Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación
CC4102 - Diseño y Análisis de Algoritmos

Tarea 3
Eucludean Traveling Salesman Problem

Autores: Claudio Berroeta
Sebastián Ferrada
Profesor: Pablo Barceló
Auxiliar: Miguel Romero
Ayudantes: Javiera Born
Giselle Font
2 de julio de 2014

Índice

1. Introducción

En el presente informe se pretende mostrar un análisis sobre la performance de distintas aproximaciones para resolver el problema del vendedor viajero. Este problema requiere un recorrido hamilnoniano tal que el vendedor visite las ciudades de un país solo una vez, recorriendo la menor distancia posible. Es bien conocido que la resolución de este problema de forma óptima es $NP - Hard$, i.e., toma tiempo exponencial.
A continuación, se revisarán 3 formas de aproximarse a este problema. La primera de ellas utiliza Minimum Spanning Trees sobre un grafo completo, en el cual cada ciudad es un nodo, y el peso de las aristas corresponde a la distancia de las ciudades. Luego, se recorre el árbol construído en preorden y se tiene un circuito de largo a lo más 2 veces el óptimo.
Otra aproximación consiste en ir desde cada ciudad, a la ciudad más cercana. Esta también es una 2-aproximación al problema.
Finalmente, se estudiará la performance de una heurística, es decir, un algoritmo que suele funcionar bien, pero que no entrega garantías sobre sus resultados, por lo que pueden llegar a ser bastante malos en ciertas ocaciones. Esta heurística calcula la envoltura convexa sobre las ciudades y luego se buscan puntos que minimicen ciertas distancias y se van agregando al circuíto.
Además de revisar cuántas veces más distancia se recorre con los algoritmos respecto al óptimo, se medirán los tiempos que toma la ejecución de estos.

1.1. Hipótesis

1. Se espera que la altura de los árboles sea $O(\log(n))$, pues cada árbol intentará particionar el conjunto de puntos en dos mitades con la misma cantidad de elementos cada una.

2. Dado lo anterior, se espera que la construcción tome tiempo $O(n \log(n))$, pues recorremos los n puntos en los $\log(n)$. niveles que se espera tener.

3. Dada la topología de la estructura, se espera que las consultas de vecino más cercano, se ejecuten en tiempo $O(\log(n))$.

4. Además, se espera que el espacio utilizado por el árbol no sea superior a $O(n)$.

5. Dado que tanto la media, como la mediana son medidas de centralización casi igualmente certeras, no se espera que la elección de uno por sobre el otro cause diferencias importantes en cuanto a la eficiencia de la construcción del árbol o del balanceo de la estructura resultante, lo que implica que tampoco habría una optimización en el tiempo de consulta ni en la altura del árbol ni en el espacio utilizado.

6. El tener puntos de baja discrepancia, se espera que nos permita descartar ramas completas del árbol en el segundo recorrido en la consulta de punto más cercano, pues hay una menor probabilidad de intersección/colisión respecto a una distribución aleatoria de puntos, por lo que se espera que las consultas en árboles construidos sobre puntos de baja discrepancia sean más rápidas.

7. Claramente, los experimentos en memoria secundaria demorarán mayor tiempo en ejecutarse, con $O(n)$ accesos a disco, en promedio durante la construcción del árbol (n escrituras) y cantidad logarítmica de accesos para la búsqueda (Asumiendo un nodo por bloque, sin buffer).

1.2. Ambiente Operacional

Los experimentos fueron ejecutados en un Notebook con Sistema Operativo Windows 8 de 64 bits, 8 Gb de memoria RAM y procesador intel Core i7 de 3.630 GHz. Para la implementación se utilizó el Java Development Kit v1.7.

2. Diseño de las Aproximaciones y Experimentos

Cada aproximación será implementada por una clase que extienda de la interfaz `EuclideanTSPResolver` cuyo método recibirá la lista de puntos debidamente leídos desde los archivos indicados[1]

2.1. Aproximación por Punto más Cercano

Un árbol tiene un nodo raíz, sabe como construirse a partir de un set de puntos y es capaz de recibir consultas de vecino más cercano a un punto. Este árbol debe "decidir" cómo particionar los nodos, es por esto que se crearon dos tipos de árboles, un árbol que particiona por mediana y otro por media aritmética. Entonces usando un Template Pattern, el árbol KD genérico sabe cómo construirse y son los hijos los que definen el criterio para seleccionar la recta de partición, según el siguiente diagrama de clases:

Por otro lado, son Nodos quienes guardan la información del árbol, pues este solo tiene un puntero al nodo raíz. Hay nodos de dos tipos, los internos guardan las rectas que particionan el espacio y las hojas guardan la información de los puntos, por lo que se tiene el siguente diagrama de clases:

Los nodos también son los responsables de recorrer el árbol durante las consultas, hacia los hijos y hacia el padre, además son los que calculan la intersección entre los sectores que delimitan y el círculo de consulta.

2.2. Aproximación por Envoltura Convexa

La construcción se hace recursivamente, con caso base un conjunto de puntos de tamaño 1. En cada fase se particiona el conjunto de acuerdo al criterio de mediana o media, intercalando el eje en cada nivel. El código es bastante simple y se presenta a continuación:

```
public KDNode constructKdtree(List<KDPoint> points, Axis axis){
       if(points.size() == 1 ){
           return new KDLeaf(points.get(0));
       }
       KDLine line = getLine(points, axis) ; //This line depends on the Tree citeria
       List<List<KDPoint>> partition = makePartition(points,line);
       return new KDInternalNode(line,
                               constructKdtree(partition.get(0),axis.negated()),
                               constructKdtree(partition.get(1),axis.negated()));
   }
```

Las diferentes implementaciones de getLine y el método partition pueden encontrarse en los anexos

[1] http://www.math.uwaterloo.ca/tsp/world/countries.html

2.3. Diseño de los Experimentos

Los experimentos consistieron en construir árboles KD con arreglos de puntos de tamaños sucesivamente más grandes (tamaño 2^i, $i \in \{10, ..., 20\}$ en las diferentes combinaciones de criterio de partición y de distribución de puntos y medimos su tiempo, la altura del árbol y la cantidad de espacio en memoria que utiliza. Luego a cada sabor de árbol se le realizan consultas y se mide el tiempo que demora en contestar. Cada experimento se repite las veces necesarias para que el error sea menor al 5 % (i.e., hasta que $\frac{stdv}{mean} \leq 0{,}05$). Para detalles de la implementación de la batería de experimentos, referirse al código fuente adjunto.

3. Resultados

Los diferentes criterios de partición y tipo de distribución de puntos nos generaban 4 instancias de análisis para cada parámetro a medir.

3.1. Canadá

El experimento consistía en medir cuánto demora en ser contruído el árbol. Los resultados se pueden visualizar en el siguiente gráfico:

A simple vista podemos confundir el orden de construcción, pero analizando por separado podemos decir que el tiempo de construcción es del orden de $O(nlog(n))$.

3.2. Djibouti

El experimento consistía en calcular la altura que alcanzaban los KDTrees de un tamaño determinado. Los resultados se pueden visualizar en el siguiente gráfico:

Claramente podemos apreciar que las alturas de los árboles son del orden de $O(log(n))$, destacando que los árboles con puntos distribuidos de manera uniforme tienen una altura mas baja en comparación a los árboles de distribución aleatoria.

3.3. Finland

El experimento consistía en medir cuánto demoraban en encontrar los vecinos más cercanos de una serie de puntos. Los resultados se pueden visualizar en el siguiente gráfico:

Se puede apreciar que el tiempo de consulta es del orden $O(log(n))$, también se puede visualizar que los puntos de baja discrepancia demoran un poco menos en comparación a los puntos aleatorios.

3.4. Greece

El experimento consistía en saber cuánto espacio ocupa cada KDTree, se usaron valores constantes en bytes para cada nodo.

Podemos ver que el tamaño de un KDTree es del orden $O(n)$ y además que para cada tipo de KDTree el tamaño ocupado es el mismo, esto se debe a que se insertan la misma cantidad de nodos y no existen estructuras externas.

3.5. Italy

3.6. Japan

3.7. Oman

3.8. Qatar

3.9. Sweden

3.10. Uruguay

3.11. Vietnam

3.12. Western Sahara

3.13. Zimbabwe

4. Conclusiones

1. Efectivamente la altura de los árboles es del orden $O(\log(n))$ y además los KDTrees creados a partir de puntos de baja discrepancia poseen una menor altura, es decir la construcción es un poco más balanceada, notando que la diferencia no es completamente sustancial, pues la cantidad de nodos del árbol (espacio utilizado) es igualmente lineal.

2. La construcción de KDTrees tomó tiempo $O(n\log(n))$, tal como fue previsto. Aunque los tiempos entre los árboles con criterio $mean$ y $median$ son muy distantes, lo atribuimos al $overhead$ del cálculo de la mediana para cada conjunto de puntos, el cual es $O(n)$.

3. También se comprobó que las consultas de vecino más cercano se ejecutaron en tiempo $O(\log(n))$ para todo tipo de construcción y sin importar la distribución de puntos.

4. Se verificó que el espacio ocupado por el árbol es del orden de $O(n)$. Todos los árboles de la misma cantidad de puntos ocupan el mismo espacio, luego tienen similar o idéntica cantidad de nodos, luego la diferencia en alturas entre árboles construídos según mediana y degún media son discordantes solo por un par de ramas y no por el último nivel completo.

5. Nuestra hipótesis sobre las diferencias sobre la construcción del árbol fue errónea, dado que en la construcción del árbol el $mean$ fue mucho menor al otro, esto debido a que calcular min y max resultó ser más rápido que calcular la mediana..

6. Que los puntos estén distribuidos de una manera uniforme presentó leves mejoras en cuanto a los tiempos de consulta, tal como se predijo en la hipótesis, aunque esta diferencia demostró ser bastante leve.

5. Anexos

5.1. Criteros de Selección de Línea de Partición

Cadanodo, particiona el espacio según la recta que corta al eje del nivel en la mediana o en la media de la coordenada correspondiente en cada set de puntos.
Selección según media

```
protected KDLine getLine(List<KDPoint> points, Axis axis) {
       return new KDLine(axis, calcMean(points,axis)); //mean between min and max of
           point set
    }
```

Selección según mediana

```
protected KDLine getLine(List<KDPoint> points, Axis axis) {
       return new KDLine(axis, calcMean(points,axis)); //median calculated over
           groups of five
    }
```

5.2. Vecino más cercano

Primero se busca el nodo que marca el mismo cuadrante que el punto de consulta

```
/*En nodos internos*/
public KDLeaf searchNeighbor(KDPoint q) {
       if(q.getCoord(line.getAxis())<= line.getPos()){
           return left.searchNeighbor(q);
       }
       else
         return right.searchNeighbor(q);
 }

 /* en nodos hoja*/
 public KDLeaf searchNeighbor(KDPoint q) {
       return this; //To change body of implemented methods use File | Settings |
           File Templates.
}
```

Luego se busca un best fit

```
/*En nodos internos*/
public KDLeaf anotherSearch(KDNode aChild, double currentDistance, KDPoint q) {
        KDLeaf bestLeft = new KDLeaf(new KDPoint(Double.POSITIVE_INFINITY,
            Double.POSITIVE_INFINITY)),
               bestRight = new KDLeaf(new KDPoint(Double.POSITIVE_INFINITY,
                   Double.POSITIVE_INFINITY));
        if(left!=aChild && left.intersects(q,currentDistance)){
            bestLeft = left.anotherSearch(aChild, currentDistance, q);
        }
        if(right!=aChild && right.intersects(q, currentDistance)){
            bestRight = right.anotherSearch(aChild, currentDistance, q);
        }
        return (bestLeft.distance(q)>bestRight.distance(q))? bestRight:bestLeft;
}
/*En nodos hoja */
public KDLeaf anotherSearch(KDNode currentBest, double currentDistance, KDPoint q){
    if( point.distance(q)<currentDistance){
        return this;
    }
    return new KDNullNode(null);//Contiene un punto a distiancia infinita de cualquier
        otro
 }
```