Universidad Simón Bolívar
Dpto. de Computación y Tecnología de la Información
CI3661 - Taller de Lenguajes de Programación I
Enero-Marzo 2011

Programación Lógica

En este proyecto Ud. desarrollará en Prolog la infraestructura necesaria para resolver problemas determinísticos de información completa que se puedan estudiar como un modelo de estados. Problemas como el Cubo de Rubik, el n-puzzle y encontrar rutas en mapas son susceptibles de ser estudiados de esta manera.

Un **modelo de estados** se define como la tupla (S, s_0, G, A, a, f, c), tal que:

- S es un conjunto finito y discreto de estados.
- $s_0 \in S$ es el estado inicial.
- $G \subseteq S$ es el conjunto de estados objetivo.
- A es el conjunto de acciones.
- La función $a : S \to \mathbb{P}(A)$ tal que, dado un estado $s \in S$, $a(s)$ son todas las acciones aplicables en ese estado.
- La función de transición $f : (S, A) \to S$.
- La función de costo $c : (S, A) \to \mathbb{R}$.

Se desea desarrollar un sistema que al recibir la especificación en modelo de estados de un problema, lo resuelva automáticamente, entendiendo por **solución** la secuencia de acciones $a_0, \ldots, a_{n-1}$, que lleva desde el estado inicial s_0 hasta un estado objetivo $s_n \in G$ a través de la secuencia de estados $s_0, \ldots, s_n$; es decir, de tal forma que $s_{i+1} = f(a_i, s_i)$ y $a_i \in a(s_i)$. Una solución es **óptima** si minimiza

$$P_n = \Sigma_{i=0}^{n-1} c(s_i, a_i)$$

que corresponde al costo total.

¿Cómo encontrar soluciones?

Existen varios algoritmos para la búsqueda progresiva en grafos de soluciones, pero para este proyecto se requiere que Ud. investigue el algoritmo de búsqueda informada A^*[1] a ser utilizada para resolver problemas especificados usando modelos de estados.

Ud. deberá proveer una implantación para el predicado:

```
a_star(Start,Actions)
```

que se satisface cuando `Actions` es la lista de acciones de costo mínimo obtenidas a través del algoritmo A^* que, al ser aplicadas sobre el estado inicial `Start`, llevan a un estado objetivo. Si hay varias soluciones de costo mínimo, el predicado se satisface con todas éstas.

Es importante notar que los algoritmos de búsqueda informada, como A^*, son *independientes* del problema en particular. Por tanto, su implantación del algoritmo debe ser lo más genérica posible. Los *únicos* predicados dependientes del problema a utilizar son los descritos a continuación:

- `action(Start,Action,End)`, que se satisface si en el estado inicial `Start` se puede aplicar la acción `Action`, llegando al estado final `End`.
- `heuristic(State,Value)`, que se satisface cuando `Value` es el valor de la heurística para el estado `State`, tal como es definida en el algoritmo A^*. Para ser admisible, esta heurística no puede sobreestimar el costo de llegar del estado actual a algún estado objetivo.
- `cost(State,Action,Cost)`, que se satisface cuando `Cost` es el costo de realizar la acción `Action` en el estado `State`.
- `objective(State)`, que se satisface cuando el estado `State` es un estado objetivo.

Implantación

Para demostrar que su implantación de A^* es completamente general, Ud. debera utilizar el mismo conjunto de predicados para resolver *dos* problemas: el puente frágil y el n-puzzle.

El puente frágil

Cuatro exploradores (Curly, Larry, Moe y Shemp) están tratando de atravesar un puente de roca dentro de una cueva. El puente es demasiado frágil y solamente soporta el peso de *dos* personas al mismo tiempo. Más aún, la cueva es oscura y el puente es angosto, así que necesitan utilizar una linterna para poder cruzar sin tropezarse. El problema es que solamente tienen una linterna a la cual le restan 60 minutos de batería.

Además, la torpeza de los exploradores hace que cada uno demore diferentes cantidades de tiempo en cruzar el puente en cualquiera de las direcciones:

Explorador	Tiempo en Cruzar
Curly	20 min
Larry	10 min
Moe	5 min
Shemp	25 min

Es obvio que no pueden cruzar todos al mismo tiempo, pues el puente solamente soporta el peso de dos cualesquiera de ellos. Por otro lado, también es obvio que siendo necesaria la linterna para cruzar, cada vez que dos de ellos crucen, alguno de los dos deberá cruzar de regreso para traer la linterna. Entonces, ¿en cuál orden deben cruzar el puente sin que se agote la batería?

Para garantizar la evaluación de las destrezas pertinentes a este curso, Ud. deberá completar su implantación particular según la siguiente especificación:

- Los estados serán denotados con el functor `state/3`. El primer argumento del functor `state` será una lista con los exploradores que están en el lado izquierdo del puente, el segundo argumento será una lista con los exploradores que están en el lado derecho del puente y el tercer argumento será un átomo `left` o `right` para indicar en cuál lado del puente se encuentra la linterna. Los exploradores serán denotados con los átomos `curly`, `larry`, `moe` y `shemp`, naturalmente.
- Las acciones serán denotadas con el functor `move/2`. El primer argumento del functor move será la dirección del movimiento denotada con el átomo `left` o `right`, y el segundo argumento será una lista con los exploradores que participan en el movimiento.

- La deducción de la heurística a utilizar corre por su cuenta.
- El costo de cada acción corresponde al máximo entre los tiempos de cruce de los participantes en el movimiento.
- El estado inicial siempre será

  ```
  state([curly,larry,moe,shemp],[],left)
  ```

  y el estado objetivo es

  ```
  state([],[curly,larry,moe,shemp],right)
  ```

  o *cualquier* permutación de la segunda lista, por supuesto.

Será necesario que implante el predicado `showmoves(State,Actions)` que permita mostrar por la salida estándar los estados sucesivos del cruce dada una secuencia de acciones. Este predicado deberá presentar el estado inicial `State` a base de caracteres, y luego ir mostrando los estados sucesivos después de la aplicación de cada una de las acciones. Por ejemplo, el estado inicial debería verse como

```
curly larry moe shemp @ |_______|
```

mientras que un estado intermedio podría ser

```
curly       moe         |_______| @       larry     shemp
```

y el estado final

```
                        |_______| @ curly larry moe shemp
```

n-puzzle

El juego de n-puzzle[2] consiste de un cuadrado en el que hay n fichas deslizantes enumeradas del 1 al n. Las fichas están colocadas en una cuadrícula de $k \times k$ siendo $k = \sqrt{n+1}$. El objetivo del juego es ordenar las fichas en orden creciente de izquierda a derecha y de arriba hacia abajo. Este juego ha sido estudiado profundamente en la literatura de Inteligencia Artificial, así que no tendrá ninguna dificultad en encontrar respuesta a todas las dudas que pueda tener en cuanto al juego mismo y las estrategias para resolverlo.

Para garantizar la evaluación de las destrezas pertinentes a este curso, Ud. deberá completar su implantación particular según la siguiente especificación:

- Los estados serán representados como una lista de k elementos. Cada elemento será a su vez una lista de k elementos. Cada sublista corresponde a una *fila* de la cuadrícula, de arriba hacia abajo, y contienen los elementos de la fila según su posición de izquierda a derecha. Los elementos serán representados por los dígitos de 1 a n y el átomo `empty`.
- Las acciones serán denotadas con los átomos `up`, `down`, `left` y `right`, para indicar la dirección en la que ha de moverse la ficha hacia el espacio vacío.

- Utilizaremos la heurística conocida como distancia Manhattan. El valor de esta heurística consiste en la suma de las distancias Manhattan de *todas* las fichas. La distancia Manhattan de una ficha es el número de movimientos que habría que hacer para llevarla a su posición final suponiendo que no hay más fichas en la cuadrícula. Esto puede calcularse con una fórmula aritmética trivial de deducir.
- Todas las acciones tendrán costo 1.
- El único estado objetivo en el juego de n-puzzle sería entonces

```
[[1,2,...,k],...,[(k-1)k+1,...,k*k-1,empty]]
```

Será necesario que implante el predicado `showmoves(State,Actions)` que permita mostrar por la salida estándar los estados sucesivos del tablero dada una secuencia de acciones. Este predicado deberá presentar el estado inicial `State` como una cuadrícula simple a base de caracteres, y luego ir mostrando los estados sucesivos después de la aplicación de cada una de las acciones. Por ejemplo, si estuviera resolviendo 15-puzzle, un estado cualquier ha de verse como

```
+--+--+--+--+
| 2| 6| 7|11|
+--+--+--+--+
| 3|  | 5| 8|
+--+--+--+--+
| 1|10|12| 4|
+--+--+--+--+
|15|13|14| 9|
+--+--+--+--+
```

Su implantación debe ser lo más genérica posible, siendo capaz de funcionar para *cualquier* n. En la práctica su proyecto debe funcionar sin problemas con $n = 3$ y $n = 8$, y posiblemente con algunas instancias de resolver con $n = 15$ o $n = 24$ previa parametrización del interpretador de Prolog en una máquina con suficientes recursos, pues el espacio de búsqueda es $\theta(n!)$.

Entrega de la Implantación

Ud. debe entregar un archivo `.tar.gz` o `.tar.bz2` (**no** puede ser `.rar`, ni `.zip`) que al expandirse genere un directorio con el nombre de su grupo (e.g. `G42`) dentro del cual encontrar **solamente**:

- El archivo `astar.pro` conteniendo el código fuente Prolog para implantar los predicados generales de A^*, y los archivos `bridge.pro` y `npuzzle.pro` que contienen el código fuente Prolog para implantar los predicados específicos para el problema correspondiente.
- Se evaluará el correcto estilo de programación [4]:
  - Indentación adecuada y consistente en *cualquier* editor – la excusa "profe, en el mío se ve bien" es inaceptable.

    - Cada predicado debe estar completa y correctamente documentado usando el sistema Simple de especificación de argumentos. Es **inaceptable** que la documentación tenga errores ortográficos.

- **Fecha de Entrega.** Viernes 2011-03-11 (Semana 9) hasta las 17:59 VET
- **Valor de Evaluación.** Veinticinco (25) puntos.

Referencias

[1] Algoritmo de Búsqueda A^* según Wikipedia
http://en.wikipedia.org/wiki/A*_search_algorithm

[2] *Fifteen Puzzle* (una instancia de n-puzzle) según Wikipedia
http://en.wikipedia.org/wiki/Fifteen_puzzle

[3] Aplicación de A^* para búsqueda de rutas en un juego
http://theory.stanford.edu/~amitp/GameProgramming/

[4] http://www.ai.uga.edu/mc/plcoding.pdf