Universidad de Santiago de Chile
Departamento ingeniería informática
Ingeniería civil informática
Algoritmos Avanzados
Laboratorio

2-2018

Generando el camino de costo mínimo

Valentina Ligueño
e-mail: valentina.ligueno@gmail.com

Generating the minimum cost path

RESUMEN: *El siguiente documento va a describir el funcionamiento de un algoritmo recursivo que tiene como finalidad encontrar el camino de costo mínimo en un grafo no dirigido y completo, con tal de que todos los nodos de este sean visitados*

PALABRAS CLAVE: grafo completo, costo, algoritmo recursivo, nodo, fuerza bruta.

ABSTRACT: *The following document, it's going to describe how it works an recursive algorithm whose purpose is to find the minimum cost path in a no-directional and complete graph, provided that all the nodes are visited*

KEYWORDS: complete graph, cost, recursive algorithm, node, brute force.

1 INTRODUCCIÓN

Este documento, tiene como tema principal, la explicación de cómo encontrar un camino de costo mínimo en un grafo, asegurándose de pasar una vez por cada nodo, en este grafo todos los nodos son adyacentes unos con otros. Por lo tanto se espera encontrar la permutación de estos N nodos, que nos genere un costo mínimo según el valor de las aristas que conectan nodo con nodo

.

2 DESCRIPCIÓN DEL PROBLEMA

Como problema, se entrega un enunciado en el que existe una persona que necesita abastecer un reino, y para ello debe pasar por distintos puntos de abastecimiento, intentando recorrer la menor distancia posible, entonces para resolver este problema, la opción escogida es modelar lo en un grafo, donde los nodos representan los puntos de abastecimiento del reino, y las aristas representan las distancias entre unos y otros, por esto se procede a generar un algoritmo que dado un conjunto de nodos y costos entregue como resultado un camino a seguir que pase por cada punto recorriendo lo menos posible.

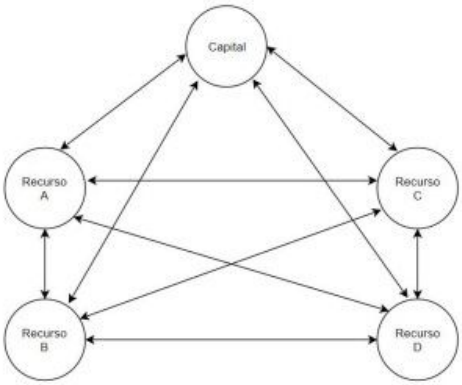

Figura 1: Grafo explicativo

3 MARCO TEÓRICO

Un grafo es una forma de representar un conjunto de información, en el cual se tiene un conjunto de vértices o nodos unidos por ciertos enlaces llamados normalmente aristas, las cuales pueden o no tener un valor asociado, el cual representa el costo de ir de un nodo a otro. En este documento se va a trabajar con un grafo completo, el cual es un grafo en que todos sus nodos son adyacentes, es decir están todos conectados entre sí.

Para resolver el problema de encontrar un camino mínimo que pase por todos los nodos de un grafo completo, se utilizará enumeración explícita o mejor conocida como fuerza bruta, esta forma de resolución de problema consiste en generar todas las posibles soluciones de un problema y a través de ciertas restricciones se

irán descartando soluciones de este conjunto hasta llegar a la solución óptima según lo requerido por el problema. Como la cantidad de nodos que se entregarán en cada prueba puede ser diferente, el algoritmo debe utilizar necesariamente recursión.

.

4 DESCRIPCIÓN DE LA SOLUCIÓN

Para la resolución del problema se tienen los siguientes supuestos:

- El archivo de entrada siempre estará en el formato correcto, es decir, en su primera línea estará la cantidad de nodos existentes y en las siguientes estarán los nodos y sus costos, con el formato de : nodo1 nodo2 costo, por ejemplo 1 2 15
- El grafo entregado en el archivo de entrada siempre será completo
- Nunca se incluirá el nodo 0 en la entrada, que representa a la capital, según lo especificado en la figura 1.
- Considere n el número de nodos.

4.1 IDEA

El problema se decide abordar con fuerza bruta, por lo que primero se tomará el archivo de entrada y se va a generar una matriz de adyacencia con los costos respectivos, luego se va a tomar la cantidad de nodos existentes y se creará un conjunto desde el 1 hasta n. Después se tomará este conjunto y se le aplicará una función que va a generar todas las permutaciones posibles de este conjunto, dichas permutaciones deben ser de tamaño n, el número de permutaciones siempre será n!. Obtenida esta lista con el conjunto de permutaciones, se irá calculando el costo de cada una de estas, y al mismo tiempo se va a ir asignando el costo a una variable mínimo que irá siendo reemplazada solo si encuentra un valor más bajo que el que tiene establecido, se guarda también el conjunto asociado a este costo. Finalmente al terminar de recorrer la lista, el programa entrega cual es la permutación que genera el costo mínimo, y su costo, al cual se le sumará 2, ya que es el costo de ir o volver desde la capital a cualquier nodo (punto de abastecimiento).

4.2 REPRESENTACIÓN Y ESTRUCTURA

Los datos del archivo de entrada se almacenan en una matriz de nxn, en donde si se tiene matriz[i][j] = c. i y j son nodos y la arista que los une tiene un costo c.

El conjunto inicial es representado por un arreglo de char, esto se decida por la facilidad de trabajar con este tipo de dato en C. Por lo tanto si son 4 nodos el conjunto a permutar es: “1234”, y sus permutaciones por lo tanto son un arreglo de arreglos de char, viéndose entonces como [“1234”,”1243”,”2134”...].

4.3 PSEUDOCÓDIGO

```
bruteForce(conjutoInicial):conjuntoSoluciones
   largo = largo(conjuntoInicial)

   visitados = lista vacía  // visit
   resultados = lista vacía //result

   encolar(puzzle,Q1)
   si (largo == 0):
       retornar [“”]

   sino
   nuevoConjunto = conjuntoInicial[1,:]
   listaPermutaciones =
bruteForce(nuevoConjunto)
   listaFinal = [“”]
   largoP = largo(listaPermutaciones);

   para listasP en lista_permutaciones
     listaTemp =
insertarMultiple(listasP,conjuntoInicial[0])
     para listasInsert en listaTemp
        listaFinal.agregar(listasInsert)

   retornar listaFinal

buscarMinimo(listaFinal): void
   minimo = 12345678
   indice = -1;
   para permut en listaFinal
      costo = 0
      para j desde 0 hasta largo(permut)-1
        costo = costo + matrizAdy[j][j+1]
```

```
      si (costo < minimo)
        minimo = costo
        indice = indiceDe(permut)

   minimo = minimo + 2
   Ahora se escribe en el archivo de salida el
costo encontrado y el camino a seguir,
incluyendo el nodo 0 al inicio y al final de la
permutación, que corresponde a la capital

#Comentarios y aclaraciones
- InsertarMultiple es una función que dado una
  lista y un elemento, inserta dicho elemento
  en todas las posiciones de la lista
  entregada, generando una lista nueva por
  cada inserción, por ejemplo con [1,2] y 3,
  entrega de salida =
  [[3,1,2],[1,3,2],[1,2,3]]
- En este apartado se omitió la lectura del
  archivo, y el posterior traspaso a la matriz
  de adyacencia ya que se considera más
  importante la generación de las
  permutaciones y finalmente encontrar la
  solución
- Este seudo-código se ve distinto al código
  real implementado en c, porque en c se
  necesitan más pasos y manejo de memoria para
  lograr algunas cosas, pero en esencia hacen
  lo mismo, solo que en el seudo-código se ve
  más directo, la implementación real es más
  engorrosa.
```

4.4 TRAZA

4.5

Entrada:

```
3
1 2 5
1 3 7
2 3 4
```

Matriz de adyacencia generada:

	1	2	3
1	0	5	7
2	5	0	4
3	7	4	0

```
conjuntoInicial = “123”
bruteForce(“123”)
   nuevoConjunto = “23”
   listaPermutaciones1 = bruteForce(“23”)
      nuevoConjunto = “3”
      listaPermutaciones2 = bruteForce(“3”)
        nuevoConjunto = “”
        listaPermutaciones3 = bruteForce(“”)
        listaPermutaciones3 = [“”]
        para listasP en listaPermutaciones3
          listaTemp = insertarMultiple(“”,3)
          listaTemp = [“3”]
          listaFinal.agregar(“3”)
          listaFinal = [“3”] #Se retorna
      listaPermutaciones2 = [“3”]
      para listasP en listaPermutaciones2
           listaTemp = insertarMultiple(“3”,2)
           listaTemp = [“23”,“32”]
              listaFinal.agregar(“23”)
              listaFinal.agregar(“32”)
          listaFinal = [“23”,“32”] #Se retorna

   listaPermutaciones1 = [“23”,“32”]
   para listasP en listaPermutaciones1
       listaTemp = insertarMultiple(“23”,1)
       listaTemp = [“123”,”213”,”231”]
          listaFinal.agregar(“123”)
          listaFinal.agregar(“213”)
          listaFinal.agregar(“231”)

       listaTemp = insertarMultiple(“32”,1)
       listaTemp = [“132”,”312”,”321”]
          listaFinal.agregar(“132”)
          listaFinal.agregar(“312”)
          listaFinal.agregar(“321”)

          retornar listaFinal =
       [“123”,”213”,”231”,“132”,”312”,”321”]
       #Se retorna y termina el proceso
       recursivo, generando así todas las
       permutaciones del conjunto de nodos 1,2
       y 3, por lo tanto solo falta el cálculo
       del costo para conocer la permutación
       con el camino mínimo
buscarMinimo([“123”,”213”,”231”,“132”,”312”,”32
1”])
   minimo = 12345678
   indice = -1;
   para permut en listaFinal
      costo = 0
      para j desde 1 hasta largo(“123”)-1
        costo = 0 + 5 #costo + matAdy[1][2]
        costo = 5 + 4 #costo + matAdy[2][3]
      costo = 9
      minimo = 9
      indice = 0
      para j desde 0 hasta largo(“213”)-1
        costo = 0 + 5
        costo = 5 + 7
      costo = 12
      #Se omitirá escribir el proceso de
cálculo completo, ya que sigue la misma lógica
      para (“231”) el costo es = 11
      para (“132”) el costo es = 11
      para (“312”) el costo es = 12
      para (“321”) el costo es = 9
   minimo = 9 + 2
   minimo = 11

Ahora entonces se escribe el resultado en un
archivo que se ve de la siguiente manera:
```

```
Costo mínimo: 11
0-1-2-3-0
```

4.6 ORDEN DE COMPLEJIDAD

Para bruteForce, como es un algoritmo recursivo se tiene un tiempo de

$$T(n) = T(n-1) + (n^3)$$

Queda lo anterior porque solo realiza una llamada recursiva a la vez, por lo tanto no hay constante acompañando a T(n-1), se resta 1 porque el conjunto va disminuyendo en un elemento cada vez y finalmente n^3 porque es el orden de la complejidad de la parte no recursiva del algoritmo.
Entonces si luego se utiliza la fórmula para algoritmos recursivos por sustracción. El algoritmo finalmente tiene un orden de :

$$\sim O(n^4)$$

Para buscarMinimo() se tiene una complejidad del orden de $\sim O(n^2)$ porque trabaja con un doble for. Finalmente como $n^4 > n^2$ la complejidad final es resultante entre bruteForce y buscarMinimo es de $\sim O(n^4)$. Se calcula solo la de estas funciones porque son las más importantes y principales para la resolución del problema

5 ANÁLISIS DE LA SOLUCIÓN

La solución implementada trabaja muy bien, hasta los 9 nodos, pero de ahí en adelante no cumple su objetivo, el problema es que se implementó todo con arreglos de char, y al ser 10, el número ocuparía 2 posiciones de un arreglo y el programa lo ve como si fueran números diferentes, al tener “10” un elemento es el 1 y el otro es el 0, por lo que ya no entrega de resultado lo que debería. Al comienzo se intentó hacerlo con arreglos de entero, pero el problema es que en C, no existe una forma precisa de calcular el largo de estos arreglos y el algoritmo no entregaba las soluciones correctas, trabajar con char entrega mayor precisión a cálculo, pero lo limita en la cantidad de nodos que puede operar

5.1 ANÁLISIS DE IMPLEMENTACIÓN

¿Se detiene el algoritmo?
Sí, hablando específicamente de bruteForce, que es el más importante, se detiene en el momento en que genera todas las permutaciones, y en la parte recursiva comienza a devolverse cuando extrae todos los elementos del conjunto dejando un arreglo de chat vacío, desde ahí comienza a generar cada combinación hasta llegar a la primera recursión que se hizo y devuelve la lista con soluciones.

¿Se detiene con la solución?
Si, ya que en buscarMinimo(), es donde comienza a calcular los costos de las soluciones encontradas, y al terminar este proceso, se tiene el costo mínimo y el indice de que solución lo posee. Se hace la aclaración de que pueden haber muchas soluciones con el mismo costo, pero el algoritmo solo almacena la primera de estas que encuentra.

¿Es eficiente?
Si, ya que tiene una complejidad polinómica, específicamente n^3.
¿Se puede mejorar?
Se podría buscar alguna manera de eliminar tanto ciclo anidado existente, también una mejora efectiva sería lograr calcular el costo a medida que la permutación sea generada e ir de inmediato verificando si es el mínimo costo, así a terminar la función bruteForce ya se tendría una solución.

¿Otra idea?
Otra idea sería utilizar otro método de resolución de problemas, tales como algoritmo con retroceso o ramificación y acotamiento, siendo este último el que sería más eficiente para este caso, ya que a medida que va

recorriendo los nodos descarta posibilidades y no genera el conjunto completo de soluciones.

5.2 EJECUCIÓN

Formato de archivo de entrada:
Número de nodos
nodoA nodoB costo
nodoA nodoC costo

```
3
1 2 5
1 3 7
2 3 6
```

Para compilar el programa:
`$ gcc principal.c funciones.c -o programa`

para ejecutar el programa:
`$ ./programa`

Y para entrar al modo DEBUG del programa:
`$ gcc principal.c funciones.c -DDEBUG -o programa`

Importante aclarar que para que el programa compile, el archivo de entrada debe ser tal cual como se explica en el formato, asegurándose de que todos los nodos tengan conexión entre sí, si no se sigue el formato lo más probable es que o el programa no funcione o entrege una solución incorrecta. También asegurarse de que sean máximo 9 nodos, por la razón que se explica en el apartado análisis de la solución.
Cuando el programa comience a ejecutarse, le pedirá que ingrese el nombre del archivo, debe ser muy preciso, si el programa no encuentra el archivo que usted indicó le volverá a pedir el ingreso de nombre hasta que le escriba un nombre de un archivo existente. Solo basta con esto y el programa indicará cuando termine la ejecución y escribirá el resultado en un archivo llamado salida.out

6 CONCLUSIONES

El algoritmo desarrollado calcula las permutaciones en un tiempo muy corto, el problema es que al ser trabajado con caracteres y arreglos de caracteres, no permite trabajar con más de 10 nodos, si el mismo algoritmo fuera implementado para arreglos de enteros, se podría ampliar el dominio de este algoritmo, permitiendo conjuntos más grandes. Esta es la única limitación encontrada en el algoritmo.
Ahora hablando de lo aprendido al realizar este laboratorio, ayudó a la comprensión más completa del concepto de enumeración explícita o mejor conocida como fuerza bruta, ya que es muy distinto ver esta definición de manera teórica que llevarla a la práctica.
Finalmente, se concluye que por lo menos para este problema la mejor manera de abordarlo no es fuerza bruta, ya que tomando otras estrategias de resolución de problemas, se podría reducir el tiempo de ejecución del algoritmo, para que este no tenga que calcular todas las posibles soluciones existentes para el problema.

7 REFERENCIAS

- “plantilla Paper Lab Avanzados”, Nicolás Gutiérrez, 2018, disponible en : www.ejemplo.com/ngutierrez/Avanzados
- “Complejidad de algoritmos recursivos”, Alberto Castro, 2011, disponible en: http://btocastro.blogspot.com/2011/07/complejidad-de-algoritmos-recursivos.html
- “Combinaciones, permutaciones y otras diversiones”, Ariel Ortiz, 2016, disponible en: http://edupython.blogspot.com/2016/06/combinaciones-permutaciones-y-otras.html