UNIVERSIDAD DE BUENOS AIRES

FACULTAD DE INGENIERÍA

Año 2017 - 1er Cuatrimestre

Teoría de Algoritmos I

Trabajo Práctico 2

TEMA: Reducciones y Algoritmos de Caminos Mínimos

FECHA: 24 de junio de 2017

INTEGRANTES:

KEKLIKIAN, Nicolas - #96480

<nkeklikian@gmail.com>

GUZZARDI, Gonzalo - #94258

< gonzaloguzzardi@gmail.com>

COVA, Alejo - #94325

<covaalejo@gmail.com>

Índice

1. Algoritmos de camino minimo

1.1. Aspectos generales de los algoritmos.

Sabiendo que se tiene un digrafo completo con pesos en las aristas:

- $Dijkstra \in \mathcal{O}(|V|^2)$: Camino más corto de un nodo a todos los nodos.
- $Bellman - Ford \in \mathcal{O}(|V||E|) \rightarrow \mathcal{O}(|V||V|(|V| - 1)/2|) \in \mathcal{O}(|V|^3)$: Camino más corto de un nodo a todos los nodos. Permite caminos con peso negativo.
- $Floyd - Warshall \in \mathcal{O}(|V|^3)$: Camino entre todos los pares de nodos. Permite caminos con peso negativo.

1.2. Algoritmo de Dijsktra (Greedy)

1.2.1. Explicación del algoritmo

Dado un grafo y un vértice del grafo como fuente, el algoritmo encuentra los caminos mínimos de dicha fuente a todos los vértices del respectivo grafo.

El algoritmo genera un árbol de camino minimo, usando la fuente seleccionada como raíz del árbol. Mantenemos dos conjuntos, un conjunto contiene los vértices incluidos en el árbol de camino mínimo, otro conjunto incluye los vértices aún no incluidos en el árbol de camino mínimo. En cada paso del algoritmo, encontramos un vértice que está en el otro conjunto (conjunto de aún no incluido) y tiene una distancia mínima de la fuente.

A continuación se detallan los pasos utilizados en el algoritmo de Dijkstra para encontrar el camino más corto desde un vértice de fuente única a todos los otros vértices en el gráfico dado:

1. Cree un conjunto SPT (conjunto de árbol de trayecto más corto) que mantiene un seguimiento de los vértices incluidos en el árbol de trayecto mínimo, es decir, cuya distancia mínima de origen es calculada y terminada. Inicialmente, este conjunto está vacío.
2. Asigne un valor de distancia a todos los vértices en el gráfico de entrada. Inicializar todos los valores de distancia como infinito. Asigne el valor de la distancia como 0 para el vértice de origen de modo que se seleccione primero.
3. Mientras SPT no incluye todos los vértices:
   - Elija un vértice u que no esté en SPT y tenga un valor de distancia mínima.

- Se incluye u a SPT.
- Actualiza el valor de la distancia de todos los vértices adyacentes de u. Para actualizar los valores de distancia, iterar a través de todos los vértices adyacentes. Para cada vértice adyacente v, si la suma del valor de la distancia de u (desde la fuente) y el peso de la arista (camino) u-v, es menor que el valor de la distancia de v, entonces actualice el valor de la distancia de v.

A continuación se detalla un ejemplo práctico del algoritmo.

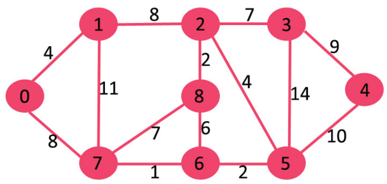

Figura 1: Condición inicial del algoritmo de Dijkstra.

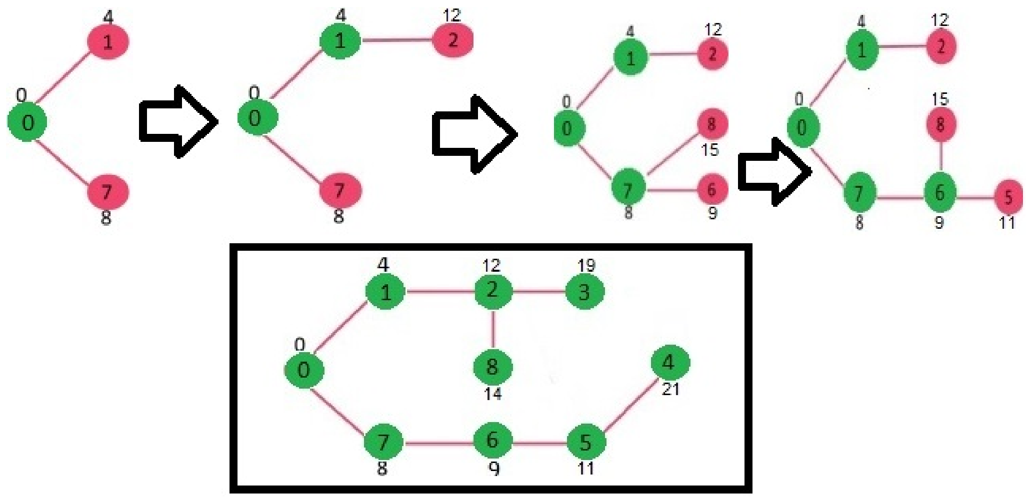

Figura 2: Proceso y resultado del algoritmo de Dijkstra.

1.3. Algoritmo de Bellman-Ford (Programación dinámica)

1.3.1. Explicación del algoritmo

Al igual que otros problemas de programación dinámica, el algoritmo calcula las rutas más cortas de manera bottom-up. Primero calcula las distancias más cortas para los trayectos más cortos que tienen al menos un vértice a en el trayecto. A continuación, calcula las rutas más cortas con un máximo de 2 vértices, y así sucesivamente. Después de la i-ésima iteración del bucle exterior, se calculan los trayectos más cortos con como máximo i vértices. Puede haber un máximo |V| - 1 vértices en cualquier camino simple, es por eso que el lazo externo ejecuta |v| - 1 vez. La idea es, suponiendo que no hay un ciclo de peso negativo, si hemos calculado las trayectorias más cortas con un máximo de vértices i, entonces una iteración sobre todos los vértices garantiza dar el camino más corto con los extremos (i+1).

A continuación se detalla un ejemplo práctico del algoritmo.

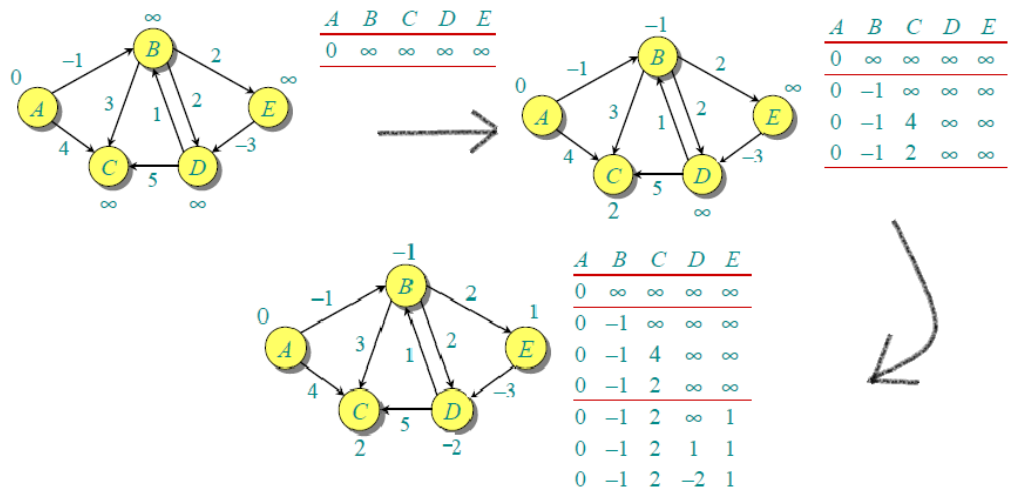

Figura 3: Ejemplo completo del algoritmo de Bellman-Ford.

1.3.2. Rendimiento y conclusiones teóricas

1.4. Algoritmo de Floyd-Warshall (Programación dinámica)

1.4.1. Explicación del algoritmo

El algoritmo de Floyd-Warshall compara todas las rutas posibles a través del grafo entre cada par de vértices.

Considerando un grafo G con vertices |V| numerados de 1 a N y considerando la funcion shortest-Path(i,j,k) la cual retorna el camino mas corto desde i a j usando vértices del conjunto 1,2,. . . ,k+1. solamente como puntos intermedios a lo largo del camino.

A continuación se explica el algoritmo en pseudocodigo:

```
Sea dist un arreglo de distancias min. inicializadas a INF de dim. |V|x|V|
Para cada vertice v
        dist[v][v] <- 0
Para cada arista (u,v)
        dist[u][v] <- w(u,v)   // el peso de la arista o camino (u,v)
Para k desde 1 a |V|
```

```
Para i desde 1 a |V|
        Para j de 1 a |V|
                Si dist[i][j] > dist[i][k] + dist[k][j]
                        dist[i][j] <- dist[i][k] + dist[k][j]
                Sino
                        nada.
```

A continuación se detalla un ejemplo práctico del algoritmo.

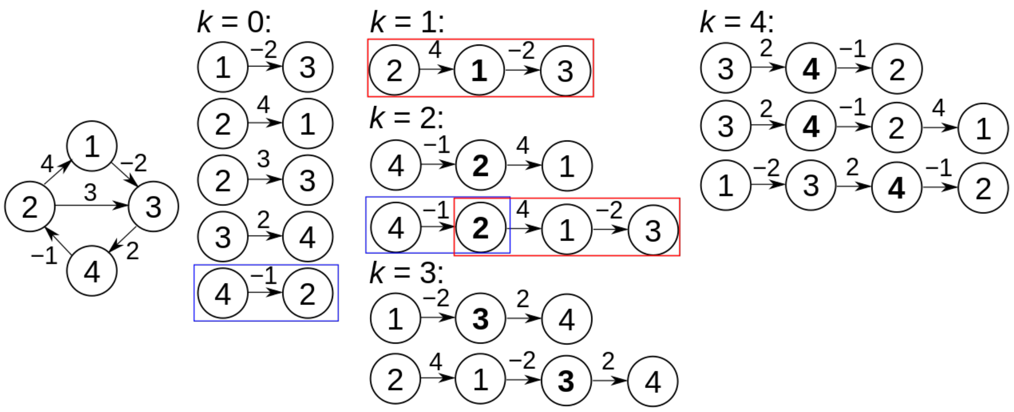

Figura 4: Ejemplo completo del algoritmo de Floyd-Warshall.

1.4.2. Rendimiento y conclusiones teóricas

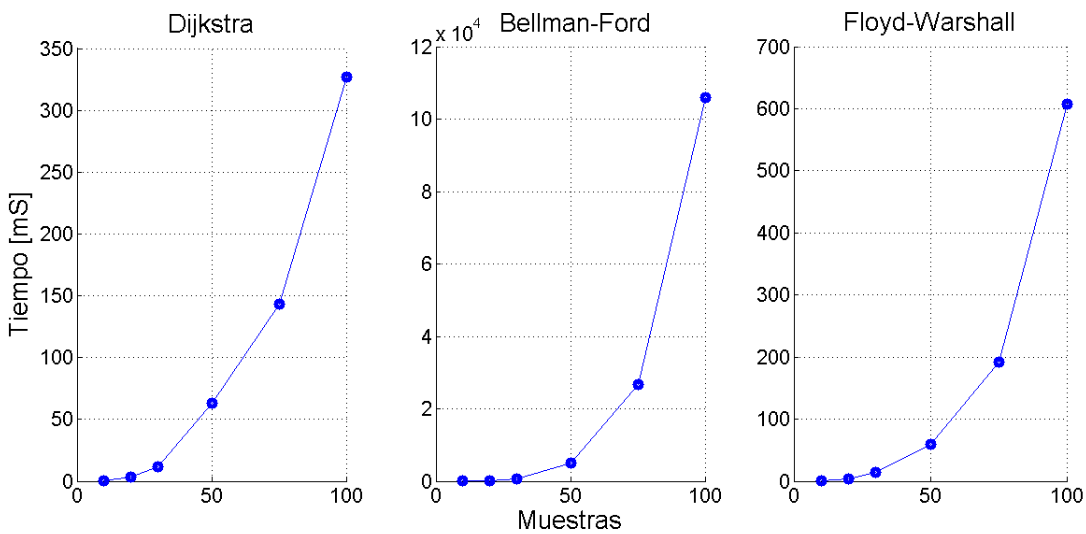

Figura 5: Resultados obtenidos.

Como era de esperar, Djikstra es el algoritmo que más rápido encuentra el camino mínimo con las limitaciones de que solo admite un vértice como origen y no admite aristas con pesos negativos.

El algoritmo de Floyd-Warshall,por otro lado, sí admite aristas de peso negativo con el costo de encontrar solución en un tiempo ligeramente mayor a Djisktra.

Finalmente, el algoritmo Bellman-Ford puede utilizarse en problemas con múltiples vértices que actúan como origen y en grafos con aristas de peso negativo, con el costo de correr en un tiempo considerablemente más alto a los algoritmos anteriores.

2. Clases de complejidad

A continuación se encuentran los algoritmos correspondientes a cada problema:

Algorithm 1 Subconjunto compatible: Tiempos de publicación y plazos (GREEDY)

```
procedure CERTIFICADOR_EJ1(S, K)        ▷ Certifica el subconjuto S para un tamaño K.
    N, d, r, k ∈ ℕ        ▷ Tareas, tiempo de finalizacion e inicio y cantidad de elementos.
    SORT(S)        ▷ Ordenar S en orden creciente de tiempo de finalización. O(n log n)
    S_aux = S[0]        ▷ Seleccionar la primera la primera actividad de S.
    prev = 0        ▷ Actividad previa acumulada
    for i ∈ S.quantity − 1 do        ▷ O(n)
        if S[i].r ≥ S[prev].d then
            S_aux[++prev] = S[i]        ▷ Se agrega el elemento al subconjunto
    if S_aux.k ≥ K then
        return TRUE        ▷ Solución valida
    else
        return FALSE        ▷ Solución invalida
```

$\Rightarrow EJ_1 \in \mathcal{O}(n \log n) \Rightarrow EJ_1 \in P$

Algorithm 2 Subconjunto compatible: SCHEDULE-RELEASE-TIMES (SRT)

$\Rightarrow CLIQUE \in NPC$

G = [E, V], v, u ∈ V, e = [u, v] ∈ E

procedure REDUCCION($E, V, CERTIFICADOR\_EJ1$)

  $r_{ji} \in R_i, t_{ji} \in T_i$ ▷ Tiempo de inicio y fin (j) del trabajo i. $R_i, T_i \in \mathbb{N}$ y $R_i < T_i$

  N = |V| ▷ La cantidad de vértices es la cantidad de tareas

  for all v ∈ V **do** ▷ Si ∃ e=[u,v] ⇒ ∃ intervalos disjuntos en las tareas u y v.

    AGREGAR(R,v.r) ▷ R = $[r_1, r_2, ..., r_N]$

    AGREGAR(T,v.t) ▷ T = $[t_1, t_2, ..., t_N]$

  return $SRT\_SOLVER(R, T, N, CERTIFICADOR\_EJ1)$

$\Rightarrow CLIQUE \leq_p SRT \Rightarrow$ SRT∈ NPH

⇒ ∃ Certificador ∈ P que certifica una solución al problema SRT ⇒ $SRT \in$ NP.

$NP \cap NPH = NPC \Rightarrow SRT \in$ **NPC**

Algorithm 3 Subconjunto compatible: Caminos hamiltonianos en grafos

$Path \in V$ ▷ Lista de vértices que constituyen un camino G

procedure CERTIFICADOR_EJ3($E, V, Path$)

vertexCount = 0

for all $v \in Path$ **do** ▷ Checkea que la cantidad de nodos del camino sea |V|$\mathcal{O}(|V|)$

vertexCount++

if $vertexCount \neq |V|$ **then** ▷

return $FALSE$ ▷ Solución inválida

walkedNodes = 1

currentNode = Path[0]

while (**do**$currentNode.next \neq NULL$) ▷ Checkea que los nodos formen un camino. $\mathcal{O}(|V|)$

walkedNodes++

currentNode = currentNode.Next

if $walkedNodes \neq vertexCount$ **then** ▷

return $FALSE$ ▷ Solución inválida

return $TRUE$ ▷ Solución válida

$\Rightarrow HAMILTONIAN\_CYCLE \in NPC$

G_{aux} = Copy(G) ▷ Construimos un nuevo Grafo G_{aux} a partir de G.

G_{aux}.addVertex(u_{aux}) ▷ Agrega un vértice u_{aux}

G_{aux}.addEdge(e.u, u_{aux}) ▷ agrega una arista y conecta e.u con u_{aux}

G_{aux}.addVertex(v_{aux}) ▷ agrega un vértice v_{aux}

e.addEdge(e.v, v_{aux}) ▷ agrega otra arista y conecta e.v con v_{aux}

G_{aux}.removeEdge(e) ▷ Remueve la arista e

existsHCycle = HamiltonianPath(G, u_{aux}, v_{aux})▷ $\exists HPath \in G_{aux} \Leftrightarrow \exists HCycle \in$ G.I[u_{aux},v_{aux}].

return $existsHamiltonianCycle$

$HAMILTONIAN\_CYCLE \leq_p HAMILTONIAN\_PATH$

$\Rightarrow EJ_3 \in NPC$

Algorithm 4 Subconjunto compatible: Caminos hamiltonianos en digrafos aciclicos

procedure CERTIFICADOR_EJ3(E, V)

orderedList = TopologicalSort(E,V) ▷ Ordenamiento topologico del grafo G. $\mathcal{O}(|E| + |V|)$.

for i∈ orderedList.length-1 **do** ▷ $\mathcal{O}(|V|)$

nextVertex = orderedList.getNext()

if $vertex.hasEdgeToward(nextVertex)) \neq NULL$ **then** ▷ Si el vértice no tiene una arista hacia el siguiente

return $FALSE$ ▷ Solución inválida

return $TRUE$ ▷ Todos los vértices de la lista tienen una arista hacia el vértice siguiente.

$\Rightarrow EJ_4 \in \mathcal{O}(|E| + |V|) \Rightarrow EJ_5 \in P$

Algorithm 5 Subconjunto compatible: Ciclos negativos en digrafo con pesos (Bellman-Ford)

$v, u \in \mathbb{V}$ ▷ Vértices.

$s \in \mathbb{V}$ ▷ Vértice inicial del grafo.

$v.length \in \mathbb{N}$ ▷ Longitud del camino mas corto de s a v.

procedure CERTIFICADOR_EJ5(E, V)

$BELLMAN\_FORD(E, V)$ ▷ Todos los caminos mínimos de s a cualquier v. $\mathcal{O}(|E| \cdot |V|)$

for all $(u, v) \in E$ **do** ▷ $\mathcal{O}(|E|)$

if $u.length + weigth(u, v) < v.length$ **then** ▷ Si tenemos un ciclo negativo

return $TRUE$ ▷ Solución válida

return $FALSE$ ▷ Solución inválida

$\Rightarrow EJ_5 \in \mathcal{O}(|E||V|) \Rightarrow EJ_5 \in P$

Algorithm 6 Subconjunto compatible: Ciclos negativos nulos

procedure CERTIFICADOR_EJ6(E, V)

  acum = 0 // peso acumulado

  for $v \in S$ **do**

    acum += weight(vertex, vertex.next)

  return (acum == 0)

S = a1, a2, ... , an ▷ Conjunto de enteros

Construimos un digrafo G con 2n vertices, por cada ai creamos 2 vertices y los unimos con una arista de peso ai de vi a ui. Luego creamos una arista de peso cero de todos los vj a ui y de todos los uj a vi, con i distinto de j. Numero total de aristas agregadas = 2n(n+1) + n.

G = NULL ▷ Grafo vacio

for int i = 0; i <S.length; ++i **do**

  G.addVertex (vi); // agrega vertice vi

  G.addVertex (ui); // agrega vertice ui // agrega arista con peso ai de vi a ui. Agrega n aristas

  G.addEdge (vi, ui, S[i]);

  for int j = 0; j <S.length; ++j **do** ▷ agrega 2n (n+1) aristas

    if i≠j **then**

      G.addEdge (uj, vi, 0) ▷ agrega aristas de peso cero de uj a vi

      G.addEdge (vi, uj, 0) ▷ agrega aristas de peso cero de vi a uj

Existe un subset no vacio cuya suma vale cero si y solo si encontramos en el digrafo G un ciclo de peso cero. De esta forma, este problema es al menos tan dificil como el subset problem que es NP-Completo. Por ejemplo, si a1 + a2 + a4 = 0, // existira un ciclo nulo en el grafo G: v1, u1, v2, u2, v4, u4, v1.

existeSubsetZero = existeCicloNulo (G)

$\Rightarrow EJ_6 \in NPC$