PRÁCTICA 2
ALGORITMIA BÁSICA

MEMORIA

Curso 2018-2019

Rubén Rodríguez Esteban 737215
José María Vallejo Puyal 720044

1. Introducción

El objetivo es diseñar un algoritmo de ramificación y poda que calcule el máximo ingreso total para la compañía uzbeka sabiendo que dicho ingreso aumenta con la distancia del trayecto a recorrer y el número de pasajeros con la restricción de que en ningún momento la capacidad del tren puede excederse. Se han empleado los siguientes tipos abstractos de datos:

- Nodo: representa cada uno de los nodos del árbol de ramificación y poda que almacena el espacio ocupado en cada uno de los fragmentos del viaje, el pedido actual y el beneficio actual.

- Pedido: refleja cada uno de los pedidos con los datos acerca de la estación inicial, la estación final y el número de pasajeros del pedido.

- Heap: representa un montículo de prioridad de mínimos para ordenar los pedidos por cantidad de estaciones recorridas.

Los clientes que hacen un pedido solo ocupan el tren en el tramo indicado. En el resto de tramos la capacidad disponible no es reducida por dicho pedido. Dicha limitación a la hora de comprobar la capacidad del tren es importante ya que al insertar un pedido solo se debe comprobar la capacidad del tren en el tramo del pedido para decidir si es factible. Por ello se usa un vector que registra la capacidad de cada tramo en cada nodo y se comprueba si el número de billetes a insertar es menor que la menor capacidad disponible en ese tramo.

2. Diseño

Para poder resolver este problema, se ha empleado una tupla $X = (x_1, \ldots, x_p)$ de tamaño fijo en el número de pedidos. de manera que cada elemento x_i toma valor 1 si se acepta el pedido i-ésimo o 0 si no lo es. El espacio de posibles soluciones es un árbol binario, donde en cada nivel k ($1 <= k <= p$), se ha decidido si añadir o no los primeros k pedidos. El árbol completo tiene un total de $2^{(p+1)} - 1$ nodos. El problema se ha modelado empleando una estrategia de mínimo coste convirtiendo el problema de maximización original en uno de minimización expresando el beneficio obtenido como un valor negativo.

2.1 Función de poda U(x)

Para calcular la función de poda de cada nodo se usa una cota, cuyo valor inicial es la cota superior que va a almacenar la mejor ganancia obtenible desde el nodo actual. El mejor valor que puede tomar es el resultado de sumar al beneficio actual la ganancia de todos los pedidos que caben por completo en el tramo de recorrido del pedido. Matemáticamente U(x) se puede expresar de la siguiente forma:

$$cota_k = C_k + \Sigma^{p}_{i=k+1} r_i * n_i \text{ si } n - h >= p_i$$
$$U(x) = \min (U_x)$$

donde C_k es el valor de la cota hasta el pedido k, r_i es el precio del i-ésimo billete, n_i es el número de pasajeros del pedido i-ésimo, n la capacidad del tren, h los pasajeros de la parada con más gente a subir al tren y U_x el conjunto de todos los valores $cota_k$ de un nodo x.

2.2 Función de coste c(x)

El cálculo de la función de coste de un nodo es obtenido de forma directa en base al valor mínimo de las cotas de ese nodo, puesto que cada vez que se consigue una cota con un beneficio menor que el coste real, se actualiza el valor del coste con dicho valor de cota. Matemáticamente la función de coste c(x) se puede expresar de la siguiente forma:

$$c(x) = U(x)$$

2.3 Función de coste estimado c^(x)

Para calcular la función de coste estimado c^(x) se ha optado por considerar que en cada fragmento de viaje de un pedido se contarán los pasajeros que quepan de no caber todos. Así el número de pasajeros puede variar en cada tramo dependiendo de la capacidad disponible del tren. Matemáticamente se puede expresar el coste estimado como sigue:

$$\hat{c}(x) = B_k + \Sigma^{p}_{i=k+1}\, p_i \text{ si } np_j > n - estacion_j$$
$$\hat{c}(x) = B_k + \Sigma^{p}_{i=k+1}\, n - estacion_j$$

donde B_k es el coste acumulado hasta el pedido k, p_i es el número de pasajeros en el pedido i-ésimo, np_j es la diferencia entre la capacidad de pasajeros del tren y los que pasajeros que van as subirse en la j-ésima estación y $estacion_j$ son los pasajeros a subirse en la estación j-ésima.

3. Compilación y ejecución

Para compilar el material entregado se proporciona un archivo makefile que reúne los ficheros auxiliares necesarios para poder generar el archivo principal. Para la compilación se ha empleado el estándar de c++11 junto con el flag de compilación -03 con el fin de optimizar el rendimiento. Los comandos para poder llevar a cabo la compilación figuran en la cabecera del fichero makefile.

4. Pruebas de corrección

Para determinar si el algoritmo de ramificación y poda es correcto se ha desarrollado un algoritmo que calcula el beneficio de las instancias del problema empleando una solución de fuerza bruta. Dicho script, denominado *algFuerzaBruta.py*, hecho en Python. Dicho algoritmo muestra para cada instancia del problema el máximo beneficio y el tiempo de ejecución empleado, tanto en microsegundos como en segundos. El script se ejecuta por medio del siguiente comando: *python algFuerzaBruta.py*

Tras ejecutar el algoritmo de fuerza bruta se ha podido verificar que el algoritmo de ramificación y poda es correcto ya que ambos alcanzan la misma solución. Para poder facilitar la labor de corrección del profesorado se ha desarrollado un script en bash denominado *profitChecker.sh* que permite comparar los beneficios obtenidos por ambos algoritmos, devolviendo si son idénticos o no. Para poder ejecutar el script se deben indicar como parámetros los nombres de los ficheros de texto que contienen los beneficios obtenidos por el algoritmo de ramificación y poda y por el algoritmo de fuerza bruta, respectivamente. El siguiente comando muestra como ejecutar el script: *./profitChecker.sh resultados.txt resultadosFB.txt*

5. Pruebas de eficiencia

Para poder determinar la eficiencia del algoritmo de ramificación y poda se ha desarrollado un script en Python que compara para cada instancia del problema los tiempos en segundos empleados por ambos algoritmos indicando en cuántas instancias es más eficiente el algoritmo de ramificación y poda. El siguiente comando explica cómo ejecutar el script en cuestión: *python timeChecker.py*

Los gráficos siguientes muestran el tiempo de ejecución invertido por los dos algoritmos para resolver las 8000 instancias del problema. Tal y como se puede observar, el algoritmo de ramificación y poda ha resuelto todas las instancias en un tiempo que oscila entre los 0 segundos y los 0,02 segundos aproximadamente. Sin embargo, el tiempo máximo empleado por el algoritmo de fuerza bruta oscila entre los 0.2 segundos los 1.5 segundos como máximo. En más del 90% de las instancias es más rápido el algoritmo de ramificación y poda, probando así, que es más eficiente.