DECSAI
Universidad de Granada

Problema del Viajante del Comercio

Técnicas Backtracking y Branch and Bound

ugr Universidad de Granada

ÍNDICE

ENUNCIADO DEL PROBLEMA

Dado un conjunto de ciudades y una matriz con las distancias entre todas ellas, un viajante debe recorrer todas las ciudades exactamente una vez, regresando al punto de partida, de forma tal que la distancia recorrida sea mínima. Más formalmente, dado un grafo G, conexo y ponderado, se trata de hallar el ciclo hamiltoniano de mínimo peso de ese grafo.

SOLUCIÓN TEÓRICA. BACKTRACKING

ALGORITMO:

- Comienza por la ciudad 1 (cualquiera)
- Intenta meter en el tour la siguiente ciudad no visitada
- Si no quedan ciudades por visitar, el algoritmo hace backtracking proponiendo una ciudad válida para el nivel anterior

IMPLEMENTACIÓN BACKTRACKING

```
double suma(list<int> rama, double* afinidades,int n){
  double sum=0;
  list<int>::iterator it;
  list<int>::iterator next_it;
  for(it=rama.begin(); it!=prev(rama.end());++it){
    next_it=it;
    ++next_it;
    sum += afinidades[(*it)*n+(*(next_it))];
  }
  sum+= afinidades[(*(rama.begin()))*n+(*(prev(rama.end())))];
  return sum;
}

list<int> BacktrackingTSP(list<int> rama, double* afinidades, list<int> sinusar,double& minima_suma, int n, double& cont, double& total)
{
  int tam = sinusar.size();
  double sum=suma(rama,afinidades,n);
  if(tam <= 1 && minima_suma>sum && sum>0){
    rama.splice(rama.end(),sinusar);
    minima_suma=sum;
    return rama;
  }else{
    list<int>::iterator it;
    list<int> res;
    list<int> aux2;
    for(it = sinusar.begin(); it != sinusar.end(); ++it){
      int aux = *it;
      it = sinusar.erase(it);
      rama.push_back(aux);
      aux2 = BacktrackingTSP(rama, afinidades, sinusar,minima_suma, n, cont, total);
      if(aux2.size() != 0){
        res = aux2;
      }
      it = sinusar.emplace(it,aux);
      rama.remove(aux);
    }
    return res;
  }

}
```

RESULTADOS Y GRÁFICAS (dataset custom.tsp)

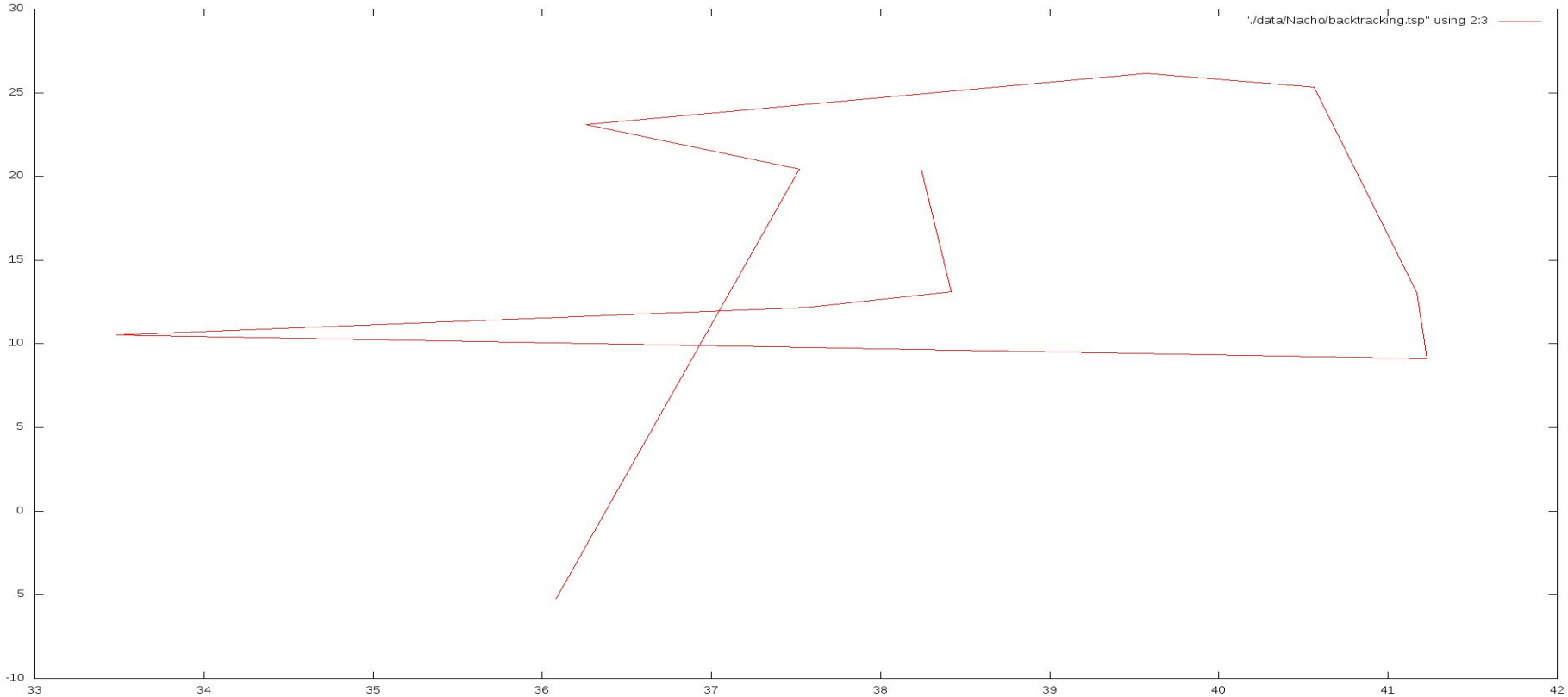
DIMENSION: 11

1	38.24	20.42
7	38.42	13.11
6	37.56	12.19
5	33.48	10.54
9	41.23	9.1
10	41.17	13.05
3	40.56	25.32
2	39.57	26.15
4	36.23	23.12
8	37.52	20.44
11	36.08	-5.21

El tamaño del recorrido es: 46.8093

TIEMPO: 11 nodos, 6.38558s

RESULTADOS Y GRÁFICAS (dataset custom.tsp)



SOLUCIÓN TEÓRICA. BRANCH AND BOUND

ALGORITMO:

- Cota inferior: Solución greedy Vecinos más cercanos (Práctica anterior)
- Solución base: recorrido greedy. Si encuentra un recorrido mejor mediante branch and bound, modifica ese recorrido.
- Como todo branch and bound, encuentra el óptimo

IMPLEMENTACIÓN BRANCH AND BOUND

```
double DistanciaRecorrido(list<Ciudad>& lista)
{
  double dist = 0;
  for(list<Ciudad>::iterator i = lista.begin(); i !=lista.end();++i)
  {
    list<Ciudad>::iterator sig=i;
    ++sig;
    dist+=Distancia(i->coord_x, sig->coord_x, i->coord_y, sig->coord_y);
  }
  return dist;
}

list<Ciudad> BranchGredy(list<Ciudad> rama, list<Ciudad> sinusar, list<Ciudad>& mejor_sol)
{
  if(DistanciaRecorrido(rama) >= DistanciaRecorrido(mejor_sol))
  {                          // Si la afinidad perdida es más grande que la afinidad
    list<Ciudad> res;         // perdida en la mejor distribucion hasta el momento, men,
    res.clear();           // desiste del tema.
    return res;
  }
  else
  {
    int tam = sinusar.size();
    if(tam <= 1){     // Si queda una persona por sentar, men, has llegado
      rama.splice(rama.end(),sinusar);
      mejor_sol=rama;
      return rama;
    }else{
      list<Ciudad>::iterator it;
      list<Ciudad> res;
      list<Ciudad> aux2;
      for(it = sinusar.begin(); it != sinusar.end(); it++){ // Crea una rama por cada persona que no este sentada
        Ciudad aux = *it;
        it = sinusar.erase(it);
        rama.push_back(aux);
        aux2 = BranchGredy(rama, sinusar, mejor_sol);
        if(aux2.size() > 0){
          res = aux2;
        }
        it = sinusar.emplace(it,aux);
        rama.remove(aux);
      }
      return res;
    }
  }
}
```

IMPLEMENTACIÓN BRANCH AND BOUND

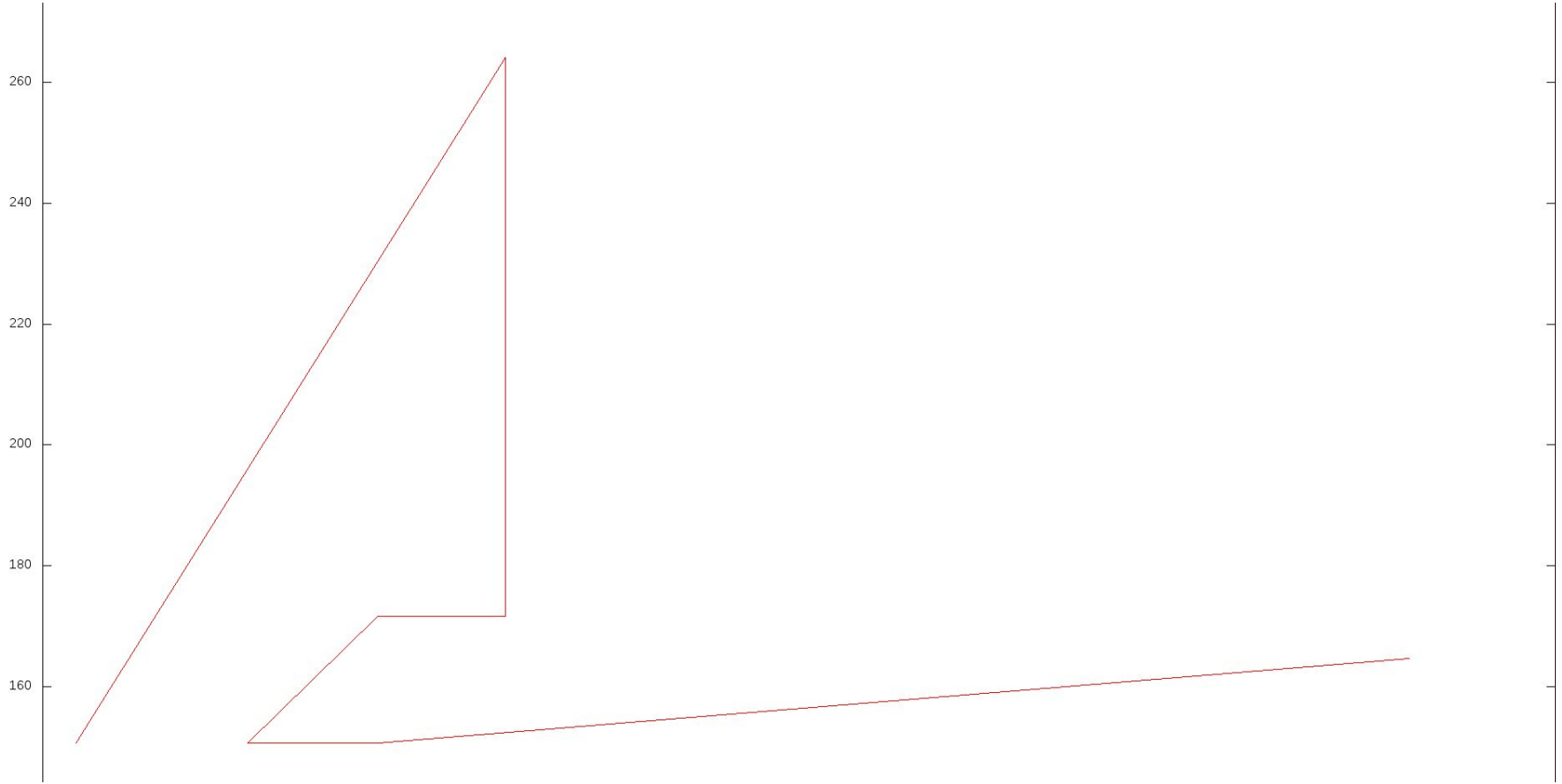
```
double BranchNBound(vector<Ciudad> mapa, vector<Ciudad>& solucion)
{
  vector<Ciudad> sol;
  TSP_vecino_mas_cercano(sol, mapa);
  double cota = DistanciaRecorrido(sol);
  list<Ciudad> rama_inicial;
  list<Ciudad> sinusar;
  rama_inicial.push_back(mapa[0]);
  for(int i = 1; i < (int)mapa.size();++i)
    sinusar.push_back(mapa[i]);
  cout << cota << endl;
  list<Ciudad> solucion_lista;
  for(int i = 1; i < (int)sol.size();++i)
    solucion_lista.push_back(sol[i]);
  BranchGredy(rama_inicial, sinusar, solucion_lista);
  for(list<Ciudad>::iterator it = solucion_lista.begin(); it!=solucion_lista.end();++it)
    solucion.push_back(*it);
  return DistanciaRecorrido(solucion);
}
```

RESULTADOS Y GRÁFICAS (custom.tsp tras greedy)

1	155.42	150.65
10	226.42	264.15
9	226.42	23.65
8	226.42	207.15
7	226.42	186.15
6	226.42	171.65
5	205.42	171.65
3	183.92	150.65
4	205.42	150.65
2	375.92	164.65

TIEMPO: 10 nodos, 4.271 segundos

RESULTADOS Y GRÁFICAS (custom.tsp tras greedy)



INTERPRETACIÓN

Como hemos podido observar, en todos estos problemas elegir backtracking como solución es un desacierto, dada su ineficiencia computacional y en memoria. Branch and bound es la alternativa, dado que la poda de las ramas que no son solución ahorran mucho tiempo de cómputo y espacio en memoria.