INSTITUTO TECNOLÓGICO DE COSTA RICA **II SEMESTRE 2020**
INGENIERÍA EN COMPUTACIÓN
IC-4700 LENGUAJES DE PROGRAMACIÓN **TAREA PROGRAMADA 2**

Los grafos del espacio de estados son usados para representar problemas. Los nodos del grafo son estados de un problema. Una arista existe entre dos nodos si existe una regla de transición, también llamada movida, que transforme un estado en el otro. Resolver un problema consiste en encontrar una ruta que a partir de un estado inicial dado, alcance un estado de solución deseado siguiendo una secuencia de movidas.

Búsqueda por profundida: *Depth first*

En este tipo de búsqueda el grafo es explorado siguiendo sistemáticamente las aristas más anidadas primero. A continuación se muestra un marco general para resolver problemas usando búsqueda *depth-first* en un grafo de espacio de estados. *Movidas* es la secuencia de movidas requeridas para alcanzar un estado final a partir de *Estado*, donde *Historia* contiene los estados previamente visitados.

```
solve_dfs(Estado,Historia,[]) :- final_state(Estado).
solve_dfs(Estado,Historia,[Movida|Movidas]) :-
      move(Estado,Movida),
      update(Estado,Movida,Estado2),
      legal(Estado2),
      not member(Estado2,Historia),
      solve_dfs(Estado2,[Estado2|Historia],Movidas).
```

Para probar el marco se utiliza el siguiente código. `Problema` es el nombre del problema que se quiere resolver y `Movidas` mostrará la solución si hay alguna.

```
test_dfs(Problema,Movidas) :-
     initial_state(Problema,Estado),
     solve_dfs(Estado,[Estado],Movidas).
```

El marco no indica cómo serán representados los estados. Las movidas son especificadas por medio de un predicado `move(Estado,Movida)`, donde `Movida` es una movida aplicable a `Estado`. El predicado `update(Estado,Movida,Estado2)` encuentra un `Estado2` que se puede alcanzar usando `Movida` a partir de `Estado`. Es posible tener otro marco que mezcle la movida con la actualización de estados.

La validez de una posible movida es revisada por el predicado `legal(Estado2)` que revisa si `Estado2` cumple con las restricciones del problema. El programa lleva una historia de los estados visitados para evitar caer en un ciclo infinito. Esto se realiza revisando que el nuevo estado no aparezca en la historia. La secuencia de movidas que llevan del estado inicial al estado final es construida incrementalmente en el tercer argumento de `solve_dfs`.

Para resolver un problema usando el marco anterior, el programador debe decidir cómo representar los estados y cómo especificar los procedimientos de `move`, `update` y `legal`. Una representación adecuada es fundamental para el éxito del marco.

El marco se puede usar para resolver por ejemplo el problema de la zorra, la cabra y el maíz: un granjero tiene una zorra, una cabra y un saco de maíz, y se encuentra en la margen izquierda de un río. El granjero tiene un bote que puede llevarlo a él y alguna de esas tres cosas y él debe transportar todo a la otra margen del río. El problema es que no puede dejar la zorra y la gallina solos porque no quedaría gallina, ni tampoco puede dejar la gallina y el maíz solos porque desaparecería el maíz. Determinar cómo el granjero debe transportar las cosas para que no pierda ninguna.

Los estados para el problema de la zorra, la gallina y el maíz son estructuras de la forma

```
zgm(Bote,Izq,Der)
```

donde `Bote` es el lado del río donde el bote se encuentra en ese momento, `Izq` es la lista de ocupantes de la margen izquierda del río, y `Der` es la lista de ocupantes de la margen derecha.

```
update(zgm(B,I,D),Carga,zgm(B1,I1,D1)):-
update_Bote(B,B1),
update_margenes(Carga,B,I,D,I1,D1).


initial_state(zgm,zgm(izq,[zorra,gallina,maiz],[])).
final_state(zgm(der,[],[zorra,gallina,maiz])).

move(zgm(izq,I,D),Carga):-member(Carga,I).
move(zgm(der,I,D),Carga):-member(Carga,D).
move(zgm(B,I,D),solo).

update_Bote(izq,der).
update_Bote(der,izq).

update_margenes(solo,B,I,D,I,D).
update_margenes(Carga,izq,I,D,I1,D1):-
      select(Carga,I,I1),
      insert(Carga,D,D1).
update_margenes(Carga,der,I,D,I1,D1):-
      select(Carga,D,D1),
      insert(Carga,I,I1).

insert(X,[Y|Ys],[X,Y|Ys]):-precedes(X,Y).
insert(X,[Y|Ys],[Y|Zs]):-precedes(Y,X),insert(X,Ys,Zs).
insert(X,[],[X]).

select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]):-select(X,Ys,Zs).

precedes(zorra,X).
precedes(X,maiz).

legal(zgm(izq,I,D)):-not ilegal(D).
legal(zgm(der,I,D)):-not ilegal(I).

ilegal(L):-member(zorra,L),member(gallina,L).
ilegal(L):-member(gallina,L),member(maiz,L).
```

La gran mayoría de los problemas interesantes tienen un espacio de búsqueda demasiado grande para ser recorrido tan exhaustivamente como lo hace *depth-first*. Dos heurísticas que se pueden

usar para realizar una búsqueda más racional son *hill-climbing* y *best-first-search*. Ambas dirigen la búsqueda de la solución basándose en una función de evaluación que aproximadamente mide la "cercanía" de un estado a la solución.

Hill climbing

Es una generalización de *Depth-first* donde el estado sucesor escogido es aquel que tenga la más alta evaluación entre los posibles estados, y no simplemente el primero generado. El archivo que se adjunta **`hill-climbing.pl`** muestra un programa que implementa dicho tipo de búsqueda y lo usa para resolver el problema del granjero anterior. El programa se usa el predicado **`findall`** para generar todas las posibles movidas a partir de un estado. Luego el predicado **`evaluate_and_order`** evalúa todas las movidas y genera una lista ordenada de pares (movida valor).

Best-first search

Este tipo de búsqueda toma un vistazo global a todo el espacio de búsqueda. En cada iteración, el mejor estado de todos aquellos que no han sido recorridos es el escogido. El archivo que se adjunta **`best-first.pl`** muestra un programa que implementa este tipo de búsqueda. Una "frontera" es mantenida conforme la búsqueda avanza; en ella se almacenan ordenados de acuerdo con alguna función de evaluación aquellos estados que aún no han sido recorridos. A cada paso, la movida que genera el estado con la mejor evaluación es escogida. Se debe llevar un conjunto de movidas en lugar de solo una; de ahí que predicados como **`updates`** y **`legals`** trabajan sobre conjuntos de movidas. Una desventaja de este método es que la ruta de movidas debe ser explícitamente mantenida dentro del estado ya que los estados pueden ser explorados en partes muy remotas del grafo.

Tarea

La tarea programada consiste en resolver el siguiente problema de modo que se puedan aplicar los tres mecanismos de búsqueda anteriores. Para ello debe modelar una estructura estado que registro el estado en que se encuentra el problema. Debe implementar las siguientas siguientes relaciones:

- **`final_state`**
- **`move`**
- **`update`**
- **`legal`**
- **`initial_state`**
- **`value`** solo para hill-climbing y best-first

Problema

Unos amigos van caminando de regreso a su albergue y se topan con un desvenciajdo puente de madera vieja. El puente es muy débil y solo es capaz de soportar el peso de algunos de ellos a la vez. Tienen prisa ya que es anocheciendo y deben cruzar en el menor tiempo posible. Llevan una linterna que permite que los que van cruzando el puente puedan ver, pero que no sirve para iluminar todo el puente. De modo que si varios de ellos cruzan, uno de los que cruzan debe devolverse con la linterna para permitir que los otros puedan cruzar.

Debido a que todos ellos tienen diferentes niveles de condición física, y algunos están lastimados, a cada uno de ellos le toma un tiempo distinto cruzar el puente.

Escriba un conjunto de relaciones de Prolog que modele el problema anterior y permita encontrar una forma de que todos puedan cruzar el puente en un tiempo total no mayor a un valor dado. Se debe poder adjuntar esas relaciones a los tres métodos de búsqueda anteriores y resolver el problema.

Probar su programa para los siguientes casos.

Caso 1. Se tienen las siguientes personas y tiempos para pasar el puente:

Alberto	1
Beatriz	2
Carlos	5
Dora	10
Emilio	15

Pueden pasar por el puente 2 a la vez, intentar hacerlo en 28 minutos.

Caso 2. Mismas personas y tiempos del caso 1.
Pueden pasar 3 a la vez, intentar hacerlo en 21 minutos.

Caso 3. Se agrega una persona más al caso 1:

Julio	20

Pueden pasar por el puente 2 a la vez, intentar hacerlo en 42 minutos.

Caso 4. Mismas personas y tiempos del caso 3.
Pueden pasar 3 a la vez, intentar hacerlo en 30 minutos.

Consideraciones finales

Para la tarea puede usar SWI-Prolog, el cual está disponible en SWI-Prolog downloads. Ese es el motor básico del lenguaje. Luego de instalarlo puede usar SWI-Prolog-Editor para disponer de un ambiente más amigable con editor integrado. La facilidad de ***trace*** es particularmente útil. Un par de comandos de trace muy convenientes son ***creep*** (ENTER) que ejecuta un paso a la vez, y ***skip*** ('s') que resuelve una submeta en un solo paso.

La fecha de entrega de esta tarea es el lunes 30 de noviembre a las 10pm.