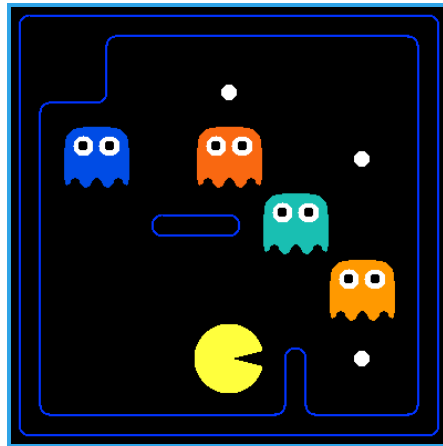




Universidad  
Rey Juan Carlos

## Práctica 2: Agentes lógicos<sup>1</sup>

Inteligencia Artificial



### Introducción

En esta práctica se usarán/escribirán funciones Python que generarán sentencias lógicas para describir las físicas de Pacman, que denominamos pacphysics. Después se usará un solucionador SAT, pycosat, para resolver las inferencias lógicas asociadas con planning (generar la secuencia de acciones para alcanzar las localizaciones objetivo y comer todos los puntos).

De nuevo, se utilizará el autograder para puntuar las respuestas de esta práctica. Se puede ejecutar:

```
python3 autograder.py
```

El código de este proyecto consta de varios archivos de Python, algunos de los cuales habrá que leer y comprender para completar la tarea y algunos de los cuales pueden ignorarse por no ser directamente relevantes al trabajo a realizar. Todo el código y los archivos de apoyo pueden descargarse como un archivo zip desde el Aula Virtual.

---

<sup>1</sup> Esta práctica es una adaptación y traducción de una práctica de la Asignatura de Inteligencia Artificial de la Universidad de Berkeley, disponible [aquí](#).

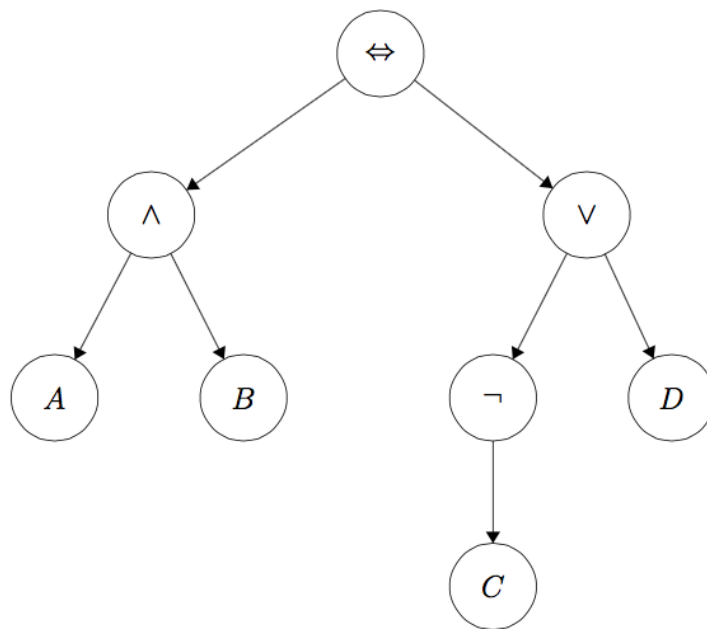
<b>Archivos a editar</b>	
<code>logicPlan.py</code>	Donde se implementarán todos los algoritmos de planificación lógica.
<b>Archivos que deberían consultarse</b>	
<code>logic.py</code>	Código de lógica proposicional con modificaciones para la práctica. Contiene varias funciones auxiliares para trabajar con lógica.
<code>logicAgents.py</code>	El archivo que define la planificación lógica del problema que Pacman encontrará en este proyecto.
<code>pycosat_test.py</code>	Función principal de prueba rápida que verifica que el módulo <code>pycosat</code> esté instalado correctamente.
<code>game.py</code>	La lógica interna del mundo Pacman. Lo único relevante aquí es la clase <code>Grid</code> .
<code>test_cases/</code>	Directorio que contiene los casos de prueba para cada pregunta.
<b>Archivos que pueden ignorarse</b>	
<code>pacman.py</code>	El archivo principal que ejecuta los juegos de Pacman.
<code>logic_util.py</code>	Funciones auxiliares para <code>logic.py</code> .
<code>util.py</code>	Funciones auxiliares para otros proyectos.
<code>graphicsDisplay.py</code>	Gráficos para Pacman.
<code>graphicsUtils.py</code>	Soporte para gráficos Pacman.
<code>textDisplay.py</code>	ASCII para Pacman.
<code>ghostAgents.py</code>	Agentes para controlar fantasmas.
<code>keyboardAgents.py</code>	Interfaces de teclado para controlar Pacman.
<code>layout.py</code>	Código para leer archivos de diseño y almacenar su contenido.

## La clase `Expr`

En la primera parte de este proyecto, se trabajará con la clase `Expr` definida en `logic.py` para construir oraciones lógicas proposicionales. Un objeto `Expr` se implementa como un árbol con operadores lógicos ( $\wedge$ ,  $\vee$ ,  $\neg$ ,  $\Rightarrow$ ,  $\Leftrightarrow$ ) en cada nodo y con literales ( $A$ ,  $B$ ,  $C$ ) en las hojas. La oración:

$$(A \wedge B) \Leftrightarrow (\neg C \vee D)$$

se representaría como el árbol



Para instanciar un símbolo llamado 'A', llamamos al constructor así:

```
A = Expr('A')
```

La clase `Expr` permite usar operadores de Python para construir expresiones. Los operadores de Python disponibles y sus significados son los siguientes:

- `~ A`:  $\neg A$
- `A & B`:  $A \wedge B$
- `A | B`:  $A \vee B$
- `A >> B`:  $A \Rightarrow B$
- `A % B`:  $A \Leftrightarrow B$

Entonces, para construir la expresión  $A \wedge B$ , se escribe:

```
A = Expr ('A')
```

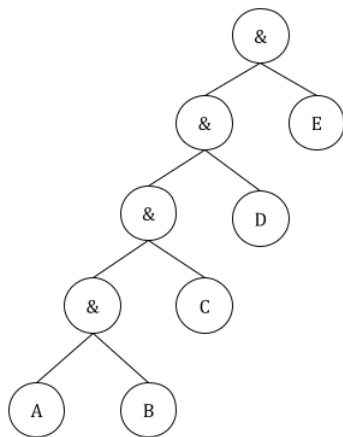
```
B = Expr ('B')
```

```
a_and_b = A & B
```

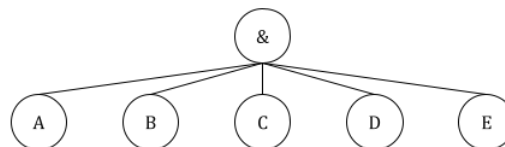
(Tener en cuenta que **A** a la izquierda del operador de asignación en ese ejemplo es solo un nombre de variable de Python, es decir, `simbolo1 = Expr ('A')` habría funcionado igual de bien).

## Una nota sobre `conjoin` y `disjoin`

Otra cosa importante para tener en cuenta es que se deben usar los operadores `conjoin` y `disjoin` siempre que sea posible. `conjoin` crea una expresión encadenada `&` y `disjoin` crea una expresión lógica encadenada `|` (OR lógico). Un ejemplo sería que se quisiera comprobar que las condiciones A, B, C, D y E son ciertas. La forma directa de evaluar esta sentencia sería `condition = A & B & C & D & E` pero esto se traduce realmente como `((((A & B) & C) & D) & E)`, lo que en realidad crea un árbol lógico anidado (mira el diagrama 1 debajo) y se convierte en una pesadilla para depurar. `conjoin` crea un árbol plano (diagrama 2).



(1) `A & B & C & D & E`



(2) `conjoin([A, B, C, D, E])`

## Símbolos de nombrado de Prop

En el resto del proyecto, se debe utilizar la siguiente convención de variables:

### Reglas

- Cuando se introduce una variable, debe empezar por un caracter en mayúscula (incluyendo `Expr`).

- Solo estos caracteres deberían aparecer en los nombres de variables: `A-Z`, `a-z`, `0-9`, `_`, `^`, `[`, `]`.
- Los conectores lógicos (`&`, `|`) no pueden aparecer en nombres de variables. Así, `Expr('A & B')` es ilegal porque intenta crear una única constante de un símbolo llamado `'A & B'`. Se debería utilizar `Expr('A') & Expr('B')` en su lugar para crear la expresión lógica.

## Símbolos de Pacphysics

- `PropSymbolExpr(pacman_str, x, y, time=t)`: si Pacman está en la posición (x,y) en el tiempo t o no. Se escribe `P[x,y]_t`.
- `PropSymbolExpr(wall_str, x, y)`: si hay o no pared en la posición (x, y), se escribe `WALL[x,y]`.
- `PropSymbolExpr(action, time=t)`: si Pacman toma la acción `action` en el tiempo t o no, donde la `action` es un elemento de `DIRECTIONS` que escribe `North_t`.
- En general, `PropSymbolExpr(str, a1, a2, a3, a4, time=a5)` crea la expresión `str(a1, a2, a3, a4)_a5` donde `str` es tan solo un string.

Hay una documentación más detallada de `Expr` en la clase `logic.py`.

## Configuración del solver SAT

El SAT (satisfacibilidad) toma una expresión lógica que codifica las reglas del mundo y devuelve un modelo (asignaciones verdaderas y falsas a símbolos lógicos) que satisface esa expresión si tal modelo existe. Para encontrar de manera eficiente un posible modelo a partir de una expresión, aprovechamos el módulo `pycosat`, que es un wrapper de Python alrededor de la biblioteca `picoSAT`.

### Para instalar este software:

En la línea de comandos, ejecutar:

```
pip3 install pycosat.
```

### Probando la instalación de pycosat:

Después de descomprimir el código del proyecto y cambiar al directorio del código del proyecto, ejecutar:

```
python3 pycosat_test.py
```

Esto debería dar como resultado:

```
[1, -2, -3, -4, 5].
```

**Esto es fundamental para completar el proyecto.**

# Pregunta 1: calentamiento de la lógica

Esta pregunta está enfocada en practicar el uso de expresiones lógicas. Se utilizará el tipo de datos `Expr` en el proyecto para representar oraciones lógicas proposicionales.

Completar las funciones `sentence1()`, `sentence2()`, `sentence3()`, `findModelCheck()`, `entails(premise, conclusion)`, `plTrueInverse(assignments, inverse_statement)` en el archivo `logicPlan.py`, con instancias específicas de `logic.Expr`.

- Para `sentence1()`, crear una instancia de `logic.Expr` que represente que las siguientes tres oraciones son verdaderas, sin ninguna simplificación lógica, simplemente encadenando las sentencias en el formulario en este orden.

$$A \vee B$$

$$\neg A \Leftrightarrow (\neg B \vee C)$$

$$\neg A \vee \neg B \vee C$$

- Para `sentence2()`, crear una instancia de `logic.Expr` que represente que las siguientes cuatro frases son verdaderas. Nuevamente, sin simplificaciones lógicas, simplemente introduciéndolos en este formulario en este orden.

$$C \Leftrightarrow (B \vee D)$$

$$A \Rightarrow \neg (B \wedge \neg D)$$

$$\neg (B \wedge \neg C) \Rightarrow A$$

$$\neg D \Rightarrow C$$

Para los problemas de planificación más adelante en el proyecto, se tendrán símbolos con nombres como:

$$P[3,4,2]$$

que representan que Pacman está en la posición (3,4) en el tiempo 2. Se usarán en expresiones lógicas como la anterior en lugar de A, B, C o D. La función de `logic` `PropSymbolExpr` ofrece una herramienta útil para crear símbolos como `P[3,4,2]` que tienen información numérica codificada en su nombre (para esto, podemos escribir `PropSymbolExpr('P', 3, 4, 2)`).

- Para `sentence3()` y usando el constructor `logic.PropSymbolExpr` crear los símbolos `PacmanAlive_0`, `PacmanAlive_1`, `PacmanBorn_0` y `PacmanKilled_0`. Pista: recuerda que `PropSymbolExpr(str, a1, a2, a3, a4, time=a5)` crea la expresión `str[a1,a2,a3,a4]_a5` donde `PacmanAlive_0` es un string. Se deberían crear varios strings para este problema. Después, crea una instancia de `PacmanAlive_0` que codifique las siguientes 3 frases en lógica proposicional y sin ninguna simplificación:
  1. Pacman está vivo en el tiempo 1 si y solo si estaba vivo en el tiempo 0 y no fue asesinado en el tiempo 0 o no estaba vivo en el tiempo 0 y nació en el tiempo 0.
  2. En el tiempo 0, Pacman no puede estar a la vez vivo y muerto.
  3. Pacman nace en el tiempo 0.

Devolver la conjunción de las tres sentencias anteriores.

- `findModelCheck()`:
  1. Mira cómo funciona el método `findModel(sentence)`: utiliza `to_cnf` para convertir sentencias de entrada a forma normal conjuntiva (la forma que necesita el solucionador SAT), y después las pasa al solucionador SAT para encontrar una asignación satisficible al símbolo en `sentence`. Un modelo es un diccionario de los símbolos en su expresión y su correspondiente asignación True o False. Puedes testear tus `sentence1()`, `sentence2()`, `sentence3()` con `findModel` abriendo una sesión interactiva de Python y ejecutando `from logicPlan import *` y `findModel(sentence1())` y queries similares para los otros dos. ¿La salida es lo que esperabas?
  2. Basado en lo anterior, completa `findModelCheck` para que devuelva algo que tenga la forma exactamente de `findModel(Expr('a'))` en una sesión interactiva de Python si las letras en minúscula estuvieran permitidas. No se debería usar `findModel` ni `Expr`, simplemente recrea la salida directamente. Por ejemplo, si la salida fuese `[(MyVariable, True)]`, algo parecido a la solución sería `return [("MyVariable", True)]`
- `entails(premise, conclusion)`: devuelve True si y solo si la `premise` implica la `conclusion`. Pista: `findModel` es de ayuda en este caso. Piensa en qué debe ser no satisficible para que la implicación sea True y que significa que algo sea no satisficible.
- `plTrueInverse(assignments, inverse_statement)`: devuelve True si y solo si el `inverse_statement` es True dados las asignaciones.

Antes de continuar, prueba instanciando una sentencia pequeña, como  $A \Rightarrow \neg(B \wedge \neg D)$  y llama a `to_cnf` sobre ello. Inspecciona la salida t y asegúrate de que la entiendes.

Para probar y depurar el código, ejecutar:

```
python3 autograder.py -q q1
```

## Pregunta 2: ejercicios lógicos

Implementar las siguientes tres expresiones lógicas en `logicPlan.py`:

- `atLeastOne(literales)`: Devuelve una sola expresión (`logic.Expr`) en CNF que es verdadera solo si al menos una expresión en la **lista** de entrada es verdadera. Cada expresión de entrada será un literal.
- `atMostOne(literales)`: devuelve una sola expresión (`logic.Expr`) en CNF que es verdadera solo si como máximo una expresión en la **lista** de entrada es verdadera. Cada expresión de entrada será un literal. Pista: Utiliza `to_cnf`. Si tienes  $n$  literales y como mucho uno de ellos es cierto, el CNF resultante debería ser la conjunción de  $\frac{n}{2}$  cláusulas.
- `exactlyOne(literales)`: devuelve una sola expresión (`logic.Expr`) en CNF que es verdadera solo si exactamente una expresión en la **lista** de entrada es verdadera. Cada expresión de entrada será un literal. Si llamas a alguno de los métodos previamente implementados (`atLeastOne` o `atMostOne`) llama a `atLeastOne` primero para pasar el autograder de la q3.

Las definiciones de funciones se han esbozado en `logicPlan.py`. No modificar la forma de la función.

Cada uno de estos métodos toma una lista de literales `logic.Expr` y devuelve una única expresión `logic.Expr` que representa **la relación lógica apropiada entre las expresiones** en la lista de entrada. **Un requisito adicional es que el Expr devuelto debe estar en CNF (forma normal conjuntiva) sin utilizar la función `logic.to_cnf` (ni las funciones auxiliares `logic.eliminate_implications`, `logic.move_not_inwards`, y `logic.distribute_and_over_or`).**

Al implementar el agente en las preguntas posteriores, no habrá que realizar conversiones a CNF hasta justo antes de enviar la expresión al solucionador de SAT (momento en el que puede usar `findModel` de la pregunta 1). `logic.to_cnf` implementa el algoritmo de la *sección 7.5* en AIMA. Sin embargo, para ciertas entradas en el peor de los casos, la implementación directa de este algoritmo puede resultar en oraciones de tamaño exponencial. De hecho, una implementación no CNF de una de estas tres funciones es uno de esos peores casos.

Por lo tanto, si se va a utilizar la funcionalidad de `atLeastOne`, `atMostOne` o `exactOne` para una pregunta posterior, es muy recomendable usar las funciones implementadas aquí para evitar usar accidentalmente una alternativa que no sea CNF y pasarla a `logic.to_cnf`. Si se hace esto, el código será tan lento que ni siquiera podrá resolver un laberinto de 3x3 sin paredes.

Se puede utilizar la función `logic.pl_true` para probar la salida de las expresiones. `pl_true` toma una expresión y un modelo y devuelve `True` si y sólo si la expresión es verdadera dado el modelo.

Para probar y depurar el código, ejecutar:

```
python3 autograder.py -q q2
```



## Pregunta 3: pacphysics y Satisfabilidad

En esta pregunta, se implementarán expresiones lógicas de pacphysics y se aprenderá cómo probar dónde está el Pacman y dónde no construyendo la base de conocimiento (KB) apropiada de expresiones lógicas.

Se implementarán las siguientes funciones en `logicPlan.py`:

- `pacmanSuccessorAxiomSingle`: esto genera una expresión definiendo las condiciones suficientes y necesarias para que el Pacman esté en (x,y) en el tiempo t:
  - Lee la construcción de `possible_casuses` proporcionada.
  - Necesitas rellenar la sentencia de devolución, que estará en `Expr`. Asegúrate de usar `disjoin` y `conjoin` donde sea apropiado.
- `pacphysicsAxioms`: Aquí se generará un conjunto de axiomas de físicas. Para el timestep t:
  - Argumentos:
    - Requerido: `t` = tiempo, `all_coords` y `non_outer_wall_coords` son listas de tuplas (x,y).
    - Possibly-None: se usarán estas para llamar a funciones. No es necesaria mucha lógica.
      - `walls_grid` solo se pasa a través de `successorAxioms` y describe paredes conocidas.
      - `sensorModel(t: int, non_outer_wall_coords) -> Expr` devuelve una sola `Expr` que describe las reglas de observación.
      - `successorAxioms(t: int, walls_grid, non_outer_wall_coords) -> Expr` describe las reglas de transición. Esto es, cómo las localizaciones previas y acciones del Pacman afectan a la localización actual.
  - Algoritmo:
    - Para todas las (x,y) en `all_coords` añade la siguiente implicación (forma if-then): si hay una pared en (x,y), entonces el Pacman no está en (x,y) en el tiempo t.
    - Pacman exactamente uno de los `non_outer_wall_coords` en el tiempo t.
    - Pacman toma exactamente una de las cuatro acciones en `DIRECTIONS` en el tiempo t.
    - Sensores: añade el resultado de `sensorAxioms`. Todos los llamadores excepto `checkLocationSatisfiability` hacen uso de esto. Se deja al criterio del alumno cómo manejar el caso donde no se quiere ningún axioma de sensor.
    - Transiciones: añade el resultado de `successorAxioms`. Todos los llamadores hacen uso de esto.

- Añade cada una de las sentencias superiores a `pacphysics_sentences`. Como puedes ver en el estamento de `return`, se hará un conjoin y se devolverá.
- Sintaxis de paso de funciones:
  - Sea `def myFunction(x, y, t): return PropSymbolExpr('hello', x, y, time=t)` una función que queremos usar.
  - Sea `def myCaller(func: Callable): ...` un caller que quiere utilizar la función.
  - Pasamos la función en: `myCaller(myFunction)` (ten en cuenta que `myFunction` no se llama con `()` después de él).
  - Utilizamos `myFunction` teniendo dentro de `myCaller` lo siguiente: `useful_return = func(0, 1, q)`.
- `CheckLocationSatisfability`: dada la transición `(x0_y0, action0, x1_y1)`, `action1` y un `problem`, se deberá escribir una función que devuelva una tupla con 2 modelos `(model1, model2)`.
  - En `model1`, Pacman está en `(x1, y1)` en el tiempo `t = 1` dado `x0_y0, action0, action1`, probando que es posible que Pacman esté ahí. Notablemente, si `model1` es `False`, sabemos que Pacman está garantizado que no está ahí.
  - En `model2`, Pacman no está en `(x1, y1)` en el tiempo `t = 1` dado `x0_y0, action0, action1`, probando que es posible que Pacman no esté ahí. Notablemente, si `model2` es `False`, entonces está garantizado que el Pacman esté ahí.
  - `action1` no tiene efecto en determinar si el Pacman está o no en la localización. Está ahí solo para hacer la igualdad para la solución del autograder.
  - Para implementar este problema se necesitará añadir las siguientes expresiones lógicas a la KB:
    - Añadir a la KB: `pacphysics_axioms(...)` con los timesteps adecuados. No hay `sensorModel` porque sabemos todo respecto al mundo. Donde es necesario, utiliza `allLegalSuccessorAxioms` para las transiciones teniendo en cuenta que esto es para las reglas de transmisión regular del Pacman.
    - Añadir a la KB: la posición actual de Pacman `(x0, y0)`.
    - Añadir a la KB: que Pacman toma `action0`.
    - Añadir a la KB: que Pacman toma `action1`.
  - Haz una query al solucionador SAT con `findModel` para los dos modelos descritos anteriormente. Las queries deberían ser diferentes. Se puede revisar `entails` para recordar cómo son las queries.

Recuerda: la variable que dice si Pacman está en (x,y) o no en un tiempo t es `PropSymbolExpr(pacman_str, x, y, time=t)`, si hay pared en (x, y) `PropSymbolExpr(wall_str, x, y)`, y si la acción se ha tomado en el tiempo t es `PropSymbolExpr(action, time=t)`.

Para probar y depurar el código, ejecute:

```
python3 autograder.py -q q3
```

## Pregunta 4: Path Planning con lógica

Pacman está intentado encontrar el final del laberinto (la posición objetivo). Implementa el siguiente método usando lógica proposicional para planificar la secuencia de acciones que harán que Pacman llegue hasta el objetivo.

Cuidado, los métodos de ahora en adelante serán bastante lentos en ejecución. Esto es así porque un resolutor SAT es muy general y simplemente analiza la lógica, no como algoritmos previos que empleaban un algoritmo específico creado por un humano para un tipo específico de problema. El algoritmo principal de Pycosat está en C, que es normalmente un lenguaje mucho más rápido de ejecutar que Python y aún contando con eso, es lento en este caso.

- `positionLogicPlan(problem)`: dada una instancia de `logicPlan.PlanningProblem`, devuelve una secuencia de acciones string para que el agente Pacman las ejecute.

No se implementará un algoritmo de búsqueda, sino que se crearán expresiones que representarán `pacphysics` para todas las posibles posiciones en todos los instantes de tiempo. Esto significa que en cualquier paso de tiempo, se debería añadir reglas generales para todas las posibles localizaciones en la malla, donde las reglas no asuman nada sobre la posición actual de Pacman.

Se necesitará codificar las siguientes sentencias en la base de conocimiento, siguiendo el siguiente pseudocódigo:

- Añadir a la KB: el conocimiento inicial será la localización de Pacman en el instante de tiempo 0.
- For t in range(50) (el Autograder no se ejecutará en entornos que requieran más de 50 pasos de tiempo):
  - `Print(time step)`: para ver que el código se está ejecutando.
  - Añadir a la KB: Conocimiento inicial. Pacman solo puede estar en `exactlyOne` de las localizaciones en `non_wall_coords` en el instante de tiempo t. Esto es similar a `pacphysicsAxioms`, pero no utilizan ese método ya que estamos utilizando `non_wall_coods` al general la lista de posibles localizaciones en primer lugar (y después `walls_grid`).
  - ¿Existe alguna asignación que satisface las variables dadas en la base del conocimiento hasta ahora? Utiliza `findModel` y pasa el Goal Assertion y KB.
    - Si existe, devuelve la secuencia de acciones desde el inicio al objetivo utilizando `extractActionSequence`.

- Aquí Goal Assertion es la expresión que comprueba que Pacman está en el objetivo en el tiempo  $t$ .
- Añade a la KB: Pacman toma exactamente una action por cada instante de tiempo.
- Añade a la KB: sentencias de Transition Model. Llama a `pacmanSuccessorAxiomSingle(...)` para todas las posibles posiciones de Pacman en `non_wall_coords`.

Para probar el código en laberintos más pequeños usaremos:

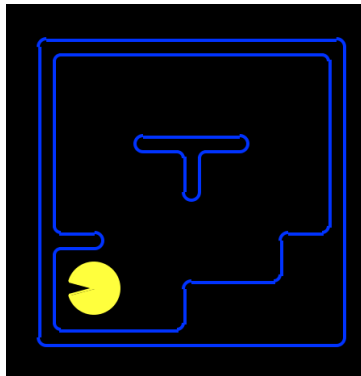
```
python3 pacman.py -l maze2x2 -p LogicAgent -a fn=plp
```

```
python3 pacman.py -l tinyMaze -p LogicAgent -a fn=plp
```

Para probar y depurar el código, ejecutar:

```
python3 autograder.py -q q4
```

Ten en cuenta que con la forma en que tenemos la cuadrícula de Pacman, el espacio más a la izquierda y más abajo que Pacman puede ocupar (suponiendo que no haya una pared allí) es (1, 1), como se muestra a continuación (no (0, 0)).



Resumen de la Pacphysics utilizada en Q3 y Q4:

- Para todo  $x, y, t$ : si hay un muro en  $(x, y)$ , entonces Pacman no está en  $(x, y)$  en  $t$ .
- Para cada  $t$ : Pacman está exactamente en una de las ubicaciones descritas por todos los posibles  $(x, y)$ . Se puede optimizar con el conocimiento de las paredes exteriores o de todas las paredes, siguiendo las especificaciones para cada función.
- Para cada  $t$ : Pacman toma exactamente una de las acciones posibles.
- Para cada  $t$  (excepto para  $t = ??$ ): Modelo de transición: Pacman está en  $(x, y)$  en  $t$  si y solo si estaba en  $(x - dx, y - dy)$  en  $t-1$  y actuó  $(dx, dy)$  en  $t-1$ .

Ten en cuenta que lo anterior siempre es válido independientemente de cualquier juego específico, acción, etc. A las reglas anteriores de siempre verdadero / axioma, agregamos información consistente con lo que sabemos.

Pistas para la depuración:

- Si está encontrando una solución de longitud 0 o longitud 1: ¿es suficiente simplemente tener axiomas para saber dónde está Pacman en un momento dado? ¿Qué le impide estar también en otros lugares?
- Como sanity check, verifica que si Pacman está en (1, 1) en el tiempo 0 y en (4, 4) en el tiempo 6, nunca estuvo en (5, 5) en ningún momento intermedio.
- Si tu solución tarda más de un par de minutos en terminar de ejecutarse, es posible que desee volver a revisar la implementación de `exactlyOne` y `atMostOne`, y asegurarse de que está usando la menor cantidad de cláusulas posible.

## Pregunta 5: comiendo toda la comida

Implemente el siguiente método usando la lógica proposicional para planificar la secuencia de acciones de Pacman que lo llevarán a la meta, mientras come.

`foodLogicPlan(problem)`

Dada una instancia de `logicPlan.PlanningProblem`, devuelve una secuencia de cadenas de acción para que las ejecute el agente Pacman.

Esta pregunta tiene el mismo formato general que la pregunta 4. Las notas y sugerencias de la pregunta 4 también se aplican a esta pregunta. Hay que implementar ciertos axiomas de estado sucesor adicionales que no se implementaron en las preguntas anteriores.

Lo que se cambiará de la pregunta anterior:

- Inicializa las variables `Food[x,y]_t` con el código `PropSymbolExpr(food_str, x, t, time=t)`, donde cada variable es verdadera si y solo si hay un alimento en (x, y) en el momento t.
- Cambia la afirmación del objetivo: su oración de afirmación de la meta debe ser verdadera si y solo si se ha comido toda la comida. Esto sucede cuando todos los `Food[x,y]_t` son falsos.
- Agrega un axioma del sucesor de comida: ¿Cuál es la relación entre `Food[x,y]_t+1` y `Food[x,y]_t` y `Pacman[x,y]_t`? El axioma del sucesor de alimentos solo debe involucrar estas tres variables, para cualquier (x, y) y t dados. Piense en cómo se ve el modelo de transición para las variables alimentarias y agregue estas oraciones a su base de conocimientos en cada paso de tiempo.

Para ejecutar el código:

```
python3 pacman.py -l testSearch -p LogicAgent -a
fn=flp,prob=FoodPlanningProblem
```

No se aceptarán soluciones que requieran más de 50 pasos de tiempo.

Para probar y depurar su código, ejecute:

```
python3 autograder.py -q q5
```

## Entregables

- Código Python de los ejercicios propuestos en el archivo `logicPlan.py`.
- El código debe ir debidamente comentado explicando su funcionalidad de forma obligatoria. Debe ser legible y estar debidamente tabulado.
- Se utilizarán sistemas anticopia y se podrá requerir explicación individual de la práctica en caso de duda.
- Esta entrega se realizará en formato zip (incluyendo los archivos .py) vía Aula Virtual.
- Fecha de entrega: la entrega de la práctica se realizará **el martes 15 de noviembre** al finalizar la clase de prácticas (11:00-13:00). Durante la sesión, se propondrá una pequeña modificación a la práctica que deberá ser también entregada por los alumnos, por lo que la asistencia ese día será obligatoria.