Facultad de Ciencias Exactas

Tecnicatura Universitaria en Desarrollo de Aplicaciones Informáticas

Trabajo Práctico 4

"Algoritmo Greedy"

Meliendrez Agustín

Tandil, Mayo 2018

1. Introducción

A partir de la cátedra “Programación 3”, perteneciente la carrera “Tecnicatura Universitaria en Desarrollo de Aplicaciones Informáticas” (TUDAI), de la Facultad de Exactas, perteneciente a la Universidad Nacional del Centro de la Provincia de Buenos Aires, se planteó la problemática de implementar un “Algoritmo Greedy” el cual permita que, dado un cierto conjunto ciudades de una provincia, y desde una ciudad como punto de origen, se obtenga el recorrido total mínimo a una ciudad-puerto, sabiendo que, mientras más distancia deba recorrer el vehículo de transporte, mayor será el costo asociado al flete.

Para ello se implementara un algoritmo “Dijsktra”, en forma de pseudo-código, con el cual se hará un seguimiento hipotético, explicando su funcionamiento a partir de las iteraciones y los cambios que se van produciendo, demostrando con los datos que se vayan obteniendo.

2. Desarrollo

Para la solución del problema a resolver, se selecciono el algoritmo Dijsktra, ya que es un algoritmo que busca resolver el problema de “los caminos más cortos”, aportando la mejor solución, o la solución más optima, a cada problema.

Dicho algoritmo, permite hallar los caminos más cortos desde un vértice, considerado como el punto de origen, a los restantes nodos de un determinado grafo.

Para la implementación del pseudo-código, se implementaran los siguientes métodos:

- **Solucion():** Determina si los candidatos seleccionados han alcanzado una solución.
- **Factible():** Determina si es posible llegar a una solución con un candidato seleccionado.
- **Seleccionar():** Determina el mejor candidato del conjunto a seleccionar en un momento dado.
- **getRutaPuertoCercano():** Consiste en la llamada al método “**getRutasDijsktra()**”, para comprobar las rutas a los puertos, devolviendo el camino mas cercano.
- **getRutasDijsktra():** Consiste en la implementación del algoritmo Dijsktra.
- **getDistancia ():** Obtiene la distancia entre un punto origen, y un punto destino (el peso de la arista).

Por su parte, también se deben considerar la utilización de las siguientes variables globales:

- **ciudadesVisitadas[ ]:** Consiste en un arreglo de las ciudades por las cuales ya se conocen las distancias y los caminos.
- **ciudadesPuerto[ ]:** Consiste en un arreglo de todas las ciudades-puerto de un grafo determinado.
- **distancias[ ]:** Consiste en un arreglo, en el cual, en cada posición, posee una ciudad de padre (indicando que están unidas por una arista), y una distancia (consiste en el peso de la arista a la ciudad padre, sumado a la distancia total para llegar a una ciudad origen).

2.1 Implementación de métodos

Cabe destacar que, si bien se utilizo, mayormente, sintaxis de “Java”, esto consiste en un pseudo-código, por lo que no compila si se lo transcribe.

```
public getRutaPuertoCercano(Grafo g, Vertice origen) {
    // Calcula la ruta más corta de la ciudad de origen a las demás ciudades del grafo
    getRutasDijsktra(g, origen);
    // Busca que ciudades del grafo son Ciudades Puerto.
    for (ciudad en g) {
        if (ciudad.getPuerto()) {
            ciudadesPuerto.add(ciudad);
        }
    }

    distanciaPuerto = 0;
    puertoElegido = null;

    for (puerto en ciudadesPuerto) {    //Obtiene que puerto es el mas cercano desde el punto de origen.
        if (factible(puerto)) {         //Se fija si el puerto seleccionado posee una ruta a la ciudad origen
            if(distanciaPuerto > getDistancia (origen, puerto) || distanciaPuerto == 0) {
                distanciaPuerto = getDistancia (origen, puerto)
                puertoElegido = puerto;
            }
        }
    }
    tmp = puertoElegido;
    ruta = null;

    Imprimir ("El puerto elegido es " + puertoElegido,getNombre() + " ubicado a " + distanciaPuerto);

    //Va insertando los "padres" desde la ciudad destino hasta que llegue al origen
    while (!tmp.equals(origen)) {
        ruta.addFirst(tmp); //Inserta la ciudad en la que esta en el principio, para que luego devuelva la ruta ordenada
        tmp = distancias[tmp].getCiudadOrigen();
    }

    if (solucion(ruta)) //Comprueba que haya encontrado una ruta
        return ruta;
    else
        return "No hay solucion";
}
```

Imagen 1: Método “getRutaPuertoCercano”. Fuente: “Elaboración propia”

```
// encuentra la ruta más corta desde una ciudad inicial a las demás
private void getRutasDijsktra(Grafo g, Vertice origen) {
    ciudadesNoVisitadas; // cola de prioridad para recorrrer las ciudades
    Vertice v = origen;  // ciudad inicial
    for (ciudad en g) {
        distancias[ciudad] = (x, ∞);
    }

    ciudadesNoVisitadas.add(v);                          // Agregar la ciudad inicial a la lista de no visitados
    distancias(v).distancia = 0;

    while(!ciudadesNoVisitadas.isEmpty()) {              // mientras que la lista no esta vacia
        tmp = seleccionar(ciudadesNoVisitadas);              // saca el primer elemento // Aca deberia seleccionar el minimo!!!!
        ciudadesVisitadas.add(tmp);                      // lo manda a la lista de terminados

        for(ciudad en tmp.getAdyacentes()) {             // revisa los nodos adyacentes la ciudad tmp
            if (ciudadesVisitadas.contiene(ciudad))      // si ya fue agregado a la lista de visitados continua
                continue;
            else if (ciudadesNoVisitadas.contiene(ciudad)) // si ya fue agregado a la lista de no visitados continua
                continue;
            else
                ciudadesNoVisitadas.add(ciudad);         //si no, lo agrega

            // si ya está en no visitados, actualiza las distancias
            distanciaTemporal = distancia[tmp].distancia + getDistancia (tmp, ciudad);
            //Comprueba si la distancia en el arreglo es mayor a la distancia actual
            if (distanciaTemporal < distancia [ciudad]) {
                distancia [ciudad].distancia = distanciaTemporal;
                distancia [ciudad].origen = tmp;
            }
        }
    }
```

Imagen 2: Método “getRutasDijsktra”. Fuente: “Elaboración propia”

```
public seleccionar (Arreglo a) {
    distanciaTemporal = 0;
    verticeRetorno;

    for(vertice en a) {
        distanciaCiudad = distancias[vertice].distancia;

        if (distanciaTemporal == 0){
            distanciaTemporal = distanciaCiudad;
            verticeRetorno = vertice
        }
        else{
            if (distanciaTemporal > distanciaCiudad) {
                distanciaTemporal = distanciaCiudad
                verticeRetorno = vertice
            }
        }
    }
    retun verticeRetorno
}
```

Imagen 3: Método “seleccionar”. Fuente: “Elaboración propia”

```
public factible(puerto) {
    return distancias[puerto].distancia != ∞;
}
```

Imagen 4: Método “factible”. Fuente: “Elaboración propia”

```
public solucion (ruta) {
    return !ruta.isEmpty();
}
```

Imagen 5: Método “solucion”. Fuente: “Elaboración propia”

2.2 Aplicación práctica

En base al pseudo-código presentado en el apartado anterior, se realizara un seguimiento para comprobar como funcionaria en una aplicación práctica concreta.

Para ello se utilizara el siguiente grafo (de ciudades de España), otorgado por la cátedra “Programación 3”, al cual se le aplicaron modificaciones, para disminuir la cantidad de ciudades que presentaba.

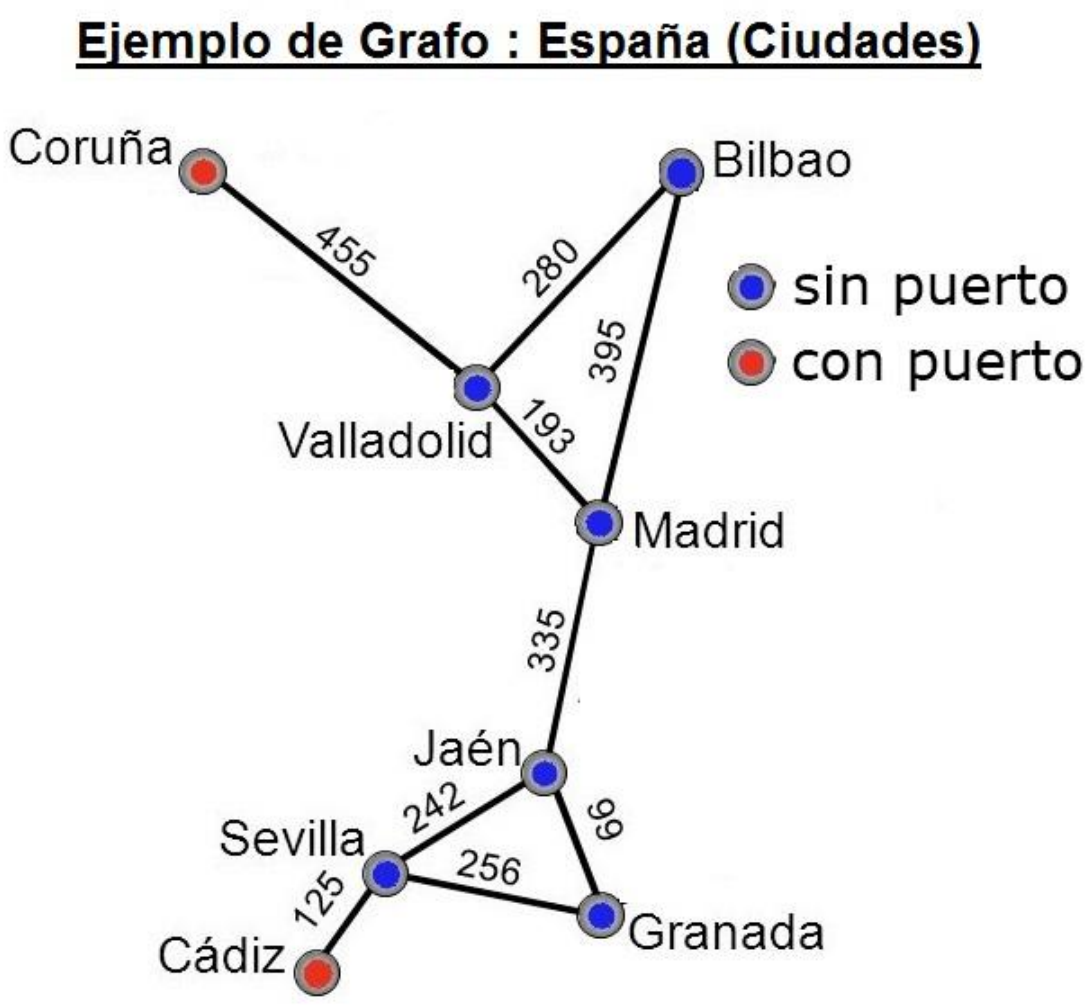

Imagen 6: “Ciudades de España”. Fuente: “*Cátedra Programación 3 -TUDAI-FCE-UNCPBA*”.

Para su ejemplificación, se tomara como punto de origen la ciudad de “Madrid”, de la cual se quiere obtener el puerto más cercano.

De esta manera se invocara al método “getRutaPuertoCercano (ciudadesEspaña, Madrid)”.

Dentro de dicho método, se hace una llamada a la método “getRutasDijsktra(ciudadesEspaña, Madrid)”, que consiste en el algoritmo disjktra para obtener las rutas más optimas.

Dentro de este, se comienza a completar la información dentro de la variable “distancias”. En un primer momento se completa todo como vacio, y la distancia de origen se completa en 0.

	Madrid	**Bilbao**	**Valladolid**	**Coruña**	**Jaén**	**Sevilla**	**Granada**	**Cádiz**
I1	Madrid, 0	X, ∞	X, ∞	X, ∞	X, ∞	X, ∞	X, ∞	X, ∞

Ciudades visitadas = [ ];
Ciudades no visitadas = [Madrid];

Posteriormente, se tomara, desde el origen, las ciudades adyacentes, y se completara la información en las variables “distancias” y “ciudadesNoVisitadas”, eliminando de esta ultima la ciudad con la que se comenzó(Madrid), incluyéndola en la variable “ciudadesVisitadas”.

	Madrid	**Bilbao**	**Valladolid**	**Coruña**	**Jaén**	**Sevilla**	**Granada**	**Cádiz**
I2	Madrid, 0	Madrid, 395	Madrid, 193	X, ∞	Madrid, 355	X, ∞	X, ∞	X, ∞

Ciudades visitadas = [Madrid];
Ciudades no visitadas = [Bilbao – Valladolid - Jaén];

Luego de que completo la información, se busca entre las ciudades no visitadas, seleccionándose aquella que esté más cerca. En este caso las opciones disponibles son: Bilbao (395), Valladolid (193) y Jaén (355), por lo que la próxima iteración comenzara como origen en Valladolid.

Desde Valladolid, se obtienen los adyacentes (Coruña [455] y Bilbao [280]). Para cada uno, se comprueba si ya existe en “ciudades visitadas”. Si no existe, y existe dentro de las “ciudades no visitadas”, comprobara las distancias para comprobar si debe modificarlas.

En el primer caso (Coruña), no existe en los “no visitados”, por lo que la completara con su distancia sumado a la distancia anterior (455 + 193 = 648). En el segundo caso (Bilbao), este ya existe en los “no visitados”, por lo que comprobara la diferencia de las distancias.

La distancia existente es de 395. Por su parte, la nueva distancia se conforma por la distancia en la ciudad que se está (Valladolid → 193), sumado a la

distancia que lo separa de Bilbao (280), lo que da como resultado 473. Como dicha distancia es mayor, no existe una modificación de sus datos.

Finalizando con esta iteración, se elimina “Valladolid” de “no visitados”, y se agrega a “ciudades visitadas”. Posteriormente, se agrega “Coruña” a ciudades “no visitadas”

	Madrid	**Bilbao**	**Valladolid**	**Coruña**	**Jaén**	**Sevilla**	**Granada**	**Cádiz**
I3	Madrid, 0	Madrid, 395	Madrid, 193	Valladolid, 648	Madrid, 355	X, ∞	X, ∞	X, ∞

Ciudades visitadas = [Madrid - Valladolid];
Ciudades no visitadas = [Bilbao – Jaén - Coruña];

Para la próxima iteración se toma como punto de inicio a “Jaén”, que está a una distancia de 355.

Al igual que en las anteriores iteraciones, se obtienen los adyacentes y se comprueba las distancias en caso de ser necesario.

En este caso, los adyacentes son Sevilla (242) y Granada (99). Como ninguno de los dos existe, se agregan a las “distancias” (sumando previamente la distancia de ciudad padre → Jaén → 355) y a las “ciudades no visitadas” y se traspasa a Jaén” desde las ciudades “no visitadas” a las “visitadas”.

	Madrid	**Bilbao**	**Valladolid**	**Coruña**	**Jaén**	**Sevilla**	**Granada**	**Cádiz**
I4	Madrid, 0	Madrid, 395	Madrid, 193	Valladolid, 648	Madrid, 355	Jaén, 597	Jaén, 454	X, ∞

Ciudades visitadas = [Madrid – Valladolid - Jaén];
Ciudades no visitadas = [Bilbao – Coruña – Sevilla - Granada];

Continuando con las iteraciones, el próximo a seleccionar es “Bilbao”. Dentro de este, se comprueba que los adyacentes que posee ya se encuentran dentro de “ciudades visitadas”. Por lo que la modificación realizada consiste en pasar a “Bilbao” desde las ciudades “no visitadas” a las “visitadas”.

	Madrid	**Bilbao**	**Valladolid**	**Coruña**	**Jaén**	**Sevilla**	**Granada**	**Cádiz**
I5	Madrid, 0	Madrid, 395	Madrid, 193	Valladolid, 648	Madrid, 355	Jaén, 597	Jaén, 454	X, ∞

Ciudades visitadas = [Madrid – Valladolid – Jaén - Bilbao];
Ciudades no visitadas = [Coruña – Sevilla - Granada];

La próxima ciudad seleccionada es “Granada”. El único adyacente disponible para comprobar distancias es “Sevilla”. Se comprueba que la distancia de Sevilla es de 597, mientras que la distancia desde Granda a Sevilla es de 710 (256 + 454), por lo que no existen modificaciones en las distancias. Se traspasa a “Granada” desde ciudades “no visitadas” a ciudades “visitadas”.

	Madrid	Bilbao	Valladolid	Coruña	Jaén	Sevilla	Granada	Cádiz
I6	Madrid, 0	Madrid, 395	Madrid, 193	Valladolid, 648	Madrid, 355	Jaén, 597	Jaén, 454	X, ∞

Ciudades visitadas = [Madrid – Valladolid – Jaén – Bilbao - Granada];
Ciudades no visitadas = [Coruña – Sevilla];

La nueva iteración comienza con la elección de “Sevilla” como origen. Desde ella se observa que la ciudad adyacente que posee es “Cádiz”. La distancia a ella es de 597 (distancia a Sevilla) + 125 (distancia entre Sevilla y Cádiz), dando una distancia total de 722.

Luego de completar los datos, agrega a las “ciudades no visitadas” a “Cádiz”, y realiza el traspaso de “Sevilla” desde las ciudades “no visitadas” a las “visitadas”.

	Madrid	Bilbao	Valladolid	Coruña	Jaén	Sevilla	Granada	Cádiz
I7	Madrid, 0	Madrid, 395	Madrid, 193	Valladolid, 648	Madrid, 355	Jaén, 597	Jaén, 454	Sevilla, 722

Ciudades visitadas = [Madrid – Valladolid – Jaén – Bilbao – Granada - Sevilla];
Ciudades no visitadas = [Coruña - Cádiz];

Por último, en las próximas dos iteraciones, al no poseer adyacentes en ninguno de los dos casos, no existen modificaciones en la información obtenida hasta el momento, y se van traspasando a las “ciudades visitadas” (primero “Coruña” y luego “Cádiz”).

	Madrid	Bilbao	Valladolid	Coruña	Jaén	Sevilla	Granada	Cádiz
I1	Madrid, 0	X, ∞	X, ∞	X, ∞	X, ∞	X, ∞	X, ∞	X, ∞
I2	Madrid, 0	Madrid, 395	Madrid, 193	X, ∞	Madrid, 355	X, ∞	X, ∞	X, ∞
I3	Madrid, 0	Madrid, 395	Madrid, 193	Valladolid, 648	Madrid, 355	X, ∞	X, ∞	X, ∞
I4	Madrid, 0	Madrid, 395	Madrid, 193	Valladolid, 648	Madrid, 355	Jaén, 597	Jaén, 454	X, ∞
I5	Madrid, 0	Madrid, 395	Madrid, 193	Valladolid, 648	Madrid, 355	Jaén, 597	Jaén, 454	X, ∞
I6	Madrid, 0	Madrid, 395	Madrid, 193	Valladolid, 648	Madrid, 355	Jaén, 597	Jaén, 454	X, ∞
I7	Madrid, 0	Madrid, 395	Madrid, 193	Valladolid, 648	Madrid, 355	Jaén, 597	Jaén, 454	Sevilla, 722
I8	Madrid, 0	Madrid, 395	Madrid, 193	Valladolid, 648	Madrid, 355	Jaén, 597	Jaén, 454	Sevilla, 722
I9	Madrid, 0	Madrid, 395	Madrid, 193	Valladolid, 648	Madrid, 355	Jaén, 597	Jaén, 454	Sevilla, 722

Tabla de iteraciones. Fuente: “Elaboración propia”.

Una vez obtenidas las distancias a todas las ciudades, teniendo como punto de partida a la ciudad de “Madrid”, se obtienen cuales de ellas son “ciudades-puerto”.

Como se puede observar en la “imagen 6”, las ciudades puerto obtenidas son “Coruña” y “Cádiz”.

A partir de ello, el próximo paso consiste en comprobar las distancias y seleccionar aquella ciudad que se encuentra a una distancia menor. Las distancias correspondientes son 648 (Coruña) y 722 (Cádiz), por lo que la “ciudad-puerto” seleccionada será “Coruña”.

Posteriormente, se reconstruirá la ruta correspondiente, desde el punto destino (Coruña) hacia la ciudad origen (Madrid).

De esta manera busca en la información de la variable “distancias” la ciudad destino (Coruña). Una vez que encuentra “Coruña”, ingresa en el comienzo de una variable de retorno, denominada “ruta”, a dicha ciudad. Luego, observa quien es su “ciudad padre” y repite la operación (ingresando la nueva ciudad al comienzo del arreglo). En este caso es Sevilla. Desde Sevilla se obtiene la “ciudad padre”, el cual es Jaén. Por último, obtiene que la “ciudad padre” de Jaén es la ciudad de Madrid. Al observar que Madrid es la ciudad de origen, finaliza la búsqueda y retorna la “ruta”.

La secuencia de la información obtenida durante la conformación del arreglo “ruta” es:

	Ruta
I1	Coruña
I2	Sevilla – Coruña
I3	Jaén – Sevilla – Coruña
I4	Madrid – Jaén – Sevilla – Coruña

3. Conclusión

Como se puede observar, el algoritmo “Dijkstra” es muy útil a la hora de resolver problemas en los cuales se deba obtener los caminos más cortos desde un nodo (ciudad) de origen, hacia los demás nodos dentro de un grafo (en este caso un conjunto de ciudades).

Por último, Cabe destacar que, si bien se ha presentado una implementación determinada, existen diversas variaciones de implementación según sea el caso y/o del problema concreto a resolver.