TRABAJO PRÁCTICO ESPECIAL

Solucionador del juego Flow Game de forma óptima y aproximada

Estructura de datos y algoritmos
Segundo Cuatrimestre, Noviembre 2016

Martín Biagini
Joaquín Filipic
Ramiro Olivera Fedi
Julián Antonielli

Objetivo

El objetivo del trabajo práctico es desarrollar una aplicación que resuelva niveles de una variante del juego Flow, obteniendo soluciones exactas y aproximadas.

Algoritmos

Para la resolución del problema de forma óptima se utilizó un algoritmo de *backtracking* (Fuerza Bruta), podando casos imposibles para reducir la complejidad del algoritmo y evaluar menos casos. Para el caso de búsqueda aproximada se utilizó un método de *Hill Climbing.* A continuación pasamos a explicar ambos algoritmos, las decisiones tomadas, y más tarde los resultados obtenidos.

Parseo y modelado

El parseo de la entrada es trivial. Consiste en leer *Strings* de los argumentos y separarlos por espacio en elementos.

Para representar el tablero utilizamos una matriz, y a su vez, para representar cada elemento, que en el juego representa un casillero, optamos por crear una clase *Point.* Dicha clase persiste el color del casillero, si se trata de un nodo de inicio o de fin, su posición, y la dirección del camino que lo atraviesa.

Validación de entrada y soluciones triviales

Antes de pasar a cualquiera de los algoritmos de solución, pasamos el tablero por una serie de validaciones que se encargan de chequear la validez de la entrada como tablero del juego. Entre estos chequeos incluimos:

- Entrada bien formada
- Cantidad de nodos por color
- Tamaño del tablero

De no pasar estas validaciones devolvemos el mensaje de error: *Entrada no válida.* Ahorrandonos la carga de procesamiento y búsqueda de una solución que sabemos de antemano no será correcta.

Una vez pasadas las validaciones, y parseado el argumento de modo de solución, se pasa el tablero al algoritmo adecuado.

Algoritmo de búsqueda para la solución óptima

Utilizamos un algoritmo de *backtracking* recursivo, que funciona de la siguiente manera:

1. Se recorre la matriz, generando una lista de tuplas de nodos (puntos de colores originales), con el comienzo y fin de cada camino, marcando cada nodo según corresponda (color, tipo de nodo, nodos de fin, etc...).

2. Se itera sobre la lista de nodos definida previamente, tomando el nodo actual de la lista e iterando sobre un conjunto de direcciones, representadas por tuplas de dos componentes, donde cada componente puede ser 1 o -1, representando el *step* en el eje x y del eje y para cada dirección.

3. Se chequea que el casillero buscado siguiendo la dirección definida esté dentro de los confines de la matriz. De estarlo, se llama a la función de manera recursiva con el siguiente punto.

4. Si el próximo casillero es un nodo, se repite el proceso previo, con el próximo nodo de la lista.

5. Si ya se unieron todos los nodos, se calcula la cantidad de casilleros libre, y se guarda la matriz, junto con dicha cantidad en un objeto del tipo *Solution.*

6. Si la cantidad de casilleros libres es 0 se retorna true. De no serlo, se retorna false y se continúa por un camino distinto continuando con la recursión.

Algoritmo de búsqueda para la solución aproximada

Utilizamos un algoritmo de *Hill climbing* recursivo que funciona de la siguiente manera:

1. Se recorre la matriz, generando una lista de tuplas de nodos (puntos de colores originales), con el comienzo y fin de cada camino, marcando cada nodo según corresponda (color, tipo de nodo, nodos de fin, etc...).

2. Se itera sobre la lista de nodos definida previamente de manera aleatoria, tomando el nodo actual de la lista e iterando sobre un conjunto de direcciones, representadas por tuplas de dos componentes, donde cada componente puede ser 1 o -1, representando el *step* en el eje x y del eje y para cada dirección. Las direcciones se recorren de tal manera, que el camino busca acercarse lo más rápido posible al nodo final de un determinado color.

3. Se encuentra el primer camino posible, y luego se itera intentando ampliar el camino agregando pares de nodos, codos y cambiando direcciones, hasta que esto ya no es posible. Se almacena esta solución.

4. Si queda tiempo, se repite el proceso en busca de una mejor solución. Al acabarse el tiempo, se devuelve la mejor solución almacenada.

Resultados

A continuación compararemos la efectividad de distintos algoritmos para tableros particulares, dejando en evidencia la razón por la cual optamos por los algoritmos de solución definitivos.

Para realizar la comparación, primero definiremos los algoritmos contra los cuales compararemos.

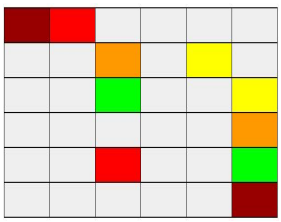
Algoritmo de búsqueda para la solución óptima

- Algoritmo de *backtracking* con poda (A): Algoritmo definitivo.
- Algoritmo de *backtracking* sin poda (B): Algoritmo análogo al definitivo sin validaciones o cortes en la construcción de caminos que se saben no encontrarán la solución óptima.

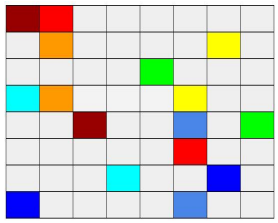
Algoritmo de búsqueda para la solución aproximada

- Algoritmo *Hill climbing* sin direcciones dirigidas (C)*:* Algoritmo análogo al definitivo sin (O con una muy básica) la función adecuada para elegir el mejor camino para la siguiente iteración.
- Algoritmo *Hill climbing* con direcciones dirigidas (D)*:* Algoritmo definitivo.

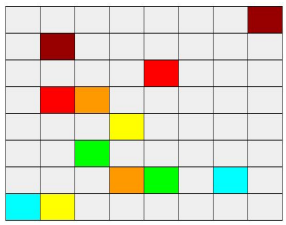
A continuación mostraremos los casos que utilizaremos para comparar los tiempos de cada algoritmo:



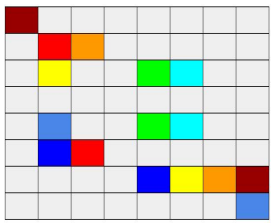
Caso I



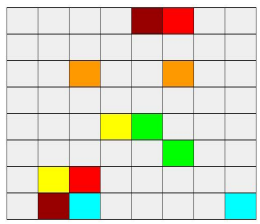
Caso II



Caso III

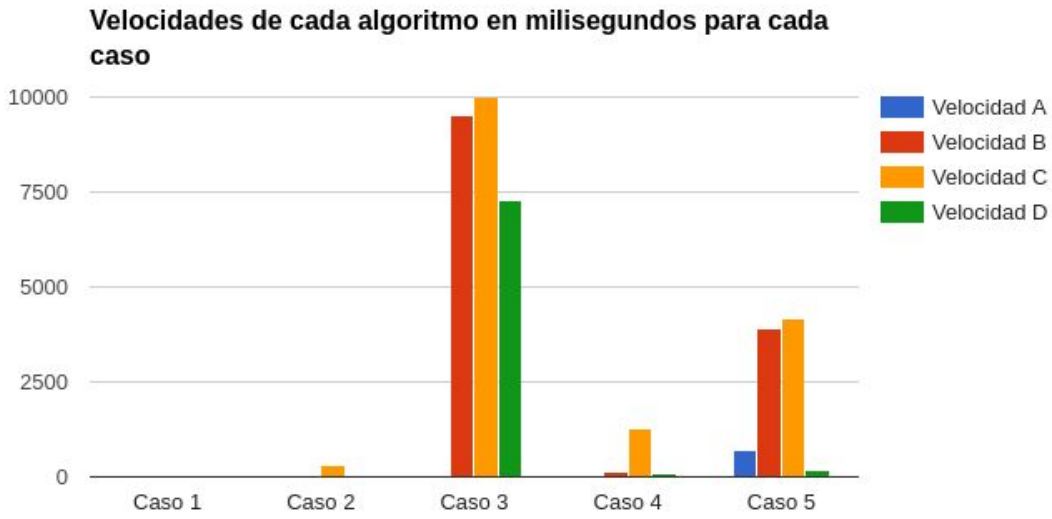


Caso IV



Caso V

Medimos ahora los tiempos de ejecución para cada uno de los algoritmos, poniendo 10 segundos como tiempo máximo a los algoritmos de *Hill Climbing*, y los graficamos para realizar una comparación de los tiempos de ejecución:

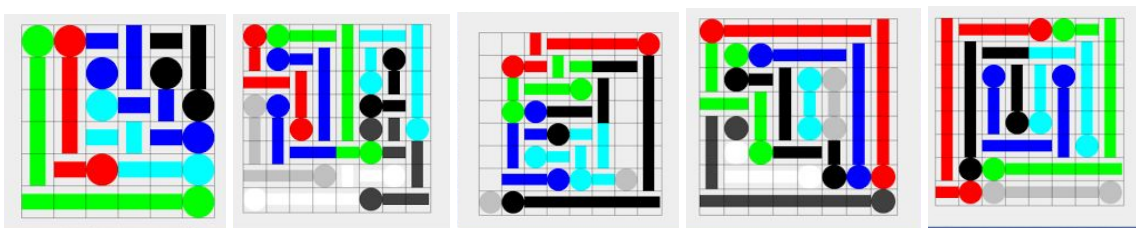


Como puede verse en el gráfico, el algoritmo *A* es estrictamente mejor y más rápido que el algoritmo *B,* lo cual tiene sentido, ya que el *A* consiste en una optimización del algoritmo *B.* Con relación a los algoritmos de *Hill Climbing,* puede verse que el *D* es siempre más rápido que el *C,* lo que implica que iterar sobre las direcciones para acercarse a los nodos implica una optimización de tiempo de búsqueda.

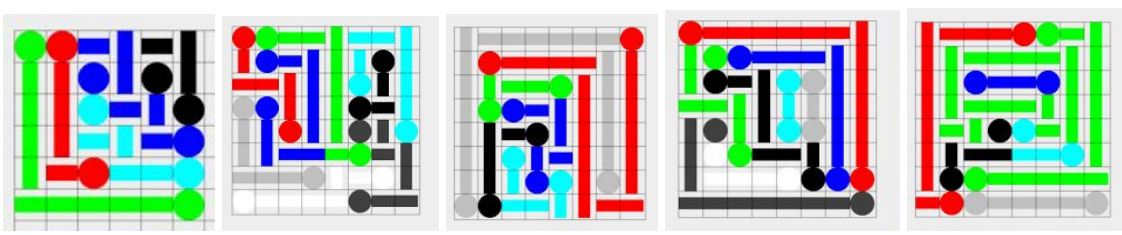
En cuanto a las diferencias de tiempo entre el *Hill Climbing* y el algoritmo de búsqueda óptimo, se deben a que el primero busca la solución más rápida y luego itera sobre esa solución, haciendo de este algoritmo excelente para la búsqueda de máximos locales, pero no tan bueno como buscador de la solución óptima. Es por esta razón que para analizar mejor la comparación entre las velocidades de estos algoritmos, debemos ver la solución generada en el tiempo expuesto

Mostramos ahora las soluciones encontradas por los algoritmos de *Hill Climbing* para las velocidades mostradas en el gráfico:

Algoritmo *Hill climbing* sin direcciones dirigidas



Algoritmo *Hill climbing* con direcciones dirigidas



Como puede verse, en el tiempo límite de 10 segundos, los algoritmos de *Hill Climbing* lograron encontrar en la mayoría de los casos la solución óptima, tardando como era de esperarse un tiempo mayor al de la solución óptima. Cabe destacar el caso 5, en el cual el algoritmo con direcciones dirigidas al nodo final, encontró una solución óptima distinta y en menor tiempo que el algoritmo diseñado con esta finalidad.

Se estudiaron casos donde existen soluciones no óptimas para el tablero, para comparar los tiempos entre los algoritmos, y se encontró que en promedio los algoritmos de *Hill Climbing* retornan una solución válida en menos de 4 segundos, mientras que el algoritmo de búsqueda de la solución óptima puede llegar a tardar más de 90 minutos.

Conclusiones

Analizando el enunciado y el problema con cuidado, podemos notar que se trata de un problema NP, ya que dar con un algoritmo para chequear si un tablero determinado es solución del problema es relativamente fácil y es fácil ver que la complejidad de este algoritmo es de orden polinómico, en particular de O(N^2).

Dicho esto, hubiésemos podido reducir en tiempo polinómico nuestro problema a alguno de los problemas NP-completos más estudiados (SAT, Clique, Vertex cover) y utilizar alguno de esos algoritmos para lograr resultados excepcionales.

No obstante, decidimos no ir por ese camino ya que hubiese requerido implementar un algoritmo genérico, y no hubiese requerido de nuestra parte diseñar el algoritmo ni la estructura de datos, que consideramos claves para este trabajo.

Dicho esto, y analizando los resultados expuestos en la sección anterior, consideramos que nuestros algoritmos son efectivos y eficientes para encontrar soluciones a tableros relativamente pequeños (De hasta 10x10). La poda en los casos del *backtracking* y el ajuste mediante el recorrido de las direcciones del algoritmo de *Hill climbing* fueron clave a la hora de optimizar los algoritmos de búsqueda de soluciones, y presentan el diferencial frente a los algoritmos analizados.