PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
Facultad de Ingeniería
IIC2613 Inteligencia Artificial
2024-2

Tarea 2

Programación Lógica para el Razonamiento

Fecha de entrega: viernes 13 de septiembre a las 23:59 hrs

Aspectos generales

Formato y plazo de entrega

El formato de entrega son archivos con extensión `.lp` con un PDF para las respuestas teóricas. El lugar de entrega es en el repositorio de la tarea, en la branch por defecto, hasta el viernes 13 de septiembre a las 23:59 hrs. Para crear tu repositorio, debes entrar en el enlace del anuncio de la tarea en Canvas. Por último, recuerda que los cupones de atraso son días **no hábiles** extra.

Integridad Académica

Este curso se adhiere al Código de Honor establecido por la universidad, el cual tienes el deber de conocer como estudiante. Todo el trabajo hecho en esta tarea debe ser **totalmente individual**. La idea es que te des el tiempo de aprender estos conceptos fundamentales, tanto para el curso, como para tu formación profesional. Las dudas se deben hacer exclusivamente al cuerpo docente a través de las *issues* en GitHub.

Por otra parte, sabemos que estás utilizando material hecho por otras personas, por lo que es importante reconocerlo de la forma apropiada. Todo lo que obtengas de internet debes citarlo de forma correcta (ya sea en APA, ICONTEC o IEEE). Cualquier falta a la ética y/o a la integridad académica será sancionada con la reprobación del curso y los antecedentes serán entregados a la Dirección de Pregrado.

Comentarios adicionales

El objetivo de esta tarea es que puedan desarrollar la capacidad de modelar problemas a partir del lenguaje natural y resolverlos utilizando ASP en Clingo. Es fundamental que pongan énfasis en las justificaciones de sus respuestas, cuidando la redacción, ortografía; manteniendo el código ordenado y comentado. Aquellas respuestas que solo presenten resultados o código (sin contexto ni comentarios) no serán consideradas, mientras que tareas desordenadas pueden ser objeto de descuentos.

Se recomienda fuertemente apoyarse de los apuntes del curso para el desarrollo de la tarea, además de la bibliografía oficial del curso. En particular el uso de Clingo y modelación de problemas.

1. Reflexión y Teoría (1 pts.)

Mucha gente argumenta que el podcast de Lex Fridman en YouTube es imperdible para todo quien se dedique a la inteligencia artificial. Lex hace entrevistas a investigadores de diversos temas en el área, planteando preguntas muy interesantes, abriendo la discusión a temas más allá de algoritmos o modelos de inteligencia artificial. Para esta pregunta debes realizar lo siguiente

1. (0,5 puntos) Ve el extracto de la entrevista (click acá) que hizo Lex Friedman a François Chollet el 2020. Resume el argumento que tiene François en contra del Test de Turing y explica por qué Lex está en descuerdo con uno de sus puntos (subtítulos disponibles). ¿Crees que si los investigadores en IA se pusieran como objetivo pasar el Test de Turing no lograríamos importantes avances? Argumenta. Escribe tu respuesta en no más de media página[1].

2. (0,5 puntos) Ve ahora la entrevista del mismo Lex a Melanie Mitchel (click acá) específicamente la discusión respecto de los *conceptos*. Da un ejemplo de un concepto que se pueda representar en ASP. Especula sobre qué sistema computacional sería necesario implementar para permitir que un sistema de programación en lógica realice analogías; da un ejemplo concreto. Escribe tu respuesta en no más de media página.

Qué entregar en esta parte

En un subdirectorio `Parte1/` un archivo `respuesta.pdf` con tu respuesta.

[1] Escrita a 11pt, fuente Arial/Times/Computer Modern. Recomendamos (no obligamos!) usar LaTeX para el informe.

2. Teoría de ASP (1,5 puntos)

1. (0,5 puntos)

   a) Usa la definición del modelo de un programa (entregada en las diapos de clases) para demostrar que el programa $\Pi = \{p, \leftarrow p\}$ no tiene modelos. (*Pista:* supón que tiene un modelo y luego muestra que eso lleva a una contradicción.)

   b) Usa la definición de modelo de programa con negación para demostrar que el siguiente programa tiene dos modelos.

$$\Pi = \{p \leftarrow not\ q,$$
$$q \leftarrow not\ p\},$$

2. (0,5 puntos) Demuestra que la propiedad de monotonía se cumple para programas en lógica sin negación, y sin reglas con cabeza vacía. Específicamente, demuestra que si Π y Π' son dos programas tales que $\Pi \subseteq \Pi'$ y M es un modelo de Π, entonces existe un modelo M' de Π' tal que $M \subseteq M'$. Para esta demostración usa la definición de modelo de un programa sin negación y considera que las reglas $Head \leftarrow Body$ son tales que $|Head| = 1$. (*Ayuda:* esta demostración la puedes realizar por inducción en el número de reglas que tiene un programa. Debes demostrar que si se agrega una nueva regla al programa (cuya cabeza tiene exactamente un átomo) toda consecuencia del antiguo programa siguen siendo consecuencias del nuevo.)

3. (0,5 puntos) Demuestra, usando la definición de modelo de un programa sin negación, que todo programa que tiene reglas de la forma $Head \leftarrow Body$, sin negación, y con $|Head| \leq 1$ tiene a lo más un modelo. (*Ayuda:* puedes hacer la demostración usando inducción en el número de reglas de un programa).

Qué entregar en esta parte

En un subdirectorio `Parte2/` un archivo `respuesta.pdf` con tu respuesta.

3. DCConecta (1,5 puntos)

Pre Introducción

Un elemento que no hemos visto en detalle en clases es que clingo soporta operaciones matemáticas simples. Por ejemplo, podemos escribir el siguiente programa para calcular el área/perímetro de un rectángulo.

```
rectangulo(r1, 4, 7).
area(X, A) :- rectangulo(X, Alto, Ancho), A = Alto*Ancho.
perimetro(X, P) :- rectangulo(X, Alto, Ancho), P = 2*(Alto+Ancho).
perimetro(X, P) :- circulo(X, Radio), P = 6.28*Radio.
```

En el resto de esta pregunta usaremos operaciones aritméticas, pero también otros elementos del lenguaje que te pediremos que aprendas desde el libro. De hecho, antes de seguir leyendo el enunciado, anda al libro [2] y revisa el operador `#count`, ya que lo necesitaremos luego.

Introducción a DCConecta

Como parte de tu pasantía en DCConecta, una innovadora red social, te han asignado muchas tareas. Por suerte recuerdas que con la materia de Answer Set Programming con Clingo puedes hacer las tareas que te piden mucho más rápido. En DCConecta, las personas pueden establecer enlaces de amistad con otros usuarios. La plataforma quiere analizar rutas de conexión, gestionar la visibilidad de los enlaces, y optimizar las interacciones entre los usuarios.

Aquí hay algunas reglas básicas para definir un conjunto de amigos:

```
amistad(juan,maria).
amistad(maria,elena).
amistad(elena,fernando).

usuario(X) :- amistad(X,Y).
usuario(Y) :- amistad(X,Y).
```

Actividades

3.1. Rutas de Conexión con Conteo de Enlaces (0,5 puntos)

Tu primer desafío es definir un predicado `cadena/3` (predicado de nombre cadena con aridad 3) que determine si existe una cadena de amistades entre dos personas y, además, que cuente cuántas amistades hay en esta cadena. Aquí tienes un ejemplo:

```
cadena(X,Y,Largo) :- amistad(X,Y), Largo=1. %Caso base
cadena(X,Y,Largo) :- cadena(X,Z,L1), cadena(Z,Y,L2), Largo=L1+L2.
```

[2] Libro de Valdimir Lifschitz sobre Answer Set Programming disponible en https://www.cs.utexas.edu/~vl/teaching/378/ASP.pdf

Con estos predicados la red funciona, pero si se introducen ciclos en la red (por ejemplo, en el programa de ejemplo conectar "fernando" con "juan" generaría un ciclo) se producen conflictos. Ve que pasa si haces lo anterior y propón una solución para evitar que el predicado `cadena/3` se quede atrapado en ciclos utilizando el operador `#count`. Esto no significa que no debas considerar las amistades en un ciclo pero te debes asegurar de tratar el problema que surge al tener ciclos en el grafo con únicamente los predicados anteriores. Asegúrate de que la solución cuente correctamente todos los caminos posibles sin ciclos que hay en la red, sin omitir ninguna cadena válida.

3.2. Visibilidad de Amistades en la Plataforma (0,5 puntos)

En DCConecta, algunas conexiones entre usuarios pueden ser visibles mientras que otras permanecen ocultas. Un usuario puede ser visible o no visible y este estado se representa con el predicado `visible/1`. Un enlace o amistad visible es aquel que otros usuarios pueden ver. Una amistad se considerará visible cuando al menos una de las personas en la amistad es visible. Además considera que de ahora en adelante los grafos con los que se trabajan son no dirigidos (la amistad es bidireccional) agregando el predicado.

```
amistad(X,Y) :- amistad(Y,X).
```

Considerando lo anterior define los siguientes dos predicados:

- Predicado `amistad_visible/2`: Define este predicado para indicar si un enlace entre dos usuarios es visible en la red.
- Predicado `conectados_por_n/3`: Define un predicado que determine si existen dos usuarios conectados a través de una cadena con exactamente n amistades no visibles. Asegúrate de que todas las cadenas posibles sean considerados en la red, sin perder ninguna.

3.3. Ruta con Menor Número de Amistades Ocultas (0,5 puntos)

El equipo de desarrollo quiere encontrar la cadena entre dos usuarios que minimice el número de amistades ocultas. Escribe el predicado `cadena_minima_n/3` para expresar que la cadena entre dos personas cualquiera con el menor número de amistades ocultas tiene exactamente n amistades no visibles.

Puede ser útil recordar que en Clingo puedes usar operaciones de comparación como por ejemplo:

```
es_menor_que(X,Y) :- persona(X, Edad1), persona(Y, Edad2), Edad1<Edad1.
```

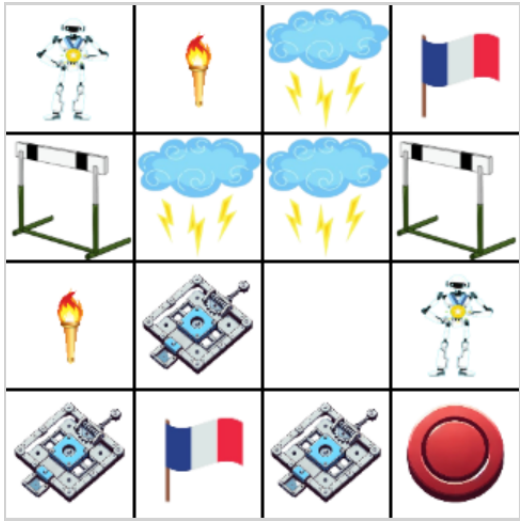
Qué entregar en esta parte

En un subdirectorio `Parte3/` de tu repositorio de entrega, para las partes 2 y 3 crea un archivo `<p>.lp` con la solución de cada pregunta `<p>`. Incluye comentarios explicando lo que hacen tus predicados; estos serán considerados en la corrección. El código comentado no se considera parte de la solución. La solución para la parte 1 debe ser descrita en un archivo `Parte3/respuestas.pdf`, que debe quedar en el mismo directorio.

4. DCCarrera Olímpica (2 puntos)

Introducción:

El DCC comienza el semestre con la terrible noticia de que el 70 % de su staff y estudiantes se encuentran en estado sedentario por pasar mucho tiempo viendo Netflix. Inspirado por los atletas de París 2024, Jorge decide establecer un programa de entrenamiento olímpico para acabar de una vez por todas con el sedentarismo y mejorar el bienestar del departamento.
Sin embargo, los reacios sedentarios deciden diseñar robots para que compitan en lugar de ellos y han pedido de tu conocimiento en Clingo para que ayudes a que sus robots triunfen en la DCCarrera Olímpica. El código ya desarrollado por programadores contiene robots, obstáculos, vallas, trampas, condiciones climáticas adversas y antorchas que deben ser llevadas a su objetivo.
A continuación un mapa ejemplo del problema:



En el código base que te entregaremos, esta implementado lo siguiente:

- Un robot puede desplazarse hacia arriba, pero no hacia ninguna otra dirección.
- Un robot no puede moverse hacia cierta dirección si es que no está mirando a ella.
- Se define cuando una valla está adyacente al robot por la derecha.

- Un robot puede esperar en una posición por un tiempo.
- Un robot es capaz de levantar una antorcha, únicamente si está adyacente a él. Además, la antorcha pasa a estar en la posición del robot y este último mantiene su localización.
- Un robot ejecuta una acción en cada tiempo.
- Una antorcha puede ser llevada únicamente por un robot.
- Un robot no puede llevar más de una antorcha al mismo tiempo.
- Si hay una antorcha en el suelo, un robot no puede pasar por encima de ella.
- No puede haber más de un robot en la misma casilla.
- Un robot no puede pasar por un obstáculo.
- Luego de un tiempo *bound*, cada antorcha deberá estar en alguna de las banderas objetivo del mapa.
- Si a una trampa no se ha desactivado, seguirá estando activada en el tiempo.
- Un botón está activado si previamente ha sido presionado.
- Se revisa si un `olimpicRobot` está en una casilla objetivo en el tiempo T, para casos en el que no hay testimonio.

Los predicados más importantes que modelan este programa son los siguientes:

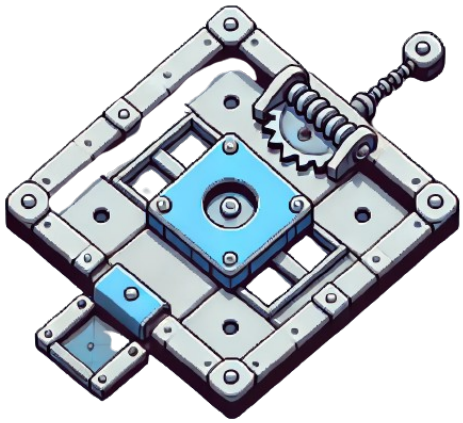
```
rangeX(0..X).       % Determina la dimensión del mapa en el eje X
rangeY(0..Y).       % Determina la dimensión del mapa en el eje Y
olimpicRobot(R).    % Determina la existencia de un OlimpicRobot
obstacle(X,Y).      % Indica la existencia de un obstáculo en X,Y (mal tiempo)
trap(X,Y,S,T).      % Indica la existencia de una trampa en X,Y en estado S (activa 1 o
↪  apagada 0) en el tiempo T
button(X,Y).        % Indica la existencia de un botón en X,Y
goal(X,Y).          % Declara una posición meta en X,Y
hurdle(H).          % Determina la existencia de una valla H.
hurdleOn(H,X,Y)     % Determina la ubicación (X,Y) de la valla H.
baton(B).           % Determina la existencia de un testigo
batonOn(B, X, Y, Z, T) % Expresa que un baton B está en X,Y,Z (1 si lo tiene un robot, 0
↪  si no) en el tiempo T.
time(1..bound).      % Define los tiempos T, desde 1 a bound
olimpicRobotOn(R,X,Y,T,D,Z).     % Expresa que un OlimpicRobot está en X,Y,Z (0 si
↪  está caminando, 1 si está saltando) en el tiempo T, apuntando en la dirección D.
action(A).          % Declara la acción A, con A = acción disponible
exec(R,A,T).        % Indica que el OlimpicRobot ejecuta la acción A en el tiempo T.
```

Carpetas y archivos

Carpetas

- `tests`: Contiene circuitos olímpicos de ejemplo a resolver.
- `imgs`: Contiene los sprites de la visualización.

(a) Imagen de una valla



(b) Imagen de una trampa

(c) Imagen de robot en movimiento

(d) Imagen del testimonio

Figura 1: Componentes a implementar

Archivos

Los archivos que tendrás para resolver el problema son:

- `tests/testX.lp`: Archivo con los tests de pruebas olímpicas a resolver, no modificar.
- `DCCarrera.lp`: Archivo que contiene las reglas que solucionan los mapas. Acá debes completar las reglas para que resuelvan el problema.
- `process.py`: Archivo que contiene el código que se encarga de leer la solución entregada por clingo en la consola, parsearla y generar un archivo `output.txt`. No debes modificarlo.
- `DCCarrera_Olimpica.html`: Archivo que contiene el visualizador del mapa junto con la solución entregada. Al abrirlo en un navegador se debe ingresar al botón "Seleccionar archivo" y cargar la solución entregada en `output.txt`. No lo debes modificar.

- `output_example.txt`: Archivo que contiene un output de ejemplo para que puedas probar el visualizador (`DCCarrera_Olimpica.html`).

Actividades

4.1. Actividad 1: Completar código para que el robot se mueva y pueda cambiar su orientación (0,5 puntos)

La primera tarea para que te vayas familiarizando con el programa, consiste en completar el código para que un robot se mueva en todas las direcciones y este sea capaz de modificar su mirada en 90 grados.

1. Completar el código para que el robot pueda moverse para arriba, abajo, hacia la derecha o hacia la izquierda. Se te proporciona el siguiente código:

```
olimpicRobotOn(R, X, Y - 1, T + 1, D, Z) :- exec(R, up, T), olimpicRobotOn(R, X, Y,
↪  T, D, Z), time(T + 1).
```

   Nota: Es necesario utilizar las acciones ya definidas como:

```
action(up).
action(down).
action(left).
action(right).
```

   Tip: No es necesario preocuparse de cómo es que las acciones se activan, por el momento, solo es necesario modelar qué es lo qué pasa cuando estas se activan.

2. Realizar el código para que el robot sea capaz de modificar su mirada **en 90 grados**. Para esto, es necesario saber que el argumento `D` de `olimpicRobotOn(R,X,Y,T,D,Z)` toma el valor de:

   - 1, si está mirando hacia arriba
   - 2, si está mirando hacia abajo
   - 3, si está mirando hacia la izquierda
   - 4, si está mirando hacia la derecha

   Nota: Es necesario utilizar las acciones ya definidas como:

```
action(lookUp).
action(lookDown).
action(lookLeft).
action(lookRight).
```

   Para comprobar la Actividad 1 se encuentra el `test01.lp`.

4.2. Actividad 2: Completar código para que el robot pueda saltar una valla (0,5 puntos)

A continuación, deberás completar el código para que el robot sea capaz de saltar una valla.
Para saltar, el robot debe estar mirando hacia donde salta y debe ser adyacente por tal lado a la valla

(revisar definiciones de `hurdleAdyacentToRobotToGoToRight`). Además, un robot no puede estar en la misma posición que una valla si es que su `Z` es 0 (ya definido en el código) ni hacer salto en "L", es decir, debe ejecutar la acción "`jumpHurdle`" desde una celda anterior a la valla en un tiempo `T`, pasar por esta en un tiempo `T+1` (en el cual se debe ejecutar "`getDown`") y finalizar en el espacio correspondiente en un tiempo `T+2`.

1. Completar el código para `hurdleAdjacentToRobotToGo{Direction}`. Se les otorga el código:

```
hurdleAdjacentToRobotToGoRight(H, R, T) :- olimpicRobotOn(R, X - 1, Y, T, 4, Z),
↪  hurdleOn(H, X, Y), olimpicRobot(R), time(T), hurdle(H).

hurdleAdjacentToRobotToGoRight(H, R, 0) :- olimpicRobotOn(R, X - 1, Y, 0, 4, Z),
↪  hurdleOn(H, X, Y), olimpicRobot(R), hurdle(H).
```

   Y deben completar el código para `hurdleAdjacentToRobotToGoUp`, `hurdleAdjacentToRobotToGoDown` y `hurdleAdjacentToRobotToGoLeft`.

2. Realizar el código para que al ejecutarse la acción "`jumpHurdle`", entonces el robot avance hacia la posición de la valla adyacente. Hint: es necesario ver que ocurre en el tiempo 0 que no está definido por `time(T)`.

   Nota: Es necesario utilizar la acción:

```
action(jumpHurdle).
```

3. Realizar el código para que al ejecutarse la acción "`getDown`", entonces el robot avance hacia el lugar correspondiente al haber pasado la valla y vuelva a `Z = 0`.

   Nota: Es necesario utilizar la acción:

```
action(get).
```

Para probar la actividad 2, se utilizan los `tests: testV1.lp, testV2.lp, testV3.lp, testV4.lp` y `testV5.lp`.

4.3. Actividad 3: Completar código para que el robot pueda tomar el testimonio (0,5 puntos)

A continuación, deberás completar el código para que el robot sea capaz de levantar el testimonio.
Para levantarlo, el robot debe estar mirando hacia donde esta ubicado el testimonio y debe ser adyacente por tal lado (revisar definiciones de `batonAdjacentToOlimpicRobot`).
Para esto, se te entregan las siguientes funcionalidades ya implementadas:

1. Predicado para calcular la cantidad de testimonios adyacentes a un robot.

```
batonsAdjacentToOlimpicRobot(R, T, N) :- olimpicRobot(R), N = #count{B :
↪  batonAdjacentToOlimpicRobot(B, R, T)}, time(T).

batonsAdjacentToOlimpicRobot(R, 0, N) :- olimpicRobot(R), N = #count{B :
↪  batonAdjacentToOlimpicRobot(B, R, 0)}.
```

2. Se puede levantar el testimonio.

```
1{olimpicRobotLiftBaton(R, B, T): batonAdjacentToOlimpicRobot(B, R, T)}1 :- exec(R,
↪  liftBaton, T), time(T + 1), olimpicRobot(R).
```

3. Si hay un testimonio encima de un robot, hay una testimonio recogido. Si no está recogido el testimonio está en el piso.

```
batonPickedUp(B, T) :- batonOnOlimpicRobot(B, R, T).

batonOn(B, X, Y, 0, T + 1) :- batonOn(B, X, Y, Z, T), Z = 0, not batonPickedUp(B, T +
↪  1), time(T + 1), baton(B).
```

4. No puede haber más de un robot en la misma casilla y más de un testimonio en la misma casilla.

```
:- olimpicRobotOn(R1, X, Y, T, D1, Z1), olimpicRobotOn(R2, X, Y, T, D2, Z2), R1 !=
↪  R2.

:- batonOn(B1, X, Y, Z, T), batonOn(B2, X, Y, Z, T), B1 != B2, Z = 0.
```

5. Si el robot tiene el testimonio y se mueve, el testimonio también.

```
batonOn(B, X, Y, 1, T) :- olimpicRobotOn(R, X, Y, T, D, Z), batonOnOlimpicRobot(B,
↪  R, T), time(T).

batonOnOlimpicRobot(B, R, T + 1) :- batonOnOlimpicRobot(B, R, T), time(T + 1).
```

6. Predicado para revisar que un testimonio esta en una casilla objetivo en el tiempo T.

```
at_goal(B, T) :- batonOn(B, X, Y, Z, T), goal(X, Y), olimpicRobotOn(R, X, Y, T, D,
↪  0).

:- baton(B), not at_goal(B, bound).
```

7. Predicado que indica que el testimonio pasa a levantado por un `olimpicRobot`, además pasa a estar en la posición del `olimpicRobot`, y este último mantiene su localización. Se indica que el `olimpicRobot` lleva el testimonio. Esto se hace para todas las direcciones posibles.

```
3{batonOn(B, X - 1, Y, 1, T + 1); olimpicRobotOn(R, X - 1, Y, T + 1, D, Z);
↪  batonOnOlimpicRobot(B, R, T + 1)}3 :- olimpicRobotLiftBaton(R, B, T),
↪  olimpicRobotOn(R, X - 1, Y, T, D, Z), time(T + 1), batonOn(B, X, Y, Z, T),
↪  olimpicRobot(R), Z = 0.

3{batonOn(B, X + 1, Y, 1, T + 1); olimpicRobotOn(R, X + 1, Y, T + 1, D, Z);
↪  batonOnOlimpicRobot(B, R, T + 1)}3 :- olimpicRobotLiftBaton(R, B, T),
↪  olimpicRobotOn(R, X + 1, Y, T, D, Z), time(T + 1), batonOn(B, X, Y, Z, T),
↪  olimpicRobot(R), Z = 0.
```

```
3{batonOn(B, X, Y - 1, 1, T + 1); olimpicRobotOn(R, X, Y - 1, T + 1, D, Z);
↪  batonOnOlimpicRobot(B, R, T + 1)}3 :- olimpicRobotLiftBaton(R, B, T),
↪  olimpicRobotOn(R, X, Y - 1, T, D, Z), time(T + 1), batonOn(B, X, Y, Z, T),
↪  olimpicRobot(R), Z = 0.

3{batonOn(B, X, Y + 1, 1, T + 1); olimpicRobotOn(R, X, Y + 1, T + 1, D, Z);
↪  batonOnOlimpicRobot(B, R, T + 1)}3 :- olimpicRobotLiftBaton(R, B, T),
↪  olimpicRobotOn(R, X, Y + 1, T, D, Z), time(T + 1), batonOn(B, X, Y, Z, T),
↪  olimpicRobot(R), Z = 0.
```

Ahora, con los predicados presentes, tu tarea será implementar las siguientes reglas:

1. Un robot es capaz de levantar el testimonio únicamente si está adyacente a él. Se define qué es que esté adyacente.
2. No se puede levantar el testimonio si el robot no está adyacente.
3. Un robot no puede llevar más de un testimonio al mismo tiempo.
4. Un testimonio puede ser llevado por un único robot.
5. Un robot solo puede recoger un testimonio que esté en la dirección que está mirando.

Para probar la actividad 3, se utilizan los `tests: testB1.lp, testB2.lp, testB3.lp, testB4.lp` y `testB5.lp`.

4.4. Actividad 4: Completar código para que el robot pueda desactivar las trampas (0,5 puntos)

En esta actividad, deberás completar el código necesario para que el robot pueda desactivar las trampas cuando presione el botón rojo. En concreto debes completar las siguientes dos reglas:

1. Completar la regla de que una trampa se desactiva cuando se da la orden de desactivación. Se te proporcionará el siguiente código base para que te familiarices con la lógica:

```
% Regla que define la orden de desactivación de una trampa.
disableTrap(T) :- buttonActivated(X, Y, T), time(T).

% Si una trampa no se le ha dado la orden de desactivación, seguirá estando activada
↪  en el tiempo.
trap(X, Y, S, T2) :- trap(X, Y, S, T1), time(T2), T2 = T1 + 1, not disableTrap(T2).
```

2. Completar la regla de que un botón se activa cuando un `olimpicRobot` está encima del botón y además NO ha sido presionado previamente.

   Se te entregará el siguiente código base que define si un botón ha sido presionado anteriormente.

```
% Se define si el botón ha sido activado previamente
activatedEarlier(X, Y, T) :- buttonActivated(X, Y, T1), T1 < T, time(T).
```

Para probar la actividad 4, se utilizan los `tests: testR1.lp, testR2.lp, testR3.lp, testR4.lp` y `testR5.lp`. Por último, si quieres probar un test con vallas, testimonios, botones y trampas, utiliza el `testfull.lp`.

Comandos a utilizar

- Para comenzar a resolver el problema puede ser útil querer manipular solamente el archivo de Clingo y ver la salida en la consola. Para esto se puede utilizar el siguiente comando:

  `clingo tests/testX.lp DCCarrera.lp`

  Con esto, se podrá ver la salida de Clingo en la consola y verificar si las reglas están funcionando correctamente. En esta parte, puedes modificar los statements `#show` para que te sea más fácil depurar el código.

- Una vez que tengas una solución que te parezca correcta, puedes ejecutar el siguiente comando para generar el archivo `output.txt`:

  `clingo FILENAME.lp tests/TESTNAME.lp -c bound=N | python3 process.py`

  Este comando ejecutará clingo y luego procesará la salida para generar el archivo `output.txt`. Es muy importante que, en esta parte, mantengas los *statements* `#show` que se entregaron inicialmente en el archivo `DCCarrera.lp` para que el proceso de parseo funcione correctamente. Si el comando `python` no funciona, puedes intentar con `python3` o `py`.

- A continuacion se incluyen los *statements* `#show` inicialmente entregados en el archivo `.lp`:

```
% Statements visualizer
#show time/1.
#show olimpicRobotOn/6.
#show batonOn/5.
#show obstacle/2.
#show hurdleOn/3.
#show rangeX/1.
#show rangeY/1.
#show goal/2.
#show trap/4.
#show exec/3.
#show button/3.
```

- Finalmente, puedes abrir el archivo `DCCarrera_Olimpica.html` en un navegador y cargar el archivo `output.txt` que se generó en el paso anterior. De esta forma, podrás visualizar el mapa y la solución entregada por clingo.

Qué entregar en esta parte

En un subdirectorio `Parte4/`:

- Deberás entregar el archivo `DCCarrera.lp` con las reglas implementadas para resolver el problema.
- El archivo `DCCarrera.lp` debe estar correctamente comentado para que se entienda la lógica de las reglas implementadas.