Tarea 4: Procesamiento de Grafos

Alberto Pascal Garza
Análisis y Diseño de Algoritmos

Profesor Vicente Cubells
ITESM Campus Santa Fe
C.P: 01389
Email: A01023607@itesm.mx

***Abstract*—En este reporte de investigación se hablará sobre la exportación de grafos a distintos formatos para así poder ser representados visualmente en programas como gephi. Este procedimiento es nos facilita el entendimiento de nuestro grafo dado a que se puede ver gráficamente la relación entre los distintos nodos . Además, se analizará el proceso contrario. Es decir, se analizará la forma en que podemos importar los contenidos de nuestros grafos a un archivo de texto para su utilización y manipulación en otros formatos.Es necesario hacer uso de la biblioteca de Snap-4.0**

I. Introducción

Los grafos son una estructura de datos compuesta por una serie de nodos, vértices y aristas. Son ampliamente utilizados para representar información. Es una de las formas más versátiles de ver información gráfica dado que se puede representar con cualquier tipo de estructuras (es decir, pilas, colas, árboles). Algunas de sus aplicaciones van desde tomar decisiones en videojuegos, las relaciones entre las amistades de usuarios de redes sociales o cualquier otro tipo de redes. Además, podemos utilizar métodos de búsqueda y de exploración tales como Prim, Kruskal, Bellman Ford, Floyd Warshall, BFS, DFS y Dijkstra para hacer de su uso más eficiente.

II. Metodología

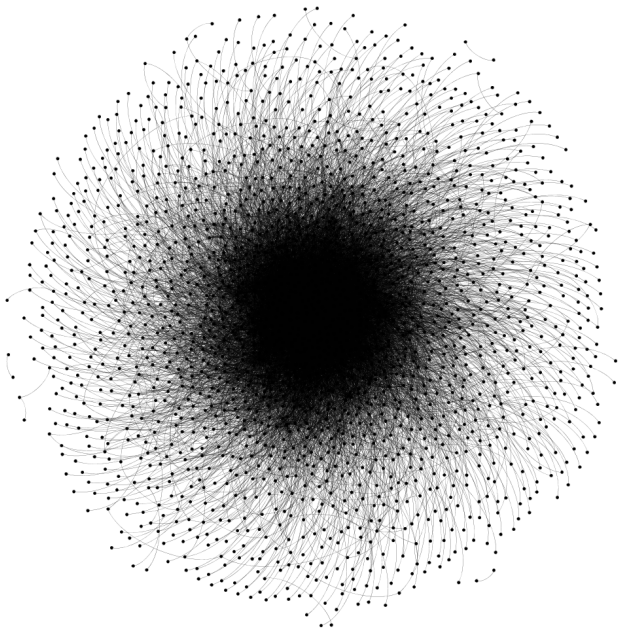
Para realizar la actividad se generó un código capaz de leer de un archivo de texto un grafo con una gran cantidad de nodos. Para poder hacer esto, se utilizó la librería de Snap-4.0 (obtenida de Standford Network Analysis Project). Además, dicho código se modificó para poder exportar el grafo leído a los formatos GraphML, GEXF, GDF y JSON Graph Format. De esta forma, se pudieron utilizar los archivos exportados al programa Gephi, el cual es cpaz de representar gráficamente nuestros datos y poder obtener información más relevante.

A. Procedimiento

1) Descargar la información del grafo (dataset): Para obtener la información del grafo que se empleó en esta actividad se descargó de la página oficial de la universidad de Stanford (https://snap.stanford.edu/data/index.html.) un dataset que contuviera la información en formato ".txt". De esta forma se podrá implementar en el código fuente de conversión.

2) Implementación de las funciones en C++: Para poder hacer uso de las funciones se instaló previamente la librería de SNAP. Para hacer esto, se descargó el arhcivo de la librería y se escribió dentro de la terminal el comando "make all". El código del programa se escribió usando el lenguaje C++. Fue necesaro poner el código dentro de la carpeta de SNAP para poder hacer uso de "stdafx.h", que permite la utilización de las funciones para cargar el grafo (LoadEdgeListo). Además, se hizo uso del objeto "pUNGraph" que permite utilizar iteradores para escribir parista por arista el archivo de salida en todos los formatos antes mencionados.

3) Importación del grafo exportado a Gephi: Una vez que nuestro grafo fue exportado mediante la ejecución de nuestro código fuente es necesario importarl a la herramienta Gephi. De esta forma, Gephi podrá representar gráficamente a nuestro grafo. Para esta actividad se utilizó el dataset que contenía los mensajes de texto mandados dentro de una red social de la Irvin, en California. A continuación se muestra su representación gráfica:



III. Ventajas y desventajas de los formatos

Los visualizadores de grafos nos permiten tener una mejor claridad de nuestro grafo para poder analizarlo con mayor precisión. Sin embargo, de todos los formatos en los que podemos existen algunos que nos proporcionan más privilegios que otros. Como tal no existe un mejor formato para todas las situaciones sino que debemos adaptarnos según los requisitos que se nos presenten. Es importante mencionar que sólamente GraphML, GDF y GEXF pueden reconocer las posiciones, colores y tamaño de todos los atributos. Lamentablemente, aunque éstos últimos tres son de los más usados, presentan distintas funcionalidades entre sí. GraphML es el que nos permite detallar más la información. GEXF nos permite definir en un espaico temporal los valores de nuestro grafo y GDF tiene la ventaja de ser el más ligero de los formatos. En cuanto a sus desventajas, tanto GraphML como GEXF son de los formatos más pesados debido a la cantidad de información y detalle que manejan por lo que en algunos dispositivos pueden ser problemáticos. GDF es el más ligero de todos pero presenta la desventaja de no tener tanto detalle como los dos anteriores. Por esto mismo, existe una cuarta opción llamada JSON, la cuál nos permite balancear la simpleza de la información con el redimiento del grafo para utilizarlo en cualquier dispositivo.

A. GraphML

GraphML está hecho a partir de una estructura de XML. Por esto mismo, tiene una ventaja en cuanto a lo versátil que puede ser al momento de intercambiar datos del grafo entre plataformas. Además, es el formato estándar más legible y relativamente sencillo de entender. Posee la capacidad para representar los grafos tales como sus pesos y cada una de sus aristas que, además, representa mediante el uso de distintos colores, tamaños y posición. Nos provee de la capacidad de crear grafos con jerarquías y establecer valores predeterminados para los atributos específicos de los nodos y aristas.

B. GEXF

Así como GraphML, GEXF se basa en la estructura XML para así poder tener más versatilidad de transferencia de datos. GEXF tiene todas las capacidades de GraphML, pero cuenta con la ventaja adicional de darle un tiempo de vida especifico a cada nodo, arista o atributo. De esta manera se puede definir en qué momento en el tiempo se deben de alterar los valores o simplemente eliminarse un nodo o arista específico haciendo de este formato ideal para proyectos más dinámicos. La manera en que se implementa esta capacidad es mediante dos formas, un atributo de tipo ”double” o con el formato internacional de fecha y hora (yyyy-mm-dd).

C. GDF

GDF es un formato más ligero y , por lo tanto, cuenta con una menor complejidad espacial. Se utiliza para la creación de grafos y a diferencia de los formatos anteriores, solo almacena la información necesaria que pudiese necesitar un grafo con atributos tales como pesos en las aristas, atributos para los nodos/aristas, atributos de visualización y la posibilidad de establecer valores predeterminados. Por lo tanto carece de funcionalidades interesantes como el tiempo de vida de los nodos/aristas/atributos y la posibilidad de hacer grafos con jerarquías.

D. JSON Graph Format

JSON, que es en realidad JavaScript Object Notation, es un formato para intercambiar datos. Es una de las alternativas a XML. Utiliza JavaScript y puede ser leido por cualquier lenguaje de programación. Por esto mismo, JSON puede ser utilizado para el intercambio de información entre distintas tecnologías. Es un formato simple que genera archivos de menor tamaño y posee una velocidad de procesamiento alta aunque su estructura es difícil de interpretar a simple vista. Debido a esto, es ideal para dispositivos con menores prestaciones como los teléfonos móviles en donde la duración de la batería y el consumo de datos es un factor muy importante.

IV. Complejidad temporal y espacial de cada función

A. Exportación a GraphML

- Complejidad temporal: O(N+A) (N=Nodos, A=Aristas)
- Complejidad espacial: S(1)
- Tiempo de ejecución: 15 ms

B. Exportación a GEXF

- Complejidad temporal: O(N+A)
- Complejidad espacial: S(1)
- Tiempo de ejecución: 21 ms

C. Exportación a GDF

- Complejidad temporal: O(N+A)
- Complejidad espacial: S(1)
- Tiempo de ejecución: 7 ms

D. Exportación a JSON

- Complejidad temporal: O(N+A)
- Complejidad espacial: S(1)
- Tiempo de ejecución: 11 ms

References

[1] Oscar Blancarte *JSON vs XML*. recuperado el 22 de octubre de 2017 de: https://www.oscarblancarteblog.com/2014/07/18/json-vs-xml/

[2] J. McAuley and J. Leskovec *Learning to Discover Social Circles in Ego Networks* recuperado el 22 de Octubre de 2017 de: NIPS, 2012

[3] Alejandro Esquiva *¿Qué es y para qué sirve JSON?* recuperado el 22 de octubre de 2017 de: https://geekytheory.com/json-i-que-es-y-para-que-sirve-json/

[4] Gephi *Supported Graph Formats* recuperado el 22 de octubre de 2017 de: https://gephi.org/users/supported-graph-formats/

[5] Gephi *GEXF File Format* recuperado el 22 de octubre de 2017 de: https://gephi.org/gexf/format/dynamics.html

[6] Gephi *GraphML* recuperado el 22 de octubre de 2017 de: https://gephi.org/users/supported-graph-formats/graphml-format/

[7] Gephi *GDF Format* recuperado el 22 de octubre de 2017 de: https://gephi.org/users/supported-graph-formats/gdf-format/

Appendix

https://github.com/AlbertoPascal/Tarea4_algoritmos

```cpp
#include <iostream>
#include <fstream>
#include <chrono>
#include "stdafx.h"


void GraphML(PUNGraph graph);
void GEXF(PUNGraph graph);
void GDF(PUNGraph graph);
void JSON(PUNGraph graph);


using namespace std::chrono;

//Un main para exportar un grafo a formatos como JSON,
//GraphML, GDF y GEXF. Esto para abrirlos en Gephi



int main(int argc, char* argv[]) {

	PUNGraph g = TSnap::LoadEdgeList<PUNGraph>("migrafo.txt", 0, 1);

	high_resolution_clock::time_point t1 = high_resolution_clock::now();
	GraphML(g);
	high_resolution_clock::time_point t2 = high_resolution_clock::now();
	auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1).count();
	std::cout << "GraphML: " << duration << std::endl;

	t1 = high_resolution_clock::now();
	GEXF(g);
	t2 = high_resolution_clock::now();
	duration = std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1).count();
	std::cout << "GEXF: " << duration << std::endl;

	t1 = high_resolution_clock::now();
	GDF(g);
	t2 = high_resolution_clock::now();
	duration = std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1).count();
	std::cout << "GDF: " << duration << std::endl;

	t1 = high_resolution_clock::now();
	JSON(g);
	t2 = high_resolution_clock::now();
	duration = std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1).count();
	std::cout << "JSON: " << duration << std::endl;



	return 0;
}



void GraphML(PUNGraph graph)
{
	std::ofstream file("grafo.graphml");
	if (file.is_open())
	{
		file << "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n";
		file << "<graphml xmlns=\"http://graphml.graphdrawing.org/xmlns\" xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\" xsi:schemaLocation=\"http://graphml.graphdrawing.o
		file << "<graph id=\"G\" edgedefault=\"directed\">\n";

		for (PUNGraph::TObj::TNodeI NI = graph->BegNI(); NI < graph->EndNI(); NI++)
		{
			file << "<node id=\"" << NI.GetId() << "\"/>\n";
		}

		int i = 1;
		for (PUNGraph::TObj::TEdgeI EI = graph->BegEI(); EI < graph->EndEI(); EI++, ++i)
		{
			file << "<edge id=\"e" << i << "\" source=\"" << EI.GetSrcNId() << "\" target=\"" << EI.GetDstNId() << "\"/>\n";
		}

		file << "</graph>\n";
		file << "</graphml>\n";
		file.close();
	}
}


void GEXF(PUNGraph graph)
{
	std::ofstream file("grafo.gexf");
	if (file.is_open())
	{
		file << "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n";
		file << "<gexf xmlns=\"http://www.gexf.net/1.2draft\" version=\"1.2\">\n";
		file << "<graph mode=\"static\" defaultedgetype=\"directed\">\n";

		file << "<nodes>\n";
		for (PUNGraph::TObj::TNodeI NI = graph->BegNI(); NI < graph->EndNI(); NI++)
		{
			file << "<node id=\"" << NI.GetId() << "\" />\n";
		}
		file << "</nodes>\n";

		file << "<edges>\n";
		int i = 1;
		for (PUNGraph::TObj::TEdgeI EI = graph->BegEI(); EI < graph->EndEI(); EI++, ++i)
		{
			file << "<edge id=\"" << i << "\" source=\"" << EI.GetSrcNId() << "\" target=\"" << EI.GetDstNId() << "\" />\n";
		}
		file << "</edges>\n";

		file << "</graph>\n";
		file << "</gexf>\n";
		file.close();
	}
}
```

```
void GDF(PUNGraph graph)
{
	std::ofstream file("grafo.gdf");
	if (file.is_open())
	{
		file << "nodedef>name VARCHAR\n";
		for (PUNGraph::TObj::TNodeI NI = graph->BegNI(); NI < graph->EndNI(); NI++)
		{
			file << NI.GetId() << "\n";
		}

		file << "edgedef>source VARCHAR, destination VARCHAR\n";
		for (PUNGraph::TObj::TEdgeI EI = graph->BegEI(); EI < graph->EndEI(); EI++)
		{
			file << EI.GetSrcNId() << "," << EI.GetDstNId() << "\n";
		}

		file.close();
	}
}

void JSON(PUNGraph graph)
{
	std::ofstream file("grafo.json");
	if (file.is_open())
	{
		file << "{ \"graph\": {\n";
		file << "\"nodes\": [\n";
		for (PUNGraph::TObj::TNodeI NI = graph->BegNI(); NI < graph->EndNI();)
		{
			file << "{ \"id\": \"" << NI.GetId() << "\" }";
			if (NI++ == graph->EndNI())
				file << " ],\n";
			else
				file << ",\n";
		}

		file << "\"edges\": [\n";
		for (PUNGraph::TObj::TEdgeI EI = graph->BegEI(); EI < graph->EndEI(); )
		{
			file << "{ \"source\": \"" << EI.GetSrcNId() << "\", \"target\": \"" << EI.GetDstNId() << "\" }";
			if (EI++ == graph->EndEI())
			{
				file << " ]\n";
			}
			else
			{
				file << ",\n";
			}
		}
		file << "} }";

		file.close();
	}
}
```