

Trabajo práctico Especial

Estructura de datos y Algoritmos

Informe de desarrollo

Moreno, Juan Pablo - 54182

Ibars Ingman, Gonzalo - 54126

Pascale, Juan Martín - 55027

1. Introducción

Se propuso como trabajo práctico el desarrollo de una aplicación que resuelva niveles del juego Pipe Dream mediante una solución exacta o una aproximada utilizando la técnica hill climbing.

La aplicación abre archivos de texto que representan un nivel a resolver. El mismo cuenta con cierta cantidad de tuberías (*pipes*) disponibles, el punto y dirección del pipe inicial (origen) y posiciones bloqueadas (si las hay). El objetivo del juego es construir la cañería más larga posible que permita transportar el agua desde el origen, hacia afuera del tablero.

Además, se desarrolló una interfaz gráfica que permite ver tanto la solución final como los pasos intermedios.

El objetivo del presente informe es exponer la manera en que se obtuvieron soluciones exactas y aproximadas para este problema, incluyendo los algoritmos creados y su orden de complejidad.

2. Estructuras

3.1 Tablero

El juego consta de un tablero de posiciones, las cuales pueden contener una pared, una tubería, una posición inicial o bien ser una posición vacía.

La interfaz *BasicBoard* contiene los métodos que deberían existir en cualquier implementación que se use para representar el tablero de posiciones.

La orientación del tablero está dada por un sistema de referencia cartesiano, es decir, consta de coordenadas y direcciones. La clase *Point* contiene la información y los métodos necesarios para describir una coordenada y el enum *Dir* es utilizado para describir un punto cardinal: *NORTH* (Norte), *SOUTH* (Sur), *EAST* (Este), *WEST* (Oeste), que puede ser de origen o destino.

El tablero de posiciones está representado por la clase *Board* que implementa la interfaz *BasicBoard*. Ésta contiene una matriz de *Tile*, un enum dentro de la clase, utilizado para representar el estado de cada posición del tablero. Además, contiene una variable del tipo *Point* y una de tipo *Dir* que indican la coordenada en el tablero identificada como origen del agua y la dirección hacia la cual fluye respectivamente.

La clase *Board* implementa los siguientes métodos:

Método	Descripción
<i>isEmpty</i>	Recibe una coordenada y devuelve un booleano indicando si la posición en el tablero está vacía. Si la coordenada no pertenece al tablero devuelve <i>false</i> .
<i>putPipe</i>	Recibe una coordenada y una tubería y la coloca en esa posición.
<i>removePipe</i>	Recibe una coordenada y remueve una tubería de esa posición si la hay.
<i>hasPipe</i>	Recibe una coordenada y devuelve un booleano indicando si hay una tubería en esa posición.
<i>withinLimits</i>	Recibe una coordenada y devuelve un booleano indicando si esa posición pertenece al tablero.
<i>isBlocked</i>	Recibe una coordenada y devuelve un booleano indicando si esa posición está bloqueada al paso de una tubería. Se implementó este método en vez de usar <i>isEmpty</i> para

	evaluar el caso en el que hay una tubería en forma de cruz pero el camino de tuberías puede pasar. (Es decir, la posición no está vacía y a la vez no está bloqueada)
<i>clear</i>	Elimina todas las tuberías del tablero.

Para la implementación de las tuberías, se utilizó un enum que contiene objetos denominados con una letra que indica el tipo de tubería seguida de un número para el índice, a excepción de la tubería en forma de cruz para la cual se usó *CROSS*. Para las tuberías en forma de “L” se utilizó *Li* con $i = 1, 2, 3, 4$; y para las tuberías rectas *Ij* con $j = 1, 2$.

Cada tubería o pipe posee los siguientes métodos:

Método	Descripción
<i>canFlow</i>	Recibe la dirección de la cual proviene el flujo y devuelve un booleano indicando si es compatible con la pieza, es decir, si puede fluir a través de ella.
<i>flow</i>	Recibe la dirección desde donde proviene el flujo y retorna otra dirección que indica hacia dónde va el flujo después de pasar por la tubería.

En un principio, cada posición había sido representada como una instancia particular de una clase *Tile*, pero al ser muy ineficiente en cuanto a espacio se optó por modelar *Tile* como un enum. Además al realizar pruebas con el algoritmo de solución exacta sobre un mismo tablero, se pudo observar que la implementación con el enum tardaba *3 minutos* menos que la anterior, por lo que se concluyó que resultaba más beneficioso utilizarla.

3.2 Tuberías disponibles

A la hora de modelar *Pipes* disponibles, se usó la clase *PipeBox*, que contiene dos arreglos. El primero es de *Pipe*, y es utilizado como una forma fácil de obtener un *Pipe* por el número de índice. El segundo es de enteros y almacena en la posición correspondiente al índice de cada *Pipe* la cantidad disponible de dicho *Pipe*.

Los siguientes métodos son parte de la lógica necesaria en *PipeBox*:

Método	Descripción
--------	-------------

<i>getPipe</i>	Recibe el número de índice de <i>Pipe</i> y devuelve dicho <i>Pipe</i> .
<i>getSize</i>	Recibe el número de índice de <i>Pipe</i> y devuelve la cantidad de <i>Pipes</i> disponibles.
<i>addOnePipe</i>	Recibe el número de índice de <i>Pipe</i> y agrega uno a la caja.
<i>removeOnePipe</i>	Recibe el número de índice de <i>Pipe</i> y remueve uno de la caja.

3.3 Soluciones

Para la implementación de la estrategia hill climbing, se pensó la forma de representar una solución, para esto se implementó la clase *Solution*. Dicha clase, internamente almacena una colección de *Pipe*. Así, se definió “Solución del juego” como una secuencia de *Pipes* tal que al colocarlos de forma ordenada desde el origen y teniendo en cuenta la dirección del flujo a medida que varía, se llega a algún borde del tablero con flujo saliente.

Solution permite almacenar/retirar un *Pipe* desde el principio o el final de la lista, y conoce su longitud y la devuelve mediante el método *Size*.

Se definió “vecino” como una solución de mayor longitud. *Solution* implementa el método *findBestNeighbor* que itera sobre los vecinos y devuelve el mejor, es decir, cada solución “sabe” cuáles son sus vecinos. Se explicará en detalle el funcionamiento en la parte de heurísticas de la solución aproximada.

3. Algoritmos

4.1 Algoritmo exacto

4.1.1 - Explicación general

Se implementó un algoritmo que dado un tablero con un estado inicial (es decir, un tablero que contiene un origen y puede contener paredes) y un *PipeBox* encuentre el camino más largo para llevar el agua hacia afuera del tablero. Para esto se utilizó *Backtracking*. (http://es.wikipedia.org/wiki/Vuelta_atr%C3%A1s)

Se tiene un método que funciona como wrapper del método recursivo de backtracking. cuyo pseudocódigo orientado a la aplicación es el siguiente:

```

backtrackingRecursivo(board, pipebox, point, from, currentSolution)
{
    //Caso base - Encontró una solución
    IF (!withinLimits(point)){
        addSolutionIfBetter(currenSolution);
        return;
    }
    //Caso especial: Hay una cruz - Puede pasar
    IF (!isEmpty(point) && !isBlocked(point)){
        PIPE = getPipe(point);
        DIR = flow(PIPE, from);

        addPipeWithoutPipebox(PIPE, point, currentSolution);

        //Llamada recursiva con la siguiente posición
        backtracking(board, pipebox, nextPoint(point, DIR), opposite(DIR),
currentSolution)
        removePipeWithoutPipebox(point, currentSolution);
        return;
    }

    //Si la posición está libre, se prueba construir un path con todos los pipes
    posibles.
    FOR (pipe IN pipebox){
        IF canFlow(pipe, from){
            DIR = flow(pipe, from);

            addPipe(PIPE, point, pipebox, currentSolution);

            backtracking(board, pipebox, nextPoint(point, DIR),
opposite(DIR), currentSolution)
            removePipe(point, pipebox, currentSolution);
        }
    }
}

```

Como caso base, si se llega a una posición fuera del tablero significa que se encontró una solución debido a que se pudo construir un camino hasta esa posición. Si ésta es mejor que la encontrada hasta el momento, se copia la colección de *Pipes* para aplicarla en caso de no encontrar una mejor. La colección ofrece la posibilidad de obtener el tamaño en $O(1)$ para comparar el largo de la solución con otras soluciones de forma eficiente.

En caso de estar fuera del caso base, el agua continúa en una posición dentro del tablero. Si esa posición no está libre y a la vez no está bloqueada (haciendo referencia al método *isBlocked*), el algoritmo se encuentra ante un pipe en forma de cruz, por lo que puede pasar a través del mismo y continuar armando el camino de tuberías. En ese caso, se coloca el pipe en la colección para aumentar en uno la longitud del camino pero no se la descuenta del PipeBox porque no se está utilizando una nueva tubería sino la misma.

Si la posición actual está libre, se itera sobre los pipes disponibles en el PipeBox, y en caso de que sea posible colocar alguno, se lo coloca, se llama recursivamente. Luego de probar con todos los Pipes posibles se libera la posición en el tablero antes de retroceder un espacio en el recorrido, es decir, se actualiza el estado del tablero a como estaba antes de hacer la llamada recursiva.

En resumen, el algoritmo coloca, dado un origen, todos los pipes en la posición siguiente que conecten con la tubería anterior, se mueve a la siguiente posición dada por la variación del flujo después de pasar por el pipe y vuelve a repetir este último paso con los pipes disponibles. Cuando el algoritmo llega a una posición en la que no puede continuar, ya sea porque está bloqueada o porque no tiene pipes disponibles, retrocede para evaluar otros caminos.

Se realiza una poda en la longitud máxima del camino, se calcula cuál sería la longitud de un camino que use todos los pipes y las cruces dos veces, y si se encuentra un camino de esa longitud entonces no tiene sentido seguir buscando soluciones ya que se encontró la mejor.

4.2 Algoritmo aproximado

4.2.1 Hill climbing

Para la búsqueda de la solución aproximada, se utiliza *hill climbing* (http://es.wikipedia.org/wiki/Algoritmo_hill_climbing). El algoritmo parte de una solución inicial aleatoria y en cada iteración, mientras pueda correr dependiendo de distintos factores relacionados con el contexto de cada problema (en este caso el tiempo restante), genera vecinos para la solución actual y aplica el mejor. Es independiente de la función “obtener vecinos” y del problema. Cuando luego de una iteración no encuentra un mejor vecino, si tiene tiempo restante puede partir de otra solución inicial. El sentido de que la solución inicial sea aleatoria es escapar a los máximos locales. Si la solución inicial fuera siempre la misma, siempre se llegaría a la misma solución final, es decir, no se podría escapar al máximo local. (Definimos qué es un vecino en este caso en la sección 3.3)

Pseudocódigo (Sacado de la fuente)

Hill Climbing Algorithm

```
currentNode = startNode;
loop do
  L = NEIGHBORS(currentNode);
  nextEval = -INF;
  nextNode = NULL;
  for all x in L
    if (EVAL(x) > nextEval)
      nextNode = x;
      nextEval = EVAL(x);
  if nextEval <= EVAL(currentNode)
    //Return current node since no better neighbors exist
    return currentNode;
  currentNode = nextNode;
```

4.2.2 Solución al azar

Como hill climbing parte de una solución inicial, el objetivo es encontrar rápidamente una solución y que la misma sea aleatoria para escapar a los máximos locales. Para esto, se utiliza una modificación del algoritmo de armado de camino por *backtracking*. El procedimiento consiste en mezclar en cada llamada recursiva el orden de las tuberías en el *PipeBox*. De esta manera, cada vez que se llame al método para encontrar la solución inicial, se van a probar las distintas posibilidades de pipes a colocar en un orden diferente y hay una probabilidad muy baja que el algoritmo retorne la misma solución que retorno en llamadas anteriores.

4.2.3 Soluciones vecinas

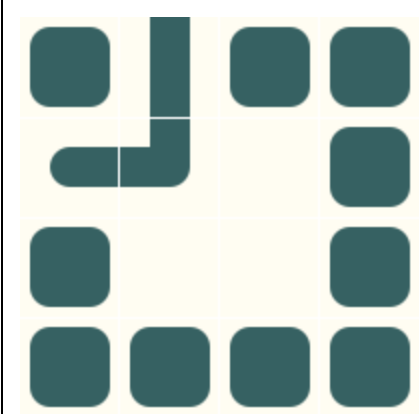

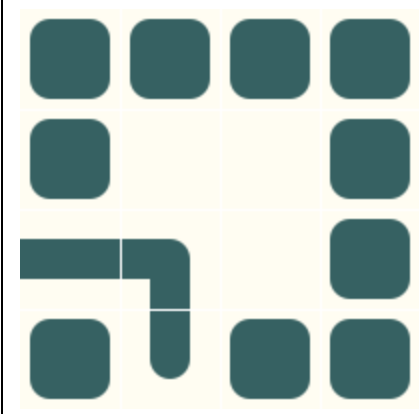
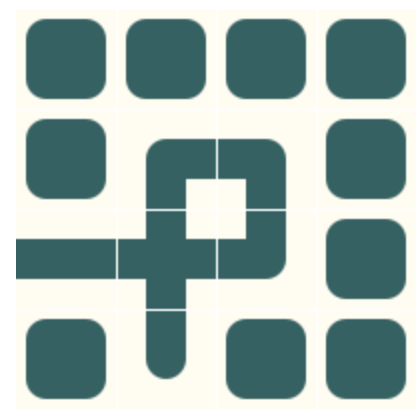
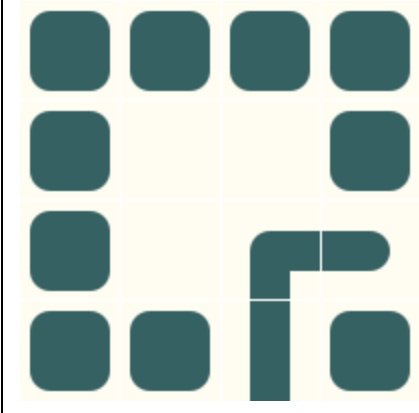

Una vez encontrada la solución al azar, para encontrar las vecinas se aplican heurísticas. Este procedimiento consiste en usar ciertos conocimientos que se tienen sobre el problema y la solución actual para modelar la misma de forma tal que a partir de ella se obtenga una solución mejor. Se establecen ciertas modificaciones predeterminadas frente a situaciones usuales (target) en la solución para mejorarla.

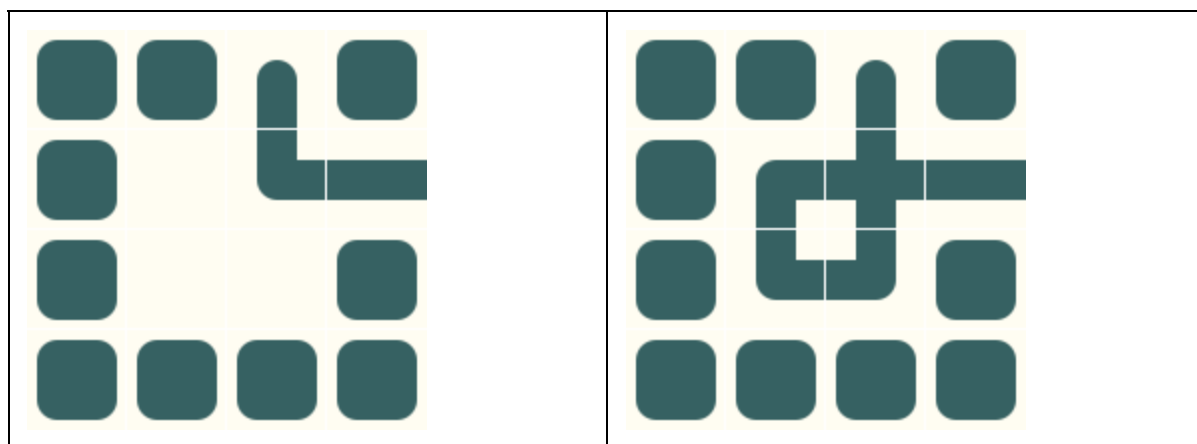
Previamente, se realizan validaciones teniendo en cuenta la disponibilidad de tuberías y la disponibilidad de espacio en el entorno alrededor del target.

4.2.4 Heurísticas aplicadas

En los próximos gráficos, se muestran los casos que se tienen en cuenta al aplicar las heurísticas. No interesa la tubería de entrada ni la de salida. Los casos vistos ocurren en matrices de 2x2.

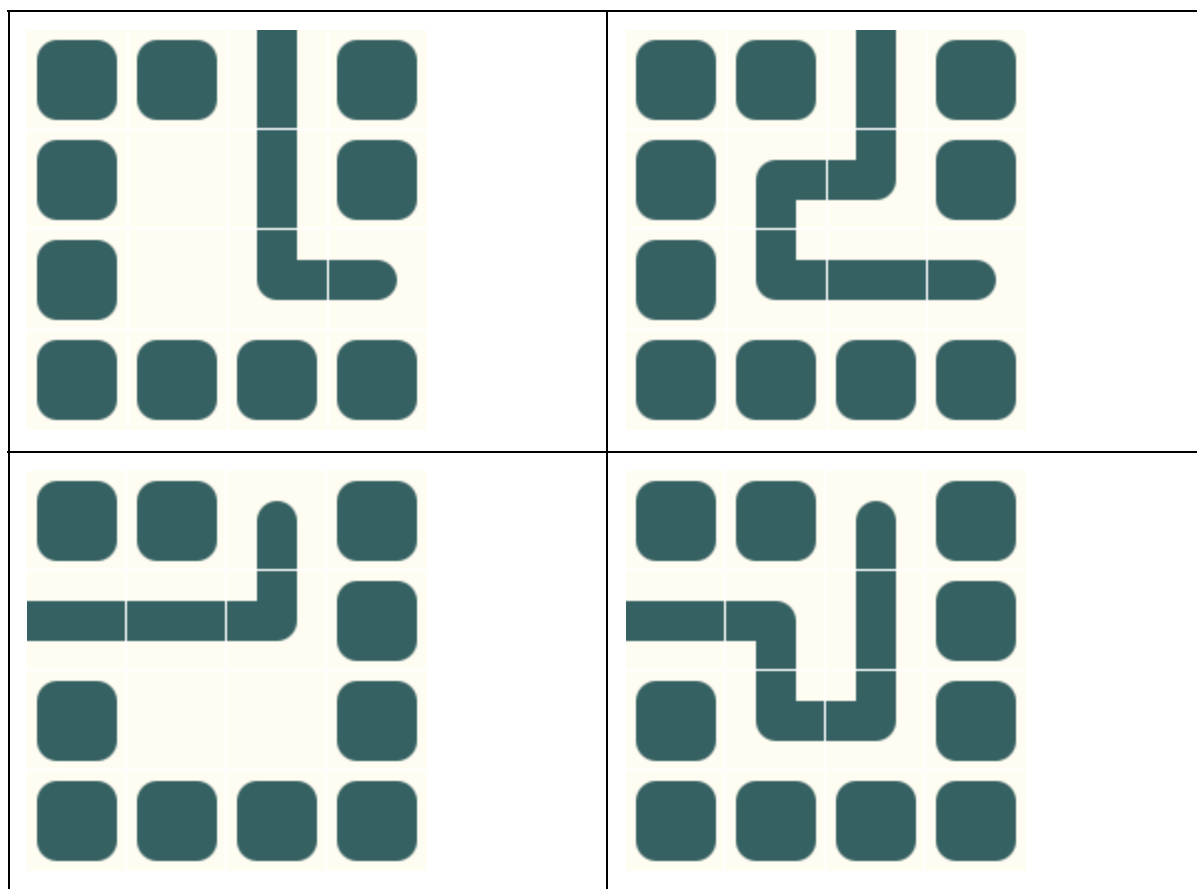
1_ Sustitución de un pipe en L por una cruz completando el camino con un cuadrado.

Target	Solución
	
	
	



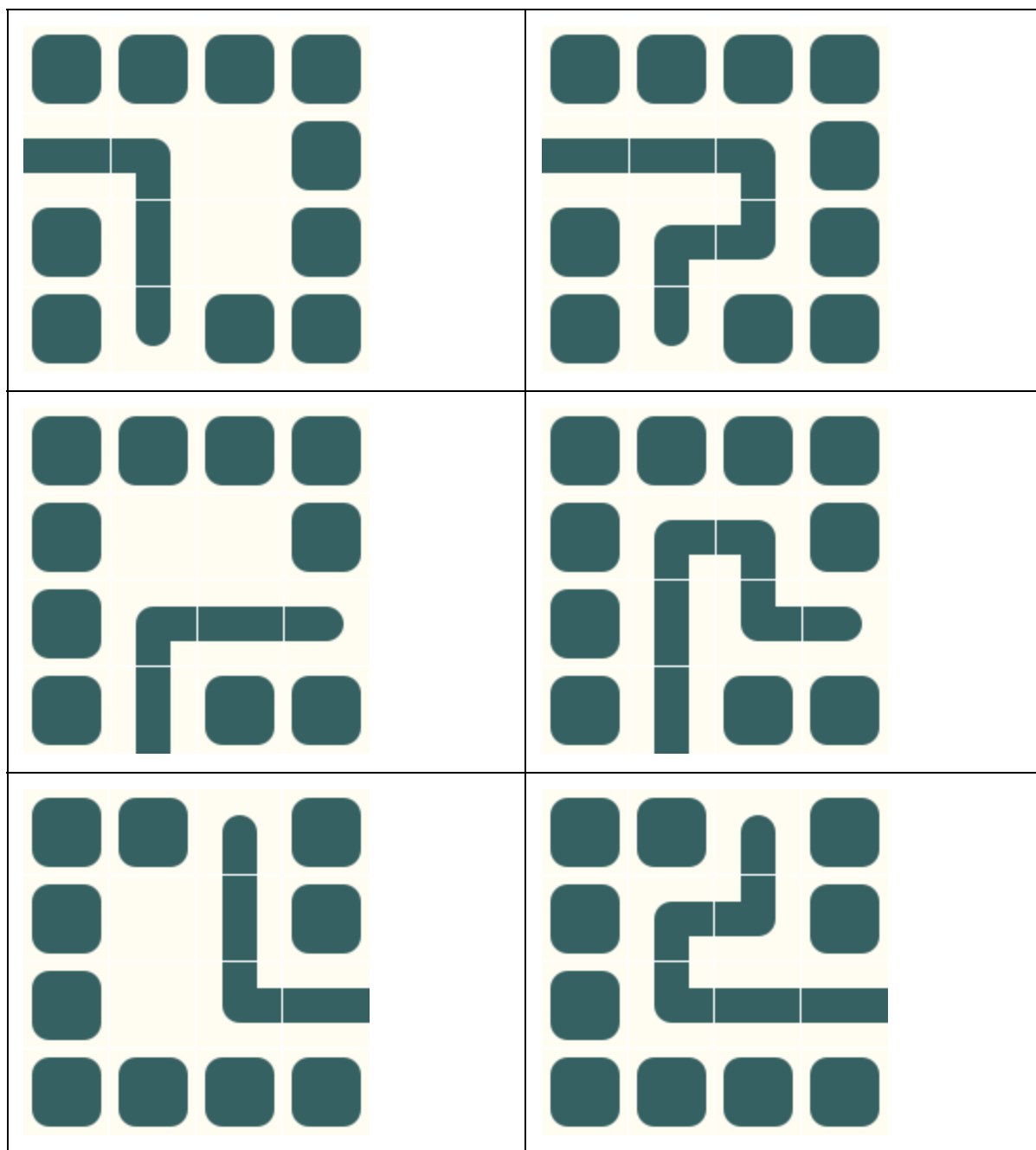
2_ Estiramiento de un pipe en L seguido de un pipe en I:

Target	Solución



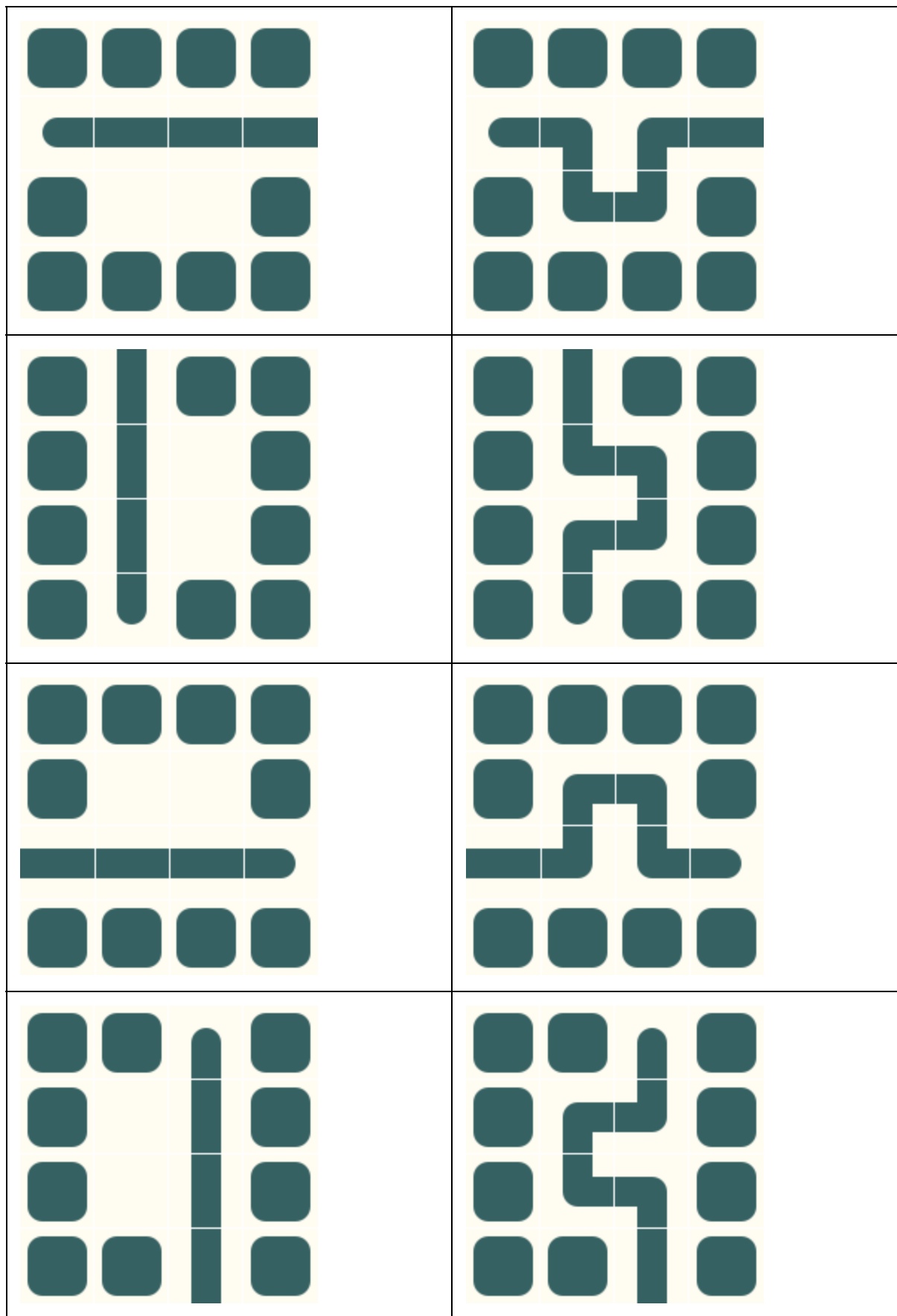
3_ Estiramiento de un pipe en I seguido de un pipe en L (Caso inverso al anterior):

Target	Solución



4_ Estiramiento de dos pipes en l seguidos.

Target	Solución
--------	----------



Al conocer las heurísticas que se aplican, se puede saber de antemano qué solución de las que se proponen es más larga por lo que se realizó una optimización de hill climbing en la que la función *findBestNeighbor* encuentra los vecinos y devuelve el *findBestNeighbor* puede dejar de recorrer los vecinos antes si encuentra posible aplicar una solución de tipo 1_ (la más larga de las heurísticas propuestas).

4.2.5 Armado de la solución

Los métodos que aplican las heurísticas devuelven una solución parcial solamente con la heurística. El método que llamó recibe la heurística y genera la solución definitiva entre la actual y la parcial.

4.3 Comparaciones entre algoritmos

Arcvivo	Tiempo Exact	Longitud Solución (Exact)	Logitud Solución (Aprox - 1 min)	Logitud Solución (Aprox - 2 min)
archivo1.txt	0 s	17	*	
archivo2.txt	23s	40		
archivo3.txt	54s	44		

* No fue posible obtener estos datos ya que, en algunas heurísticas, existe alguna verificación que se está omitiendo en el momento de verificar si el PipeBox posee esa tubería, y se obtienen, en algunos casos, soluciones de longitud mayor a la exacta.

4. Conclusiones

Si bien el algoritmo exacto devuelve la mejor solución, esta es muy costosa y requiere tiempo de procesamiento. Si se implementaran más podas, disminuiría el tiempo de procesamiento en tableros grandes ya que existirían muchas ramas que no se evaluarían, pero en tableros chicos aumentaría porque la posibilidad de poda es menor y se requiere realizar más comparaciones para evaluar si es posible hacer una poda.

En el caso del algoritmo aproximado, se podrían obtener mejores soluciones implementando más heurísticas. El punto en contra es que cuantas más heurísticas se implementen, más comparaciones se realizarán por lo que la eficiencia se verá afectada.