Teoría de Algoritmos – TP1

“Amigos en Facebook”

Materia: 75.29 Teoría de Algoritmos

Alumno: Juan Sebastián Goldberg

Padrón: 82078

Mail: sebas.goldber@gmail.com

Fecha: 14-OCT-2014

Contenido

[Análisis y Diseño 3](#__RefHeading__391_277142692)

[Popularidad 3](#__RefHeading__3870_1637238414)

[Influencias 3](#__RefHeading__3872_1637238414)

[Obtención de cantidad de caminos mínimos entre dos vértices 3](#__RefHeading__3874_1637238414)

[Obtención de cantidad de caminos entre dos vértices cualesquiera que tiene como intermediario un determinado vértice 3](#__RefHeading__3876_1637238414)

[Cálculo de influencias 4](#__RefHeading__3878_1637238414)

[Recomendaciones 5](#__RefHeading__3880_1637238414)

[Diagramas 5](#__RefHeading__3882_1637238414)

[Módulos 5](#__RefHeading__3884_1637238414)

[Clases 5](#__RefHeading__3886_1637238414)

[Justificación de órdenes del lenguaje y librerías 5](#__RefHeading__3888_1637238414)

[Código Fuente 6](#__RefHeading__3658_1637238414)

[tp1.py 6](#__RefHeading__3660_1637238414)

[grafo.py 12](#__RefHeading__3662_1637238414)

[lista\_ordenada.py 17](#__RefHeading__3664_1637238414)

Análisis y Diseño

A continuación se realiza el análisis y se proponen soluciones para realizar cada una de las operaciones solicitadas en el trabajo práctico.

Popularidad

La forma de resolver esta operación es simplemente contar el grado de salida (o entrada) de cada vértice del grafo no dirigido. Si se implementa el grafo con una lista de adyacencia, la operación de obtener el grado de salida para un vertice cualquiera es O(1), con lo cual obtener la popularidad para todos los vertices del grafo tendrá orden O(|V|).

Influencias

La obtención del indice de influencia para cualquier vertice se obtiene simplemente aplicando la fórmula indicada en el enunciado del trabajo práctico.

Las dificultades aquí obviamente son:

1. obtener la cantidad de caminos mínimos de entre dos vértices,
2. y obtener la cantidad de caminos entre dos vértices cualesquiera que tiene como intermediario un determinado vértice.

Obtención de cantidad de caminos mínimos entre dos vértices

Determinar este valor en realidad no es muy complicado. Para poder llevar a cabo esta tarea, simplemente es cuestión de adaptar el algoritmo de Dijkstra de forma que se acepte más de un camino mínimo. Este algoritmo adaptado se puede visualizar en el método <#calcular_camino_minimo>. Como se puede observar en la implementación lo que se hizo fue simplemente incorporar la posibilidad de tener múltiples padres en un camino dado. El orden para obtención de caminos mínimos para un vertice dado es O(|E|\*log(|V|)), que es básicamente el orden de cualquier Dijkstra que utiliza cola de prioridad.

Una vez obtenidos todos los caminos mínimos para un vertice u, obtener la cantidad de caminos mínimos a cualesquier vértice v, simplemente es hacer un recorrido en anchura de v a u utilizando los padres calculados en Dijkstra. El motivo de que sea un recorrido en anchura, es básicamente el hecho de que podemos pensar que la cantidad de caminos de v a u, es la suma de la cantidad de caminos de cada vértice padre de v hacia u, siendo el caso base de esta definición recursiva 1 para cuando v es igual a u. Este algoritmo se puede visualizar en el método <#get_cantidad_caminos_minimos>. Su orden es O(|V|+|E|) que es básicamente el orden para realizar un recorrido en anchura.

Obtención de cantidad de caminos entre dos vértices cualesquiera que tiene como intermediario un determinado vértice

Dados tres vertices del grafo u, v y w distintos. Obtener todos los caminos mínimos que van de u a v que pasan por w puede parecer una tarea compleja, pero en realidad no lo es.

Realizar esta tarea es tan simple como verificar si existe un camino mínimo de u a v, de u a w y de w a v, siendo las correspondientes distancias d(u,v), d(u,w) y d(w,v). Si se verifica que d(u,v) es igual a la suma entre d(u,w) y d(w,v), esto significa que w forma parte de al menos un camino mínimo de u a v. Luego la cantidad de caminos mínimos de u a v que pasan por w, será la cantidad de caminos mínimos que van de u a w multiplicado por la cantidad de caminos mínimos que van de w a v.

Teniendo precalculados las distancias de los caminos mínimos y la cantidad de caminos mínimos entre dos vertices, el orden de esta tarea se ve reducido a O(1).

La implementación de este algoritmo se puede visualizar en el método <#get_cantidad_caminos_minimos_con_intermediario>.

Cálculo de influencias

Habiendo solucionado lo mencionado en las secciones anteriores, simplemente nos queda aplicar la formula dada en el enunciado por cada vertice, pero para poder hacer esto, previamente deberíamos pre calcular todos los caminos mínimos entre cada par de vértices, y deberíamos pre calcular la cantidad de caminos mínimos entre cada par de vértices. Luego, sí aplicaríamos el cálculo de influencias.

Para poder determinar el orden tenemos lo siguiente:

* Calculo de caminos mínimos:
  + O(|V|\*|E|\*log(|V|))
  + Básicamente un Dijkstra por cada vertice.
* Pre procesamiento de la cantidad de caminos mínimos:
  + O((|V|\*\*2)\*(|V|+|E|))
  + En este caso se realiza un recorrido en anchura por cada par de vertices.
* Calculo del índice:
  + O(|V|\*\*3)
  + Para el calculo del índice es necesario verificar todas las combinaciones posibles del conjunto VxVxV de forma de poder aplicar la formula dada.

Con lo cual vemos que el orden del algoritmo es el correspondiente a O((|V|\*\*2)\*(|V|+|E|)). El algoritmo que se encarga de obtener los índices de influencia es <#get_influencias>.

Recomendaciones

Para poder realizar recomendaciones, la idea principal es la de verificar si dos vértices están conectados y en caso de no estarlo verificar cuantos vértices tienen en común.

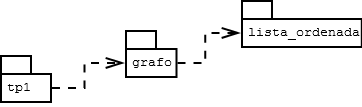
Para verificar si dos vértices u y v están conectados, en caso de utilizar listas de adyacencia ordenadas que soporten acceso por índice, simplemente basta con realizar una búsqueda binaria. Dicha operación será de orden O(log(|Au|)) donde Au es el conjunto de aristas que salen de u.

Para verificar cuántos vértices tienen en común u y v, en caso de no estar conectados, basta con calcular la intersección entre Au y Av. Utilizando listas de adyacencia ordenadas que soporten acceso por índice, el orden de la operación de intersección se ve disminuido a O(|Au|+|Av|).

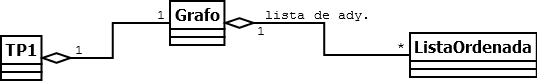
Luego, obtener las recomendaciones para el vértice u tendrá orden O(Au\*|V|+|A|), método <#recomendaciones_para>, y para obtener recomendaciones para todos los vértices el orden ascenderá a O(|V|\*|A|), método <#recomendaciones>.

Diagramas

Módulos



Clases



Justificación de órdenes del lenguaje y librerías

El lenguaje utilizado para la implementación del trabajo práctico es python.

Los órdenes referentes a operaciones realizadas con las estructuras provistas por el lenguaje y librerías se obtuvieron a partir de la siguiente documentación (en todos los casos se trabajo con los casos pesimistas):

* Ordenes para estructuras de datos nativas:
  + <https://wiki.python.org/moin/TimeComplexity>
* Documentación de módulo estándar de colas de prioridad:
  + <https://docs.python.org/2/library/heapq.html>
* Documentación de módulo estándar para realizar búsquedas binarias e inserción ordenada:
  + <https://docs.python.org/2/library/bisect.html>

Código Fuente

Para acceder a la totalidad del código fuente puede clonar el siguiente repositorio: <https://github.com/facultad/7529-tp1>

A continuación se incorpora el código fuente del trabajo práctico, y para ganar claridad y disminuir la cantidad de código se excluyeron los casos de prueba de cada módulo.

La mayoría de los métodos fueron documentados de forma de registrar el orden de los algoritmos implementados.

tp1.py

|  |
| --- |
| *#!/usr/bin/python* |
| *# coding=utf-8* |
| **from** grafo **import** Grafo |
| **import** re |
| **from** heapq **import** heappop, heappush |
| **import** sys |
|  |
| NODEDEF\_TYPE **=** 1 |
| EDGEDEF\_TYPE **=** 2 |
| re\_nodedef **=** re**.**compile('^nodedef>.\*') |
| re\_edgedef **=** re**.**compile('^edgedef>.\*') |
|  |
| **class** **Node**: |
|  |
| **def** **\_\_init\_\_**(self, id, description): |
| self**.**id **=** int(id) |
| self**.**description **=** description |
|  |
| @staticmethod |
| **def** **create\_node\_from\_gdf\_line**(line): |
| """ |
| O(1) |
| """ |
| fields **=** line**.**split(',') |
| **return** Node(id**=**fields[0], description**=**fields[1]) |
|  |
|  |
| **class** **TP1**: |
|  |
| **def** **\_\_init\_\_**(self, filepath): |
| """ |
| O(max(|E|,|V|)\*|V|) |
| """ |
| self**.**grafo **=** Grafo() |
| self**.**vertice\_from\_id **=** {} |
| linetype **=** None |
|  |
| **with** open(filepath) **as** f: |
| **for** line **in** f: |
| line **=** line**.**strip() |
| **if** re\_nodedef**.**match(line): |
| linetype **=** NODEDEF\_TYPE |
| **elif** re\_edgedef**.**match(line): |
| linetype **=** EDGEDEF\_TYPE |
| **elif** linetype **==** NODEDEF\_TYPE: *# |V|* |
| *# O(1)* |
| node **=** Node**.**create\_node\_from\_gdf\_line(line) |
| *# O(|V|)* |
| self**.**vertice\_from\_id[node**.**id] **=** self**.**grafo**.**add\_node(node) |
| **elif** linetype **==** EDGEDEF\_TYPE: *# |E|* |
| id1, id2 **=** map(int, line**.**split(',')) |
| *# O(|V|)* |
| v1, v2 **=** self**.**vertice\_from\_id[id1], self**.**vertice\_from\_id[id2] |
| *# O(log(|V|))* |
| self**.**grafo**.**connect(v1, v2, both**=**True) |
|  |
| **def** **get\_popularidad**(self): |
| """ |
| O(|V|) |
| Obtiene un listado L donde el componente L[i] contiene un |
| listado de vertices con popularidad i. len(L) == |V|. |
| """ |
|  |
| popularidad **=** [] |
|  |
| *# O(|V|)* |
| **for** i **in** self**.**grafo**.**iternodes(): |
| popularidad**.**append([]) |
|  |
| *# O(|V|)* |
| **for** u **in** self**.**grafo**.**iternodes(): |
| *# O(1)* |
| grado **=** self**.**grafo**.**get\_grado\_salida(u) |
| popularidad[grado]**.**append(self**.**grafo**.**get\_node\_data(u)) |
|  |
| **return** popularidad |
|  |
| **def** **mostrar\_popularidad**(self, popularidad): |
| """ |
| O(|V|) |
| """ |
| **for** i **in** xrange(len(popularidad)**-**1, **-**1, **-**1): |
| **if** len(popularidad[i]) **==** 0: |
| **continue** |
| **print** "#%s: %s" **%** (i, [ x**.**description **for** x **in** popularidad[i]]) |
|  |
| **def** **mostrar\_influencias**(self, influencias): |
| """ |
| O(|V|\*\*2) |
| """ |
|  |
| vertices\_por\_influencia **=** {} |
|  |
| *# O(|V|\*\*2)* |
| **for** i **in** xrange(len(influencias)): *# |V|* |
| *# O(|V|)* |
| vertices **=** vertices\_por\_influencia**.**get( |
| influencias[i],set()) |
| *# O(|V|)* |
| vertices**.**add(self**.**grafo**.**get\_node\_data(i)**.**description) |
| *# O(|V|)* |
| vertices\_por\_influencia[influencias[i]] **=** vertices |
|  |
| *# O(|V|)* |
| influencias\_ordenadas **=** [(influencia, vertices |
| ) **for** influencia,vertices **in** vertices\_por\_influencia**.**iteritems()] |
|  |
| *# O(|V|\*log(|V|))* |
| influencias\_ordenadas**.**sort(reverse**=**True) |
|  |
| *# O(|V|)* |
| **for** x **in** influencias\_ordenadas: |
| **print** x |
|  |
| **def** **mostrar\_recomendaciones**(self, recomendaciones): |
| """ |
| O(n\*log(n)) |
| """ |
|  |
| *# O(n\*log(n))* |
| recomendaciones**.**sort(key**=lambda** x:x[2], reverse**=**True) |
|  |
| *# O(n)* |
| \_recomendaciones **=** [ ( |
| self**.**grafo**.**get\_node\_data(u)**.**description, |
| self**.**grafo**.**get\_node\_data(v)**.**description, |
| amigos\_comun ) **for** u, v, amigos\_comun **in** |
| recomendaciones ] |
|  |
| *# O(n)* |
| **for** persona, recomendacion, amigos\_comun **in** \_recomendaciones: |
| **print** '%s: %s (%s amigo(s) en común)' **%** ( |
| persona, recomendacion, amigos\_comun) |
|  |
|  |
| **def** **get\_influencias**(self): |
| """ |
| O((|V|\*\*2)\*(|V|+|E|)) |
| Se obtiene el índice de influencia por cada vertice. |
| """ |
| *# O(n)* |
| influencias **=** [0 **for** w **in** self**.**grafo**.**iternodes()] |
|  |
|  |
| *# Preprocesamiento de cantidad de caminos mínimos* |
| *# O((|V|\*\*2)\*(|V|+|E|))* |
| **for** u **in** self**.**grafo**.**iternodes(): *# |V|* |
| **for** v **in** self**.**grafo**.**iternodes(): *# |V|* |
| cantidad\_u\_v **=** self**.**grafo**.**get\_cantidad\_caminos\_minimos(u,v) *# O(|V|+|E|)* |
|  |
| *# O(|V|\*\*3)* |
| **for** u **in** self**.**grafo**.**iternodes(): *# |V|* |
| **for** v **in** self**.**grafo**.**iternodes(): *# |V|* |
| **if** u**>=**v: |
| **continue** |
| *# O(1) c/preprocesamiento* |
| cantidad\_u\_v **=** self**.**grafo**.**get\_cantidad\_caminos\_minimos(u,v) |
| **if** cantidad\_u\_v **==** 0: |
| **continue** |
| **for** w **in** self**.**grafo**.**iternodes(): *# |V|* |
| **if** w**==**u **or** w**==**v: |
| **continue** |
| influencias[w] **+=** float( |
| *# O(1) (c/preprocesamiento)* |
| self**.**grafo**.**get\_cantidad\_caminos\_minimos\_con\_intermediario(u,w,v) |
| ) **/** cantidad\_u\_v |
| **return** influencias |
|  |
| **def** **calcular\_caminos\_minimos**(self): |
| """ |
| O(|V|\*|E|\*log(|V|)) |
| """ |
| *# |V|* |
| **for** i **in** self**.**grafo**.**iternodes(): |
| *# O(|E|\*log(|V|))* |
| self**.**grafo**.**calcular\_camino\_minimo(i) |
|  |
| **def** **get\_vertice\_from\_id**(self, id): |
| """ |
| O(|V|) |
| """ |
| **return** self**.**vertice\_from\_id[id] |
|  |
| **def** **recomendaciones\_para**(self, u): |
| """ |
| Au: Cantidad de aristas que salen de u. |
| O(Sum(v in V,Au+Av)) = O(Au\*|V|+|A|) |
| Devuelve un heap con las recomendaciones. Solo se recomienda en |
| caso que exista algún amigo en común. |
| """ |
| *# O(Au)* |
| recomendaciones **=** [] |
| **for** v **in** self**.**grafo**.**iternodes(): |
| **if** u**==**v: |
| **continue** |
| *# O(log(Au))* |
| **if** self**.**grafo**.**conectados(u,v): |
| **continue** |
| *# O(Au+Av)* |
| cantidad\_conexiones\_en\_comun **=** **-** len( |
| self**.**grafo**.**conexiones\_en\_comun(u,v)) |
| **if** cantidad\_conexiones\_en\_comun **==** 0: |
| **continue** |
| *# O(log(|V|))* |
| heappush(recomendaciones, (cantidad\_conexiones\_en\_comun, v)) |
| **return** recomendaciones |
|  |
| **def** **recomendaciones**(self): |
| """ |
| O(Sum(u in V,Au\*|V|+|A|)) = O(|V|\*Sum(u in V,Au)+|V|\*|A|) = |
| = O(|V|\*|A|+|V|\*|A|) = O(|V|\*|A|) |
| Devuelve un listado donde cada item tiene: |
| (vertice, recomendacion, amigos\_en\_comun) |
| """ |
| recomendaciones **=** [] |
| **for** u **in** self**.**grafo**.**iternodes(): |
| recomendaciones\_u **=** self**.**recomendaciones\_para(u) *# O(Au\*|V|+|A|)* |
| max\_amigos\_comun **=** None |
| **while** len(recomendaciones\_u) **>** 0: |
| *# O(1)* |
| amigos\_comun, recomendacion **=** heappop(recomendaciones\_u) |
| **if** max\_amigos\_comun **is** **not** None **and** amigos\_comun **>** max\_amigos\_comun: |
| **break** |
| recomendaciones**.**append( |
| (u, recomendacion, **-**amigos\_comun)) |
| max\_amigos\_comun **=** amigos\_comun |
| **return** recomendaciones |
|  |
|  |
|  |
|  |
| **def** **inicio\_seccion**(nombre): |
| """ |
| O(1) |
| """ |
| **print** '-----------------%s-----------------' **%** nombre |
|  |
| **def** **fin\_seccion**(): |
| """ |
| O(1) |
| """ |
| **print** '-------------------------------------------' |
| **print** |
|  |
| **def** **reporte\_amigos\_facebook\_gdf**(filepath): |
| """ |
| O((|V|\*\*2)\*(|V|+|A|)) |
| """ |
|  |
| tp1 **=** TP1(filepath) *# O(max(|E|,|V|)\*|V|)* |
|  |
| tp1**.**calcular\_caminos\_minimos() *# O(|V|\*|E|\*log(|V|))* |
|  |
| inicio\_seccion('Archivo %s' **%** filepath) |
|  |
| inicio\_seccion('popularidad') |
| popularidad **=** tp1**.**get\_popularidad() *# O(|V|)* |
| tp1**.**mostrar\_popularidad(popularidad) *# O(|V|)* |
| fin\_seccion() |
|  |
| inicio\_seccion('influencias') |
| influencias **=** tp1**.**get\_influencias() *# O((|V|\*\*2)\*(|V|+|A|))* |
| tp1**.**mostrar\_influencias(influencias) *# O(|V|\*\*2)* |
| fin\_seccion() |
|  |
| inicio\_seccion('recomendaciones') |
| recomendaciones **=** tp1**.**recomendaciones() *# O(|V|\*|A|)* |
| tp1**.**mostrar\_recomendaciones(recomendaciones) *# O(n\*log(n))* |
| fin\_seccion() |
|  |
| fin\_seccion() |
|  |
| **def** **reporte\_amigos\_facebook**(): |
| **for** filepath **in** sys**.**argv[1:]: |
| reporte\_amigos\_facebook\_gdf(filepath) *# O((|V|\*\*2)\*(|V|+|A|))* |
|  |
|  |
| **if** \_\_name\_\_ **==** '\_\_main\_\_': |
| **if** len(sys**.**argv) **==** 1: |
| unittest**.**main() |
| **else**: |
| reporte\_amigos\_facebook() |

grafo.py

|  |
| --- |
| *#!/usr/bin/python* |
| *# coding=utf-8* |
| **import** unittest |
| **from** heapq **import** heappop, heappush |
| **from** lista\_ordenada **import** ListaOrdenada |
|  |
| **class** **CaminoInexistente**(**Exception**): |
|  |
| **def** **\_\_init\_\_**(self, u, v, **\***args, **\*\***kwargs): |
|  |
| super(CaminoInexistente, self)**.**\_\_init\_\_( |
| 'Camino inexistente de %s a %s.' **%** (u, v), |
| **\***args, **\*\***kwargs) |
|  |
| **class** **Grafo**: |
|  |
| **def** **\_\_init\_\_**(self, cantidad\_vertices**=**0, pesos**=**[]): |
| """ |
| O(1) si los valores de los parámetros son los definidos |
| por defecto. |
| O(|E|\*log(|V|)) |
| """ |
| self**.**cantidad\_aristas **=** 0 |
| self**.**cantidad\_vertices **=** 0 |
| self**.**node\_data **=** [] |
| self**.**pesos **=** [] |
| self**.**cantidad\_caminos\_minimos **=** [] |
| self**.**recorridos **=** [] |
| self**.**lista\_ady **=** [] |
| **for** i **in** xrange(cantidad\_vertices): |
| self**.**add\_node() |
| **for** peso **in** pesos: |
| **if** peso[2] **<** 0: |
| **raise** **Exception**('Una arista no puede tener peso negativo.') |
| *# O(log(|V|))* |
| self**.**connect(peso[0], peso[1], peso[2]) |
| self**.**distancia **=** {} |
| self**.**padre **=** {} |
|  |
| **def** **get\_grado\_salida**(self, u): |
| """ |
| O(1) |
| Obtiene el grado de salida de u. |
| """ |
| **return** len(self**.**pesos[u]) |
|  |
| **def** **add\_node**(self, node\_data**=**None): |
| """ |
| Complejidad: O(1) |
| """ |
| self**.**cantidad\_vertices **+=** 1 |
| self**.**pesos**.**append({}) |
| self**.**node\_data**.**append(node\_data) |
| self**.**lista\_ady**.**append(ListaOrdenada()) |
| self**.**cantidad\_caminos\_minimos**.**append( |
| [0 **for** i **in** xrange(self**.**cantidad\_vertices **-** 1)]) |
| self**.**recorridos**.**append( |
| [None **for** i **in** xrange(self**.**cantidad\_vertices **-** 1)]) |
| **for** i **in** self**.**iternodes(): |
| self**.**cantidad\_caminos\_minimos[i]**.**append(0) |
| self**.**recorridos[i]**.**append(None) |
|  |
| **return** self**.**cantidad\_vertices **–** 1 |
|  |
| **def** **get\_node\_data**(self, u): |
| """ |
| Complejidad: O(1) |
| """ |
| **return** self**.**node\_data[u] |
|  |
| **def** **connect**(self, u, v, peso**=**1, both**=**False): |
| """ |
| O(log(Au)) si no se conecta a ambos. |
| O(max(log(Au),log(Av))) si se conecta a ambos. |
| """ |
| self**.**pesos[u][v] **=** peso |
| *# O(log(|V|))* |
| self**.**lista\_ady[u]**.**insert(v) |
| **if** both: |
| self**.**pesos[v][u] **=** peso |
| *# O(log(|V|))* |
| self**.**lista\_ady[v]**.**insert(u) |
| self**.**cantidad\_aristas **+=** 1 |
|  |
| **def** **ady**(self, v): |
| **return** self**.**pesos[v] |
|  |
| **def** **calcular\_camino\_minimo**(self, vertice): |
| """ |
| O(|E|\*log(|V|)) |
| """ |
| distancia **=** [None] **\*** self**.**cantidad\_vertices |
| padre **=** [set() **for** i **in** self**.**iternodes()] |
| visitado **=** [False] **\*** self**.**cantidad\_vertices |
| distancia[vertice] **=** 0 |
| heap **=** [] |
| heappush(heap, (distancia[vertice], vertice)) |
| **while** heap: |
| (\_, v) **=** heappop(heap) |
| visitado[v] **=** True |
| **for** w, peso **in** self**.**ady(v)**.**iteritems(): |
| **if** visitado[w]: |
| **continue** |
| **if** ( distancia[w] **is** None **or** |
| distancia[v] **+** peso **<** distancia[w] ): |
| distancia[w] **=** distancia[v] **+** peso |
| heappush(heap, |
| (distancia[w], w)) |
| padre[w] **=**set([ v ]) |
| **elif** (distancia[v] **+** peso **==** distancia[w]): |
| heappush(heap, |
| (distancia[w], w)) |
| padre[w]**.**add(v) |
|  |
| self**.**distancia[vertice] **=** distancia |
| self**.**padre[vertice] **=** padre |
|  |
| **def** **verificar\_existe\_camino**(self, u, v): |
| """ |
| O(1) |
| """ |
| self**.**get\_distancia(u,v) |
|  |
| **def** **get\_recorrido\_anchura\_caminos\_minimos**(self, u, v): |
| """ |
| Obtiene el recorrido en anchura por caminos mínimos |
| desde u hasta v. |
| O(|E|+|V|) |
| Está claro que por cada vertice que no hayamos visitado |
| intentamos acceder a sus padres para el camino minimo |
| buscado, con lo cual iteraremos como máximo sobre todos |
| los vertices y sobre todas las aristas. |
| """ |
| *# Se verifica que exista un camino de u a v* |
| self**.**verificar\_existe\_camino(u,v) |
|  |
| visitado **=** [False **for** i **in** self**.**iternodes()] |
| q **=** [v] |
| recorrido **=** [] |
| **while** len(q)**>**0: |
| w **=** q**.**pop(0) |
| **if** visitado[w]: |
| **continue** |
| visitado[w] **=** True |
| recorrido**.**insert(0,w) |
| **for** padre **in** self**.**padre[u][w]: |
| q**.**append(padre) |
| **return** recorrido |
|  |
| **def** **get\_cantidad\_caminos\_minimos**(self, u, v, intentar\_al\_reves**=**True): |
| """ |
| O(1) mejor caso si hubo preprocesamiento. |
| O(|V|+|E|) en caso que no haya habido preprocesamiento. |
| Se obtiene la cantidad de recorridos de u a v. |
| Previamente se debe haber llamado a calcular\_camino\_minimo(u) |
| o calcular\_camino\_minimo(v). |
| """ |
| *# O(|V|+|E|)* |
| **try**: |
| recorrido **=** self**.**get\_recorrido\_anchura\_caminos\_minimos(u,v) |
| **except** CaminoInexistente: |
| **return** 0 |
|  |
| *# O(|V|+|E|): Idem explicación get\_recorrido\_anchura\_caminos\_minimos* |
| **for** w **in** recorrido: |
| **if** self**.**cantidad\_caminos\_minimos[u][w] **<>** 0: |
| **continue** |
| **if** w**==**u: |
| self**.**cantidad\_caminos\_minimos[u][w] **=** 1 |
| **else**: |
| **for** padre **in** self**.**padre[u][w]: |
| self**.**cantidad\_caminos\_minimos[u][w] **+=** ( |
| self**.**cantidad\_caminos\_minimos[u][padre] ) |
| **return** self**.**cantidad\_caminos\_minimos[u][v] |
|  |
| **def** **get\_cantidad\_caminos\_minimos\_con\_intermediario**(self, u, w, v): |
| """ |
| O(1) mejor caso si hubo preprocesamiento. |
| O(|V|+|E|) en caso que no haya habido preprocesamiento. |
| Se obtiene la cantidad de caminos mínimos entre u y v que |
| pasan por w. |
| """ |
| **try**: |
| **if** (self**.**get\_distancia(u,w) **+** self**.**get\_distancia(w,v) |
| ) **>** self**.**get\_distancia(u,v): |
| **return** 0 |
| **return** ( self**.**get\_cantidad\_caminos\_minimos(u,w) **\*** |
| self**.**get\_cantidad\_caminos\_minimos(w,v)) |
| **except** CaminoInexistente: |
| **return** 0 |
|  |
| **def** **get\_distancia**(self, u, v, intentar\_al\_reves**=**True): |
| """ |
| Se obtiene la distancia. |
| Previamente se debe haber llamado a calcular\_camino\_minimo(u) |
| o calcular\_camino\_minimo(v). |
| """ |
| **if** u **in** self**.**distancia: |
| distancia **=** self**.**distancia[u][v] |
| **if** distancia **is** None: |
| **raise** CaminoInexistente(u,v) |
| **return** distancia |
| **if** intentar\_al\_reves: |
| **return** self**.**get\_distancia(v, u, intentar\_al\_reves**=**False) |
| **raise** **Exception**('Debe calcular previamente el camino mínimo.') |
|  |
| **def** **get\_recorrido**(self, u, v, intentar\_al\_reves**=**True): |
| """ |
| Se obtiene uno de los posibles recorridos. |
| Previamente se debe haber llamado a calcular\_camino\_minimo(u) |
| o calcular\_camino\_minimo(v). |
| """ |
| **if** u **in** self**.**padre: |
| w **=** [v] |
| recorrido **=** [] |
| **while** len(w) **>** 0: |
| recorrido**.**insert(0, iter(w)**.**next()) |
| w **=** self**.**padre[u][iter(w)**.**next()] |
| **return** recorrido |
| **if** intentar\_al\_reves: |
| **return** reversed( |
| self**.**get\_recorrido(v, u, intentar\_al\_reves**=**False)) |
| **raise** **Exception**('Debe calcular previamente el camino mínimo.') |
|  |
| **def** **get\_recorridos**(self, u, v): |
| **if** u**==**v: |
| **return** [[u]] |
| **if** u **in** self**.**padre: |
| **if** self**.**recorridos[u][v] **is** None: |
| self**.**recorridos[u][v] **=** [] |
| **for** w **in** self**.**padre[u][v]: |
| **for** recorrido **in** self**.**get\_recorridos(u,w): |
| self**.**recorridos[u][v]**.**append(recorrido**+**[v]) |
| **return** self**.**recorridos[u][v] |
| **raise** **Exception**('Debe calcular previamente el camino mínimo.') |
|  |
| **def** **conectados**(self, u, v): |
| """ |
| O(log(|V|)) |
| Verifica si u y v están conectados. |
| """ |
| *# O(log(n))* |
| **return** self**.**lista\_ady[u]**.**has(v) |
|  |
| **def** **conexiones\_en\_comun**(self, u, v): |
| """ |
| Au: Cantidad de aristas que salen de u. |
| Av: Cantidad de aristas que salen de v. |
| O(Au+Av) |
| """ |
| **return** self**.**lista\_ady[u]**.**intersection( |
| self**.**lista\_ady[v]) |
|  |
| **def** **iternodes**(self): |
| **return** xrange(self**.**cantidad\_vertices) |
|  |
|  |
|  |
|  |
|  |

lista\_ordenada.py

|  |
| --- |
| *#!/usr/bin/python* |
| *# coding=utf-8* |
|  |
| **import** bisect |
|  |
| **class** **ListaOrdenada**(): |
|  |
| **def** **\_\_init\_\_**(self): |
| self**.**lista **=** [] |
|  |
| **def** **iteritems**(self): |
| **return** iter(self**.**lista) |
|  |
| **def** **insert**(self, node): |
| """ |
| O(n\*log(n)) |
| """ |
| i **=** bisect**.**bisect\_left(self**.**lista, node) |
| **if** i **<>** len(self**.**lista) **and** self**.**lista[i] **==** node: |
| **raise** **Exception**('El nodo %s ya existe en la lista.' **%** node) |
| **return** self**.**lista**.**insert(i, node) |
|  |
| **def** **has**(self, node): |
| """ |
| O(n\*log(n)) |
| """ |
| i **=** bisect**.**bisect\_left(self**.**lista, node) |
| **if** i **<>** len(self**.**lista) **and** self**.**lista[i] **==** node: |
| **return** True |
| **return** False |
|  |
| **def** **intersection**(self, other): |
| """ |
| O(len(self.lista)+len(other.lista)) = O(n1+n2) |
| """ |
| len\_self **=** len(self**.**lista) |
| len\_other **=** len(other**.**lista) |
| i\_self **=** 0 |
| i\_other **=** 0 |
| intersection **=** [] |
|  |
| **while** i\_self **<** len\_self **and** i\_other **<** len\_other: |
| x\_self **=** self**.**lista[i\_self] |
| x\_other **=** other**.**lista[i\_other] |
|  |
| **if** x\_self **<** x\_other: |
| i\_self **+=** 1 |
| **elif** x\_self **>** x\_other: |
| i\_other **+=** 1 |
| **else**: |
| intersection**.**append(x\_self) |
| i\_self **+=** 1 |
| i\_other **+=** 1 |
|  |
| **return** intersection |