Algorítmica

Curso 2023-2024

Grupo Viterbi

PRÁCTICA 2-DIVIDE Y VENCERÁS

Integrantes:

Miguel Ángel De la Vega Rodríguez	miguevrod@correo.ugr.es
Alberto De la Vera Sánchez	joaquin724@correo.ugr.es
Joaquín Avilés De la Fuente	adelaveras01@correo.ugr.es
Manuel Gomez Rubio	e.manuelgmez@go.ugr.es
Pablo Linari Perez	e.pablolinari@go.ugr.es

Facultad de Ciencias UGR
Escuela Técnica Ingeniería Informática UGR
Granada
2023-2024

Índice general

Apartado 1

Autores

- **Miguel Ángel De la Vega Rodríguez:** 20%
  - Algoritmos del Viajante
  - Redacción memoria sección Viajante
- **Joaquín Avilés De la Fuente:** 20%
  - Programacion SumaMax (DyV)
  - Estudio eficiencia teórica algoritmos específicos y DyV
  - Tests de eficiencia
- **Alberto De la Vera Sánchez:** 20%
  - Redacción LaTeX
  - Estudio de umbrales teóricos y empíricos de DyV
  - Graficas y ajustes
- **Manuel Gomez Rubio** 20%
  - Programacion SumaMax (Kadane)
  - Programacion Losetas
- **Pablo Linari Pérez:** 20%
  - Programacion SumaMax (Kadane)
  - Programacion Losetas

Apartado 2

Equipo de trabajo

- **Miguel Ángel De la Vega Rodríguez:** (Ordenador donde se ha realizado el computo)
  - AMD Ryzen 7 2700X 8-Core
  - 16 GB RAM DDR4 3200 MHz
  - NVIDIA GeForce GTX 1660 Ti
  - 1 TB SSD NvMe
  - Debian 12 Bookworm
  - Compilador GCC 12.2.0

Apartado 3

Viajante

En esta sección se analiza todo aquello referente al cuarto problema propuesto, el problema del viajante. Siguiendo las directrices indicadas, se han estudiado y diseñado diferentes algoritmos "Greedy"que aproximan el problema, con el objetivo final de comparar cual nos proporciona mejores resultados en cuanto a términos de eficiencia y precisión. Para determinar esto último, también cabe destacar detalles de menor importancia como la complejidad de los algoritmos, la estabilidad frente a distintas entradas o la posibilidad de mejora de los mismos mediante la elección de distintos puntos de partida o con mejoras posteriores como pueden ser aquellas que proporcionan algoritmos como λ-opt o genéticos.

De aqui en adelante, describiremos por secciones los algoritmos implementados y al final proporcionaremos una sección comparativa sobre la cual nos basaremos para determinar conclusiones, en particular, elegiremos el algoritmo cuya solución consideremos más conveniente. En lo que sigue se muestra únicamente el resultado obtenido por el algoritmo sin mejoras posteriores, este tipo de mejoras suponen una pequeña desvirtuación del objetivo de encontrar el mejor algoritmo y es por ello, que su uso se limita a la conclusión final.

3.1 Algoritmo *Nearest Neighbour*

Tal y como el nombre indica, el primer algoritmo implementado no es nada más, ni nada menos que el primer algoritmo que probablemente se le puede ocurrir a cualquier persona que se enfrente a este problema. La idea es sencilla e intuitiva, nos dan un conjunto de puntos y queremos encontrar el camino más corto que los recorra, para ello, **elegimos** un punto inicial y a partir de ese punto, en un proceso iterativo, elegimos en cada paso el siguiente punto más cercano al actual, o lo que es lo mismo, el punto para el cual la distancia al punto actual es la menor (No confundir con el punto para el cual la distancia total es la menor, ya que, aunque parecido, no es lo mismo). Este proceso se repite hasta que todos los puntos han sido visitados.

A continuación se proporciona la implementación propuesta del algoritmo en cuestión:

```
1  vector<Point> nearestNeighborTSP(const vector<Point>& points) {
2    vector<Point> path;
3    vector<bool> visited(points.size(), false);
4    path.reserve(points.size());
5    path.emplace_back(points[0]);
6    visited[0] = true;
7
8    for (int i = 0; i < points.size() - 1; ++i) {
9      double minDistance = numeric_limits<double>::max();
10     int nearestNeighbor = -1;
```

```
    for (int j = 0; j < points.size(); ++j) {
      if (!visited[j]) {
        double distance = path[i].distanceTo(points[j]);
        if (distance < minDistance) {
          minDistance = distance;
          nearestNeighbor = j;
        }
      }
    }
    path.emplace_back(points[nearestNeighbor]);
    visited[nearestNeighbor] = true;
  }

  return path;
}
```

Como se puede apreciar, se ha hecho uso de un vector de puntos visitados (Programación dinámica) para evitar visitar un punto más de una vez, esto no supone una complejidad notable, sin embargo, si que supone una gran mejora en términos de eficiencia.

3.2 Algoritmo *Ordenación*

Como el nombre indica, este algoritmo consiste en ordenar los puntos de partida de acuerdo a un criterio especifico (en nuestro caso los hemos ordenado por la coordenada x de menor a mayor, teniendo en cuenta también para puntos cercanos en x, la coordenada y). Una vez ordenados los puntos, simplemente recorremos el vector de puntos en el orden en el que se encuentren, de esta forma, el camino total trata de reducir cruces y distancias con la idea intuitiva de que si dos puntos están cerca en el plano, probablemente, la distancia que se recorra al pasar por ellos, sea mínima. Como veremos en el siguiente algoritmo y en la conclusión, esto en la práctica no es del todo cierto. Primero veamos la implementación del algoritmo:

```
vector<Point> orderedTSP(const vector<Point>& points) {
      vector<Point> tour;
      tour.reserve(points.size());
      copy(points.begin(), points.end(), back_inserter(tour));
      sort(tour.begin(), tour.end());

      return tour;
}
```

Como se puede apreciar, la implementación es muy sencilla, simplemente se copian los puntos de partida a un vector auxiliar, se ordenan y se devuelve el vector ordenado, sin embargo cuando vemos el resultado de la ejecución, nos damos cuenta de que aunque los puntos esten muy cerca respecto a la coordenada x, los saltos que se dan en la coordenada y pueden ser muy grandes, lo que hace que el camino total sea mucho mayor de lo esperado, además para cerrar el camino, la distancia es la máxima posible en cuanto a x. Para mejorar esto, podemos observar que si nuestro problema son los saltos en la coordenada y y la distancia de cierre, podemos intentar minimizarlos, de donde surge el siguiente algoritmo.

3.3 Algoritmo *Circular*