EI1022/MT1022 - Algoritmia

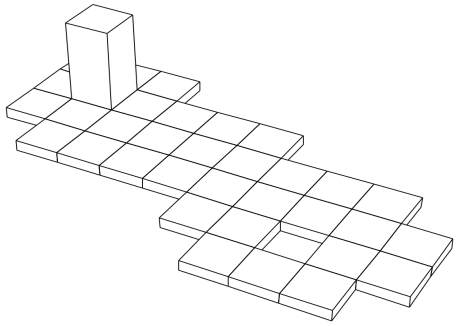
Entregable 3 - Fecha de entrega 22 de noviembre de 2018

Briker es un juego clásico de tipo puzle. Puedes ver su mecánica de juego en este vídeo:

https://www.youtube.com/watch?v=eqlyz03sx24

Diviremos el trabajo en tres partes:

- La primera parte consiste en implementar algunas funciones auxiliares que utilizaremos en las otras dos partes.
- La segunda parte consiste en, dado un puzle concreto, encontrar una solución cualquiera.
- La tercera parte consiste en, dado un puzle concreto, encontrar la solución con el menor número de pasos (una cualquiera en caso de empate).



Primera parte

En el fichero `brikerdef.py` tenemos algunas clases ya implementadas y otras con algunos métodos que debéis implementar:

- La clase `Move` es simplemente una clase con los identificadores de las cuatro decisiones (o movimientos) posibles: `Move.Left`, `Move.Right`, `Move.Up` y `Move.Down`. A cada uno se le asigna el carácter que usaremos al escribir la solución:

```
class Move:
    Left = "L"
    Right = "R"
    Up = "U"
    Down = "D"
```

- La clase `Pos2D` es una clase sencilla para representar puntos en dos dimensiones. Contiene dos métodos, `add_row` y `add_col`, para devolver un nuevo punto basado en la posición del punto original. Esta es la implementación (sólo se muestra la parte más relevante):

```
class Pos2D:
    def __init__(self, row, col):
        self.row = row
        self.col = col

    def add_row(self, d) -> "Pos2D":
        return Pos2D(self.row + d, self.col)

    def add_col(self, d) -> "Pos2D":
        return Pos2D(self.row, self.col + d)
```

- La clase `Level` permite crear un nivel a partir de un fichero de texto. El fichero de texto utiliza el siguiente código para las baldosas (`'-'`: posición inválida, `'o'`: posición válida, `'S'`: posición inicial del bloque, `'T'`: posición final del bloque). Ejemplo de fichero de texto:

```
ooo-------
oSoooo----
ooooooooo-
-ooooooooo
-----ooToo
------ooo-
```

La fila 0 columna 0 se corresponderá con el carácter de más a la izquierda de la primera fila. A continuación, se muestra la implementación de `Level`; como podéis observar, tenéis que implementar los métodos `is_valid`, `get_startpos` y `get_targetpos`:

```
class Level:
    def __init__(self, filename: str):
        self._mat = [line.strip() for line in open(filename).readlines()]
        self.rows = len(self._mat)
        self.cols = len(self._mat[0])
        self._sPos = None  # IMPLEMENTAR - Debe contener la Pos2D de la casilla con la S
        self._tPos = None  # IMPLEMENTAR - Debe devolver la Pos2D de la casilla con la T
        raise NotImplementedError

    def is_valid(self, pos: Pos2D) -> bool:
        # IMPLEMENTAR - Debe devolver False para cualquier posición fuera del tablero o
        # para posiciones marcadas con '-'. Para todos los demás casos debe devolver True.
        raise NotImplementedError

    def get_startpos(self) -> Pos2D:
        return self._sPos

    def get_targetpos(self) -> Pos2D:
        return self._tPos
```

- Por último, la clase `Block`, que representará el estado del bloque que estamos moviendo. El bloque consiste en dos cubos unidos. En esta clase debéis implementar los métodos `valid_moves` y `move`. Esta es la implementación (sólo se muestra la parte más relevante):

```
class Block:
    def __init__(self, b1: Pos2D, b2: Pos2D):
        assert isinstance(b1, Pos2D) and isinstance(b2, Pos2D)
        if b2.row < b1.row or (b2.row == b1.row and b2.col < b1.col):
            self._b1, self._b2 = b2, b1
        else:
            self._b1, self._b2 = b1, b2

    def is_standing(self) -> bool:  # true si el bloque está de pie
        return self._b1.row == self._b2.row and self._b1.col == self._b2.col

    def is_standing_at_pos(self, pos: Pos2D) -> bool:
        # Devuelve true si el bloque está de pie en la posición indicada en el parámetro
        return self.is_standing() and self._b1.row == pos.row and self._b1.col == pos.col

    def is_lying_on_a_row(self) -> bool:  # true si el bloque está tumbado en una fila
        return self._b1.row == self._b2.row and self._b1.col != self._b2.col

    def is_lying_on_a_col(self) -> bool:  # true si el bloque está tumbado en una col
        return self._b1.row != self._b2.row and self._b1.col == self._b2.col

    def valid_moves(self, is_valid_pos: Callable[[Pos2D], bool]) -> Iterable[Move]:
        # IMPLEMENTAR - Debe devolver los movimientos válidos dada la posición actual
        # Debe utilizar la funcion is_valid_pos para comprobar cada casilla
        raise NotImplementedError

    def move(self, m: Move) -> "Block":
        # IMPLEMENTAR - Debe devolver un NUEVO objeto 'Block', sin modificar el original
        raise NotImplementedError
```

Los atributos `_b1` y `_b2` contienen las coordenadas de los dos cubos que forman el bloque.

Para poder hacer su trabajo, el método `valid_moves` necesita saber si una Pos2D pertenece o no al tablero. Por este motivo recibe como parámetro la función `is_valid_pos`. Dicha función será simplemente el método `is_valid` del objeto de tipo `Level` que representa al nivel:

```
level = Level(filename)
for move in block.valid_moves(level.is_valid): ...
```

Esto permite que ambas clases estén menos acopladas.

Para saber si el bloque ha alcanzado la casilla final, podemos utilizar la función `is_standing_at_pos` pasándole la posición de la casilla final.

Aunque, por simplicidad, no aparece en el código anterior, la clase `Block` también implementa los métodos `__eq__` y `__hash__`. Esto permite utilizar objetos de esta clase como miembros de un conjunto (`example_set.add(block)`) o bien como claves en un diccionario (`example_dict[block] = 7`).

Segunda parte

Tras implementar los métodos pendientes de las clases anteriores ya podréis implementar el programa `entregable3a.py`. Se trata de un programa de línea de órdenes que recibirá como único parámetro el nombre de un fichero de texto con un puzle, buscará una solución y finalmente la mostrará por pantalla como una cadena de texto (por ejemplo: RDRULLLDDLD).

Para encontrar una solución deberéis seguir el esquema de búsqueda con retroceso, implementando una función `briker_vc_solve(level)` que reciba un objeto de tipo `Level` y devuelva una cadena de texto con la solución. El *solver* deberá crear una solución parcial inicial que utilizará para recorrer el espacio de estados por primero en profundidad. Veamos algunos consejos sobre cómo implementar la función `briker_vc_solve(level)`:

- Sobre el esquema de búsqueda con retroceso: Dado que el espacio de estados contendrá ciclos, será necesario utilizar control de visitados. Por lo tanto, podremos utilizar la función del esquema que realiza la búsqueda con control de visitados: `BactrackingVCSolver.solve()`.
- Sobre la clase para las soluciones parciales (`BrikerVC_PS`):
    - Puedes implementarla como una clase interna de la función `briker_vc_solve`.
    - Deberá cumplir la interfaz `PartialSolutionWithVisitedControl`.
    - Los objetos de la clase `BrikerVC_PS` tendrán dos atributos: una tupla con los movimientos (decisiones) que llevamos tomadas y un objeto de la clase `Block` representando la posición actual del bloque.

Tercera parte

En esta última parte vamos a implementar el programa que encuentra la solución más corta: `entregable3b.py`. Este programa tiene un funcionamiento casi idéntico a `entregable3a.py`, con la importante diferencia de que debe encontrar y mostrar la solución más corta.

Para encontrarla, deberéis seguir el esquema de búsqueda con retroceso con optimización, implementando una función `briker_opt_solve(level)` que reciba un objeto de tipo `Level` y devuelva una cadena de texto con la solución. Veamos algunos consejos sobre cómo implementar la función `briker_opt_solve(level)`:

- Sobre el esquema de búsqueda con retroceso:
    - En este caso necesitaremos utilizar la función del esquema que realiza la búsqueda con optimización: `BactrackingOptSolver.solve()`.
    - Recordad que esta función también utiliza control de visitados, por lo que los ciclos no supondrán ningún problema.
- Sobre la clase para las soluciones parciales (`BrikerOpt_PS`):
    - Puedes implementarla como una clase interna de la función `briker_opt_solve`.
    - Deberá cumplir la interfaz `PartialSolutionWithOptimization`.
    - Los objetos de la clase `BrikerOpt_PS` tendrán los mismos dos atributos que la clase `BrikerVC_PS` utilizada en la segunda parte del entregable: una tupla con los movimientos (decisiones) que llevamos tomadas y un objeto de la clase `Block` representando la posición actual del bloque.

Comprobador de soluciones y otros ficheros auxiliares

En el aula virtual tenéis disponible un fichero `e3_aux.zip` que contiene:

- `BrikerViewer.jar`: Un programa gráfico para comprobar soluciones. Se trata de un programa Java y necesita Java 1.8 para funcionar. Este programa tiene dos modos de funcionamiento:
  - Modo 'Comprobar solución': `java –jar BrikerViewer.jar level1.txt solution1.txt`. La tecla RETURN lanza una animación con los movimientos de la solución, la tecla SPACE sólo avanza un paso de la solución y la tecla ESCAPE reinicia la ejecución.
  - Modo 'Juego': `java –jar BrikerViewer.jar level1.txt`. Puedes mover el bloque con las teclas del cursor. La tecla ESCAPE mueve el bloque a su posición inicial.
- `level1.txt, solution1.txt`: Problema de ejemplo y una mejor solución (hay varias con la misma longitud).
- `level2.txt`: Otro problema de ejemplo. Su solución óptima tiene 37 movimientos.
- `bt_scheme.py`: Módulo con el esquema de búsqueda con retroceso.
- `brikerdef.py`: Módulo con las clases `Move`, `Pos2D`, `Level` y `Block`.
- `entregable3a.py`: Esqueleto del *solver* con control de visitados.
- `entregable3b.py`: Esqueleto del *solver* con optimización.

Resumen del trabajo a realizar

a) Primera parte:
   - `brikerdef.py`: Implementar los métodos que faltan en las clases `Level` y `Block`.

b) Segunda parte:
   - `entregable3a.py`: Implementar un programa de línea de órdenes que reciba un nombre de fichero con un puzle y muestre una solución por pantalla. Para encontrar la solución, deberéis implementar la función `briker_vc_solve(Level)` y su clase interna `BrikerVC_PS` para representar las soluciones parciales.

b) Tercera parte:
   - `entregable3b.py`: Implementar un programa de línea de órdenes que reciba un nombre de fichero con un puzle y muestre una solución por pantalla. Para encontrar la solución, deberéis implementar la función `briker_opt_solve(Level)` y su clase interna `BrikerOpt_PS` para representar las soluciones parciales..

c) Memoria del entregable. Una memoria en la que se detallen los pasos seguidos y dificultades experimentadas, a la vez que se presente una solución para el problema en cuestión. También debe contener el análisis del coste temporal y espacial de los métodos de la clase `BrikerOpt_PS`. Por último, debéis incluir en la memoria tres puzles de ejemplo junto con las soluciones obtenidas.

d) Actas de las reuniones mantenidas hasta la entrega. Recordad que uno de vosotros será el secretario y se encargará de tomar nota en las reuniones de trabajo. El cargo de secretario es rotativo: será una persona diferente para cada entregable.

e) Valoración personal. Cada miembro del grupo deberá escribir una breve valoración del trabajo realizado y de los resultados obtenidos.

Los apartados c, d y e se entregarán en papel, en clase, en la fecha indicada. Como respuesta a la tarea correspondiente en el aula virtual se enviará un fichero comprimido con todo (a, b, c, d y e).

Fecha de entrega de la memoria: lunes 26 de noviembre de 2018, en clase.

Fecha de entrega al aula virtual: jueves 22 de noviembre de 2018.

No os quedéis colgados. Recordad que hay tutorías.