El problema del viajante de comercio
Inteligencia Artificial

Vicente J. Ferrer Dalmau (43152879A)
<vicente@jdalmau.es>

1 de abril de 2008

Índice

1. Enunciado

Un señor viajante de comercio debe visitar n ciudades en su trabajo diario $c_1, c_2, ...c_n$ de manera que una de ellas, la ciudad c_i, constituye el punto de partida y llegada del camino a seguir en la visita. Se considera que existen rutas directas entre cada par de ciudades del conjunto considerado, de manera que cada ruta tiene un costo específico asociado. El problema consiste en determinar, partiendo y llegando a una misma ciudad c_i, una ruta óptima que permita visitar las n ciudades consideradas en un tiempo mínimo. El mencionado problema debe resolverse aplicando cada una de las siguientes técnicas:

TÉCNICAS DE BÚSQUEDA NO INFORMADA

- Primero en profundidad.
- Primero en profundidad con ramificación y poda.
- Búsqueda de costo uniforme.

TÉCNICAS DE BÚSQUEDA INFORMADA

- El vecino más próximo.
- Método de escalada.
- Búsqueda A*, en la que debe diseñarse –como mínimo- una heurística haciendo uso de la técnica del problema relajado, para calcular la función h'.

Debe hacerse un análisis de cada una de las técnicas empleadas, determinando los casos en que cada técnica funciona mejor/peor; asimismo, deben compararse las diferentes técnicas, concluyendo sobre las ventajas de cada una respecto a las demás.

Respecto a la búsqueda A*, debe diseñarse al menos una heurística para el cálculo de h', utilizando para ello la técnica del problema relajado.

Se valorará la capacidad de iniciativa de los estudiantes además de la presentación de los resultados y la calidad del análisis de cada técnica.

2. Búsqueda primero en profundidad

El algoritmo Primero en Profundidad es un algoritmo que nos permite realizar búsquedas en un árbol o grafo. Formalmente es una búsqueda no informada que progresa expandiendo el primer nodo hijo en el árbol de búsqueda y continua hasta encontrar un nodo objetivo o hasta encontrar un nodo sin hijos. En ese momento, la búsqueda realiza una vuelta atrás (backtracking) volviendo al nodo más reciente que no había sido explorado completamente, para elegir el siguiente de sus hijos.

Características del algoritmo

Aspectos positivos:

- La **memoria** necesaria es relativamente pequeña, ya que en cada momento sólo hay que guardar la ruta que se está analizando.
- Si hay muchas soluciones posibles, la búsqueda en profundidad es **rápida** ya que tiene muchas posibilidades de encontrar una solución después de haber explorado únicamente una parte del árbol. En nuestro caso esta ventaja no se da porque tenemos que recorrer todo el árbol para dar la mejor solución.

Aspectos negativos:

- Tiene un coste temporal **exponencial**, lo que para problemas grandes lo hace poco recomendable (en nuestro caso un número elevado de ciudades).
- En algunos casos el recorrido de ciertas rutas puede conllevar problemas, por ejemplo si tenemos rutas cíclicas o infinitas quedaríamos **estancados**, o bien podríamos encontrar soluciones no óptimas después de grandes recorridos, o incluso llegar a una solución óptima pero después de haber analizado muchas rutas incorrectas. En definitiva y continuando con la conclusión de la primera desventaja, si la estructura sobre la que realiza la búsqueda es muy grande o infinita no es recomendable usar este algoritmo.

Evaluación del algoritmo de búsqueda

Completitud: No, ya que puede quedarse atrapado al descender por el camino equivocado en árboles de búsqueda muy profundos o incluso infinitos. En esos casos se encuentra con un bucle infinito y no devuelve nunca una solución.

Complejidad temporal: b^m, lo que supone un coste inaceptable[1].

Complejidad espacial: bm, sólo necesita almacenar una rama del árbol de búsqueda.

Resultados

A continuación se muestra la evolución temporal del algoritmo en función del número de ciudades del problema. Para cada una de las ejecuciones se atenderá al número de nodos que han sido expandidos durante la búsqueda (en el caso de este algoritmo siempre se expande el árbol completo):

Figura 1: El valor se dispara entre 10 y 11 ciudades.

[1] Dónde **b** es el factor de ramificación, en nuestro caso las ciudades aplicables, **d** es la profundidad de la solución y **m** es la máxima profundidad del árbol de búsqueda.

Número de ciudades	Nodos expandidos
4	22
5	89
7	2677
8	18740
9	149921
10	1349290
11	13492901

Cuadro 1: Valores obtenidos en la ejecución.

En el gráfico puede observarse el claro comportamiento **exponencial** del algoritmo. Cabe mencionar que sólo fue posible ejecutarlo con **11 ciudades** cómo máximo.

3. Búsqueda primero en profundidad con ramificación y poda

El algoritmo primero en profundidad con Ramificación y poda es una variante del algoritmo visto en el apartado anterior mejorado sustancialmente. El término (del inglés, Branch and Bound) se aplica mayoritariamente para resolver cuestiones o problemas de optimización.

La característica de esta técnica con respecto a la anterior es que el algoritmo se encarga de detectar en qué ramificación las soluciones dadas ya no están siendo óptimas (es decir, mejoran alguna obtenida préviamente), para «podar» esa rama del árbol y no continuar malgastando recursos y procesos en casos que se alejan de la solución óptima.

Características del algoritmo

Aspectos positivos:

- Bajo consumo de memoria.
- Rápido si existen muchas soluciones.
- Permite su aplicación para un mayor número de ciudades que en el caso anterior.

Aspectos negativos:

- Coste temporal todavía exponencial.

Evaluación del algoritmo de búsqueda

Completitud: No, ya que, de nuevo, puede quedarse atrapado al descender por el camino equivocado en árboles de búsqueda muy profundos o incluso infinitos. En esos casos se encuentra con un bucle infinito y no devuelve nunca una solución.

Complejidad temporal: exponencial.

Complejidad espacial: lineal.

Resultados

En este apartado se muestra la evolución temporal del algoritmo en función del número de ciudades del problema. Para cada una de las ejecuciones se atenderá al número de nodos que han sido expandidos durante la búsqueda. Hay que decir que para un número de ciudades constante, al variar los valores de su matriz de distancias, el número de nodos expandidos variará, es decir, el porcentaje del árbol de búsqueda que se visita varía.

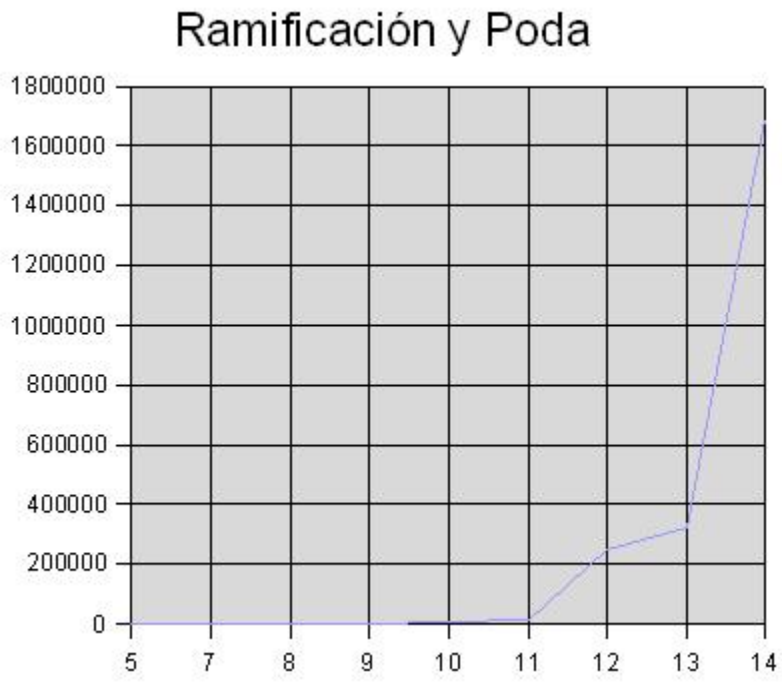

Figura 2: Número de nodos expandidos en función del número de ciudades.
Aquí el valor se dispara con unas cuantas ciudades más que
en el caso anterior.

Número de ciudades	Nodos expandidos
4	18
5	81
7	259
8	708
9	2253
10	8345
11	16149
12	246241
13	319918
14	1689825

Cuadro 2: Valores obtenidos en la ejecución.

De nuevo en la gráfica observamos el comportamiento **exponencial** del algoritmo, aunque aquí el crecimiento importante se produce para un valor más elevado de ciudades. Este algoritmo permite ejecutarse con un valor máximo de **14 ciudades**.

4. Búsqueda de costo uniforme

Mediante este algoritmo la búsqueda se realiza expandiendo todos los nodos hijo de un nodo padre y seleccionando aquel cuyo coste sea menor. Se procede de esta forma hasta que se encuentra la mejor solución. Observemos que si todos los nodos tienen el mismo coste tenemos un recorrido en anchura.

La búsqueda de coste uniforme encontrará la mejor solución si se cumple que el coste del camino nunca decrece a medida que avanzamos, requisito que se cumple para el problema del viajante de comercio. Para otro tipo de problemas dónde si pudiera producirse nunca sabríamos cuándo podríamos encontrar un coste negativo. El resultado provocaría la necesidad de realizar una búsqueda exhaustiva en todo el árbol.

Este algoritmo puede implementarse fácilmente con la ayuda de una estructura de datos **cola ordenada por prioridad**, dónde aquellos nodos con menor coste (del nodo inicial hasta dicho nodo) son más prioritarios.

Características del algoritmo

Aspectos positivos:

- Es un algoritmo de búsqueda completo, si existe una solución siempre la encuentra.
- Es un algoritmo de búsqueda óptimo, siempre encuentra la mejor solución.

Aspectos negativos:

- La complejidad temporal sigue siendo elevada.
- Los requerimientos de espacio son elevados en tanto que hay que guardar los nodos de cada nivel, ya que constituyen posibles soluciones.

Evaluación del algoritmo de búsqueda

Completitud: Sí.

Complejidad temporal: b^d dónde d representa la profundidad de la solución.

Complejidad espacial: b^d.

Resultados

En este apartado se muestra la evolución temporal del algoritmo en función del número de ciudades del problema. Para cada una de las ejecuciones se atenderá al número de nodos que han sido expandidos durante la búsqueda. Hay que decir que para un número de ciudades constante, al variar los valores de su matriz de distancias, el número de nodos expandidos variará, es decir, el porcentaje del árbol de búsqueda que se visita varía. Además al mantenerse una cola de prioridad para los nodos abiertos es frecuente ver cómo el programa se queda sin memoria, para algunas configuraciones de esa misma matriz de distancias, y no puede completar la búsqueda; tal efecto se pone de manifiesto a partir de **12 ciudades**.

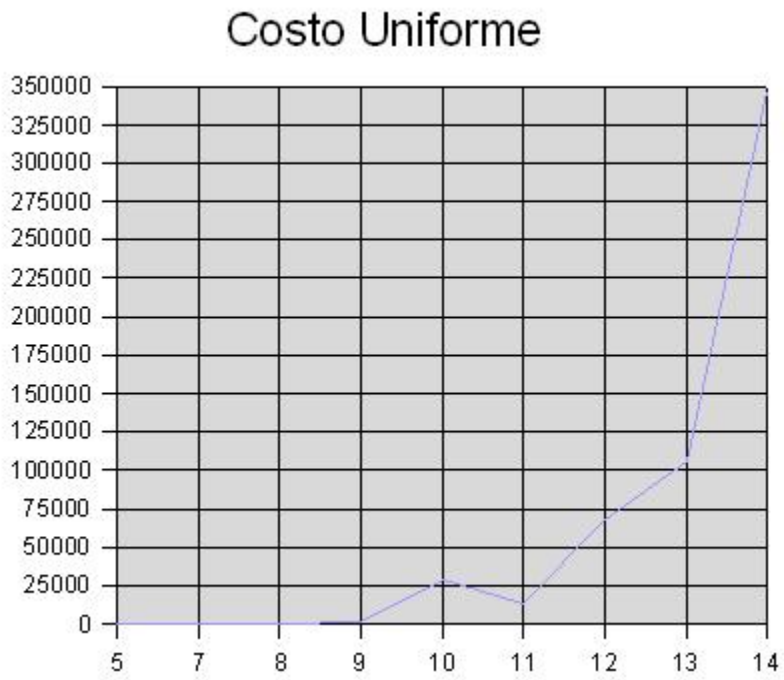

Figura 3: Número de nodos expandidos en función del número de ciudades.

Número de ciudades	Nodos expandidos
4	13
5	42
7	136
8	641
9	1703
10	29221
11	13595
12	67942
13	106030
14	347424

Cuadro 3: Valores obtenidos en la ejecución.

De nuevo en la gráfica observamos el comportamiento **exponencial** del algoritmo, aunque aquí hay que tener muy en cuenta el tema de **memoria** (cola de prioridad para los nodos abiertos) cómo ya se mencionó anteriormente. Además atendiendo al tiempo de ejecución, pese a visitar menos nodos resulta por término general más **lento** que el anterior algoritmo (Ramificación y Poda) debido de nuevo a la necesidad de gestionar la cola de prioridad. Este algoritmo permite ejecutarse con un valor máximo de **14 ciudades**.

5. Búsqueda A estrella

El algoritmo A estrella es un algoritmo de búsqueda para grafos que encuentra el camino de menor coste entre un nodo inicial y un nodo meta.

Usa una función heurística (denotada f'(x), es una aproximación a f(x), función que proporciona la verdadera evaluación de un nodo) para determinar el orden en que la búsqueda visita nodos en el árbol. La mencionada función es la suma de otras dos funciones: una función que indica el coste del camino seguido hasta un cierto nodo (denotada g(x)) y una estimación admisible de la distancia hasta la meta (h'(x)). La función de evaluación resulta entonces $f'(x) = g(x) + h'(x)$.

Empezando en un nodo inicial dado, el algoritmo expande el nodo con el menor valor de f'(x). A estrella mantiene un conjunto de soluciones parciales almacenadas en una **cola de prioridad**, cómo en el algoritmo del apartado anterior. La prioridad asignada a un camino x viene determinada por la función f'(x). El proceso continua hasta que una meta tiene un valor f'(x) menor que cualquier otro nodo en la cola (o hasta que el árbol ha sido completamente recorrido).

Características del algoritmo

Aspectos positivos:

- Ningún otro algoritmo óptimo garantiza expandir menos nodos que A estrella.

Aspectos negativos:

- Alto consumo de memoria.

Evaluación del algoritmo de búsqueda

Completitud: Sí.

Complejidad temporal: exponencial (debido a la heurística utilizada).

Complejidad espacial: exponencial.

Resultados

En cuanto a la **función heurística** que se ha utilizado para resolver este problema, para obtener el valor de la estimación **h'(x)** dado un nodo consideramos la profundidad a la que se encuentra dentro del árbol de búsqueda. Cuanto más cercano esté al número de ciudades del problema (por tanto a una solución) mejor valoración tendrá. Para calcular dicho valor consideraremos además de la profundidad del nodo una estimación del coste del camino entre dos ciudades cualquiera:

Si por ejemplo, los costes de las rutas entre las ciudades del problema pertenecen a una distribución uniforme U(100) y tomamos un coste aproximado de 15 unidades cómo coste entre dos ciudades cualquiera, tendremos:

$h'(x) = (n - l(x)) * 15$

dónde**:**

n representa el número de ciudades del problema.

l(x) es la profundidad del nodo x.

En el caso considerado el valor h'(x) puede sobreestimar al valor real del coste de la ruta entre el nodo considerado y un nodo meta, ya que al tener valores de una U(100) óbviamente tendremos rutas con costes mayores a 15 (constante considerada). Lo anterior implica que en este caso el algorimo no es **óptimo**. A pesar de lo anterior tras ejecutar el algoritmo se ha comprobado que las soluciones ofrecidas por el algoritmo son prácticamente siempre las mejores, y en caso de no serlo, son bastante aproximadas.

Veamos ahora la evolución temporal del algoritmo en función del número de ciudades del problema. Para cada una de las ejecuciones se atenderá al número de nodos que han sido expandidos durante la

Número de ciudades	Nodos expandidos
5	37
7	422
10	808
13	8528
20	223
50	232058
75	14370
100	5278

Cuadro 4: Valores obtenidos en la ejecución.

búsqueda. Hay que decir que para un número de ciudades constante, al variar los valores de su matriz de distancias, el número de nodos expandidos variará, es decir, el porcentaje del árbol de búsqueda que se visita varía.

Pese a utilizar de nuevo colas de prioridad aquí la memoria no supone tanto problema cómo en el algorimo anterior ya que A estrella converge mucho más deprisa hacia una solución.

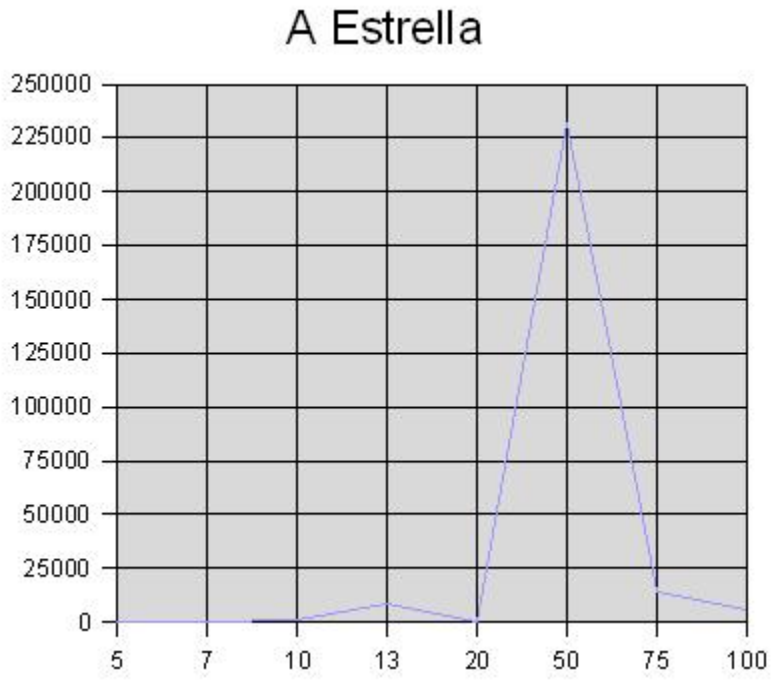

Figura 4: Nodos expandidos en función del número de ciudades.

Atendiendo al gráfico no se muestra el carácter **exponencial** del algoritmo ya que puede ejecutarse para más de 100 ciudades. Cómo se dijo en clase el principal problema de este algoritmo es que suele quedarse sin memoria y así ocurre cuándo se ha intentado probar para más de 100 ciudades. Aún así

no se debe dudar de su coste exponencial en este caso ya que el error de la función heurística crece más rápido que el logaritmo del costo real de la ruta.

Este algoritmo permite ejecutarse con un valor máximo de **100 ciudades**, si no se presentan problemas de memoria (tamaño de las colas de prioridad).

6. Búsqueda del vecino más próximo

El algoritmo del Vecino más próximo fué uno de los primeros algoritmos usados para encontrar la solución al problema del viajante de comercio. Rápidamente encuentra una solución pero generalmente ésta no es la **óptima**.

Este algoritmo es fácil de implementar y se ejecuta rápido pero puede en ocasiones perder rutas más cortas. Cómo guía general, si las últimas etapas de la ruta son comparables en coste a las primeras, entonces la ruta será razonable; si son mucho más grandes la solución hallada será muy mejorable.

Características del algoritmo

Aspectos positivos:

- Es un algoritmo muy rápido.
- Su consumo de memoria muy bajo.

Aspectos negativos:

- No es óptimo.

Evaluación del algoritmo de búsqueda

Completitud: Sí.

Complejidad temporal: O(m)

Complejidad espacial: O(m)

Resultados

Veamos ahora la evolución temporal del algoritmo en función del número de ciudades del problema. Para cada una de las ejecuciones se atenderá al número de nodos que han sido expandidos durante la búsqueda.

Figura 5: Nodos expandidos en función del número de ciudades.

Número de ciudades	Nodos expandidos
5	6
10	11
20	21
50	51
75	76
100	101

Cuadro 5: Valores obtenidos en la ejecución.

Observando el gráfico queda patente el caracter **lineal** del algoritmo.

Este algoritmo prescinde de la optimalidad en la solución para lograr un tiempo de ejecución mínimo. Se recomienda usar este algoritmo cuándo no se dispone de gran poder computacional y/o no se necesita encontrar la mejor solución.

En general este algoritmo no suele dar malas soluciones. Es muy improbable que de el peor de los casos. Puede ejecutarse para **cientos de ciudades**.

7. Método de escalada

El algoritmo de escalada consiste en recorrer las distintos nodos explorando en cada momento aquel cuyo coste f(x)', que incluye el coste real de la ruta g(x), y el coste de h(x)', aproximación a h(x), sea menor.

Constituye un caso particular de una búsqueda en profundidad sin posibilidad de vuelta atrás.

Es parecido al algoritmo del apartado anterior, pero para el problema de esta práctica no tiene sentido utilizarlo, ya que no obtiene buenas rutas.

Características del algoritmo

Aspectos positivos:

- Es un algoritmo tan rápido cómo el Vecino más Próximo.
- Tiene un bajo consumo de memoria.

Aspectos negativos:

- No es óptimo.
- Puede quedar estancado.

Evaluación del algoritmo de búsqueda

Completitud: No, es irrevocable (no puede volver atrás).

Complejidad temporal: O(m)

Complejidad espacial: O(m)

Resultados

Se muestra a continuación la evolución temporal del algoritmo en función del número de ciudades del problema. Para cada una de las ejecuciones se atenderá al número de nodos que han sido expandidos durante la búsqueda.

Figura 6: Nodos expandidos en función del número de ciudades.

Número de ciudades	Nodos expandidos
5	6
10	11
20	21
50	51
75	76
100	101

Cuadro 6: Valores obtenidos en la ejecución.

Queda claro por lo tanto que es muy parecido al anterior algorimo siendo también **lineal.** Aquí es imposible evaluar lo bueno que es un estado por sí mismo (habría que hacerlo en relación a las demás rutas), con lo cual es mejor **evitar el uso** de este algoritmo.

8. Comparativa

Veamos ahora una serie de gráficos que dejen constancia del número de nodos expandidos para distinto número de ciudades aplicando los seis algoritmos vistos en secciones anteriores. Para cierto número de ciudades no todos los algoritmos podrán ser ejecutados.

Figura 7: Comparativa para 11 ciudades.

Algoritmo	Nodos expandidos
Primero en Profundidad (A)	13492901
Ramificación y Poda (B)	16149
Costo Uniforme (C)	13595
A Estrella (D)	813
Vecino más Próximo (E)	12
Escalada (F)	12

Cuadro 7: Comparativa para 11 ciudades.

Figura 8: Comparativa para 14 ciudades.

Algoritmo	Nodos expandidos
Ramificación y Poda (B)	1689825
Costo Uniforme (C)	347424
A Estrella (D)	699
Vecino más Próximo (E)	15
Escalada (F)	15

Cuadro 8: Comparativa para 14 ciudades.

Figura 9: Comparativa para 100 ciudades.

Algoritmo	Nodos expandidos
A Estrella (D)	5278
Vecino más Próximo (E)	101
Escalada (F)	101

Cuadro 9: Comparativa para 100 ciudades.

Conclusiones

- Para el algoritmo de Búsqueda en Profundidad el número de nodos visitados crece desorbitadamente en comparación con los demás algoritmos únicamente para 11 ciudades.
- Los algoritmos de Ramificación y Poda y Costo Uniforme permiten ejecutarse con unas pocas ciudades más que el algoritmo anterior, concretamente 14. Además pese a que el algoritmo de Costo Uniforme expande menos nodos su ejecución es más lenta que el de Ramificación y poda; todo se debe a que debe gestionar una cola de prioridad.
- A Estrella presenta el mejor comportamiento. Permite obtener en la mayoría de casos una solución óptima con muy pocos nodos recorridos.
- Los algoritmos del Vecino más Próximo y Escalada recorren un número lineal de nodos pero no garantizan encontrar la mejor solución, por eso son considerados peores que A Estrella.
- Para concluir puede afirmarse que dada una gran cantidad de ciudades, no debe utilizarse el algoritmo de Búsqueda en Profundidad, la **mejor opción será aplicar el algoritmo A Estrella** que encontrará la mejor solución en muy poco tiempo.

9. Código fuente

Esta práctica se ha implementado mediante el lenguaje Java (entorno Netbeans 5.5). Para ejecutar la aplicación se adjunta un archivo con extensión jar (TravelingSalesMan.jar) en la carpeta dist de ejecución directa en sistemas operativos de tipo Windows y Linux.

Consideraciones previas:

- *Rutas*: Las distancias entre las diferentes ciudades del problema se han recogido en forma de matriz n*n siendo n el número de ciudades. Así la diagonal de la misma es cero y además el valor de la ruta de la ciudad **A** a la **B** coincide con el valor de **B** a **A**. Dicha matriz aparece en la clase RoutesMatrix.

A continuación se muestran las clases en las que se divide la aplicación.

9.1. AStar

```
1  /*
2   * AStar.java
3   *
4   * $LastChangedDate: 2008-03-30 22:35:24 +0200 (dom, 30 mar 2008) $
5   * $LastChangedRevision: 15 $
6   * Vicente J. Ferrer Dalmau
7   * < vicente@jdalmau.es >
8   *
9   * Implementation of the A* (A Star) algorithm.
10  */
11
12 package travelingsalesman;
13
14 import java.util.ArrayList;
15 import java.util.Comparator;
16 import java.util.PriorityQueue;
17 import java.util.Vector;
18
19 /**
20  *
21  * @author Vicente J. Ferrer Dalmau
22  */
23 public class AStar {
24
25     RoutesMatrix distances;
26     int sourceCity;
27     PriorityQueue<Town> opened = new PriorityQueue<Town>(1000,
28         new Comparator<Town>() {
29           public int compare(Town a, Town b) {
30             return a.f - b.f;
31           }
32         }
33       );
34     String result = new String();
35
36     ArrayList optimumRoute;
```

```
37      int nodes = 0;
38      int optimumCost = Integer.MAX_VALUE;
39
40      // Estimation of the cost between two cities, it can overestimate the
            real value (h' > h),
41      // so the algorithm it's not optimum.
42      int HEURISTICCONSTANT = 15;
43
44      /**
45       * Gets the heuristic value for a given depth
46       * The level 0 has the maximum value.
47       */
48      private int getHeuristicValue (int level) {
49
50          return HEURISTICCONSTANT * (distances.getCitiesCount() - level);
51      }
52
53      /** Creates a new instance of AStar */
54      public AStar(RoutesMatrix matrix, int sourceCity) {
55
56          distances = matrix;
57          this.sourceCity = sourceCity;
58      }
59
60      /**
61       * executes the algorithm
62       */
63      public String execute() {
64
65          // have we found the solution?
66          boolean solution = false;
67
68          // start the timer
69          long startTime = System.currentTimeMillis();
70
71          // initial town
72          opened.add(new Town(sourceCity, 0, getHeuristicValue(0), 0));
73
74          while (!opened.isEmpty() && !solution) {
75              // gets the city with lower g value
76              Town currentTown = opened.poll();
77              nodes++;
78
79              // rebuild the followed route for the selected town
80              Town aux = currentTown;
81              ArrayList followedRoute = new ArrayList();
82              followedRoute.add(aux.number);
83              while (aux.level != 0) {
84                  aux = aux.parent;
85                  followedRoute.add(0, aux.number);
86              }
87
```

```
88              if (currentTown.level == distances.getCitiesCount()) {
89                  solution = true;
90                  optimumRoute = followedRoute;
91                  optimumCost = currentTown.g;
92              } else {
93
94                  for (int i=0; i<distances.getCitiesCount(); i++) {
95                      // have we visited this city in the current followed
                            route?
96                      boolean visited = followedRoute.contains(i);
97                      boolean isSolution = (followedRoute.size() ==
                            distances.getCitiesCount())&&(i == sourceCity);
98
99                      if (!visited || isSolution) {
100                         Town childTown = new Town(i, currentTown.g +
                                distances.getCost(currentTown.number, i),
101                                     getHeuristicValue(currentTown.level + 1),
                                            currentTown.level + 1);
102                         childTown.parent = currentTown;
103                         opened.add(childTown);
104                     }
105                 }
106             }
107         }
108         long endTime = System.currentTimeMillis();
109
110         result = "------------------------------------------\n";
111         result +=  "A_ESTRELLA:\n";
112         result += "------------------------------------------\n";
113         result += "MEJOR_SOLUCIÓN:_\t"+optimumRoute.toString() + "\nCOSTE
                :_\t\t"+optimumCost+"\n";
114         result += "NODOS_VISITADOS:_\t"+nodes+"\n";
115         result += "TIEMPO_TRANSCURRIDO:_\t"+(endTime-startTime)+"_ms\n";
116         result += "------------------------------------------\n";
117
118         return result;
119     }
120
121 }
```

9.2. BranchAndBound

```
1  /*
2   * BranchAndBound.java
3   *
4   * $LastChangedDate: 2008-03-30 22:35:24 +0200 (dom, 30 mar 2008) $
5   * $LastChangedRevision: 15 $
6   * Vicente J. Ferrer Dalmau
7   * < vicente@jdalmau.es >
8   *
9   * Implementation of the branch and bound search algorithm.
10  */
```

```
11
12 package travelingsalesman;
13
14 import java.util.ArrayList;
15
16 /**
17  *
18  * @author Vicente J. Ferrer Dalmau
19  */
20 public class BranchAndBound {
21
22     RoutesMatrix distances;
23     int sourceCity;
24     String result = new String();
25
26     ArrayList initialRoute, optimumRoute;
27     int nodes = 0;
28     int routeCost = 0;
29     int optimumCost = Integer.MAX_VALUE;
30
31     /** Creates a new instance of BranchAndBound */
32     public BranchAndBound(RoutesMatrix matrix, int sourceCity) {
33
34         distances = matrix;
35         this.sourceCity = sourceCity;
36     }
37
38     /**
39      * executes the algorithm
40      */
41     public String execute () {
42
43         initialRoute = new ArrayList();
44         initialRoute.add(sourceCity);
45         optimumRoute = new ArrayList();
46         nodes++;
47
48         result = "------------------------------------------\n";
49         result +=  "RAMIFICACIÓN Y PODA:\n";
50         result += "------------------------------------------\n";
51
52         long startTime = System.currentTimeMillis();
53         search(sourceCity, initialRoute);
54         long endTime = System.currentTimeMillis();
55
56         result += "MEJOR SOLUCIÓN: \t"+optimumRoute.toString() + "\nCOSTE
               : \t\t"+optimumCost+"\n";
57         result += "NODOS VISITADOS: \t"+nodes+"\n";
58         result += "TIEMPO TRANSCURRIDO: \t"+(endTime-startTime)+" ms\n";
59         result += "------------------------------------------\n";
60
```

```
61          return result;
62      }
63
64
65      /**
66       * @param from node where we start the search.
67       * @param route followed route for arriving to node "from".
68       */
69      public void search (int from, ArrayList followedRoute) {
70
71          // we've found a new solution
72          if (followedRoute.size() == distances.getCitiesCount()) {
73
74              followedRoute.add(sourceCity);
75              nodes++;
76
77              // update the route's cost
78              routeCost += distances.getCost(from, sourceCity);
79
80              if (routeCost < optimumCost) {
81                  optimumCost = routeCost;
82                  optimumRoute = (ArrayList)followedRoute.clone();
83              }
84
85              // DEBUG
86              //result += followedRoute.toString() + "// COSTE: "+routeCost
                        + "|n";
87
88              // update the route's cost (back to the previous value)
89              routeCost -= distances.getCost(from, sourceCity);
90          }
91          else {
92              for (int to=0; to<distances.getCitiesCount(); to++){
93                  if (!followedRoute.contains(to)) {
94
95                      // update the route's cost
96                      routeCost += distances.getCost(from, to);
97
98                      if (routeCost < optimumCost) {
99                          ArrayList increasedRoute = (ArrayList)
                                followedRoute.clone();
100                         increasedRoute.add(to);
101                         nodes++;
102                         search(to, increasedRoute);
103                     }
104
105                     // update the route's cost (back to the previous
                            value)
106                     routeCost -= distances.getCost(from, to);
107                 }
108             }
109         }
```

```
110     }
111
112 }
```

9.3. DepthFirstSearch

```
1  /*
2   * DepthFirstSearch.java
3   *
4   * $LastChangedDate: 2008-03-30 22:35:24 +0200 (dom, 30 mar 2008) $
5   * $LastChangedRevision: 15 $
6   * Vicente J. Ferrer Dalmau
7   * < vicente@jdalmau.es >
8   *
9   * Implementation of the depth first search algorithm.
10  */
11
12 package travelingsalesman;
13
14 import java.util.ArrayList;
15
16 /**
17  *
18  * @author Vicente J. Ferrer Dalmau
19  */
20 public class DepthFirstSearch {
21
22     RoutesMatrix distances;
23     int sourceCity;
24     String result = new String();
25
26     ArrayList initialRoute, optimumRoute;
27     int nodes = 0;
28     int routeCost = 0;
29     int optimumCost = Integer.MAX_VALUE;
30
31     /** Creates a new instance of DepthFirstSearch */
32     public DepthFirstSearch(RoutesMatrix matrix, int sourceCity) {
33
34         distances = matrix;
35         this.sourceCity = sourceCity;
36     }
37
38     /**
39      * executes the algorithm
40      */
41     public String execute () {
42
43         initialRoute = new ArrayList();
44         initialRoute.add(sourceCity);
45         optimumRoute = new ArrayList();
46         nodes++;
```

```
47
48          result = "-----------------------------------------\n";
49          result +=  "PRIMERO EN PROFUNDIDAD:\n";
50          result += "-----------------------------------------\n";
51
52          long startTime = System.currentTimeMillis();
53          search(sourceCity, initialRoute);
54          long endTime = System.currentTimeMillis();
55
56          result += "MEJOR SOLUCIÓN: \t"+optimumRoute.toString() + "\nCOSTE
                : \t\t"+optimumCost+"\n";
57          result += "NODOS VISITADOS: \t"+nodes+"\n";
58          result += "TIEMPO TRANSCURRIDO: \t"+(endTime-startTime)+" ms\n";
59          result += "-----------------------------------------\n";
60
61          return result;
62      }
63
64      /**
65       * @param from node where we start the search.
66       * @param route followed route for arriving to node "from".
67       */
68      public void search (int from, ArrayList followedRoute) {
69
70          // we've found a new solution
71          if (followedRoute.size() == distances.getCitiesCount()) {
72
73              followedRoute.add(sourceCity);
74              nodes++;
75
76              // update the route's cost
77              routeCost += distances.getCost(from, sourceCity);
78
79              if (routeCost < optimumCost) {
80                  optimumCost = routeCost;
81                  optimumRoute = (ArrayList)followedRoute.clone();
82              }
83
84              // DEBUG
85              //result += followedRoute.toString() + "// COSTE: "+routeCost
                    + "\n";
86
87              // update the route's cost (back to the previous value)
88              routeCost -= distances.getCost(from, sourceCity);
89          }
90          else {
91              for (int to=0; to<distances.getCitiesCount(); to++){
92                  if (!followedRoute.contains(to)) {
93
94                      ArrayList increasedRoute = (ArrayList)followedRoute.
                            clone();
95                      increasedRoute.add(to);
```

```
96                          nodes++;
97
98                          // update the route's cost
99                          routeCost += distances.getCost(from, to);
100
101                         search(to, increasedRoute);
102
103                         // update the route's cost (back to the previous
                                value)
104                         routeCost -= distances.getCost(from, to);
105                     }
106                 }
107             }
108     }
109 }
```

9.4. HillClimbing

```
1  /*
2   * HillClimbing.java
3   *
4   * $LastChangedDate: 2008-03-30 20:41:41 +0200 (dom, 30 mar 2008) $
5   * $LastChangedRevision: 14 $
6   * Vicente J. Ferrer Dalmau
7   * < vicente@jdalmau.es >
8   *
9   * Implementation of the Hill Climbing algorithm (simple version).
10  */
11
12 package travelingsalesman;
13
14 import java.util.ArrayList;
15
16 /**
17  *
18  * @author Vicente J. Ferrer Dalmau
19  */
20 public class HillClimbing {
21
22     RoutesMatrix distances;
23     int sourceCity;
24
25     String result = new String();
26
27     ArrayList followedRoute;
28     int nodes = 0;
29     int routeCost = 0;
30
31     int HEURISTICCONSTANT = 15;
32
33     /**
34      * Gets the heuristic value for a given depth
```

```
35          * The level 0 has the maximum value.
36          */
37         private int getHeuristicValue (int level) {
38 
39             return HEURISTICCONSTANT * (distances.getCitiesCount() - level);
40         }
41 
42         /** Creates a new instance of HillClimbing */
43         public HillClimbing(RoutesMatrix matrix, int sourceCity) {
44 
45             distances = matrix;
46             this.sourceCity = sourceCity;
47         }
48 
49         /**
50          * executes the algorithm
51          */
52         public String execute () {
53 
54             followedRoute = new ArrayList();
55             followedRoute.add(sourceCity);
56             nodes++;
57 
58             long startTime = System.currentTimeMillis();
59             search(sourceCity);
60             long endTime = System.currentTimeMillis();
61 
62             result = "-----------------------------------------\n";
63             result +=  "MÉTODO DE ESCALADA:\n";
64             result += "-----------------------------------------\n";
65             result += "MEJOR SOLUCIÓN: \t"+followedRoute.toString() + "\
                   nCOSTE: \t\t"+routeCost+"\n";
66             result += "NODOS VISITADOS: \t"+nodes+"\n";
67             result += "TIEMPO TRANSCURRIDO: \t"+(endTime-startTime)+" ms\n";
68             result += "-----------------------------------------\n";
69 
70             return result;
71         }
72 
73         /**
74          * @param from node where we start the search.
75          */
76         public void search (int from) {
77 
78             int currentTown = from;
79 
80             while (nodes != distances.getCitiesCount()) {
81                 // choose the closest town
82                 int lowestDistance = Integer.MAX_VALUE;
83                 int chosen = -1;
84                 for (int i=0; i < distances.getCitiesCount(); i++) {
```

```
85                  if (!followedRoute.contains(i)) {
86                      int tempDistance = routeCost + getHeuristicValue(
                            nodes-1); // f = g + h
87                      if (tempDistance < lowestDistance) {
88                          lowestDistance = tempDistance;
89                          chosen = i;
90                      }
91                  }
92              }
93              routeCost += distances.getCost(currentTown, chosen);
94              followedRoute.add(chosen);
95              currentTown = chosen;
96              nodes++;
97          }
98          // add the last town
99          routeCost += distances.getCost(currentTown, sourceCity);
100         followedRoute.add(sourceCity);
101         nodes++;
102     }
103 }
```

9.5. Main

```
1  /*
2   * Main.java
3   *
4   * $LastChangedDate: 2008-04-01 01:27:52 +0200 (mar, 01 abr 2008) $
5   * $LastChangedRevision: 22 $
6   * Vicente J. Ferrer Dalmau
7   * < vicente@jdalmau.es >
8   *
9   */
10
11 package travelingsalesman;
12
13 import javax.swing.table.DefaultTableModel;
14 import java.awt.*;
15 import java.awt.event.*;
16 import javax.swing.*;
17 import javax.swing.table.*;
18
19 /**
20  *
21  * @author Vicente J. Ferrer Dalmau
22  */
23 public class Main extends javax.swing.JFrame {
24
25     static RoutesMatrix routes;
26     int sourceCity = 0;
27
28     /** Creates new form Main */
29     public Main() {
```

```
30          initComponents();
31
32          // init the routes matrix
33          routes = new RoutesMatrix(4);
34
35          // init the cities checkbox
36          for (int i = 2; i <= 100; i++)
37              jComboBox1.addItem(i);
38          for (int i = 2; i <= 10; i++)
39              jComboBox1.addItem(i*100);
40          for (int i = 0; i <= 3; i++)
41              jComboBox2.addItem(i);
42          jComboBox1.setSelectedIndex(2);
43          jComboBox2.setSelectedIndex(0);
44
45          // configure the JTable to display de routes matrix
46          routes.drawJTable(jTable1);
47      }
48
49      /** This method is called from within the constructor to
50       * initialize the form.
51       * WARNING: Do NOT modify this code. The content of this method is
52       * always regenerated by the Form Editor.
53       */
54      // <editor-fold defaultstate="collapsed" desc=" Generated Code ">//
            GEN-BEGIN:initComponents
55      private void initComponents() {
56          jPanel1 = new javax.swing.JPanel();
57          jScrollPane2 = new javax.swing.JScrollPane();
58          jTable1 = new javax.swing.JTable();
59          jButton6 = new javax.swing.JButton();
60          jButton7 = new javax.swing.JButton();
61          jComboBox1 = new javax.swing.JComboBox();
62          jLabel1 = new javax.swing.JLabel();
63          jLabel2 = new javax.swing.JLabel();
64          jComboBox2 = new javax.swing.JComboBox();
65          jPanel2 = new javax.swing.JPanel();
66          jButton1 = new javax.swing.JButton();
67          jButton2 = new javax.swing.JButton();
68          jButton3 = new javax.swing.JButton();
69          jPanel3 = new javax.swing.JPanel();
70          jButton4 = new javax.swing.JButton();
71          jButton5 = new javax.swing.JButton();
72          jButton8 = new javax.swing.JButton();
73          jPanel4 = new javax.swing.JPanel();
74          jScrollPane1 = new javax.swing.JScrollPane();
75          jTextArea1 = new javax.swing.JTextArea();
76          jPanel5 = new javax.swing.JPanel();
77          jButton9 = new javax.swing.JButton();
78
79          getContentPane().setLayout(new org.netbeans.lib.awtextra.
                AbsoluteLayout());
```

```
80
81          setDefaultCloseOperation(javax.swing.WindowConstants.
                EXIT_ON_CLOSE);
82          setTitle("IA - El Viajante de Comercio (Vicente J. Ferrer Dalmau)
                ");
83          setBackground(new java.awt.Color(255, 255, 255));
84          setCursor(new java.awt.Cursor(java.awt.Cursor.HAND_CURSOR));
85          setResizable(false);
86          jPanel1.setLayout(new org.netbeans.lib.awtextra.AbsoluteLayout())
                ;
87
88          jPanel1.setBorder(javax.swing.BorderFactory.createTitledBorder("
                Configurar Ciudades"));
89          jScrollPane2.setHorizontalScrollBarPolicy(javax.swing.
                ScrollPaneConstants.HORIZONTAL_SCROLLBAR_ALWAYS);
90          jScrollPane2.setAutoscrolls(true);
91          jTable1.setModel(new javax.swing.table.DefaultTableModel(
92              new Object [][] {
93                  {null, null, null, null},
94                  {null, null, null, null},
95                  {null, null, null, null},
96                  {null, null, null, null}
97              },
98              new String [] {
99                  "Title 1", "Title 2", "Title 3", "Title 4"
100             }
101         ));
102         jTable1.setAutoResizeMode(javax.swing.JTable.AUTO_RESIZE_OFF);
103         jScrollPane2.setViewportView(jTable1);
104
105         jPanel1.add(jScrollPane2, new org.netbeans.lib.awtextra.
                AbsoluteConstraints(10, 20, 870, 200));
106
107         jButton6.setText("Regenerar Matriz");
108         jButton6.addMouseListener(new java.awt.event.MouseAdapter() {
109             public void mouseClicked(java.awt.event.MouseEvent evt) {
110                 jButton6MouseClicked(evt);
111             }
112         });
113
114         jPanel1.add(jButton6, new org.netbeans.lib.awtextra.
                AbsoluteConstraints(590, 230, -1, -1));
115
116         jButton7.setText("Modificar distancias");
117         jButton7.addMouseListener(new java.awt.event.MouseAdapter() {
118             public void mouseClicked(java.awt.event.MouseEvent evt) {
119                 jButton7MouseClicked(evt);
120             }
121         });
122
123         jPanel1.add(jButton7, new org.netbeans.lib.awtextra.
                AbsoluteConstraints(730, 230, 150, -1));
```

```
124
125         jComboBox1.addItemListener(new java.awt.event.ItemListener() {
126             public void itemStateChanged(java.awt.event.ItemEvent evt) {
127                 jComboBox1ItemStateChanged(evt);
128             }
129         });
130
131         jPanel1.add(jComboBox1, new org.netbeans.lib.awtextra.
                AbsoluteConstraints(90, 230, -1, -1));
132
133         jLabel1.setText("N\u00ba Ciudades:");
134         jPanel1.add(jLabel1, new org.netbeans.lib.awtextra.
                AbsoluteConstraints(10, 230, -1, 20));
135
136         jLabel2.setText("Ciudad inicial:");
137         jPanel1.add(jLabel2, new org.netbeans.lib.awtextra.
                AbsoluteConstraints(160, 230, 80, 20));
138
139         jComboBox2.addItemListener(new java.awt.event.ItemListener() {
140             public void itemStateChanged(java.awt.event.ItemEvent evt) {
141                 jComboBox2ItemStateChanged(evt);
142             }
143         });
144
145         jPanel1.add(jComboBox2, new org.netbeans.lib.awtextra.
                AbsoluteConstraints(250, 230, -1, -1));
146
147         getContentPane().add(jPanel1, new org.netbeans.lib.awtextra.
                AbsoluteConstraints(0, 0, 890, 270));
148
149         jPanel2.setLayout(new org.netbeans.lib.awtextra.AbsoluteLayout())
                ;
150
151         jPanel2.setBorder(javax.swing.BorderFactory.createTitledBorder("B
                \u00fasqueda No Informada"));
152         jButton1.setText("Primero en profundidad");
153         jButton1.addMouseListener(new java.awt.event.MouseAdapter() {
154             public void mouseClicked(java.awt.event.MouseEvent evt) {
155                 jButton1MouseClicked(evt);
156             }
157         });
158
159         jPanel2.add(jButton1, new org.netbeans.lib.awtextra.
                AbsoluteConstraints(10, 20, 170, -1));
160
161         jButton2.setText("Ramificaci\u00f3n y poda");
162         jButton2.addMouseListener(new java.awt.event.MouseAdapter() {
163             public void mouseClicked(java.awt.event.MouseEvent evt) {
164                 jButton2MouseClicked(evt);
165             }
166         });
167
```

```
168         jPanel2.add(jButton2, new org.netbeans.lib.awtextra.
                AbsoluteConstraints(190, 20, 150, -1));
169
170         jButton3.setText("Costo uniforme");
171         jButton3.addMouseListener(new java.awt.event.MouseAdapter() {
172             public void mouseClicked(java.awt.event.MouseEvent evt) {
173                 jButton3MouseClicked(evt);
174             }
175         });
176
177         jPanel2.add(jButton3, new org.netbeans.lib.awtextra.
                AbsoluteConstraints(350, 20, 140, -1));
178
179         getContentPane().add(jPanel2, new org.netbeans.lib.awtextra.
                AbsoluteConstraints(0, 480, 500, 60));
180
181         jPanel3.setLayout(new org.netbeans.lib.awtextra.AbsoluteLayout())
                ;
182
183         jPanel3.setBorder(javax.swing.BorderFactory.createTitledBorder("B
                \u00fasqueda Informada"));
184         jButton4.setText("Vecino m\u00e1s pr\u00f3ximo");
185         jButton4.addMouseListener(new java.awt.event.MouseAdapter() {
186             public void mouseClicked(java.awt.event.MouseEvent evt) {
187                 jButton4MouseClicked(evt);
188             }
189         });
190
191         jPanel3.add(jButton4, new org.netbeans.lib.awtextra.
                AbsoluteConstraints(100, 20, 170, -1));
192
193         jButton5.setText("A*");
194         jButton5.addMouseListener(new java.awt.event.MouseAdapter() {
195             public void mouseClicked(java.awt.event.MouseEvent evt) {
196                 jButton5MouseClicked(evt);
197             }
198         });
199
200         jPanel3.add(jButton5, new org.netbeans.lib.awtextra.
                AbsoluteConstraints(10, 20, 80, -1));
201
202         jButton8.setText("Escalada");
203         jButton8.addMouseListener(new java.awt.event.MouseAdapter() {
204             public void mouseClicked(java.awt.event.MouseEvent evt) {
205                 jButton8MouseClicked(evt);
206             }
207         });
208
209         jPanel3.add(jButton8, new org.netbeans.lib.awtextra.
                AbsoluteConstraints(280, 20, 100, -1));
210
211         getContentPane().add(jPanel3, new org.netbeans.lib.awtextra.
```

```
            AbsoluteConstraints(500, 480, 390, 60));
212
213         jPanel4.setLayout(new org.netbeans.lib.awtextra.AbsoluteLayout())
                ;
214
215         jPanel4.setBorder(javax.swing.BorderFactory.createTitledBorder("
                Resultados"));
216         jTextArea1.setColumns(20);
217         jTextArea1.setEditable(false);
218         jTextArea1.setRows(5);
219         jScrollPane1.setViewportView(jTextArea1);
220
221         jPanel4.add(jScrollPane1, new org.netbeans.lib.awtextra.
                AbsoluteConstraints(10, 20, 870, 180));
222
223         getContentPane().add(jPanel4, new org.netbeans.lib.awtextra.
                AbsoluteConstraints(0, 270, 890, 210));
224
225         jPanel5.setLayout(new org.netbeans.lib.awtextra.AbsoluteLayout())
                ;
226
227         jPanel5.setBorder(javax.swing.BorderFactory.createTitledBorder("B
                \u00fasqueda No Informada & Informada"));
228         jButton9.setText("Todo");
229         jButton9.addMouseListener(new java.awt.event.MouseAdapter() {
230             public void mouseClicked(java.awt.event.MouseEvent evt) {
231                 jButton9MouseClicked(evt);
232             }
233         });
234
235         jPanel5.add(jButton9, new org.netbeans.lib.awtextra.
                AbsoluteConstraints(400, 20, 100, -1));
236
237         getContentPane().add(jPanel5, new org.netbeans.lib.awtextra.
                AbsoluteConstraints(0, 540, 890, 60));
238
239         pack();
240     }// </editor-fold>//GEN-END:initComponents
241
242     private void jButton9MouseClicked(java.awt.event.MouseEvent evt) {//
            GEN-FIRST:event_jButton9MouseClicked
243
244         try {
245             DepthFirstSearch s1 = new DepthFirstSearch(routes, sourceCity
                    );
246             jTextArea1.setText(s1.execute());
247             BranchAndBound s2 = new BranchAndBound(routes, sourceCity);
248             jTextArea1.setText(jTextArea1.getText() + s2.execute());
249             UniformCost s3 = new UniformCost(routes, sourceCity);
250             jTextArea1.setText(jTextArea1.getText() + s3.execute());
251             AStar s6 = new AStar(routes, sourceCity);
252             jTextArea1.setText(jTextArea1.getText() + s6.execute());
```

```
253             NearestNeighbour s4 = new NearestNeighbour(routes, sourceCity
                    );
254             jTextArea1.setText(jTextArea1.getText() + s4.execute());
255             HillClimbing s5 = new HillClimbing(routes, sourceCity);
256             jTextArea1.setText(jTextArea1.getText() + s5.execute());
257         } catch (java.lang.OutOfMemoryError e) {
258                 String msg = "La memoria no es suficiente para ejecutar
                        alguno de los métodos con las "+
259                         ((Integer)jComboBox1.getSelectedItem()).intValue
                            ()+" ciudades generadas.";
260                 JOptionPane.showMessageDialog(new JFrame(), msg, "Error",
                        JOptionPane.ERROR_MESSAGE);
261         }
262
263     }//GEN-LAST:event_jButton9MouseClicked
264
265     private void jComboBox1ItemStateChanged(java.awt.event.ItemEvent evt)
            {//GEN-FIRST:event_jComboBox1ItemStateChanged
266
267         if (evt.getStateChange() == evt.SELECTED) {
268             // new number of cities
269             int cities = ((Integer)jComboBox1.getSelectedItem()).intValue
                    ();
270             routes = new RoutesMatrix(cities);
271             routes.drawJTable(jTable1);
272             // alter the 2nd comboBox (source city)
273             jComboBox2.removeAllItems();
274             for (int i=0; i<cities; i++)
275                 jComboBox2.addItem(i);
276             jComboBox2.setSelectedIndex(0);
277         }
278     }//GEN-LAST:event_jComboBox1ItemStateChanged
279
280     private void jComboBox2ItemStateChanged(java.awt.event.ItemEvent evt)
            {//GEN-FIRST:event_jComboBox2ItemStateChanged
281
282         if (evt.getStateChange() == evt.SELECTED) {
283             // select the new source city
284             sourceCity = ((Integer)jComboBox2.getSelectedItem()).intValue
                    ();
285         }
286     }//GEN-LAST:event_jComboBox2ItemStateChanged
287
288     private void jButton8MouseClicked(java.awt.event.MouseEvent evt) {//
            GEN-FIRST:event_jButton8MouseClicked
289
290         try {
291         HillClimbing s = new HillClimbing(routes, sourceCity);
292         jTextArea1.setText(s.execute());
293         } catch (java.lang.OutOfMemoryError e) {
294                 String msg = "La memoria no es suficiente para ejecutar
                        el Método de Escalada con las "+
```

```
295                         ((Integer)jComboBox1.getSelectedItem()).intValue
                                ()+" ciudades generadas.";
296                 JOptionPane.showMessageDialog(new JFrame(), msg, "Error",
                        JOptionPane.ERROR_MESSAGE);
297         }
298     }//GEN-LAST:event_jButton8MouseClicked
299
300     private void jButton4MouseClicked(java.awt.event.MouseEvent evt) {//
            GEN-FIRST:event_jButton4MouseClicked
301
302         try {
303         NearestNeighbour s = new NearestNeighbour(routes, sourceCity);
304         jTextArea1.setText(s.execute());
305         } catch (java.lang.OutOfMemoryError e) {
306                 String msg = "La memoria no es suficiente para ejecutar
                        la búsqueda del Vecino más Proximo con las "+
307                         ((Integer)jComboBox1.getSelectedItem()).intValue
                                ()+" ciudades generadas.";
308                 JOptionPane.showMessageDialog(new JFrame(), msg, "Error",
                        JOptionPane.ERROR_MESSAGE);
309         }
310     }//GEN-LAST:event_jButton4MouseClicked
311
312     private void jButton5MouseClicked(java.awt.event.MouseEvent evt) {//
            GEN-FIRST:event_jButton5MouseClicked
313
314         try {
315         AStar s = new AStar(routes, sourceCity);
316         jTextArea1.setText(s.execute());
317         } catch (java.lang.OutOfMemoryError e) {
318                 String msg = "La memoria no es suficiente para ejecutar
                        la búsqueda A Estrella con las "+
319                         ((Integer)jComboBox1.getSelectedItem()).intValue
                                ()+" ciudades generadas.";
320                 JOptionPane.showMessageDialog(new JFrame(), msg, "Error",
                        JOptionPane.ERROR_MESSAGE);
321         }
322     }//GEN-LAST:event_jButton5MouseClicked
323
324     private void jButton3MouseClicked(java.awt.event.MouseEvent evt) {//
            GEN-FIRST:event_jButton3MouseClicked
325
326         try {
327         UniformCost s = new UniformCost(routes, sourceCity);
328         jTextArea1.setText(s.execute());
329         } catch (java.lang.OutOfMemoryError e) {
330                 String msg = "La memoria no es suficiente para ejecutar
                        la búsqueda de Costo Uniforme con las "+
331                         ((Integer)jComboBox1.getSelectedItem()).intValue
                                ()+" ciudades generadas.";
332                 JOptionPane.showMessageDialog(new JFrame(), msg, "Error",
                        JOptionPane.ERROR_MESSAGE);
```

```
333             }
334         }//GEN-LAST:event_jButton3MouseClicked
335 
336         private void jButton2MouseClicked(java.awt.event.MouseEvent evt) {//
                GEN-FIRST:event_jButton2MouseClicked
337 
338             try {
339             BranchAndBound s = new BranchAndBound(routes, sourceCity);
340             jTextArea1.setText(s.execute());
341             } catch (java.lang.OutOfMemoryError e) {
342                     String msg = "La memoria no es suficiente para ejecutar
                            la búsqueda con Ramificación y Poda con las "+
343                             ((Integer)jComboBox1.getSelectedItem()).intValue
                                    ()+" ciudades generadas.";
344                     JOptionPane.showMessageDialog(new JFrame(), msg, "Error",
                             JOptionPane.ERROR_MESSAGE);
345             }
346         }//GEN-LAST:event_jButton2MouseClicked
347 
348         private void jButton7MouseClicked(java.awt.event.MouseEvent evt) {//
                GEN-FIRST:event_jButton7MouseClicked
349 
350             // configure the JTable to display de routes matrix
351             for (int i=0; i<routes.getCitiesCount(); i++) {
352                 for (int j=0; j<routes.getCitiesCount(); j++) {
353                     int value = Integer.parseInt(jTable1.getValueAt(i, j+1).
                            toString());
354                     if (value >= 0 && value <=routes.getMaxDistance()) {
355                         routes.setCost(i, j, value);
356                     }
357                 }
358             }
359 
360             // regenerate the jTable
361             routes.drawJTable(jTable1);
362         }//GEN-LAST:event_jButton7MouseClicked
363 
364         private void jButton6MouseClicked(java.awt.event.MouseEvent evt) {//
                GEN-FIRST:event_jButton6MouseClicked
365 
366             int cities = ((Integer)jComboBox1.getItemAt(jComboBox1.
                    getSelectedIndex())).intValue();
367 
368             routes = new RoutesMatrix(cities);
369             routes.drawJTable(jTable1);
370         }//GEN-LAST:event_jButton6MouseClicked
371 
372         private void jButton1MouseClicked(java.awt.event.MouseEvent evt) {//
                GEN-FIRST:event_jButton1MouseClicked
373 
374             try {
375             DepthFirstSearch s = new DepthFirstSearch(routes, sourceCity);
```

```
376             jTextArea1.setText(s.execute());
377             } catch (java.lang.OutOfMemoryError e) {
378                     String msg = "La memoria no es suficiente para ejecutar
                            la búsqueda Primero en Profundidad con las "+
379                                 ((Integer)jComboBox1.getSelectedItem()).intValue
                                        ()+" ciudades generadas.";
380                     JOptionPane.showMessageDialog(new JFrame(), msg, "Error",
                             JOptionPane.ERROR_MESSAGE);
381             }
382         }//GEN-LAST:event_jButton1MouseClicked
383 
384         public void printResults(String results) {
385 
386             jTextArea1.setText(results);
387         }
388 
389         /**
390          * @param args the command line arguments
391          */
392         public static void main(String args[]) {
393             java.awt.EventQueue.invokeLater(new Runnable() {
394                 public void run() {
395                     new Main().setVisible(true);
396                 }
397             });
398         }
399 
400         // Variables declaration - do not modify//GEN-BEGIN:variables
401         private javax.swing.JButton jButton1;
402         private javax.swing.JButton jButton2;
403         private javax.swing.JButton jButton3;
404         private javax.swing.JButton jButton4;
405         private javax.swing.JButton jButton5;
406         private javax.swing.JButton jButton6;
407         private javax.swing.JButton jButton7;
408         private javax.swing.JButton jButton8;
409         private javax.swing.JButton jButton9;
410         private javax.swing.JComboBox jComboBox1;
411         private javax.swing.JComboBox jComboBox2;
412         private javax.swing.JLabel jLabel1;
413         private javax.swing.JLabel jLabel2;
414         private javax.swing.JPanel jPanel1;
415         private javax.swing.JPanel jPanel2;
416         private javax.swing.JPanel jPanel3;
417         private javax.swing.JPanel jPanel4;
418         private javax.swing.JPanel jPanel5;
419         private javax.swing.JScrollPane jScrollPane1;
420         private javax.swing.JScrollPane jScrollPane2;
421         private javax.swing.JTable jTable1;
422         private javax.swing.JTextArea jTextArea1;
423         // End of variables declaration//GEN-END:variables
424 
```

```
425 }
```

9.6. NearestNeighbour

```
1  /*
2   * NearestNeighbour.java
3   *
4   * $LastChangedDate: 2008-03-30 20:41:41 +0200 (dom, 30 mar 2008) $
5   * $LastChangedRevision: 14 $
6   * Vicente J. Ferrer Dalmau
7   * < vicente@jdalmau.es >
8   *
9   * Implementation of the Nearest Neighbour algorithm.
10  */
11
12 package travelingsalesman;
13
14 import java.util.ArrayList;
15
16 /**
17  *
18  * @author Vicente J. Ferrer Dalmau
19  */
20 public class NearestNeighbour {
21
22     RoutesMatrix distances;
23     int sourceCity;
24
25     String result = new String();
26
27     ArrayList followedRoute;
28     int nodes = 0;
29     int routeCost = 0;
30
31     /** Creates a new instance of NearestNeighbour */
32     public NearestNeighbour(RoutesMatrix matrix, int sourceCity) {
33
34         distances = matrix;
35         this.sourceCity = sourceCity;
36     }
37
38     /**
39      * executes the algorithm
40      */
41     public String execute () {
42
43         followedRoute = new ArrayList();
44         followedRoute.add(sourceCity);
45         nodes++;
46
47         long startTime = System.currentTimeMillis();
48         search(sourceCity);
```

```
49          long endTime = System.currentTimeMillis();
50
51          result = "-----------------------------------------\n";
52          result +=  "VECINO MÁS PRÓXIMO:\n";
53          result += "-----------------------------------------\n";
54          result += "MEJOR SOLUCIÓN: \t"+followedRoute.toString() + "\
                nCOSTE: \t\t"+routeCost+"\n";
55          result += "NODOS VISITADOS: \t"+nodes+"\n";
56          result += "TIEMPO TRANSCURRIDO: \t"+(endTime-startTime)+" ms\n";
57          result += "-----------------------------------------\n";
58
59          return result;
60      }
61
62      /**
63       * @param from node where we start the search.
64       */
65      public void search (int from) {
66
67          int currentTown = from;
68
69          while (nodes != distances.getCitiesCount()) {
70              // choose the closest town
71              int lowestDistance = Integer.MAX_VALUE;
72              int chosen = -1;
73              for (int i=0; i < distances.getCitiesCount(); i++) {
74                  if (!followedRoute.contains(i)) {
75                      int tempDistance = distances.getCost(currentTown, i);
76                      if (tempDistance < lowestDistance) {
77                          lowestDistance = tempDistance;
78                          chosen = i;
79                      }
80                  }
81              }
82              routeCost += distances.getCost(currentTown, chosen);
83              followedRoute.add(chosen);
84              currentTown = chosen;
85              nodes++;
86          }
87          // add the last town
88          routeCost += distances.getCost(currentTown, sourceCity);
89          followedRoute.add(sourceCity);
90          nodes++;
91      }
92  }
```

9.7. RoutesMatrix

```
1   /*
2    * RoutesMatrix.java
3    *
4    * $LastChangedDate: 2008-03-30 22:35:24 +0200 (dom, 30 mar 2008) $
```

```
5   * $LastChangedRevision: 15 $
6   * Vicente J. Ferrer Dalmau
7   * < vicente@jdalmau.es >
8   *
9   * This class defines all the distances between the cities in the problem
        .
10  * For two cities, x and y, the both distances x to y and y to x remain
        the same.
11  */
12
13  package travelingsalesman;
14
15  import java.awt.Color;
16  import java.awt.Component;
17  import java.util.Random;
18  import javax.swing.JTable;
19  import javax.swing.JTextField;
20  import javax.swing.table.DefaultTableCellRenderer;
21  import javax.swing.table.DefaultTableModel;
22
23
24  /**
25   *
26   * @author Vicente J. Ferrer Dalmau
27   */
28  public class RoutesMatrix {
29
30      private int[][] theMatrix;
31      private int cities;
32      // all the distance values will be in a Uniform(MAXDISTANCE)
33      private int MAXDISTANCE = 100;
34
35      /** Creates a new instance of RoutesMatrix
36       */
37      public RoutesMatrix(int cities) {
38
39          theMatrix = new int[cities][cities];
40          this.cities = cities;
41
42          // fill the matrix with random values
43          // a new random generator (seed based on the current time)
44          Random generator = new Random();
45
46          for (int i=0; i<cities; i++) {
47              for (int j=i; j<cities; j++) {
48                  if (i == j)
49                      theMatrix[i][j] = 0;
50                  else {
51                      theMatrix[i][j] = generator.nextInt(MAXDISTANCE);
52                      theMatrix[j][i] = theMatrix[i][j];
53                  }
54              }
```

```
55          }
56      }
57
58      /**
59       * returns the number of cities
60       */
61      public int getCitiesCount() {
62
63          return cities;
64      }
65
66      /**
67       * gets the cost of going from city "a" to city "b"
68       */
69      public int getCost(int a, int b) {
70
71          return theMatrix[a][b];
72      }
73
74      /**
75       * sets the cost of going from city "a" to city "b"
76       */
77      public void setCost(int a, int b, int cost) {
78
79          theMatrix[a][b] = cost;
80      }
81
82      /**
83       * gets the array of costs as an Object[][] array.
84       */
85      public Object[][] getCosts () {
86
87          Object [][] array = new Object[cities][cities+1];
88          for (int i=0; i<cities; i++){
89              for (int j=0; j<cities; j++){
90                  if (j == 0) {
91                      array[i][0] = i;
92                      array[i][j+1] = theMatrix[i][j];
93                  }
94                  else
95                      array[i][j+1] = theMatrix[i][j];
96              }
97          }
98          return array;
99      }
100
101     /**
102      * gets the cities in an Object[] array.
103      */
104     public Object[] getCities () {
105
106         Object[] array = new Object[cities+1];
```

```
107         for (int i=0; i<cities; i++) {
108             if (i == 0) {
109                 array[i] = "␣";
110                 array[i+1] = 0;
111             }
112             else
113                 array[i+1] = i;
114         }
115         return array;
116     }
117 
118     /**
119      * gets the maximum distance between two cities.
120      */
121     public int getMaxDistance() {
122 
123         return MAXDISTANCE;
124     }
125 
126     public String toString() {
127 
128         String str = new String();
129         for (int i=0; i<cities; i++) {
130             for (int j=0; j<cities; j++) {
131                 if (j == cities - 1)
132                     str += theMatrix[i][j] + "\n";
133                 else
134                     str += theMatrix[i][j] + ",␣";
135             }
136         }
137         return str;
138     }
139 
140     // Redefine the behaviour of the table
141     public class MyRender extends DefaultTableCellRenderer {
142        public Component getTableCellRendererComponent(JTable table,
143           Object value,
144           boolean isSelected,
145           boolean hasFocus,
146           int row,
147           int column)
148        {
149           super.getTableCellRendererComponent (table, value, isSelected,
                 hasFocus, row, column);
150           this.setOpaque(true);
151           this.setToolTipText("");
152           if (column == 0) {
153                 this.setBackground(Color.LIGHT_GRAY);
154                 this.setHorizontalAlignment(JTextField.CENTER);
155           }
156           else if (column -1 == row) {
157                 this.setBackground(Color.LIGHT_GRAY);
```

```
158             }
159             else {
160                     this.setBackground(Color.WHITE);
161                     this.setToolTipText("De la ciudad "+row+" a la "+(column
                            -1));
162             }
163             return this;
164         }
165     }
166
167     /**
168      * Shows the content of the matrix in a JTable object.
169      */
170     public void drawJTable (JTable j) {
171
172         DefaultTableModel dtm = new DefaultTableModel() {
173             public boolean isCellEditable(int row, int column) {
174                 if (column != 0 && (column - 1 != row))
175                     return true;
176                 else
177                     return false;
178             }
179         };
180         dtm.setDataVector(this.getCosts(), this.getCities());
181         // set the background of the first column
182         j.setModel(dtm);
183         j.setDefaultRenderer (Object.class, new MyRender());
184     }
185
186 }
```

9.8. Town

```
1  /*
2   * Town.java
3   *
4   * $LastChangedDate: 2008-03-28 21:37:21 +0100 (vie, 28 mar 2008) $
5   * $LastChangedRevision: 9 $
6   * Vicente J. Ferrer Dalmau
7   * < vicente@jdalmau.es >
8   *
9   * Contains all the important information about a Town.
10  */
11
12 package travelingsalesman;
13
14 /**
15  *
16  * @author Vicente J. Ferrer Dalmau
17  */
18 public class Town {
19
```

```
20      public int number;
21      public int f, g, h;
22      public int level;
23      public Town parent = null;
24
25      /** Creates a new instance of Town */
26      public Town(int number, int g, int h, int level) {
27
28          this.number = number;
29          this.g = g;
30          this.h = h;
31          this.f = this.g + this.h;
32          this.level = level;
33      }
34
35  }
```

9.9. UniformCost

```
1   /*
2    * UniformCost.java
3    *
4    * $LastChangedDate: 2008-03-30 20:41:41 +0200 (dom, 30 mar 2008) $
5    * $LastChangedRevision: 14 $
6    * Vicente J. Ferrer Dalmau
7    * < vicente@jdalmau.es >
8    *
9    * Implementation of the Uniform Cost algorithm.
10   */
11
12  package travelingsalesman;
13
14  import java.util.ArrayList;
15  import java.util.Comparator;
16  import java.util.PriorityQueue;
17
18  /**
19   *
20   * @author Vicente J. Ferrer Dalmau
21   */
22  public class UniformCost {
23
24      RoutesMatrix distances;
25      int sourceCity;
26      PriorityQueue<Town> toExpand = new PriorityQueue<Town>(200,
27          new Comparator<Town>() {
28            public int compare(Town a, Town b) {
29              return a.g - b.g;
30            }
31          }
32        );
33      String result = new String();
```

```
34
35      ArrayList optimumRoute, followedRoute;
36      int nodes = 0;
37      int routeCost = 0;
38      int optimumCost = Integer.MAX_VALUE;
39
40      /** Creates a new instance of UniformCost */
41      public UniformCost(RoutesMatrix matrix, int sourceCity) {
42
43          distances = matrix;
44          this.sourceCity = sourceCity;
45      }
46
47      /**
48       * executes the algorithm
49       */
50      public String execute() {
51
52          // have we found the solution?
53          boolean solution = false;
54
55          // start the timer
56          long startTime = System.currentTimeMillis();
57
58          // initial town
59          toExpand.add(new Town(sourceCity, 0, 0, 0));
60
61          while (!toExpand.isEmpty() && !solution) {
62              // gets the city with lower g value
63              Town currentTown = toExpand.poll();
64              nodes++;
65
66              // rebuild the followed route for the selected town
67              Town aux = currentTown;
68              followedRoute = new ArrayList();
69              followedRoute.add(aux.number);
70              while (aux.level != 0) {
71                  aux = aux.parent;
72                  followedRoute.add(0, aux.number);
73              }
74
75              if (currentTown.level == distances.getCitiesCount()) {
76                  solution = true;
77                  optimumRoute = followedRoute;
78                  optimumCost = currentTown.g;
79              } else {
80
81                  for (int i=0; i<distances.getCitiesCount(); i++) {
82                      // have we visited this city in the current followed
                            route?
83                      boolean visited = followedRoute.contains(i);
84                      boolean isSolution = (followedRoute.size() ==
```

```
                        distances.getCitiesCount())&&(i == sourceCity);
85
86                      if (!visited || isSolution) {
87                          Town childTown = new Town(i, currentTown.g +
                                distances.getCost(currentTown.number, i), 0,
                                currentTown.level + 1);
88                          childTown.parent = currentTown;
89                          toExpand.add(childTown);
90                      }
91                  }
92              }
93          }
94          long endTime = System.currentTimeMillis();
95
96          result = "-----------------------------------------\n";
97          result +=  "BÚSQUEDA DE COSTO UNIFORME:\n";
98          result += "-----------------------------------------\n";
99          result += "MEJOR SOLUCIÓN: \t"+optimumRoute.toString() + "\nCOSTE
                :\t\t"+optimumCost+"\n";
100         result += "NODOS VISITADOS: \t"+nodes+"\n";
101         result += "TIEMPO TRANSCURRIDO: \t"+(endTime-startTime)+" ms\n";
102         result += "-----------------------------------------\n";
103
104         return result;
105     }
106
107 }
```