

# **ESTRUCTURAS DE DATOS Y ALGORITMOS**

## **TPE**

Bartolomé, Francisco  
Vitali, José Angel  
Dantur, Juan Pablo

**PRIMER CUATRIMESTRE**

**02/06/2015**

**COMISIÓN S**



## **1. Introducción**

El objetivo del trabajo práctico es desarrollar una aplicación que resuelva niveles de una variante del juego Pipe Dream, obteniendo soluciones exactas y aproximadas. Este informe explicará y discutirá las decisiones tomadas a lo largo del trabajo, los algoritmos y estructuras utilizados y descartados, así como comparaciones de tiempos de ejecución para justificar las mismas.

## 2. Algoritmos utilizados

### 2.1 Solución exacta

Para encontrar la solución mas larga, se utilizó una variante del algoritmo BackTracking. Este algoritmo consiste en crear un arbol de posibilidades, y hacer un recorrido DFS, ya que se quiere encontrar la solución de mayor longitud, es decir, el camino más largo desde la raíz (el tablero inicial), hasta una hoja válida (un tablero resuelto). De reconocer que un camino no es válido se vuelve atrás y se exploran otras opciones. Si se pidiera la solución más corta, el recorrido debería ser por niveles, BFS, ya que en este caso la primer solución encontrada sería la más corta.

Se propusieron ciertas podas para acortar el tiempo de ejecución. Algunas se utilizaron y otras no. En primera instancia se propuso una poda que genera una cota superior para la longitud máxima del camino encontrado, de manera que si se encuentra un camino con esta longitud, el algoritmo corta la ejecución, ya que cualquier otra solución va a ser de longitud menor o igual a la encontrada. Si bien ésto solo ayuda en los casos en que la cota sea igual a la máxima longitud posible, se decidió usarla, ya que se utiliza solo una vez y no es temporalmente costosa.

Archivo	Con poda	Sin poda
prueba1.txt	29 milisegundos	120 milisegundos
prueba2.txt	1270 milisegundos	1912 milisegundos
prueba3.txt	5 milisegundos	16 milisegundos
Prueba4.txt	110578 milisegundos	241796 milisegundos

En todos estos casos, la poda calcula exactamente la longitud del maximo camino posible. Se observa que el tiempo se reduce considerablemente con la poda implementada.

Otra poda propuesta calcula la posibilidad de llegar a una solucion desde un camino incompleto cualquiera, es decir si a partir de ese punto existe una solucion. Dicho método es de complejidad temporal  $O(n)$  siendo  $n$  la cantidad de casilleros del tablero.

Pseudocodigo:

```
pathExists: {tablero, posicionActual, set} -> {boolean}{
  IF(posicionActual fuera de tablero)
    RETURN true;

  set.put(posicionActual);

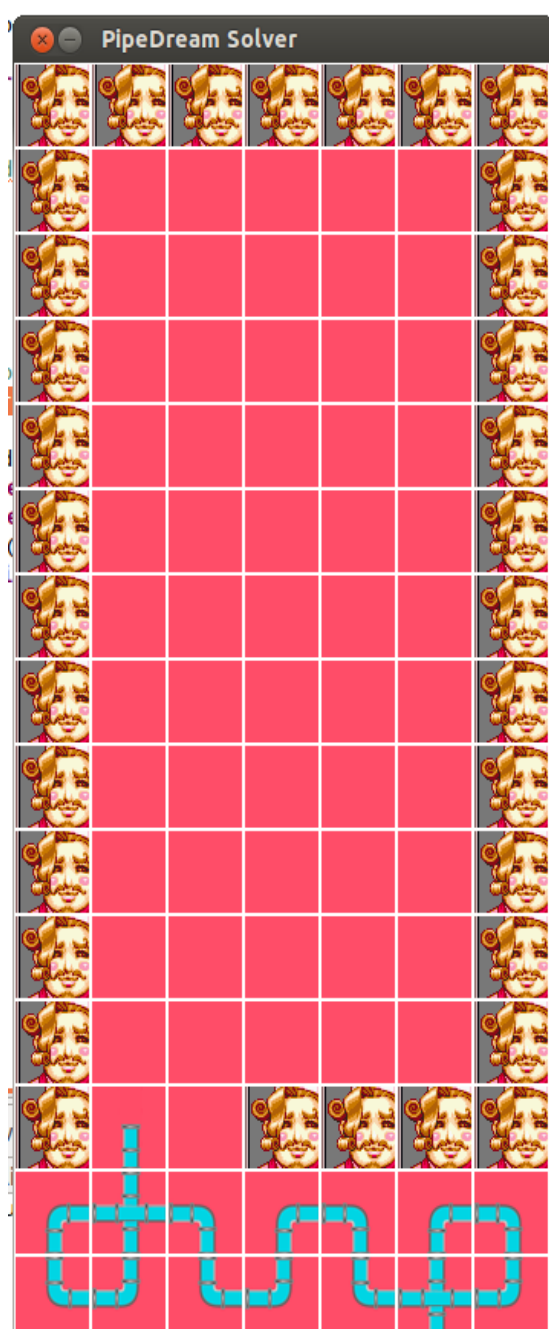
  IF(set.contains(posicionActual) || posicionActual es pared)
    RETURN false;

  RETURN pathExists de los casilleros aledaños;
}
```

Si bien aporta mucho en determinadas situaciones en las que se llega a un camino parcial que nunca tendra solución evitando cálculos innecesarios, al utilizarse en cada llamado recursivo es muy costosa en la mayoría de los casos. Por este motivo, se decidió prescindir de esta poda.

Archivo	Sin Poda	Con Poda
EjemploEncrucijada.txt	+15 min	1532 milisegundos
Ejemplo5x5.txt	5616 milisegundos	25381 milisegundos
Ejemplo4x4.txt	124 milisegundos	1068 milisegundos
Prueba4.txt	156443 milisegundos	+15 min

Como puede apreciarse en la tabla, en la mayoría de los casos la poda es demasiado costosa. Sin embargo, existen casos puntuales en los que la poda reduce el tiempo de ejecución en gran medida.



En la imagen anterior se ve el tablero resuelto del archivo ejemploEncrucijada.txt en el que se observa que si por algún motivo la tubería se dirigiese hacia la parte superior del tablero no se encontraría una solución, analizando un subtablero (en este caso) de 11x5 innecesariamente. Este análisis puede ser evitado con la poda propuesta.

```
exactSolverRec: {tablero, longitudMax, mejorsolucion} -> {boolean} {  
  IF(tablero.posicionActual fuera del tablero){  
    IF(tablero.longitud > mejorsolucion.longitud)  
      mejorsolucion = tablero;  
    RETURN true;  
  }  
  IF(tablero.posicionActual es un caño doble){  
    tablero.posicionActual = la siguiente del caño doble;  
    RETURN exactSolverRec;  
  }  
  flag = false;  
  FOR c -> {caños compatibles}{  
    Cambiar estado;  
    ret = exactSolverRec;  
    IF(ret){  
      IF(tablero.longitud == longitudMax)  
        RETURN true;  
      flag = true;  
    }  
    Volver al estado anterior;  
  }  
  RETURN flag;  
}
```

## 2.2 Solución aproximada

En este caso el objetivo es hallar una solución que no sea necesariamente la máxima, pero se aproxime lo más posible. Para realizar esta tarea se implementó un algoritmo que utiliza la técnica de Hill Climbing.

La misma consiste en tomar una solución y mejorarla buscando soluciones vecinas hasta llegar a un máximo local, repitiendo esto durante un intervalo determinado de tiempo. Al terminar este intervalo, la mejor solución encontrada hasta el momento será la respuesta dada por el programa.

Heurísticas:

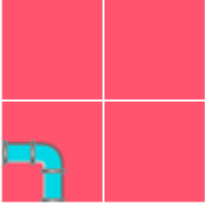
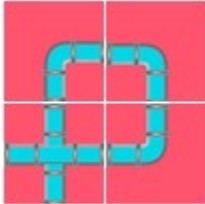

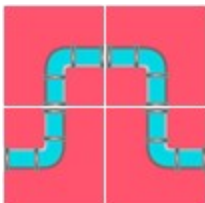
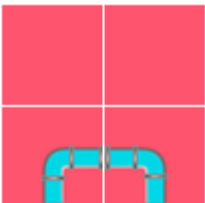
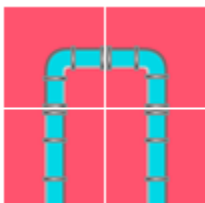
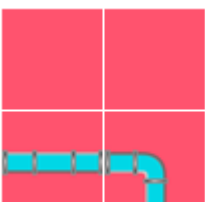
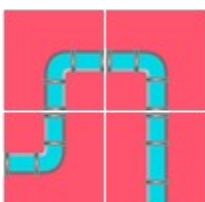
Para definir un vecino se decidió recorrer el tablero dividiéndolo en submatrices de 2x2 y mejorando los subcaminos contenidos en las mismas. Esto se hizo aprovechando la interfaz HashMap de la API de Java y una clase propia llamada SubMatrix. Definimos un vecino como un tablero solución que difiere de otro en una submatriz. Por la forma de definir un vecino, todos los vecinos de un tablero serán una mejor solución. De esta manera se puede concluir que se ha llegado a un máximo local cuando un tablero solución no tiene más vecinos. Se propuso que al encontrar ciertas submatrices se las cambie si es posible por otra determinada submatriz de mayor longitud:

```
hillSolver{tablero, tiempo} -> {matriz} {  
  instancio mejorSol;  
  WHILE(tenga tiempo){  
    IF(getRandomSolution(tablero)){  
      aux = getMaxLocal(tablero, tiempo que queda);  
      mejorSol = mejorSol.longitud > aux.longitud ? mejorSol : aux;  
    }  
  }  
  RETURN mejorSol;  
}
```

```

flag2=true
flag4=true
Create possibleNeighbors2 map
Create possibleNeighbors4 map
subm=null
while (flag2 or flag4)
    if (time is up)
        return board
    end
    flag2=false
    flag4=true
    while (flag4)
        flag4=false
        for i=0 to Rows
            for j=0 to Columns
                if (time is up)
                    return board
                end
                subm=board[i,j],[i+1,j],[i,j+1],[i+1,j+1]
                if (possibleNeighbors4 has subm)
                    change board
                    if (board was changed)
                        flag4=true
                    end
                end
            end
        end
    end
    end
    for i=0 to Rows and !flag2
        for j=0 to Columns and !flag2
            if (time is up)
                return board
            end
            subm=board[i,j],[i+1,j],[i,j+1],[i+1,j+1]
            if (possibleNeighbors4 has subm)
                change board
                if (board was changed)
                    flag2=true
                end
            end
        end
    end
end
return board

```

SubMatriz		Solución
	Grupo 1	
	Grupo 2	
	Grupo 2	
	Grupo 2	

(aquí se muestran solo algunas submatrices, además se utilizaron variantes de las mismas, que constan en rotarlas y espejarlas)

Nótese que las primeras relaciones mejoran la longitud del camino en cuatro unidades (grupo 1), el resto la mejoran en dos (grupo 2). De esta manera se puede discernir antes del análisis del tablero cuales podran ser los mejores vecinos. De esta manera no necesariamente hace falta reconocer todos los vecinos de un tablero para saber si un vecino es el mejor. Para esto se utilizaron dos `HashMap<SubMatrix, SubMatrix>`, uno que contiene las relaciones que mejoran el tablero en cuatro unidades y otro que contiene las que lo mejoran en dos, aprovechando que el método `get()` del `HashMap` tiene una complejidad de  $O(1)$ . Se pensó recorrer el tablero buscando coincidencias del grupo 1 hasta no encontrar coincidencias, luego recorrer buscando alguna coincidencia del grupo 2. Al encontrar alguna coincidencia del grupo uno, inmediatamente se vuelve a buscar coincidencias del primer grupo. Si tampoco se encuentran coincidencias del grupo 2, entonces se llegó a un maximo local.



### 3. Estructuras de datos utilizadas

Estructuras utilizadas en las dos variantes para resolver el problema:

Para representar a los caños se utilizó una clase abstracta Pipe, representando cada caño específico con una clase que extienda de Pipe que contiene los extremos del caño, una entrada y salida. Se consideró a una pared como un tipo de caño. Cada Pipe tiene un método que devuelve una nueva instancia del determinado Pipe. Además, se implementó el hashCode() que devuelve un numero constante para cada tipo de distinto de Pipe. Se decidió que el equals() se base en el valor de retorno de la igualdad entre los hashCode().

Para representar al tablero se hizo uso de una matriz de Pipes, siendo null las posiciones vacías de la matriz. Los tipos y cantidad de caños a disposición para resolver el problema se guardan en un HashMap<Pipe, Integer>, donde el Pipe es una instancia del tipo de caño y el Integer es la cantidad de caños de ese tipo que hay disponibles. Estas estructuras se encuentran dentro de la clase Board, que también almacena la longitud de la solución del tablero y las coordenadas del caño inicial.

La mejor solución encontrada hasta el momento durante la ejecución del programa se guarda en la clase Matrix, que tiene una matriz de Pipes, la longitud del camino y el tiempo que transcurrió hasta encontrar el mismo.

Estructura utilizada para resolver con Hill Climbing:

Para representar una submatriz de 2x2 se utilizó la clase SubMatrix, la cual contiene una matriz de Pipes de dimension constante 2x2, y un HashMap<Pipe, Integer> semejante al que se encuentra en Board, solo que este describe el tipo y la cantidad de caños que hay en la submatriz representada. Como se necesitaba distinguir cada tipo de SubMatrix, se implementó los métodos hashCode() y equals(). El primero se basó en la concatenación de los hashCode() de cada Pipe que conforma la submatriz o 0 en caso de que la posicion requerida sea nula, intercalando cada posición con un valor constante dterminado que no refiere a ningún tipo de Pipe relevante, con motivo de que el hashCode() de cada submatriz quede univocamente determinado. Decidimos que el equals() se base en el valor de retorno de la igualdad entre los hashCode().

Se tiene además, dos HashMap<SubMatrix, SubMatrix> que almacenan las relaciones entre submatrices que se pueden mejorar y su repectiva mejora. Se utilizan dos mapas ya que uno alberga los tipos de submatrices que de ser optimizadas generan un incremento en la longitud del camino en dos unidades y el otro los tipos de submatrices que al ser optimizadas generan un incremento de cuatro unidades.

#### **4.Posibles mejoras al Trabajo Práctico**

Si bien ambos algoritmos funcionan de manera correcta, existen posibles mejoras que se pudiesen haber hecho con motivo de mejorar la eficiencia de estos.

En el algoritmo que calcula la solución exacta, ambas podas podrían haber sido mejoradas. El método `maxPossibleLength()` calcula una cota superior para la longitud de la respuesta, por lo que sólo resultará de utilidad en el caso que dicha cota coincida exactamente con la máxima longitud. La misma está únicamente basada en la cantidad de caños disponibles, por lo tanto, solo acertará la máxima longitud en el caso en que la cantidad de caños disponibles sea menor a la cantidad de casilleros inicialmente vacíos. Se podría haber buscado otra manera de calcularla que también se base en los casilleros vacíos, es decir que el máximo camino posible también podría ser igual a la cantidad de casilleros vacíos (mas el caño inicial) sumada a la cantidad de casilleros en los que se puede completar un loop. De esta manera podría calcular de dos maneras distintas esta cota, y retornar la menor de las dos. Así incrementan los casos en los que el método calcula una cota mayor equivalente a la máxima longitud posible y optimizando el algoritmo en más casos.

Respecto a la poda del método `pathExists()` (poda que finalmente se decidió no usar), se podría haber encontrado alguna forma de decidir en qué casos es conveniente usarla, y en qué casos no. Si bien por experiencia descubrimos que los tableros con mayor cantidad de casilleros o con mayor porcentaje de paredes suelen tener menor diferencia en el tiempo en que tarda la ejecución del algoritmo, en todos los casos probados concluimos que es mejor prescindir de la poda. A pesar de esto, se cree que existe una relación entre la cantidad de casilleros o paredes y la decisión de hacer uso de la poda, pero para esto se deberían probar ejemplos de tableros con dimensiones mucho mayores, a pesar que el tiempo de ejecución de estos sería muy alto.

En el algoritmo que calcula una solución aproximada, se cree que agregar más casos de vecindad al `HashMap` en el que están establecidos, hubiese optimizado la búsqueda de un máximo local.

## **5. Conclusión:**

En el algoritmo exacto se observó que si bien el mismo encuentra soluciones exactas en un tiempo aceptable para tableros pequeños, es muy costoso a la hora de resolver un tablero de una dimensión importante. Es en estos casos el método que utiliza Hill Climbing muestra su utilidad, ya que en un tiempo acotado encuentra una solución aceptable.

A la hora de implementar el algoritmo de Backtracking no hubo mayores complicaciones, ya que no fue difícil determinar cuándo un caso es solución o cuándo hay que volver hacia atrás y explorar otras posibilidades. Además se aprendió la importancia de la implementación de podas, ya que se encontraron podas útiles y otras que a pesar de parecer muy útiles, el tiempo que demora su ejecución no favorece al algoritmo.

Por otro lado, al implementar el Hill Climbing hubo que dedicar mucho tiempo a la hora de decidir qué se consideraba como un vecino. Finalmente se encontró una definición que además de satisfacer lo necesitado para Hill Climbing permitió reducir los tiempos al poder decidir que un vecino era el mejor sin evaluar todos los vecinos de una solución.