Práctica 4. Parte 2. Algoritmos ByB - Backtracking - Problema del viajante de comercio

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

26 de mayo de 2019

1. Descripción del problema

En este caso, hemos propuesto una solución para el problema del viajante de comercio implementando un algoritmo *Branch and Bound*. Además, hemos ejecutado la versión con técnica de *Backtracking* para comparar la eficiencia entre ambos algoritmos.

2. Cotas global y cota local

Para implementar el algoritmo de *Branch and Bound*, necesitamos dos cotas:

Por un lado tenemos la **cota global**, que determina la mínima distancia encontrada hasta el momento para un circuito generado. Si aún no se ha encontrado ninguna solución, estará inicializada de acuerdo a una versión *Greedy* del problema (en este caso, se ha usado la aproximación basada en Inserción).

Por otro lado, también necesitamos una **cota local**, que nos servirá a la hora de comprobar la factibilidad (criterio de poda) y en la extracción de los nodos en la cola, de modo que exploraremos antes los nodos más prometedores.

2.1. Cálculo de la cota local

La cota local consta de tres sumandos:

- Distancia actual: Para todas las ciudades ya seleccionadas, la suma de la distancia entre ellas (sin cerrar el circuito).
- La distancia mínima entre la última ciudad seleccionada y cualquiera de las ciudades no seleccionadas aún.
- Para cada ciudad no seleccionada, la mínima distancia entre ella y cualquiera de las otras no seleccionadas o la primera ciudad seleccionada.

3. Representación de la solución

A la hora de representar el problema, hemos usado un TDA *Solucion*, para implementar nuestro algoritmo de *Branch and Bound* de forma más clara y organizada.

- **Información almacenada en el TDA Solucion**:
  - x: almacena la solución generada hasta el momento. Se representa por medio de un vector de enteros, de modo que la componente x_i del mismo será la ciudad iésima que visitaremos.
  - n: es el número de ciudades.
  - *distancias*: representadas con una matriz, en la que cada elemento d[i][j] tiene asociado la distancia entre la ciudad i y la ciudad j. Por tanto, deben coincidir d[i][j] con d[j][i].
  - *cotaLocal*: contiene la cota local de acuerdo a las ciudades seleccionadas hasta el momento. La hemos descrito con más detalle en la sección anterior.
  - *distanciaActual*: es la distancia acumulada para las ciudades ya seleccionadas hasta el momento.
  - *ciudadesYaSeleccionadas*: representadas como un vector de booleanos, su componente x_i será *true* si la ciudad i está ya seleccionada y por tanto almacenada en el vector solucion x.
  - **Atributos de clase**
    - *mejorSolución*: es un vector de n elementos que representa la mejor solución encontrada hasta el momento. Este valor se inicializa a partir de una solución Greedy del problema.
    - *cotaGlobal*: almacena la distancia de la mejor solución descrita en el apartado anterior.
- **Operaciones del TDA Solucion**:
  - *Constructor*: realiza una serie de operaciones:
    - Inicializa las variables del TDA a sus valores por defecto. Debemos tener en cuenta que, a la hora de inicializar la matriz distancias, el primer elemento es el $d[1][1]$, por lo que debemos añadirle una fila y columna basura para poder acceder con índices que empiecen por 1.
    - Seleccionamos en primer lugar la ciudad 0 para no generar soluciones equivalentes y reducir así el número de nodos hoja de $n!$ a $(n-1)!$.
  - *factible()*: únicamente comprueba si la cotaLocal es menor que la cotaGlobal. No es necesario comprobar si el hijo estaba ya seleccionado o no, ya que esto se garantiza por la implementación de nuestra función $generaHijos()$.

- *generaHijos()*: este método devuelve un vector con todos los hijos del nodo que lo ejecuta, al añadir cualquiera de las ciudades **no seleccionadas** al vector solución. En el caso de que a la solución generada le falte solamente una ciudad (y la primera), se completa con la ciudad en cuestión. Actualizamos correctamente los atributos de los nodos hijos (ciudadesYaSeleccionadas, distanciaActual, cotaLocal, etc...)
- *actualizarCotaLocal()*: Se llama desde *generaHijos*() en caso de que no quede sólamente una ciudad más por añadir (como hemos explicado antes), hemos escrito esta función para extraer la lógica del método *generaHijos*() y poder hacer un poco más mantenible el código.
- *esSolucion()*: devuelve si hemos llegado a una solución, es decir, si el tamaño del vector solución coincide con el número de ciudades a explorar. En ese caso, siempre se actualiza la cota global y la mejor solución encontrada. Esto es así, porque en nuestro algoritmo ByB, comprobamos la factibilidad al sacar el nodo de la cola, y antes de llamar a *esSolucion*(). De esta forma, si es factible y es solución, significa que es una solución mejor que la mejor encontrada, ya que en la variable cota local se almacena la distancia real si es un nodo hoja.
- *operator 'menor que'*: necesario para almacenar ordenádamente las soluciones en la cola con prioridad que utilizamos en el algoritmo ByB. Para explorar primero los nodos más prometedores, estos se ordenan en la cola de forma ascendente en función de su cota local.
- *inicializarCotaGlobal*: inicializa la cotaGlobal y la mejorSolucion a mediante una solución Greedy del problema (en nuestro caso, hemos usado la aproximación basada en Inserción).
- *obtenerSoluciónÓptima*: devuelve la solución óptima.

4. Algoritmo ByB

El algoritmo Branch and Bound sigue el siguiente esquema:

```
void branchAndBound (Solucion sol){
  priority_queue<Solucion> cola;
  cola.push(sol);
  bool fin = false;
  n_nodos = tamq = npoda = 0;

  while (!cola.empty() && !fin){
        Solucion enodo = *cola.begin();
        cola.pop();

        fin = !enodo.factible();
```

```
        if (!fin){
                vector<Solucion> hijosDelEnodo =
                                enodo.generaHijos();

                for (Solucion hijo : hijosDelEnodo){
                        if (hijo.factible()){
                                if (!hijo.esSolucion())
                                   cola.push(hijo);
                        }
                        else
                                npoda++
                }
        }
        else
                npoda += cola.size();
        if (cola.size() > tamq)
                tamq = cola.size();

        n_nodos++;
  }
}
```

5. Estudio empírico

Hemos decidido comparar el algoritmo Branch & Bound con una versión Backtracking que también hemos construido ya no solo en implementación, sino también en tiempo.

Además, hemos preparado unos scripts y escrito un *Makefile*, para compilar, ejecutar y pintar para los dos algoritmos los siguientes mapas con estos comandos:

- **make probar10:** Algoritmo Branch and Bound y el "mapa" Ulysses10
- **make probar14:** Algoritmo Branch and Bound y el "mapa" Ulysses14
- **make probar16:** Algoritmo Branch and Bound y el "mapa" Ulysses16
- **make probarburma:** Algoritmo Branch and Bound y el "mapa" Burma14
- **make probar10back:** Algoritmo Backtracking y el "mapa" Ulysses10
- **make probar14back:** Algoritmo Backtracking y el "mapa" Ulysses14
- **make probar16back:** Algoritmo Backtracking y el "mapa" Ulysses16

- **make probarburmaback:** Algoritmo Backtracking y el "mapa" Burma14

He aquí una comparativa de los tiempos en los distintos mapas:

Mapa	Branch & Bound	Backtracking
Ulysses10	0.0561154	0.032728
Ulysses14	27.554	38.1479
Burma14	2.30142	4.7288
Ulysses16	745.77	1232.72

También hemos hecho un estudio del número de podas y nodos expandidos en los distintos mapas con el algoritmo Branch & Bound.

Mapa	Nodos expandidos	Tamaño máximo de la cola	Número de podas
Ulysses10	1937	1673	6227
Ulysses14	655875	378692	3050716
Burma14	45714	18446	248076
Ulysses16	13504373	10020741	71153628

Para obtener el número de podas, simplemente hemos usado una variable *npoda* que se incrementa en una unidad cuando encontramos un hijo no factible y que se incrementa en el tamaño de la cola de prioridad cuando encontramos el mejor nodo de ésta y no es solución. Esto último ocurre ya que cuando sacamos de la cola el nodo más prometedor y este tiene una cota local mayor que la global, no tiene sentido seguir explorando, entonces todos los nodos restantes de la cola que no son explorados son podados también.

Para obtener el número de nodos expandidos hemos usado una variable *n_nodos* que se incrementa en una unidad cada vez que exploramos un nuevo nodo.

Por supuesto, también hemos dibujado las diferentes soluciones a los mapas, he aquí los circuitos hallados:

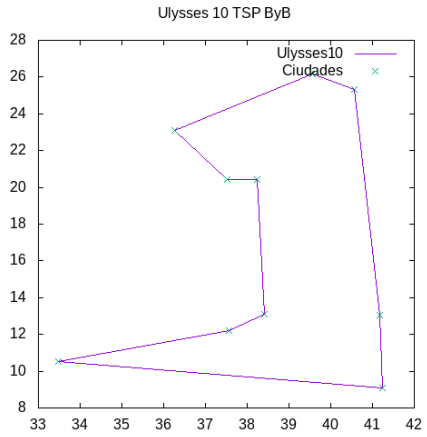

Figura 1: Circuito Ulysses 10 con Branch & Bound. Distancia: 45

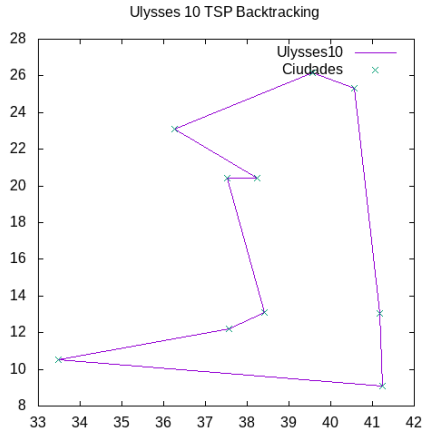

Figura 2: Circuito Ulysses 10 con Backtracking. Distancia: 45

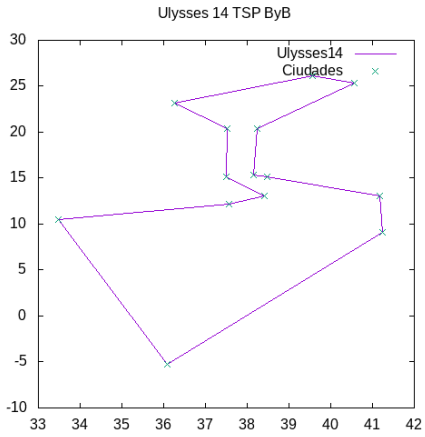

Figura 3: Circuito Ulysses 14 con Branch & Bound. Distancia: 68

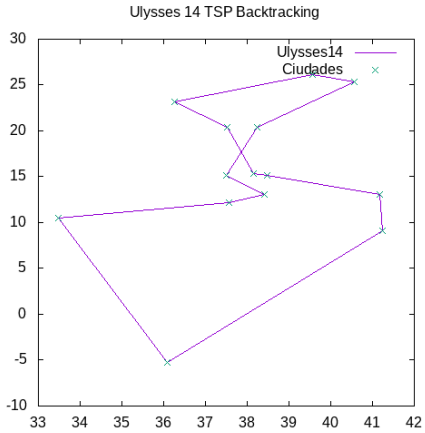

Figura 4: Circuito Ulysses 14 con Backtracking. Distancia: 68

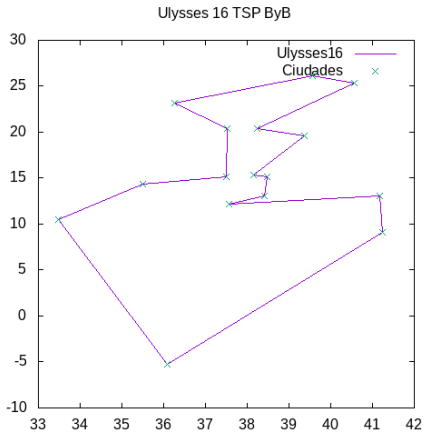

Figura 5: Circuito Ulysses 16 con Branch & Bound. Distancia: 71

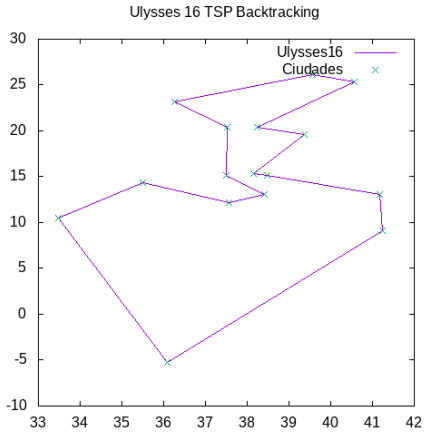

Figura 6: Circuito Ulysses 16 con Backtracking. Distancia: 71

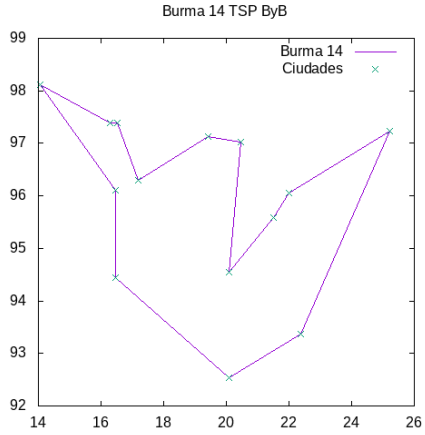

Figura 7: Circuito Burma 14 con Branch & Bound. Distancia: 30

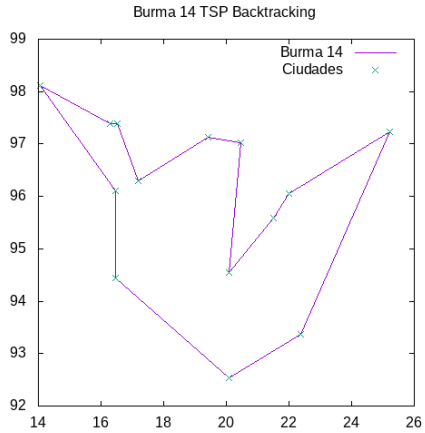

Figura 8: Circuito Burma 14 con Backtracking. Distancia: 30

6. Algoritmo Backtracking VS ByB

Como vemos en el estudio anterior, al explorar ramas más prometedoras, el algoritmo Branch & Bound nos da unos tiempos mucho más bajos. No obstante, la complejidad en espacio es mucho mayor. Como ambos algoritmos encuentran una solución óptima, la distancia del circuito es la misma para ambos recorridos, aunque en algunos casos no se halle el mismo camino.