UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

IMPLEMENTACIÓN DE GRAFO CON ETIQUETAS USANDO ESTRUCTURAS DE DATOS COMPACTAS

INFORME FINAL DE CC6908 PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

Josefa Robert

PROFESOR GUÍA:
Gonzalo Navarro

SANTIAGO DE CHILE
2023

1. Introducción

Cada vez son más las áreas donde se necesita tratar con una enorme cantidad de información digital. Esto trae los retos de no solo almacenar y gestionar eficientemente, sino también de poder explotar plenamente el valor de los propios datos, es decir, de ser capaz de realizar consultas sobre los datos que devuelvan resultados relevantes en tiempos razonables. Con este fin, el problema de diseñar estructuras de datos que sean eficientes en términos de espacio y tiempo se ha vuelto un tema de interés.

Los grafos son una conocida estructura combinatorial utilizados para codificar dependencias en una multitud de fenómenos del mundo real, por lo que son de gran interés en el mundo de la computación y las matemáticas. Es común querer trabajar con grafos de gran tamaño, por lo que representaciones compactas han sido estudiadas para diversos tipos de grafos [6, 10].

En este trabajo nos enfocamos en el caso de grafos dirigidos con etiquetas en sus aristas. Este tipo de grafos se usan, por ejemplo, en el modelamiento de estructuras web, de redes sociales, de redes de recomendación, de dependencias semánticas y de bases de datos que usan el *Resource Description Framework* (RDF) [3, 7].

El modelo RDF, en conjunto con su lenguaje de consultas estándar SPARQL, han ganado popularidad recientemente y diversas estructuras compactas que soportan operaciones SPARQL se han desarrollado [4]. Motivados por esto, se propone en este trabajo de título implementar la representación compacta para grafos etiquetados descrita en [21], con el fin de utilizarla para modelar sentencias RDF.

Esta representación está construida usando bitvectors y wavelet trees, dos estructuras tales que el uso del espacio es óptimo en el sentido de la teoría de la información. Además, la propuesta de [21] soporta operaciones que responden a preguntas básicas sobre el grafo (*e.g.* listar los vecinos de un nodo y entregar los nodos objetivos de todos los arcos con una cierta etiqueta) en tiempos teóricos competitivos. Más aún, los wavelet trees tienen la capacidad de encontrar eficientemente las distintas etiquetas que salen de un nodo fijo, lo que resulta útil en varias heurísticas que resuelven *property paths* de SPARQL.

2. Estado del Arte

2.1. RDF

El *Resource Description Framework*, o RDF para abreviar, es un modelo de base de datos que tiene como sentencias atómicas a tripletas de la forma $(\text{sujeto}, \text{predicado}, \text{objeto})$, donde el sujeto corresponde a una entidad (p. ej., una persona o una escuela), el predicado a una característica de la entidad (p. ej., edad o ubicación) y el objeto al valor del predicado para la entidad en cuestión (p. ej., 32 o Santiago).

Este modelo es usado en diversas áreas, como la medicina, la lingüística y la geografía, y existen proyectos masivos como *Linking Open Data Project* o *WikiData* que contienen cientos de millones de tripletas [22].

Existen muchos esfuerzos destinados a diseñar representaciones para datos RDF (y los respectivos algoritmos que resuelven consultas SPARQL) eficientes en espacio y tiempo. Luo *et al.* [16] clasifican estos esfuerzos según tres perspectivas: la relacional, la de entidades y la de grafos.

El primer grupo plantea al RDF como un tipo de base de datos relacional y gran parte de la literatura se enfoca en adaptar técnicas previamente desarrolladas para esta clase de bases de datos [4]. Un ejemplo sencillo en esta línea es una tabla relacional con las columnas sujeto, predicado y objeto.

La perspectiva de entidades toma una visión nodo-céntrica e interpreta los sujetos y objetos de la RDF como entidades con propiedades, donde estas últimas corresponden a los predicados. Para este tipo de representación es común usar estructuras de datos de índices invertidos en conjunto con técnicas del área de *information retrieval* [16].

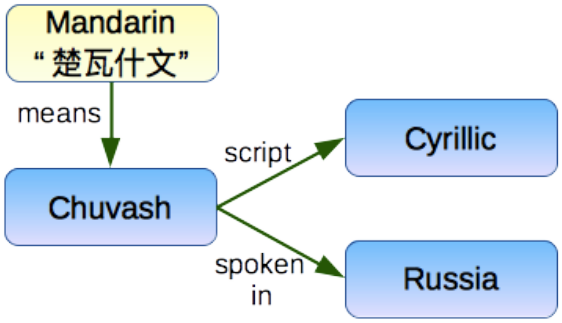


Figura 1: Representación en grafo de las tripletas { ⟨楚瓦什文, means, Chuvash⟩, ⟨Chuvash, script, Cyrillic⟩, ⟨Chuvash, spoken in, Russia⟩}. Imagen extraída de Lexvo.org.

Por último, la perspectiva basada en grafos, como su nombre lo indica, representa la información como un grafo donde los sujetos y objetos son los nodos y los predicados son etiquetas de aristas dirigidas. Para este método, es común querer adaptar técnicas usadas en otras bases de datos orientadas a grafos. La Figura 1 muestra una representación en grafo de un extracto de la base de datos RDF de Lexvo.org [19].

2.2. SPARQL

Entre los varios lenguajes de consulta para RDF (como RQL, RDQL o SeRQL), SPARQL -un acrónimo recursivo para *SPARQL Protocol and RDF Query Language*- ha sido aceptado por la comunidad como el lenguaje estándar [2].

El núcleo de las consultas SPARQL son los *patrones triples*, los cuales son una tripleta RDF con la diferencia de que una o más variables pueden sustituir los valores del sujeto, el predicado y el objeto. A su vez, una conjunción de patrones de triples forma un *patrón de grafo básico* (o BGP, por sus siglas en inglés). La Figura 2 muestra un ejemplo de BGP con dos variables y la consulta SPARQL equivalente.

La sintaxis de SPARQL es similar a la del lenguaje SQL y los resultados obtenidos se devuelven en forma de tablas (“select queries”) o en una plantilla para generar nuevos datos

RDF ("construction queries").

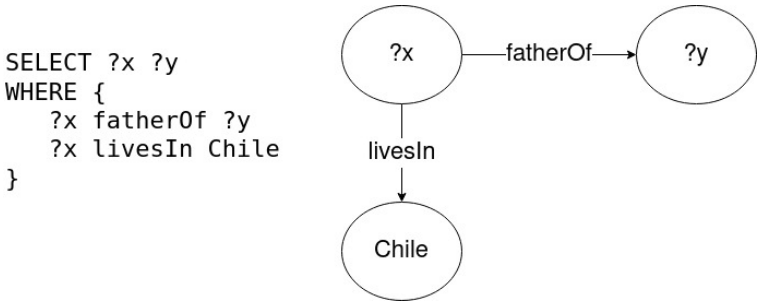


Figura 2: Una BGP y su respectiva consulta SPARQL, correspondiente a los padres que habitan en Chile y sus hijos.

Property Paths

Dentro de las nuevas características que fueron añadidas en SPARQL 1.1 se encuentran los *property paths* [2]. Esta operación encuentra todos los caminos que cumplen una expresión regular dada, y donde los extremos del camino pueden ser términos RDF o variables (las variables no pueden utilizarse como parte de la ruta en sí).

Un caso trivial es un property path de largo 1, el cual es equivalente a un patrón triple. Un caso más complejo es la consulta $(?x, \text{subclass}^*, ?y)$ que encuentra subclases. Notar que subclass^* puede calzar recursivamente subclases, por lo que el camino entre $?x$ e $?y$ puede tener un largo arbitrario.

Buscar property paths es equivalente a resolver *regular path queries* (RPQs) [11]. Hay una extensa literatura sobre las RPQs que consiste mayoritariamente de trabajos teóricos. Algunas de las heurísticas usadas para evaluar RPQs son utilizar consultas SQL recursivas [12], obtener el producto de grafos entre la data original y la expresión regular[5], obtener respuestas preliminares que cubran prefijos, infijos o sufijos de la expresión para luego conectarlas [23], e identificar predicados de la RPQ con pocas conexiones para eliminar varios caminos candidatos [15].

2.3. Estructuras de Datos Compactas

Operaciones rank, select y access

Tres herramientas básicas utilizadas en numerosas estructuras de datos compactas son rank, select y access (abreviadas RSA). Estas operaciones se definen como siguen.

Sea $S[1, n]$ una secuencia de símbolos sobre el alfabeto $\Sigma = \{1, \ldots, \sigma\}$ y $a \in \Sigma$ una letra. Definimos,

- $\mathsf{access}(S, i)$ retorna el símbolo en $S[i]$, para cualquier $1 \leq i \leq n$.
- $\mathsf{rank}_a(S, i)$ retorna el número de veces que a aparece en $S[1, i]$, para cualquier $0 \leq i \leq n$; en particular $\mathsf{rank}_a(S, 0) = 0$.
- $\mathsf{select}_a(S, j)$ retorna la posición de la j-ésima aparición de a en S, para cualquier $j \geq 0$; en particular $\mathsf{select}_a(S, 0) = 0$ y $\mathsf{select}_c(S, j) = n + 1$ si $j > \mathsf{rank}_a(S, n)$.

Existe una gran variedad de técnicas para implementar estas operaciones dependiendo de la naturaleza de la estructura base. Llamamos *estructura RSA* a una estructura que soporta las operaciones rank, select y access.

Bitvectors

Un bitvector B es una estructura RSA sobre una secuencia de tamaño n y alfabeto $\Sigma = \{0, 1\}$. Los bitvectors son la base de muchas estructuras compactas, por lo que se han desarrollado numerosas representaciones adaptadas a diferentes aplicaciones.

En este trabajo usaremos los llamados bitvectors planos. Esta es una representación sin compresión y que con solo $o(n)$ bits extra logra soportar las operaciones RSA en tiempo constante. Munro [20] y Clark [8] lograron estos tiempos mediante técnicas de agrupación consistentes en dividir la información en super-bloques, bloques y mini-bloques de tamaño conveniente y en guardar resultados parciales precomputados sobre ellos.

Wavelet Trees

Para representar secuencias de símbolos, una de las estructuras RSA más utilizadas es el wavelet tree[14]. Esta representación utiliza $n \log \sigma + o(n \log \sigma) + O(\sigma \log n)$ bits y resuelven las operaciones RSA en $O(\log \sigma)$.

El wavelet tree es un árbol binario donde cada nodo representa una subsecuencia sobre un subalfabeto de los originales. Es decir, si se quiere almacenar una secuencia S, se dividirá el alfabeto en dos partes de aproximadamente igual tamaño, y luego se repetirá este proceso en cada nodo hasta que el alfabeto alcance tamaño 1 en las hojas. Para representar la partición en cada nodo (exceptuando las hojas, que son solo conceptuales), se utiliza un bitvector que demarca el orden de las letras que pertenecen a cada subalfabeto, indicando con un 1 si una letra dada está en el alfabeto correspondiente al hijo derecho y un 0 si está en el izquierdo. Para responder las operaciones RSA solo se requiere conocer a los bitvectors y al alfabeto original.

En la Figura 3 se muestra un ejemplo de wavelet tree para la secuencia wavelettree. Se muestran en negro los bitvectors que conforman el wavelet tree y en gris la secuencia que representa cada nodo.

Una de las desventajas del wavelet tree es que se necesitan $\Omega(\sigma \log n)$ tan solo para almacenar punteros, lo que se vuelve considerablemente costoso para alfabetos grandes. Motivados por esto, Mäkinen y Navarro [17, 18] introdujeron el *levelwise wavelet tree*, que guarda un único bitvector para cada nivel (en vez de uno por cada nodo) y realiza cálculos sobre la marcha para recuperar las divisiones originales. Esto elimina los $\Omega(\sigma \log n)$ bits usados en punteros, reduciendo el tamaño del árbol a $n \log \sigma + o(n \log \sigma)$. Sin embargo, los algoritmos sobre esta representación tienen la desventaja de ser innecesariamente lentos en la práctica [9].

Un rediseño del levelwise wavelet tree, llamado *wavelet matrix*, fue presentada por Claude *et al.* [9]. La principal diferencia es que los bitvectors de cada nivel del árbol se permutan, dejando todos los hijos 0 del nivel a la izquierda y los hijos 1 a la derecha. Esto permite

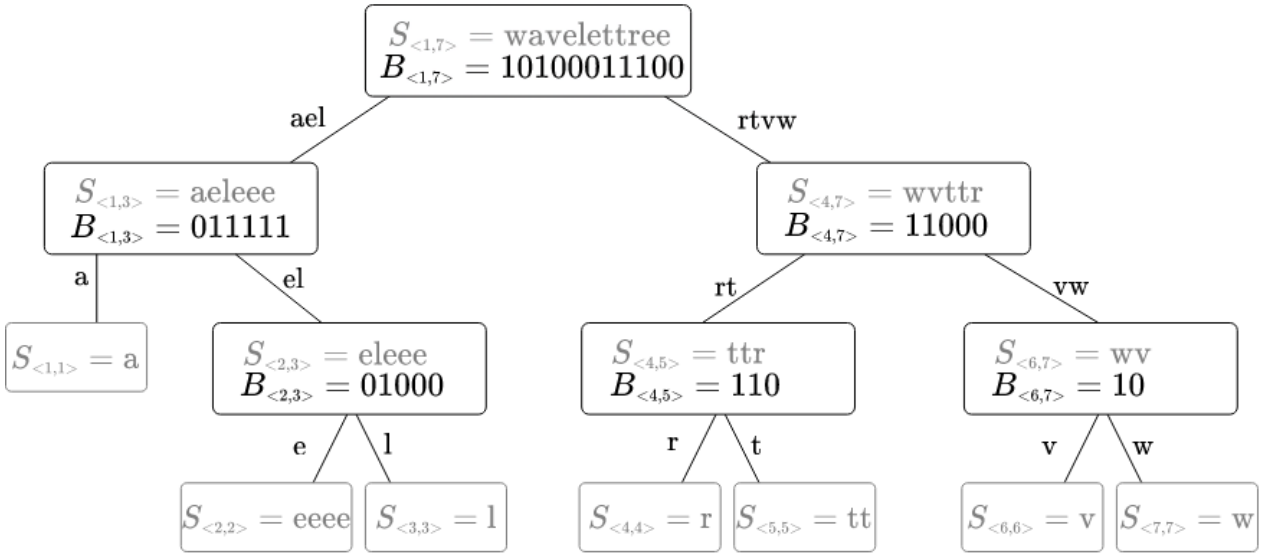


Figura 3: Representación de la secuencia S = wavelettree utilizando un wavelet tree. Se muestra en gris la información conceptual y en negro la representación real.

una construcción más sencilla y tiempos que en la práctica son más eficientes, a la vez que se conserva la cantidad de espacio utilizado y, así, la ventaja de ser eficiente para alfabetos grandes.

Librería de estructuras de datos sucinta

Para la construcción de la representación de [21] se planea usar intensivamente *the succinct data structure library* (SDSL) desarrollada por Simon Gog en C++11 [13]. Esta librería contiene implementaciones de diversas estructuras de datos sucintas, entre ellos bitvectors y wavelet trees, y ha demostrado en la práctica obtener espacios y tiempos cercanos a los teóricos.

3. Objetivos

3.1. Objetivo General

El objetivo principal de este trabajo de título es implementar la representación compacta de grafos con etiquetas de [21], en conjunto con las operaciones detalladas en la Sección 4 y los métodos que permitan procesar una base de datos RDF.

3.2. Objetivos Específicos

1. Implementar la función que inicializa L y N.
2. Implementar la función que inicializa B y B_L.
3. Implementar las operaciones descritas en la Sección 4.
4. Evaluar el espacio y tiempo de construcción y el espacio final usado por la estructura.

5. Evaluar el tiempo de las operaciones y su funcionamiento.

3.3. Evaluación

Para evaluar el correcto funcionamiento de la estructura se utilizarán dos bases de datos de diferentes tamaños. La primera es obtenida de Lexvo.org [19], y contiene alrededor de $3{,}5 \cdot 10^6$ tripletas con información sobre lenguas, alfabetos y otras entidades relacionadas con el lenguaje humano. Debido a su reducido tamaño, estos datos planean usarse para hacer un test inicial de la estructura implementada y de las operaciones que debe soportar.

La segunda base de datos es Wikidata [22], compuesta de $\sim 10^9$ tripletas, y que corresponde a una carga de trabajo más realista en cuanto a la escala en la que se espera poder aplicar la solución desarrollada. En ambos casos, las operaciones definidas deben entregar los outputs adecuados con tiempos cercanos a los descritos en la Sección 4 y el espacio utilizado en memoria debe ser cercano al calculado teóricamente.

4. Solución Propuesta

En este trabajo, un grafo dirigido con aristas etiquetadas es un conjunto de tripletas $G \subseteq V \times V \times [1, \lambda]$, donde V es el conjunto de vértices y $[1, \lambda]$ el conjunto de etiquetas. Asumimos por simplicidad que los nodos y las etiquetas son enteros contiguos enumerados desde 1. También definimos $n = |V|$ y $e = |G|$.

4.1. Detalle de la representación del grafo

La representación de [21] se basa en las estructuras N, L, B_L y B descritas a continuación.

- N es la concatenación $N = N_1 N_2 \dots N_\lambda$, donde N_l es una lista con todos los $\{v : (u, v, l) \in G\}$ ordenados de acuerdo a u y luego de acuerdo a v. N contiene e elementos.
- L es la concatenación $L = L_1 L_2 \dots L_n$, donde L_u es una lista con todas las etiquetas $\{l : (u, v, l) \in G\}$ ordenados de acuerdo a v y luego de acuerdo a l. L contiene e elementos.
- B_L tiene la forma $10^{|N_1|}10^{|N_2|}1 \dots 0^{|N_\lambda|}$. Demarca los inicios y finales de los N_l que conforman N. B_L contiene $e + \lambda$ elementos.
- B tiene la forma $10^{|L_1|}10^{|L_2|}1 \dots 0^{|L_n|}$. Demarca los inicios y finales de los L_u que conforman L. B contiene $e + n$ elementos.

Las secuencias N y L se implementarán como wavelet trees, y B_L y B usando la representación plana de bitvectors. En conjunto, el grafo necesitaría $e(\log n + \log \lambda)(1 + o(1)) + O(n + \lambda)$ bits.

4.2. Detalle de las operaciones

Dividimos las operaciones que debe soportar la representación en dos grupos. El primer grupo corresponde a operaciones clásicas de la literatura y el segundo a operaciones menos comunes pero que, gracias a la representación con wavelet trees, pueden implementarse sin problemas y que aportan información adicional sobre el grafo.

Operaciones clásicas

- $\mathsf{adj}(G, u, v)$: retorna 1 si existe una arista de u a v; equivalentemente si (u, v, l), para cualquier l, pertenece a G o no.
- $\mathsf{neigh}(G, u)$: retorna la lista de vecinos de u, es decir, $\{v : (u, v, l) \in G\}$.
- $\mathsf{rneigh}(G, v)$: retorna la lista de vecinos reversos de v, es decir, $\{u : (u, v, l) \in E\}$.
- $\mathsf{outdegree}(G, u)$: retorna el número de vecinos de u, es decir, $|\mathsf{neigh}(G, u)|$.
- $\mathsf{indegree}(G, v)$: retorna el número de vecinos reversos de v, es decir, $|\mathsf{rneigh}(G, v)|$.

Calcular indegree y el j-ésimo elemento de neigh y rneigh son operaciones que toman $O(\log \lambda)$. Para outdegree basta $O(1)$ y adj toma $O(\lambda \log \lambda)$.

Las operaciones adj_l, neigh_l, rneigh_l, $\mathsf{outdegree}_l$ e $\mathsf{indegree}_l$ se definen de manera análoga a las operaciones clásicas pero sobre el subgrafo $G_l = \{(u, v) : (u, v, l) \in G\}$, es decir, considerando solo las aristas con etiqueta l. La complejidad de estas operaciones es $O(\log \lambda)$ (donde para neigh_l y rneigh_l la complejidad corresponde a obtener el j-ésimo elemento).

Operaciones adicionales

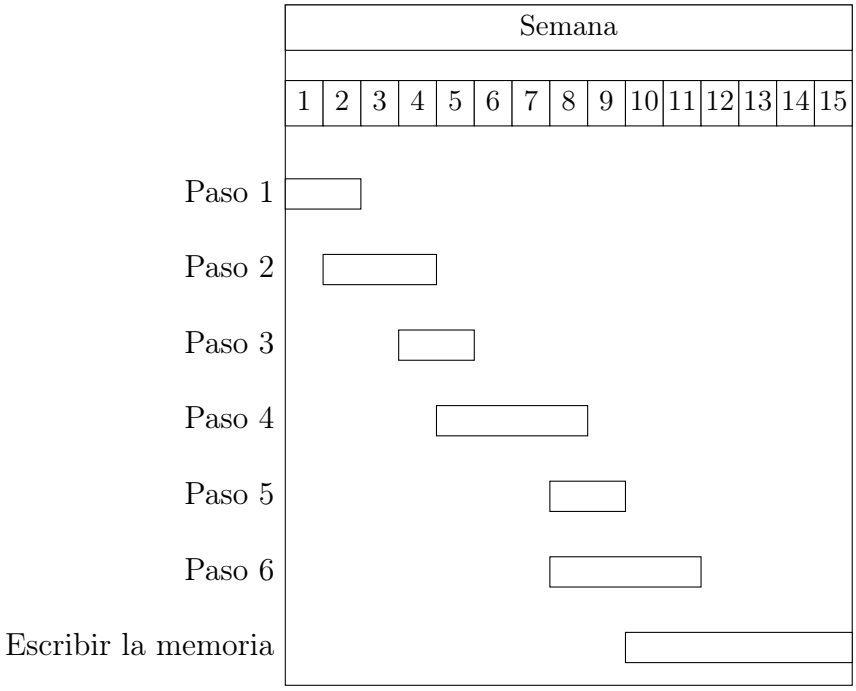
- $\mathsf{access}_l(G, j)$: retorna la j-ésima arista con etiqueta l en G_l.
- $\mathsf{count}_l(G)$: retorna el número de aristas con etiqueta l en G, es decir, $|\{(u, v) : (u, v, l) \in G\}|$.
- $\mathsf{sources}_l(G)$: retorna los nodos u que son origen de un vértice con etiqueta l, es decir, $\{u : (u, v, l) \in G\}$.
- $\mathsf{targets}_l(G)$: retorna los nodos v que son destino de un vértice con etiqueta l, es decir, $\{v : (u, v, l) \in G\}$.

La complejidad de access_l y count_l es $O(\log \lambda)$, y para $\mathsf{sources}_l$ y $\mathsf{targets}_l$ se tiene $O(n \log \lambda)$.

La operaciones se implementan usando combinaciones de las operaciones RSA sobre las estructuras N, L, B y B_L. La idea base es que las delimitaciones de N y L según etiquetas de las aristas y según los nodos de origen se pueden obtener usando select sobre B_L y B, respectivamente. Luego, la mayoría de las operaciones que nos interesan se calculan usando rank y access sobre los índices de estas subsecciones.

5. Plan de Trabajo

1. Familiarizarse con las estructuras de la librería SDSL para implementar las estructuras base.
2. Implementar L y N usando wavelet trees.
3. Implementar B y B_L usando bitvectors.
4. Implementar el primer grupo de operaciones descritas en la Sección 4.
5. Implementar el segundo grupo de operaciones descritas en la Sección 4.
6. Evaluar la estructura con las bases de datos Lexvo.org y Wikidata.



6. Trabajo Adelantado

La primera parte del trabajo consistió en revisar la documentación de la librería SDSL y analizar los *Unit Tests* provistos por sus creadores.

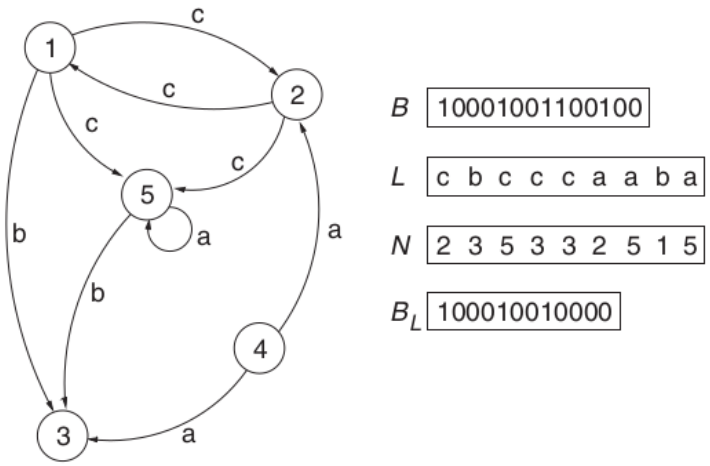


Figura 4: Grafo extraído de [21] utilizado en el código. Las etiquetas a, b y c corresponden a 1, 2 y 3 respectivamente en la data.

La segunda parte se centró en repasar conceptos básicos de C++11 por la falta de familiaridad con el lenguaje. Para esto se implementó un boceto de la indexación de [21] usando los arrays de C++ en lugar de las estructuras sucintas.

Las operaciones indegree, outdegree y la obtención de las etiquetas de los vecinos de un nodo se implementaron exitosamente. Dado que las funciones implementadas ocupan las

operaciones RSA como caja negra, este código podrá adaptarse fácilmente a la versión final con las estructuras sucintas.

En el repositorio [1] se encuentra el código con el ejemplo extraído del capítulo 9 de [21] (ver Figura 4).

Referencias

[1] `https://github.com/j-rparra/compactGraph`.

[2] *SPARQL 1.1 Query Language*. Informe técnico, W3C, 2013. `http://www.w3.org/TR/sparql11-query`.

[3] Akoglu, Leman, Mary Mcglohon y Christos Faloutsos: *Anomaly Detection in Large Graphs*. Mayo 2011.

[4] Ali, Waqas, Muhammad Saleem, Bin Yao, Aidan Hogan y Axel Cyrille Ngonga Ngomo: *A survey of RDF stores and SPARQL engines for querying knowledge graphs*. The VLDB Journal, 31, Noviembre 2021.

[5] Arroyuelo, Diego, Aidan Hogan, Gonzalo Navarro y Javiel Rojas-Ledesma: *Time- and Space-Efficient Regular Path Queries on Graphs*, Noviembre 2021.

[6] Blelloch, Guy E. y Arash Farzan: *Succinct Representations of Separable Graphs*. En Amir, Amihood y Laxmi Parida (editores): *Combinatorial Pattern Matching*, páginas 138–150, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg, ISBN 978-3-642-13509-5.

[7] Bonchi, Francesco, Aristides Gionis, Francesco Gullo y Antti Ukkonen: *Distance oracles in edge-labeled graphs*. Enero 2014.

[8] Clark, David: *Compact PAT trees*. Tesis de Doctorado, 1997. `http://hdl.handle.net/10012/64`.

[9] Claude, Francisco y Gonzalo Navarro: *The Wavelet Matrix*. En Calderón-Benavides, Liliana, Cristina González-Caro, Edgar Chávez y Nivio Ziviani (editores): *String Processing and Information Retrieval*, páginas 167–179, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg, ISBN 978-3-642-34109-0.

[10] Crochemore, Maxime y Renaud Vérin: *On compact directed acyclic word graphs*, páginas 192–211. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. `https://doi.org/10.1007/3-540-63246-8_12`.

[11] Cruz, Isabel F., Alberto O. Mendelzon y Peter T. Wood: *A Graphical Query Language Supporting Recursion*. SIGMOD Rec., 16(3):323–330, dec 1987, ISSN 0163-5808. `https://doi.org/10.1145/38714.38749`.

[12] Dey, Saumen, Victor Cuevas, Sven Köhler, Eric Gribkoff, Michael Wang y Bertram Ludäscher: *On implementing provenance-aware regular path queries with relational query*

engines. páginas 214–223, Marzo 2013.

[13] Gog, Simon, Timo Beller, Alistair Moffat y Matthias Petri: *From Theory to Practice: Plug and Play with Succinct Data Structures*. En *13th International Symposium on Experimental Algorithms, (SEA 2014)*, páginas 326–337, 2014.

[14] Grossi, Roberto, Ankur Gupta y Jeffrey Scott Vitter: *High-Order Entropy-Compressed Text Indexes*. SODA '03, página 841–850, USA, 2003. Society for Industrial and Applied Mathematics, ISBN 0898715385.

[15] Koschmieder, André y Ulf Leser: *Regular Path Queries on Large Graphs*. En Ailamaki, Anastasia y Shawn Bowers (editores): *Scientific and Statistical Database Management*, páginas 177–194, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg, ISBN 978-3-642-31235-9.

[16] Luo, Yongming, François Picalausa, George Fletcher, Jan Hidders y Stijn Vansummeren: *Storing and Indexing Massive RDF Datasets*. Semantic Search over the Web, páginas 31–60, Enero 2012.

[17] Mäkinen, Veli y Gonzalo Navarro: *Position-Restricted Substring Searching*. En *Latin American Symposium on Theoretical Informatics*, 2006.

[18] Mäkinen, Veli y Gonzalo Navarro: *Rank and select revisited and extended*. Theor. Comput. Sci., 387:332–347, 2007.

[19] Melo, Gerard de: *Lexvo.org: Language-Related Information for the Linguistic Linked Data Cloud*. Semantic Web, 6(4):393–400, August 2015.

[20] Munro, J. Ian: *Tables*. En *Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science*, página 37–42, Berlin, Heidelberg, 1996. Springer-Verlag, ISBN 3540620346.

[21] Navarro, Gonzalo: *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.

[22] Vrandečić, Denny y Markus Krötzsch: *Wikidata: A Free Collaborative Knowledgebase*. Commun. ACM, 57(10):78–85, sep 2014, ISSN 0001-0782. `https://doi.org/10.1145/2629489`.

[23] Wang, Xin, Junhu Wang y Zhang Xiaowang: *Efficient Distributed Regular Path Queries on RDF Graphs Using Partial Evaluation*. páginas 1933–1936, Octubre 2016.