Facultad de Ingeniería de la UBA



75.29 Teoría de Algoritmos

Trabajo Práctico N°1

**Cuatrimestre**: Segundo de 2014

**Fecha de entrega**: 20 de Octubre de 2014

**Fecha de re-entrega**: 7 de Noviembre de 2014

**Alumnos**:

|  |  |  |
| --- | --- | --- |
| **Apellido y Nombre** | **Padrón** | **Email** |
| Parnisari, María Inés | 92235 | [maineparnisari@gmail.com](mailto:maineparnisari@gmail.com) |
| Buffevant, César | 76700 | [buffevant@gmail.com](mailto:buffevant@gmail.com) |

Indice

Contents

[Metodología de Trabajo 4](#_Toc405152702)

[Análisis y Diseño 4](#_Toc405152703)

[Estructuras de datos 4](#_Toc405152704)

[Nodos y Grafos 4](#_Toc405152705)

[Algoritmos auxiliares 6](#_Toc405152706)

[Lectura del archivo GDF 6](#_Toc405152707)

[Ordenamiento 6](#_Toc405152708)

[Listado por popularidad 6](#_Toc405152709)

[Influencias 7](#_Toc405152710)

[Recomendaciones de amigos 8](#_Toc405152711)

[Diagrama de Clases 9](#_Toc405152712)

[Calculo de Órdenes 10](#_Toc405152713)

[Lectura del archivo 10](#_Toc405152714)

[Listado por Popularidad 10](#_Toc405152715)

[Influencias 10](#_Toc405152716)

[Recomendaciones de amigos 10](#_Toc405152717)

[Contenido de la entrega 10](#_Toc405152718)

[Código Fuente 11](#_Toc405152719)

[IIdentifiable.cs 11](#_Toc405152720)

[Node.cs 11](#_Toc405152721)

[Graph.cs 14](#_Toc405152722)

[BaseGraphReader.cs 16](#_Toc405152723)

[GraphReader.cs 17](#_Toc405152724)

[GraphBuilder.cs 18](#_Toc405152725)

[Influences.cs 19](#_Toc405152726)

[InfluencesCollection.cs 20](#_Toc405152727)

[InfluencesExtension.cs 21](#_Toc405152728)

[Recommendation.cs 23](#_Toc405152729)

[RecommendationsCollection.cs 23](#_Toc405152730)

[RecommendationsExtension.cs 26](#_Toc405152731)

[FacebookGraph.cs 29](#_Toc405152732)

[FacebookUser.cs 29](#_Toc405152733)

[Logger.cs 31](#_Toc405152734)

[Main.cs 32](#_Toc405152735)

# Metodología de Trabajo

El presente trabajo se realizó en el lenguaje de programación C#, propietario de Microsoft. El mismo puede ejecutarse bajo Windows mediante Visual Studio, o en un entorno Linux con el framework de código abierto [Mono](http://www.mono-project.com/).

Para el desarrollo implementamos tests unitarios y de integración automatizados utilizando el framework [NUnit](http://www.nunit.org/), que nos permitió alcanzar una cobertura de código de aproximadamente 88%.

Para el versionado del código de fuente utilizamos un [repositorio GitHub](https://github.com/miparnisari/teoriadealgoritmos) privado.

# Análisis y Diseño

## Estructuras de datos

Las estructuras de datos provistas por el lenguaje que utilizamos son las siguientes:

* List<T>: <http://msdn.microsoft.com/es-AR/library/6sh2ey19(v=vs.110).aspx>
* Dictionary<T>: [http://msdn.microsoft.com/es-AR/library/xfhwa508(v=vs.110).aspx](http://msdn.microsoft.com/en-us/library/xfhwa508(v=vs.110).aspx)
* HashSet<T>: [http://msdn.microsoft.com/es-AR/library/bb359438(v=vs.110).aspx](http://msdn.microsoft.com/en-us/library/bb359438(v=vs.110).aspx)

Para evitar tener que modificar la estructura interna de cada una ante un exceso de objetos almacenados, asumimos una cantidad inicial de 1000 de nodos adyacentes para un Nodo, y de 1000 nodos totales para un Grafo.

A continuación describimos las estructuras de datos que implementamos nosotros.

### Nodos y Grafos

Desarrollamos una estructura de datos genérica para representar el grafo, independiente del trabajo práctico en particular. El único supuesto que tenemos es que la información a ser almacenada en los nodos pueda ser identificada mediante un identificador.

La información a guardar debe implementar la siguiente interface:

**public interface** IIdentifiable<TId> **where** TId : IComparable  
{  
 TId Id  
 {  
 **get**;  
 }  
}

Los nodos son representados por la clase Node<TData, TId>. El primer parámetro representa la familia de tipos utilizados para la data, y el segundo parámetro representa la familia de tipos utilizadas para el identificador:

**public class** Node<TData, TId>   
 **where** TData : IIdentifiable<TId>   
 **where** TId: IComparable

Cada nodo guarda la lista de nodos adyacentes al mismo en un diccionario indexado por su Id, por lo que el acceso a cualquier nodo adyacente por su Id se hace en O(1).[[1]](#footnote-1)

**private readonly** IDictionary<TId, Node<TData, TId>> adjacentNodes;  
  
De ésta forma, obtener el grado de un nodo es también O(1)[[2]](#footnote-2), simplemente es la cantidad de items en el diccionario:

**public int** Degree {  
 get {  
 **return** adjacentNodes.Values.Count;  
 }  
}

La clase utilizada para representar al grafo es Graph<TData, TId>:

**public class** Graph<TData, TId>   
 **where** TData : IIdentifiable<TId>  
 **where** TId : IComparable

Los nodos pertenecientes al grafo se mantienen en un diccionario indexado por sus Ids:

**private readonly** IDictionary<TId, Node<TData, TId>> nodes

Dado el Id de un nodo, éste puede ser recuperado en O(1) debido al uso del diccionario.

La información proveniente de Facebook es representada mediante la clase FacebookUser:

**public class** FacebookUser : IIdentifiable<long>

La única información que estamos guardando es el Id y el Nombre.

La implementación concreta del grafo para FacebookUser es simplemente la especialización de los parámetros de tipo:

**public class** FacebookGraph : Graph<FacebookUser, long>

## Algoritmos auxiliares

### Lectura del archivo GDF

Para la lectura del archivo y creación del grafo se implementaron las siguientes clases:

* BaseGraphReader: Contiene la lógica relacionada con el formato particular del archivo. Realiza la lectura a partir de un Stream de datos. Ésta clase utiliza dos *delegates[[3]](#footnote-3)* para manejar los siguientes eventos: nuevo nodo y nueva arista.
* GraphReader: Hereda la mayor parte de su funcionalidad de BaseGraphReader, pero la especializa para leer archivos desde el sistema de archivos.
* GraphBuilder: Ésta es la clase a ser utilizada por el código cliente. Recibe como parámetro un BaseGraphReader, el cual utilizará para la lectura del archivo. Le provee al reader los dos delegates los cuales contiene la lógica particular relacionada con la construcción del grafo.

La idea de tener estás tres clases separadas es primeramente reducir las responsabilidades de cada una. Los *readers* lidian con el formato del archivo, mientras que el GraphBuilder lidia con la construcción particular de nuestro grafo. Luego, nos brinda la posibilidad de tener una especialización para leer el contenido GDF desde un string, la cual fue utilizada para los tests automáticos.

### Ordenamiento

Implementamos HeapSort para ordenar los nodos del grafo. En la clase GraphSortExtensions está definido un método de extensión[[4]](#footnote-4) para Graph llamado Sort con la implementación de HeapSort. El método devuelve un enumerable de nodos, con la secuencia ordenada de nodos de acuerdo al comparador pasado como parámetro. El comparador es simplemente una función que recibe un par de nodos y devuelve un boolean, representando una relación de orden total.

El orden de HeapSort es O(nlog n).

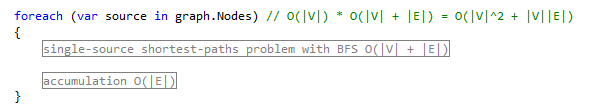
### Listado por popularidad

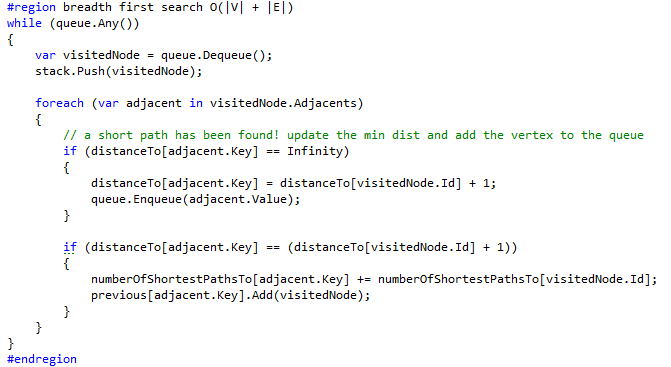
Para el listado de nodos de acuerdo a su popularidad, simplemente ordenamos los nodos del grafo mediante HeapSort y los listamos:

**foreach** (**var** node **in** graph.Sort((nodeA, nodeB) => nodeA.Degree >= nodeB.Degree))

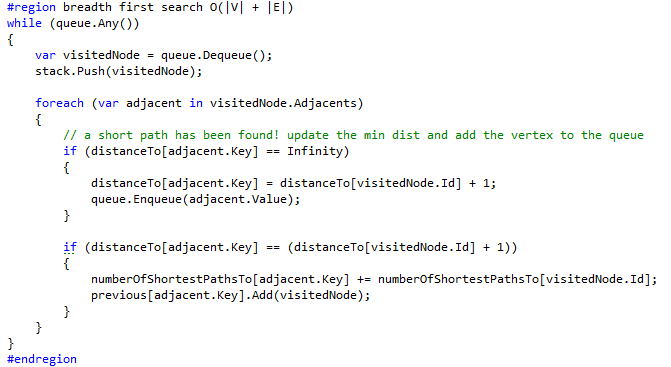
### Influencias

Calcular la influencia de todos los nodos se reduce a, para cada nodo, calcular todos los caminos mínimos hacia todos los demás nodos utilizando el algoritmo de Breadth First Search, y luego recorrer dichos caminos actualizando a cada paso la influencia de cada nodo.





Para calcular todos los caminos mínimos desde un nodo, se utiliza el algoritmo de BFS adaptado para que, en vez de devolver el primer camino mínimo encontrado, devuelva todos los que haya. Esto se logra almacenando el conjunto de nodos (previous) que se utilizaron para llegar a otro nodo, y luego se reali



### Recomendaciones de amigos

Para las recomendaciones se implementó en la clase RecommendationsExtension el extension method llamando GetRecommendations(). La idea detrás del algoritmo es la siguiente: llamemos L0 al nodo raíz del árbol generado por el BFS, L1 al primer nivel, L2 al segundo y así sucesivamente. Los nodos en L1 son los amigos directos del nodo en L0. Los nodos en L2 son los nodos amigos de los que están en L1 pero que no son amigos directos del nodo en L0. Por lo tanto, son las posibles recomendaciones. Ahora, los amigos en común son necesariamente los que están en L1. Como BFS genera el árbol por niveles, a medida que se recorren los nodos de L1 se incrementan un contador de amigos en común para los nodos en L2. De ésta forma, el BFS solo necesita completar el segundo nivel del árbol. Éste proceso se realiza para todos los nodos del grafo.

Más allá de las estructuras necesarias para implementar BFS, se realizaron las siguientes modificaciones:

* La cola en vez de almacenar los nodos almacena una par (Nodo, int), donde el entero representa el nivel del árbol donde está ese nodo.

**var** queue = **new** Queue<Tuple<**int**, Node<TData,TId>>>(nodesCount);

* Se mantiene un diccionario de amigos en común indexado por el Id del nodo.

**var** friendsInCommon = **new** Dictionary<TId, **int**>(nodesCount);

Al encolar los nodos adyacentes se verifica si estoy en un nodo en L1. Si es el caso, para todos los adyacentes que estén el L2 se aumenta la cantidad de amigos en común:

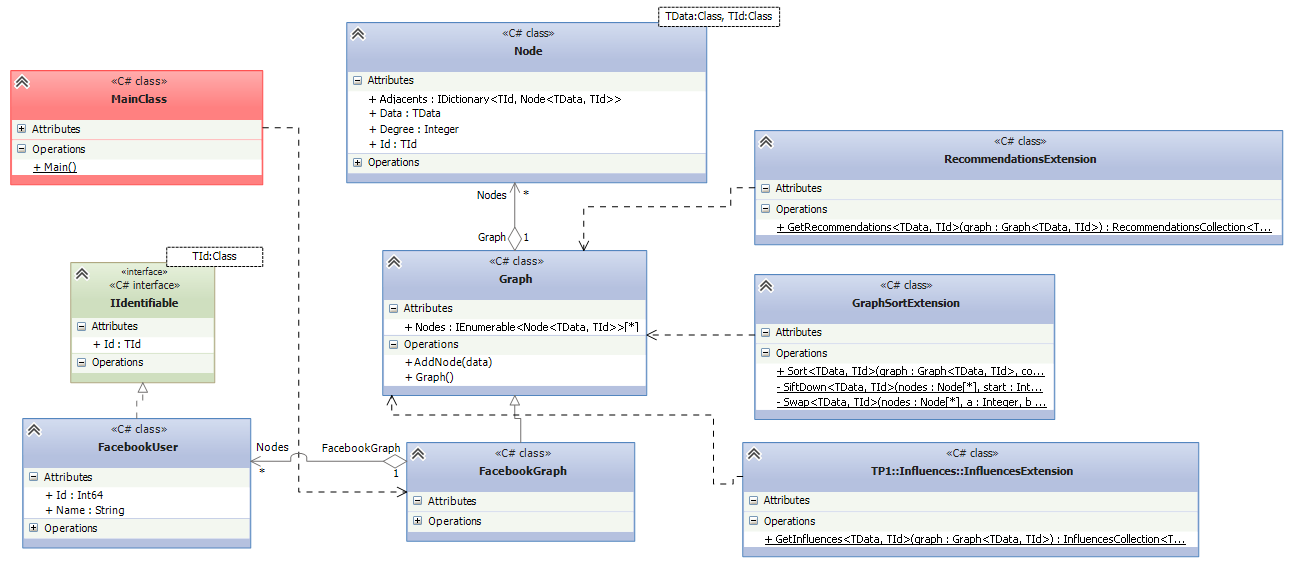
if (!friendsInCommon.ContainsKey(adj.Value.Id)) // O(1)  
 friendsInCommon.Add(adj.Value.Id, 0); // O(1)  
friendsInCommon[adj.Value.Id] += 1; // O(1)

Luego, se actualiza la recomendación

if (friendsInCommon [adj.Value.Id] > recomendation.FriendCount)   
{

recomendation.FriendCount = friendsInCommon [adj.Value.Id];  
 recomendation.PersonToRecommend = adj.Value;  
}

# Diagrama de Clases



# Calculo de Órdenes

## Lectura del archivo

Debido a la estructura del archivo (lista primero los nodos y luego las aristas) la construcción del grafo se puede realizar con una sola pasada. Durante la lectura de los nodos, se construyen el diccionario a nivel de grafo (O(1)). Luego, durante la lectura de las aristas, se tienen que buscar los dos nodos, y asignar uno a la lista de adyacentes del otro, y viceversa. La búsqueda de los dos nodos es O(1) por la utilización de diccionarios, y la creacion de la arista agregandolos al diccionario de nodos adyacentes, también.

Por lo tanto, la construcción del grafo a partir del file GDF es O(k) donde k es la cantidad de líneas en el archivo. O en términos de parámetros del grafo, O(n+m) donde n es la cantidad de nodos y m la cantidad de aristas.

## Listado por Popularidad

Descontando la lectura de los nodos del archivo, la única operación que se realiza es el ordenamiento de los nodos mediante HeapSort. Por lo tanto toda la operación es O(n log n).

## Influencias

La implementación se basa en “[A Faster Algorithm for Betweenness Centrality](http://www.inf.uni-konstanz.de/algo/publications/b-fabc-01.pdf)” de Ulrik Brandes. En el paper, el teorema 8 establece que para grafos sin pesos el algoritmos es O(N\*A).

## Recomendaciones de amigos

El algoritmo es básicamente un BFS para cada nodo. Todas las demás operaciones son O(1).

Así que el orden total sería O(N \* (N + A)) = O(N2 + NA), donde N es la cantidad de nodos y A la cantidad de aristas.

De hecho ésta cota es un tanto pesimista ya que no todo el nodo es recorrido en el BFS, sino que al haber completado el nivel 2 el algoritmo se detiene. Por lo tanto, el orden O(N + A) funciona como una cota superior. Se mantiene esa cota a falta de parámetros más significativos en función de los cuales expresar el orden.

# Contenido de la entrega

* Binaries: Directorio con los binarios.
* SourceCode: Directorio con el código fuente.
  + ConsoleApplication1: aplicación que utiliza los tres algoritmos.
  + TP1: Class library con los algoritmos y las estructuras de datos.
  + TP1.Test: Test unitarios.
  + Makefiles
* Enunciado.pdf
* Instrucciones de uso.md
* Informe.pdf

# Código Fuente

## IIdentifiable.cs

using System;  
namespace TP1.Graph  
{  
 public interface IIdentifiable<TId> where TId : IComparable  
 {  
 TId Id  
 {  
 get;  
 }  
 }  
}

## Node.cs

using System;  
using System.Collections.Generic;  
  
namespace TP1.Graph  
{  
 /// <summary>  
 /// Representation of a Graph's node.  
 /// </summary>  
 public class Node<TData, TId>   
 where TData : IIdentifiable<TId>   
 where TId: IComparable  
 {  
 private const int InitialCapacity = 1000;  
 private readonly IDictionary<TId, Node<TData, TId>> adjacentNodes;  
  
 /// <remarks>O(1)  
 /// </remarks>  
 public Node(int initialCapacity = InitialCapacity)  
 {  
 this.adjacentNodes = new Dictionary<TId, Node<TData, TId>>(initialCapacity);  
 }  
  
 public Node(TData data)  
 : this()  
 {  
 Data = data;  
 }  
  
 public TData Data  
 {  
 get;  
 private set;  
 }  
  
 public TId Id  
 {  
 get  
 {  
 return this.Data.Id;  
 }  
 }  
  
 /// <remarks>O(1)  
 /// </remarks>  
 public void AddAdjacentNode(Node<TData, TId> node)  
 {  
 if (!adjacentNodes.ContainsKey(node.Id)) // O(1) - Source: http://msdn.microsoft.com/en-us/library/kw5aaea4(v=vs.110).aspx  
 {  
 adjacentNodes.Add(node.Id, node); // Best case O(1), worst case O(N) - source: http://msdn.microsoft.com/en-us/library/k7z0zy8k(v=vs.110).aspx  
 }  
 }  
  
 /// <remarks>O(1)  
 /// </remarks>  
 public IDictionary<TId, Node<TData, TId>> Adjacents  
 {  
 get  
 {  
 return this.adjacentNodes;  
 }  
 }  
  
 /// <summary>  
 /// Gets the node degree (i.e. number of edges incident to the vertex)  
 /// </summary>  
 /// <remarks>O(1)  
 /// </remarks>   
 public int Degree  
 {  
 get  
 {  
 return adjacentNodes.Values.Count; // O(1) - source: http://msdn.microsoft.com/en-us/library/6z6y38za(v=vs.110).aspx  
 }  
 }  
  
 public override bool Equals(object obj)  
 {  
 var otherNode = obj as Node<TData, TId>;  
 if (otherNode != null)  
 {  
 return this.Id.Equals(otherNode.Id);  
 }  
 return false;  
 }  
  
 public override int GetHashCode()  
 {  
 return this.Id.GetHashCode();  
 }  
  
 public override string ToString()  
 {  
 return this.Id.ToString();  
 }  
 }  
}

## 

## Graph.cs

using System;  
using System.Collections.Generic;  
  
namespace TP1.Graph  
{  
 /// <summary>  
 /// Representation of an undirected graph.   
 /// </summary>  
 public class Graph<TData, TId>  
 where TData : IIdentifiable<TId>  
 where TId : IComparable  
 {  
 private const int InitialCapacity = 1000;  
 private readonly IDictionary<TId, Node<TData, TId>> nodes;  
  
 /// <remarks>O(1)  
 /// Source: http://msdn.microsoft.com/en-us/library/tk84bxf4(v=vs.110).aspx  
 /// </remarks>  
 public Graph(int initialCapacity = InitialCapacity)  
 {  
 this.nodes = new Dictionary<TId, Node<TData, TId>>(initialCapacity);  
 }  
  
 /// <summary>  
 /// Gets the node with the specified Id.  
 /// </summary>  
 /// <remarks>O(1)  
 /// </remarks>  
 public Node<TData, TId> this[TId id]  
 {  
 get  
 {  
 return nodes.ContainsKey(id) ? nodes[id] : null; // O(1) - Source: http://msdn.microsoft.com/en-us/library/kw5aaea4(v=vs.110).aspx  
 }  
 }  
  
 /// <remarks>O(1)  
 /// </remarks>  
 public void AddNode(TData data)  
 {  
 if (!nodes.ContainsKey(data.Id))  
 {  
 var node = new Node<TData, TId>(data);  
 nodes.Add(node.Id, node);  
 }  
 }  
  
 /// <remarks>O(1)  
 /// </remarks>  
 public IEnumerable<Node<TData, TId>> Nodes  
 {  
 get  
 {  
 return this.nodes.Values; // O(1) - Source: http://msdn.microsoft.com/en-us/library/ekcfxy3x(v=vs.110).aspx  
 }  
 }  
  
 /// <remarks>O(1)  
 /// </remarks>  
 public int NodeCount  
 {  
 get  
 {  
 return this.nodes.Count; // O(1) - Source: http://msdn.microsoft.com/en-us/library/zhcy256f(v=vs.110).aspx  
 }  
 }  
  
 /// <remarks>O(1)  
 /// </remarks>  
 public int EdgeCount  
 {  
 get  
 {  
 return ((this.NodeCount - 1) \* this.NodeCount) / 2;  
 }  
 }  
 }  
}

## 

## 

## BaseGraphReader.cs

using System;  
using System.IO;  
  
namespace TP1.GraphReader  
{  
 public abstract class BaseGraphReader  
 {  
 private bool StillInNodeDefinition(string[] values)  
 {  
 int value;  
 if (Int32.TryParse(values[values.Length - 1], out value))  
 return value >= 1;  
 return false;  
 }  
  
 public abstract Stream Stream  
 {  
 get;  
 }  
  
 /// <remarks>O(N) - N = number of lines in the file - O(K+M) K = number of nodes, M = number of edges  
 /// </remarks>  
 public void Read(Action<string[]> onNodeRead, Action<string[]> onEdgeRead)  
 {  
 if (onNodeRead == null)  
 throw new ArgumentNullException("onNodeRead");  
 if (onEdgeRead == null)  
 throw new ArgumentNullException("onEdgeRead");  
  
 // open the file  
 using (var stream = Stream)  
 {  
 using (var reader = new StreamReader(stream))  
 {  
 // skip the first line  
 reader.ReadLine();  
 // process node definitions  
 while (!reader.EndOfStream) // O(K) - K = number of nodes  
 {  
 var values = reader.ReadLine().Split(',');  
 if (!StillInNodeDefinition(values))  
 break;  
 // call handler to process a new node  
 onNodeRead(values);  
 }  
 // process edge definitions  
 while (!reader.EndOfStream) // O(M) - M = number of edges  
 {  
 var values = reader.ReadLine().Split(',');  
 if (values.Length == 2)  
 {  
 // call handler to process a new edge  
 onEdgeRead(values);  
 }  
 }  
 }  
 }  
 }  
 }  
}

## GraphReader.cs

using System;  
using System.IO;  
  
namespace TP1.GraphReader  
{  
 public class GraphReader : BaseGraphReader  
 {  
 private readonly string filePath;  
  
 public GraphReader(string filePath)  
 {  
 this.filePath = filePath;  
 }  
  
 public override Stream Stream   
 {  
 get   
 {  
 return File.OpenRead (this.filePath);  
 }  
 }  
 }  
}

## 

## 

## GraphBuilder.cs

namespace TP1.GraphReader  
{  
 public class GraphBuilder  
 {  
 private readonly BaseGraphReader reader;  
 public GraphBuilder(BaseGraphReader reader)  
 {  
 this.reader = reader;  
 }  
  
 /// <remarks>O(N) - N = number of lines in the file - O(K+M) K = number of nodes, M = number of edges  
 /// </remarks>  
 public FacebookGraph Build()  
 {  
 var graph = new FacebookGraph();  
   
 // start reading  
 reader.Read(  
 // process a new node - O(1)  
 values =>  
 {  
 var id = long.Parse(values[0]);  
 var name = values[1];  
 graph.AddNode(new FacebookUser(id, name));  
 },  
 // process a new edge - O(1)  
 values =>   
 {  
 var nodeSource = graph[long.Parse(values[0])];  
 var nodeDest = graph[long.Parse(values[1])];  
 if (nodeSource != null && nodeDest != null)  
 {  
 nodeSource.AddAdjacentNode(nodeDest);  
 nodeDest.AddAdjacentNode(nodeSource);  
 }  
 });  
  
 return graph;  
 }  
 }  
}

## 

## Influences.cs

using System;  
using TP1.Graph;  
  
namespace TP1.Influences  
{  
 public struct Influence<TData, TId>  
 where TData : IIdentifiable<TId>  
 where TId : IComparable  
 {  
  
 public Influence(Node<TData, TId> node, double value)  
 : this()  
 {  
 Node = node;  
 Value = value;  
 }  
  
 public Node<TData, TId> Node { get; private set; }  
  
 public double Value { get; private set; }  
  
 public override string ToString()  
 {  
 return string.Format("{0} - {1}", Node.Data.Id, Value);  
 }  
 }  
}

## 

## 

## InfluencesCollection.cs

using System;

using System.Collections.Generic;

using System.Linq;

using TP1.Graph;

namespace TP1.Influences

{

public class InfluencesCollection<TData, TId>

where TData : IIdentifiable<TId>

where TId : IComparable

{

private const int InitialCapacity = 1000;

/// <remarks>O(1)

/// Source: http://msdn.microsoft.com/es-AR/library/4kf43ys3(v=vs.110).aspx

/// </remarks>

public InfluencesCollection(int initialCapacity = InitialCapacity)

{

this.Values = new List<Influence<TData, TId>>(initialCapacity);

}

public List<Influence<TData, TId>> Values { get; set; }

/// <remarks>O(1) best case, O(N) worst case

/// Source: http://msdn.microsoft.com/es-AR/library/3wcytfd1(v=vs.110).aspx

/// </remarks>

public void Add(Influence<TData, TId> influence)

{

this.Values.Add(influence);

}

public IEnumerable<Influence<TData, TId>> OrderByDescending()

{

return this.Values.OrderByDescending(i => i.Value);

}

}

}

## 

## 

## InfluencesExtension.cs

using System;  
using System.Collections.Generic;  
using System.Linq;  
using TP1.Graph;  
  
namespace TP1.Influences  
{  
 public static class InfluencesExtension  
 {  
 const long Infinity = long.MaxValue - 1;  
  
 /// <remarks>Reference: http://www.inf.uni-konstanz.de/algo/publications/b-vspbc-08.pdf </remarks>  
 /// <summary>  
 /// Calculates the betweenness centrality of all the nodes in a graph using the Brandes algorithm  
 /// </summary>  
 /// <returns></returns>  
 public static InfluencesCollection<TData, TId> GetInfluences<TData, TId>(this Graph<TData, TId> graph)  
 where TData : IIdentifiable<TId>  
 where TId : IComparable  
 {  
 #region initialize data structures and variables  
  
 // distance from source to to v ∈ V  
 var distanceTo = new Dictionary<TId, long>(graph.NodeCount);  
  
 // list of predecessors on shortest paths from source  
 var previous = new Dictionary<TId, HashSet<Node<TData, TId>>>(graph.NodeCount); // O(1)  
  
 // number of shortest paths from source to v ∈ V  
 var numberOfShortestPathsTo = new Dictionary<TId, double>(graph.EdgeCount);  
  
 // dependency of source on v ∈ V  
 var dependency = new Dictionary<TId, double>(graph.NodeCount);  
  
 // used in BFS  
 var queue = new Queue<Node<TData, TId>>(graph.NodeCount);  
  
 // nodes reachable from a source  
 var stack = new Stack<Node<TData, TId>>(graph.NodeCount);  
  
 var betweenness = new Dictionary<TId, double>(graph.NodeCount);  
  
 foreach (var node in graph.Nodes)  
 {  
 betweenness[node.Id] = 0;  
 }  
  
 #endregion  
  
 foreach (var source in graph.Nodes) // O(|V|) \* O(|V| + |E|) = O(|V|^2 + |V||E|)  
 {  
 #region single-source shortest-paths problem with BFS O(|V| + |E|)  
  
 foreach (var node in graph.Nodes)  
 {  
 previous[node.Id] = new HashSet<Node<TData, TId>>();  
 distanceTo[node.Id] = Infinity;  
 numberOfShortestPathsTo[node.Id] = 0;  
 dependency[node.Id] = 0;  
 }  
  
 numberOfShortestPathsTo[source.Id] = 1;  
 distanceTo[source.Id] = 0;  
 queue.Enqueue(source);  
  
 #region breadth first search O(|V| + |E|)  
 while (queue.Any())   
 {  
 var visitedNode = queue.Dequeue();  
 stack.Push(visitedNode);  
  
 foreach (var adjacent in visitedNode.Adjacents)  
 {  
 // a short path has been found! update the min dist and add the vertex to the queue  
 if (distanceTo[adjacent.Key] == Infinity)  
 {  
 distanceTo[adjacent.Key] = distanceTo[visitedNode.Id] + 1;  
 queue.Enqueue(adjacent.Value);  
 }  
  
 if (distanceTo[adjacent.Key] == (distanceTo[visitedNode.Id] + 1))  
 {  
 numberOfShortestPathsTo[adjacent.Key] += numberOfShortestPathsTo[visitedNode.Id];  
 previous[adjacent.Key].Add(visitedNode);  
 }  
 }  
 }  
 #endregion  
  
 #endregion  
  
 #region accumulation O(|E|)  
  
 while (stack.Any())  
 {  
 var element = stack.Pop();  
 foreach (var node in previous[element.Id])  
 {  
 dependency[node.Id] += (numberOfShortestPathsTo[node.Id] / numberOfShortestPathsTo[element.Id]) \* (1 + dependency[element.Id]);  
 }  
 if (!element.Equals(source))  
 {  
 betweenness[element.Id] += dependency[element.Id];  
 }  
 }  
 #endregion  
 }  
  
 var result = new InfluencesCollection<TData, TId>(graph.NodeCount);  
 foreach (var influence in betweenness)  
 {  
 result.Add(new Influence<TData, TId>(graph[influence.Key], influence.Value));  
 }  
  
 return result;  
 }  
 }  
}

## Recommendation.cs

using System;  
using TP1.Graph;  
   
namespace TP1.Recommendations  
{  
 public class Recommendation<TData, TId>  
 where TData : IIdentifiable<TId>  
 where TId : IComparable  
 {  
 public Node<TData, TId> Person { get; set; }  
   
 public Node<TData, TId> PersonToRecommend { get; set; }  
   
 public int FriendCount { get; set; }  
 }  
}

## RecommendationsCollection.cs

using System;  
using System.Collections;  
using System.Collections.Generic;  
using TP1.Graph;  
  
namespace TP1.Recommendations  
{  
 public class RecommendationsCollection<TData, TId> : IEnumerable  
 where TData : IIdentifiable<TId>  
 where TId : IComparable  
 {  
 public RecommendationsCollection()  
 {  
 this.Recommendations = new List<Recommendation<TData, TId>>();  
 }  
  
 public List<Recommendation<TData, TId>> Recommendations { get; private set; }  
  
 /// <remarks>O(1) best case, O(N) worst case  
 /// Source: http://msdn.microsoft.com/es-AR/library/3wcytfd1(v=vs.110).aspx  
 /// </remarks>  
 public void Add(Recommendation<TData, TId> recommendation)  
 {  
 if (recommendation.PersonToRecommend != null)  
 {  
 this.Recommendations.Add(recommendation);  
 }  
 }  
  
 /// <remarks>O(1)  
 /// Source: http://msdn.microsoft.com/en-us/library/27b47ht3%28v=vs.110%29.aspx  
 /// </remarks>  
 public int Count { get { return this.Recommendations.Count; } }  
  
  
   
 /// <remarks>O(1)  
 /// Source: http://msdn.microsoft.com/en-us/library/0ebtbkkc%28v=vs.110%29.aspx  
 /// </remarks>  
 public Recommendation<TData, TId> this[int index]  
 {  
 get { return this.Recommendations[index]; }  
 }  
  
 /// <remarks>O(1)  
 /// Source: http://msdn.microsoft.com/en-us/library/b0yss765%28v=vs.110%29.aspx  
 /// </remarks>  
 public IEnumerator GetEnumerator()  
 {  
 return this.Recommendations.GetEnumerator();  
 }  
 }  
}

## 

## 

## RecommendationsExtension.cs

using System;  
using System.Collections.Generic;  
using System.Linq;  
using TP1.Graph;  
  
namespace TP1.Recommendations  
{  
 public static class RecommendationsExtension  
 {  
 /// <remarks>O(N^2 + NE)  
 /// </remarks>  
 public static RecommendationsCollection<TData, TId> GetRecommendations<TData, TId>(this Graph<TData, TId> graph)  
 where TData : IIdentifiable<TId>  
 where TId : IComparable  
 {  
 var recommendations = new RecommendationsCollection<TData, TId>();  
  
 // O(N)  
 foreach (var node in graph.Nodes)  
 {  
 // O(N + E) where N is the number of nodes and E the number of edges  
 Recommendation<TData, TId> recommendation = Bfs (graph, node);  
  
 if (recommendation.PersonToRecommend != null)  
 recommendations.Add(recommendation); // O(1)  
 }  
 return recommendations;  
 }  
  
 private static Recommendation<TData, TId> Bfs<TData, TId>(Graph<TData, TId> graph, Node<TData, TId> root)  
 where TData : IIdentifiable<TId>  
 where TId : IComparable  
 {  
 var recomendation = new Recommendation<TData, TId> { Person = root };  
 var nodesCount = graph.Nodes.Count();  
 // queue holding node and the level in the BFS tree  
 var queue = new Queue<Tuple<int, Node<TData,TId>>>(nodesCount);  
 queue.Enqueue (Tuple.Create(0, root));  
 // nodes already visited by bfs  
 var visited = new Dictionary<TId, Node<TData,TId>>(nodesCount){ { root.Id, root } };  
 // friends in common  
 var friendsInCommon = new Dictionary<TId, int>(nodesCount);  
 // O(N) where N is the number of nodes  
 while (queue.Any())   
 {  
 var t = queue.Dequeue (); // O(1)  
 // stop looping at third level of the bfs tree  
 if (t.Item1 == 3)   
 break;  
 // O(En) where En is the number of adjacents nodes to Node n.  
 foreach (var adj in t.Item2.Adjacents)   
 {  
 if (!visited.ContainsKey (adj.Key)) { // O(1)  
 visited.Add (adj.Key, adj.Value); // O(1)  
 queue.Enqueue (Tuple.Create(t.Item1 + 1, adj.Value)); // O(1)  
 }  
 // t is at L1 and adj is at L2 - O(1)  
 if (t.Item1 == 1 &&   
 !adj.Value.Equals(root) && // O(1)  
 !root.Adjacents.ContainsKey(adj.Value.Id)) { // O(1)  
 // update the friends count of the nodes at L2  
 if (!friendsInCommon.ContainsKey(adj.Value.Id)) // O(1)  
 friendsInCommon.Add(adj.Value.Id, 0); // O(1)  
 friendsInCommon[adj.Value.Id] += 1; // O(1)  
 // update recommendation - O(1)  
 if (friendsInCommon [adj.Value.Id] > recomendation.FriendCount)   
 {  
 recomendation.FriendCount = friendsInCommon [adj.Value.Id];  
 recomendation.PersonToRecommend = adj.Value;  
 }  
 }  
 }  
 }  
  
 return recomendation;  
 }  
 }  
}

**GraphSortExtension.cs**

using System;  
using System.Collections.Generic;  
using System.Linq;  
using TP1.Graph;  
  
namespace TP1.Sort  
{  
 public static class GraphSortExtension  
 {  
 /// <remarks>O(1)  
 /// </remarks>  
 private static void Swap<TData, TId>(Node<TData, TId>[] nodes, int a, int b)  
 where TData : IIdentifiable<TId>  
 where TId : IComparable  
 {  
 var temp = nodes[a];  
 nodes[a] = nodes[b];  
 nodes[b] = temp;  
 }  
  
 /// <remarks>O(N)  
 /// </remarks>  
 private static void SiftDown<TData, TId>(Node<TData, TId>[] nodes, int start, int end, Func<Node<TData, TId>, Node<TData, TId>, bool> comparer)  
 where TData : IIdentifiable<TId>  
 where TId : IComparable  
 {  
 var root = start;  
 while(root \* 2 + 1 <= end) {  
 var child = root \* 2 + 1;  
 var swap = root;  
 if (comparer(nodes[swap], nodes[child]))  
 swap = child;  
 if (child + 1 <= end && comparer(nodes[swap], nodes[child + 1]))  
 swap = child + 1;  
 if (swap != root)  
 {  
 Swap(nodes, root, swap);  
 root = swap;  
 }  
 else   
 return;  
 }  
 }  
  
 /// <summary>  
 /// Sort the specified graph.  
 /// </summary>  
 /// <remarks>O(N Log N)  
 /// </remarks>  
 /// <param name='graph'>Graph to be sorted .</param>  
 /// <param name='comparer'>Comparer.</param>  
 /// <returns>A sorted list of nodes.</returns>  
 public static IEnumerable<Node<TData, TId>> Sort<TData, TId>(this Graph<TData, TId> graph, Func<Node<TData, TId>, Node<TData, TId>, bool> comparer)  
 where TData : IIdentifiable<TId>  
 where TId : IComparable  
 {  
 var nodes = graph.Nodes.ToArray();  
 var count = nodes.Count();  
  
 // heapify   
 var start = (int)Math.Floor((count - 2.0) / 2.0);  
 while (start >= 0) {  
 SiftDown(nodes, start, count-1, comparer);  
 start = start - 1;  
 }  
  
 // sort  
 var end = count - 1;  
 while (end > 0) {  
 Swap(nodes, end, 0);  
 end--;  
 SiftDown(nodes, 0, end, comparer);  
 }  
 return nodes;  
 }  
 }  
}

## FacebookGraph.cs

using TP1.Graph;  
   
namespace TP1  
{  
 public class FacebookGraph : Graph<FacebookUser, long>  
 {  
 public FacebookGraph()  
 : base()  
 {  
   
 }  
 }  
}

## FacebookUser.cs

using System.Diagnostics;  
using TP1.Graph;  
   
namespace TP1  
{  
 [DebuggerDisplay("Name: {Name}")]  
 public class FacebookUser : IIdentifiable<long>  
 {  
 public FacebookUser(long Id, string name)  
 {  
 this.Id = Id;  
 this.Name = name;  
 }  
   
 public long Id  
 {  
 get;  
 private set;  
 }  
   
 public string Name  
 {  
 get;  
 private set;  
 }  
   
 public override string ToString()  
 {  
 return this.Name;  
 }  
 }  
}

## 

## 

## Logger.cs

using System;  
using System.IO;  
   
namespace ConsoleApplication  
{  
 public class Logger : IDisposable  
 {  
 private static StreamWriter streamWriter;  
   
 public Logger(string fileName)  
 {  
 streamWriter = new StreamWriter(fileName);  
 }  
   
 public void Log(string message)  
 {  
 System.Console.WriteLine(message);  
 streamWriter.WriteLine(message);  
 streamWriter.Flush();  
 }  
   
 public void Dispose()  
 {  
 streamWriter.Close();  
 streamWriter.Dispose();  
 GC.SuppressFinalize(this);  
 }  
 }  
}

## 

## 

## Main.cs

using System.Configuration;  
using System.Diagnostics;  
using TP1.Influences;  
using TP1.Sort;  
using TP1.Recommendations;  
using TP1.GraphReader;  
   
namespace ConsoleApplication  
{  
 class MainClass  
 {  
 public static void Main()  
 {  
 var stopwatch = new Stopwatch();  
 stopwatch.Start();  
 var logger = new Logger(ConfigurationManager.AppSettings["logFileName"]);  
 try  
 {  
 var filePath = ConfigurationManager.AppSettings["inputFilePath"];  
 var builder = new GraphBuilder(new GraphReader(filePath));  
 var graph = builder.Build();  
   
 // Punto 1  
 logger.Log("POPULARIDAD:");  
 foreach (var node in graph.Sort((nodeA, nodeB) => nodeA.Degree >= nodeB.Degree))  
 {  
 logger.Log(node.Data.Name + " [" + node.Degree + "]");  
 }  
   
 // Punto 2  
 logger.Log("INFLUENCIAS:");  
 foreach (var influence in graph.GetInfluences().OrderByDescending())  
 {  
 logger.Log(influence.Node.Data.Name + " [" + influence.Value + "]");  
 }  
   
 // Punto 3  
 logger.Log("RECOMENDACIONES DE AMIGOS:");  
 foreach (var r in graph.GetRecommendations().Recommendations)  
 {  
 logger.Log(r.Person.Data.Name + " => " + r.PersonToRecommend.Data.Name + " [" + r.FriendCount + "]");  
 }  
 }  
 catch (System.Exception)  
 {  
 logger.Log("Error al leer el archivo de entrada.");  
 }  
 finally  
 {  
 stopwatch.Stop();  
 logger.Log("Tiempo de ejecucion total (en segundos): " + stopwatch.Elapsed.TotalSeconds);  
 logger.Dispose();  
 System.Console.Write("Presione una tecla para finalizar...");  
 System.Console.ReadLine();  
 }  
 }  
 }  
}

1. *Recuperar un valor utilizando su clave es muy rápido, cerca de O(1), porque la clase Dictionary<TKey, TValue> se implementa como una tabla hash. Fuente:* [*http://msdn.microsoft.com/es-es/library/xfhwa508(v=vs.110).aspx*](http://msdn.microsoft.com/es-es/library/xfhwa508(v=vs.110).aspx) [↑](#footnote-ref-1)
2. *La obtención del valor de esta propiedad [Count] es una operación O(1). Fuente:* [*http://msdn.microsoft.com/es-es/library/zhcy256f(v=vs.110).aspx*](http://msdn.microsoft.com/es-es/library/zhcy256f(v=vs.110).aspx) [↑](#footnote-ref-2)
3. Un delegate es un tipo de referencia que puede utilizarse para encapsular un método con nombre o anónimo. Los delegados son similares a los punteros a funciones de C++ [↑](#footnote-ref-3)
4. Los métodos de extensión permiten "agregar" métodos a los tipos existentes sin crear un nuevo tipo derivado, recompilar o modificar de otra manera el tipo original.

   Fuente: <http://msdn.microsoft.com/es-es/library/bb383977.aspx> [↑](#footnote-ref-4)