## horizontal line



Práctica 4 : Problema del Viajante de Comercio

2019

**──**

Realizado por:

Miguel Ángel Campos Cubillas

Alejandro Pinel Martínez

Guillermo Palomino Sánchez

Pablo Lombardero Ros

Nikita Stetskiy

# Problema : Viajante de Comercio

El problema del viajante de comercio ha sido tratado con anterioridad en prácticas de nuestra asignatura como objeto de estudio para algoritmos voraces. En aquella práctica se estudiaron métodos de este tipo para encontrar soluciones razonables (no necesariamente óptimas) a este problema.

Si se desea obtener una solución óptima para el problema se deben utilizar métodos más potentes y costosos, como la vuelta atrás (backtracking) y la ramificación y poda, que exploren el espacio de posibles soluciones de manera más exhaustiva.

Así, un algoritmo backtracking comenzaría en la ciudad 1 (suponemos que la ciudad inicial es la ciudad final, a modo de tour) e intentaría incluir como parte del tour la siguiente ciudad aún no visitada, continuando de este modo hasta completar el 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 (ciudades aún no 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 con 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 mínimo coste posible de entrar y salir a estos nodos. Para calcular este coste de entrar y salir, se guarda el valor de las dos aristas de menor coste de cada nodo y se estima el valor de una arista con la media entre el valor entrante y saliente de cada nodo. 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 el momento ( 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. 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 el árbol de búsqueda (la solución parcial que tratamos de expandir), se empleará el criterio LC o “más prometedor”. En este caso consideramos como nodo más prometedor aquel que presente el menor valor de cota inferior. Para ello se debe 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 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.

# Objetivo

El objetivo de esta práctica se basa en conseguir diseñar un programa que utilice la técnica de ramificación y acotación para resolver el problema del viajante de comercio en las condiciones descritas en la descripción del problema, empleando la función de acotación comentada.

Las restricciones del problema que representan el algoritmo son:

-Restricciones Explícitas: La solución es una permutación de las etiquetas de las ciudades.

-Restricciones Implícitas: La suma del coste de las aristas de la solución debe ser mínimo con respecto al resto de soluciones.

# Algoritmo

En esta sección vamos a hablar del algoritmo utilizado para resolver el problema propuesto. En este caso, cada ciudad está representada por un nodo, el cual está compuesto por su posición en el mapa y su costo, cuyo valor usaremos para calcular el camino más óptimo a seguir, además, se usará un vector de enteros para llevar un seguimiento de los nodos que forman parte de la solución.

Debido a los objetivos a alcanzar y a los requisitos necesarios para resolver el problema, se ha optado por utilizar el algoritmo Branch&Bound aplicado a dicho problema. Para comenzar, el algoritmo usa una cola con prioridad, en la que se ordena los nodos en base a su estimación de coste y un vector que almacenará las distancias mínimas para las cotas, o lo que es lo mismo, las dos aristas más pequeñas. Tras esto, se coge el primer nodo de la lista y se calcula una cota inicial utilizando un algoritmo greedy, una vez obtenido, se realiza el algoritmo en sí.

Con la cota global obtenida, se procede a coger el primer nodo de la lista con prioridad y se comprueba si es factible, de serlo, se pasa al siguiente nodo de la lista y se comprueba si forma parte de la solución, si no forma parte de ella se añade su costo al total del camino, siendo su costo, el costo de entrar al nodo y el del camino que lleva a él[[1]](#footnote-0) y se comprueba si es factible, en caso de serlo se comprueba si es el último nodo por recorrer. Si es el último nodo, se añade el costo total a la cota global y se devuelve la solución, en caso contrario se sigue con el proceso hasta que la lista con prioridad esté vacía.

|  |
| --- |
| void Mapa::BranchBround() {  const int N = GetNCiudades();  priority\_queue<Nodo, vector<Nodo>, ComparaNodos > apo;  bool fin=false;  //Inicializamos el vector de distancias mínimas para las cotas  vector<vector<double> > distanciasminimas;  DistanciasMinimas(distanciasminimas);  //Inicializamos la solución con sólo el nodo 0  solucion = vector<int>(N, -1);  solucion[0] = 0;  Nodo nodo(solucion, 0, 0);  apo.push(nodo);    //Inicializamos la cota global lanzando un algoritmo greedy  GreedyCiudadProxima();  int cota\_global = CostoSolucion(solucion);  do {  nodo = apo.top();  apo.pop();  if (EsFactible(nodo, cota\_global, distanciasminimas)) {  NNodosExp++;  nodo.pos++;  int costoinicial = nodo.costo;  for (int i = 0; i < N; i++) {  if (!presenteEnSolucion(nodo.sol, i)) {  nodo.sol[nodo.pos] = i;  nodo.costo = costoinicial + calcularDistancia(nodo.sol[nodo.pos - 1], nodo.sol[nodo.pos]);  if (nodo.pos == N - 1)  nodo.costo += calcularDistancia(nodo.sol[N - 1], nodo.sol[0]);  if (EsFactible(nodo, cota\_global, distanciasminimas))  if (nodo.pos == N - 1) {  cota\_global = nodo.costo;  solucion = nodo.sol;  }  else{  apo.push(nodo);  NPoda++;  }    }  }  }  else{  fin=true;  NPoda += apo.size();  }    if(NMaxNodos < apo.size()){  NMaxNodos = apo.size();  }    } while ( (!apo.empty()) && (!fin));  } |

En cuanto a la factibilidad, lo que se tiene en cuenta es el coste de entrada y de salida del último nodo visitado, es decir, la mitad del valor de sus aristas más pequeñas y lo añade a la cota local, para luego compararla con la cota global. Si la cota local es mayor que la global, no merece la pena seguir calculando el recorrido y por tanto no es factible.

|  |
| --- |
| bool Mapa::EsFactible(Nodo & n, int cota\_global, vector<vector<double> > & distanciasminimas) const{  double cota\_local = n.costo;  if (n.pos < GetNCiudades() - 1) {  if (n.pos >= 1 && calcularDistancia(n.sol[0], n.sol[1]) == distanciasminimas[0][n.sol[0]])  cota\_local += distanciasminimas[1][n.sol[0]]/2;  else  cota\_local += distanciasminimas[0][n.sol[0]]/2;  if (n.pos >= 1 && calcularDistancia(n.sol[n.pos], n.sol[n.pos - 1]) == distanciasminimas[0][n.sol[n.pos]])  cota\_local += distanciasminimas[1][n.sol[n.pos]]/2;  else  cota\_local += distanciasminimas[0][n.sol[n.pos]]/2;  for(int i = 0; i<GetNCiudades(); i++){  if (!presenteEnSolucion(n.sol, i)){  cota\_local += distanciasminimas[0][i]/2;  cota\_local += distanciasminimas[1][i]/2;  }  }  }  n.estimacion = cota\_local;  return cota\_local < cota\_global;  } |

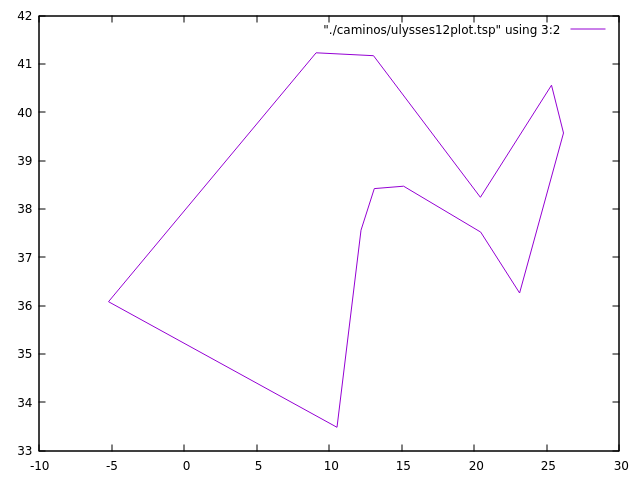
# 

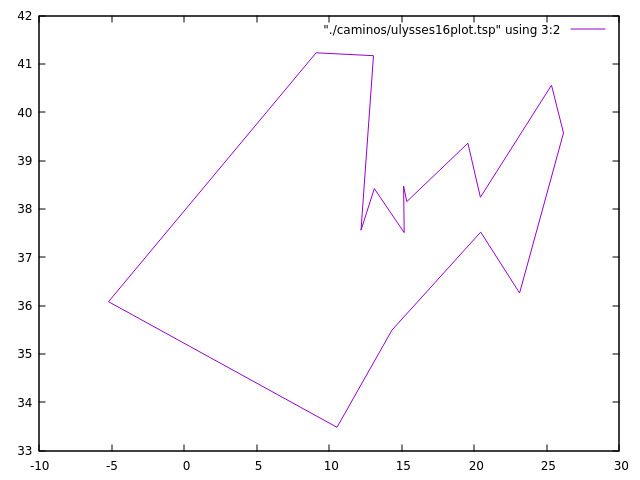
# Datos obtenidos para distinto conjunto de ciudades

|  |  |  |
| --- | --- | --- |
|  | ulysses12 | ulysses16 |
| Coste | 68 | 71 |
| Tiempo utilizado en calcular la ruta | 0.178478 seg | 123.63 seg |
| Número máximo de nodos almacenados | 61432 | 46806712 |
| Número de nodos explorados | 165750 | 78062433 |
| Número de podas realizadas | 195440 | 128276185 |

# Gráficas de la rutas obtenidas

# 





# Aclaración sobre ulysses16

Como se puede ver, existen ligeras variaciones entre el camino óptimo proporcionado de ulysses16 y el generado. Esto es debido a las aproximaciones realizadas al calcular las distancias entre ciudades, debido a las propias del problema de utilizar números enteros.

Los resultados mostrados han sido generados con las distancias aproximadas al entero más cercano (al calcular el coste de ulysses16 según este método, mejoraba el coste del camino óptimo proporcionado). Otras aproximaciones, como truncar las distancias o utilizar valores en coma flotante dan resultados ligeramente diferentes

# Conclusión

Se puede observar una clara mejoría con respecto a los resultados obtenidos con los algoritmos greedy, a costa del rendimiento del programa y de un tiempo de ejecución mucho más elevado. Esto nos plantea un dilema, ¿utilizar un algoritmo más rápido pero que no garantice una solución óptima o utilizar un algoritmo pesado que encuentre las mejores soluciones posibles?

Como suele suceder en la mayoría de situaciones, la respuesta a dicho dilema depende de las características del problema y de los recursos a disposición, ya que por ejemplo para este caso, pese a que creemos que dado el tiempo necesario, el algoritmo habría sido capaz de sacar el resultado para ulysses18, a mitad de la ejecución se producía un error al intentar reservar espacio para la cola, por requerir esta demasiado espacio.

1. El valor del camino que conecta dos nodos a y b se calcula como sqrt[ (b.x - a.x)²+(b.y - a.y)²], redondeando el resultado al entero más cercano [↑](#footnote-ref-0)