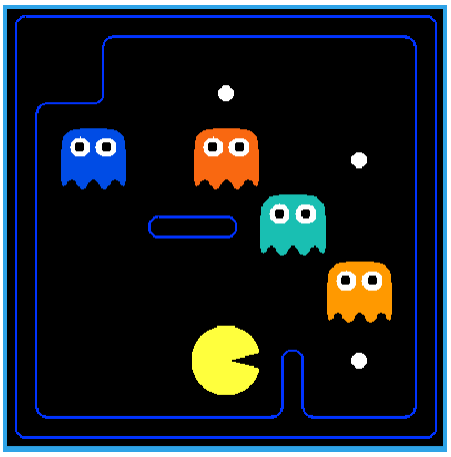
Universidad Rey Juan Carlos

Práctica 2: Agentes Lógicos[1]

Inteligencia Artificial
Curso 2023/2024



Introducción

En esta práctica se usarán/escribirán funciones Python que generarán sentencias lógicas para describir las físicas de Pacman, que denominamos **PacPhysics (PacFísicas)**. Después se usará un solucionador SAT, **pycosat**, para resolver las inferencias lógicas asociadas con planificación (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.

[1] Esta práctica es una adaptación y traducción de una práctica de la Asignatura de Inteligencia Artificial de la Universidad de Berkeley, disponible aquí.

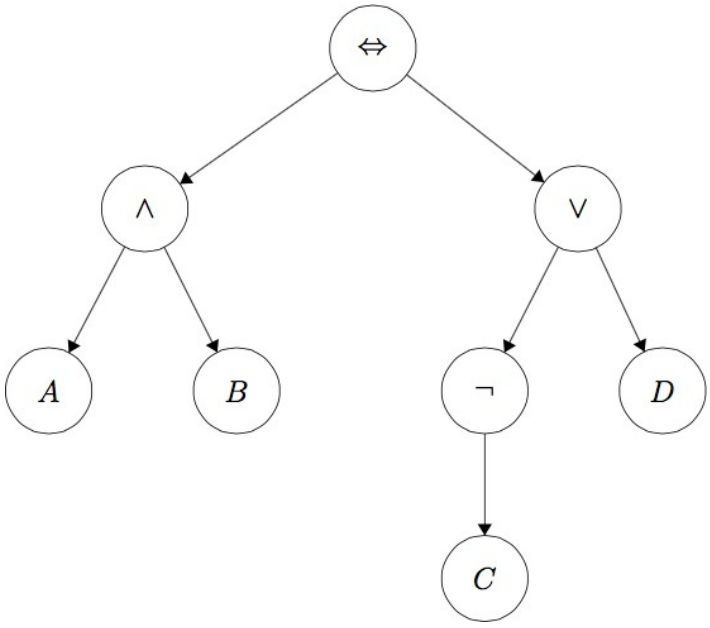
Archivos a editar	
`logicPlan.py`	Donde se implementarán todos los algoritmos de planificación lógica.
Archivos que deberían consultarse	
`logic.py`	Código de lógica proposicional con modificaciones para la práctica. Contiene varias funciones auxiliares para trabajar con lógica.
`logicAgents.py`	El archivo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

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

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`: $A \Leftrightarrow B$

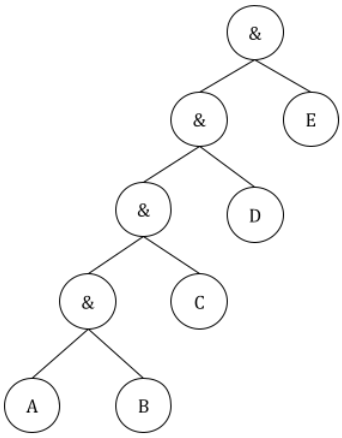
Entonces, para construir la expresión $A \wedge B$, se escribe:

```
A = Expr('A')

B = Expr('B')

a_and_b = A & B
```

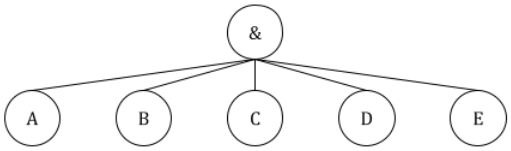
(Tener en cuenta que `a` a la izquierda del operador de asignación en ese ejemplo es solo un nombre de variable de Python, es decir, `simbolo1 = Expr('A')` habría funcionado igual de bien).

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 `&` (AND lógico) 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` por su parte crea un árbol plano (diagrama 2).



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



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

Nombres de símbolos proposicionales (importante)

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

Reglas

- Cuando se introduce una variable, debe empezar por un carácter 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, escribe `P[x,y]_t`.
- `PropSymbolExpr(wall_str, x, y)`: si hay o no pared en la posición (x, y), 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`; escribe por ejemplo `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 solucionador SAT (fundamental para completar la práctica)

El solucionador 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]
```

Pregunta 1: calentamiento

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

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

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

$$A \vee B$$

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

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

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

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

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

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

$$\neg D \Rightarrow C$$

  Nota: Para los problemas de planificación más adelante en la práctica, se tendrán símbolos con nombres como:

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

  que representan que Pacman está en la posición $(3,4)$ en el instante de tiempo 2. Se usarán en expresiones lógicas como la anterior en lugar de A, B, C o D. Como se mencionó anteriormente, la función de `logic.py` `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)`).

- `sentence3()`: usando el constructor `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 `str` es un *string*. Se deberían crear varios *strings* para este problema.

Después, crea una instancia de `Expr` 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 nacer.
3. Pacman nace en el tiempo 0.

Devolver la conjunción de las tres sentencias anteriores.

- `entails(premise, conclusion)`: devuelve True si y solo si la `premise` implica la `conclusion`. Pista: `findModel`, que toma como entrada una sentencia lógica proposicional (Expr) y devuelve un modelo que la satisface (si existe), puede ser de ayuda en este caso. Piensa en qué debe ser no satisfacible para que la implicación sea `True` y que significa que algo sea no satisfacible.
- `plTrueInverse(assigments, inverse_statement)`: devuelve `True` si y solo si el (`not inverse_statement`) es verdadero dadas las asignaciones.

Para probar y depurar el código, ejecutar:

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

Nota: Forma Normal Conjuntiva

Una sentencia se encuentra descrita en forma normal conjuntiva si corresponde a una conjunción (&) de claúsulas definidas únicamente por los operadores lógicos conjunción (&), disyunción (|) y negación (~). Para poder convertir una sentencia a Forma Normal Conjuntiva, se ha implementado la función `to_cnf`, que funciona de forma similar al método de conversión a CNF que se ha visto en clase.

Antes de continuar, prueba instanciando una sentencia pequeña, como $A \Rightarrow \neg(B \wedge \neg D)$ y pásala a forma normal conjuntiva (CNF) a través del uso de la función `to_cnf(sentence)`. Inspecciona la salida t y asegúrate de que la entiendes.

Pregunta 2: ejercicios lógicos

Implementar las siguientes tres expresiones lógicas utilizando las funciones cuya definición ya se ha esbozado en `logicPlan.py` sin modificar la forma de las mismas:

- `atLeastOne(literales)`: Devuelve una sola expresión (`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 (`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 `itertools.combinations`.
- `exactlyOne(literales)`: devuelve una sola expresión (`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.

Cada uno de estos métodos toma una lista de literales `Expr` y devuelve una única expresión `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*** `to_cnf` ***(ni otras funciones auxiliares)***

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
```

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). Tal y como se ha mencionado anteriormente, `to_cnf` funciona de forma similar al método de conversión a CNF que se ha visto en clase. Sin embargo, para ciertas entradas, en el peor de los casos, la implementación directa de este método 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 `to_cnf`. Si se hace esto, el código será tan lento que ni siquiera podrá resolver un laberinto de 3x3 sin paredes.

Pregunta 3: PacPhysics y satisfacibilidad

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_causes proporcionada. Verás que contiene las posibilidades para que Pacman esté en (x,y) en el tiempo t.
    - Necesitas rellenar la sentencia que se devuelve, que estará en `Expr`. Asegúrate de usar `disjoin` y `conjoin` donde sea apropiado. Pista: hay que relacionar la posición de Pacman en (x,y) con las posibles causas.
- `pacphysicsAxioms`: Aquí se generará un conjunto de axiomas de físicas. Para el tiempo t:
    - Argumentos:
        - Requerido: `t` = tiempo, `all_coords` y `non_outer_wall_coords` son listas de tuplas (x,y).
        - Opcionales: se usarán estas para llamar a funciones. No es necesaria mucha lógica.
            - `walls_grid` solo se pasa a través de `successorAxioms` y describe pareces 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 sentencia (forma *if-then*): si hay una pared en (x,y), entonces el Pacman no está en (x,y) en el tiempo t.
        - Pacman exactamente una 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 `sensorModel`. Todos los llamadores de `pacphysicsAxioms` 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 de `pacphysicsAxioms` hacen uso de esto. Es importante tener en cuenta que `successorAxioms` no debe ser llamada para el caso en que el *timestamp* t es 0.
        - Añade cada una de las sentencias superiores a `pacphysics_sentences`. Como puedes ver en la línea 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 dos 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 solucionador 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, aun contando con eso, es lento en este caso.

- `positionLogicPlan(problem)`: dada una instancia de `logicPlan.PlanningProblem`, devuelve una secuencia de acciones en forma de 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 el tablero, 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: 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 utilices ese método ya que estamos utilizando `non_wall_coods` al generar 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 pásale 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`.

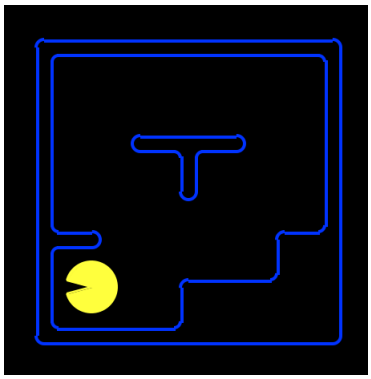
Para probar el código en laberintos más pequeños usaremos:

```
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 las PacPhysics utilizadas 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 (unir con o: (x - dx, y - dy) en t-1 y tomó la acción (dx, dy) en t-1).

Ten en cuenta que lo anterior siempre es válido independientemente de cualquier partida específica, acciones, etc. A las reglas anteriores siempre verdaderas/axiomas, agregamos información consistente con lo que sabemos.

Pistas para la depuración:

- Si encuentras 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 quieras volver a revisar la implementación de `exactlyOne` y `atMostOne`, y asegurarte de que está usando la menor cantidad de cláusulas posible.

Pregunta 5: comerse toda la comida

Pacman quiere comerse toda la comida en el tablero. Implementa el siguiente método usando la lógica proposicional para planificar la secuencia de acciones de Pacman que lo llevarán a la meta:

- `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 sucesión de estados 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 el código, ejecutar:

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

Preguntas opcionales

Funciones auxiliares para el resto del Proyecto

Para las preguntas restantes, dependeremos de las siguientes funciones auxiliares, a las que hará referencia el pseudocódigo para la localización, mapeo y SLAM.

(Aux1) Agregar información de PacPhysics, acción y percepción a la KB

- Agregar a la KB: `pacphysics_axioms(...)`, que escribiste en la pregunta 3. Utiliza `sensorAxioms` y `allLegalSuccessorAxioms` para la localización y el mapeo, y `SLAMSensorAxioms` y `SLAMSuccessorAxioms` solo para SLAM.
- Agregar a la KB: Pacman toma la acción prescrita por `agent.actions[t]`
- Obtener las percepciones llamando a `agent.getPercepts()` y pasa las percepciones a `fourBitPerceptRules(...)` para la localización y el mapeo, o `numAdjWallsPerceptRules(...)` para SLAM. Agrega las reglas de percepción resultantes a la `KB`.

(Aux2) Encontrar posibles ubicaciones de Pacman con la KB actualizada

- `possible_locations = []`
- Iterar sobre `non_outer_wall_coords`.
    - ¿Podemos demostrar si Pacman está en (x, y)? ¿Podemos demostrar si Pacman no está en (x, y)? Usa `entails` y la `KB`.
    - Si existe una asignación satisfactoria donde Pacman está en (x, y) en el tiempo t, agrega (x, y) a `possible_locations`.
    - Agregar a la KB: ubicaciones (x, y) donde Pacman está probadamente en el tiempo t.
    - Agregar a la KB: ubicaciones (x, y) donde Pacman no está probadamente en el tiempo t.
    - Sugerencia: verifica si los resultados de `entails` se contradicen entre sí (es decir, KB implica A y también implica ¬A). Si lo hacen, imprime retroalimentación para ayudar en la depuración.

(Aux3) Encontrar ubicaciones de pared demostrables con la KB actualizada

- Iterar sobre `non_outer_wall_coords`.
    - ¿Podemos demostrar si hay una pared en (x, y)? ¿Podemos demostrar si no hay una pared en (x, y)? Usa `entails` y la KB.
    - Agregar a la KB y actualizar `known_map`: ubicaciones (x, y) donde existe probadamente una pared.
    - Agregar a la KB y actualizar `known_map`: ubicaciones (x, y) donde probadamente no hay una pared.
    - Sugerencia: verifica si los resultados de `entails` se contradicen entre sí (es decir, KB implica A y también implica ¬A). Si lo hacen, imprime retroalimentación para ayudar en la depuración.

Observación: agregamos las ubicaciones conocidas de Pacman y las paredes a la KB para no tener que rehacer el trabajo de encontrarlas en pasos temporales posteriores. Esta información es técnicamente redundante, ya que la demostramos usando la KB en primer lugar.

Pregunta 6: localización

Pacman comienza con un mapa conocido, pero una posición inicial desconocida. Tiene un sensor de 4 bits que indica si hay alguna pared a su alrededor en formato NSEW mediante una lista de 4 booleanos (Por ejemplo, 1001 significa que hay una pared al norte y al oeste de Pacman). Así al realizar un seguimiento de estas lecturas del sensor y la acción tomada en cada instante, Pacman puede determinar su ubicación.

Implementa la función `localization(problem, agent)` que permite a Pacman a determinar las posibles ubicaciones en cada instante:

- Dada una instancia de `logicPlan.LocalizationProblem` y una instancia de `logicAgents.LocalizationLogicAgent`, repite para los instantes t entre 0 y `agent.num_steps - 1` una lista de posibles ubicaciones (x_i, y_i) en t: [(x_0_0, y_0_0), (x_1_0, y_1_0), ...]. No debes preocuparte por el funcionamiento de los generadores
- Para que Pacman haga uso de la información del sensor durante la localización, debes utilizar dos funciones ya definidas: `sensorAxioms` (por ejemplo, Blocked[Direction]_t ↔ [(P[x_i, y_j]_t ∧ WALL[x_i+dx, y_j+dy]) ∨ (P[x_i', y_j']_t ∧ WALL[x_i'+dx, y_j'+dy]) ...]) y `fourBitPerceptRules`, que traducen las percepciones en el tiempo t en sentencias lógicas.

Te recomendamos encarecidamente implementar la función de acuerdo al siguiente pseudocódigo:

- Agregar al `KB`: dónde están las paredes (`walls_list`) y dónde no están (no en `walls_list`).
- `for t in range(agente.num_timesteps)`:
    - (Aux1) Agregar información de pacphysics, acción y percepción a la `KB`.
    - (Aux2) Encontrar posibles ubicaciones de Pacman con la `KB` actualizado.
    - Llamar a `agent.moveToNextState(action_t)` en la acción actual del agente en el instante t.
    - Devolver las posibles ubicaciones mediante `yield`

Nota: el Pacman que vemos por pantalla se encuentra en el tiempo que se está calculando actualmente, por lo que las posibles ubicaciones y las paredes conocidas y los espacios libres corresponden al instante inmediatamente anterior.

Para probar y depurar el código, ejecutar:

```
python3 autograder.py -q q6
```

Pregunta 7: mapeo

Ahora Pacman conoce su posición inicial, pero desconoce la ubicación de las paredes (excepto por el hecho de que el borde de las coordenadas exteriores son paredes). De forma similar al ejercicio anterior, tiene un sensor de 4 bits que indica si hay alguna pared a su alrededor en formato NSEW mediante una lista de 4 booleanos.

Implementa las funciones que ayudarán a Pacman a determinar la ubicación de las paredes:

- `mapping(problem, agent)`: Dado una instancia de `logicPlan.MappingProblem` y una instancia de `logicAgents.MappingLogicAgent`, generará repetidamente conocimiento sobre el mapa para los instantes t entre 0 y `agent.num_steps-1`, [[1, 1, 1, 1], [1, -1, 0, 0], ...] en t. Ten en cuenta que, de nuevo, no necesitas preocuparte por cómo funcionan los generadores.
- `known_map`:
    - `known_map` es un array 2D (lista de listas) de tamaño (problem.getWidth()+2, problem.getHeight()+2), dado que contamos con paredes que rodean el problema.
    - Cada entrada de `known_map` es 1 si se garantiza que (x, y) es una pared en el instante de tiempo t, 0 si (x, y) se garantiza que no lo es y -1 si (x, y) aún es ambiguo para ese instante t. La ambigüedad ocurre cuando no se puede demostrar que (x, y) es una pared, pero tampoco que no lo sea.

Te recomendamos encarecidamente implementar la función de acuerdo al siguiente pseudocódigo:

- Obtén la ubicación inicial (pac_x_0, pac_y_0) de Pacman, y añádela a la KB. También agrega si hay una pared en esa ubicación.
- `for t in range(agente.num_timesteps):`
    - (Aux1) Agrega información de PacPyshics, acción y percepción a la KB.
    - (Aux3) Encuentra ubicaciones de pared demostrables con la KB actualizada.
    - Llama a `agent.moveToNextState(action_t)` en la acción actual del agente en el instante t.
    - Devuelve el mapa conocido a través de `yield known_map`

Para probar y depurar el código, ejecutar:

```
python3 autograder.py -q q7
```

Pregunta 8: SLAM

A veces, Pacman está perdido en un mapa desconocido. En SLAM (Localización y Mapeo Simultáneos), Pacman conoce sus coordenadas iniciales, pero no sabe dónde están las paredes. Tambien puede tomar acciones inadvertidamente ilegales (por ejemplo, ir hacia el norte cuando hay una pared en esas coordenadas) lo que agregará incertidumbre a su ubicación. Además Pacman ya no cuenta con un sensor de 4 bits que indique si hay una pared alguna de las 4 coordenadas de su alrededor, sino que solo tiene un sensor de 3 bits que revela el número de paredes que le rodean. (Esto es algo similar aa los indicadores de fuerza de la señal wifi; 000 = no adyacente a ninguna pared; 100 = adyacente a exactamente 1 pared; 110 = adyacente a exactamente 2 paredes; 111 = adyacente a exactamente 3 paredes. Estos 3 bits se representan con una lista de 3 booleanos.) Por lo tanto, en lugar de usar `sensorAxioms` y `fourBitPerceptRules`, se utilizarán `SLAMSensorAxioms` y `numAdjWallsPerceptRules`.

Implementa las sentencias que ayuden a Pacman a determinar sus posibles ubicaciones en cada instante (1) y la ubicación de las paredes (2).

- `slam(problem, agent)`: Dado una instancia de `logicPlan.SLAMProblem` y `logicAgents.SLAMLogicAgent`, generará repetidamente una tupla de dos elementos:
  - `known_map` en 't' (del mismo formato que en la pregunta 6)
  - una lista de las ubicaciones posibles de Pacman en el instante 't' (en el mismo formato que en la pregunta 5)

Para pasar el autograder, deberás implementar la función de acuerdo al siguiente pseudocodigo:

- Obtén la ubicación inicial `(pac_x_0, pac_y_0)` de Pacman y añádela a la KB. Actualiza `known_map` en consecuencia y añade la expresión apropiada a la KB.
- `for t in range(agente.num_timesteps):`
  - (Aux1) Agrega información de PacPhysics, acción y percepción a la KB. Utiliza `SLAMSensorAxioms`, `SLAMSuccessorAxioms`, y `numAdjWallsPerceptRules`.
  - (Aux3) Encuentra ubicaciones de pared demostrables con la KB actualizada. Asegúrate de agregar esto a la KB antes del próximo paso.
  - (Aux2) Encuentra posibles ubicaciones de Pacman con la KB actualizada.
  - Llama a `agent.moveToNextState(action_t)` en la acción actual del agente en el instante t.
  - `yield known_map, possible_locations`

Para probar y depurar tu código, ejecuta:

```
python3 autograder.py -q q8
```

Nota: esta última ejecución puede resultar bastante pesada, tardando aproximadamente 3,5 minutos en completarse en un procesador medio.

Entregables

- Código Python de los ejercicios propuestos en el archivo `logicPlan.py`.
- La entrega se realizará en formato zip vía Aula Virtual el día de la modificación o a través de un repositorio alojado en el **GitLab de la EIF** si quiere optarse a los 2 puntos correspondientes a la evaluación continua de la práctica.
- Dicho repositorio debe compartirse con los profesores a través del espacio habilitado para ello en Aula Virtual, debe ser privado hasta la fecha de entrega y debe contener **todos los archivos** correspondientes a la P2.
- El código tiene que ir obligatoriamente comentado explicando su funcionalidad. Debe ser legible y estar debidamente tabulado.
- Se utilizarán sistemas anticopia y se podrá requerir explicación individual de la práctica en caso de duda.
- La entrega de la práctica se realizará el **10 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. La entrega se realizará en formato zip vía Aula Virtual.