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

Tarea 1
IIC2613 - Inteligencia Artificial
Primer Semestre, 2021
Entrega: 26 de abril de 2021

Objetivo y Regla Importante

Esta tarea tiene dos objetivos. El primero es que reflexiones sobre los requisitos que deben tener nuestros sistemas inteligentes. El segundo objetivo es que te familiarices aún más con las aplicaciones y definiciones de la herramienta de la programación en lógica. Esta tarea requiere que entregues código y un informe.

Estoy consciente que algunas respuestas a las preguntas de esta tarea podrías encontrarlas en Internet, en libros, o conversarlas con tus compañeros. Como esta tarea tiene un fin pedagógico, la regla de integridad académica que pido que cumplas es la siguiente:

> *Toda respuesta a una pregunta teórica debe ser escrita por ti, individualmente. Esto significa que al momento de editar la respuesta, no puedes estar mirando o usando un texto escrito por otra persona, un video hecho por otra persona, o cualquier material que haya sido desarrollado por otra persona. En otras palabras, lo que sea que escribas* ***debe provenir directa y únicamente de ti****. Sin embargo,* ***antes*** *de comenzar a escribir tu respuesta puedes estudiar y comentar la pregunta con una compañera o compañero, leer libros, mirar videos, o realizar cualquier otra acción que te ayude a responder la pregunta. Exactamente la misma regla se cumple respecto del código que entregues.*

Tu entrega deberá incluir un archivo llamado `INTEGRIDAD.txt`, en donde afirmas que has leído, entendido y respetado la regla de arriba. De no cumplirse esto, **tu tarea no será corregida y en consecuencia obtendrás la nota mínima, tal como si no hubieses entregado**. Recuerda subir todo el material relacionado a tu tarea en tu repositorio personal de GitHub a más tardar la fecha de la entrega a las 23:59, para esto deben crear el directorio `T1/` que contenga todos los archivos y subdirectorios especificados.

Parte 1: ¿Qué necesitamos para lograr inteligencia? (1 punto)

1. Ve la entrevista (click acá) que hizo Lex Friedman a François Chollet a mediados del año pasado. 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. 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 proogramación 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. Antes de hacer commit, asegúrate que leíste la página 1 de esta tarea.

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

Parte 2: Programación Básica en *Answer Set Programming* (ASP) (1,5 puntos)

Un elemento que en clases no vimos (aunque sí en ayudantías), fue 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 pediré aprendas desde el libro. De hecho, antes de seguir leyendo, anda al libro y revisa el operador `#count`, ya que lo necesitaremos luego.

Para el resto de este problema, supondremos que podemos definir grafos usando un predicado `arco/2` y un predicado `nodo/1`. Por ejemplo:

```
arco(a,b).
arco(b,c).
arco(b,e).
arco(e,f).
nodo(X) :- arco(X,Y).   % los nodos son aquellos mencionados en los arcos
nodo(Y) :- arco(X,Y).
```

1. La fase de instanciación siempre es necesaria, incluso para programas que contienen operaciones aritméticas. Esto implica que, cuando usamos aritmética, es necesario tomar ciertas precauciones. Por ejemplo, imagina que quisiéramos definir el predicado `camino` tal como lo hicimos en ayudantía, pero extendido para que además calcule el *largo* de un camino, de esta forma:

   ```
   camino(X,Y,Largo) :- arco(X,Y), Largo=0.
   camino(X,Y,Largo) :- camino(X,Y,L1), camino(X,Y,L2), Largo=L1+L2.
   ```

   Prueba cómo funciona este predicado junto con el grafo que fue definido arriba. Ahora ve qué ocurre cuando agregas un ciclo al grafo. Explica brevemente, pero de la forma más clara posible, que es lo que está pasando. (Si necesitas una fuente de iluminación explícita, anda a la pregunta 4 de esta parte y es posible que encuentres algo que te sirva.)

   Ahora propón una manera de evitar el problema que se genera en un grafo con ciclos usando el operador `#count` de tal manera que, con seguridad e independiente del grafo definido por el usuario, el predicado `camino/3` sea capaz de contar el largo de todo camino sin ciclos existente en el grafo. (Esto no significa que no debe considerar los caminos con ciclos. Lo que debes asegurar es que no se le 'escape' ningún camino sin ciclos existente en el grafo.)

2. En esta pregunta y la siguiente trabajaremos con grafos no dirigidos, que se pueden modelar agregando la siguiente regla:

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

   Describiremos la situación de un pequeño pueblo de una localidad rural de bajos recursos que hasta el momento no tiene iluminación en las calles durante la noche. El alcalde desea proveer cierto tipo de iluminación en las calles, y quiere aprovechar que si instala una luminaria en una esquina, entonces todas las calles que llegan a esa esquina se encuentran iluminadas también. Para representar esta situación usaremos un grafo no dirigido en donde las equinas son nodos y las calles, arcos.

Suponiendo que un usuario define el predicado `iluminado/1` que expresa que un nodo del grafo está iluminado, escribe el predicado `iluminada/2` que es tal que `iluminada(X,Y)` aparece en un modelo únicamente cuando (`X`,`Y`) es un arco que representa a una calle que está iluminada.

Además, escribe el predicado `conectado_k/3` que es tal que `conectado_k(X,Y,K)` aparece en el modelo si y solo si existe un camino entre `X` e `Y` con exactamente `K` arcos **no** iluminados. Tal como en la pregunta anterior, asegura que `conectado_k` no 'pierda' caminos en el grafo.

3. Las comparaciones entre números también están disponibles en clingo. Un ejemplo es el siguiente predicado:

```
mayor_superficie(X,Y) :- area(X, A1), area(Y, A2), A1>A2.
```

   Escribe el predicado `camino_minimo(X,Y,K)` para expresar que el camino entre `X` e `Y` que tiene menos calles sin iluminar tiene exactamente `K` calles sin iluminar.

4. Dado un grafo de n nodos y m arcos, escribe una expresión matemática que sea una buena aproximación para el tamaño del programa instanciado[2] que resulta para un programa que contiene los predicados de las partes 2 y 3. Justifica que tu resultado es correcto para cualquier grafo. Usa los supuestos más razonables posibles si necesitas hacerlos. Si quieres explorar de qué tamaño es la instanciación de un programa específico, puedes usar el comando `gringo` desde la línea de comando, de la siguiente manera:

```
$ gringo --text programa.lp
```

5. En esta parte resolverás el problema del alcalde, quien quiere ubicar luminarias en su ciudad, de manera que no exista ninguna ruta entre cada par de esquinas que estén conectadas por un camino con más de una calle sin iluminar.

   Tu programa debe ser disyuntivo, permitiendo explorar las distintas formas de iluminar las esquinas. Para encontrar el óptimo, usa la directiva de minimización `#minimize`, también explicada en el libro.

Qué entregar en esta parte

En un subdirectorio `Parte2/` de tu repositorio de entrega, para las partes 2, 3 y 5, crea un archivo `<p>.lp` con la solución de cada pregunta `<p>`. Incluye comentarios explicando lo que hacen tus predicados; éstos serán considerados en la corrección. El código comentado no se considera parte de la solución. La solución para las partes 2 y 4 deben ser descritas en un archivo `Parte2/respuestas.pdf`, que debe quedar en el mismo directorio. Antes de hacer commit, asegúrate que leíste la página 1 de esta tarea.

[2]Si conoces la notación $\Theta(\cdot)$, por favor úsala para mayor claridad.

Parte 3: Teoría de ASP (1,5 puntos)

1. Demuestra que la propiedad de monotonía se cumple para programas en lógica sin negación. 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.

2. Demuestra, usando la definición de modelo de un programa sin negación, que todo programa que tiene reglas de la $Head \leftarrow Body$, 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).

3. Para programas sin negación, en clases dimos dos formas de definir modelos: una matemática y una algorítmica. El algoritmo funciona con reglas del estilo $Head \leftarrow Body$, donde $Head$ es un conjunto que tiene un solo átomo, se veía más o menos así:

   (1) $M := \emptyset$

   (2) Selecciona una regla de Π de la forma $Head \leftarrow Body$ tal que $Body \subseteq M$ y tal que $Head$ no esté contenido en M.

   (3) Si el paso 2 encuentra una regla, hacer $M := M \cup Head$ y volver al paso 2. En caso contrario, retornar M.

   Supón ahora que permitimos incorporar algunas reglas con negación y que estas reglas podrían generar múltiples modelos. No permitiremos disyunción en los programas, es decir, puedes suponer que toda regla de la forma $Head \leftarrow Body$ es tal que $|Head| = 1$.

   Entrega un algoritmo recursivo, que sea en el peor caso exponencial en el número de átomos que aparecen negados en tu programa, y que usa al algoritmo descrito arriba como subrutina para calcular correctamente todos los modelos. Piensa en la versión más eficiente posible. (Ayuda: La eficiencia de tu algoritmo dependerá si tomas o no en cuenta la idea de que los modelos son minimales mientras calculas el conjunto respuesta.)

   Argumenta por qué crees que el algoritmo es correcto, utilizando ejemplos sencillos, pero que ilustren casos relevantes.

Qué entregar en esta parte

En un subdirectorio `Parte3/` un archivo `respuesta.pdf` con tu respuesta. Ten en cuenta que cuando te pido un algoritmo, estoy pidiendo un pseudocódigo y no una implementación detallada. El pseudocódigo debe ser claro. Antes de hacer commit, asegúrate que leíste la página 1 de esta tarea.

Parte 4: El mundo del Wumpus (2 puntos)

El juego

El mundo del Wumpus es un juego con *observabilidad parcial*, en donde un agente se desplaza por una grilla, que inicialmente se encuentra cubierta. Cada vez que el agente ingresa a una celda cubierta, esta queda visible, para siempre para el agente. Las reglas son las siguientes:

I. En la grilla hay un número indeterminado positivo de pozos (*pits*[3]) y un número positivo de *wumpus*.

II. En una misma celda no puede haber tanto un *wumpus* como un *pit*.

III. Todas las celdas de la grilla—cuyo tamaño es conocido para el agente—están inicialmente cubiertas, excepto por la celda inicial del agente.

IV. En cada celda vecina a un pozo el agente percibe una brisa (*breeze*). Una celda (x, y) es vecina de otra (x', y') ssi $|x - x'| + |y - y'| = 1$.

V. En cada celda vecina a un wumpus el agente percibe un hedor (*stench*).

VI. Existe una celda en el mapa que contiene oro (*gold*).

VII. Las acciones disponibles para el agente son: moverse a una celda (x, y)—en cuyo caso dicha celda debe ser vecina de la celda actual—o disparar (*shoot*) a la celda (x, y). El número máximo de disparos es igual al número de wumpus en la grilla.

VIII. El agente debe satisfacer la *regla de seguridad*, que establece que toda celda la que éste se mueve, demostrablemente, no contiene ni wumpus ni pit.

IX. Si el agente ejecuta *shoot* sobre (x, y) y en dicha celda había un *wumpus*, entonces éste desaparece.

X. El agente debe satisfacer la *regla de benevolencia*, que establece que solo se puede disparar contra un *wumpus* si esa es la única manera de avanzar en el juego.

XI. Una vez que un agente entra a una celda ésta queda observable por el resto del juego. Si, por ejemplo, ya no se percibe *breeze* en una celda, el agente lo sabrá.

XII. Si el agente visita una celda que contiene un *wumpus* o un *pit*, el agente pierde y el juego termina.

XIII. Si el agente visita una celda que contiene *gold*, el agente gana y el juego termina.

Para jugar una versión ligeramente distinta de este juego, puedes hacer click acá.

Semántica de mundos posibles

Para modelar mundos parcialmente conocidos usando programación en lógica, adoptamos lo que se conoce como *semántica de mundos posibles*. En la práctica, describimos el conocimiento de un agente con un programa que, en general, posee múltiples modelos. Cada modelo representa un *mundo posible*. Diremos que el agente *sabe* que un átomo `p` es verdad, cuando `p` aparece en todos los modelos del programa. En cambio, cuando `p` está en algunos modelos y en otros no, entonces diremos que el agente *no sabe si* `p` es verdadero ni falso. Finalmente, cuando `p` no aparece en ningún modelo, diremos que el agente sabe que `p` es falso.

[3] A lo largo de este enunciado, insisto en usar palabras en inglés para ser consistente con la implementación, que está en inglés. Esto es porque, en el mundo globalizado de hoy, creo que ya no es buena práctica usar otro idioma en tu código, especialmente si es una aplicación de IA.

Lo que debes hacer

1. **(1/3 del puntaje)** Escribe un programa que, bajo el esquema de mundos posibles, sea capaz de modelar el conocimiento del agente en el mundo del wumpus.

   Supondremos que para cada celda (x, y) que es visible para el agente, el programa contendrá una regla de la forma

   ```
   alive(x,y).
   ```

   y, además, podría contener o no cada una de las siguientes reglas

   ```
   sense_breeze(x,y). % para expresar que se siente una brisa en (x,y)
   sense_stench(x,y). % para expresar que se siente un hedor en (x,y)
   ```

   Además supondremos que para cada celda (x, y) de la grilla, todo modelo contiene un átomo de la forma `cell(x,y)`. Observa que para definir una grilla de 4×4 basta con una sola línea con la sentencia "`cell(0..3,0..3).`".

   En tu programa, usa el átomo `pit(X,Y)` para expresar que existe un pozo en (`X`, `Y`) y `wumpus(X,Y)` para expresar que existe un wumpus en (`X`, `Y`). En tu programa podría ser útil la siguiente definición:

   ```
   neigh(X,Y,Xp,Yp) :- cell(X,Y), cell(Xp,Yp), |Xp-X|+|Yp-Y|=1.
   ```

   Nombra al archivo resultante `wumpus.lp`. Es importante que este archivo no contenga la definición de una grilla específica, ni de un cierto input sensorial específico. Para hacer pruebas, te recomiendo definas la grilla y el input sensorial en otro archivo.

2. **(2/3 del puntaje)** Terminar de implementar la versión del juego, completando la clase dentro de `agent.py`. Específicamente debes completar el método `get_action` de la clase `Agent`. Recuerda que por la regla de seguridad, tu método debe garantizar que toda movida del agente es segura, es decir que conduce a una celda que el agente sabe que no contiene a un wumpus ni un pit. **En otras palabras, tu agente no "se la juega" si no puede actuar seguro**. En vez, en caso que no se pueda tomar una jugada segura, retorna una acción que indica que ha demostrado que el problema no tiene solución. (Ve en el código cómo se hace esto).

   Algunas consideraciones importantes:

   - Para partir haciendo esta parte, mira el archivo `wumpus.py`. En la línea 9 se construye un objeto de tipo `Environment`. El segundo argumento del constructor es un booleano que indica si es que el número de wumpus y número de pozos es observable o no para el agente. Es **muy importante** que tu clase `Agent` sea capaz de distinguir entre estos dos modos, porque esto determina si ciertos problemas tienen solución (o no).

     Cuando un ambiente es observable, podrás obtener el número de pits y wumpus, respectivamente, con los métodos `get_num_pits()` y `get_num_wumpus()`. Para revisar si un cierto ambiente es observable, puedes usar el método `is_observable()`. Los detalles de estos métodos están en la clase `Environment`.
   - No olvides el principio de benevolencia, según el cual no puedes disparar a un wumpus a menos que no tengas ninguna acción segura disponible que te haga avanzar en el juego.

   El siguiente es un ejemplo de ejecución con los parámetros usados en el archivo de prueba (mapa `map.txt`).

```
$ python3 wumpus.py
['.', 'W', '.', 'A']
['.', 'G', '.', '.']
['.', '@', '.', '@']
['@', '.', '.', '.']
Perceptions: []
Agent juega: ('goto', 0, 2)
['.', 'W', 'A', '.']
['.', 'G', '.', '.']
['.', '@', '.', '@']
['@', '.', '.', '.']
Perceptions: ['sense_stench(0,2)']
Agent juega: ('goto', 0, 3)
['.', 'W', '.', 'A']
['.', 'G', '.', '.']
['.', '@', '.', '@']
['@', '.', '.', '.']
Perceptions: ['sense_stench(0,2)']
Agent juega: ('goto', 1, 3)
['.', 'W', '.', '.']
['.', 'G', '.', 'A']
['.', '@', '.', '@']
['@', '.', '.', '.']
Perceptions: ['sense_breeze(1,3)', 'sense_stench(0,2)']
Agent juega: ('goto', 1, 2)
['.', 'W', '.', '.']
['.', 'G', 'A', '.']
['.', '@', '.', '@']
['@', '.', '.', '.']
Perceptions: ['sense_breeze(1,3)', 'sense_stench(0,2)']
Agent juega: ('goto', 1, 1)
['.', 'W', '.', '.']
['.', 'A', '.', '.']
['.', '@', '.', '@']
['@', '.', '.', '.']
Encontraste el oro en 5 pasos!! Felicidades!
```

Usando el visualizador

Nuestra tarea viene con un visualizador. Una vez que tu código esté funcionando, solo debes abrir `index.html` usando un browser y cargar el archivo `simulation.txt`, que se genera automáticamente después de una ejecución. (En los archivos base, viene ya un archivo `simulation.txt` que corresponde al ejemplo de arriba.)

Qué entregar en esta parte

- Un archivo `Parte4/wumpus.lp` con la parte 1.
- Un archivo `Parte4/agent.py` con la parte 2.

Antes de hacer commit, asegúrate que leíste la página 1 de esta tarea.