Sistemas de Inteligencia Artificial

Trabajo práctico especial 1

Grupo 5
Julián E. Gutiérrez F. (51141)
Alexis Medvedeff (50066)
Javier Perez Cuñarro (49729)

ITBA
Primer Cuatrimestre 2014

Introducción

Este trabajo especial consiste en la creación de un Sistema de Producción usado para resolver el problema del juego Solitario Mahjong. Entendemos por problema la resolución de una partida del juego, partiendo de un tablero con una cierta configuración inicial y llegando, en lo posible, a un tablero sin fichas, es decir, ganado.
Con este objetivo en mente, partimos del motor de inferencias provisto por la cátedra, analizamos la naturaleza del juego e implementamos tres métodos de búsqueda no informados y dos métodos informados.

Objetivo del juego

El objetivo del Solitario Mahjong es despejar el tablero, removiendo las fichas de a pares jugada tras jugada.
Una par válido consiste en dos piezas "libres" y del mismo tipo o idénticas dependiendo del caso.

Reglas[1]

Acerca de las fichas:

- El juego consta de 144 fichas, es decir 72 pares a eliminar del tablero original.
- Las fichas tienen “palos” al igual que las cartas. Estos son: bambú, caracteres, círculos, dragones, flores, estaciones y vientos.
- Puede haber fichas superpuestas. En ese caso suponemos que el jugador conoce esas fichas a pesar de que no pueda verlas por estar cubiertas.
- En nuestro caso no admitiremos una ficha que está cubierta parcialmente por otra, con lo cual solo estaremos tomando un subconjunto de todos los tableros válidos del juego. De todas formas, consideramos que ese número es suficientemente significativo para obtener conclusiones.

Se pueden remover de a pares:

- Fichas iguales.
- Las fichas que representan “estaciones” se pueden emparejar entre sí, cualquiera de ellas.
- Las fichas que representan “flores” se pueden emparejar entre sí, cualquiera de ellas.

Además, las fichas deben estar libres, es decir:

- No hay otras piezas por encima o cubriéndolas parcialmente.

[1] http://www.ehowenespanol.com/reglas-del-solitario-mah-jong-lista_505506/

- No hay otras fichas a la izquierda o a la derecha de las mismas. Es decir, al menos uno de los lados de la ficha está libre.

Fin del juego:

- El jugador gana cuando logra eliminar los 72 pares de fichas del tablero inicial.
- De otro modo, si no quedan movimientos legales el juego termina. El jugador entonces pierde la partida.

Heurísticas y Costos

Proponemos dos heurísticas admisibles, es decir, que no sobreestiman el costo real de la mejor solución:

- H1: Cantidad de fichas disponibles para jugar en ese estado.

Para esta heurística, la función de costo correspondiente es C1 = 2. El costo es constante y uniforme y representa la cantidad de fichas que se juegan en cada caso.

- H2: Cantidad de pares que se pueden armar para ese estado.

Para esta heurística, la función de costo correspondiente es C2 = 1. El costo es constante y uniforme y representa la cantidad de pares que se juegan en cada caso.

Nótese que la segunda heurística es algo más costosa de calcular (computacionalmente hablando) pero es más acertada, pues en la primera pueden haber muchas fichas jugables pero que no formen pares. En cuanto a los costos, si bien no afectaría la ejecución del algoritmo el número que se use ya que nos encontramos con valores constantes, decidimos mantener coherencia entre las unidades para que sea mas claro.

Implementación

Decidimos representar el tablero como un vector tridimensional. Esta solución desperdicia algo de memoria por los espacios vacíos pero permite un acceso muy fácil a cada uno de ellos. En cada posición del vector se guarda un número entero (int). Los números representan a cada una de las fichas, habiendo dos ocurrencias de cada número en el caso de las fichas comunes y cuatro en el caso de los comodines. Como los comodines son sólo utilizables con los de su mismo tipo, esta representación resulta adecuada.
Al basarnos en la implementación de GPS (conocido como Resolvedor de Problemas Generales en español), notamos que se simplifica mucho agregar un nuevo algoritmo al motor y que la mayoría de la “acción” ocurre en el método *addNode*.

Estrategias de búsqueda no informadas

En el caso de *Breadth First*, basta con agregar los nuevos nodos siempre al final de la lista de nodos abiertos, mientras que en *Depth First* se hace al principio.

Estrategias de búsqueda Informadas

Para el caso de *Greedy Search*, se ordena la lista dejando primero el elemento para el cual la heurística sea mínima.
Para la implementación de *A** ordenamos la lista en cada ocasión que se invoca al *addNode* mediante *Collections.sort().* Consideramos que, si bien impacta en la performance del algoritmo, tiene una complejidad de O(n log n) y lo consideremos aceptable, siendo O(n) en su peor caso.

Observaciones

Hay muchas variantes del Solitario Mahjong. En la versión web[2] encontramos que respetan los pares entre grupos “comodín”, mientras que en la versión de escritorio “Mahjong Solitaire Epic”[3], los pares deben coincidir en todo aspecto para ser válidos. Tomamos como válida la versión web, aunque no sería difícil adaptar el motor para soportar la otra variante.

En todo momento el motor puede conocer todas las fichas en la mesa, si bien en la realidad eso puede no ocurrir si uno se encuentre frente a un tablero ya armado y no se le permite espiar. Así y todo nos encontramos frente a un problema NP-Completo[4][5], ofrecemos una demostración[6] por reducción a 3-SAT, el famoso problema de satisfacibilidad booleana.

Se implementaron tableros de distintos tamaños y características, con menos de 72 pares de fichas, para poder probar de manera rápida las implementaciones.

Factor de ramificación del árbol

Resulta interesante comparar la cantidad de nodos expandidos con dicho factor, ya que el cociente entre estos valores nos dirá cuántos nodos “se ahorraron” de expandir y en cierta medida qué tan efectivas son nuestras heurísticas. Dicho valor lo calculamos según la orientación finalizar el partido es decir a intentar vaciar el tablero lo antes posible. con lo cual tenemos 72 pares posibles y consideramos nuestro árbol máximo de busqueda como 2^72. Aunque podríamos utilizar enfoque por grupos.[7] Bajo el enfoque elegido es claro que H2 resulta más efectiva.

Limitaciones

[2] http://www.jugarjuegos.com/juegos/juegos_gratis/mahjong/solitario.htm
[3] https://itunes.apple.com/mx/app/mahjong-solitaire-epic/id687207809?mt=8
[4] http://www.ics.uci.edu/~eppstein/cgt/hard.html#shang
[5] http://en.wikipedia.org/wiki/NP-complete
[6] http://arxiv.org/pdf/1203.6559v1.pdf
[7] http://arxiv.org/pdf/1203.6559v1.pdf (Ver enfoque por grupos)

- Se podría mejorar los tiempos de ejecución implementando los métodos sin hacer sorting con las bibliotecas de java.
- Estamos al tanto de que la representación elegida para el tablero no es la más eficiente en términos computacionales. La elección se basó en que manipular vectores de enteros nos resultaba sencillo y por ende, no nos distraía de la tarea principal de implementar los métodos de búsqueda.
- La misma crítica se puede hacer para el algoritmo de búsqueda de fichas jugables, que podría ser llevado a una forma más óptima.

Conclusiones

Luego de haber corrido las pruebas con varios tableros de distintos tamaños, observando la tabla que presentamos en el anexo se desprende que dentro de los métodos no informados el más eficiente es DFS. Esto se deba posiblemente a que todas las soluciones son óptimas y se encuentran en un nodo hoja y DFS es el primero en extenderse hasta las hojas. Por otra parte Profundización Iterativa (*ID*) y BFS expanden muchos más nodos, haciendo que tarden más en encontrar dicha solución.

Por otro lado, si consideramos los algoritmos informados, podríamos decir que la heurística h2 resulta más efectiva en tableros grandes pero si consideramos tableros pequeños la cantidad de pares válidos tiene mayor posibilidad de coincidir con las fichas jugables por lo que los tiempos son similares.

Cuando consideramos el tablero tridimensional vemos que la máquina virtual en algunos casos se queda sin memoria mientras que en otros corre por tiempo indeterminado. Atribuimos esto a algunas de las limitaciones que mencionamos.

Para finalizar el análisis decidimos agrupar las estrategias utilizadas en:

- Algoritmos por nivel: A* BFS ID

para tableros pequeños equiparan performance, pero para el tablero “grande” A* corre por tiempo indefinido (>1h) mientras que los otros se quedan sin memoria, por la gran cantidad de nodos que expanden.

- Algoritmos de profundidad: Greedy, DFS

para tableros pequeños DFS resulta de gran eficiencia, mientras que greedy corriendo según H2 logra superarlo para tableros más grandes, notamos nuevamente que greedy corre por tiempo indefinido (>1h) para el tablero grande mientras que DFS rápidamente agota la memoria del sistema.

Anexo

Tabla de resultados[8]

	BFS	**DFS**	**IDDFS**
Nodos Expandidos	one:14 two:161 three:11562 four:9 five:OutOfMemory	one: 5 two: 7 three: 11 four: 4 five:OutOfMemory	one: 8 two: 11 three: 15 four: 7 five:OutOfMemory
Profundidad de la solución	one:5 two:7 three:11 four:4 five:OutOfMemory	one: 5 two: 7 three: 11 four: 4 five:OutOfMemory	one: 5 two: 7 three: 11 four: 4 five: OutOfMemory
Estados Generados	one:19 two:235 three:16646 four:12 five:OutOfMemory	one: 7 two: 11 three: 20 four: 6 five:OutOfMemory	one: 10 two: 15 three: 24 four: 10 five:OutOfMemory
Nodos en frontera	one:5 two:74 three:5084 four:3 five:OutOfMemory	one: 2 two: 4 three: 9 four: 2 five:OutOfMemory	one: 0* two: 0* three: 0* four: 0* five:OutOfMemory
Tiempo de ejecución	one: 0.0040s two: 0.0026s three:26.64s four:0.0s (despreciable)	one: 0.0040s two: 0.0020s three: 0.0020s four: 0.0010s five:OutOfMemory	one: 0.011s two: 0.002s three: 0.004s four: 0.003s

[8]Cálculos realizados con procesador Intel i5 de 2.6Mhz bajo Mac OSX

*: Se considera que éste cálculo resulta erroneo. Se debe revisar la implementación.

	five:OutOfMemory		five:OutOfMemory

	Greedy-H1	**A*-H1**	**Greedy-H2**	**A*-H2**
Nodos Expandidos	one: 14 two: 161 three: 11562 four: 9 five: --	one: 14 two: 161 three: 11562 four: 9 five:	one: 5 two: 7 three: 49 four: 5 five: --	one: 7 two: 52 three: 3995 four: 5 five:
Profundidad de la solución	one: 5 two: 7 three: 11 four: 4 five: --	one: 5 two: 7 three: 11 four: 4 five:	one: 5 two: 7 three: 11 four: 4 five: --	one: 5 two: 7 three: 11 four: 4 five:
Estados Generados	one: 19 two: 235 three: 16646 four: 12 five: --	one: 19 two: 235 three: 16646 four: 12 five:	one: 7 two: 11 three: 81 four: 7 five: --	one: 10 two: 86 three: 6551 four: 7 five:
Nodos en frontera	one: 5 two: 74 three: 5084 four: 3 five: --	one: 5 two: 74 three: 5084 four: 3 five:	one: 2 two: 4 three: 32 four: 2 five: --	one: 3 two: 34 three: 2556 four: 2 five:
Tiempo de ejecución	one: 0.011s two: 0.051s three: 35.994s four: 0.001s five: >60min	one: 0.003s two: 0.068s three: 34.79s four: 0.001s five:	one: 0.001s two: 0.002s three: 0.027s four: 0.0001s five: >60min	one: 0.002s two: 0.028s three: 11.02s four: 0.001s five: