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: 30-OCT-2014

Correcciones:

* Remplazar Dijkstra por DFS en cálculo de caminos mínimos
* Indicar como se muestra ordenada la popularidad
* Mejorar orden en el cálculo de influencias

Contenido

Contenido 2

Análisis y Diseño 3

Popularidad 3

Influencias 3

Obtención de cantidad de caminos mínimos entre dos vértices 3

Cálculo de influencias 3

Recomendaciones 4

Diagramas 4

Módulos 4

Clases 4

Justificación de órdenes del lenguaje y librerías 4

Código Fuente 5

tp1.py 5

grafo/grafo.py 9

grafo/grafo\_no\_pesado.py 13

grafo/lista\_ordenada.py 15

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|).

Para ordenar los vértices por popularidad, teniendo en cuenta que la popularidad de cada vértice va de 0 a |V|-1, entonces simplemente se utiliza un array con |V| posiciones (de 0 a |V|-1) y por cada posición k se tiene un lista en la que se agrega cada vértice que tiene popularidad k, con lo cual el costo de mantener ordenada la popularidad es O(1) y de mostrar la popularidad de los vértices en forma ordenada es O(|V|).

Influencias

La obtención del índice de influencia para cualquier vértice 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. Realizar en forma eficiente la obtención de la influencia por cada vertice en función de la cantidad de cáminos mínimos ya calculada.

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 DFS para cálculo de caminos mínimos en grafos no pesados, de forma que se acepte más de un camino mínimo.. 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(|V|+|E|), que es básicamente el orden de cualquier DFS.

Para obtener la cantidad de caminos mínimos, dentro del mismo DFS aplicado para un vertice s, lo que hacemos es, para cualquier t distinto de s, si encontramos un camino mínimo menor al ya existente inicializamos la cantidad en 1, pero en caso de ser igual, simplemente incrementamos dicha cantidad.

Cálculo de influencias

La forma ingenua de realizar este cálculo es realizar cada una de las sumas determinada por la formula definida en el enunciado, lo que da un orden O(|V|\*\*3). Sin embargo existe una forma mucho más eficiente de realizar esta operación en un orden O(|V|\*(|V|+|E|)). Una explicación detallada de este algoritmo se puede obtener desde el siguiente link:

<http://www.inf.uni-konstanz.de/algo/publications/b-fabc-01.pdf>

En esta publicación Ulrik Brandes explica cómo hacer para disminuir el orden utilizando el concepto de dependencia entre un vertice hacia otro y encuentra un relación recursiva para obtener su valor de los padres respecto de sus hijos.

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.

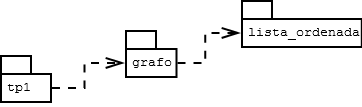
Para verificar si dos vértices u y v están conectados, en caso de utilizar listas de adyacencia ordenadas que soporten acceso por índice, simplemente basta con realizar una búsqueda binaria. Dicha operación será de orden O(log(|Au|)) donde Au es el conjunto de aristas que salen de u.

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|).

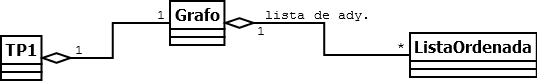
Luego, obtener las recomendaciones para el vértice u tendrá orden O(Au\*|V|+|A|), método <#recomendaciones_para>, y para obtener recomendaciones para todos los vértices el orden ascenderá a O(|V|\*|A|), método <#recomendaciones>.

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.

Si desea visualizar los cambios aplicados entre la primer entrega y la segunda entrega puede utilizar el comando **git diff entrega1 entrega**. O si lo desea puede visualizarlo en la siguiente página: https://github.com/facultad/7529-tp1/compare/entrega1...entrega2

tp1.py

|  |
| --- |
| *#!/usr/bin/python* |
| *# coding=utf-8* |
| **from** grafo **import** GrafoPesoUnitario |
| **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, clase\_grafo**=**GrafoPesoUnitario): |
| """ |
| O(max(|E|,|V|)\*|V|) |
| """ |
| self.grafo **=** clase\_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|\*(|V|+|E|)) |
| Se obtiene el índice de influencia por cada vertice. |
| """ |
| self.grafo.calcular\_influencias() |
| **return** [ self.grafo.get\_influencia(u) **for** u **in** self.grafo.iternodes() ] |
|  |
| **def** **calcular\_caminos\_minimos**(self): |
| """ |
| O(|V|\*(|E|+|V|) |
| """ |
| self.grafo.calcular\_caminos\_minimos() |
|  |
| **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 |

grafo/grafo.py

|  |
| --- |
| *#!/usr/bin/python* |
| *# coding=utf-8* |
| **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): |
| """ |
| O(1) |
| """ |
| self.cantidad\_aristas **=** 0 |
| self.cantidad\_vertices **=** 0 |
| self.node\_data **=** [] |
| self.pesos **=** [] |
| self.cantidad\_caminos\_minimos **=** [] |
| self.recorridos **=** [] |
| self.lista\_ady **=** [] |
| self.influencias **=** [] |
| 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**) |
| self.influencias.append(0) |
|  |
| **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** **verificar\_existe\_camino**(self, u, v): |
| """ |
| O(1) |
| """ |
| self.get\_distancia(u,v) |
|  |
| **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) |
|  |
| **def** **get\_influencia**(self, u): |
| """ |
| O(1) |
| """ |
| **return** self.influencias[u] |
|  |
| **def** **calcular\_caminos\_minimos**(self): |
| """ |
| O(|V|\*(|E|+|V|) |
| """ |
| *# |V|* |
| **for** i **in** self.iternodes(): |
| *# O(|V|+|E|)* |
| self.calcular\_camino\_minimo(i) |
|  |
| **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). |
| """ |
| **if** self.cantidad\_caminos\_minimos[u][v] <> 0: |
| **return** self.cantidad\_caminos\_minimos[u][v] |
|  |
| **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 |

grafo/grafo\_no\_pesado.py

|  |
| --- |
| *#!/usr/bin/python* |
| *# coding=utf-8* |
|  |
| **from** grafo **import** Grafo |
|  |
| **class** **GrafoPesoUnitario**(Grafo): |
|  |
| **def** **bfs**(self, u): |
| """ |
| O(|V|+|E|) |
| Devuelve un listado comenzando poe la raiz (u) y terminando |
| por los vertices a mayor distancia de u. |
| Es requisito que ya se hayan calculado los caminos mínimos |
| para el vértice u |
| """ |
|  |
| s**=**[] |
|  |
| visitado **=** [**False**] **\*** self.cantidad\_vertices |
| q **=** [u] |
|  |
| **while** **len**(q)**>**0: |
| w **=** q.pop(0) |
|  |
| **if** visitado[w]: |
| **continue** |
|  |
| visitado[w] **=** **True** |
|  |
| s.append(w) |
|  |
| **for** a **in** self.ady(w): |
| **if** ( self.distancia[u][w] **+** 1 **==** self.distancia[u][a] ): |
| q.append(a) |
| **return** s |
|  |
| **def** **\_\_acumular\_influencias**(self, u): |
| *# O(|V|+|E|)* |
| S **=** self.bfs(u) |
|  |
| *# O(|E|)* |
| *# Se acumulan las influencias a partir del calculo de* |
| *# dependencias del vertice u hacia el resto.* |
| dependencias **=** [0 **for** i **in** self.iternodes()] |
| **while** **len**(S) **>** 0: |
| w **=** S.pop() |
| **for** v **in** self.padre[u][w]: |
| dependencias[v] **+=** (**float**(self.cantidad\_caminos\_minimos[u][v]) **/** |
| **float**(self.cantidad\_caminos\_minimos[u][w])) **\*** ( |
| 1 **+** dependencias[w]) |
| **if** w <> u: |
| self.influencias[w] **+=** dependencias[w] |
|  |
| **def** **calcular\_influencias**(self): |
| """ |
| # O(|V|\*(|V|+|E|)) |
| """ |
| **for** u **in** self.iternodes(): *#|V|* |
| self.\_\_acumular\_influencias(u) *# O(|V|+|E|)* |
|  |
| **def** **calcular\_camino\_minimo**(self, u): |
| """ |
| Calcula las distancias de u al resto de los vertices. |
| O(|E|+|V|) |
| """ |
|  |
| distancia **=** [**None**] **\*** self.cantidad\_vertices |
| padre **=** [**set**() **for** i **in** self.iternodes()] |
| visitado **=** [**False**] **\*** self.cantidad\_vertices |
| q **=** [u] |
| distancia[u] **=** 0 |
| cantidad\_caminos\_minimos **=** [0] **\*** self.cantidad\_vertices |
| cantidad\_caminos\_minimos[u] **=** 1 |
|  |
| **while** **len**(q)**>**0: |
| w **=** q.pop(0) |
|  |
| **if** visitado[w]: |
| **continue** |
|  |
| visitado[w] **=** **True** |
|  |
| **for** adyacente **in** self.ady(w): |
|  |
| **if** ( distancia[adyacente] **is** **None** **or** |
| distancia[w] **+** 1 **<** distancia[adyacente] ): |
| q.append(adyacente) |
| distancia[adyacente] **=** distancia[w] **+** 1 |
| padre[adyacente] **=** **set**([ w ]) |
| cantidad\_caminos\_minimos[adyacente] **=** cantidad\_caminos\_minimos[w] |
| **elif** (distancia[w] **+** 1 **==** distancia[adyacente]): |
| padre[adyacente].add(w) |
| cantidad\_caminos\_minimos[adyacente] **+=** cantidad\_caminos\_minimos[w] |
|  |
| self.distancia[u] **=** distancia |
| self.padre[u] **=** padre |
| self.cantidad\_caminos\_minimos[u] **=** cantidad\_caminos\_minimos |
|  |
| **def** **get\_cantidad\_caminos\_minimos**(self, u, v, intentar\_al\_reves**=True**): |
| """ |
| O(1) |
| """ |
| **return** self.cantidad\_caminos\_minimos[u][v] |

grafo/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 |