Memoria

Prácticas

Algoritmia Básica

Práctica 4

Autores:

Hugo Mateo Trejo, 816678

Paula Oliván Usieto, 771938

Resolución del problema con ramificación y poda

Toma de decisiones

Las primeras decisiones que se tuvieron en cuenta fueron las siguientes: cómo mostrar las soluciones del problema (con tuplas de tamaño fijo o variable) y cómo tomar las decisiones en el espacio de estados.

En cuanto a las tuplas en el problema se decidieron usar tuplas de tamaño fijo. En cada nivel del árbol se elige un pedido a añadir a la solución.

Debido a las normas de acotación, el espacio de estados tendrá $2^n - 1$ nodos los cuales todos ellos podrán ser posibles soluciones al problema. Sin embargo, por los predicados acotadores explicados anteriormente, los nodos que no sean hojas serán no óptimos.

Para eliminar estados repetidos, uno de los predicados acotadores implica que no se podrá asignar un índice de pedido menor al índice del nodo que se está generando, de manera que la tupla solución [1,0,1] donde cogemos el pedido 1 y 3 sólo podrá ser alcanzado desde la rama en donde el primer pedido asignado es 1.

El otro predicado acotador limita el número de pasajeros que puede transportar el tren entre dos estaciones, que será la capacidad que se lee del fichero de texto pruebas.txt. Es decir, ningún estado podrá ser solución si se están asignando pedidos a la solución cuyo número de billetes en intervalos entre estaciones iguales superan la capacidad marcada para ese tren concreto.

Fórmulas para realización del algoritmo

Nos encontramos ante un problema de maximización de beneficios, lo cual es equivalente a hacer la minimización del siguiente coste:

$$coste = nTicketsTotal - nTicketVendidos.$$

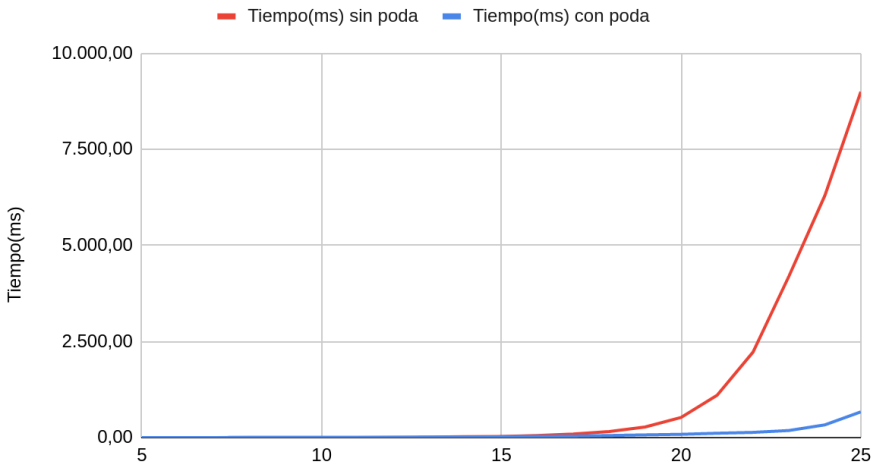
Para poder llegar a la solución óptima sin recorrer todo el espacio de estados viendo los mejores resultados de los nodos que cumplan los predicados acotadores se hace uso de una fórmula de coste de cada nodo, junto a una función de cota y la heurística.

Cada vez que vamos a visitar un nodo primero se comprueba si es un nodo que se debe podar, esto se da si el valor de la función de cota, que es la mayor cantidad de beneficio que podemos perder es menor que la heurística del nodo. Esto significa, que el nodo que estamos visitando como mínimo nos dará unas pérdidas mayores a las máximas pérdidas actuales que podemos tener en el peor de los casos.

La fórmula de la función de cota, teniendo en cuenta que todos los nodos ser solución es: $U = coste$

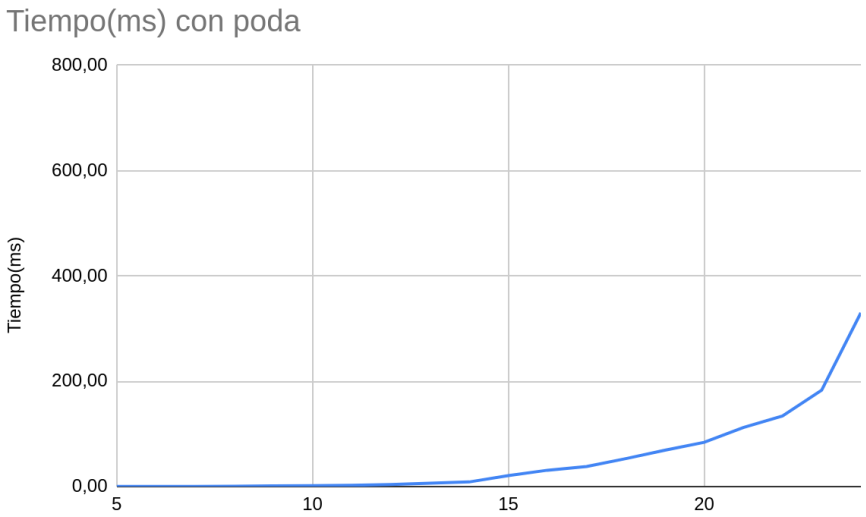
La fórmula de la función heurística será: $\widehat{c} = \sum_{i=1}^{j} nTickets(i)\ t.q.\ j = pedidoActual$

Pruebas realizadas y análisis de resultados



Se ve claramente que el problema crece de forma exponencial en 2^n. El algoritmo con poda crece con una constante mucho menor al sin poda, gracias a que elimina algunos estados sin necesidad de recorrerlos.

Sin embargo, esta reducción sólo es en constante y en la gráfica inferior se aprecia cómo también crece de forma exponencial.



Para comprobar la corrección del programa se han ejecutado varias pruebas y se han comprobado de forma manual comparándolas con la resolución hecha a mano. En ambas coinciden los resultados que se obtenían, por tanto se concluyó que el algoritmo de ramificación y poda era correcto. Posteriormente los resultados también serán comparados con el algoritmo de programación lineal para comprobar que ambos resultados coinciden entre ellos y con la solución a papel.

Resolución del problema con programación lineal

Toma de decisiones y fórmulas del algoritmo

En primer lugar se tuvo que decidir cuál de las librerías disponibles para Python de programación lineal era la más adecuada. Después de ver las ventajas y desventajas de cada una se decidió utilizar Python-PuLP, ya que era la más sencilla de utilizar y la más recomendada para personas que se iniciaran en la programación lineal; nuestro caso ya que todos los problemas de este tipo que hemos resuelto han sido a papel.

Posteriormente se decidieron las inecuaciones que se debían crear, junto a las variables asociadas a las mismas. Las inecuaciones serán tratadas posteriormente ya que van asociadas a las restricciones del problema. En cuanto a variables se crearon las siguientes:

- coste por reserva:

$$reserva\_nTickets(i) \ * \ (reserva\_estacionFinal(i) - reserva\_estacionInicial(i))$$

- volumen: es un entero que almacena la ocupación del tren en los nodos
- nTickets_totales: variable que almacena el beneficio total que nos va a retribuir la venta de cierto pedido

Para calcular estas variables y poder resolver el problema también es necesario tener en cuenta los parámetros que nos proporciona el problema; en nuestro caso es el inicio de cada bloque del fichero de pruebas y el contenido del mismo. Contamos pues con los parámetros explicados a continuación:

- tren_capacidad_maxima: contiene el número de la tercera variable de la línea de definición del bloque. Marca la restricción principal a la que debe atenerse la venta de tickets.
- nEstaciones: contiene el número de la segunda variable de la línea de definición del bloque. No será mayor que 7.
- nPedidos: contiene el número de la tercera variable de la línea de definición del bloque. No será mayor que 22.
- reserva_estacionInicial[]: vector que almacena, para todos los pedidos de un cierto bloque, la estación desde la cual van a salir. Se añade la primera columna del interior de un bloque.
- reserva_estacionFinal[]: vector que almacena, para todos los pedidos de un cierto bloque, la estación a la cual quieren llegar los distintos pedidos. Se añade la primera columna del interior de un bloque.
- reserva_nTickets[]: vector que almacena, para todos los pedidos de un cierto bloque, el número total de billetes que quieren obtener. Se añade la primera columna del interior de un bloque.

Una vez esto fue consensuado, se creó la función objetivo para poder evaluar el problema en cada punto válido. La función objetivo era maximizar el beneficio de la venta de billetes para un cierto recorrido del tren especificado en el bloque del fichero de pruebas. Como se puede observar al contrario que en el algoritmo anterior donde se decidió convertir

la maximización() en una minimización -(), en este caso la maximización se mantuvo como tal ya que gracias a la herramienta PuLP esto resultaba más sencillo.

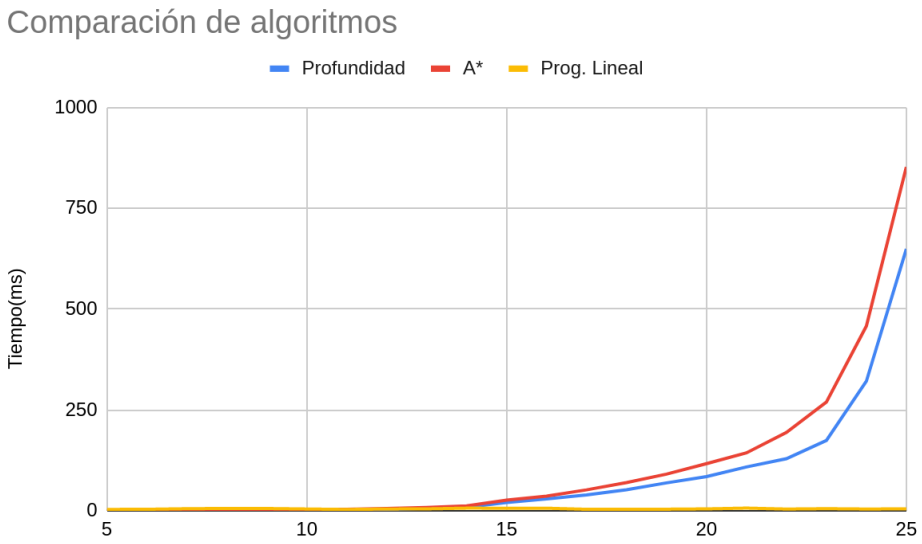
La fórmula de la función objetivo obtenida es por tanto: $\sum_{i=0}^{n} c(i) \ * \ var(i)$

Las variables son binarias, 0 o 1.

Para finalizar se definieron las restricciones, las cuales eran bastante similares a las que se crearon en el problema de ramificación y poda. Estas fueron las siguientes:

- No se podrá en ningún tramo del recorrido exceder la capacidad máxima del tren, la cual ha sido definida por el valor n del inicio del bloque.

Pruebas realizadas y análisis de resultados



Aunque no se aprecia, tanto el algoritmo de ramificación y poda como el A* son notablemente más rápidos que el de programación lineal en profundidades pequeñas. Sin embargo, ambos algoritmos crecen rápidamente de forma exponencial al aumentar la profundidad. El algoritmo de programación lineal, por otro lado, se mantiene completamente constante en tiempo al aumentar la profundidad.

Probablemente el tiempo aumente al incrementar mucho el número de pedidos, que implica aumentar el número de dimensiones, pero los algoritmos de programación lineal están pensados para trabajar con números muy grandes de dimensiones y se aprecia claramente en los resultados.

En promedio el algoritmo simplex tiene tiempo polinomial, lo que dista mucho del coste exponencial de la ramificación y poda. Incluso con podando ramas, que reducen la complejidad del algoritmo, se puede observar claramente que la programación lineal funciona mejor en profundidades grandes.

Respecto al algoritmo de profundidad frente al A* (anchura guiada por heurística), el de profundidad es ligeramente mejor. El de profundidad explora primero ramas muy profundas, lo que puede que en este caso permite podar ramas antes que en el A*, esto dependerá del problema al que nos enfrentemos y cómo se calculen c, $\widehat{c}$ y U. Además al utilizar una estructura de datos adicional y basarse solo en la pila permite al hardware predecir y ejecutar más rápido el algoritmo. El A* por otro lado no es recursivo, lo que implica una ventaja para el hardware.

Concluimos entonces que, el mejor algoritmo de exploración depende del problema y en este caso el de profundidad es ligeramente mejor.

Respecto a la corrección, se realizaron distintas pruebas para comprobar en primer lugar que a mano y con los algoritmos salía lo mismo. Cuando esto se comprobó se probaron planificaciones de venta con más pedidos y estaciones obteniendo en todos los casos los 3 algoritmos los mismos resultados óptimos, por tanto se confirma que todos los algoritmos son correctos.