Trabajo Práctico Especial

Estructuras de Datos y Algoritmos

2do cuatrimestre, 2016

Fernán Oviedo

Axel Fratoni

Gastón Rodríguez

Martín Victory

Segundo Fariña

Contenido

Introducción

El objetivo del presente trabajo práctico fue desarrollar un programa que resolviera diferentes tableros del juego “Flow”. El juego se desarrolla de la siguiente manera: inicialmente se da una grilla de con pares de puntos del mismo color y espacios vacíos. El objetivo es encontrar una serie de caminos que unan los puntos de los mismos colores entre sí (exactamente un camino por color) de forma tal que los caminos no se crucen y quede cubierta la mayor cantidad de casilleros posibles de la grilla. A continuación, se muestra un estado inicial y un estado final de la grilla.

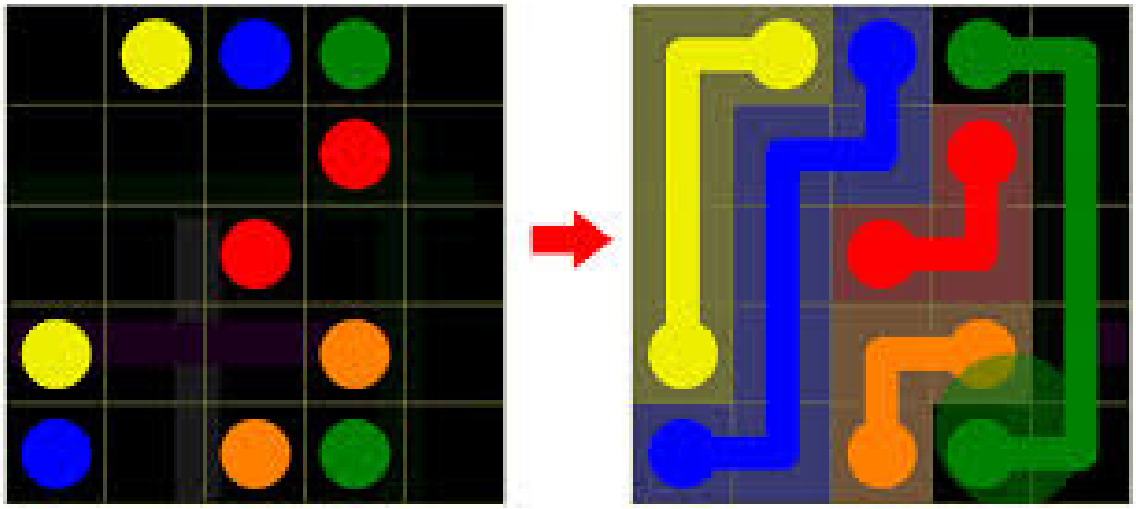


Figura 1: juego de Flow sin resolver y resuelto (la imagen es cortesía de StackOverflow)

El programa debía disponer de dos algoritmos: uno que encontrara la solución exacta en el tiempo que le fuera necesario; y otro que buscara una solución aproximada en una determinada cantidad de tiempo pasada por parámetro. Éste último debía ser un algoritmo del tipo *hill climbing*.

El programa debería estar implementado en Java y debería leer el archivo de entrada y los parámetros según el formato especificado por la cátedra, así como mostrar la solución en una ventana gráfica.

Estructura general

El diseño del programa se basa enel patrón MVP. En éste, se presentan tres componentes fundamentales:

- El **modelo** que contiene la representación interna del juego
- La **vista** que se ocupa de leer los parámetros y el archivo y de mostrar la solución en una ventana gráfica
- El **presentador** que maneja la resolución del juego y decide cuándo mostrar algo por pantalla

Esta división está representada por la jerarquía de paquetes en la que se encuentra seccionado el trabajo: *frontEnd*, *controller*, y *backEnd*.

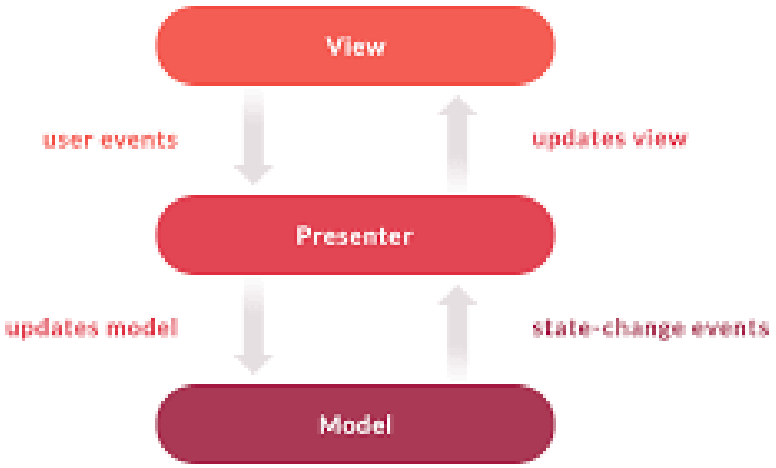


Figura 2: MVP (la imagen es cortesía de Macoscope)

Es importante destacar que la salida estándar (en general, la consola) no es administrada exclusivamente por el *frontEnd*. Esto tiene que ver con que no consideramos a la salida estándar como el *output* con el que interactúa el usuario, sino como un *log* de estados del programa e información adicional, que bien podría ser un archivo si se quisiera. Lo que el usuario *ve* es la pantalla gráfica. Debido a esto, cualquiera de las tres partes del programa puede imprimir por salida estándar.

Solución exacta

El algoritmo de solución exacta aplica fuerza bruta con podas. La idea es que, a partir de un estado, se generan todos los estados siguientes y se encolan en una cola de estados. Luegose desencola cada uno de ellos para ver si es solución. Si no es solución, se encolan todos sus siguientes estados y se sigue desencolando estados sucesivamente hasta encontrar una solución o hasta vaciar la cola. De esta forma, si pensamos en un *game-tree*, lo que se está haciendo es generarlo y recorrerlo en modalidad BFS. Quien resuelve el juego es un objeto de la clase *ExactSolver* del *middleEnd*.

Para la solución exacta, contamos con diversas implementaciones de estados en el *backEnd*. Cada una tiene diferentes formas internas de representar un estado y diferentes nociones de lo que son sus siguientes estados. Sin embargo, todas respetan un contrato general, por lo que el *controller* no debe preocuparse por la representación interna de éstos.

Como el contrato general de los estados no implica que no puedan aparecer estados repetidos, el *ExactSolver* debe mantener registro de qué estados han sido visitados. Esto se hace para no repetir ramas del *game-tree* que ya estén siendo analizadas por otra parte. Esto es muy importante, ya que el árbol crece exponencialmente. Esta poda es la única a cargo del *ExactSolver*. Todas las demás están a cargo del *backEnd*.

En nuestros tipos de estados, tenemos dos formas de representar un tablero. Una es usando una matriz de objetos; la otra es usando una matriz de bytes. La representación como matriz de objetos se manipula con objetos de tipo *Square*. Los miembros de esta clase son: un entero que representa el color, un elemento que representa el elemento del casillero (que puede ser una línea, un punto o nada) y dos direcciones que indican hacia dónde apunta el elemento. Tanto los elementos como las direcciones son tipos enumerados, por lo que son instancias fijas de clases y no hace falta crearlas y destruirlas, sino sólo referenciarlas. Por otra parte, la representación como matriz de bytes es más complicada. En cada byte se almacena los datos de un casillero. En los cuatro bits superiores se almacena el color; en los cuatro bits inferiores, se

almacena el tipo de elemento que se encuentra allí. La siguiente tabla muestra la relación entre los cuatro bits menos significativos y el elemento que representan:

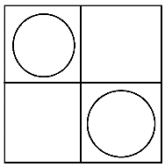
4 bits menos significativos	Elemento representado
0000	Lugar vacío
0001	Semilínea hacia arriba
0010	Semilínea hacia la derecha
0100	Semilínea hacia abajo
1000	Semilínea hacia la izquierda
0011	Línea arriba-derecha
0110	Línea abajo-derecha
1100	Línea abajo-izquierda
1001	Línea arriba-izquierda
0101	Línea arriba-abajo
1010	Línea izquierda-derecha
1110	Punto con semilínea hacia arriba
1101	Punto con semilínea hacia la derecha
1011	Punto con semilínea hacia abajo
0111	Punto con semilínea hacia la izquierda
1111	Punto

Tabla 1: relación entre cuatro bits inferiores de un byte y su elemento representado

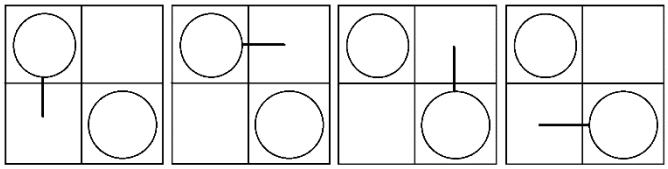
La ventaja que tiene este método es que la lógica de poner o sacar líneas se hace a través de la operación *xor*, la cual es realizada rápidamente por un procesador (es una operación de la *ALU*) y es transparente a lo que ya hay en el casillero. Además, al utilizar bytes en lugar de objetos se utiliza mucha menos memoria.

Por cada uno de este tipo de representación, tenemos dos tipos de implementaciones. Las dos que hicimos en un primer momento (*ByteState* y *SquareState*) tenían como noción de estado siguiente aquél que implicara agregar una línea entre dos casilleros desde algún casillero que ya estuviera ocupado hacia uno que estuviera inmediatamente al lado (es decir, no una línea rodeada de casilleros vacíos) y aceptara un agregado de línea. A continuación, se muestran los estados siguientes de un estado:

Estado inicial:

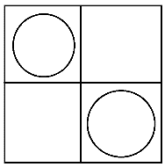


Estados siguientes:

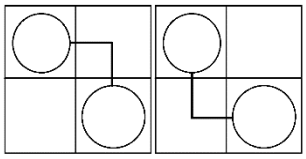


Ambas implementaciones eran totalmente funcionales. Sin embargo, resolver un tablero de 5x5 llevaba alrededor de 13 minutos. Como nos parecía demasiado pobre, decidimos hacer una implementación nueva. Así surgieron dos nuevas implementaciones: *NewByteState* y *NewSquareState*. Cada una de ellas consta de la misma representación interna de un tablero, pero tienen una forma distinta de entender y generar siguientes estados. En este caso, un estado siguiente es aquél que conecte un par de puntos más que el anterior.

Estado inicial:



Estados siguientes:



Además de cambiar la noción de “estado siguiente”, se agregaron varias optimizaciones. Por un lado, si al estar generando estados siguientes se encuentra una condición por la cual se pueda haber aislado dos puntos del mismo color de forma tal que ya no se puedan conectar, se realiza un chequeo para ver si efectivamente se han aislado puntos. De ser así, se corta con la

búsqueda desde ese estado. Otra optimización que se hizo fue que en la generación de estados se chequee a su vez si el nuevo estado generado es la solución óptima. Si lo es, entonces el *ExactSolver* aborta su resolución lanzando una *BestPathPossibleAlert* que contiene la mejor solución. Esta alerta es una clase que extiende de *Throwable* y la usamos tanto para cortar la recursión de la búsqueda de los estados siguientes como para evitarle al *ExactSolver* revisar todos los estados que tenía encolados. Por último, una gran optimización que se implementó fue hacer a los estados inmutables. Para ello, se utilizó el *BuilderPattern*. Con este patrón, los estados no se construyen y luego se "settean", sino que se les pide un *builder* sobre el cual se arma el estado que se desea y luego se le pide a éste que lo construya. La gran ventaja que tiene esto es que los estados son inmutables. Esto por un lado permite asegurar invariantes muy fuertes desde la construcción. Pero lo más importante es que como no cambian, tampoco lo hacen sus *hash codes*. Esta cualidad permite que el *hashCode* sólo se calcule una vez; como optamos por la modalidad *lazy*, esto se hace la primera vez que se necesita. Esta optimización influye en el uso intensivo de *HashSets*que hacen tanto el *backEnd* como el *controller*.

Nota aclaratoria

La representación *NewByteState* no acepta tableros de más de 255 filas ni columnas. Esto tiene que ver con un tratamiento de la dimensión del tablero que se realiza con bytes. A fines prácticos, no supone ningún desmedro de la implementación.

Por otro lado, vale mencionar que la implementación *SquareState* quedó deprecada y no debe ser utilizada.

Solución aproximada

Para obtener una solución aproximada implementamos un algoritmo basado en *back tracking* y *hill climbing*, usando el primer método para obtener una solución (no necesariamente exacta) lo más rápido posible y el segundo para optimizarla.

El análisis del tablero para la primera parte del algoritmo se basa en solo darle importancia a lo que denominamos *Ends of Tracks (EOT)*, siendo en un principio los puntos del tablero y luego el casillero que contiene el final de un camino que sale de un punto y aún no se ha unido con su par. En cada paso avanza un solo *EOT* una sola posición y esto puede pasar por dos motivos: era un movimiento implícito o se tomó una decisión.

Para saber qué acción tomar se hacen análisis de “grados de libertad”, que consiste en ver hacia cuantos casilleros se puede mover un *EOT* en un solo paso sin cortar ningún otro camino y sin entrar en una zona desde la cual no pueda alcanzar a su par. Este número puede variar entre 1 y 4. Si se tiene en el tablero *EOT* s de grado 1 significa que están obligados a moverse en esa dirección en lo que llamamos movimientos implícitos. Al ser esta la acción más rápida del algoritmo, cuando se resuelven tableros con muchos puntos y muy juntos es probable que se llegue a una solución exacta en un tiempo considerablemente bajo.

Cuando no hay movimientos implícitos para realizar, se debe tomar una decisión. El primer paso para hacerlo es buscar el *EOT* más apto para ser movido, siendo según nuestro criterio aquel que esté a solo un paso de unirse con su par. Si se da esta condición se avanza en la dirección más cercana a él. Si no, se toman todos los *EOT* que tengan el menor grado de libertad del tablero y se utiliza una función aleatoria para elegir el siguiente a mover y en qué dirección hacerlo.

El siguiente paso de la toma de decisiones consta de una pila y un *HashSet* de estados, siendo un estado una copia del tablero en un momento dado. Cuando ya se ha elegido al *EOT* que será movido, se debe apilar el

estado del tablero previo al movimiento para que, si luego de varios movimientos posteriores se llega a la conclusión de que se ha tomado una decisión errónea, sea posible acudir a la pila y continuar desde ese estado tomando una decisión diferente. En el *HashSet* se almacenan los estados luego de realizar el movimiento de la decisión, de forma tal que al momento de desapilar un estado, se tenga conocimiento de las decisiones tomadas anteriormente.

El ciclo del algoritmo consiste en realizar un análisis de movimientos implícitos cada vez que se toma una decisión ya que esta puede afectar los grados de libertad de otros *EOT* haciéndolos disminuir. Luego, en caso de no tener decisiones posibles por tomar se des apila un estado anterior y se repite hasta alcanzar una solución o que la pila de decisiones quede vacía y no se la pueda cargar más debido a que se sabe que cualquier decisión que se pueda tomar es inviable.

Se considera una solución a aquel tablero que no contenga *EOT*s. Cuando se llega a una, comienza el proceso de *hill climbing* donde el interés reside en los tramos de los caminos que sean rectos a lo largo de dos casilleros y sean aledaños a dos casilleros vacíos. La presencia de estos tramos implica que el camino puede ser modificado de la siguiente forma:

La idea del algoritmo es encontrar todos estos tramos y realizarles esta operación haciendo que el tablero se llene progresivamente. Una vez que el tablero se llena lo máximo posible, se lo guarda como una potencial solución a presentar gráficamente y se vuelve al proceso de *back tracking* intentando llegar a una solución aproximada distinta e intentando que el proceso de *hill climbing* devuelva un tablero con menos espacios vacíos, evitando así los máximos locales.

Tabla comparativa

A continuación, se detalla una tabla con diferentes tiempos de ejecución (todos en segundos) para diversas implementaciones de solución.

	3x3	5x5	8x8	8x8 bis	9x9	10x10
ByteState	0,077	14,94	…	…	…	…
NewByteState	0,071	0,12	…	…	…	…
NewByteState + poda	0,073	0,095	27,772	…	3,508	…
NewByteState + poda + hashCode	0,079	0,089	0,869	…	3,131	…
NewSquareState	0,148	0,208	…	…	…	…
NewSquareState + poda	0,158	0,188	10,595	…	12,816	…
NewSquareState + hashCode	0,163	0,196	6,753	…	11,969	…

Tabla 2: comparación de tiempos de ejecución para diferentes tableros y distintas implementaciones

"Poda" refiere a la verificación de que no se ha imposibilitado la conexión entre dos nodos. "hashCode" refiere a la optimización de calcularlo una sola vez. Todas las pruebas aquí utilizadas se encuentran en el directorio "Pruebas/" del proyecto. Los puntos suspensivos en la tabla significan que no obtuvimos una respuesta del algoritmo en un tiempo razonable.

No realizamos una tabla comparativa de tiempos con el algoritmo de solución aproximada por dos motivos. En primer lugar, nuestro algoritmo tiene un componente aleatorio, lo cual hace que el tiempo de resolución de un tablero pueda variar drásticamente entre ejecución y ejecución (en algunos casos, el tiempo se multiplicaba por 100 de una resolución a otra). Por otro lado, el algoritmo está pensado para encontrar la mejor solución posible en un tiempo determinado, por lo que tanto la solución encontrada como el tiempo tardado dependerán del tiempo que se le dé para correr.

Utilización del programa

Para poder compilar el proyecto, se proporciona un *Buildfile* de *Ant* (*build.xml*). Para poder utilizarlo, es necesario cambiar el valor de JAVA_HOME por del directorio en el que esté instalado el JDK. Para ello, se debe editar la línea 6, cambiando el valor del atributo 'value' por la ruta del JDK (es decir, en este directorio deben estar '/bin/javac' y '/bin/java'). El directorio que se encuentra en la entrega es el necesario para correrlo en Pampero. Para compilarlo, alcanza con ejecutar el comando *ant* desde la consola. Una vez compilado, se genera en el mismo directorio un archivo *tpe.jar*, el cual se puede correr como se indica en la consigna del trabajo. Sin embargo, para correrlo en Pampero, se debe hacer de la siguiente forma:

/usr/lib/jvm/java-8-jdk/bin/java -tpe.jar <archivo> <modo> <tiempo>

Esto se debe a que el JDK por defecto es el *open-jdk*, que no viene con las librerías de JavaFX.

Cabe aclarar que en el caso en el que se ejecute con la modalidad *progress*, el paso a paso del algoritmo no se verá en la ventana gráfica, sino que se verá en la consola. Sólo se mostrará la solución final en la ventana gráfica. Esto se debe a que no pudimos hacer que el algoritmo imprimiera de forma gráfica durante la búsqueda de la solución por un problema de *threads* de JavaFX. La solución era crear otro *thread* aparte del de UI para el procesamiento. Sin embargo, se nos desaconsejó desde la cátedra que hiciéramos procesamiento multihilo, por lo que optamos por dejarlo así.

Otra cuestión que surge al pedir progreso es que a veces el controlador demora en comenzar a imprimir. Esto tiene que ver con que el procesamiento de los siguientes estados puede llevar mucho tiempo y el *backEnd* no devuelve los estados hasta que el conjunto de siguientes estados esté completo. La solución a este problema involucraba nuevamente *multithreading*: el *backEnd* y el *controller* debían compartir una cola en la cual el *backEnd* dejaría cada estado nuevo y el *controller* los consumiría. La complejidad de esto nos desalentó la idea de implementarlo.

Conclusiones

Finalmente, la mejor solución exacta que conseguimos fue aquélla que no sólo utilizaba un algoritmo más eficiente, sino también optimizaba el uso de recursos. Ésta fue *NewByteState*. Es notable la diferencia de tiempos que presenta con NewSquareState, siendo que esta última es una implementación diferente del mismo algoritmo.

Por el lado de la solución aproximada, no entregamos muchas implementaciones, ya que se trabajó siempre sobre la misma. Sin embargo, ésta fue suficiente para bajar muchísimo el tiempo de resolución respecto de la mejor solución exacta.