Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

Tarea 1 – Bloques Cooperativos y Competitivos
IIC2613 - Inteligencia Artificial
Segundo Semestre, 2017
Entrega: 29 de Septiembre, hasta las 23:59

1. Objetivo

El objetivo de esta tarea es que usted practique conceptos de programación en lógica, aplicado a la inteligencia artificial. En particular veremos cómo es posible representar problemas de búsqueda en los que dos agentes cooperan o compiten.

2. El Mundo de Bloques con Colores

El mundo de bloques con colores (MBC) es similar al mundo de bloques tradicional: se supone la existencia de una mano robótica capaz de tomar un bloque que está al tope de alguna torre de bloques, para desplazarlo sobre la mesa o sobre otro bloque. La mesa puede almacenar un número ilimitado de torres.

El MBC extiende al mundo de bloques con dos agentes (robóticos) más: *rob* y *bor*. El primero (rob), puede pintar bloques que están al tope de alguna torre, mientras que el segundo puede pintar bloques que están sobre la mesa. En todo momento, cada bloque tiene un único color.

Para representar el MBC usamos una representación un poco distinta al mundo de bloques visto en clases. Las siguientes proposiciones que usamos:

- $on(x, y)$: es verdadero cuando el bloque x está directamente sobre el bloque y.
- $ontable(x)$: es verdadero cuando el bloque x está directamente sobre la mesa.
- $clear(x)$: es verdadero cuando el bloque x se encuentra al tope de una torre de bloques (es decir, está "libre").
- $color(b, c)$: es verdadero cuando el color del bloque b es c.

Las acciones del MBC son:

- $move\_to\_table(x)$: mueve al bloque x sobre la mesa.
- $move\_to\_block(x, y)$: mueve el bloque x sobre el bloque y.
- $paint(g, x, c)$: el agente g ($g \in \{rob, bor\}$) pinta el bloque x de color c.

3. Parte 1: Búsqueda Cooperativa

Complete el archivo base `tarea1_base.pl` disponible en el sitio del curso en la sección *parte 1*. Como notará, nuestra implementación se basa en el ejemplo de clases, que es una forma de usar el *Cálculo de Situaciones* para resolver este problema. Una diferencia importante es que ahora se representan los efectos de las acciones mediante *efectos condicionales*. En particular, se usan los predicados:

- `is_conditional_positive_effect(A,C,F)` para expresar que `F` es verdadero como resultado de ejecutar `A` si es que `C` se cumplía en la situación donde se ejecutó `A`.
- `is_conditional_negative_effect(A,C,F)` para expresar que `F` es falso como resultado de ejecutar `A` si es que `C` se cumplía en la situación donde se ejecutó `A`.

Para comprender completamente cómo es que funcionan estos predicados, recomiendo que usted mire la definición del predicado `holds` del archivo base y la compare con la definición del mismo predicado en el ejemplo de clases.

El predicado `legal` (definido en el archivo base) enumera situaciones alcanzables desde `s0`. De esta manera, una vez que usted complete esta parte, debiera poder hacer las siguientes consultas:[1]

```
?- legal(S),holds(color(c,blue),S).
S = do(paint(rob, c, blue), do(move_to_block(b, a), do(move_to_block(a, d), s0)))

?- legal(S),holds(color(c,blue),S),holds(color(b,red),S).
S = do(paint(bor, b, red), do(paint(rob, c, blue),
       do(move_to_table(b), do(move_to_block(a, d), s0))))
```

Observe que podemos entender estas soluciones como "cooperativas" en el sentido que tanto rob como bor trabajan en conjunto para resolver el problema.

4. Parte 2: Búsqueda Competitiva

A veces es razonable considerar al ambiente como un adversario. Este es frecuentemente el enfoque que se toma en el problema conocido como *síntesis de controladores*. En ese problema el objetivo es similar al de búsqueda, pero suponemos que el ambiente puede efectuar acciones arbitrarias que podrían "estropear" lo que hemos hecho. La síntesis de controladores tiene una íntima relación con el problema de generación automática de programas, en donde podríamos pensar que el usuario (el input del programa) es el "ambiente". En esta parte veremos cómo resolver un caso particular de la síntesis de controladores usando Prolog.

En nuestra versión del problema de síntesis, tenemos los mismos elementos que en un problema de búsqueda tradicional: estado inicial; un conjunto de acciones, donde cada acción puede ser entendida como una función que recibe un estado y retorna otro; una función que retorna qué acciones son ejecutables en un estado dado; un objetivo. La novedad acá es que las acciones se dividen en dos conjuntos disjuntos: las acciones *controlables* y las acciones *incontrolables*. Nuestro objetivo es encontrar una secuencia $a_1 a_2 \dots a_n$ de acciones controlables, tal que cada secuencia $a_1\sigma_1 a_2\sigma_2 \dots a_n\sigma_n$, donde σ_i es alguna secuencia de 0 o 1 acción incontrolable ($i \in \{1, \dots, n\}$), llega al objetivo si es ejecutada sobre el estado inicial. En otras palabras, es posible interpretar el problema como uno en donde cada vez que nosotros ejecutamos una acción (controlable), el ambiente puede "responder" ejecutando cero o una acción (incontrolable). Debemos asegurar que llegamos al objetivo **independiente** de lo que el ambiente haga.

Específicamente, en el MBC interpretaremos a las acciones ejecutadas por *bor* como el ambiente (es decir, *bor* es nuestro adversario). En el programa base, están los predicados `controllable` y `uncontrollable`, que determinan cuándo las acciones son de un tipo u otro.

El objetivo final de esta parte es que usted programe el predicado `plies(Acciones, Situaciones)` que se satisface cuando `Acciones` es una secuencia de acciones controlables y `Situaciones` es una lista de situaciones a las que se puede llegar al agregar inmediatamente después de cada acción en `Acciones` a una acción incontrolable. El predicado `plies` debe funcionar cuando ambos parametros están sin instanciar.

```
?- plies(Actions,SitSet).
Actions = [],
SitSet = [s0] ;
```

[1] Tome encuenta que las respuestas que usted obtenga pueden diferir de lo que se muestra acá.

```
Actions = [paint(rob, a, blue)],
SitSet = [do(paint(rob, a, blue), s0), do(paint(bor, c, red), do(paint(rob, a, blue), s0)),
          do(paint(bor, d, red), do(paint(rob, a, blue), s0))] ;

Actions = [paint(rob, d, blue)],
SitSet = [do(paint(rob, d, blue), s0), do(paint(bor, c, red), do(paint(rob, d, blue), s0)),
          do(paint(bor, d, red), do(paint(rob, d, blue), s0))] ;

Actions = [move_to_block(a, d)],
SitSet = [do(move_to_block(a, d), s0), do(paint(bor, c, red), do(move_to_block(a, d), s0)),
          do(paint(bor, d, red), do(move_to_block(a, d), s0))] ;

Actions = [move_to_block(d, a)],
SitSet = [do(move_to_block(d, a), s0), do(paint(bor, c, red), do(move_to_block(d, a), s0))] ;

Actions = [move_to_table(a)],
SitSet = [do(move_to_table(a), s0), do(paint(bor, c, red), do(move_to_table(a), s0)),
          do(paint(bor, d, red), do(move_to_table(a), s0)),
          do(paint(bor, a, red), do(move_to_table(a), s0))]
```

Para implementar este predicado, conviene que usted implemente los predicados auxiliares que se describen en el archivo base. En particular, usted deberá implementar el predicado `holdsAll`, que será tal que `holdsAll(F,Set)` se satisface cuando `Set` es un conjunto de situaciones donde se cumple que `holds(F,S)` para cada `S` que está en `Set`. Con este predicado, se puede computar soluciones a problemas de síntesis, como el de pintar el bloque `c` de color azul:

```
?- plies(Actions,SitSet),holdsAll(color(c,blue),SitSet).
Actions = [move_to_block(a, d), move_to_block(b, a), move_to_block(c, b), paint(rob, c, blue)],
SitSet = (...omitido...)
```

5. Bonus

Se dará un bonus de hasta 1 punto si usted consigue hacer que la búsqueda cooperativa de la parte 1 funcione significativamente más rápida que con la estrategia del predicado `legal` y más rápida que con la heurística "goal counting", que se provee en el archivo de bonis. Para ello el profesor ha dejado disponible una implementación de A* es Prolog, que usted podría adaptar al problema de la parte 1.

6. Entrega

Las método de entrega se dará a conocer a tiempo.