Memoria de Backtracking y Branch and Bound

Rubén Morales Pérez Francisco Javier Morales Piqueras
Bruno Santindrian Manzanedo Ignacio de Loyola Barragan Lozano
Francisco Leopoldo Gallego Salido

7 de junio de 2016

Índice

1. División en dos equipos **2**
1.1. Enunciado . . . 2
1.2. Explicación . . . 2
1.3. Eficiencia . . . 2
1.4. Comparando ambos algoritmos . . . 3
1.4.1. Tiempos . . . 4
1.4.2. Diferencia de nivel . . . 5
1.4.3. Diferencia de jugadores . . . 6
1.5. Conclusión . . . 6

2. Viajante de comercio **7**
2.1. Enunciado . . . 7
2.2. Explicación . . . 7
2.3. Eficiencia . . . 8
2.3.1. Eficiencia del backtracking . . . 8
2.3.2. Eficiencia de las cotas . . . 8
2.4. Comparativa . . . 8
2.4.1. Cortes producidos . . . 8
2.4.2. Colas con prioridad . . . 9
2.4.3. Nodos procesados . . . 9
2.4.4. Tiempos . . . 9
2.5. Conclusión . . . 9

3. Bibliografia **9**

1. División en dos equipos

1.1. Enunciado

Se desea dividir un conjunto de n personas para formar dos equipos que competirán entre sí. Cada persona tiene un cierto nivel de competición, que viene representado por una puntuación (un valor numérico entero). Con el objeto de que los dos equipos tengan un nivel similar, se pretende construir los equipos de forma que la suma de las puntuaciones de sus miembros sea lo m ás similar posible. Diseña e implementa un algoritmo vuelta atrás para resolver este problema. Realizar un estudio empírico de la eficiencia de los algoritmos.

1.2. Explicación

Para simplificar el problema y darle mayor flexibilidad identificamos a cada jugador única y exlusivamente por su puntuación, aunque pueda haber dos jugadores con la misma puntuación. Esto permite que si hay dos jugadores con nivel de competición x e y con $x = y$ y cada uno está en un equipo podrían intercambiar sus posiciones sin descompensar los equipos.
Tenemos n jugadores con puntuación respectiva x_i $\forall i \in I = [1, 1000] \cap \mathbb{N}$. Si inicialmente tuviésemos los niveles de los jugadores en otro rango $J = [a, b]$ $a, b \in \mathbb{N} : a < b$ podemos dejarlo así, ya que el rango de los niveles no afecta al problema, salvo que apliquemos una transformación que reduzca el rango, ya que podríamos perder información al aproximar los números a enteros.
En cualquier caso la transformación al dominio del problema del intervalo J es: $\forall j \in J$ aplicamos una función $F : [a, b] \to [1, 1000] \cap \mathbb{N}$, teniendo en cuenta que $E(x)$ es la función parte entera:

$$F(x) = \begin{cases} E\left(\frac{x-a}{b-a} \cdot 1000\right) + 1 & \text{si } \left(\frac{x-a}{b-a} \cdot 1000\right) < E\left(\frac{x-a}{b-a} \cdot 1000\right) + 1/2 \\ E\left(\frac{x-a}{b-a} \cdot 1000\right) & \text{en otro caso} \end{cases}$$

1.3. Eficiencia

En cuanto a la eficiencia de este algoritmo, podemos representar los n jugadores como un vector de booleanos, donde si un jugador toma el valor *true* está en un equipo y si toma el valor *false* está en otro. De esta forma observamos que podemos representar la división por equipos en un único vector. Tomando el primer jugador, puede estar en un equipo o en otro, 2 combinaciones. Cogemos el siguiente jugador, puede estar en uno u otro, independientemente de donde estuviese el anterior, 2^2 combinaciones. El tercer jugador vuelve a tener 2 opciones, independientes del resto, 2^3 combinaciones.
Al tratarse de un algoritmo de vuelta atrás (backtraking) tenemos que recorrer todas las opciones, por lo que el problema queda representado como un árbol binario donde cada nodo tiene dos opciones, estar en un equipo o en otro.

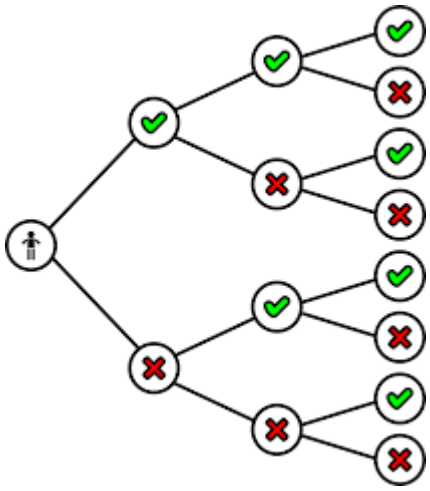

Figura 1: Distribución de equipos

De aquí deducimos la eficiencia del algoritmo con respecto al número total de jugadores $O(n) = 2^n$. Llegados a este punto nos podríamos plantear si podemos mejorar el algoritmo, curiosamente una forma de mejorarlo será ponerle una restricción, ya que nos ahorrará cálculos. La restricción nombrada será forzar que el número de jugadores en los dos equipos sea parecido, la diferencia de jugadores entre ambos equipos no será mayor de 1. Esto hace que si hay un número de jugadores par ambos equipos siempre tengan el mismo número de jugadores. Ahorramos comprobaciones, pero el algoritmo sigue teniendo eficiencia $O(n) = 2^n$ ya que pasaremos por todas las opciones, pero cuando la combinación en la que estemos no cumpla esa restricción la ignoraremos, sin hacer los cálculos asociados. La ventaja de este planteamiento se verá más adelante.

1.4. Comparando ambos algoritmos

El rango de jugadores usado ha sido $I = [3, 26] \cap \mathbb{N}$, vemos como el estudio empírico corrobora las anteriores deduciones. Hemos ajustado funciones $f(n) = a \cdot 2^n \ a \in \mathbb{R} \ \forall n \in I$. Sus coeficientes r^2 son 1 en ambos casos (hay pocos elementos debido a la eficiencia) y el término a está especificado en las gráficas, aunque también se encuentra en el archivo *Equipos/Datos/ajustes.txt*.

1.4.1. Tiempos

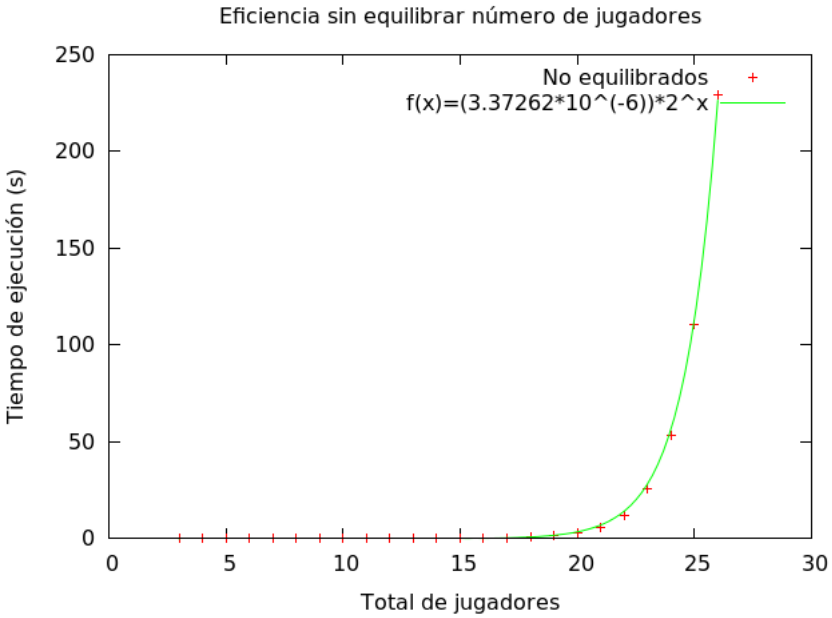

Figura 2: Distribución de equipos sin equilibrar

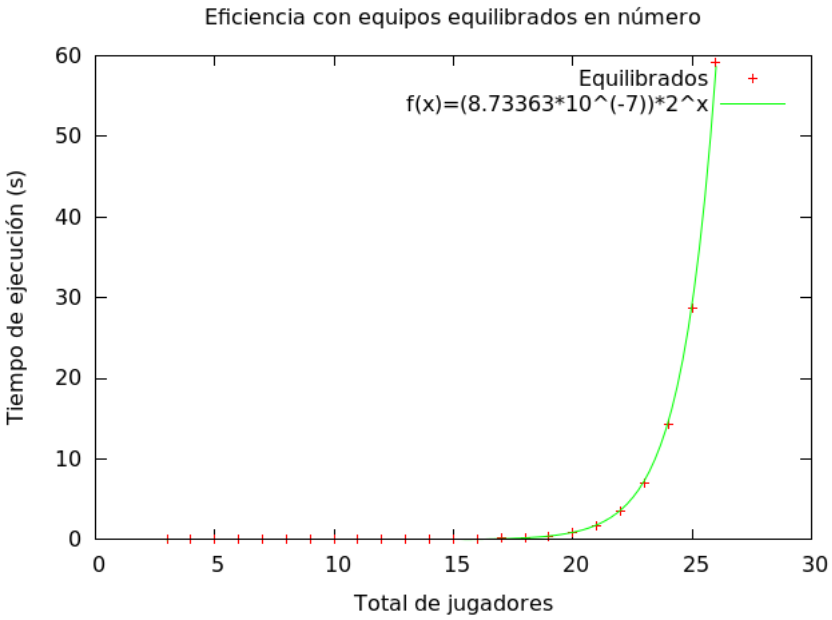

Figura 3: Distribución de equipos equilibrados en número

Comparando ambos tiempos:

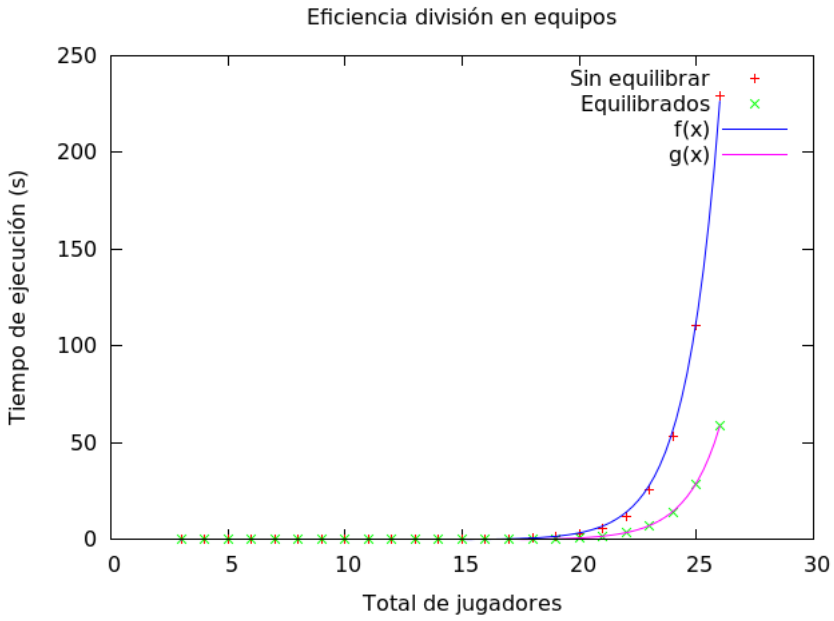

Figura 4: Comparativa de tiempo

1.4.2. Diferencia de nivel

Recordemos que un jugador tiene un nivel entre 1 y 1000, es lógico pensar que con pocos jugadores el margen de optimización del algoritmo es menor debido al gran peso que toma la aleatoriedad con la que se determinan los niveles.

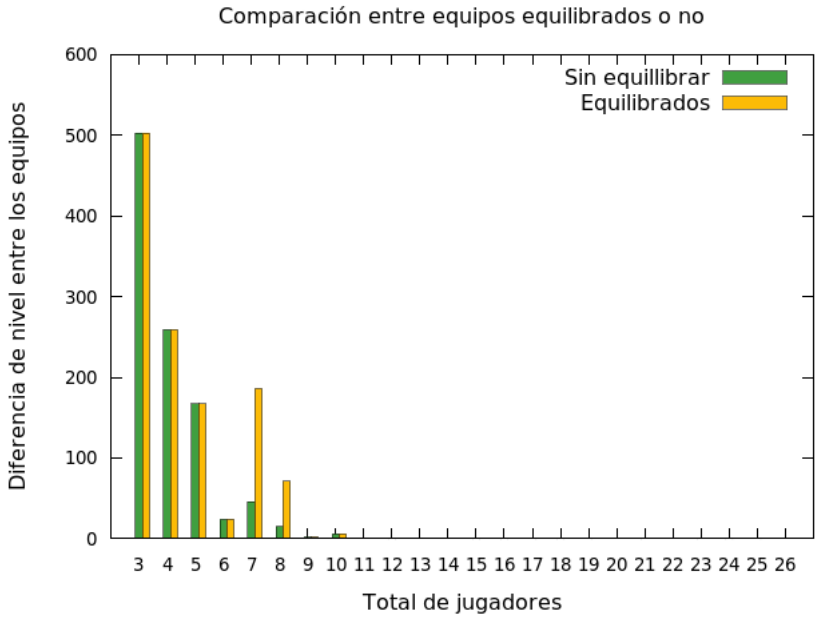

Figura 5: Diferencia de nivel en los equipos

1.4.3. Diferencia de jugadores

El primer algoritmo tiende a desajustar el número de jugadores, llegando a tener los equipos una diferencia de 4 jugadores cuando siendo 24 en total.

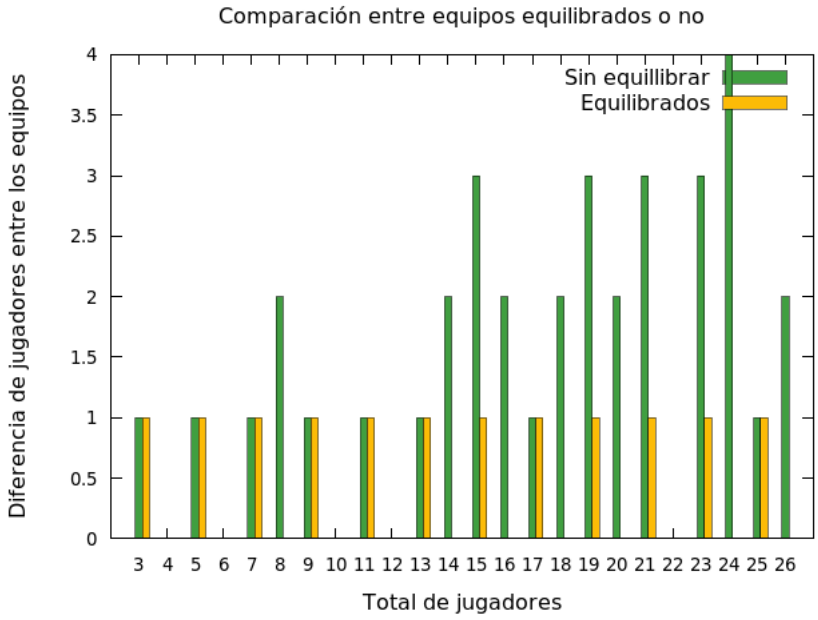

Figura 6: Diferencia en número de jugadores

1.5. Conclusión

Al ver esta eficiencia exponencial, si tenemos valores mayores de 30 nos plantearemos algoritmos más rápidos. Para ejecuciones con un número pequeño de elementos los algoritmos planteados nos sirven, pero si aumentamos el número esa fuerza bruta no es tan buena idea.
Una forma de reducir las constantes ocultas es intentar ahorrarnos comprobaciones. Podríamos podar del árbol de soluciones, lo que denominaremos "Branch and Bound". También podríamos conformarnos con que la diferencia de nivel fuese menor que un $n \in \mathbb{N}$, de forma que no seguiríamos recorriendo el árbol una vez encontrada una solución satistactoria. Otra opción para disminuir el tiempo es paralelizar el problema, podemos hacer que cada procesador compruebe una rama diviendo el tiempo total.
Como hemos comprobado, el segundo algoritmo iguala al primero en eficacia, pero lo supera en eficiencia. Esto nos hace pensar que puede no ser necesario siempre recorrer el árbol de soluciones para asegurarnos la optimalidad.

2. Viajante de comercio

2.1. Enunciado

El problema del viajante de comercio ya se ha comentado y utilizado en la práctica sobre algoritmos voraces, donde se estudiaron métodos de este tipo para encontrar soluciones razonables (no óptimas necesariamente) a este problema. Si se desea encontrar una solución óptima es necesario utilizar métodos más potentes (y costosos), como la vuelta atrás y la ramificación y poda, que exploren el espacio de posibles soluciones de forma más exhaustiva.
Así, un algoritmo de vuelta atrás comenzaría en la ciudad 1 (podemos suponer sin pérdida de generalidad, al tratarse de encontrar un tour, que la ciudad de inicio y fin es esa ciudad) e intentaría incluir como parte del tour la siguiente ciudad aún no visitada, continuando de este modo hasta completar un tour. Para agilizar la búsqueda de la solución se deben considerar como ciudades válidas para una posición (ciudad actual) sólo aquellas que satisfagan las restricciones del problema (en este caso ciudades que aún no hayan sido visitadas). Cuando para un nivel no queden más ciudades válidas, el algoritmo hace una vuelta atrás proponiendo una nueva ciudad válida para el nivel anterior.
Para emplear un algoritmo de ramificación y poda es necesario utilizar una cota inferior: un valor menor o igual que el verdadero coste de la mejor solución (la de menor coste) que se puede obtener a partir de la solución parcial en la que nos encontremos.
Una posible alternativa sería la siguiente: como sabemos cuáles son las ciudades que faltan por visitar, una estimación optimista del costo que aún nos queda será, para cada ciudad, el coste del mejor (menor) arco saliente de esa ciudad. La suma de los costes de esos arcos, más el coste del camino ya acumulado, es una cota inferior en el sentido antes descrito.
Para realizar la poda, guardamos en todo momento en una variable C el costo de la mejor solución obtenida hasta ahora (que se utiliza como cota superior global: la solución óptima debe tener un coste menor o igual a esa). Esa variable puede inicializarse con el costo de la solución obtenida utilizando un algoritmo voraz (como los utilizados en la práctica 2). Si para una solución parcial, su cota inferior es mayor que C entonces se puede realizar la poda.
Como criterio para seleccionar el siguiente nodo que hay que expandir del árbol de búsqueda (la solución parcial que tratamos de expandir), se empleará el criterio LC o “más prometedor". En este caso consideraremos como nodo más prometedor aquel que presente el menor valor de cota inferior. Para ello se debe de utilizar una cola con prioridad que almacene los nodos ya generados (nodos vivos).
Además de devolver el costo de la solución encontrada (y en su caso el tour correspondiente), se deben de obtener también resultados relativos a complejidad: número de nodos expandidos, tamaño máximo de la cola con prioridad de nodos vivos, número de veces que se realiza la poda y el tiempo empleado en resolver el problema.
Las pruebas del algoritmo pueden realizarse con los mismos datos empleados en la práctica 2 (teniendo en cuenta que el tamaño de problemas que se pueden abordar con estas técnicas es mucho más reducido que con los métodos voraces). La visualización de las soluciones también puede hacerse de la misma forma que en la práctica 2 (usando gnuplot).

2.2. Explicación

Implementaremos dos versiones del algoritmo con diferentes formas de calcular la cota inferior.

- La primera versión calculará la cota de la forma propuesta en el enunciado. Esto se traduce en utilizar la suma de los elementos mínimos de cada fila (obviando las filas que corresponden a los nodos que ya hemos recorrido) de la matriz de adyacencia.

- La segunda versión es una mejora de la primera. Dado que las matrices con las que vamos a trabajar son simétricas puede ocurrir que al elegir el mínimo de cada fila utilicemos dos elementos simétricos (escogemos dos veces el mismo arco), por tanto la cota resultante no es todo lo alta que podría ser (el grafo puede no ser conexo). Para solucionar esto, si a la hora de elegir un elemento de una fila vemos que ya hemos utilizado su simétrico, escogeremos el siguiente elemento mínimo, como resultado obtendremos que nuestra cota inferior es un grafo conexo cuyo recorrido es menor que el del ciclo que queremos hallar.

2.3. Eficiencia

2.3.1. Eficiencia del backtracking

Tenemos que evaluar todos los resultados de la forma $(1, X, 1)$ donde los $X = (x_1, x_2, ..., x_n)$ son permutaciones del conjunto de nodos sin el primero ($N - \{1\} = \{2, 3, ..., n\}$), por tanto tendremos que probar $(n-1)!$ posibles resultados.
La eficiencia será $O(n!)$.

2.3.2. Eficiencia de las cotas

Para encontrar la primera cota tenemos que recorrer los elementos de todas las filas del la matriz de adyacencia, por tanto su eficiencia será $O(n^2)$.

Para hallar la segunda cota, además de recorrer todos los elementos de la matriz, tendremos que asegurarnos de que no elijamos dos simétricos, por tanto la eficiencia será
$n * (n + k \cdot$ "*comprobar si hemos usado el simetrico*") ($k \leq n$ son las veces que tenemos que elegir otro valor).
Como siempre sabemos en que fila está el elemento simétrico de otro y solo guardamos un elemento por fila, la comprobación es $O(1)$, y por lo tanto la eficiencia del calculo de la cota será $O(n^2)$, del mismo orden que la primera cota.

2.4. Comparativa

Comparamos los tiempos gastados con las diferentes formas de calcular cotas inferiores para poda.
Las diferentes funciones para conseguir una cota inferior hacen que en un mismo mapa el algoritmo de ramificación y poda varíe. Estas variaciones se traducen en los nodos expandidos, la mayor cola de nodos vivos, la cantidad de cortes que ha hecho cada uno y el tiempo.

2.4.1. Cortes producidos

Empecemos con la cantidad de cortes producidos. Esta información puede no ser representativa de la magnitud real de los cortes aplicados. Dados dos algoritmos X, Y que han hecho x_i, y_i cortes respectivamente con $x_1 > y_1$ no nos está diciendo que el primero sea más rapido. El primer algoritmo podría haber recorrido más nodos y hacer los cortes en niveles del árbol más profundos.
Como esta gráfica solamente tiene en cuenta el número si el primer algoritmo ha podido cortar muchos nodos pero el segundo hacer solamente un corte que sea mejor que todos los cortes anteriores por haberse hecho en un nivel superior.

Figura 7: Cortes efectuados

2.4.2. Colas con prioridad

A continuación exponemos el mayor tamaño alcanzado por la cola con prioridad. Esta información nos puede dar una idea de lo bueno que es nuestro algoritmo para el cálculo de una cota inferior, es un dato mucho más valioso que la cantidad de cortes efectuados.
En los algoritmos usados nos es útil este dato, por ser algoritmos que solamente difieren en la función de la cota inferior. Hay versiones de ramificación y poda para el problema del viajante de comercio que no usan por ejemplo la aproximación greedy inicial, por lo que ha podido sobrecargar la cola con prioridad mientras busca la primera solución con la que podará después.

Figura 8: Cola con prioridad más grande

2.4.3. Nodos procesados

Otro dato interesante son los nodos que ha procesado cada algoritmo, pero esto no nos asegura que tarde menos tiempo, ya que un algoritmo con más nodos expandidos que otro ha podido usar una función para calcular las cotas inferiores demasiado costosa.

Figura 9: Nodos expandidos

2.4.4. Tiempos

Todos estos datos son interesantes académicamente, pero en la práctica el tiempo es el dato que mayor peso tendrá (además del uso de memoria) para elegir el algoritmo más eficiente o decantarnos por una alternativa greedy.

Figura 10: Tiempo algoritmo 1 y su ajuste

Figura 11: Tiempo algoritmo 2 y su ajuste

Figura 12: Comparativa de tiempos

2.5. Conclusión

Aunque los algoritmos de ramificación y poda nos aseguraran la optimalidad a problemas con tiempos de ejecución exponenciales ahorrándonos pasos, no resultan prácticos cuando el tamaño del problema crece. Los algoritmos greedy estudiados anteriormente nos pueden dar una aproximación más que razonable para el problema.

3. Bibliografia