ugr Universidad de Granada

E.T.S. DE INGENIERÍA INFORMÁTICA Y TELECOMUNICACIÓN

Departamento de Ciencias de la Computación e Inteligencia Artificial

Algorítmica

Práctica 4 – Parte 2: Branch & Bound

Curso 2017-2018
Grado en Ingeniería Informática

Álvaro García Jaén
Francisco José González García
Práxedes Martínez Moreno
Ignacio Martínez Rodríguez
Pablo Robles Molina

INDICE

1. Introducción

Para resolver el problema del viajante de comercio se proponen dos soluciones: Una basada en el algoritmo Branch and Bound y otra basada en Backtracking. Estos dos algoritmos además hacen uso de dos cotas diferentes (superior e inferior) siendo la primera global y la segunda local.
En el momento en el que la cota local de un recorrido supera la cota global se deja de explorar la rama que cuelga de ese estado, siendo la cota global la misma para todos nuestros ejemplos (la mejor solución hasta el momento) y para la cota local se proponen distintas soluciones más adelante.

2. Explicación del algoritmo Branch & Bound

```
branch&Bound(solucion[], matriz_distancias[][], ciudades[][]){
    priority_queue cola //La prioridad es la mejor cota local
    solucion <- primera_ciudad
    cota_global <- greedy(ciudades)

    do{
        sol_tmp <- cola.top()
        cola.pop()

        hijos <- generarHijos(sol_tmp, ciudades)

        for(hijo in hijos){
            if(cota_local(hijo) <= cota_global)
                if(esSolucion(hijo)){
                    solucion <- hijo
                    cota_global <- distanciaCompleta(solucion)
                }else{
                    cola.push(hijo)
                }
        }
    }while(!cola.empty())

    solucion <- primera_ciudad
}
```

El funcionamiento del algoritmo Branch and Bound es el siguiente:

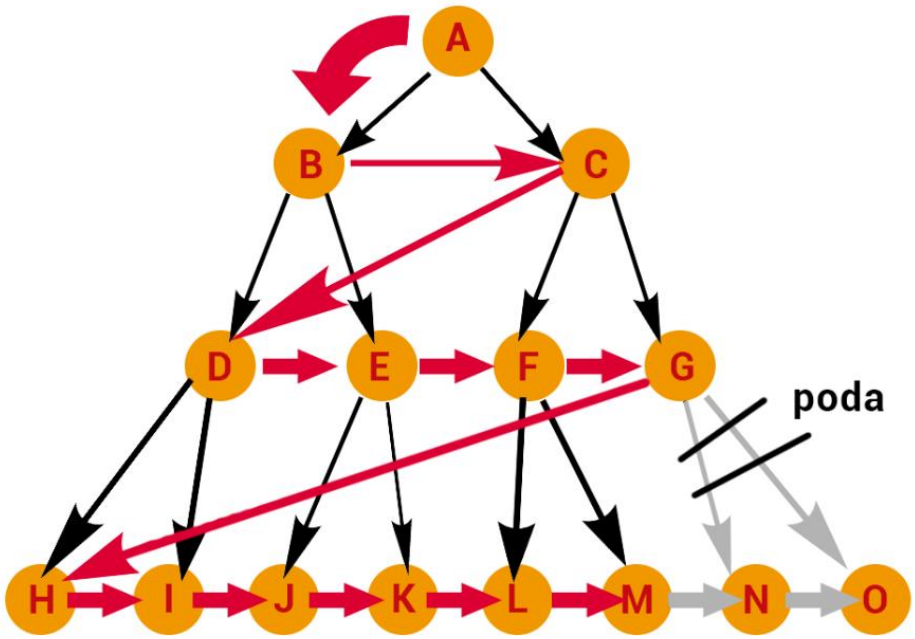
Partiendo de una cola con prioridad que contiene la primera de las ciudades se calcula la cota global haciendo uso del algoritmo Greedy, tras esto y de manera iterativa, se realizaran las siguientes acciones mientras queden caminos por explorar en la cola:

Se extrae y se elimina el primer elemento de la cola. De este calculamos sus estados hijos y para cada uno de ellos, si la cota local de ese estado satisface la cota global (se queda por debajo) se pueden dar dos casos:

- Que sea una solución al problema en cuyo caso se guarda como solución y se actualiza la cota global con la distancia de este camino.
- Que no sea solución y por tanto se mete en la cola.

Si no satisfaciese la cota global se poda y no se exploran sus estados hijos.

Finalmente tendremos que añadir a nuestro camino la primera ciudad de la que partimos.



3. Explicación del algoritmo Bakctracking

```
backTracking(estado[]){
    if(esSolucion(estado){
        if(distanciaCompleta(estado) < cota_global){
            cota_global <- distanciaCompleta(estado)
            solucion <- estado
        }
    }else{
        hijos <- generarHijos(estado)
        for(hijo in hijos)
            if(cota_local(hijo) <= cota_global)
                backTracking(hijo)
    }
}
```

Nuestro algoritmo backtracking funciona de la siguiente forma:
Empezamos comprobando que el estado que se le pasa es solución y que este sea mejor que la cota global, entonces esta pasa a ser la solución y la cota global se actualiza a su distancia.

Si no se cumple que sea solución se generan sus hijos y para cada uno de estos que satisfaga la cota global se vuelve a llamar de forma recursiva.

4. Cotas

Se han desarrollado dos tipos de cotas con el fin de encontrar la solución en el menor tiempo posible.

4.1. Cota optimista

```
cotaOptimista(estado[], ciudades[][]){
    sin_recorrer <- complementario(estado)
    min = 0
    for(ciudad in sin_recorrer)
        min += distanciaAMasCercano(ciudad)

    return min + distancia(estado)
}
```

Para calcular la cota optimista tenemos un vector de ciudades sin recorrer. Para cada una de estas buscaremos la más cercana del recorrido que tenemos hasta el momento sin que formen un ciclo, de modo que cualquier estado decente tendrá una cota menor que la cota global. Desechando así solo los estados pésimos.

```
cotaMenosOptimista(estado[], ciudades[][]){
    sin_recorrer <- complementario(estado)
    min = 0
    for(ciudad in sin_recorrer){
        //Con este metodo calculamos la suma de
        //las distancias a las dos ciudades mas cercanas
        min += distanciaADosMasCercanas(ciudad)
    }
    min = min/2
    return min + distancia(estado)
}
```

4.2. Cota menos optimista

Para calcular esta cota lo que hacemos es sumar las distancias de las dos mejores ciudades y hacemos la media de la suma total de estas distancias. De modo que tendríamos una cota más restrictiva que la anterior.
Para implementar los algoritmos Branch and Bound y BackTracking se han usado funciones auxiliares

5. Implementación

Además de implementar Branch & Bound y BackTracking se han usado funciones auxiliares que han ayudado al correcto funcionamiento de dichos algoritmos.

5.1. Funciones auxiliares

calculaMejoresCiudades

```
// Añade todas las ciudades al circuito insertando, en cada caso, la mejor
// opción
void calculaMejoresCiudades(vector<int> &cerrados,
                            const vector<vector<double>> &distancias,
                            map<int, pair<double, double>> &M) {
  vector<int> abiertos = complementario(cerrados, M);

  while (!abiertos.empty()) mejorCiudad(cerrados, abiertos, distancias, M);
}
```

distanciaCompleta

```
// Devuelve la distancia total de todas las ciudades recorridas en cierto orden
double distanciaCompleta(vector<int> ciudades,
                         const vector<vector<double>> &distancias) {
  double resultado = distancias[*(ciudades.begin())][*(ciudades.end() - 1)];

  for (auto it = ciudades.begin(); it != ciudades.end() - 1; ++it) {
    resultado += distancias[*it][*(it + 1)];
  }

  return resultado;
}
```

distanciaMinima

```
int distanciaMinima_V2(int ciudad, const vector<vector<double>> &matriz_dist) {
  int min1 = 100000, min2 = 10000000;
  int temp;
  int i_minima;
  for (int i = 1; i < matriz_dist.size(); ++i) {
    temp = matriz_dist[i][ciudad];
    if (temp < min1 && temp != 0) {
      min1 = temp;
      i_minima = i;
    }
  }
  for (int i = 1; i < matriz_dist.size(); ++i) {
    temp = matriz_dist[i][ciudad];
    if (temp < min2 && temp != 0 && i != i_minima) {
      min2 = temp;
    }
  }

  return min1 + min2;
}
```

generarHijos

```
vector<pair<int, vector<int>>> generarHijos(
  const vector<int> &sol, map<int, pair<double, double>> &M) {
      vector<int> comp = complementario(sol, M);
      vector<int> padre;
      vector<pair<int, vector<int>>> resultado;
      int cota;
      pair<int, vector<int>> par;
      for (auto ciudad : comp) {
        padre = sol;
        padre.push_back(ciudad);
        cota = optimista(padre, M);
        par.first = cota;
        par.second = padre;
        resultado.push_back(par);
      }

  return resultado;
}
```

Greddy

```
// Calcula un recorrido greedy
int greedy(map<int, pair<double, double>> &M) {
  vector<int> rec_inicial = recorridoInicial(M);
  vector<int> solucion = rec_inicial;
  vector<vector<double>> matriz_dist = matrizDistancias(M);
  calculaMejoresCiudades(solucion, matriz_dist, M);
  return distanciaCompleta(solucion, matriz_dist);
}
```

mejorCiudad

```
// Incluye la mejor ciudad posible en un circuito dado
void mejorCiudad(vector<int> &cerrados, vector<int> &abiertos,
                 const vector<vector<double>> &distancias,
                 map<int, pair<double, double>> &M) {
  vector<int> aux = cerrados;
  double distancia_aux = 0;
  aux.push_back(abiertos[0]);  // primer elemento
  vector<int> mejores_ciudades = aux;
  double dist_minima = distanciaCompleta(
      aux,
      distancias);  // Inicializamos la dist. minima a la del primer candidato

  aux = cerrados;
  for (auto it = abiertos.begin(); it != abiertos.end(); ++it) {
    for (auto it2 = aux.begin() + 1; it2 != aux.end(); ++it2) {
      aux.insert(it2, *it);
      distancia_aux = distanciaCompleta(aux, distancias);
      if (distancia_aux < dist_minima) {
        dist_minima = distancia_aux;
        mejores_ciudades = aux;
      }
      aux = cerrados;
    }
  }

  cerrados = mejores_ciudades;
  abiertos = complementario(cerrados, M);
}
```

Optimista

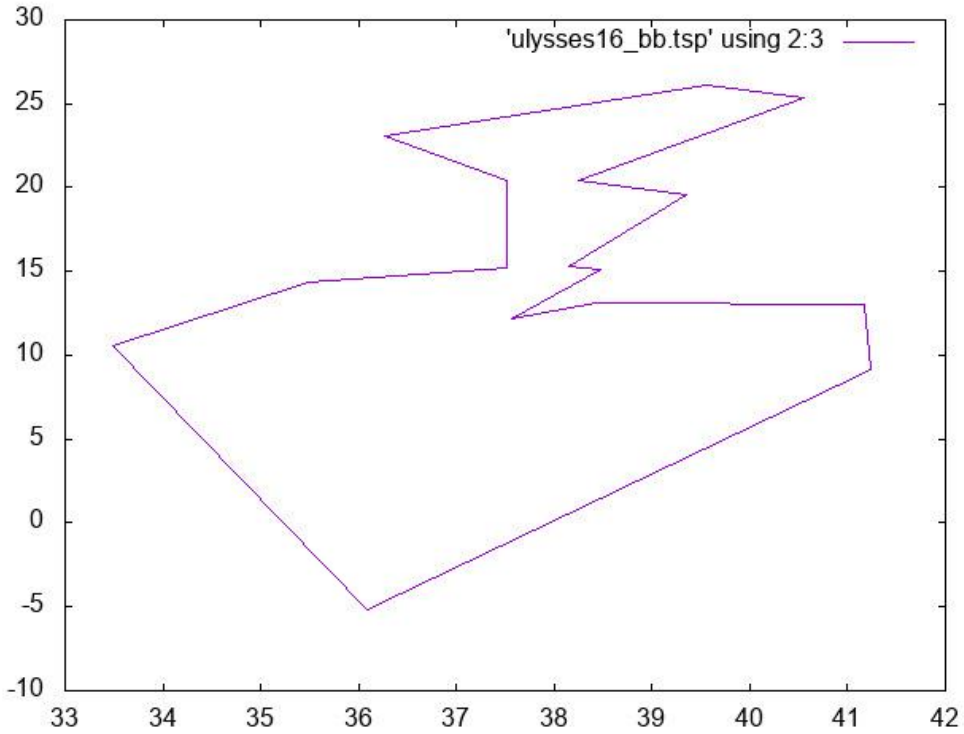
```
// Calcula el recorrido más optimista
int optimista(const vector<int> &sol, map<int, pair<double, double>> &M) {
  vector<int> sin_recorrer = complementario(sol, M);
  vector<vector<double>> matriz_dist = matrizDistancias(M);
  int min = 0;
  for (auto it = sin_recorrer.begin(); it != sin_recorrer.end(); ++it)
    min += distanciaMinima_V2(*it, matriz_dist);

  // Version 2
  min /= 2;
  return distanciaCompleta(sol, matriz_dist) + min;
}
```

El resto de código se incluye en los cpp adjuntos.

6. Resultados

Se han elegido varios archivos de datos para probar los algoritmos implementados.
A continuación se ilustran en varias graficas las soluciones obtenidas con cada algoritmo para varios conjuntos de datos.

Ulisses16 con Branch & Bound



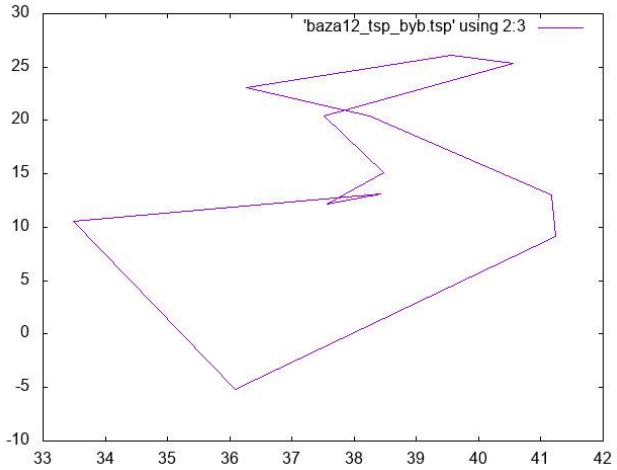
Tamaño de la cola: 17910423
Nodos expandidos: 375269353
Podas realizadas: 291138691

Camino solución

1-16-13-12-6-7-10-9-11-5-15-14-8-4-2-3-11

Ciudades: 16
Distancia(ByB) 71
Tiempo: 4.61174e+06 ms

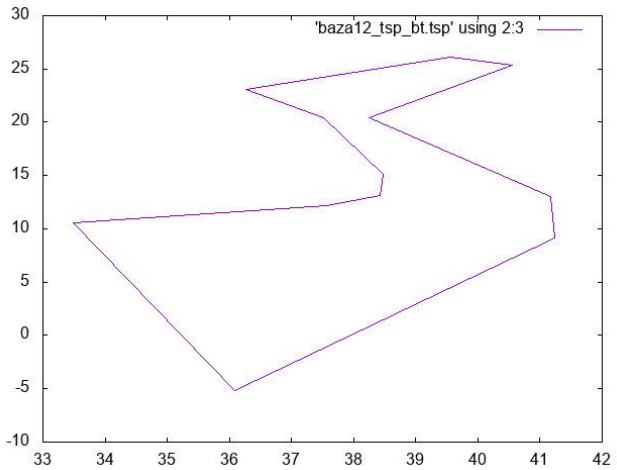
Baza12 con Branch & Bound



Size max cola: 71
Nodos expandidos: 117593
Podas realizadas: 84324

Camino solución
1-10-9-11-5-7-6-12-8-3-2-4-1
Ciudades: 12
Distancia(B&B) 72
Tiempo: 742.372

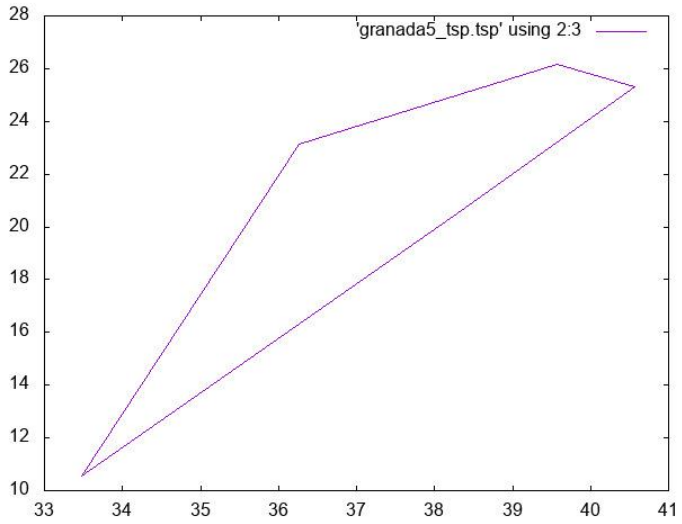
Baza12 con Backtracking



Camino solución
1-3-2-4-8-12-7-6-5-11-9-10-1

Ciudades: 12
Distancia(B&B) 68
Tiempo: 2030.85

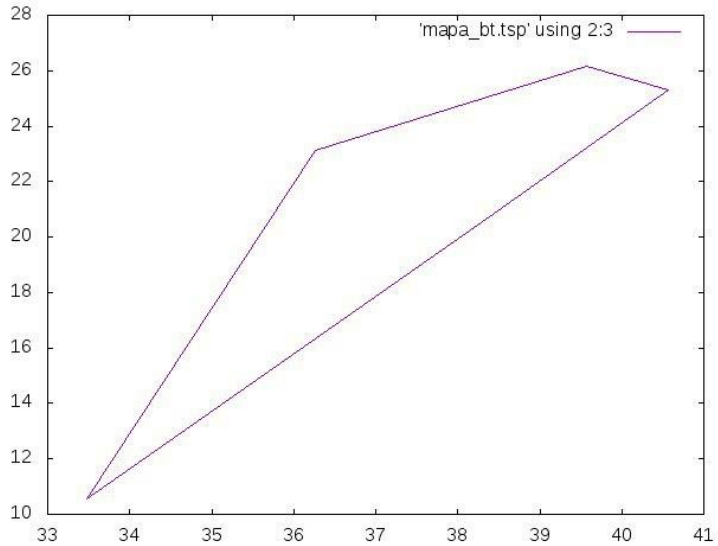
Granada5 con Branch & Bound



Tamaño de la cola: 5
Nodos expandidos: 30
Podas realizadas: 16

Camino solución
1-3-2-4-5-1
Ciudades: 5
Distancia (B&B) 34
Tiempo: 0.101375

Granada5 con Backtracking



1-3-2-4-5-1
Ciudades: 6
Distancia(Backtracking) 34
Tiempo: 1.9786