Programación Lógica y Funcional
Tarea 3

Ángel García Báez
ZS24019400@estudiantes.uv.mx

Maestría en Inteligencia Artificial

IIIA Instituto de Investigaciones en Inteligencia Artificial
Universidad Veracruzana
Campus Sur, Calle Paseo Lote II, Sección 2a, No 112
Nuevo Xalapa, Xalapa, Ver., México 91097

12 de noviembre de 2024

1. Solución al problema de la mudanza

Me estoy cambiando de casa y debo llevar a mi casa nueva a mi perro, mi gato y mi hamster que, sobra decirlo, no se llevan muy bien entre ellos. Mi mini auto solo me permite llevar a una mascota conmigo. De manera que, por ejemplo, puedo llevarme al gato, dejando solos al hamster y al perro, pero no puedo dejar juntos a éste último y al gato, ni al gato y al perro. Escribir un programa en Prolog para encontrar los movimientos válidos para pasar todas mis mascota de una casa a otra. Implemente una solución al problema mediante una búsqueda en el espacio de soluciones del problema. [25 puntos]

Para la implementación de este problema se consulto el manual de swi-prolog en búsqueda de funciones que permitieran dar pie a comportamientos que permitieran estilizar las salidas.

Lo que se sabe del problema es que el gato no puede estar con el perro y el gato no puede estar con el hámster, por lo que esos son estados inválidos que no pueden ser visitados porque entran en conflicto.

A partir de ello, se deduce que el perro y el hámster sí pueden estar juntos, ya sea en la primer casa o en la segunda casa.

Bajo esa premisa, se construyo el código de forma que verifique que las mascotas no entren en conflicto mientras van siendo transportadas de 1 en 1 de la casa 1 a la casa 2 como sigue:

```
% Estado inicial y estado objetivo
estado_inicial([due o/casa1, perro/casa1, gato/
   casa1, hamster/casa1]).
estado_final([due o/casa2, perro/casa2, gato/casa2,
    hamster/casa2]).
% Define el movimiento de cambio de casa 1 a casa 2
   y biceversa
opuesto(casa1, casa2).
opuesto(casa2, casa1).
% Estado seguro: El due o debe estar donde est n
   juntos el perro y el gato, o el gato y el
   h mster
estado_seguro([due o/D, perro/P, gato/G, hamster/H
   ]) :-
    (G = H -> D = G ; true),
    (P = G -> D = P ; true).
% Movimientos posibles
mover([due o/D, perro/D, gato/G, hamster/H], [
```

```
    due o/D2, perro/D2, gato/G, hamster/H]) :-
13      opuesto(D, D2),
14      estado_seguro([due o/D2, perro/D2, gato/G,
            hamster/H]).
15  mover([due o/D, perro/P, gato/D, hamster/H], [
    due o/D2, perro/P, gato/D2, hamster/H]) :-
16      opuesto(D, D2),
17      estado_seguro([due o/D2, perro/P, gato/D2,
            hamster/H]).
18  mover([due o/D, perro/P, gato/G, hamster/D], [
    due o/D2, perro/P, gato/G, hamster/D2]) :-
19      opuesto(D, D2),
20      estado_seguro([due o/D2, perro/P, gato/G,
            hamster/D2]).
21  mover([due o/D, perro/P, gato/G, hamster/H], [
    due o/D2, perro/P, gato/G, hamster/H]) :-
22      opuesto(D, D2),
23      estado_seguro([due o/D2, perro/P, gato/G,
            hamster/H]).
24  % Resuelve el problema, guardando el camino
25  resolver(Camino) :-
26      estado_inicial(Inicio),
27      buscar_camino(Inicio, [Inicio], Camino).
28  % Busca el camino al objetivo recursivamente
29  buscar_camino(Estado, Camino, Camino) :-
    estado_final(Estado).
30  buscar_camino(Estado, Visitados, Camino) :-
31      mover(Estado, SiguienteEstado),
32      \+ member(SiguienteEstado, Visitados),
33      buscar_camino(SiguienteEstado, [SiguienteEstado|
            Visitados], Camino).
34  % Imprime cada paso en el camino de forma
    ligeramente estilizada
35  mostrar_camino([]).
36  mostrar_camino([Paso|Resto]) :-
37      writeln(Paso),
38      mostrar_camino(Resto).
```

Al definir los posibles movimientos, definir el criterio de búsqueda, el recorrido de la casa 1 a la casa 2 y evitar que el programa caiga en bucles

logra que el programa construya poco a poco el camino a la solución como se muestra a continuación al pedirle la consulta correspondiente:

```
?. resolver(Camino), mostrar_camino(Camino).
[due o/casa2,perro/casa2,gato/casa2,hamster/casa2]
[due o/casa1,perro/casa2,gato/casa1,hamster/casa2]
[due o/casa2,perro/casa2,gato/casa1,hamster/casa2]
[due o/casa1,perro/casa2,gato/casa1,hamster/casa1]
[due o/casa2,perro/casa2,gato/casa2,hamster/casa1]
[due o/casa1,perro/casa1,gato/casa2,hamster/casa1]
[due o/casa2,perro/casa1,gato/casa2,hamster/casa1]
[due o/casa1,perro/casa1,gato/casa1,hamster/casa1]
```

La salida es la siguiente, junto con la salida que contiene toda la solución como una lista de listas que se ve más extensa, por lo que se decidió omitir para mostrarla en el documento, pero se menciona que dicha salida también corresponde a la consulta.

La solución a la que llega es mover primero al gato de casa 1 a casa 2, regresar por el perro a la casa 1 para llevarlo a la casa 2, devolver al gato de la casa 1 a la casa 2, llevar el hámster de la casa 1 a la casa 2 y finalmente, volver a llevar al gato de la casa 1 a la casa 2.

Nótese que la solución funciona tanto si se lee de arriba a abajo como de abajo hacia arriba.

2. Aplicación del algoritmo de "Mejor el primero"sobre el problema de las 8 reinas.

Aplique el algoritmo primero el mejor a un problema de su elección (diferente a los vistos en clase). Justifique la elección de sus predicados sucesor y meta. Justifique su función de costo y heurística. [**20 puntos**]:

Para poner en practica el uso del algoritmo de primero el mejor, se tomo el problema de las 8 reinas que viene descrito en el libro de Bratko (2001) al cual se le hicieron ciertas modificaciones a conveniencia para que el código pudiera funcionar empalmado con el algoritmo de búsqueda.

Para el predicado sucesor se propuso como:

```
%%% Sucesores: coloca una reina en la siguiente fila
s(Tablero, [Fila/Columna | Tablero], 1) :-
    length(Tablero, N), % Determina cu ntas reinas
       est n ya en el tablero
    Fila is N + 1, % Calcula la fila actual (
       siguiente vac a)
    between(1, 8, Columna), % Intenta ubicar la
       reina en una columna v lida
    \+ en_conflicto([Fila/Columna | Tablero]).
```

De esta forma se puede definir a un sucesor en función de la posición de la reina anterior con la finalidad de que no entre en conflicto la siguiente reina colocada con las anteriores, ademas que el costo unitario que tiene para avanzar de una fila a otra le permite avanzar de manera uniforme en la generación de los sucesores.

Para la meta se definió como sigue:

```
%%% Meta: la meta es tener 8 reinas en el tablero
   sin conflictos.
meta(Tablero) :-
    length(Tablero, 8),
    \+ en_conflicto(Tablero).
```

La idea principal del problema y que conduce a la solución, es poder acomodar a las 8 reinas en un tablero de 8x8 sin que estas entren en conflicto, por lo que la meta es eso, tener el tablero con las 8 reinas colocadas (1 por fila) de forma que no entren en conflicto entre todas ellas.

Sobre la heurística, se definió el siguiente predicado para ello:

```
h(Tablero, Conflictos) :-
```

```
    findall(1, (member(F1/C1, Tablero), member(F2/C2
       , Tablero), F1 < F2, conflicto(F1, C1, F2, C2
       )), ConflictosList),
    length(ConflictosList, Conflictos).
```

La función heurística mide la cantidad de conflictos que se presentan entre las reinas que fueron colocadas en el tablero. La heurística calcula potenciales conflictos que puedan existir entre las reinas basado en dos condiciones:

1.- Tener conflictos al estar presentes más de una reina sobre la misma columna.

2.- Verificar si hay conflictos entre las reinas de forma diagonal.

En base a esos posibles problemas que se pueden presentas, es que se guía la búsqueda para encontrar los acomodos de las reinas que generen una cantidad nula de conflictos entre ellas.

Sobre el costo, se decidió que fuera uniforme (costo 1) debido a que acomodar una nueva reina, implica moverse una fila hacia adelante, entonces esta idea fue reflejada en ese costo unitario junto con el hecho de que no se esta minimizando o maximizando una función objetivo como tal, dado que el caso es realizar el acomodo de las 8 reinas, donde cada colocación de una nueva reina no debería costar más que otra.

Finalmente, después de empalmar el algoritmo de búsqueda mejor el primero con la implementación hecha del problema de las reinas, a continuación se muestra el resultado al que se llega aplicándolo:

```
?-primeroMejor([],Sol).
Sol = [[8/6, 7/4, 6/7, 5/1, 4/3, 3/5, 2/2, 1/8],
   [7/4, 6/7, 5/1, 4/3, 3/5, 2/2, 1/8], [6/7, 5/1,
   4/3, 3/5, 2/2, 1/8], [5/1, 4/3, 3/5, 2/2, 1/8],
   [4/3, 3/5, 2/2, 1/8], [3/5, 2/2, 1/8], [2/2,
   1/8], [1/8], []]
```

La forma en que se implemento arroja más de una solución, en este caso, se procede a quedarse con la primer solución encontrada que consta del acomodo de las reinas tales que hay una reina en la fila 8 columna 6, otra en la fila 7 columna 4, la siguiente en la fila 6 columna 7, sigue la reina en la fila 5 columna 1, posteriormente la reina en la fila 3 columna 5, la reina en la fila 2 columna 2 y finalmente, la reina en la fila 1 columna 8. Dicha solución satisface el criterio de colocar a las 8 reinas sin que estas entren en conflicto.

El algoritmo se encuentra anexo en la entrega.

3. Mejora a la implementación clásica de ID3.

Revise el artículo de Quinlan, Induction of Decision Trees, Machine Learning 1: 81-106, 1986; con el objetivo de identificar aquellos aspectos que podrían mejorar nuestra implementación básica de ID3. Elija uno de ellos y agregue la mejora a nuestro programa. Reporte en una página la mejora elegida, el diseño experimental para verificar los efectos de la mejora y los resultados obtenidos. [35 puntos]

Con ayuda de las sugerencias propuestas en las diapositivas de Guerra-Hernández (2024) y a la lectura del articulo de Quinlan (1986) fue que se opto por cambiar la forma en que el algoritmo toma la decisión de los atributos que más explican la información, cambiando la entropía clásica propuesta por Quinlan a una mejora donde se toma en cuenta la ganancia de información como cociente como se expresa a continuación:

$$\text{splitInformation}(S, A) = -\sum_{i=1}^{c} \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|}$$

$$\text{gainRatio}(S, A) = \frac{\text{gain}(S, A)}{\text{splitInformation}(S, A)}$$

Esto se hizo con la finalidad de no privilegiar la selección de atributos con una gran cantidad de etiquetas distintas, la idea es medir mediante ese coeficiente que tanto ayuda cada atributo realmente a tomar la decisión.

Para poner a prueba esto se puso a prueba el algoritmo ID3 clasico sin modificaciones que veníamos viendo en clase contra la versión mejorada con esta modificación para la base de datos de zoo, la cual contiene información de animales de un zoológico. Se propone esta base debido a la presencia de un atributo que contiene el nombre de casi cada animal, por lo que es una variable con demasiadas etiquetas. La idea es que el algoritmo determine que esta no es una muy buena variable para la toma de decisión del árbol y que tome las que realmente están ayudando a explicar la toma la decisión.

A continuación se muestran los resultados de los arboles producidos por ambos algoritmos.

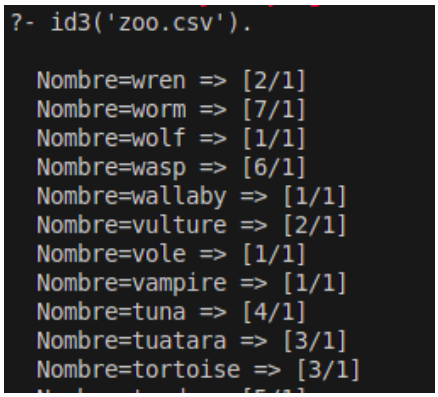

Figura 1: Imagen 1

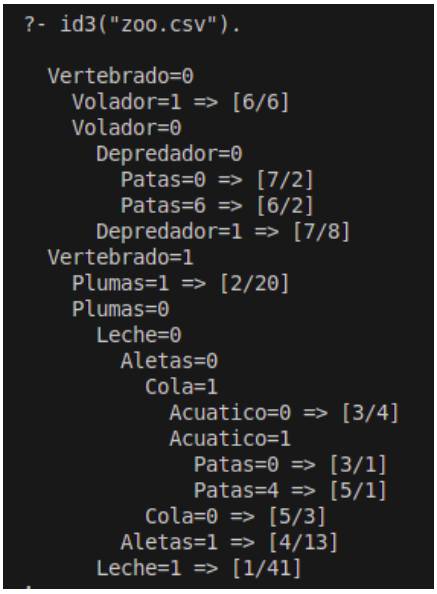

Figura 2: Imagen 2

Se observa como el ID3 clásico toma como variable que más explica todo el nombre de los animales, mientras que la versión modificada del ID3 presenta un árbol distinto en el cual omite dicha variable dado que puso en practica el nuevo criterio para determinar si un atributo conduce a la toma de decisión, logrando así explicar de forma más compacta el como llegar a clasificar cada uno de los animales en base a las características que más explican.

Referencias

Bratko, I. (2001). *Prolog programming for artificial intelligence*. Pearson education.

Guerra-Hernández, D. A. (2024). Programación Lógica y Funcional - Programación Lógica.

Quinlan, J. R. (1986). Induction of decision trees. *Machine learning*, 1:81–106.