BÚSQUEDA HEURÍSTICA A*

Aplicada al juego Tangram

Inteligencia Artificial

22 de noviembre de 2017

Bonet Peinado, Daiana
FAI - 238
daiana.bonet@fi.uncoma.edu.ar

de la Fuente, Juan
FAI-524
juan.delafuente@fi.uncoma.edu.ar

Unversidad Nacional del Comahue
Facultad de Informática

Índice general

ENUNCIADO

Ejercicio 1: Defina una heurística para el problema (se puede dar más).

Ejercicio 2: Adapte el módulo *h.pl* presentado por la cátedra, de manera tal que pueda resolver dicho problema usando la heurística del ejercicio anterior.

VERSIONES

El código implementado en la solución se encuentra disponible y accesible en el siguiente enlace: *GitHub*

Cuenta con dos versiones: *h* y *h_ v2* que difieren en el cálculo del costo. Ésto se encuentra detallado en el informe a continuación.

SOLUCIÓN

Introducción

El problema asignado consiste en encontrar e implementar una heurística en Prolog para el algoritmo de búsqueda *A** que resuelva el juego **Tangram**.

Éste juego consiste en formar siluetas de figuras con las cinco piezas dadas sin solaparlas, logrando ocupar toda la silueta con las mismas.

En esta instancia particular del juego, se cuenta con un estado inicial avanzado -es decir, existen piezas ya acomodadas-, una grilla de tamaño fijo de 6 columnas y 5 filas -representada con una matriz-, un listado de piezas que aún falta colocar y un estado final consistente con la forma de un cuadrado.

A continuación se explica el desarrollo logrado y las consideraciones necesarias.

Heurísticas

Es necesario en primera instancia definir determinados conceptos como marco teórico,

Definición

Una **Heurística** es una función de evaluación que se puede implementar con el *costo estimado* del camino menos costoso desde el nodo que estamos evaluando a una meta.

Análisis

El objetivo del **Tangram** en términos sencillos es colocar todas las piezas que aún no se encuentren en su lugar para formar la silueta. Con esto, se deduce de forma sencilla que la meta o *goal* es la silueta totalmente ocupada con las piezas en la posición correcta.

Esta meta nos remite a un problema ya tratado en clase: *8-puzzle*.

En este juego, es necesario lograr que las piezas se encuentren en un orden determinado por el valor que representan. La meta, es tener todas las piezas ordenadas de forma creciente -o análogamente, en un lugar específico como en **Tangram**-.

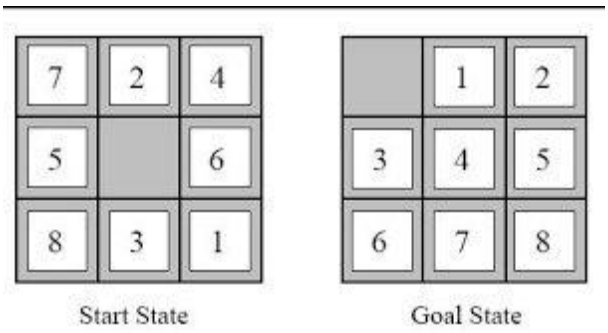

Figura 1: Representación del juego *8-puzzle*

Cómo ya hemos visto, una heurística para éste juego, consiste en *contar la cantidad de piezas que están fuera de posición*. De la misma manera, podemos definir ésta heurística cómo **Heurística 1** para nuestro problema.

Por otra parte, con un enfoque más particular a la dinámica del juego, podemos pensar en la forma en que *naturalmente* se intenta resolver el *puzzle*.

Cada vez que intentamos colocar una pieza, buscamos que la misma se encuentre limitando con otra pieza ya colocada o con un *borde* de la silueta. Dicho de otra manera, se busca que la pieza a colocar no limite con -o minimice- los espacios no ocupados ya que mientras menos espacios libres hay, más cerca estamos de completar el *puzzle*. Con esto llegamos a nuestra **Heurística 2:**

(Cantidad de piezas que limitan con algún espacio no-ocupado) / 2

Ya teniendo una base sobre la cual trabajar, es necesario saber cuando una una heurística es válida.

Definición

Sea h una Heurística, diremos que es **admisible** si **subestima** el costo, esto es, si su estimación del costo a la meta es **menor** o **igual** que el costo real. Si g es el costo real para

llegar desde el nodo actual n a la meta entonces $h(n) \leq g(n)$

Comprobación

Heurística 2

Consideremos los siguientes estados del juego:

Figura 2: Estados del juego

Aplicando la función heurística 2 h_2 sobre el estado *inicial*

$$h_2(\text{inicial}) = 5/2 \approx 2$$

y sabiendo que el costo real $g(inicial) = 2$ encontramos que parece cumplir con una heurística admisible, sin embargo, al colocar una de las piezas hallamos

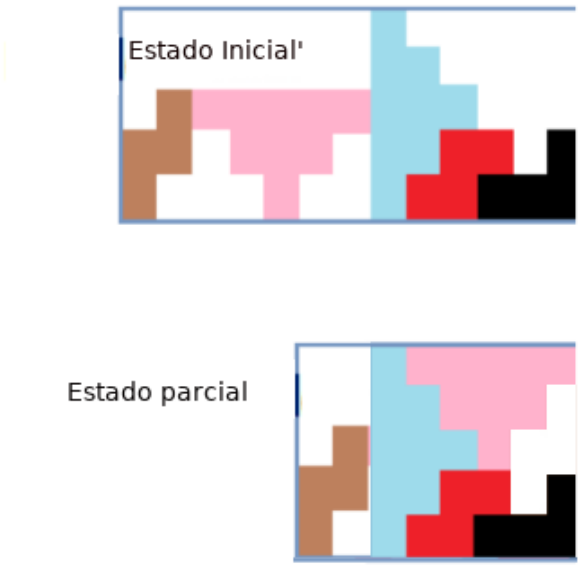

Figura 3: Estado cercano a la meta del juego

Aplicando la función heurística 2 h_2 sobre el estado *inicial'*

$$h_2(\text{inicial'}) = 4/2 = 2$$

y el costo real es $g(\textit{inicial'}) = 1$. Lo cuál ya deja de ser admisible al superar el *costo real* de lograr la meta. De ésta manera, descartamos la **heurística 2**.

Finalmente la **Heurística 1** resulta ser admisible para todos los valores reales posibles, ya que puede ser *'optimista'* al colocar una pieza en un lugar que no es su posición en el estado *meta* (ver figura 4) pero ya se descuenta de las piezas que faltan colocar.

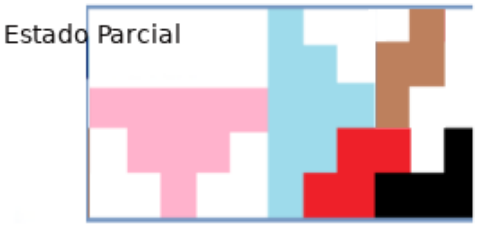

Figura 4: Ejemplo de cálculo: $h_1(n) = 1$ sin embargo $g(n) = 2$

Por el mismo motivo, la **heurística 1** no resulta ser una *buena heurística* para aproximar el costo real.

Ejemplo de cómo funcionaría el algoritmo **A*** con los costos reales y de **heurística 1**.

El ejemplo en realidad es con ramificación de factor 3, sin embargo se muestran las figuras a continuación por cuestiones de legibilidad.

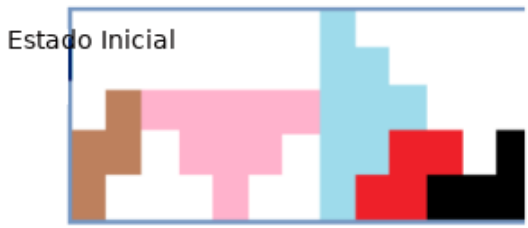

Figura 5: Ejemplo de cálculo: $h_1(n) = 2$ y $g(n) = 2$

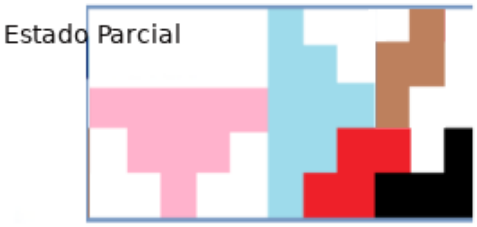

Figura 6: Ejemplo de cálculo: $h_1(n) = 1$ y $g(n) = 2$

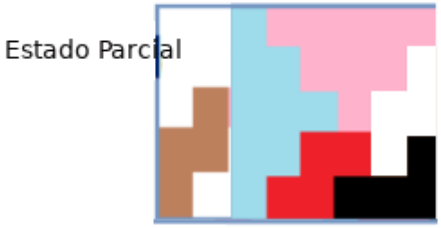

Figura 7: Ejemplo de cálculo: $h_1(n) = 1$ y $g(n) = 1$

Figura 8: Ejemplo de cálculo: $h_1(n) = 0$ y $g(n) = 0$

Costo

Ya hemos hablado y utilizado la función costo para corroborar que la heurística es válida,sin embargo aún no hemos establecido un costo más allá del intuitivamente utilizado. El costo que plantea la resolución lograda es uniforme y es de valor *1*. Esto se justifica considerando que cada movimiento de pieza independientemente del tamaño de la misma, cuenta cómo un único movimiento.

Se podrían haber establecido otros costos (de hecho, en la entrega se adjunta una versión con un costo distinto para verificar cómo se comporta el algoritmo ante otro valor) considerando el tamaño de la pieza, la cantidad de espacios en blanco que ocupa o -el que finalmente se incorpora en la otra versión- *la cantidad de piezas que aún no están en posición correcta con respecto al estado final.*

Dificultades halladas

Los inconvenientes encontrados para implementar el algoritmo **A*** y la función heurística **h** para **Tangram**, fueron mayormente problemas de representación de la instancia del juego en *Prolog*.

Se decidió que para representar la grilla se implementa una matriz de tamaño fijo en forma de *listas de listas* propio de *Prolog* con una lista de piezas separada de la grilla que '*contiene*' las piezas aún no colocadas.

También, las figuras se encuentran definidas cómo *hechos* con sus dimensiones -ancho principalmente-, número que la representa a cada una y su nombre, ya que existe un predicado con el nombre de cada pieza para '*colocarla*' en la matriz, permitiendo establecer predicados de *hi-order* y lograr una generalización un poco más abstracta del juego, facilitando implementar más piezas si se desea.

De ésta manera, es posible lograr representaciones del estado *meta*, las figuras y el estado inicial cómo se aprecia a continuación:

```
% is_goal(N) is true if N is a goal node.
is_goal([[1,2,2,2,2,2],
        [1,1,2,2,2,3],
        [1,1,1,2,3,3],
        [1,1,4,4,3,5],
        [1,4,4,5,5,5]]).

%Se considera la grilla de tamano fijo 6, no como en el enunciado.
init([[1,0,0,0,0,0],
        [1,1,0,0,0,0],
        [1,1,1,0,0,0],
        [1,1,4,4,0,5],
        [1,4,4,5,5,5]]).

%Lista de fichas sin acomodar en el estado init
listaFichasI([2,3]).

%Lista de fichas no insertadas, se calcula a partir de la lista inicial y
% 'quitando' de la lista las que se encuentren
% en Estado, para ello se revisa linea por linea el mismo
listaFichas(Estado,Lista):-
        listaFichasI(ListaI),
        nth(1,Estado,F1),subtract(ListaI,F1,R1),
        nth(2,Estado,F2),subtract(R1,F2,R2),
        nth(3,Estado,F3),subtract(R2,F3,R3),
        nth(4,Estado,F4),subtract(R3,F4,R4),
        nth(5,Estado,F5),subtract(R4,F5,Lista).
```

```
%Predicado reversible para obtener todos los datos de una ficha
%recuperar_numero(nombre,numero,anchoEnColumnas).
recuperar_numero(azul,1,3).
recuperar_numero(rosa,2,5).
recuperar_numero(marron,3,2).
recuperar_numero(rojo,4,3).
recuperar_numero(negro,5,3).

%insertar figuras para posicion especifica
%pieza(ColumnaInicial,Estado,NuevoEstado).
azul(N,[H1,H2,H3,H4,H5|To],[Hn1,Hn2,Hn3,Hn4,Hn5|To]):-
        insertarFicha(H1,N,1,1,Hn1),
        insertarFicha(H2,N,2,1,Hn2),
        insertarFicha(H3,N,3,1,Hn3),
        insertarFicha(H4,N,2,1,Hn4),
        insertarFicha(H5,N,1,1,Hn5).

negro(N,[H1,H2|To],[Hn1,Hn2|To]):-
        N1 is N+2,insertarFicha(H1,N1,1,5,Hn1),
        insertarFicha(H2,N,3,5,Hn2).

marron(N,[H1,H2,H3|To],[Hn1,Hn2,Hn3|To]):-
        N1 is N+1,insertarFicha(H1,N1,1,3,Hn1),
        insertarFicha(H2,N,2,3,Hn2),
        insertarFicha(H3,N,1,3,Hn3).

rosa(N,[H1,H2,H3|To],[Hn1,Hn2,Hn3|To]):-
  insertarFicha(H1,N,5,2,Hn1),
  N1 is N+1, insertarFicha(H2,N1,3,2,Hn2),
  N2 is N+2, insertarFicha(H3,N2,1,2,Hn3).

rojo(N,[H1,H2|To],[Hn1,Hn2|To]):-
  N1 is N+1, insertarFicha(H1,N1,2,4,Hn1),
  insertarFicha(H2,N,2,4,Hn2).
```

```
%Dada una fila,un desplazamiento N,una cantidad de piezas M
% un numero-ficha F,
% TRUE: retorna una fila con la ficha insertada en ella.
%insertarFicha(Estado,Desplazamiento,CantidadPiezas,Ficha,NuevoEstado).
insertarFicha([],0,0,_F,[]).
insertarFicha([H|T],0,0,_F,[H|T]).
insertarFicha([0|T],0,M,F,[F|Tn]):-
       M1 is M-1,insertarFicha(T,0,M1,F,Tn).
insertarFicha([],_,_,_,_):-false.
insertarFicha([H|T],N,M,F,[H|Tn]):-
       N1 is N-1, N1>=0,insertarFicha(T,N1,M,F,Tn).
```

Listing 1: Definicion de las matrices en forma de listas de listas

Esta forma de representación aporta generalidad a la implementación y una forma efectiva de representar estados dentro del *árbol* o lista de vecinos necesaria para la búsqueda, pero dificulta tanto recorrer la matriz para *colocar* una figura en su posición cómo la unificación correcta en el lugar adecuado.

El cálculo de la heurística se logra de forma sencilla al tener una lista de figuras sin colocar, ya que simplemente es consultar la longitud de la misma. Esta heurística se sabe que dista bastante de ser óptima para el problema, sin embargo es admisible y consigue tener una eficiencia relativamente buena para ejecutar el algoritmo y corroborar los resultados, lo que es significativo a nivel académico.

```
% cost(N,M,C) true if C is the arc cost for the arc from node N to node M
cost(N,M,C) :-
   neighbours(N,NN),
   member(M,NN),
   C is 1.


% Heuristica aceptable descripta en el informe como Heuristica 1.
% 'cantidad de fichas que faltan colocar'.
% Esta heuristica es teoricamente mas correcta ya que no se conoce 'goal'.
h(N,C) :-
       listaFichas(N,Fichas),
```

```
        length(Fichas,C).
```

Listing 2: Definición de la función heurística y el costo

La generación de los vecinos para el árbol que recorrerá **A***, resulta sencillamente del llamado de insertar una pieza en el estado dado. De esta manera, los vecinos son los diferentes llamados sobre las distintas piezas que queden disponibles dentro del listado de piezas. Esto facilita el llamado y la generación aunque es necesario destacar que se debe comenzar desde un estado inicial avanzado -pocas piezas restantes por colocar-, ya que el recorrido de las matrices para evaluar las inserciones de las piezas dentro de los posibles espacios ocupa tanto un considerable tiempo de computación cómo un elevado costo de espacio, también al considerar cómo estados distintos el corrimiento de una posición para cada pieza, el factor de ramificación del árbol de búsqueda es inmenso.

La generación de *neighbours* está definida por un predicado que calcula **todos** -*findall*- los estados a partir del dado y la inserción de cada una de las piezas que faltan colocar (considerando que se puede insertar sólo una por movimiento, es decir, todos los estados de colocar una pieza de las restantes en cada posición que sea posible).

Su definición está dada por los siguientes predicados:

```
%Calcula todos los posibles estados desde Estado y colocar
% una de las piezas faltantes
neighbours(Estado,NuevosEstados):-
        listaFichas(Estado,Fichas),mover_fichas(Estado,Fichas,NuevosEstados).

%Dado un estado Estado y una lista Fichas, genera un listado de
% NuevosEstados con todas las Fichas insertadas
mover_fichas(_,[],[]).
mover_fichas(Estado,[H|T],Resultado):-
        recuperar_numero(Ficha,H,_),
        findall(Lista,mover_ficha(Estado,Ficha,Lista),Estados),
        mover_fichas(Estado,T,Te),union(Estados,Te,Resultado).

%Predicado que busca una posicion valida para Ficha en Estado, true si
% NuevoEstado es Estado con Ficha colocada en algun lugar
mover_ficha(Estado,Ficha,NuevoEstado):-
        posicion_valida(0,Estado,Ficha,NuevoEstado).
```

```
%Dada una columna N, un Estado, una Ficha, true si NuevoEstado es Estado
% con Ficha colocada a partir de la columna N
posicion_valida(6,_,_,_):- false.
posicion_valida(N,[H|Matriz],Ficha,EstadoNuevo):-
       fila_valida(N,[H|Matriz],Ficha,EstadoNuevo).
posicion_valida(N,Matriz,Ficha,EstadoNuevo):-
       N1 is N+1,N1<6,posicion_valida(N1,Matriz,Ficha,EstadoNuevo).

%True si en la columna N de la Fila que encabeza el Estado se puede insertar
% la Ficha, sin alterar el resto de la fila
fila_valida(_,[],_,_):-false.
fila_valida(N,[H|Matriz],Ficha,EstadoNuevo):-
       call(Ficha,N,[H|Matriz],EstadoNuevo).
fila_valida(N,[X|Fila],Ficha,[X|EstadoNuevo]):-
       fila_valida(N,Fila,Ficha,EstadoNuevo).
```

Listing 3: Definición de neighbours y predicados relacionados

Bibliografía

[1] Poole David L. and Mackworth Alan K. *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, New York, NY, USA, 2010.