Universidad de Santiago de Chile

Facultad de Ingeniería Informática

Taller de Programación

Profesor Pablo Román Asenjo

Taller 1
Crossing River

Nicolás Aguilera

Abril 2023

Índice

1. Introducción

El siguiente informe corresponde al desarrollo del Taller 1 del curso de Taller de Programación, el cual consiste en la aplicación del Algoritmo A^* para la resolución del problema *Crossing River* en su forma general, con un conjunto de conductores $D = \{1, 2, ..., d\}$, un conjunto de objetos $C = \{d + 1, ..., c\}$ y conjuntos de restricciones R_i y R_d para el lado izquierdo y derecha, respectivamente.

2. Desarrollo

2.1. Explicación del Algoritmo

El programa consta de la implementación de algunas clases, dentro de las cuales se encuentran *CrossingRiver*, *State*, *Heap* y *Operation*, las cuales se encargan de leer el problema, extraer los principales valores y restricciones. En base a esto, el programa genera todas las operaciones que son posibles de realizar. Luego de esto comienza a operar el algoritmo el cual principalmente verifica en cada paso si es que el estado actual corresponde al estado final. Si no lo es, genera nuevos estados que no han sido sido revisados y que cumplan con las restricciones. El programa termina cuando ha llegado al estado final o cuando no quedan más estados por revisar, notificando al usuario de esto a través de la terminal.

2.2. Heurísticas o técnicas utilizadas

El algoritmo base para la resolución de este problema es el Algoritmo A^*, utilizado para problemas de búsqueda en espacio de estados, y que pone énfasis en los conocimientos previos o *Heurísticas* del problema para su resolución. En este caso el algoritmo desarrollado utiliza un *min Heap* como cola de prioridad para los estados que serán visitados. Los estados tienen un atributo llamado *distance* el cual describe la distancia tentativa a la que se encuentra del estado final. Este valor es utilizado para agregar los estados al *Heap* y ordenarlos de menor a mayor distancia, por lo que el próximo estado a analizar será siempre el de menor costo. Es decir, la técnica utilizada en este algoritmo es *Greedy* ya que en cada paso estamos tomando el estado más cercano al estado final. La heurística utilizada en este caso en bastante sencilla, ya que se considera la cantidad de elementos que aún no han cruzado (elementos a la izquierda) como la distancia tentativa hasta el estado final.

2.3. Funcionamiento del programa

La resolución del problema consta de una clase llamada *State*, la cual es una básicamente una representación de cada estado del problema, es decir, qué objetos se encuentran a la izquierda y a la derecha del río, y de qué lado del río se encuentra el bote, esto a través de una representación decimal única[1] para cada estado. Además, se tiene la clase *CrossingRiver*, la cual tiene como función principal la resolución del problema a través

[1]Se transforma el arreglo binario que representa cada estado en una potencia de 2.

del Algoritmo A^* usando colas de prioridad *open* y *closed*, mediante un *Heap*, donde la primera se encarga de guardar los estados por visitar y la segunda los ya visitados. El constructor de la clase se encarga de leer el archivo que contiene el problema e inicializar todos los valores importantes para la resolución de este: número de conductores, número de elementos, tamaño del bote, restricciones del problema, etc. Además la clase tiene un método llamado *solve()* que se encarga de resolver el problema generando todas las operaciones válidas[2] a través de un algoritmo recursivo, guardándolas en un arreglo de operaciones (clase *Operation*). El algoritmo funciona más o menos así: se agrega el estado inicial a la cola de prioridad *open* y se van generando los nuevos estados (moviendo el bote a la izquierda o derecha), verificando al mismo tiempo si estos ya se encuentran visitados o cumplen con las restricciones. Una vez generados los nuevos estados, se elimina el actual de la cola *open* y se agrega a la cola *closed*. Así, para los próximos estados se verifica si es igual al estado final, y se repite el procedimiento hasta encontrar el estado final o que no quede ningún estado por visitar.

2.4. Aspectos de implementación y eficiencia

Una de las principales ventajas de la solución propuesta es haber optado por trabajar las restricciones y estados con representaciones decimales en vez de arreglos binarios (y por lo tanto, matrices). Esto otorga una ventaja al momento de comparar y recorrer arreglos de restricciones y estados visitados, ya que se disminuye la complejidad del algoritmo de $\mathcal{O}(n^2)$ a $\mathcal{O}(n)$.

Otro elemento a considerar en términos de eficiencia es la mejora realizada en la generación de las operaciones (movimientos) posibles. Anteriormente, la función generadora de operaciones tenía una complejidad de $\mathcal{O}(2^n)$ ya que generaba todas las combinaciones de arreglos binarios de largo n y luego filtraba las operaciones válidas. Actualmente, la función considera un parámetro de entrada que contabiliza la cantidad de 1 que hay en el arreglo que se está construyendo. Cuando la cantidad de 1 es igual a la capacidad del bote, el algoritmo solo agrega 0 para los índices restantes, prescindiendo de combinaciones innecesarias. Para el caso en el que hay 3 conductores, 5 elementos y un bote de capacidad 2, se pasó de $2^8 = 256$ combinaciones generadas a 37, es decir un 14 % de la cantidad original. En este caso no se puede determinar la complejidad del algoritmo ya que depende de los parámetros del problema. Sin embargo, la complejidad ha sido reducida.

Otro aspecto que podría mejorarse tiene que ver con la memoria utilizada y está relacionado con el punto anterior, ya que el arreglo de operaciones donde estas se almacenan tiene un largo 2^n debido a que en el peor de los casos todas las operaciones posibles serían válidas. Sin embargo, esto se podría mejorar calculando previamente la cantidad de operaciones válidas y así utilizar solamente la memoria necesaria.

[2]Se refiere a las operaciones que cuenten con al menos un conductor y que respeten la capacidad del bote.

2.5. Ejecución del código

Para hacer uso de la implementación propuesta, se deben tener los archivos de las clases mencionadas en la *Explicación del Algoritmo*, es decir, extensiones .cpp y .h, además del archivo *makefile* que facilita el proceso de compilación y ejecución de cada test para cada clase. Una vez reunimos todos los archivos, se procede a abrir la *Terminal* en la carpeta donde tenemos todos los archivos y se hace lo siguiente:

- **testMain:** Para compilar el archivo, ejecutar la linea de comando *make testMain*. Para correr el archivo compilado ejecutar la linea de comando *make runTestMain*. Esto pedirá la cantidad de veces que se quiere resolver el problema y luego el nombre del archivo, el cual por defecto se llama *sistema.txt*, aunque podría tener otro nombre. Esto entrega el resultado por pantalla y el tiempo promedio que demora en solucionar cada iteración.

- **testCrossing:** Para compilar el archivo, ejecutar la linea de comando *make testCrossing*. Para correr el archivo compilado ejecutar la linea de comando *make runTestCrossing*.

- **testState:** Para compilar el archivo, ejecutar la linea de comando *make testState*. Para correr el archivo compilado ejecutar la linea de comando *make runTestState*.

- **testHeap:** Para compilar el archivo, ejecutar la linea de comando *make testHeap*. Para correr el archivo compilado ejecutar la linea de comando *make runTestHeap*.

3. Bibliografía y referencias

1. Román P. (2022). *Evaluación 1 - Taller de Programación, Cruzando el río.*