DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

Técnicas Algorítmicas Avanzadas

Viernes 9 de Mayo de 2014

Algoritmos y Estructuras de Datos III
Entrega de TP

Grupo ??

Integrante	LU	Correo electrónico
Barrios, Leandro E.	404/11	ezequiel.barrios@gmail.com
Benegas, Gonzalo	958/12	gsbenegas@gmail.com
Melnik, Jonathan	571/09	jonathanmelnik@gmail.com
Vanecek, Juan	169/10	juann.vanecek@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

En este trabajo práctico nos piden analizar el problema del *Camino Acotado de Costo Mínimo* ($CACM$), y desarrollar distintos algoritmos para resolver el mismo.

El problema consiste en que dado un Grafo $G = (V, E)$, dos funciones de peso $\omega_1, \omega_2 : V \mapsto \mathbb{R}_+$, y un natural K, encontrar un camino P entre dos nodos $u, v \in V$ con costo $\omega_1(P) \leq K$ de manera tal que $\omega_2(P)$ sea mínimo.

Donde el costo del camino $\omega_x(P)$, con $1 \leq x \leq 2$, se define como

$$\omega_x(P) = \sum_{e \text{ arista de } P} \omega_x(e)$$

$CACM$ es un problema conocido, y tiene muchas aplicaciones en la vida real. Por ejemplo, supongamos que somos una empresa de turismo que ofrece paquetes de viajes. Una situación que se nos puede presentar es que un cliente nos pide organizarle un viaje de una ciudad X a otra ciudad Y para poder llegar en el mínimo tiempo, pero nos dice que el presupuesto que cuenta para gastar en transporte es de $\$K$. Este problema se puede modelar con $CACM$, donde las ciudades son los nodos del grafo, las aristas del mismo existen si entre las ciudades en cuestión hay algún medio de transporte, ω_1 representa el costo del viaje, y ω_2 es el tiempo que toma el viaje.

Aunque si bien $CACM$ es un problema conocido, no se conocen algoritmos polinomiales que lo resuelvan y por lo tanto pertenecen al conjunto de los problemas NP. Nosotros en este TP analizaremos 6 métodos para resolverlo, una solución exacta y 5 aproximaciones a través de heurísticas. En concreto, los algoritmos que implementaremos son:

1. Un *Backtracking* como algoritmo exacto.
2. Tres heurísticas *constructivas greedy*, cada una con un criterio goloso diferente.
3. Una heurística de *búsqueda local*.
4. Una heurística *GRASP*.

Nos centraremos en experimentar sobre estos algoritmos, analizando su complejidad y la calidad de las soluciones de las heurísticas. También trataremos de definir las familias de grafos para las cuáles las heurísticas implementadas funcionan muy bien, y aquellas para las cuáles las mismas hallan una solución muy alejada de la óptima, o lo que es peor, que podrían no hallar ninguna.

2. Instrucciones de uso

3. Desarrollo del TP

3.1. Backtracking

Debido a la dificultad computacional del problema, no existe aún una solución exacta de tiempo polinomial, y a pesar que lo discutimos nosotros no pudimos encontrarla tampoco. En su defecto implementamos un algoritmo de backtracking que recorre todos los caminos posibles de u a v y se queda con el de menor ω_2 tal que $\omega_1 \leq K$.

El algoritmo funciona de la siguiente manera: en un momento dado va a tener construido un camino $P = [v_1, \ldots, v_{i-1}]$, y toma un nodo v_i, lo marca como visitado y lo agrego a P. Luego si $\omega_1(P)$ cumple que es menor que igual a K, entonces me fijo si v_i es v, en dicho caso me fijo si el peso de P según ω_2 es mejor al que hubiera encontrado como óptimo anteriormente, o lo guardo como una potencial solución si es la primera que encuentra. Si v_i no es el nodo buscado, entonces repito todo el procedimiento para cada vecino de este que no haya sido recorrido todavía.

De esta forma, el backtracking empieza con $P = \emptyset$ y tomando como nodo inicial a u, y va a parar cuando haya recorrido todas las posibles formas de llegar hasta v, y cuando lo haga va a haber guardado la solución óptima.

A continuación, escribimos el pseudocódigo de la función `main`

Algoritmo 1 main()

```
1: Camino mejorSolucion = []
2: Camino ramaActual = []
3: Nodo u, v
4: Grafo G
5: backtrack( u, u )
```

Algoritmo 2 backtrack(Nodo actual, Nodo padre)

```
1: ramaActual.push( actual )
2: visitados[actual] ← true
3: si ω1(ramaActual) < K entonces
4:     si actual = dst and ω2(ramaActual) < ω2(mejorSolucion) entonces
5:         mejorSolucion ← ramaActual
6:     sino si actual ≠ dst entonces
7:         para cada Nodo n en G.adyacentes( actual ) hacer
8:             si no visitado[n] entonces
9:                 backtrack( n, actual )
10:            fin si
11:        fin para
12:    fin si
13: fin si
14: ramaActual.pop( actual )
15: visitados[actual] ← false
```

De esta forma, cuando corremos `backtrack(src, src)` y termina, vamos a tener en `mejorSolucion` la solución óptima.

3.2. Greedy

Debido a que la solución exacta es tan costosa de conseguir, contruimos un algoritmo goloso que toma una función objetivo $f : E \mapsto \mathbb{R}_+$ para hayar la solución. La idea de esto es obtener una solución aproximada en tiempo polinomial. Analizaremos el uso de las 3 criterios y evaluaremos cuál implementación consigue soluciones de mejor calidad.

Ahora consideremos el siguiente pseudocódigo:

In: Grafo $G = (V, X)$, nodo inicial v_0, ObjectiveFunction f
Out: Arreglo π con camino mínimo a cada nodo.

1: $\pi(v) = \infty \quad \forall v \in V$
2: $\pi(v_0) = 0$
3: $S = \emptyset$
4: **para** $i = 1 \dots n - 1$ **hacer**
5: $\quad v \leftarrow$ nodo de $V \backslash S$ de mínimo π.
6: $\quad$ **para each** $w \in V \backslash S$ adyacente a v **hacer**
7: $\quad\quad \pi(w) = \text{mín}(\pi(w), \pi(v) + f(v, w))$
8: $\quad$ **fin para**
9: $\quad S = S \bigcup \{v\}$
10: **fin pararetornar** π

Este algoritmo es una modificación de Dijkstra, que en la línea 7, en vez de sumar $\pi(v)$ más el peso de la arista, como es en el algoritmo original, lo hace más el valor de una función que define el peso de la arista. Esto nos permite mucha flexibilidad a la hora de cambiar la "decisión golosa", y elegir un camino mínimo diferente a otro usando otro criterio. Así las 3 funciones que elegimos para experimentar son:

1. $f(e) = \omega_1(e)$
2. $f(e) = \omega_2(e)$
3. $f(e) = \omega_1(e)\omega_2(e)$

3.3. Local Search

Partimos desde una solución factible obtenida a partir de un algoritmo goloso. En caso de que el algoritmo anterior no devuelva una solución factible, corremos Dijkstra utilizando la sumatoria de los pesos ω_1 como función objetivo. Si Dijkstra tampoco devuelve una solución factible, podemos asegurar que no existe solución al problema[1]. En este caso, devolvemos "no".

Solución Inicial 2: Corro dijkstra con omega1 y omega2, formando c_1 y c_2. Tomo el conjunto U de nodos formados por $c_1 \cap c_2$. Para cada par de nodos n_1, n_2 adyacentes, me fijo si puedo formar un camino mejor valuado en ω_2 reemplazando el camino $c^1_{n_1,n_2}$ por $c^2_{n_1,n_2}$, siempre que el nuevo camino no se pase de K al valuarlo en ω_2. La evaluación se hace ordenando por omega2, de forma tal que el camino obtenido sea el que minimice la misma en comparación con el resto de los posibles caminos que se podrían obtener con este método.

[1] Demostrado en la sección de heurística golosa

3.4. GRASP

4. Apéndices

4.1. Código Fuente (resumen)