Práctica 4. Algoritmos Backtracking - Cena de Gala

Noelia Escalera Mejías Alejandro Menor Molinero
Javier Núñez Suárez Adra Sánchez Ruiz
Jesús Torres Sánchez

18 de mayo de 2019

1. Descripción del problema

Podemos describir el problema de la Cena de Gala de la siguiente forma:

Se va a realizar una cela de gala a la que van a asistir n invitados. Cada invitado va a tener sentados junto a él a dos comensales (uno a su izquierda y otro a su derecha). Dependiendo de las características del invitado, existe una cierta conveniencia de que dos invitados se sienten juntos, definiendo la conveniencia global de la mesa como la suma de todos los niveles de conveniencia entre cada pareja de invitados sentados de forma contigua. El objetivo principal es conseguir sentar a los invitados de forma que el nivel de conveniencia global sea el máximo.

Por tanto, nuestro algoritmo debe ser capaz de encontrar la asignación de invitados que maximice la afinidad de cada comensal con los otros dos comensales que tiene a su lado.

2. Representación del problema

La descripción del problema nos puede recordar mucho al problema del viajante de comercio. De hecho, se podría considerar el mismo problema con diferente enunciado.

Aquí en vez de ciudades, tenemos comensales. En vez de distancias, afinidades. En vez de minimizar, maximizar.

Para un número n de comensales tenemos una tupla $(x_0, ..., x_{n-1})$ de "sitios" en la mesa. Siendo la afinidad global de la configuración: $\sum_{i=1}^{n-1} afinidad(x_i, x_{i-1}) + afinidad(x_i, x_{i+1})$, además de sumar la afinidad de x_0 y x_{n-1} y viceversa.

3. Restricciones

Ya hemos visto como representar las decisiones que vamos haciendo (y deshaciendo) en el algoritmo. Es momento de hablar sobre que valores pueden tomar los elementos de la tupla.

Restricciones explícitas: Cada x_i de la tupla que representa un estado del problema puede tomar un valor de 0 a $n-1$ o *null* si todavía no hemos llegado a esa decisión. Obviamente, en un estado solución no es posible este valor nulo, tenemos que sentar a todos los comensales en la mesa.

En cuanto a las **restricciones implícitas**: Una solución al problema tiene que ser una permutación de todos los comensales de 0 a $n-1$, de forma que no podamos sentar al mismo comensal en dos sitios distintos de la mesa.

4. Descripción de las cotas usadas

En nuestra solución, hemos usado dos tipos de cotas: una **cota global**, que determina la máxima afinidad global encontrada hasta el momento. Si aún no se ha encontrado ninguna solución, estará inicializada de acuerdo a una versión Greedy del algoritmo.

Por otro lado, también necesitamos una **cota local** (tal y como se ha mencionado anteriormente), necesaria para podar una rama en el caso de que la cota local (cota optimista) no supere a la cota global. La poda se puede realizar debido a que, si una cota local optimista genera peores resultados que una solución ya encontrada, esa rama no va a contener una solución mejor, por lo que no tiene sentido recorrerla.

Como detallaremos más adelante, para el cálculo de las afinidades optimistas de los comensales que quedan por seleccionar. Usaremos el promedio de sus dos mejores afinidades, como hemos visto de forma análoga en clase para el problema del viajante de comercio.

5. Representación de la solución

A la hora de representar el problema, hemos usado un TDA *Solucion*, para implementar nuestro algoritmo de *backtracking* de una manera clara.

- **Información almacenada en el TDA** *Solucion*:
  - x: almacena la solución generada en cada momento. Se representa por medio de un vector de enteros (conceptualmente es un vector circular, representando la mesa para cenar), en el que cada componente toma el valor que representa el comensal que está sentado ahí.

- *afinidades*: implementadas con una matriz, en la que cada elemento m[i][j] tiene asociado la conveniencia de sentar juntos al invitado i con el invitado j. Por tanto, deben coincidir m[i][j] con m[j][i].
- n: es el número de comensales.
- *solucionOptima*: almacena la mejor solución encontrada hasta el momento, es decir, la que tiene una mayor afinidad global.
- *comensalesYaSentados*: es un vector de valores booleanos, su componente i será *true* si el comensal i está ya sentado. Este vector nos permite comprobar la factibilidad en tiempo constante.
- *afinidadActual*: contiene la afinidad acumulada para los comensales seleccionados en ese momento. Se usa para el cálculo de la afinidad de una solución obtenida y de la cota local de un nodo.
- *Afinidades optimistas*: es un vector en el que su elemento i es el promedio de sus dos mejores afinidades con otros dos comensales.
- *AfinidadAcabarMesa*: Almacena la mejor afinidad del primer comensal seleccionado (que forzamos al comensal 0, para evitar soluciones equivalentes). La usamos para el cálculo de la cota local de un nodo.

- **Operaciones del TDA Solucion**:
  - *Constructor*: realiza una serie de operaciones:
    - Inicializa las variables del TDA a sus valores por defecto.
    - Seleccionamos en primer lugar el comensal 0 para no generar soluciones equivalentes y reducir así el número de nodos hoja de $n!$ a $(n-1)!$.
    - Inicializamos *solucionOptima* y $maximoValor$ a partir de la versión Greedy (no óptima) del algoritmo.
    - Calcula las $afinidadesOptimistas$ y la $afinidadAcabarMesa$ para evitar hacerlo posteriormente en el cálculo de las cotas locales. El proceso de estimación es bastante sencillo: para cada comensal, buscamos las dos mayores afinidades que tenga y le asociamos el promedio de ambas.
  - *hemosTerminado(k)*: devuelve si ya hemos seleccionado el total de comensales ($k == n$).
  - *iniciaComp(k)*: inicializamos la componente k al valor NULO, que en nuestro caso es 0.
  - *sigValComp(k)*: asignamos a la componente k de la solución actual el siguiente valor ($x[k] + +$).
  - *factible(k)*: para determinar su factibilidad, debemos comprobar si el comensal $x[k]$ ya estaba sentado en otro sitio, y si la afinidad acumulada hasta el momento y la estimación optimista de los comensales que quedan por seleccionar, es mayor que la cota global. Si es factible, actualizamos la afinidad acumulada y el vector solución x.

- *todosGenerados(k)*: devuelve si se han generado todos los posibles comensales para $x[k], (x[k] == n)$.
- *procesaSolucion*: actualizamos la afinidad actual (sumándo la afinidad del último y el primero, para "cerrar" la mesa), la cota global y la solución óptima en el caso de que la afinidad actual sea mayor que la mejor solución obtenida.
- *imprimeSolucion*: la utilizamos para comprobar que nuestro algoritmo funciona correctamente, imprime la mejor solución obtenida y su afinidad.
- *vueltaAtras(k)*: deshace los cambios en la afinidadActual y en comensalesYaSentados (necesario para cuando damos una vuelta atrás en el algoritmo de Backtracking).
- *cotaLocal(k)*: genera la cota local a partir de k. Para ello, obtiene la suma de las afinidadesOptimistas asociadas a los comensales que quedan por seleccionar.

6. Algoritmo Backtracking

El algoritmo sigue el esquema básico de los algoritmos Backtracking vistos en clase:

```
void backtracking(Solucion & sol, int k = 0){
        if (sol.hemosTerminado(k))
                sol.procesaSolucion();
        else{
                sol.iniciaComp(k);
                sol.sigValComp(k);
                while (!sol.todosGenerados(k)){
                        if (sol.factible(k)){
                                backtracking(sol, k+1);
                                sol.vueltaAtras(k);
                        }
                        sol.sigValComp(k);
                }
        }
}
```

7. Estudio empírico

Hemos ejecutado el algoritmo con distintos tamaños: desde un problema de tamaño 2 a uno de tamaño 22. Se nos ha hecho imposible probar con tamaños más grandes ya que el tiempo aumentaba a pasos agigantados como se puede apreciar en la siguiente gráfica:

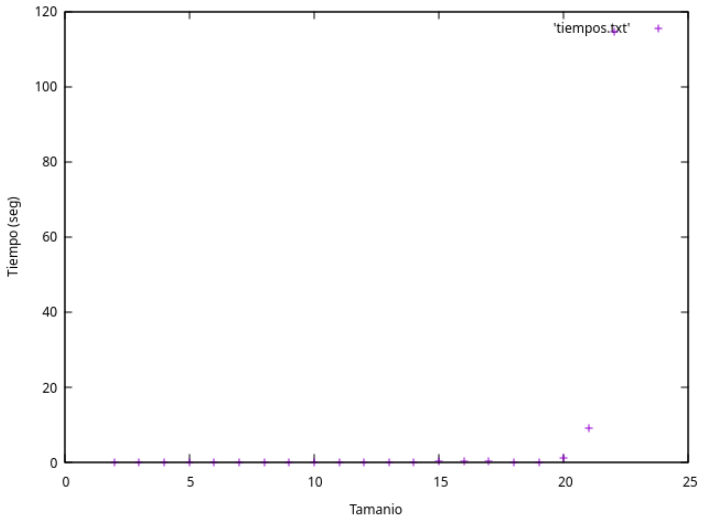

Figura 1: Tiempos de ejecución del algoritmo, desde tamaño 2 a 22

En la siguiente figura podemos apreciar los tiempos para tamaños de 2 a 20, que en la anterior imagen no se apreciaban muy bien. Como vemos no varían mucho.

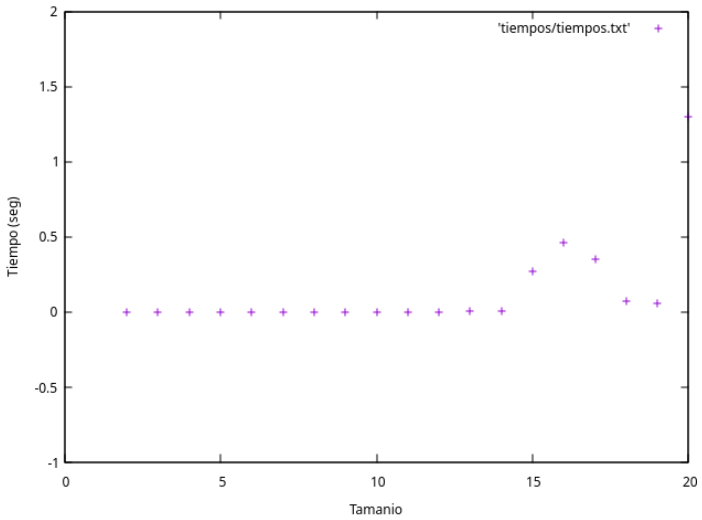

Figura 2: Tiempos de ejecución del algoritmo con zoom en X en [0-20] y zoom en Y en [-1,2]

Nuestro algoritmo tiene una eficiencia teórica de $(n-1)!$. Si ajustamos nuestra gráfica de tiempos a $f(x) = (ax-1)!$, obtenemos $a = 0{,}337065$ con un error

de 0,6055 %. He aquí la curva ajustada.

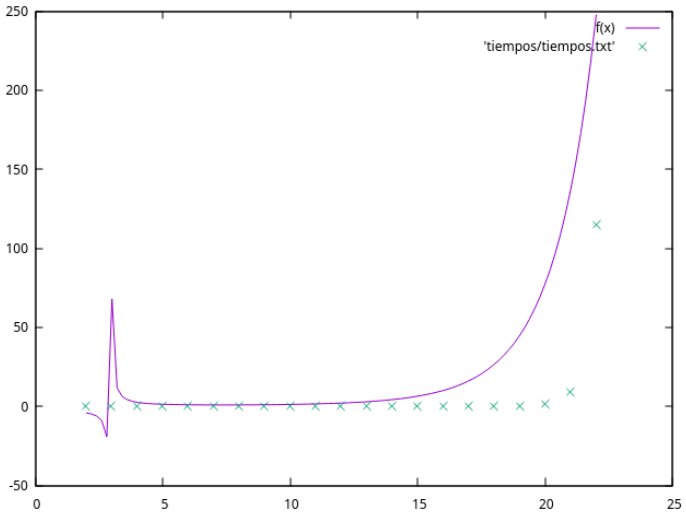

Figura 3: Tiempos de ejecución del algoritmo ajustados a f(x)=(n-1)!

8. Conclusiones

Los algoritmos de backtracking nos ayudan a alcanzar la solución óptima de un problema (cosa que con algunos algoritmos como los greedy no nos era posible). Sin embargo, esto nos da una penalización enorme en eficiencia, nuestro tamaño máximo de problema ha sido 22.