Algorítmica - Practica 4: Problema del viajante de comercio

A. Herrera, A. Moya, I. Sevillano, J.L. Suarez

26 de mayo de 2015

Contents

1 Organización de la práctica

La práctica 4 abarca de nuevo el problema del viajante de comercio, aunque en esta ocasión queremos encontrar la solución óptima mediante el empleo de ramificación y poda (*Branch and Bound*) y compararla con un programa que emplee vuelta atrás (*Backtracking*).

Para el problema se sigue la siguiente estructura:

- Enunciado del problema
- Resolución teórica del problema (con una subsección por algoritmo)
- Análisis empírico. Análisis de la eficiencia híbrida

En este último apartado se proporcionan gráficas con los resultados de los algoritmos y un análisis de la eficiencia híbrida para los mismos.

Los algoritmos se han ejecutado sobre un ordenador con las siguientes características:

- **Marca:** Toshiba
- **RAM:** 8 GB
- **Procesador:** Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz

Los resultados de las ejecuciones, las gráficas y los pdf asociados se pueden encontrar en GitHub, así como el programa construido para resolver los problemas del TSP

2 Problema 4

2.1 Enunciado del problema

Construir un programa que utilice la técnica de ramificación y acotación para resolver el problema del viajante de comercio en las condiciones descritas anteriormente, empleando una determinada función de acotación.

Opcionalmente, construir también un programa que utilice vuelta atrás, pero utilizando también la función de acotación descrita anteriormente, y realizar un estudio experimental comparativo con el algoritmo de ramificación y poda.

2.2 Solución teórica

En este apartado vamos a buscar soluciones teóricas al problema del viajante de comercio mediante el empleo de dos algoritmos, basados en las técnicas *Branch and Bound* y *Backtracking* respectivamente.

Para ambos debemos de tener en cuenta que debemos usar una función de acotación, que permita determinar qué ramas pueden ser correctas para una solución óptima y cuáles no merecen la pena resolver. Así, para obtener esta acotación empleando la técnica *Branch and Bound* debemos definir una función estimar, que permita obtener la estimación de la mejor solución con el empleo de unos u otros nodos.

2.2.1 Algoritmo Branch and Bound

Para el empleo de esta técnica primero tenemos que tener bien claro en qué consiste.

La técnica *Branch and Bound* es una técnica de ramificación y poda, de tal forma que se cuenta con una función que estima cuáles serán los valores obtenidos según se empleen unos nodos u otros. Estos permiten guardar en una variable la mejor solución considerada en cada momento, de tal forma que si la cota inferior de una solución parcial es superior a la solución que tenemos guardada en ese variable, podemos podar esa rama.

¿Cómo aplicamos esta técnica en el caso del viajante del comercio? El primer punto destacable es la definción de la función que hemos elegido para realizar la estimación.

Para cada nodo en el que nos encontremos, donde un nodo representa una solución parcial obtenida, la función de estimación nos va a dar la menor longitud que podríamos esperar para un camino con el recorrido que ya llevamos a nuestra espalda. Para ello, lo que hace esta función es, sobre el coste que ya llevamos acumulado en la solución de nuestro nodo, añade los costes mínimos que se podrían esperar de visitar cada una de las ciudades restantes no visitadas desde las distintas ciudades por visitar, teniendo en cuenta siempre que debemos cerrar el camino, esto es, añadiremos también el menor coste que podemos esperar de visitar las ciudades no visitadas desde la primera y la última ciudad de nuestro recorrido parcial.

De esta forma, lo que hacemos con esta función de estimación es obtener una cota inferior para la solución que tenemos en el nodo que estamos visitando. Es decir, tenemos la garantía de que el coste de nuestro recorrido va a ser por lo menos el obtenido con nuestra estimación. Por tanto, tendremos que en un principio los mejores candidatos a la solución óptima serán los que menor valor de estimación tengan. A estos nodos son a los que damos mayor prioridad: los visitaremos primero. En consecuencia, iremos obteniendo soluciones más o menos buenas que nos permitirán podar gran cantidad de ramas: cualquier estimación que supere el coste de la mejor solución obtenida ya no será necesario que se visite.

Veamos un pseudocódigo para la función de estimación:

Algoritmo Branch and Bound. Función estimación.

```
def estimar()
  #Inicializamos con el coste que llevamos de la solución
  estimacion = solucion_actual.coste

  # Añadimos las distancias mínimas entre las ciudades no
  # visitadas, para obtener el mínimo coste esperable
  # de dicha solución: la cota inferior
  for ciudad in noVisitadas
    estimacion += distancias.minimo(ciudad,noVisitadas)
  end

  # Cerramos el circuito (añadimos la estimación de
  # distancias de la penultima y primera ciudad ya visitadas)
  estimacion += distancias.minimo(solucion_actual.primeraVisitada,noVisitadas)
  estimacion += distancias.minimo(solucion_actual.ultimaVisitada,noVisitadas)

end
```

Una vez tenemos ya claro cual es el criterio empleado para llevar a cabo esta estimación, pasaremos a resolver el problema. Antes que nada, cabe destacar que como criterio para seleccionar el siguiente nodo que hay que expandir del árbol de búsqueda se empleará el criterio LC o "más prometedor" (donde el nodo más prometedor es el de menor valor de cota inferior). Para ello, se debe de utilizar una cola con prioridad que almacene los nodos ya generados. El resto del algoritmo simplemente pone en funcionamiento la idea ya explicada. Inicializamos la cola añadiendo el nodo raíz (la solución con la primera ciudad). Extraemos el nodo más prometedor de la cola y obtenemos sus nodos hijos. Si son hojas, comprobamos si la solución es buena y la actualizamos si se da el caso. Si no son hojas y su estimación es menor que la de la mejor solución obtenida, los añadimos de nuevo a la cola. Cuando no queden nodos en la cola el algoritmo terminará, habiendo dado con la solución óptima.

Por último, podemos inicializar la solución inicial con un algoritmo greedy rápido que nos permita obtener una solución de partida medianamente buena, y en consecuencia, podar desde el inicio el mayor número de ramas posible.

Veamos un pseudocódigo para el algoritmo Branch and Bound:

Algoritmo Branch and Bound. Empleo de la técnica Branch and Bound en el algoritmo.

```
def TSPBranchAndBround
  #Inicialización de datos:
  #Cola con prioridad
  queue = priority_queue.new

  #Añadimos nodo inicial:
  queue.push(Nodo.new(ciudades.primeraCiudad))

  # Inicializamos mejor solución con greedy.
  # En este caso, vecino más cercano.
  mejor_solucion = TSPVecinoMasCercano()

  #Mientras haya nodos en la cola:
  while !queue.empty() and queue.top().estimacion < mejor_solucion
    n = queue.pop #Extraemos primer nodo
    hijos = n.getHijos() #Obtenemos vector con sus hijos

    # Recorremos los hijos y los añadimos a la cola si su estimación
```

```
    # es buena.
    for hijo in hijos
      hijo.estimar()
      if hijo.estimacion < mejor_solucion
        queue.push(hijo)
      end
    end

    # Si el nodo es una hoja actualizamos la solución
    # si se da el caso
    if n.ciudadesVisitadas == totalCiudades
      and n.solucion < mejor_solucion
        mejor_solucion = n.solucion
    end
  end
end
```

2.2.2 Algoritmo Backtracking

Lo primero que debemos tener claro es en qué consiste la técnica Backtracking. En ella, a partir de un nodo (el comienzo inicial se da en el nodo raíz) se va recorriendo en profundidad el espacio de soluciones, siguiendo un criterio de poda de manera que si en un determinado momento las soluciones parciales son peores que la mejor solución actual en un momento determinado, podemos podar esa rama y avanzar al siguiente nodo sin podar.

El empleo de esta técnica a nuestras ciudades se basa en partir del nodo raíz, que ya nos da una solución parcial con una ciudad. Como mejor solución inicial podemos usar la obtenida al emplear un algoritmo greedy, en nuestro caso, el algoritmo del vecino más cercano (que va a ser bastante rápido pues el número de nodos no puede ser muy elevado). Al igual que se explicaba antes, en el momento que una solución parcial nos dé un resultado mayor que la mejor solución hasta ese momento, se desestima el resultado, mientras que si finalmente obtenemos una solución en esa rama mejor que la que se tenía hasta entonces, actualizamos esta variable.

Más allá del criterio de poda, el algoritmo de Backtracking es muy sencillo. Se trata de avanzar por una solución parcial mientras podamos, llegando a las distintas hojas que serán las soluciones buscadas. Cuando no podemos avanzar más, bien porque hemos llegado a una hoja o porque hemos podado la rama que seguimos, volvemos al nivel anterior y evaluamos la siguiente ciudad. Es por esto por lo que el backtracking se lleva muy bien con la recursividad y las estructuras LIFO (Último que entra primero que sale) con sus correspondientes métodos *push* y *pop*. De esta forma ya podemos escribir el pseudocódigo de nuestro algoritmo:

Algoritmo Backtracking. Empleo de la técnica Backtracking.

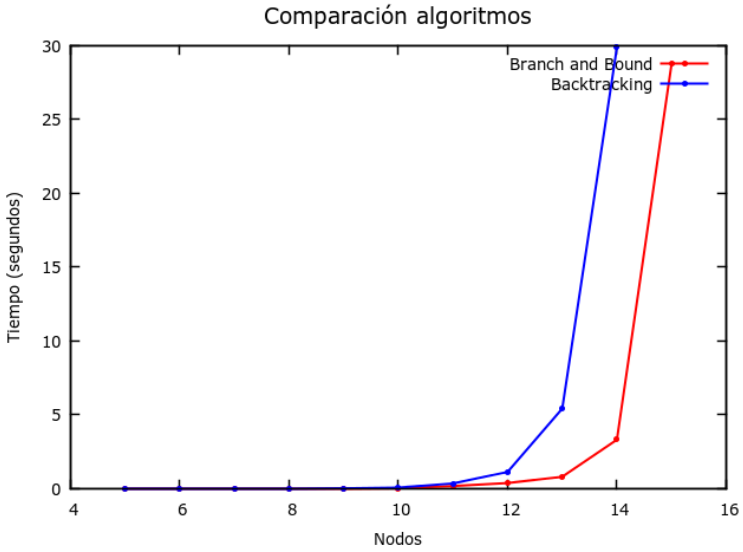
```
#Llamada inicial: TSPBacktracking(conjunto_ciudades,solucion_raiz,TSPVecinoMasCercano)
#candidatos: ciudades que quedan por visitar [set]
#sol_actual: recorrido parcial en ese momento
#mejor_solucion: mejor solución obtenida
def TSPBacktracking(candidatos, sol_actual, mejor_solucion)
  # Si la solución actual es completa, comprobamos
  # y actualizamos si se da el caso.
  if candidatos.empty and sol_actual < mejor_solucion
    mejor_solucion = sol_actual
  end

  for i in 0..candidatos.size
```

```
    #Avanzamos en la solución parcial:
    sol_actual.pushCiudad(candidatos[i])
    candidatos.eraseAt(i)

    #Si no se cumple el criterio de poda, llamamos de nuevo a la función
    if sol_actual < mejor_solucion
      TSPBacktracking(candidatos,sol_actual,mejor_solucion)
    end

    # Vuelta atrás: devolvemos los candidatos y la solución
    # a su estado en el momento de la entrada a la función
    i = candidatos.insert(sol_actual.popCiudad())
  end
```

2.3 Resultados empíricos

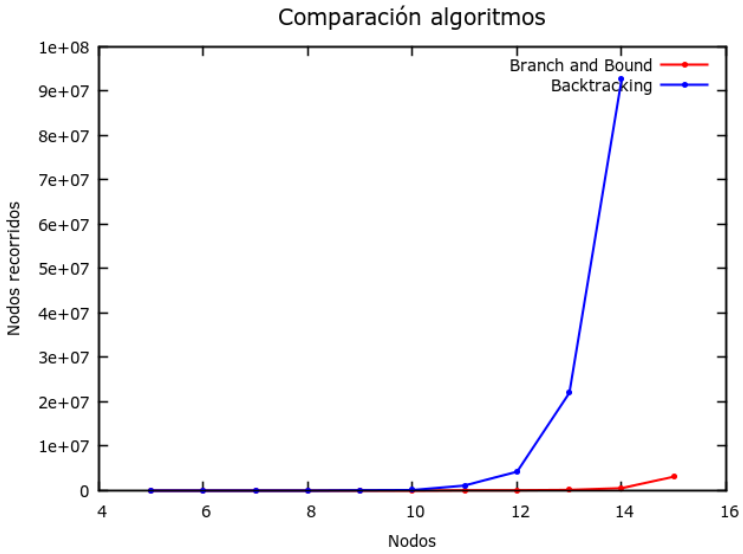
A continuación vamos a poner a prueba de forma empírica ambos algoritmos. Para ello haremos un doble estudio comparativo. En primer lugar compararemos los algoritmos según el tiempo que emplean, y en segundo lugar, según los nodos que visitan en el espacio de soluciones.

2.3.1 Tiempos



Lo primero que debemos comentar es que con Branch and Bound podemos llegar a resolver el problema para 15 ciudades sin llegar a tiempos ya inabordables. Con Bracktacking, en cambio, solo podemos llegar a 14, momento en el que tiempo empieza a crecer de forma excesiva. Como podíamos esperar, el tiempo empleado por el algoritmo backtracking es mayor que el que emplea el algoritmo Branch and Bound. También debemos destacar que, a pesar de la mejora del algoritmo de ramificación y poda respecto al de vuelta atrás, apenas nos sirve para poder resolver el problema con una ciudad más. Debemos recordar que no se conoce de momento una forma de alcanzar la solución óptima a este problema en un tiempo polinomial, y que por eso, por mucho que aumentemos las podas no vamos a poder evitar que se disparen las gráficas de tiempos tarde o temprano. Para problemas de muchas ciudades, tendremos que seguir conformándonos con las soluciones que nos proporciona un buen algoritmo greedy.

2.3.2 Nodos visitados



Podemos ver cómo el número de nodos visitados por el algoritmo Backtracking es mucho mayor que el número visitado por el Branch and Bound, lo que recalca que el algoritmo Branch and Bound es, en ese aspecto, mucho mejor que el backtracking. El criterio de poda de Branch and Bound nos permite eliminar muchísimos nodos que nos vemos obligados a visitar con Backtracking. Sin embargo, debemos tener en cuenta que la ganancia de la poda de Branch and Bound implica un mayor coste computacional, a la hora de manejar las funciones de estimación y por la necesidad de almacenar una solución por cada nodo en la cola (Backtracking puede implementarse manejando una única solución que se va modificando fácilmente en el recorrido en profundidad). A pesar de eso, como podemos apreciar en los tiempos, el aumento del coste computacional no llega a eclipsar la ganancia con las podas.