MEMORIA III

EI1022 ALGORITMIA

Bricked

AJOPA

Adrián Enríquez

Pablo Berbel

José Ángel Molina

Índice

1. Introducción

Este entregable trata sobre la resolución de un juego conocido, *Briker* en su versión más sencilla. Éste consiste en un bloque que debe moverse desde una posición inicial a una final, a través de un tablero que tiene huecos, los cuales impiden realizar algunos movimientos según la posición del bloque en el tablero. Además, dada su forma, según se va moviendo el bloque puede estar de pie, tumbado en vertical o tumbado en horizontal. En la Figura 1 se puede apreciar el juego con un visualizador proporcionado junto al enunciado del entregable.

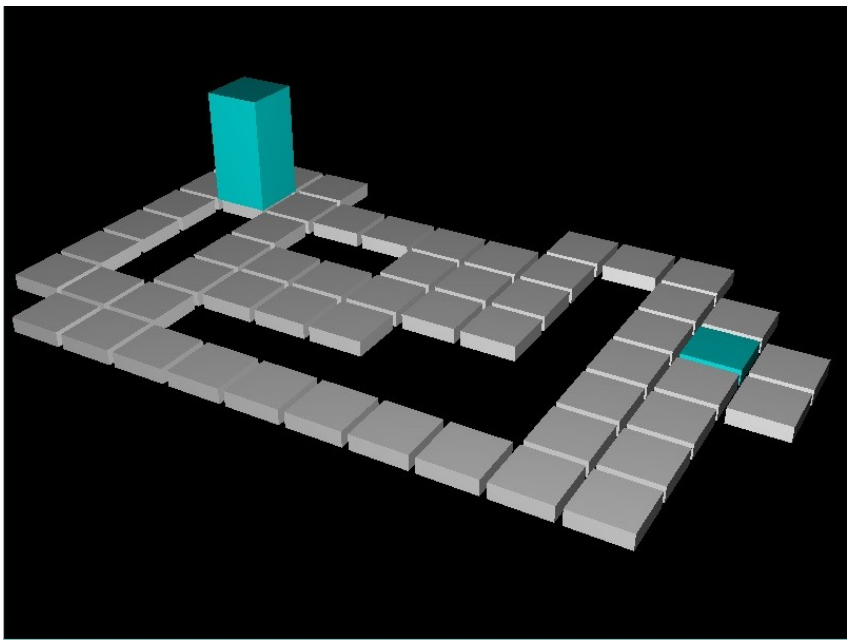

Figura 1: *Visualizador del juego Briker*

Para encontrar la secuencia de movimientos posibles que permiten resolver un determinado nivel (tablero), entra en juego el tema de la Algoritmia que se corresponde con este entregable: búsqueda con retroceso (*Backtracking*). Se trata de un método general de resolución de problemas en el cual se construye un árbol de soluciones parciales del problema, que acaba teniendo en sus hojas la solución o soluciones finales, si es que existen. A este esquema algorítmico se le puede añadir además la definición del 'estado' de una solución parcial para mantener un control de estados visitados y no generar sucesores de una solución parcial con estado ya visto, útil si se pueden producir ciclos entre las soluciones parciales. Por último puede ampliarse con la definición de una función objetivo para seleccionar la mejor solución parcial para un determinado estado, lo cual nos permitirá obtener la 'mejor' solución, según hemos definido, para el estado final.

2. Pasos seguidos en la resolución del algoritmo

Para empezar, se ha de terminar la implementación de las clases contenidas en el fichero *brikerdef.py*. Éstas clases, *Move*, *Level*, *Block* y *Pos2D* son las que representan los movimientos, tableros y bloques de nuestro juego, respectivamente, apoyándose de la última para representar casillas del tablero mediante pares fila, columna, ya que definimos el tablero como una matriz. Un bloque se define como dos *Pos2D*, ya que ocupa dos posiciones del tablero al estar tumbado, y una al estar de pie, es decir que sus dos *Pos2D* son iguales en este último caso. Otra de las partes a implementar consiste en la lectura del fichero que contiene la representación del tablero, en el constructor de *Level* para construir tal matriz. Se puede observar un ejemplo de formato de tablero tal y como lo representamos en la Figura 2.

Hay que destacar que la clase *Level* contiene un método, *isValid(Pos2D)*, que indica si dado un punto, es válido en el tablero, es decir, si no se sale de sus dimensiones o contiene el carácter '-', con el cual representamos en la matriz los huecos del tablero. Dicho método se pasa como atributo, gracias a que python es un lenguaje con características de programación funcional, a los objetos de clase *Block*, permitiendo definir en ellos el método *validMoves()*. Éste otro método comprueba los movimientos válidos para un determinado bloque, dada su posición, apoyándose en la función anteriormente descrita, que realmente está asociada a un tablero concreto, ya que comprueba si las casillas que corresponderían a la nueva posición son válidas. Gracias a *validMoves()* será posible crear las soluciones parciales sucesoras en el algoritmo de *Backtracking*, lo cual describiremos más adelante.

Para terminar con esta parte, más orientada a la definición de clases útiles para representar el problema, también ha de realizarse la implementación del método *move(Move)* de la clase *Block*, el cual devuelve un nuevo bloque desplazado según el movimiento indicado, teniendo en cuenta si está de pié, tumbado en fila o tumbado en columna.

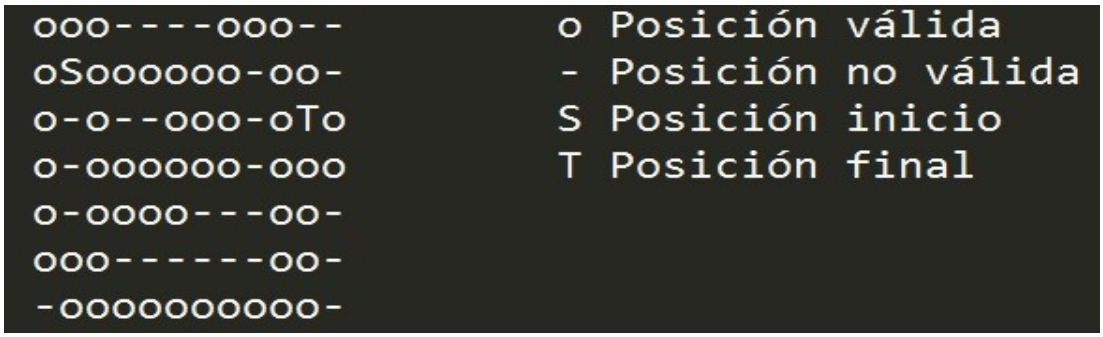

Figura 2: *Ejemplo de representación de un tablero, con leyenda explicativa*

En este punto ya se puede proceder a la creación del programa que va a encontrar la solución, secuencia de movimientos, a un determinado tablero, mediante búsqueda con retroceso. Al ser un algoritmo general, se dispone de un esquema a utilizar, el cual proporciona una clase abstracta a implementar, según la versión del algoritmo que vayamos a utilizar, y un método para encontrar la solución o soluciones a partir de una instancia de una clase que implemente a la del esquema.

Por una parte, *entregable3a.py* será un programa que dado un fichero que contenga el tablero, pasado como argumento por línea de comandos, devuelva una solución cualquiera a éste. Para ello se debe importar del esquema proporcionado la clase abstracta *PartialSolutionWithVisitedControl* y el método bt_solve_vc. La versión con control de visitados se debe a que el bloque puede llegar a una misma posición de distintas maneras y se producirían ciclos si no se tuviera en cuenta. Los atributos de la clase *Bricker_vc_PS*, solución parcial del problema, serán un bloque y una lista de decisiones, es decir, movimientos realizados hasta el momento. A continuación se implementan el resto de métodos de la clase definidos por la clase abstracta, los cuales están explicados en el código, pero en resumen, una solución parcial es solución cuando su bloque está de pie sobre la casilla final del tablero, y se define como estado de la solución al propio bloque, es decir, su posición. En cuanto a la creación de sucesores de una solución parcial, se da uno por cada movimiento válido, obtenido con *validMoves()*, el cual se añade a la lista de decisiones y mueve al bloque. Para terminar, se llama a *bt_solve_vc()* con una solución parcial inicial, es decir, con una lista de decisiones vacía y un bloque que esta de pie sobre la posición inicial del tablero.

Por último, *entregable3b.py* es la versión que no devuelve una solución cualquiera, sino la mejor, que en este caso significa con el menor número de movimientos. Para ello se utilizará el esquema del algoritmo con una función objetivo para elegir las soluciones parciales según como las puntúa dicha función. Esta vez se importa del esquema la clase *PartialSolutionWithOptimization* y el *método bt_solve_opt*. La única diferencia con respecto a la versión que devuelve cualquier solución será que la clase que representa soluciones parciales, *Briker_opt_PS*, define un nuevo método, el de la función objetivo, que tiene que devolver el tamaño de la lista de decisiones, ya que el esquema proporcionado por defecto optimiza minimizando.

Las soluciones obtenidas en nuestro caso una vez terminado el programa se pueden ver en la Figura 4 y tienen las siguientes longitudes: Para el nivel 1, 50 la solución cualquiera y 7 la óptima. Para el nivel 2, 50 la solución cualquiera y 37 la óptima. Todas han sido comprobadas con el visualizador y son correctas. En dicha figura se encuentran ejecutadas en el mismo orden mencionado antes.

```
C:\Users\Adrián\Dropbox\UJI\3º\workspace\EI1022\src\entregables\entregable3>pyth
on entregable3a.py level1.txt
<BEGIN BACKTRACKING>

La primera solución encontrada es: RRDLLULURDLLURDRRDLLLURRDRRRRDLLURDLLLURRDRRD
LULDR (longitud: 50)

<END BACKTRACKING>

C:\Users\Adrián\Dropbox\UJI\3º\workspace\EI1022\src\entregables\entregable3>pyth
on entregable3b.py level1.txt
<BEGIN BACKTRACKING>

La solución más corta es: RRDRRRD (longitud: 7)

<END BACKTRACKING>

C:\Users\Adrián\Dropbox\UJI\3º\workspace\EI1022\src\entregables\entregable3>pyth
on entregable3a.py level2.txt
<BEGIN BACKTRACKING>

La primera solución encontrada es: RRRRDLLULLLLURDDRRRULDLLLUULLDDDRDRRRRRURUULDR
ULDRU (longitud: 50)

<END BACKTRACKING>

C:\Users\Adrián\Dropbox\UJI\3º\workspace\EI1022\src\entregables\entregable3>pyth
on entregable3b.py level2.txt
<BEGIN BACKTRACKING>

La solución más corta es: RRRRDLLURDLDLLLUULLDDDRDRRRRRUURULDRU (longitud: 37)

<END BACKTRACKING>
```

Figura 4: *Salida de los programas para ambos niveles*

3. Dificultades experimentadas

Habiendo comprendido el funcionamiento del *Backtracking* y después de hacer algunos ejercicios parecidos, no ha habido grandes dificultades para implementar el condigo. El único contratiempo que podríamos mencionar fue que al principio nos daba una solución incorrecta para el nivel 2, y fue difícil encontrar el error, ya que finalmente se trataba de que el método *isValid(Pos2D)* de la clase *Level*, en *brikerdef.py*, estaba mal implementado y fallaba en algunos casos, ademas de hacer que el programa fuera lento. Dejamos esta implementación errónea al alcance del lector en la Figura 4 como dato curioso.

```
def isValid(self, pos: "Pos2D") -> "boolean":
    try:
        return self._mat[pos.row][pos.col] != '-'
    except IndexError:
        return False
```

Figura 3: *Implementación incorrecta de isValid(Pos2D)*

4. Análisis de costes

Dado que asintóticamente el coste de *entregable3a.py* es similar al de *entregable3b.py*, contando que se apoyan sobre *brikerdef.py* y el esquema de *Backtracking*, se va a realizar el análisis de costes de forma general, tratándolos a ambos como el mismo. Los resultados obtenidos serán aplicables a cualquiera de los dos.

Por una parte está el coste de leer el fichero y construir la matriz tablero, lo cual es lineal con el tamaño del tablero, pero este coste es insignificante ya que se lo come el coste de realizar una búsqueda con retroceso. A pesar de que el control de visitados lo reduce bastante, el coste de ir recorriendo las soluciones parciales posibles tiene coste exponencial con el tamaño del tablero. Esta es una de las desventajas del esquema algorítmico que se está utilizando, frente a la ventaja de ser tan general.

En conclusión, el coste resolver éste problema mediante búsqueda con retroceso es exponencial.

5. Valoraciones personales

Adrián

Una vez captado el funcionamiento de la búsqueda con retroceso, suficiente con las clases de teoría y los ejercicios realizados en las sesiones de problemas, resolver este entregable es bastante mecánico. Por otra parte es entretenido ver una aplicación de la Algoritmia enfocada a un juego, incluso se lo he enseñado a mi hermano pequeño y además de jugar un rato ha entendido desde fuera el concepto general de *Backtracking*.

José Ángel

Personalmente, esta práctica me ha gustado mucho ya que con ella he conseguido realmente entender los conceptos algorítmicos de esquema y *Backtracking*.

Creo, que al principio, a la hora de hallar la forma correcta del algoritmo, ha sido difícil *debuggear* los fallos, ya que había errores mínimos etc.

Creo que tanto el algoritmo a como b eran demasiado parecidos, por lo tanto era una tontería hacer los dos.

En general ha estado muy bien y ha servido de mucho, aunque creo que el *Sokoban* era más difícil.

Pablo

Esta práctica ha sido la más rápida de implementar, pero a la vez, la más entretenida. Gracias a las clases aportadas por el profesor, se entiende mucho mejor el funcionamiento del *Backtracking* y es mucho más fácil de programar el código, pese a la complejidad de estos algoritmos de búsqueda con retroceso. Ha sido divertido de programar, y más aún de comprobar los resultados mediante el visualizador de Java, por lo que para mí, este ha sido el mejor entregable hasta la fecha. Estoy satisfecho con el trabajo que hemos realizado y espero que sigamos así en lo poco que queda de curso.

Conjunta

Todos coincidimos en que este entregable ha sido relativamente fácil, pero adecuado y sobre todo entretenido por tratarse de un juego.

Castellón, 19 de noviembre de 2015