la tabla 3. Es realmente interesante observar la diferencia de tiempos y de nodos expandidos entre el algoritmo Greedy y el A*. Si bien los dos utilizan la misma función heurística, el costo basado en rotaciones parece hacer un muy buen trabajo para evitar tener que comparar por simetrías. La cantidad de nodos hoja y de estados generados en el A* no supera los 6000 mientras que el algoritmo Greedy ronda los 670000.

9. Conclusiones

Como conclusión podemos decir que es notable la diferencia en cuanto a tiempos de ejecución, nodos expandidos, nodos hoja, etc. entre los algoritmos informados y los no informados. Dado que las reglas aplicadas pueden dejar el nodo con un estado inválido, la cantidad de nodos a procesar crece de manera abrupta entre un nodo y su hijo. Por lo tanto, aquellos algoritmos que cuentan con información sobre el problema como para poder elegir estados que potencialmente pueden llevar a una solución son de gran ayuda para reducir el tiempo de procesamiento. Esto es evidente en tableros de dimensión superior a 2x2, donde los algoritmos informados no pudieron dar una respuesta al problema.

También queremos destacar que el efecto de las distintas combinaciones entre heurísticas y funciones de costo afecta al tiempo de procesamiento en los algoritmos informados. Puede que la heurística utilizada sea buena, pero si no es acompañada por una buena función de costo entonces no se maximiza el potencial de ambas.

Por último, a lo largo del trabajo aprendimos que la representación del problema juega un papel crucial. Si una representación no cuida la memoria utilizada, entonces correr el algoritmo es altamente costoso.

.1. Resultados

Tabla 1: Resultados para un tablero de 2x2

Algoritmo	Tiempo de Sim.	Nodos expandidos	Nodos hoja	Profundidad	Estados generados	Simetrias Detectadas
DFS	0.147 s	736	108	4	845	1056
BFS	0.435 s	7892	5612	4	13505	31872
ID	0.430 s	9380	108	4	9489	15648
Greedy con H1	0.036 s	5	119	4	125	0
Greedy con H3	0.029 s	5	119	4	125	0
Greedy con H2	0.026 s	4	116	4	121	0
A* con F1 y H1	0.028 s	4	116	4	121	0
A* con F1 y H2	0.026 s	4	116	4	121	0
A* con F1 y H3	0.027 s	4	116	4	121	0
A* con F2 y H1	0.03 s	5	119	4	125	0
A* con F2 y H2	0.059 s	13	275	4	289	0
A* con F2 y H3	0.105 s	33	575	4	609	36
A* con F3 y H1	0.031 s	5	119	4	125	0
A* con F3 y H2	0.078 s	19	341	4	361	0
A* con F3 y H3	0.095 s	29	446	4	476	9

Tabla 2: Resultados para un tablero de 3x3

Algoritmo	Tiempo de Sim.	Nodos expandidos	Nodos hoja	Profundidad	Estados generados	Simetrias Detectadas
DFS	- s	1358970	1106	8	1360077	4328064
BFS	- s	21340	1866356	2	1887697	2314512
ID	- s	1963069	422	3	1963492	2413296
Greedy con H1	0.26 s	29	3338	9	3368	9
Greedy con H2	0.142 s	9	1131	9	1141	0
Greedy con H3	0.149 s	9	1131	9	1141	0
A* con F1 y H1	0.159 s	9	1131	9	1141	0
A* con F1 y H2	0.148 s	9	1131	9	1141	0
A* con F1 y H3	0.157 s	9	1131	9	1141	0
A* con F2 y H1	0.255 s	29	3338	9	3368	9
A* con F2 y H2	0.244 s	28	3024	9	3053	0
A* con F2 y H3	1.56 s	558	68067	9	68626	2567
A* con F3 y H1	0.258 s	29	3338	9	3368	9
A* con F3 y H2	0.158 s	10	1386	9	1397	0
A* con F3 y H3	0.525 s	228	16641	9	16870	367

Tabla 3: Resultados para tableros de dimensión mayor a 3x3

Algoritmo	Tiempo de Sim.	Nodos expandidos	Nodos hoja	Profundidad	Estados generados	Simetrias Detectadas
4x4						
Greedy con H1	18.458 s	2420	648071	16	650492	8869
Greedy con H2	0.598 s	44	14632	16	14677	0
A* con F1 y H1	0.364 s	16	5968	16	5985	0
A* con F1 y H2	0.38 s	16	5968	16	5985	0
A* con F1 y H3	0.408 s	16	5968	16	5985	0
A* con F2 y H1	19.02 s	2420	648071	16	650492	8869
A* con F2 y H2	1.655 s	145	51951	16	52097	0
A* con F3 y H1	18.602 s	2420	648071	16	650492	8869
A* con F3 y H2	2.719 s	296	92972	16	93269	0
5x5						
A* con F1 y H1	8.321 s	655	191226	25	191882	8003
A* con F1 y H2	1.184 s	29	23747	25	23777	688
6x6						
A* con F1 y H2	7.19 s	251	89672	36	89924	2109