Problema del viajante de comercio
Travelling salesman problem

José Antonio Álvarez Ocete - Norberto Fernández de la Higuera
Javier Gálvez Obispo - Yábir García Benchakhtir

23 de mayo de 2018

Índice

1. Descripción del problema

En el problema del viajante de comercio (*Travelling Salesman Problem*) se nos dan las coordenadas de un conjunto finito de ciudades y se nos pide encontrar un recorrido de las misma de forma que la distancia que recorramos sea la menor posible.

2. Solucion ofrecidas

En todas las soluciones descritas hemos usado una matriz de adyaciencia, esto es, una matriz M donde m_{ij} nos indica la distancia de la ciudad i a la ciudad j. En nuestro caso se cumple además que $m_{ji} = m_{ij}$ ya que consideramos que el camino que une dos ciudades se puede recorrer en ambos sentidos.

2.1. Solución branch and bound

Branch and bound es una técnica similar a la de backtracking pero en este caso la técnica de ramificación es distinta.

En lugar de desarrollar una rama del arbol hasta llegar a una hoja desarrollando por niveles. Por tanto al mismo tiempo tenemos varios nodos vivos. No los desarrollamos todos simultaneamente sino que los desarrollamos en función de las expectativas que tenemos de cada nodo.

Cada nodo de nuestro grafo representa el estado parcial de una solución y una cota que nos indica *como de buena podría ser* dicha solución. En cada iteración desarrollamos el nodo que siga *vivo* y que mejor cota tenga.

Algorithm 1 Algoritmo Branch and Bound

```
solution = next PriorityQueue(<length, path>)
if path is completed then
    localLength = path.length
else
    minPossible = estimate path
    localLength = solution.length
end if
if localLength < best Solution then
    for city not in path do
        add city to the path
        compute new min Length
        Explore new solution if it's worth
    end for
end if
```

Como función que nos permite estimar lo buena que puede llegar a ser una solución calculamos la suma de los mínimos de cada fila en la matriz de adyaciencia que no haya sido ya visitada (no se encuentra en el camino actual).

```
1  double estimacion(vector<int> &candidates, vector<vector<double> > &cities){
2
3    double result = 0;
4    double row_min;
5
6    for(int i = 0; i < cities.size()-1; i++){
7      if(!visitado(i,candidates)){
8        result += rowMin(i, cities[i]);
9      }
```

```
    }

    return result;

  }
```

Además para mejorar la eficiencia de nuestro algoritmo hemos partido de una solución inicial que podemos considerar *buena*, es decir, próxima a la solución óptima en cuanto a distancia del recorrido se refiere. De esta manera conseguimos que nuestro algoritmo explore menos nodos (realice más podas) y por lo tanto estamos mejorando su eficiencia empírica.

La solución inicial que nos ha permitidos podar la hemos basado en el algoritmo genético que desarrollamos en la práctica anterior y que de manera rápida nos ofrecía una solución bastante buena. De esta manera a la hora de realizar poda podíamos descartar más resultados de manera eficiente.

Para decidir la ciudad de inicio para el recorrido empleamos la siguiente función:

```
int farthest(vector<pair<double, double> > &coordinates){
  pair<double,double> centro(0,0);
  double dist, max = 0;
  int mejor = 0;

  for(int i = 0; i < coordinates.size(); i++){
    centro.first += coordinates[i].first;
    centro.second += coordinates[i].second;
  }

  centro.first /= coordinates.size();
  centro.second /= coordinates.size();

  for(int i = 0; i < coordinates.size(); i++){
    dist = distance(coordinates[i], centro);
    if(dist > max){
      max = dist;
      mejor = i;
    }
  }

  return mejor;
}
```

Con esto conseguimos que nuestra primera ciudad sea la que está mas lejos del centro y en nuestro recorrido nos vamos acercando al centro. Esta técnica es una de las que se propuso en ocasiones anteriores y que hemos considerado buena idea aplicar.

2.2. Solución usando backtracking

En la solución propuesta de backtracking realizamos una exploración vertical del arbol de soluciones. Nuestro nodo en desarrollo sera el primer hijo que podamos explorar hasta llegar a un nodo terminal y una vez ahí pasará a ser nodo muerto.

Una posible implementación de dicho algoritmo sería la siguiente:

```

void backtracking(int pos, pair<double, vector<int> > sol, vector<vector<double> > &
    cities, pair<double, vector<int> > &best){
  if(pos < cities.size()){
    for(int i = 1; i < cities.size(); i++){
                if(!visitado(i, sol.second)){
                        sol.second[pos] = i;
        sol.first = compute_length(sol.second, cities);
                        if(sol.first < best.first)
          backtracking(pos + 1, sol, cities, best);
        }
    }
```

```
12    } else if(best.first < best.first)
13      best = sol;
14  }
```

Inicialmente y sin estudiar empíricamente los algoritmos vemos que ambos tienen una eficiencia teórica de $O(n!)$ ya que estamos estudiando todas las combinaciones posibles de ordenar un conjunto de n candidatos. Sin embargo cabría esperar que no se comportasen de esta manera ya que por ejemplo en el algoritmo de branch and bound no estamos desarrollando todos los nodos por lo que el cardinal de nodos que tenemos que trazar es mucho menos del total que podríamos recorrer. En el algoritmo de backtracking usamos una técnica similar que nos permite eliminar soluciones sin tener que explorar la rama al completo.

3. Análisis empírico de los algoritmos

Para estudiar la eficiencia empírica de los resultados se ha utilizado un ordenador con las siguiente caracterísiticas:

- CPU: Intel Pentium G3258 (2) @ 3.200GHz
- Memoria RAM: 7876MiB
- Kernel: 4.13.0-36-generic
- OS: Linux Mint 18.3 Sylvia x86_64

Debido a la complejidad del algoritmo no fuimos capaces de obtener resultados para mapas muy grandes y en particular para los mapas ofrecidos salvo para el caso de *ulysses16*. Es por esto que se ha procedido a la creación de mapas propios, los cuales se acompañan a este documento y en los que sí hemos podido probar nuestro algoritmo.

Cuadro 1: Comparativa de longitud y tiempo (seg) para cada algoritmo

Longitud	Branch and bound	Backtracking
19.9247	3E-06	2E-06
51.6735	6E-06	3E-06
59.4593	1.7E-05	5E-06
56.4986	5E-05	1.6E-05
119.193	0.000148	6.2E-05
150.371	0.000919	0.000281
255.254	0.001052	0.000763
361.223	0.003113	0.003867
464.618	0.015933	0.021076
476.531	0.014079	0.049719
586.246	0.193942	3.49537
755.058	0.046548	0.676337
1040.99	0.234971	9.97742
1146.6	0.311181	29.1998

Para visualizar los datos los hemos representado en una gráfica de línieas donde comparamos ambos algoritmos:

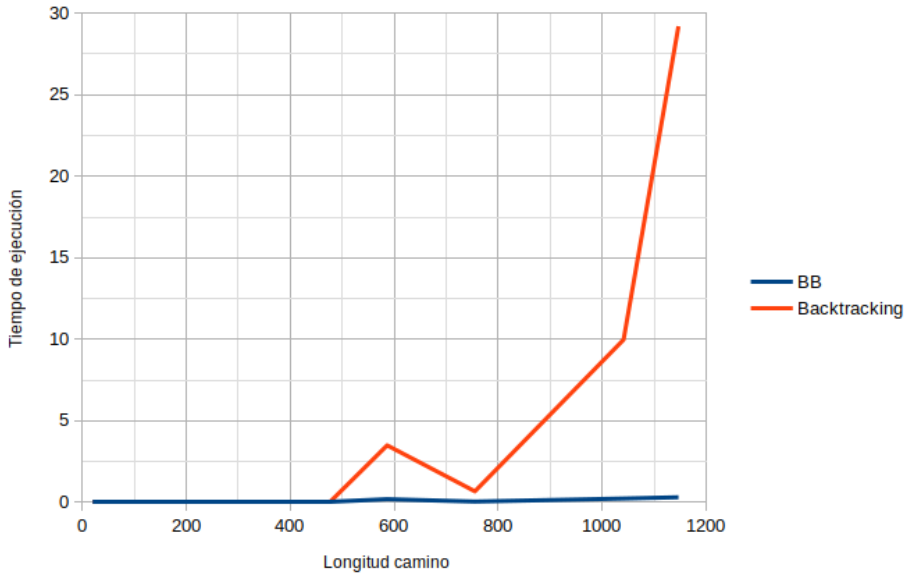

Figura 1: Comparativa de los algoritmos de backtracking y branch and bound

4. Comparativa con los algoritmos greedy

Si comparamos por ejemplo con el algoritmo greedy *cheapInserction* obtenemos los resultados siguientes:

Cuadro 2: Resultados para un algoritmo greedy

19.9247	4E-06
51.6735	2E-06
59.4593	1E-06
62.8659	2E-06
123.26	2E-06
150.371	3E-06
300.216	3E-06
361.223	3E-06
494.908	2E-06
476.531	4E-06
794.865	4E-06
581.918	4E-06
1005.65	5E-06
1070.15	5E-06
1272.67	6E-06

En esta tabla podemos contrastar de manera clara como en tiempo los algoritmos greedy ganan a los de branch and bound pero los recorridos que obtienen no son mejores. A continuación se han representado uno de los resultados que se obtienen con branch and bound y usando algoritmos greedy.

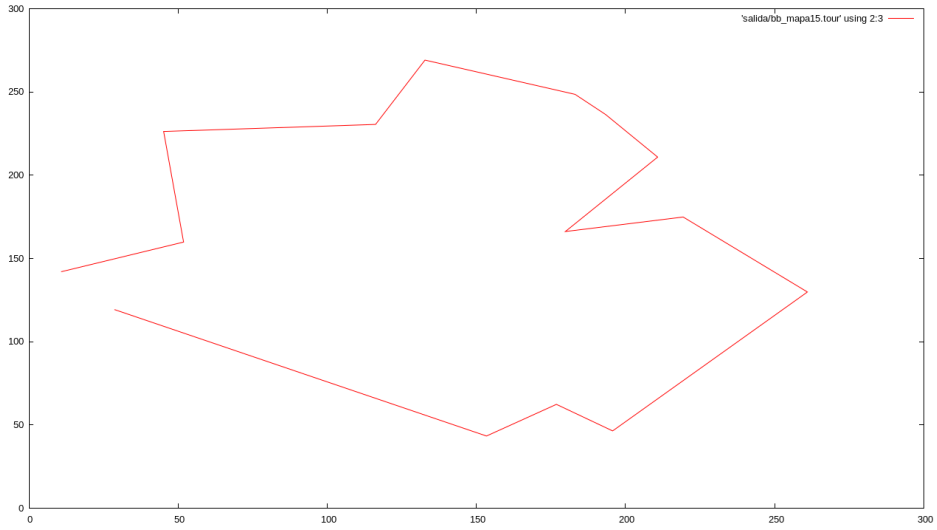

Figura 2: Algoritmo de branch and bound

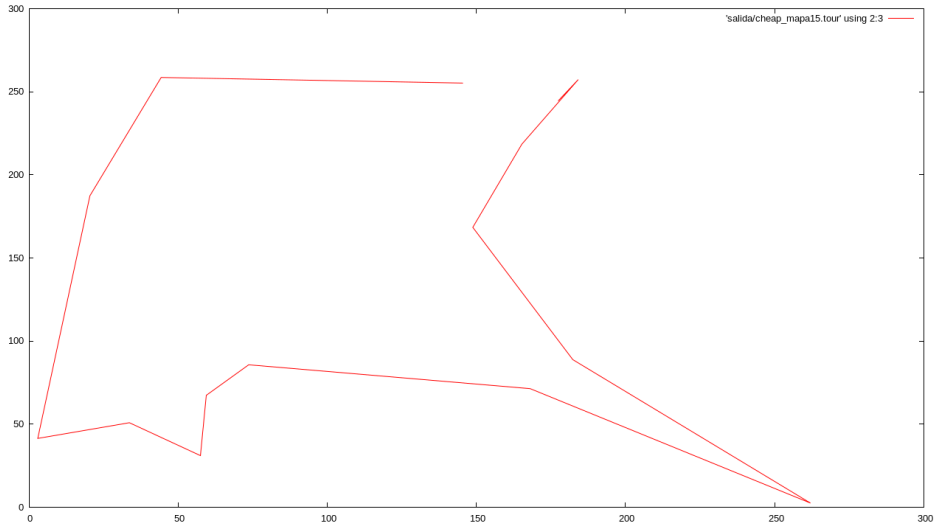

Figura 3: Algoritmo greedy

5. Conclusiones

En este estudio hemos podido comprobar como la técnica de branch and bound ofrece resultados en tiempo mejores que la de backtracking. Pese a que ambos devuelven la mejor solución siempre debido a que en ambos algoritmos estamos explorarndo todo el grafo de soluciones posibles.

En segundo lugar los resultados demuestran que la técnica de branch and bound obtiene mejores tiempos de ejecución para encontrar la misma solución que empleando la técnica de backtracking.

Estos resultados son los que cabría esperar de este tipo de algoritmos donde se prima lo buena que sea la solución más que el tiempo que tarde en obtenerse.