Memoria de la práctica 4
Algorítmica

FCO. JAVIER SÁEZ MALDONADO
LAURA GÓMEZ GARRIDO
LUIS ANTONIO ORTEGA ANDRÉS
PEDRO BONILLA NADAL
DANIEL POZO ESCALONA

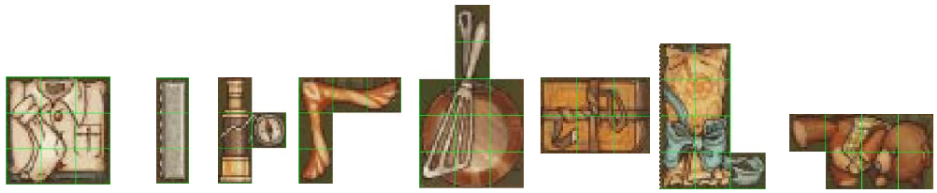
Universidad de Granada
2 de junio de 2017

Índice

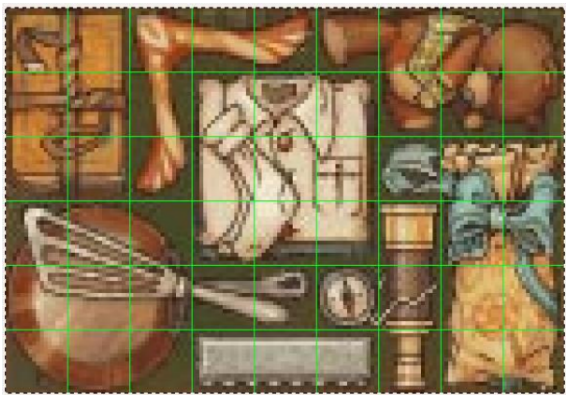
1. Problema

Referencia: El puzzle "**La Maleta de Luke**" (El Profesor Layton y la Caja de Pandora, Nintendo DS).

Diversos videojuegos de la categoría puzzle ofrecen rompecabezas que pueden ser estudiados como un problema de exploración en grafos. En particular, el puzzle "La maleta de Luke" presentado en El profesor Layton y la Caja de Pandora para Nintendo DS, supone que hay una maleta y diversos objetos que tenemos que introducir en ella sin que se solapen, cada uno de unas dimensiones determinadas. Estos problemas se pueden discretizar, de modo que en nuestro caso dividiremos la maleta en un rectángulo de 6 casillas de alto por 9 de ancho, y cada objeto también lo discretizaremos con estas mismas dimensiones:



Cada pieza, a la hora de colocarla en una posición de la maleta, puede rotarse 0^o, 90^o, 180^o o 270^o para alcanzar la solución. De este modo, el puzzle se resuelve colocando las piezas según la siguiente figura:



Se pide diseñar e implementar un algoritmo de exploración en grafos (Back-Tracking o Branch&Bound) que solucione este problema con el tamaño de maleta y piezas descritas en este apartado.

2. Solución

2.1. Elección de la técnica

Nuestro problema ha sido resuelto mediante la utilización de la técnica de resolución de algoritmos: "Backtracking", ya que hemos intentado buscar todas las soluciones posibles que existen y devolver la primera que nos aparezca, sin establecer qué solución es más optima ni explorar todo el espacio de las soluciones como hace "Branch and bound", que era la otra alternativa que teníamos para resolver nuestro problema.

2.2. Diseño de la solución

Para nuestra solución,tenemos varios elementos

- Representación: $T(x_1, \cdots, x_8)$ es un vector donde cada componente representa una columna del tablero y cada valor de x es la fila en la que se colocará la esquina superior izquierda de una pieza.
- Restricciones implícitas: x_i tendrá valores entre 0 y 5, ya que la matriz es de 6 filas.
- Restricciones explícitas: Todas las piezas deben caber dentro de la matriz sin solaparse y la matriz debe estar llena.
- Representación del árbol implícito: En cada nivel i del árbol estará la pieza
- Función objetivo: Encontrar una tupla $T(x_1, \cdots, x_8)$ tal que cada posición represente una pieza y elemento de la tupla esté la posición de la esquina superior izquierda de la pieza donde esta está colocada.
- Función de poda: Al hacer $T(x_1, \cdots, x_{k-1}) \cup x_k$ debe cumplir que la pieza x_k encaje en la posición asignada.

3. El algoritmo

3.1. Esqueleto del algoritmo

Nuestro algoritmo se puede representar mediante pseudocódigo de la siguiente manera:

```
function EncajarPiezasEnMaleta(Pieza, tablero, indice)
    if Tablero Lleno then
        return true
    end if
    for i = 1, ..., 4 do                                          ▷ Rotaciones
        for j = 0, ..., FILAS do                                  ▷ Filas
            for k = 0, ..., COLUMNAS do                           ▷ Columnas
                if No cabe Pieza[indice][Rotacion i] en posición (j,k) then
                    Pasamos a siguiente rotación o pieza
                end if
                copiaSeguridad ← tablero
                Colocamos pieza donde hemos visto que cabe
                indice++
                resuelto ← resolver(pieza, tablero, indice)       ▷ Llamada
recursiva
                if Resuelto then
                    return true
                else
                    tablero ← copiaSeguridad
                    indice − −
                end if
            end for                                               ▷ Columnas
        end for                                                   ▷ Filas
    end for                                                       ▷ Rotaciones
    return False                          ▷ Falso si no se encuentra la solución
end function
```

Se puede ver como la técnica de BackTracking se aplica en la generación mediante la llamada recursiva de un árbol de soluciones que se va explorando hasta encontrar la solución.

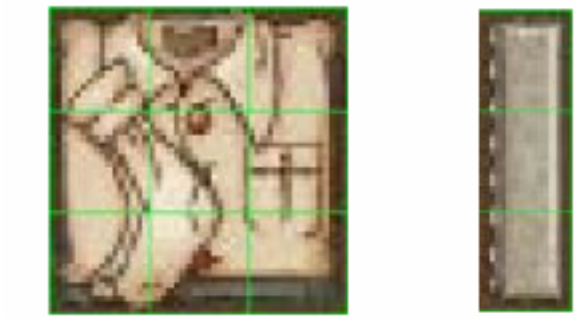
3.2. Ejemplo sencillo

Realizaremos un ejemplo muy sencillo sobre cómo funcionaría nuestro algoritmo. Pondremos las dos primeras piezas que se mostraron al principio y una matriz(un tablero) que será el siguiente:

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Donde los ceros significarán que la posición está vacía.

Si las piezas son:



se puede ver que son de 3×3 y 3×1. Para identificarlas matricialmente, la primera será, en todas sus rotaciones:

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

Y la segunda, será:

$$\begin{pmatrix} 2 \\ 2 \\ 2 \end{pmatrix}$$

en su primera rotación y

$$\begin{pmatrix} 2 & 2 & 2 \end{pmatrix}$$

en su segunda rotación.

Ahora, nuestro algoritmo empezaría y vería que la matriz inicial no está llena y por tanto comprobará si en la primera posición libre que haya, cabe la primera pieza. Como el resultado es afirmativo,la introducirá y quedará la matriz:

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

Seguidamente, intentará seguir colocando las demás piezas para ver si se puede resolver. Pasará a la segunda pieza, habiendo hecho una copia de la matriz antes de seguir por si tiene que volver. Para seguir colocando, llamará recursivamente con la siguiente pieza.

Sigue nuestro algorimo intentando ahora introducir la pieza 2 en su primera rotación mediante la llamada recursiva. El resultado de la función cabe será falso, por lo que seguirá con la siguiente rotación. En este caso, sí podrá introducirla y la introducirá, quedando la matriz(tablero) de la forma:

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \end{pmatrix}$$

que, como vemos, ya está resuelto. Aún así, volverá a llamar recursivamente sumando el índice, pero en la llamada la función llena devolverá verdadero, y así todas las llamadas recursivas devolverán verdadero y terminará la función y quedará resuelto el problema.

4. Un caso real

Un ejemplo en el que podríamos aplicar nuestro algoritmo sería en la construcción de un polideportivo. Al igual que en nuestro algoritmo, sabemos de antemano el espacio que tenemos, lo que queremos colocar dentro y cuánto ocupa cada uno. De esta forma, escribiendo en forma matricial el terreno y las pistas de los diferentes deportes que se quieran colocar, podríamos comprobar si estas caben o no en el espacio que tenemos y de qué forma cabrían.

5. Cálculo del orden de eficiencia teórica

Lo primero en esta sección es comentar es que nuestro algoritmo, debido a sus dimensiones constantes , tendrá al final un tiempo de ejecución constante y una eficiencia por tanto $O(1)$.

Ahora, intentaremos generalizar la eficiencia de nuestro problema.

Sea ahora la matriz de dimensiones $m \times n, \quad m, n \in \mathbb{N}$. Ahora, tendremos $p \in \mathbb{N}$ piezas.Nuestra función de eficiencia será:

$$T(p, n, m) = 4(T(p-1, n, m)) + mnO(cabe)$$

Con *cabe* la función que comprueba si una pieza cabe en el tablero. El orden de esta función es $m * n$. Tenemos que tener en consideración la condición de parada de T que sería Si definimos la función:

$$T(1, n, m) = nm$$

Si quisiéramos establecer un orden de eficiencia en notación O para nuestro algoritmo en el caso general, si desarrollamos en nuestra ecuación obtendríamos que es igual a:

$$4^{p-1}T(1, n, m) + (4(p-1)+1)mnO(cabe) = 4^p mn + (4p+1)mnO(cabe)$$

Lo cual, en notación O es

$$O(pmn \cdot O(cabe))$$

Y sabiendo $O(cabe)$, tenemos que eso es igual a:

$$O(pm^2n^2)$$

6. Instrucciones de compilación

Para compilar nuestro código, en la carpeta donde se tengan los archivos, abrir la terminal y ejecutar *make* para que se genere el ejecutable. Este ejecutable estará en la carpeta Ejecutables correspondiente.