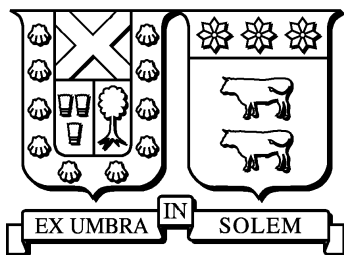


UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
SANTIAGO – CHILE



ÁRBOLES GENEALÓGICOS MEDIANTE LÓGICA DIFUSA Y BASES DE DATOS DE GRAFOS

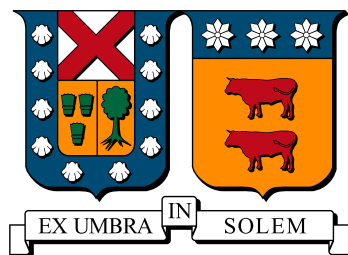
JUAN FELIPE AVALO MIRANDA

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL INFORMÁTICO

PROFESOR GUÍA: JOSE LUIS MARTÍ LARA

ENERO 2019

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
SANTIAGO – CHILE



ÁRBOLES GENEALÓGICOS MEDIANTE LÓGICA DIFUSA Y BASES DE DATOS DE GRAFOS

JUAN FELIPE AVALO MIRANDA

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL INFORMÁTICO

PROFESOR GUÍA: JOSE LUIS MARTÍ LARA
PROFESOR CORREFERENTE: CECILIA REYES COVARRUBIAS

ENERO 2019

MATERIAL DE REFERENCIA, SU USO NO INVOLUCRA RESPONSABILIDAD DEL AUTOR
O DE LA INSTITUCIÓN

Resumen

Una parte importante del estudio de la genealogía es la construcción de árboles genealógicos a partir de datos obtenidos desde diversos orígenes (registro civil, registros eclesiásticos, tradición familiar). Como provienen de fuentes heterogéneas es posible que detalles tales como nombres y apellidos tengan variantes que hacen más difícil una comparación directa. A su vez existe el problema de que los registros presenten errores de diversos tipos, tales como errores tipográficos, los cuales tienen como origen la naturaleza humana al momento de ingresar datos.

Para facilitar el trabajo del genealogista se diseñó e implementó una herramienta que usa lógica difusa para manejar la imprecisión inherente en los datos genealógicos, y una base de datos de grafo (*Neo4j*) para su almacenamiento y posterior análisis. Se presenta un algoritmo de cuatro pasos que separa la búsqueda de árboles en una comparación por nombres, por tiempos, por relaciones de grafo y una posterior unión de registros hecha con el propósito de encontrar distintos individuos.

Palabras Clave: bases de datos de grafos; lógica difusa; genealogía.

Abstract

An important part of genealogic studies is getting a family tree starting from many diverse sources (civil registry, ecclesiastical registries, family tradition). Because of their heterogeneous quality it's possible for registers like names and surnames to have variants that make a direct comparison difficult. There's also the problem of data with errors of different types, like typographical errors that arise from human nature when entering them.

To make the work of the genealogist easier a tool that uses fuzzy logic to manage the inherent imprecision in genealogical data, and uses a graph database (*Neo4j*) for their storage and analysis, will be designed and implemented. A four-step algorithm will be presented that separates the search for family trees in a name comparison, time comparison, graph relationship comparison and their following union designed to find different distinct individuals.

Keywords: graph databases; fuzzy logic; genealogy.

Índice de Contenidos

Resumen	III
Abstract	IV
Índice de Contenidos	v
Lista de Tablas	IX
Lista de Figuras	x
1. Definición del Problema	1
1.1. Sobre la Genealogía	1
1.2. Definición Inicial del Problema	2
1.3. Objetivos	3
1.3.1. Objetivos Generales	3
1.3.2. Objetivos Específicos	3
1.4. Alcances	4
2. Estado del Arte	5
2.1. Bases de Datos de Grafos	5
2.1.1. Neo4j	6

2.1.2.	OrientDB	7
2.1.3.	Titan	8
2.1.4.	Comparación de <i>BDG</i>	8
2.2.	Lógica Difusa	10
2.2.1.	Definición	10
2.2.2.	Conjuntos Difusos	11
2.2.3.	Funciones de Membresía	11
2.2.4.	Variable Difusa	14
2.2.5.	Reglas Difusas	15
2.2.6.	Sistemas de Lógica Difusa	15
2.2.7.	Aplicaciones	16
2.3.	Algoritmos de Grafos	16
2.3.1.	Caminos más cortos	16
2.3.2.	<i>Community Detection</i>	18
2.4.	Métricas de Semejanza de <i>Strings</i>	20
2.4.1.	Basados en caracteres	21
2.4.2.	Basados en <i>Tokens</i>	22
2.4.3.	Basados en Fonemas	22
2.5.	Técnicas de deduplicación	23
2.6.	Aplicaciones informáticas de genealogía	25
2.7.	Uso de bases de datos de grafos en la genealogía	26
3.	Diseño de la solución	28
3.1.	Metodología	28
3.1.1.	Registros usados	28
3.1.2.	Nodos y Relaciones	30

3.1.3.	Fundamentos del algoritmo	32
3.1.4.	Algoritmo General	33
3.2.	Arquitectura	36
3.2.1.	Base de datos	37
3.2.2.	Controlador	39
3.2.3.	Interfaz	40
3.3.	Criterios y Heurísticas	40
3.3.1.	Semejanza por Nombres	41
3.3.2.	Semejanza por Tiempos	45
3.3.3.	Semejanza por Relaciones de Grafo	49
3.4.	Visualizaciones	51
3.4.1.	Árbol Genealógico	53
3.4.2.	Búsqueda de Parientes	53
3.4.3.	Particiones	53
4.	Implantación y Validación de la Solución	57
4.1.	Hardware	57
4.2.	Conjunto de datos	58
4.3.	Metodología de las pruebas	58
4.4.	Primera prueba: Parámetros por defecto	59
4.4.1.	Pascual Basoalto	60
4.4.2.	Juan Antonio Duval	63
4.5.	Segunda prueba: No eliminar sin puntaje de grafo	63
4.5.1.	Pascual Basoalto	63
4.5.2.	Juan Antonio Duval	65
4.6.	Tercera Prueba: Usando <i>Jaro-Winkler</i> como comparación de <i>Strings</i> . .	69

4.6.1.	Pascual Basoalto con un <i>threshold</i> de 0.7	69
4.6.2.	Juan Antonio Duval con un <i>threshold</i> de 0.7	72
4.6.3.	<i>Threshold</i> en 0.8	72
4.7.	Prueba cuatro: Cambios a rangos de tiempo (<i>DEV</i>)	73
4.8.	Rendimiento de la base de datos	74
4.9.	Resumen de los resultados	75
5.	Conclusiones	76
	Bibliografía	79

Índice de tablas

2.1. Resumen de comparaciones de bases de datos.	10
3.1. <i>Inputs</i> para controlador difuso de nombres	42
3.2. <i>Outputs</i> para controlador difuso de nombres	43
3.3. Descripción de funciones de membresía	44
3.4. Tabla de reglas de nombres, con resultados de igualdad.	45
3.5. Reglas de grafo	52
4.1. Desglose de registros dentro del <i>dataset</i> de acuerdo al tipo de evento . .	58
4.2. Desglose de registros dentro del <i>dataset</i> de acuerdo al tipo de persona .	58
4.3. Desglose de registros de acuerdo al sexo de la persona	58

Índice de figuras

2.1. Gráfico de rankings de Bases de datos de grafos obtenidas desde [1] en el año 2018.	7
2.2. Función triangular con parámetros (0.2, 0.5, 0.8).	12
2.3. Función trapezoidal con parámetros (0.2, 0.3, 0.7, 0.8).	13
2.4. Función gaussiana con parámetros $\mu = 0.5$, $\sigma = 0.2$	13
2.5. Función con forma de S <i>smf</i> de skfuzzy	14
2.6. Ejemplo de camino más corto [2]	18
2.7. Ejemplo de detección de comunidades [3]	20
3.1. Ejemplo de registros modelados en <i>Neo4j</i> sin procesamiento de similitudes.	30
3.2. Diagrama del algoritmo general.	37
3.3. Arquitectura del sistema de análisis genealógico, usando un diagrama de componentes de <i>UML2</i> [4].	38
3.4. Gráfico para funciones de membresía para nombres.	43
3.5. Gráficos de las variables difusas <i>AVR</i> , <i>FNI</i> , <i>MDE</i>	49
3.6. Gráficos de las variables difusas <i>EAM</i> , <i>EAN</i> , <i>DEV</i> y <i>CoherenciaTemporal</i>	50

3.7. Variable difusa de padre. Las otras variables tienen la misma forma. . .	52
3.8. Imágenes de las tres formas de visualizar información de los resultados.	55
3.9. Imágenes de las tres formas de visualizar información de los resultados.	56
4.1. Vista general de los arboles genealógicos que contienen a “Pascual Basoalto”	60
4.2. Familia de Pascual y Bárbara Basoalto	61
4.3. Árbol genealógico centrado en Pascual Basoalto León	61
4.4. Partición de Pascual Basoalto León	62
4.5. Árbol que contiene a Juan Duval Salazar	64
4.6. Árboles que contienen a Juan Duval Canales y a Juan Duval Bernal . .	65
4.7. Vista general del árbol genealógico de Pascual Basoalto	66
4.8. Descendientes y ascendientes directos de Pascual Basoalto León	67
4.9. Árbol genealógico de José María Basoalto	68
4.10. Árbol genealógico de Juan Antonio Duval	69
4.11. Vista general de la familia Basoalto	70
4.12. Vista a la familia de Pascual Basoalto León	70
4.13. Vista a Pablo Basoalto	71
4.14. Vista a la partición de Pablo Basoalto	71
4.15. Vista a Juan Antonio Duval	72
4.16. Resultados de usar un <i>threshold</i> de 0.8	73
4.17. Árbol genealógico de Juan Antonio Duval	74

Capítulo 1

Definición del Problema

En el presente capítulo se explica cómo se define el problema a solucionar dentro de la memoria. Se incluyen una breve descripción de lo que es genealogía, la definición inicial en sí, los objetivos del trabajo y los alcances que se consideran dentro de esta memoria.

1.1. Sobre la Genealogía

El término *genealogía* hace referencia a la *ascendencia* y *descendencia* de una persona. Se puede referir al **grupo de personas** que forman parte de la familia de un sujeto, al **documento** que registra la genealogía de un individuo y a la disciplina que estudia la genealogía de una familia [5].

Para poder armar la genealogía de una familia chilena se pueden usar distintos tipos de fuentes. De las más importantes son las que provienen de las autoridades respectivas, tales como el registro civil, o antiguamente las parroquias católicas locales.

Un genealogista es el que investiga sobre genealogía. Entre sus labores usuales se encuentra la recopilación de información que después puede servir para armar una visión general de la familia que se investiga.

Existen distintas formas de mostrar la información genealógica. La más conocida es el *árbol genealógico*. Sin embargo dicha visualización funciona bien cuando no hay *consanguinidad* en la familia, esto es cuando dos parientes tienen ascendencia en común; en estos casos el árbol presenta ciclos, transformándose en una grafo, y la visualización se torna más compleja [6].

El interés por esta disciplina es algo que está presente desde siempre, ya sea por interés religioso o por curiosidad sobre las raíces de uno. Con el uso de herramientas informáticas la posibilidad de un aficionado de lograr obtener un árbol genealógico son mejores que nunca.

1.2. Definición Inicial del Problema

Uno de los problemas con los que tiene que lidiar un genealogista es saber qué tan precisa es la información que se tiene a mano.

Las fuentes de datos pueden ser de distinta procedencia. En particular, faltando datos del registro civil, se hace necesario recurrir a información proveniente de los registros de las iglesias donde guardaban las actas de matrimonio, defunción y bautizos, entre otros. Estos datos eran creados de acuerdo al criterio del sacerdote a cargo, por lo cual no poseen ningún tipo de estandarización.

Por otro lado, una vez armado el árbol podrían quedar cabos sueltos. Si, por ejemplo, de la madre de una persona solo se conoce su nombre, ¿puede ser posible encontrar si es miembro de otra rama de la familia?. Un genealogista tendría que hacer uso de su juicio para resolver esas preguntas, revisando todos los registros, tratando de buscar alguna posible relación con otra persona, lo cual es una tarea tediosa.

Con el uso de una base de datos de grafos, la búsqueda de relaciones se puede simplificar exponencialmente, permitiendo al genealogista buscar información de padres, abuelos, o primos en tercer grado, por ejemplo, de manera más simple y rápida.

Desafortunadamente, considerando lo heterogéneas que son las fuentes de datos hay

que tener un cierto grado de certeza al momento de declarar si una persona de un registro es la misma que la de otro registro completamente distinto.

Para esto, es deseable encontrar formas de automatizar la búsqueda de similitudes entre las distintas entidades descritas en los registros. De esa manera al momento de revisar posibles nuevos miembros de la familia, el genealogista podrá ver aquellos que tengan mayor probabilidad de ser familiares, y qué rol cumplían dentro de ellos.

Una vez formadas las conexiones entre los posibles miembros de las familias se debe poder encontrar relaciones que normalmente están ocultas al ojo humano, como por ejemplo poder ver si es que hay parentesco con alguna persona específica, o si hay algún patrón en el movimiento de los distintos miembros de la familia, entre otros análisis que son viables de realizar mediante un modelo de grafos.

Últimamente, el resultado esperado es tener una herramienta que permita al genealogista analizar datos demográficos de personas para obtener información de interés en forma sencilla.

1.3. Objetivos

1.3.1. Objetivos Generales

- Implementar un sistema de armado y análisis de árboles genealógicos a partir de datos no limpios y duplicados, mediante la implementación de un sistema de bases de datos de grafos, para apoyar el análisis eficiente y efectivo de información genealógica.

1.3.2. Objetivos Específicos

- Comparar distintas alternativas de algoritmos para la deduplicación efectiva de datos genealógicos representados mediante grafos, para poder decidir qué registros corresponden a la misma persona y cuáles no.

- Analizar los datos obtenidos mediante la generación de reportes, para encontrar información genealógica novedosa.
- Verificar empíricamente el rendimiento de una base de datos de grafos, determinando si su uso es factible para un conjunto de datos no limpios y duplicados de registros genealógicos.

1.4. Alcances

Los datos usados para este trabajo corresponden a registros parroquiales usados antes de la instauración del registro civil en 1885 en Chile. La información obtenida de dichas fuentes es heterogénea, dado que cada párroco escribía lo que él estimaba apropiado. Por esto solo se consideran algunos datos básicos de la persona, de acuerdo a la presencia en los *datasets* de prueba con los cuales se cuenta.

La memoria no se va a centrar específicamente en el formato de entrada de los archivos, siendo este un tema muy extenso para ser tratado en este trabajo. Por esta razón la entrada de datos va a ser rígida en comparación al potencial del sistema. Así mismo el enfoque no estará específicamente en la visualización de los árboles, ya que este aspecto es un área que todavía está en desarrollo [7] [8].

Por el origen de los datos, hay que recalcar que todo supuesto en este trabajo está justificado en alguna medida por las prácticas presentes en Chile en el siglo *XIX*. En particular los nombres de las personas siguen el estándar chileno: *Nombre*, *Apellido Paterno* y *Apellido Materno*.

Capítulo 2

Estado del Arte

En este capítulo se presentan antecedentes de las técnicas y conceptos que forman los cimientos del trabajo a nivel de teoría. Incluye elementos de bases de datos de grafos, lógica difusa y comparación de *strings*. También se analizan los avances existentes en las herramientas informáticas para genealogía.

2.1. Bases de Datos de Grafos

Las bases de datos de grafos (*BDG*) es un modelo de base de datos donde los distintos registros son almacenados usando estructuras basadas en la teoría de grafos, haciendo uso de aristas y vertices para representar entidades y sus relaciones respectivamente.

Las *BDG* han sido usadas en distintos contextos. Por ejemplo han resultado útiles para analizar datos de redes sociales [9], recomendaciones [9] y detección de fraudes [10], entre otros.

En general las *BDG* no necesariamente se adaptan a todos los casos de uso. Los usos más comunes de este tipo de modelo de base de datos son cuando hay información que se puede transformar cómodamente en relaciones y nodos. También funciona bien cuando la información que puede ingresar viene en varios formatos, ya que una *BDG*

es más flexible en la representación de datos que una base de datos relacional. En los contextos donde no funcionan tan bien son cuando se presenta un volumen alto de escrituras o se pretende hacer análisis con todo el conjunto de datos [11].

Los datos de un árbol genealógico se pueden modelar naturalmente como nodos y relaciones. Las bases de datos relacionales y NoSQL de tipo llave-valor, de columnas o de documentos modelan relaciones de tipo *ad-hoc* en forma deficiente [9]. En el caso de SQL se agrega complejidad cuando hay que unir distintas tablas, aumentándose la complejidad al momento de hacer consultas de distintas fuentes. Además para hacer *queries* recíprocas (Por ejemplo: Si A es amigo de B, ¿es B amigo de A?) se incurre en un costo computacional elevado. El mismo caso se da en los tipos de bases de datos NoSQL mencionadas anteriormente [9].

De acuerdo al ranking presentado en [1], *Neo4j* es la base de datos de grafos más popular en este momento. Le siguen *Microsoft Azure Cosmos DB*, *OrientDB* y *Titan*. De estas, *Azure* es comercial, y el resto es *open-source*.

A continuación se analizarán las tres primeras alternativas *open source*.

2.1.1. Neo4j

Neo4j [12] es una base de datos de grafos *open source*, implementada en *Java*. Como lenguaje de consultas usa *Cypher*, el cual es un lenguaje declarativo similar a *SQL*. Esto permite realizar consultas especificando qué resultados se quieren en vez de exigir la especificación de un algoritmo para obtener un resultado. Por otra parte, permite al programa hacer las optimizaciones que se estimen pertinentes, evitando que el desarrollador tenga que encargarse de ver cuál es la mejor forma de buscar los resultados dentro del grafo.

En casos donde se necesite más control, *Neo4j* tiene distintas *API* para lenguajes de programación populares, tales como *Java* [13] y *Python* [14], entre otras.

Neo4j viene en dos ediciones, ***Community*** y ***Enterprise***. La última es pagada y tiene

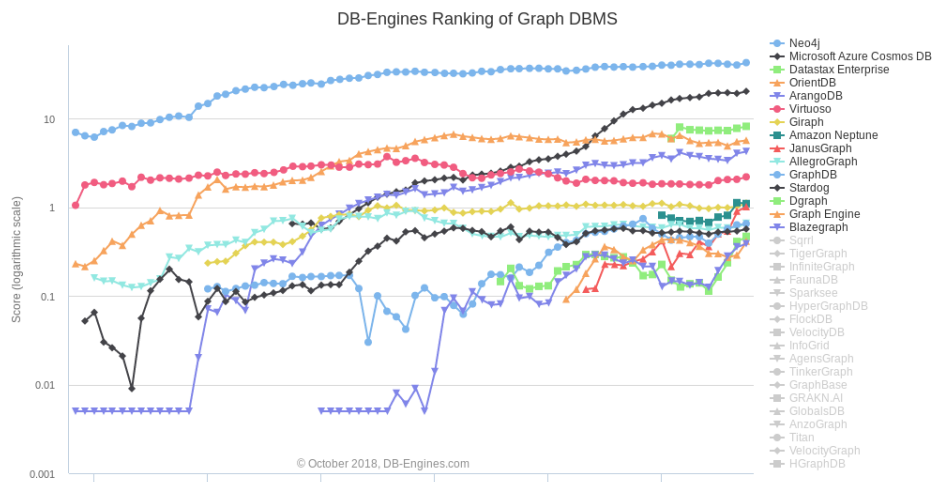


Figura 2.1: Gráfico de rankings de Bases de datos de grafos obtenidas desde [1] en el año 2018.

más capacidades, por ejemplo en el tamaño máximo de los grafos y en la capacidades para paralelización [15].

Soporta transacciones *ACID*.

2.1.2. OrientDB

OrientDB es una base de datos *NoSQL* multimodelo *open source*. Admite manejar modelos de datos de grafos, documentos, llave-valor y geoespacial entre otros [16], lo cual le da la flexibilidad de manejar diversos tipos de datos independiente de su forma.

Como lenguaje de consultas usa una variante propietaria de *SQL*. También permite comunicarse mediante el uso nativo de *Java*. Existe soporte para el concepto de clases de nodos y vértices, aceptando usar *herencia* para crear tipos de datos más especializados a partir de uno más general.

OrientDB tiene dos ediciones **Community** y **Enterprise**. De estas dos, *Enterprise* soporta más herramientas y tiene soporte para replicación, entre otros atributos [17]

Soporta transacciones *ACID*.

2.1.3. Titan

Titan [18] es una *BDG* optimizada para el manejo de grafos con una gran cantidad de vértices y nodos para alta concurrencia.

Tiene un sistema de *backends* de almacenamiento/persistencia. Por defecto hay acceso a almacenamiento en memoria *RAM*, *Apache Cassandra*, *Apache HBase* y *Oracle BerkeleyDB*. Para producción solo se recomiendan los tres últimos, ya que el almacenamiento en *RAM* no está optimizado [19].

Titan también permite interactuar con herramientas de análisis de *big data*, búsqueda de texto, y geolocalización, entre otros.

Como lenguaje de consulta solo soporta *Gremlin*, el cual es un lenguaje principalmente imperativo para recorrer grafos, y forma parte de *TinkerPop* [20].

2.1.4. Comparación de *BDG*

El tamaño de la comunidad se considera importante para este trabajo, ya que permite encontrar soluciones a problemas comunes en forma más práctica, además de tener mayor acceso a *plugins*. En este aspecto *Neo4j* es mejor al resto. Si se revisa la Figura 2.1 se puede apreciar que *Neo4j* tiene posiciones altas dentro del ranking durante un largo período de tiempo en comparación a sus competidores, lo que indica una mayor popularidad [21].

Buscando por *tags* de preguntas hechas sobre las distintas bases de datos dentro de *stackoverflow* [22] es posible encontrar una situación similar. Al momento de realizar este trabajo, la cantidad de preguntas sobre *Neo4j* es 14673, *OrientDB* tiene 2272 y *Titan* tiene 948.

Sobre los lenguajes de consulta que admiten las distintas *BDG* hay que considerar dos aristas. En primer lugar, la capacidad de realizar consultas en forma sencilla viene dada tanto por el paradigma de programación usado como por la sintaxis. En segundo lugar,

hay que considerar cuánto control admite el lenguaje.

En *Neo4j*, *Cypher* tiene la ventaja de ser de naturaleza declarativa, lo cual es ideal para consultas simples, o cuando se quiere probar algo sin preocuparse de que tan óptimo está escrita la consulta. La sintaxis es intuitiva, en el sentido de mostrar en una forma prácticamente gráfica cuales son los nodos y las relaciones con las cuáles se está trabajando.

OrientDB tiene una variante de *SQL* extendida para poder trabajar con la multitud de paradigmas que soporta que al igual que *Cypher* es declarativa. Tiene la diferencia de ser más similar al *SQL* tradicional, lo que permite una transición más rápida para alguien experimentado en dicho lenguaje. Pero como tiene que aceptar consultas a distintos tipos de datos, la sintaxis es algo más tersa en comparación al caso de *Cypher*.

Titan solo admite usar *Gremlin*, el cual es un lenguaje imperativo. Esto es ideal en casos donde se necesite recorrer un grafo en una forma particular, para motivos de optimización. Pierde la capacidad de resumir datos en forma fácil, lo cual lo hace no tan deseable cuando se desea explorar datos.

Las tres bases de datos anteriores permiten comunicarse con lenguajes de programación tales como *Java* y *Python*, permitiendo su uso con librerías, o aplicaciones ya hechas.

Titan es la única *BDG* de las analizadas que exige la integración con un *backend* de almacenamiento para persistencia de datos. El resto puede ser usada como la única base de datos dentro de un sistema, lo que las hace de configuración más fácil. Por otro lado, la configuración de *backends* por *plugins* hace a *Titan* más flexible al estar hecha para funcionar como parte de una infraestructura ya hecha. *OrientDB*, por otra parte, tiene la ventaja de poder reemplazar no solo al componente de grafos de un sistema, sino que también admite su uso para el almacenamiento de otro tipo de documentos.

En [23] se compara el rendimiento de diversas *BDG*, incluyendo las analizadas en este trabajo. Usa una batería de consultas de diversa índole, englobando operaciones de carga de datos, actualizaciones, selecciones y recorrido de grafos. En cuanto a rendimiento general, siendo este el tiempo total de todas las consultas propuestas por el paper, los

autores de [23] llegan a la conclusión de que *Neo4j* tiende a ser más rápido en gran parte de las consultas. Sobre *OrientDB*, esta también tiene un buen rendimiento siendo en algunos casos mejor que *Neo4j*, pero tiene deficiencias al momento de procesar grandes grupos de datos en memoria. *Titan* muestra un rendimiento inferior al de las dos opciones presentadas anteriormente.

Todas las alternativas tienen una versión gratuita y otra comercial.

Un resumen de las diferencias entre las bases de datos se presenta en la tabla 2.1

Tabla 2.1: Resumen de comparaciones de bases de datos.

	<i>Neo4j</i>	<i>OrientDB</i>	<i>Titan</i>
Ranking popularidad	1°	2°	3°
Modelo	Grafo	Multi-modelo	Grafo
Lenguaje de consultas	<i>Cypher</i>	<i>SQL</i> extendido	<i>Gremlin</i>
Tipo de lenguaje	Declarativo	Declarativo	Imperativo
API para distintos lenguajes	Sí	Sí	Sí
Persistencia de datos	Sí	Sí	Exige usar un <i>backend</i>
Rendimiento general	1°	2°	3°

2.2. Lógica Difusa

2.2.1. Definición

La *lógica difusa* es una rama de las matemáticas encargada de modelar cosas definidas con vaguedad. Es una de varias formas para tratar la incerteza [24]. Por ejemplo, términos como *frío* o *caliente* no pueden ser definidos con precisión, esto es, definir que a x grados está helado pero a $x + 1$ no, es algo que depende del criterio del que define

el término.

2.2.2. Conjuntos Difusos

Los elementos de un conjunto convencional solo pueden pertenecer o no al *set*. En cambio un conjunto difuso, o *fuzzy set* los elementos tienen un *grado de membresía* a dicho *set* el cual denota cuanto es lo que pertenece un valor al conjunto. Usualmente la membresía toma valores entre 0 y 1, donde 0 es que no pertenece al conjunto y 1 indica pertenencia total a él. Estos conjuntos admiten el uso de operaciones tales como *unión* e *intersección*, las cuales usualmente toman la forma del *máximo* y el *mínimo* de los valores en ambos *sets* [24]. También hay otras formas de definir dichas operaciones [25].

2.2.3. Funciones de Membresía

Para determinar en qué magnitud pertenece un elemento a un conjunto se usan *funciones de membresía*. Esta es una función que toma un valor real, usualmente conocido como *crisp*, y devuelve un valor *fuzzy* en el intervalo $[0, 1]$. Por ejemplo, la función que genera una correspondencia entre los distintos valores que puede tomar un termómetro y en qué medida ellos son miembros del conjunto difuso “Temperatura Alta” es una función de membresía. Dicha función puede tomar distintas formas, de las cuales se describen algunos ejemplos a continuación.

Función Triangular

Es una función compuesta de dos rectas, formando un triángulo en el gráfico. Usa tres parámetros (a, b, c) donde $a \leq b \leq c$. El punto b es donde está el grado de membresía más alto del conjunto. En $x \leq a$ y $x \geq c$ la función devuelve la mínima membresía. Un ejemplo se muestra en la figura 2.2

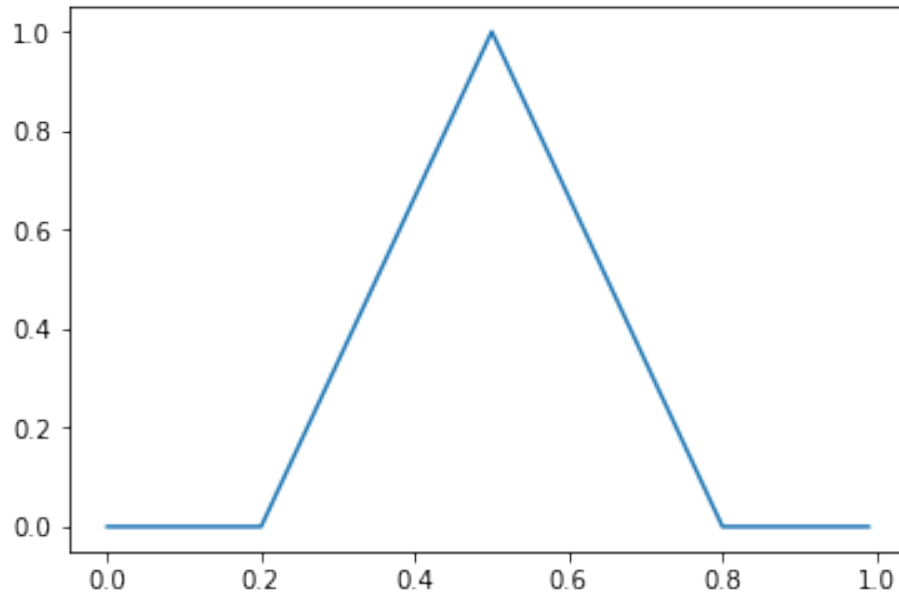


Figura 2.2: Función triangular con parámetros (0.2, 0.5, 0.8).

Función Trapezoidal

Es una función que toma cuatro puntos (a, b, c, d) , donde $a \leq b \leq c \leq d$, formando un trapecio. Entre b y c se encuentra la membresía máxima. En $x \leq a$ y $x \geq d$ se encuentran los mínimos. Esta función y la anterior son las más simples de calcular computacionalmente, y también las más fáciles de interpretar. Un ejemplo de la función se encuentra en la figura 2.3.

Función Gaussiana

Tiene forma de campana. Usa los parámetros (μ, σ) para la media y la desviación estándar, respectivamente. Tiene la ventaja de ser derivable e integrable en comparación con las funciones anteriores, lo que permite usar cálculo para encontrar sus parámetros [24]. También tiene la ventaja de ser una función conocida en el ámbito de la probabilidad y estadística [25]. Un gráfico de ejemplo se encuentra en la figura 2.4.

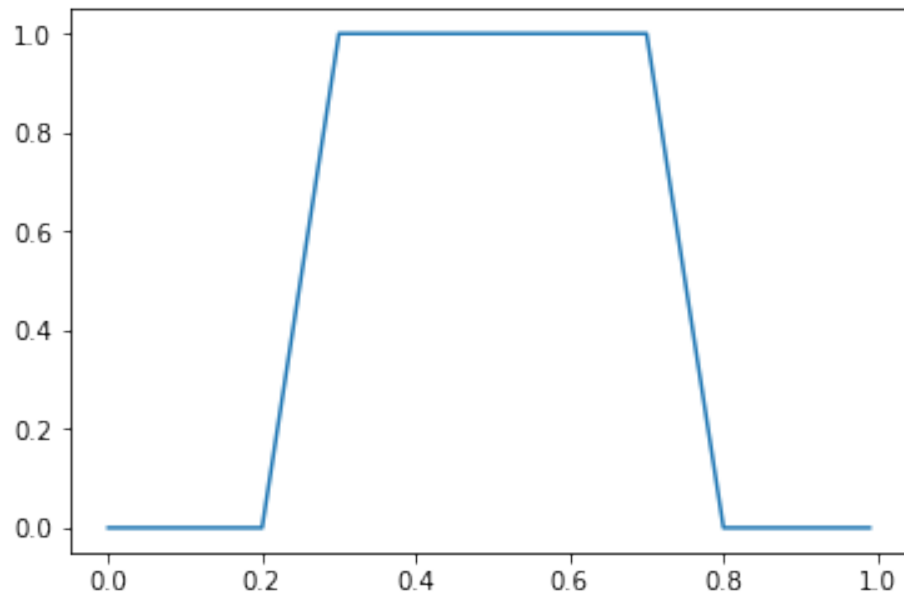


Figura 2.3: Función trapezoidal con parámetros (0.2, 0.3, 0.7, 0.8).

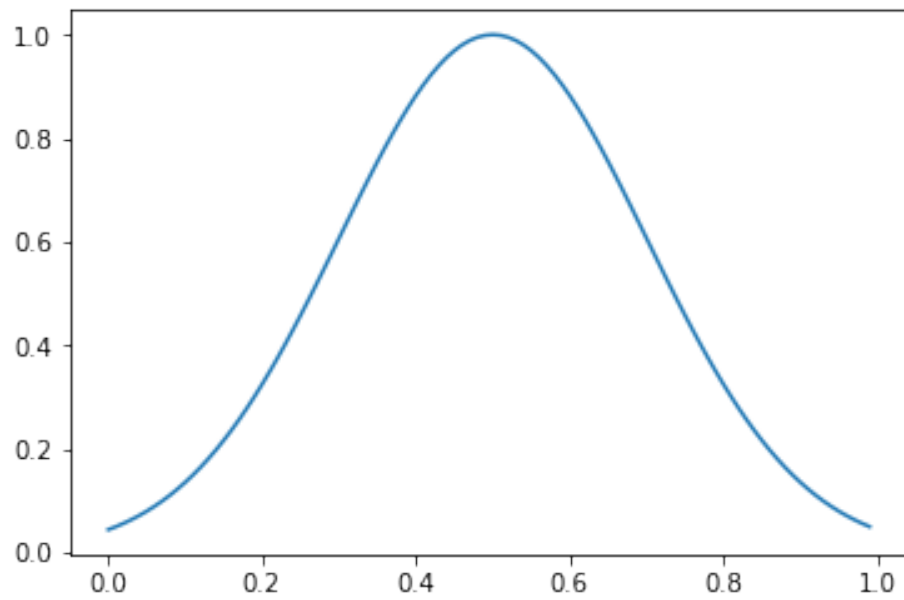


Figura 2.4: Función gaussiana con parámetros $\mu = 0.5$, $\sigma = 0.2$

Funciones con forma de S

Los *sets* anteriores se pueden considerar como *números fuzzy*, ya que cumplen con los requisitos de ser *normales*, *acotados* y *convexos* [24]. Hay ocasiones donde se quiere tener asimetría, en cuyo caso se pueden usar *sets* cuyas funciones de membresía se acerquen asintóticamente a algunos de los extremos. Entre sus variantes están las que usan funciones lineales, funciones cuadráticas y *sigmoides*. Un ejemplo se muestra en la figura 2.5.

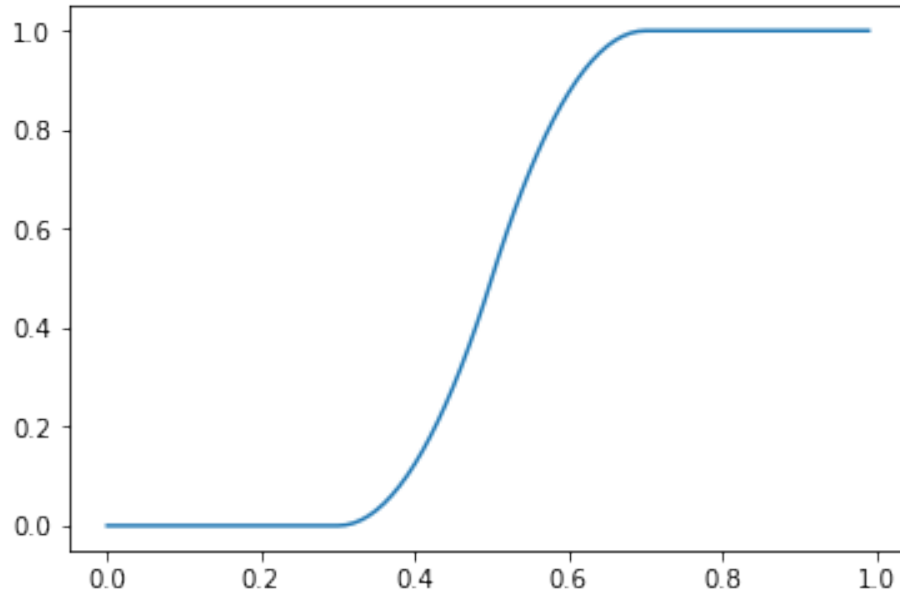


Figura 2.5: Función con forma de S *smf* de skfuzzy

2.2.4. Variable Difusa

También conocida como *variable lingüística*, es una variable que agrupa a un grupo de conjuntos difusos que están relacionados de alguna forma.

Esta forma de codificar información inicialmente propuesto por Zadeh [26] permite expresar un concepto X mediante distintos términos T , los cuales son etiquetas para sets difusos μ_T . Como ejemplo, una variable de nombre “Edad” puede estar definida

para que acepte los valores *joven*, *adulto* y *viejo*, los cuales a su vez son conjuntos difusos, cada uno con una función de membresía particular.

2.2.5. Reglas Difusas

Son reglas del tipo “**SI** x es A , **ENTONCES** y es B ”. Se puede usar para inferir información y actuar de acuerdo a ella. Está compuesta de *antecedentes* (“ x es A ”) y *consecuentes* (“ y es B ”).

Los antecedentes son *sets* difusos, valores de una variable lingüística o una combinación de ellos usando operadores lógicos (*AND*, *OR*, *NOT*) los cuales son equivalentes a operaciones de conjuntos.

Hay distintos tipos de consecuentes, pero el que se usará en esta memoria es el de tipo *Mamdani*, donde el consecuente también es un conjunto difuso.

2.2.6. Sistemas de Lógica Difusa

Cuando se tiene una lista de reglas con un mismo propósito surge un controlador difuso. En dicha lista se condensan las reglas que se quiere que el sistema siga. Son usualmente compilados con la ayuda de un experto en la materia que se quiere modelar [24].

Una vez que las reglas son activadas es necesario obtener un resultado desde el modelo *fuzzy* a un valor *crisp*, el que se terminará usando por el humano o sistema, mediante un proceso llamado *Defuzzificación*. Existen diversos métodos para obtener el valor final, los cuales incluyen *Centre of Gravity* y *Means of Maxima* entre otros.

Existen dos tipos de sistemas difusos dependiendo de los objetivos que se buscan: *Controladores Difusos* y *Sistemas de Razonamiento Difuso*. Un *Controlador Difuso* tiene como objetivo controlar a algún tipo de sistema tradicional, por lo tanto exige que tanto los *inputs* como los resultados sean valores *crisp* y no *sets* difusos ya que al sistema no le sirve como resultado “tal vez dos”. Un ejemplo de este tipo de controlador *fuzzy* es

un sistema de frenado que tome en cuenta la velocidad del vehículo y la temperatura de la rueda, dando como resultado el porcentaje de frenado que debe aplicar. Para un *Sistema de Razonamiento Difuso*, en cambio, lo que se quiere es obtener algún tipo de razonamiento o decisión entendible por un humano a partir de los *inputs*. En este caso el resultado puede o no puede ser el conjunto difuso que se obtuvo de los cálculos. Como estos dos sistemas tienen requisitos distintos se recomiendan distintos tipos de defuzzificación [27]. Por ejemplo los métodos de máximos funcionan mejor para sistemas de razonamiento, en cambio funciones tales como centroide son mejores para controladores difusos.

2.2.7. Aplicaciones

Los usos de la lógica difusa abarcan cosas tan mundanas como lavadoras y sistemas de freno anti-bloqueo, pasando por sistemas para diagnósticos médicos, predicciones meteorológicas y control de trenes subterráneos como el que se encuentra en Sendai, Japón [28]. Esto muestra la flexibilidad del paradigma *fuzzy*.

2.3. Algoritmos de Grafos

Existen distintos tipos de algoritmos que pueden ser aplicados a un grafo, los cuales se aprovechan de su estructura para la resolución de problemas específicos que surgen al tomar en cuenta la estructura que el grafo tiene.

Como hay bastantes situaciones que pueden ser reducidas a un grafo, se hace necesario conocer el tipo de preguntas que uno se puede hacer con respecto a él.

2.3.1. Caminos más cortos

El problema de encontrar el camino más corto entre dos nodos es una de las preguntas usuales que es posible hacer en un grafo, siendo en ocasiones un subpaso de algoritmos

más complejos. Consiste en, dados un nodo inicial y otro final, encontrar el camino tal que la suma de los pesos de sus vértices sean la menor posible. En casos donde el grafo no es etiquetado (los pesos son 1), es equivalente a encontrar un camino con la menor cantidad de saltos.

Los siguientes algoritmos son usados para calcular el camino más corto entre dos nodos:

- **Bellman-Ford:** este algoritmo parte asumiendo que hay una distancia infinita desde el nodo inicial a cada nodo. Luego, en cada paso va relajando la distancia de todos los vértices, hasta que al final obtiene los mejores caminos posibles. Una característica positiva que posee es que es capaz de entregar resultados cuando los pesos son negativos.
- **Dijkstra:** más rápido que Bellman-Ford [29][30], pero con la diferencia que no es capaz de lidiar con pesos negativos. Funciona bajo el concepto de nodos *visitados* y *no visitados*. A partir de la fuente el algoritmo revisa desde el *nodo actual* a todos los no visitados adyacentes, marcándolos con la menor distancia que encuentre. Luego escoge el nodo con la distancia más pequeña y lo asigna como nodo actual. Esto sigue hasta que no hayan más distancias a calcular o solo queden distancias infinitas.

Tiene distintas formas de implementarse, las que afectan el tiempo que toma en ejecutarse.

- **A*:** esta variante de Dijkstra [31], a diferencia de los anteriores, usa una heurística de la distancia más corta posible hacia el destino aparte de los pesos encontrados para calcular que camino tomar. Por ejemplo, en un mapa la aproximación sería tomar una línea recta hacia el destino. Se usa cuando se quiere encontrar una ruta esquivando obstáculos.

También existen algoritmos que calculan todos los caminos más cortos existentes entre todos los nodos. Algunos de ellos pueden usar los métodos anteriores como elemento para calcular distancias.

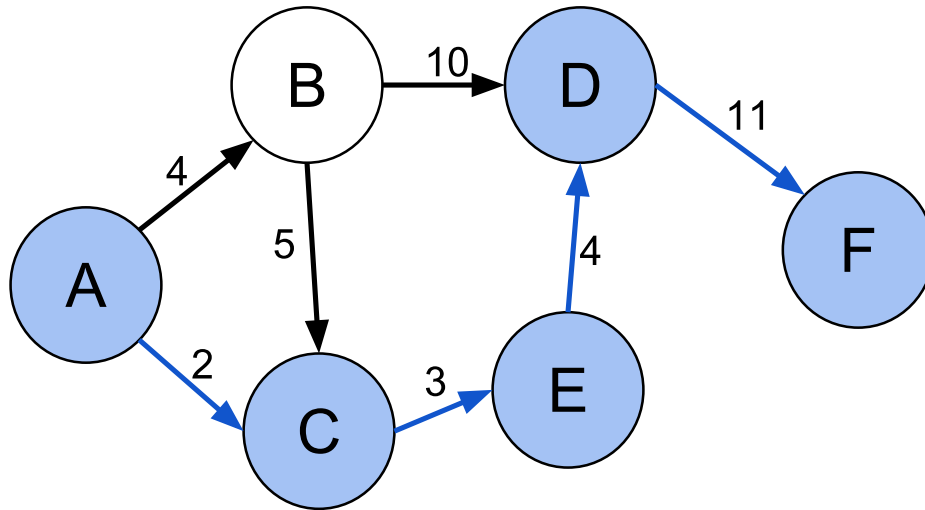


Figura 2.6: Ejemplo de camino más corto [2]

2.3.2. *Community Detection*

A medida que se van estudiando redes más complejas es posible ver que, dependiendo del grafo, sus elementos tienden a aglomerarse, formando grupos. En el contexto del estudio de grafos se conocen como *comunidad*. [32] Esto se puede visualizar mediante un ejemplo: una sala de clases con alumnos, los cuales tienen relaciones de amistad. En ese contexto se va a poder observar que no todos los estudiantes son amigos de todos, sino que hay grupos de amigos. A su vez, estos grupos no están completamente separados entre sí, porque un miembro de un *clique* puede tener amigos que no sean parte de su comunidad. Este tipo de problemas resulta útil no solo para redes sociales como la anterior, sino que puede ser usada para redes de interacciones de proteínas y citaciones bibliográficas, entre otros [33][32].

En sí mismo el problema de detección de comunidades no está definido en forma única y universal [32]. Fortunato and Hric hace notar que la ambigüedad de la definición ayuda en la búsqueda de nuevos métodos, pero también introduce ruido. A su vez destaca que no se necesita una definición universal para la ejecución de la mayoría de los métodos.

Los siguientes algoritmos de detección de comunidad son los usados por el equipo de Neo4j en su implementación [34]:

- **Componentes Conexos o *Union Find*:** este algoritmo busca si hay partes del grafo que están conectadas de alguna manera y con quien. Si el grafo está completamente conectado, lo que implica que cada nodo puede llegar a cualquier otro mediante sus conexiones, solo habrá un grupo. Si hay sectores que están desconectados se podrán encontrar más de una partición.

Union Find es usado para entender la estructura del grafo. Es importante en casos donde se exige que el grafo esté completamente conectado.

El equipo de Neo4j hace notar que una aplicación de este algoritmo está en la deduplicación en bases de datos.

- **Propagación de Etiquetas:** en este método cada nodo tiene una etiqueta y la propaga por la red. El mecanismo consiste en: por cada paso cada nodo adopta la etiqueta que está presente en la mayoría de sus vecinos.

Tiene la desventaja de que da distintos resultados cada vez que se ejecuta. Una forma de forzar un resultado único es marcar solo un subconjunto de los nodos.

- **Louvain:** maximiza un *coeficiente de modularidad* el cual mide qué tan densamente conectados están los nodos en comparación con una red arbitraria. Es uno de los algoritmos más rápidos que hay dentro de los que maximizan modularidad.

A pesar de esto, el método tiene ciertos problemas. Uno de ellos es que las soluciones que encuentra no son las intuitivas, sino que crea particiones más grandes de las que debería; la solución que uno esperaría al mirar el grafo se suele encontrar como una solución anterior no óptima. Otro problema es que hay exponencialmente muchas particiones con modularidades cercanas al óptimo, siendo este básicamente imposible de encontrar [32].

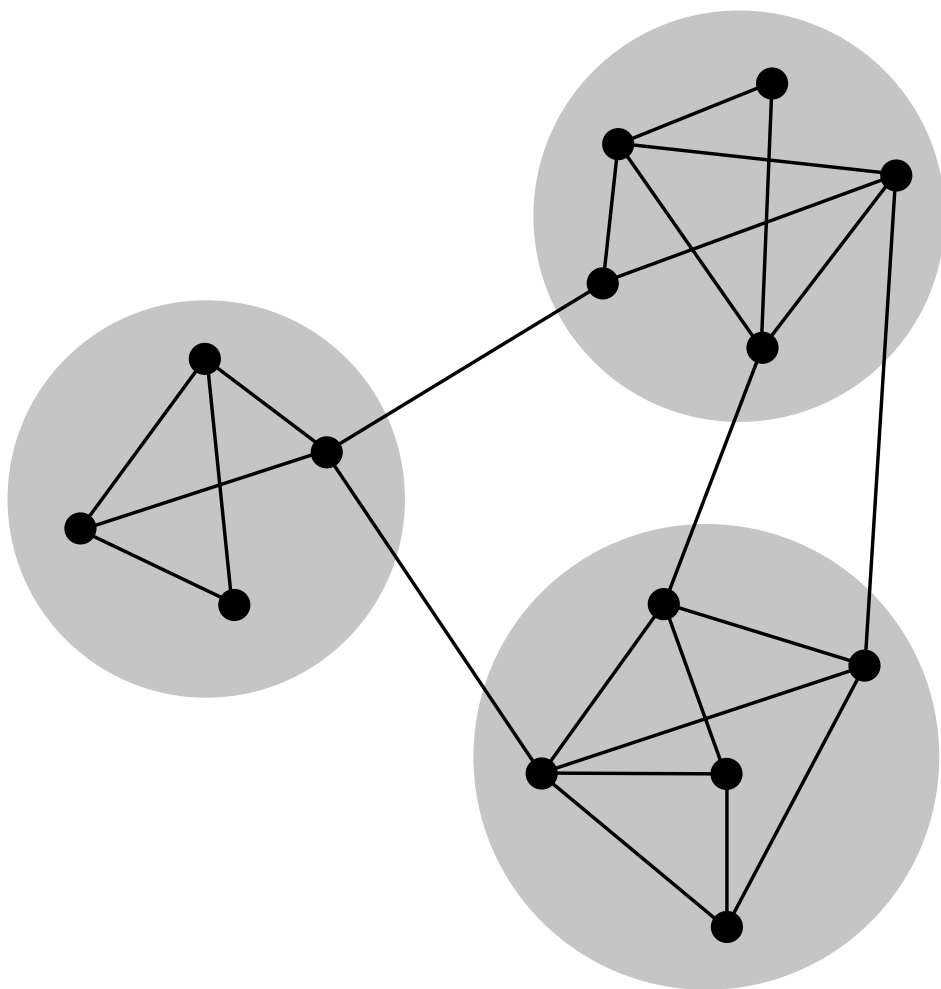


Figura 2.7: Ejemplo de detección de comunidades [3]

2.4. Métricas de Semejanza de *Strings*

Al momento de comparar datos tales como nombres y apellidos hay veces que no solo basta con indicar si son iguales. Por distintas razones es posible encontrarse con datos con errores tipográficos que fuerzan a relajar la medida de igualdad con la que se comparan dos strings. También existe el problema de la existencia de distintas representaciones para el mismo nombre. Por ejemplo, en el caso de un nombre de una persona chilena esta puede ser conocida por su:

- Primer nombre.
- Nombre completo.
- Segundo nombre.
- Apodos

En estos casos una comparación directa del string de nombre puede descartar a personas que en realidad son la misma.

En ausencia de una comparación de igualdad se pueden usar métricas de semejanza, las cuales dan un puntaje que indica qué tan similares son dos cadenas de caracteres. Existen distintas familias de métricas [35] dependiendo del fundamento sobre el cual se basa la comparación.

2.4.1. Basados en caracteres

Estos métodos comparan los caracteres de cada cadena buscando patrones que indiquen algún tipo de corrupción de alguno de ellos [36]. Uno de los ejemplos más estudiado es el de *distancia de edición* [36], que consiste en definir tres operaciones en el texto:

- Insertar un carácter.
- Modificar un carácter.
- Eliminar un carácter.

Cada una de ellas tiene un costo de 1. Entonces el puntaje indica cuántas de estas operaciones hay que hacer para obtener el primer string desde el segundo. Hay distintas variaciones de *distancia de edición*, en algunas se agregan o quitan operaciones (*Hamming*, *Affine Gap*, entre otros).

Hay otros métodos que comparan *strings* directamente que no usan los principios anteriores. *Jaro-Winkler* es uno de ellos; se usa principalmente para la comparación de

nombres y apellidos, ya que permite asignarle un peso mayor a prefijos similares dentro de dos *strings* [35].

Los algoritmos de esta familia son especialmente útiles cuando hay errores tipográficos [36][35].

2.4.2. Basados en *Tokens*

Si bien atacar el problema de comparar textos en la forma anterior sirve para detectar errores ortográficos, hay casos donde estos no son lo que se busca, especialmente cuando hay permutaciones en el orden de cada nombre ("John Smith", "Smith, John"). [35]

En métodos tales como *distancia coseno* o *jaccard* [37] lo que se hace es segmentar el texto en *tokens* y luego se calcula un puntaje de acuerdo a ellos.

Por ejemplo, *distancia coseno* [38] transforma los textos a un espacio vectorial y los compara mediante su producto punto, bajo la idea de que ambos nombres deberían apuntar en la misma dirección. *Jaccard*, en cambio, compara qué términos comparten ambas cadenas dentro del conjunto de todos los *tokens* únicos de los *strings* que se están comparando.

2.4.3. Basados en Fonemas

Los métodos anteriores funcionan bien cuando hay ambigüedades en la transcripción textual, pero les resulta imposible detectar casos donde hay palabras que suenan similar pero no se escriben igual. Algoritmos como *soundex* transforman el texto en una representación que agrupa consonantes que suenan similar [35]. Este algoritmo en particular está diseñado para apellidos de origen anglosajón, aunque funcionan bien con nombres provenientes de otros orígenes [35].

2.5. Técnicas de deduplicación

Las técnicas de deduplicación son importantes para este trabajo dada la heterogeneidad de los datos con los cuales se trabajarán, ya que no necesariamente se dispone de fuentes estandarizadas o estas pueden estar separadas geográfica y temporalmente.

La deduplicación es el proceso en el cual se encuentran múltiples registros referentes a la misma entidad [39].

De acuerdo a [39] la deduplicación de datos, o *Record Linkage*, consiste en cuatro pasos: *Limpieza y estandarización de datos*, *Indexación*, *Comparación de candidatos* y *Clasificación*. La *estandarización de datos* consiste en la transformación de los datos iniciales a una forma libre de inconsistencias, ya que es muy probable de que antes de este paso los datos estén con defectos de formato, o con campos dañados. La *indexación* es la búsqueda de registros candidatos a ser iguales. La *comparación de candidatos* se aplica a cada candidato con el resto de los registros dentro del grupo de candidatos a ser similares y les asigna a cada par un valor de similitud. La *clasificación* toma los valores de similitud y los clasifica en *similar*, *no similar* y *probablemente similar*.

Existen dos pasos conceptuales en la indexación: La *Construcción* y la *Recuperación*. En el primer paso se generan los valores bajo los cuales van a ser indexados cada registro: sus *BKV* (*Blocking key Values*); un *BKV* usualmente es la concatenación de los campos bajo los cuales se realizarán los índices. En el segundo paso se recuperan los registros de cada bloque de datos y se generan los pares que van a ser comparados [39].

Hay varias técnicas para construir los índices [39]. A continuación se mencionan algunas de ellas:

- **Blocking:** es el uso directo de los *BKV* para construir bloques no solapados (un miembro solo puede pertenecer a un único bloque). Tiene como características que la composición de los bloques depende únicamente de cómo esté definido el *BKV*, lo cual tiene como consecuencia que el tamaño de los bloques está relacionado con la frecuencia de cada *BKV*.

- **Sorted Neighborhood Indexing:** se ordenan los registros de acuerdo a su *BKV*, luego se crean los bloques de acuerdo a una ventana de un tamaño dado, la cual se va moviendo por la lista de registros. Existen distintas formas de ordenar los registros: alfabéticamente y por índice invertido. La última variante agrupa los registros con el mismo *BKV* en la misma posición de la ventana, similar al *Blocking*.
- **Q-Gram-Based Indexing:** se genera una lista de *Q-gramas* (*substrings* de tamaño q) para cada *BKV*. Estas listas se combinan de acuerdo a un valor de *threshold* $t \leq 1$. Cada una de estas combinaciones pasa a ser el índice bajo el cual los registros se agrupan. A diferencia de *Blocking* los registros pueden pertenecer a más de una lista.
- **Canopy Clustering:** se usa una forma de *clustering*, donde se comparan los registros mediante alguna medida de similitud tales como *Jaccard* o *TF-IDF* para obtener conjuntos superpuestos de registros (*Canopies*) con los cuales se pueden generar los bloques de datos. Para conseguir las *canopies*, partiendo de un conjunto de candidatos que contiene a todos los registros, se selecciona aleatoriamente un registro del conjunto de candidatos, el cual actúa como centro del conjunto. Luego se compara el centro con el resto de los elementos del conjunto de candidatos, seleccionando a aquellos que estén más cerca de una distancia d_i . De aquellos seleccionados, elimina del conjunto de candidatos a aquellos que estén más lejos de una distancia d_j .

Existen formas de comparación que son útiles con datos que tienen relaciones. En [40] se comparan conjuntos de autores y películas, buscando duplicados. Los nombres de cada uno de ellos pueden tener errores, pero por cómo se relacionan entre ellos es posible inferir cuándo son iguales. Si bien este caso es usado en datos con formato *xml*, dado que transforman los datos a un modelo de grafos, es posible usar dichos algoritmos para genealogía.

2.6. Aplicaciones informáticas de genealogía

Con la popularidad de la computación ha aparecido una gran variedad de herramientas que pueden ayudar a la investigación genealógica. No solo existen programas de uso general, tales como hojas de cálculo, editores de texto y editores de imágenes, que pueden asistir en la creación de un árbol genealógico, sino que también existen programas especializados.

En general, los software de genealogía permiten el ingreso de datos, creación y visualización de árboles. Como ejemplos de software con sus características se pueden incluir:

- **Legacy Family Tree** [41]: software con versiones gratuitas y de pago. Facilita revisar registros de bases de datos de genealogía. Admite la importación de archivos *GEDCOM*. Permite generar reportes de diversa índole, tales como familiares, individuales.
- **Family Tree Maker** [42]: software de pago con capacidad de sincronizar múltiples instancias del mismo árbol, mostrando posibles parientes mediante *hints*. Permite visualizar mapas de ancestros. Entrega reportes y gráficos del árbol familiar.
- **MyHeritage.com** [43]: plataforma web con capacidad de crear árboles genealógicos. Permite el cruce de datos con árboles de otras personas registradas dentro del sitio.

También existen bases de datos con registros vitales, tales como *ancestry.com* [44] y *familysearch.com* [45]. La cantidad de datos que tiene cada uno de ellos está en el orden de los billones [46][47]. La recolección de registros de los sitios puede estar vinculado a motivos religiosos, como en el caso de *familysearch* la cual está asociada a la iglesia mormona.

Históricamente se ha necesitado compartir información entre distintos investigadores.

Para esto existen estándares aceptados dentro de la industria de la genealogía. *GEDCOM* es un estándar creado por la iglesia mormona para compartir datos de genealogía [48]. Es un archivo de texto plano que contiene tres secciones (*header*, registros y *trailer*). Cada registro va en una línea. Los números al inicio de cada línea indican el nivel que tiene cada registro, partiendo desde cero el que es el nivel más alto. Este formato tiene debilidades, tales como la dificultad de modelar relaciones más complicadas entre las cuales están las adopciones [49]. Hay esfuerzos para crear estándares más robustos, entre los que se puede citar *GEDCOMX* [50].

2.7. Uso de bases de datos de grafos en la genealogía

Los siguientes ejemplos son instancias donde se han usado *BDG* dentro de un contexto de genealogía.

Kirby et al. compara el rendimiento entre *MariaDB* y *Neo4j*, bases de datos relacional y de grafos respectivamente, para un *dataset* de nacimientos, muertes y matrimonios [51]. La primera diferencia es que la misma consulta en *MariaDB* es mucho más larga en líneas de código que *Neo4j*, mostrando una desigualdad en la capacidad de expresión de los lenguajes de dichas bases de datos. La segunda disparidad es que el tiempo que demoró *Neo4j* en completar sus consultas fue inferior a *MariaDB*.

En [52] se analizan distintas bases de datos de grafos de la época, en el contexto del almacenamiento de árboles genealógicos. En este *paper* no solo se revisa el rendimiento de cada herramienta usada, sino que también características tales como usabilidad y mantenibilidad. A pesar de que hay aspectos que van cambiando en el tiempo, es destacable que ya se hayan hecho comparaciones de este tipo de bases de datos dentro del contexto de la genealogía.

Existen implementaciones de sistemas que cargan árboles desde archivos *GEDCOM* y generan reportes desde una interfaz web tales como las implementadas por Lozano Moya [53]. Los reportes se aprovechan de la estructura del grafo para encontrar parejas, padres, etc. Este trabajo se centra más en la implementación de una *API* para enviar

consultas al servidor.

En [54] se implementa la unión de árboles familiares mediante *fuzzy matching* y bases de datos de grafos. Se usó una combinación de *ElasticSearch*, la cual permite indexar y buscar texto natural, y *Neo4j* para la componente de grafos. Este problema se aplica cuando hay dos familias que en realidad son las mismas pero teniendo distintos miembros. Solo trabaja con núcleos familiares (padre, madre, hijos).

Efremova et al. busca realizar *entity resolution* sobre datos de interés genealógico provenientes de fuentes estructuradas y no estructuradas tales como documentos notariales [55]. Usan técnicas de *blocking* para reducir el espacio de búsqueda. Se enfoca en la extracción de datos tales como nombres y apellidos para poder decidir si dos entidades son iguales o no. Las técnicas usadas provienen del ámbito del *machine learning*, incluyendo el entrenamiento de un clasificador cuyos datos de entrenamiento son manualmente etiquetados. No usa relaciones de tipo padre/hijo.

Capítulo 3

Diseño de la solución

En este capítulo se explica el proceso usado para llegar a la solución planteada en el presente trabajo, incluyendo qué principios fueron usados como base y bajo qué criterios se escogieron los parámetros usados.

3.1. Metodología

En términos simples lo que se hizo fue tratar de unir lo más posible a las distintas entidades que conforman los registros, tomando en cuenta las características que hacen que dos registros sean similares o no. En estricto rigor existe la probabilidad de que dos personas sean unidas sin ser la misma o viceversa, especialmente cuando no hay datos más específicos. Por lo tanto, después de unir la mayor cantidad posible de registros se procede a discriminar con base en distintos atributos que pueden estar presentes, tales como edades o relaciones familiares.

3.1.1. Registros usados

Los datos usados durante el presente trabajo provienen de cuatro tipos de registros.

- **Bautizos:** representan el bautizo de una persona. En general incluyen el nombre completo del bautizado, los nombres de ambos padres, si se trata del bautizo de un hijo *legítimo* (ambos padres están casados) o *natural* (los padres no están casados) y los padrinos. También incluyen información con respecto a la fecha del bautizo y el lugar. A veces incluyen la edad de la persona a bautizar, ya que no siempre se cumple que se bautiza apenas se nace.
- **Matrimonios:** representan el matrimonio entre un novio y una novia. Tiende a incluir la información de los padres de ambos, y de los testigos de matrimonio y padrinos de los novios. En el caso de los cónyuges en sí, se incluye los nombres completos, edades, categoría (natural o legítimo), entre otros.
- **Defunciones:** representan el entierro de una persona. Suelen incluir el nombre de quien falleció y la fecha de entierro. A veces incluyen a los padres del difunto, y de estar casado, él o la cónyuge. En algunos casos también incluyen la edad de quien falleció.
- **Herencias:** en estos registros la información que se puede obtener es la de los herederos y su relación con quien deja la herencia. En algunos casos incluye cónyuges de los herederos y nombres de sus padres.

Cabe recalcar que, por la naturaleza no estandarizada de los registros eclesiásticos, existe la posibilidad de omisiones de datos importantes. Por ejemplo, es posible que alguno de los padres de la persona bautizada no aparezca por criterio del sacerdote que ingresó el acta.

Para poder usar estos registros se hace necesario transformarlos a un formato que pueda ser leído por la *BDG*. Los primeros tres tipos de registros son introducidos al sistema mediante la lectura de los archivos del *dataset*, los cuales están en formato *csv*. Los documentos de herencia fueron ingresadas directamente en *Cypher* dada la mayor complejidad de estos.

3.1.2. Nodos y Relaciones

Para poder modelar las entidades fundamentales de los registros anteriores es necesario recordar dos cosas de como *Neo4j* define a sus nodos y relaciones. En primer lugar cada nodo puede tener etiquetas (*labels*) y propiedades. La etiquetas son una forma de la *BDG* para “asignar roles o tipos a los nodos” [56]. Se pueden entender como distintos conjuntos a los cuales una entidad puede pertenecer.

Las entidades fundamentales para trabajar en genealogía son las personas. Para modelar a un individuo se usan una combinación de múltiples etiquetas. Se toma en cuenta tanto características que posee un sujeto así como eventos a los cuales asisten.

Los distintos *labels* pueden ser múltiples dentro de un mismo nodo. En el contexto en el cual se está trabajando se consideran los siguientes roles:

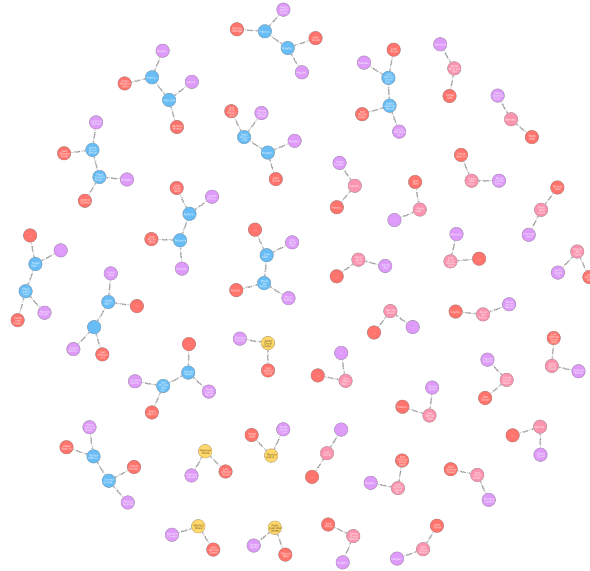


Figura 3.1: Ejemplo de registros modelados en *Neo4j* sin procesamiento de similitudes.

Personas

- **Persona:** cualquier nodo que trate sobre algún individuo recibe esta etiqueta.
Un nodo de tipo persona tiene como atributos:

- Nombres
 - Apellido paterno
 - Apellido materno
 - Fecha de nacimiento
 - Fecha de defunción
 - Sexo
 - Estado civil
 - Categoría
- **Padre:** es cualquier nodo persona que tenga algún hijo.
 - **Bautizado:** en el caso de que el nodo hable de alguien de quien se esté registrando un bautizo. Incluye como atributo el lugar y la fecha de la ceremonia.
 - **Casado:** se aplica esta etiqueta a alguno de los dos cónyuges de un matrimonio.
 - **Difunto:** se usa en el caso de que una persona sea quien esté siendo catalogado como difunto.
 - **Padrino/Testigo:** en donde una persona asiste a alguno de los eventos en la calidad de padrino o testigo respectivamente.

Eventos

- **Evento:** representa un suceso que ocurrió en la vida de una persona.
- **Matrimonio:** representa un matrimonio al que asistió un conjunto de personas.
- **Defunción:** representa a nodos que tuvieron que ver con una defunción.
- **Bautizo:** representa a nodos que estuvieron involucrados con un bautizo.

Relaciones

Para poder dar estructura al modelo, los nodos se deben relacionar entre si. Se consideran como tipos de enlaces aquellos que se generan directamente de los registros, así como aquellos que se pueden inferir a partir de la información que se tiene.

- **Padre/Hijo:** relación unidireccional entre dos personas que representa una relación de parentesco. Aparecen explícitamente en los registros trabajados.
- **Casado:** relación entre una persona y un nodo matrimonio indicando que el individuo es el novio o la novia en una boda.
- **Similitud:** relación entre dos personas que indica que existe una posibilidad de que sean iguales. Que tan parecidos son dependerán de las propiedades dentro de la relación.

3.1.3. Fundamentos del algoritmo

En principio cada registro posee entidades que en rigor que no son las mismas. Es común encontrar personas con el mismo primer nombre y apellido, pero que son completamente distintas, como es el caso de un hijo cuyo nombre sea en honor al abuelo. Otro problema presente es que no existe algún identificador único para una persona, como ocurre actualmente con el *RUT*.

Para poder encontrar una estructura genealógica, hay que encontrar formas en las cuales los distintos sujetos se pueden parecer o diferenciar. Los criterios de similitud que se pueden usar dependen mucho del criterio del genealogista. Para tales efectos se pueden comparar nombres, qué padres tienen en común, qué hijos, en qué fechas vivieron, incluso los lugares geográficos donde vivieron.

En términos generales para encontrar relaciones genealógicas durante este trabajo se privilegió un enfoque *top-down* en la construcción del programa. Primero se busca unir

en la mayor forma posible a los distintos individuos mediante alguna medida de similitud, dejando en principio grupos compuestos de varios nodos, en una forma similar al *blocking* de deduplicación. Luego se van eliminando relaciones de similitud dependiendo de algún criterio establecido en el sistema acorde a las necesidades del genealogista. Finalmente, con las similitudes restantes se unen aquellos que son candidatos a ser la misma persona.

3.1.4. Algoritmo General

Se establece un algoritmo para detectar quiénes dentro de la base de datos son candidatos a ser similares, y por ende, poder juntar cabos sueltos de un árbol genealógico. El problema principal que se presenta durante la realización del trabajo es que sin establecer algún criterio se termina comparando todos los nodos persona entre sí, lo cual puede resultar intensivo y caro computacionalmente.

Con el objetivo de evitar hacer comparaciones complejas en un número muy grande de datos se establecen cuatro pasos. Los primeros pasos clasifican información más fácil de obtener, de tal forma que los subsiguientes tengan que procesar menos pares de personas.

Por otra parte, cada comparación genera una conexión de similitud entre dos personas. En el grafo se forma una red de personas similares, donde mientras más conexos son los nodos más similares son entre sí. Por esto el algoritmo también crea una red lo más grande posible, para evitar que se pierdan datos interesantes para el genealogista. Después los pasos siguientes van podando el conjunto hasta dejar solo aquellos que de verdad tengan posibilidades de ser similares.

El algoritmo en si consiste en cuatro pasos, los cuales se explican a continuación.

Comparar Nombres

Esta parte consiste en tomar pares que aún no han sido comparados de ninguna forma y ver si dos personas tienen nombres y apellidos similares. Mediante el uso de métricas de distancia de *strings* se comparan los nombres y cada apellido por separado. De esta forma para cada parte de un nombre se tiene un puntaje, el cual mediante la ayuda de un clasificador de lógica difusa determina si se parecen o no.

Terminado este paso queda una red de similitud con todos los candidatos a ser similares. Dependiendo de lo estricto que sea la comparación se espera que haya menos pares si se usan métricas o clasificaciones más estrictas.

Como métricas de semejanzas de nombres se usaron:

- *Overlap Coefficient*
- *Jaccard*
- *Jaro Winkler*
- *Editex*

Para las métricas que usan *tokens* se permite usar los siguientes *tokenizadores*:

- *Por Q-gramas*
- *Alfanumérico*
- *Sin tokenizar*

Comparar Tiempos

Después de obtenerse los pares candidatos se procede a comparar si es posible que ambas personas hayan vivido en el mismo lugar. En algunos casos los registros llevan asociados fechas, las cuales pueden ser aprovechadas para crear una línea temporal.

Para que las personas de un registro se puedan considerar como similares, estas deben calzar dentro de un segmento de tiempo que sea razonable.

La construcción de la línea de tiempo considera las siguientes fechas importantes:

- **Fecha de nacimiento**
- **Fecha de defunción**
- **Fecha de evento genérico**

En general las dos primeras no son explícitas dentro de los registros usados para probar el programa, pero pueden ser calculadas a partir de las fechas de bautizos y de entierros. Dependiendo de si dentro de las actas incluyen las edades donde se realizó uno de los eventos vitales (bautizo, entierro) es posible calcular una fecha más o menos exacta de las fechas donde realmente vivió una persona.

Las fechas de evento genérico, usualmente matrimonios aunque también pueden incluir herencias, tienen que ir sí o sí dentro de los límites de la línea de tiempo. De no existir límites en la línea se asume que las fechas genéricas a lo menos están a una distancia temporal designada por el genealogista.

Para determinar qué casos son *coherentes temporalmente* se usa un controlador difuso. Si dos personas no son coherentes se elimina la relación de similitud entre ellos, reduciéndose el espacio de búsqueda para el siguiente paso.

Comparar Relaciones del Grafo

En el momento de terminar el paso de comparar tiempos se podría juntar personas, pero no hay garantías de que se preserve una estructura familiar correcta. En otras palabras existe la posibilidad de que, por ejemplo, un individuo tenga dos pares de padres y madres, lo cual no es realista.

Para poder determinar qué pares se pueden conservar, se procede a comparar los parecidos que tienen en:

- Cónyuges
- Padres
- Hijos

Con los puntajes obtenidos a partir de las similitudes se clasifica usando un controlador *fuzzy*, determinando si se debe conservar o eliminar el par.

En ciertas ocasiones puede no haber forma de comparar usando el grafo, ya que ninguna de las dos personas tiene padres conocidos, por ejemplo. En estos casos se puede determinar si se quieren mantener o eliminar dependiendo de lo que se busca.

Detectar Personas

Con los pasos hechos se logra acotar lo más posible la red de similitud, pero aún no se puede hablar de una sola persona. Hay que determinar qué nodos realmente son los de la misma persona. En este paso se aprovecha de la estructura de las conexiones entre personas similares, asumiendo que una red más densamente conectada indica a una persona.

Con el propósito de encontrar patrones que sugieren una mayor posibilidad de detectar a una persona, se usan algoritmos de detección de comunidad. En principio solo es necesario detectar los componentes conexos dentro del grafo de similitudes, ya que personas que ni siquiera comparten nombres deberían estar completamente desconectadas las unas de las otras. En la práctica pueden surgir estructuras de comunidad, donde si no se hace la distinción, se unen personas que se parecen en algo, pero en realidad son demasiado distintas para estarlo.

3.2. Arquitectura

En esta sección se detalla cómo se estructuró el sistema en términos de arquitectura informática. Responde a la necesidad de establecer un diseño a nivel general de qué

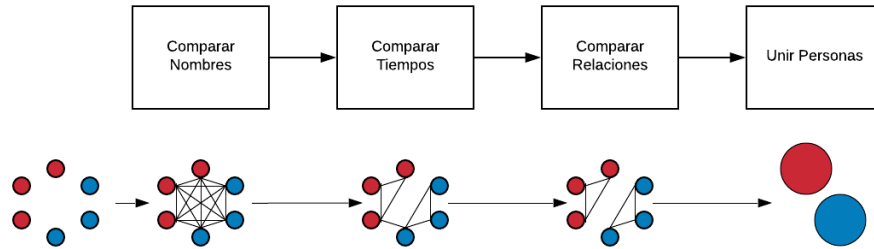


Figura 3.2: Diagrama del algoritmo general.

componentes debe contener el programa y cómo se relacionan entre sí.

El sistema para análisis de datos genealógicos fue separado en 3 partes, las cuales fueron diseñadas con el fin de minimizar la cantidad de cambios necesarios en el caso de que se decida cambiar de tecnología o de técnica usada.

El diseño está basado en una arquitectura de capas [57], donde cada uno de los módulos se comunica solo con el que está arriba o abajo en el diagrama, el cual se puede visualizar en la figura 3.3.

3.2.1. Base de datos

La capa de base de datos guarda toda la información de las entidades que conforman el sistema genealógico junto con sus relaciones. Se encarga de entregar los datos necesarios para la ejecución de los distintos pasos de las capas superiores. Además guarda sus resultados para posterior uso. Hay algoritmos de grafos que pueden ser ejecutados a este nivel, ya que la *BDG* permite una ejecución de ellos más optimizada.

Para este componente se escogió *Neo4j* ya que posee la comunidad más activa, además de tener un lenguaje de consultas fácil de entender. En conjunto con la base de datos en sí se escogieron dos *plugins*:

- **APOC**: es una biblioteca de procedimientos auxiliares para *Neo4j*. Contiene funciones que ayudan a simplificar algunas consultas, además de utilidades para

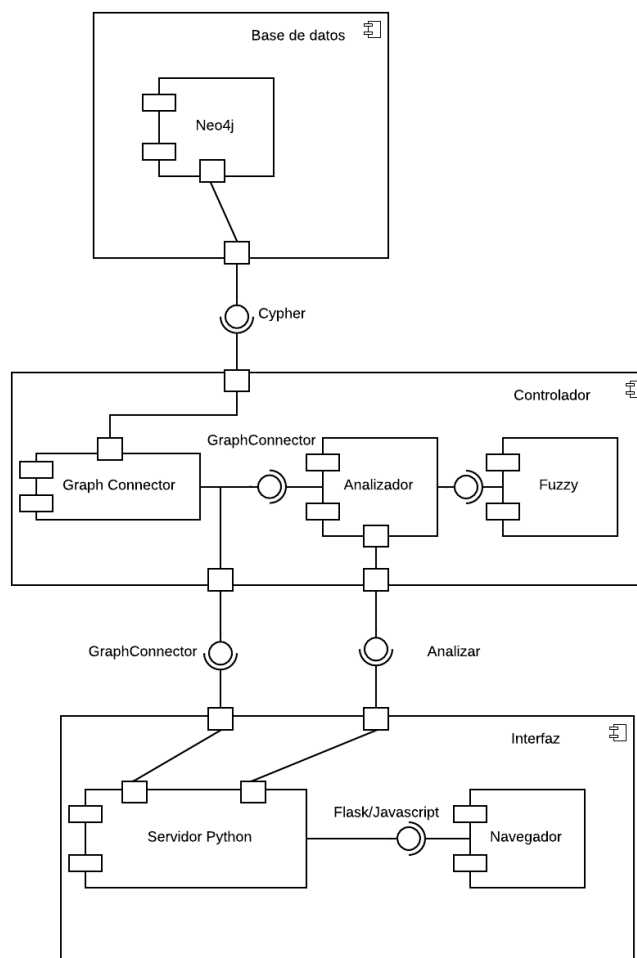


Figura 3.3: Arquitectura del sistema de análisis genealógico, usando un diagrama de componentes de *UML2* [4].

la refactorización de grafos y agregación de nodos, entre otros [58].

- **Graph Algorithms:** biblioteca centrada exclusivamente en la implementación y ejecución de algoritmos de grafos tales como *Louvain* o *componentes conexos* [59].

3.2.2. Controlador

Este subsistema se encarga de ejecutar el algoritmo de cuatro pasos para detectar personas en un contexto genealógico. Obtiene los registros de la capa de base de datos. Entrega los resultados del procesamiento a la capa de interfaz.

La capa de control, a su vez, tiene componentes que realizan distintas funciones dentro del algoritmo:

- Obtiene los resultados de las distintas consultas hechas a la *BDG* y las presenta al resto del programa.
- Modela una persona y sus relaciones a partir de los datos de la *BDG* para facilitar su uso.
- Se encarga de definir los distintos controladores difusos con sus reglas, además de controlar su ejecución.
- Ejecuta los cuatro pasos de análisis, delegando aspectos a los distintos componentes de ser necesario.

Se usó *Python* como lenguaje de desarrollo principal para esta parte. Se argumenta la mayor experiencia que se tiene en su uso, así como la existencia de múltiples bibliotecas para facilitar un desarrollo más simple en lo que es programación. Entre los paquetes escogidos se destacan:

- *Scikit-fuzzy*: provee de herramientas para lógica difusa, incluyendo herramientas para controladores lógicos estilo *Mamdani* junto con todas las operaciones necesarias para crear variables difusas, entre otras [60]

- *Py2neo*: paquete que funciona como nexo entre *Neo4j* y *Python* [14].
- *Py-stringmatching*: para poder realizar comparaciones de strings tales como *Jaro-Winkler*, además de soporte para algoritmos de *tokenización* [61].

3.2.3. Interfaz

En esta capa se recibe la información calculada en las capas anteriores y se muestra al usuario final. También transmite las instrucciones del genealogista al controlador. Permite configurar los parámetros que se van a usar para la ejecución de los algoritmos y para las visualizaciones correspondientes.

La interfaz se hizo pensando para que funcione en un navegador web. Para desarrollarla se usa *Flask* [62], el cual es un *microframework* para *Python* para la conexión entre el controlador y el navegador. Para las páginas en sí se ocupa una mezcla de *Jinja2*, el cual es el lenguaje para *templates* de *Flask*, más *HTML* y *Javascript*. Se emplea *Vis.js*, una biblioteca para *Javascript* para la visualización de los grafos.

3.3. Criterios y Heurísticas

En esta sección se introducen las distintas reglas y heurísticas usadas por los pasos del algoritmo. ¿Por qué se habla de heurística? En el ámbito de la genealogía es usual tener que usar la intuición para poder encontrar posibles enlaces. Dado que los argumentos comúnmente dados, tales como "nació más o menos cerca", o "tiene un tío que vivía en esa área, así que puede que sean parientes de algún tipo", son en esencia imprecisos es necesario partir por la premisa de que los resultados no van a ser el 100 % correctos. También se hace necesario modelar el criterio que usaría un genealogista para que un computador pueda calcular la información requerida perdiendo la menor cantidad de posibilidades dentro de lo factible. Con este fin se usaron controladores difusos para definir una serie de reglas que ayudan a encontrar una genealogía en una forma similar a la que encontraría un humano, tomando en cuenta que los datos entrantes no

son completamente confiables desde un inicio, y encontrando árboles que sean lo más certeros posibles, tomando en cuenta las restricciones a las cuales son sometidas.

Se debe hacer hincapié en que las reglas son producto de un razonamiento humano basado en la experiencia y no de resultados optimizados. Por lo tanto es imposible listar cada una de las excepciones posibles que no calzan a las reglas. Un ejemplo siempre presente es que cuando hay consanguinidad se forman ciclos en el "árbol", lo cual ha sido un dolor de cabeza para más de algún desarrollador de software de genealogía al momento de establecer condiciones mínimas en el sistema.

3.3.1. Semejanza por Nombres

Un camino para establecer si dos personas se parecen es ver si sus nombres son parecidos o no. Una solución sencilla es comparar directamente el nombre completo de ambos datos, sin embargo tiene las siguientes desventajas:

- Es muy susceptible a errores tipográficos, dado que los datos usados provienen de fuentes donde los nombres eran ingresados manualmente, lo cual hace que este tipo de casos sea común. De hecho existen casos donde hay variantes de apellidos introducidos gracias a un cambio en algún acta de nacimiento, por ejemplo.
- Si bien el nombre completo de una persona está compuesto por uno o más nombres de pila, un apellido paterno y un apellido materno, en la documentación de la época lo normal es solo escribir uno de los nombres bajo el cuál se conoce a la persona, más el apellido paterno. Una comparación directa entre un nombre completo y uno simplificado necesariamente dirá que no se parecen, lo cual hace que se pierda información de las relaciones.
- Es normal que en la documentación asociada solo se encuentre uno de los nombres de pila, siendo este el nombre con el cual se conocía dentro de su comunidad. Eso puede significar problemas porque es tradicional nombrar a uno de los hijos en honor a santos, abuelos, etc, causando que existan registros donde una persona se llame igual que otra pero sean padre e hijo. Otra posibilidad que causa esta

tendencia es que un individuo sea conocido por su segundo nombre durante toda su vida, pero no se consiga relacionar con un acta de nacimiento porque esta trae el nombre completo.

Para poder determinar una semejanza por nombres se deben tomar en cuenta problemas como los anteriores. Para un análisis más fácil no se usa el nombre completo, sino que se separa en **nombres**, **apellido paterno** y **apellido materno**. Al momento de comparar dos registros se usa una métrica de semejanza de *strings* (ver sección 2.4) para los pares *nombre* —*nombre*, *apellido paterno* —*apellido paterno* y *apellido materno*—*apellido materno*, obteniendo un puntaje entre 0 y 1 para cada par. Luego cada puntaje se usa como *input* en un controlador difuso.

Controlador Difuso

Consiste en tres *inputs* y dos *outputs* relevantes. Todos comparten el mismo estilo de función de membresía.

Tabla 3.1: *Inputs* para controlador difuso de nombres

Nombre de Variable	Descripción	Rango
<i>Nombres</i>	Cuantifica la semejanza entre dos nombres de pila. Considera como una entidad a comparar casos como "Pedro" y "Pedro Pablo"	$[0, 1]$
<i>ApellidoPaterno</i>	Semejanza entre dos apellidos paternos.	$[0, 1]$
<i>ApellidoMaterno</i>	Semejanza entre dos apellidos maternos.	$[0, 1]$

Funciones de membresía: todas las variables de nombres usan las mismas funciones de membresía.

Tabla 3.2: *Outputs* para controlador difuso de nombres

Nombre de Variable	Descripción	Rango
<i>Iguales</i>	Si, dada la combinación de nombres y apellidos, dos personas podrían ser iguales	$[0, 1]$
<i>Parentesco</i>	Si, dada la combinación de apellidos, dos personas pueden ser parientes	$[0, 1]$

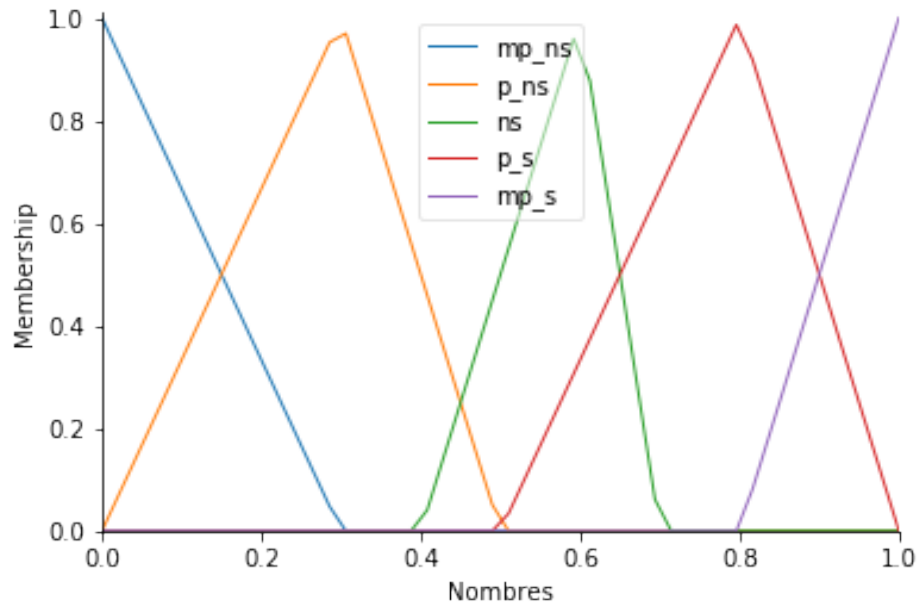


Figura 3.4: Gráfico para funciones de membresía para nombres.

Tabla 3.3: Descripción de funciones de membresía

Nombre abierto	Nombre	Descripción	Forma	Parámetros Iniciales
mp_s	Muy probablemente similar	Se usa cuando ambos datos son prácticamente iguales	Triangular	$[0.8, 1, 1]$
p_s	Probablemente similar	Se usa cuando hay una posibilidad de que los datos sean iguales, pero con algún grado de incertidumbre	Triangular	$[0.5, 0.8, 1]$
n_s	No se sabe	Por diversas razones no se puede saber si son iguales o no	Triangular	$[0.4, 0.5, 0.7]$
p_{ns}	Probablemente no similar	Existe una probabilidad no menor de que los datos sean distintos	Triangular	$[0, 0.3, 0.5]$
mp_{ns}	Muy probablemente no similar	Los datos comparados son muy distintos	Triangular	$[0, 0, 0.3]$

Reglas para nombres: en estricto rigor se usa un árbol de decisión difuso para poder decidir no solo si el nombre es similar o no, sino que también si puede que ambas personas sean parientes o no. Para ese efecto se revisa la combinación de apellidos y nombres para determinar si ambos son semejantes partiendo desde el *apellido paterno*. La lógica es que si los apellidos paternos no se parecen es mucho menos probable que los individuos se parezcan, siendo la única posibilidad que sean parientes por parte de la madre. Luego se comparan los *apellidos maternos*, donde si son iguales confirma que el par puede ser de la misma familia. Pero es usual que los apellidos maternos no aparezcan en los registros, por lo tanto de faltar este dato igual se considera que es posible que dos personas sean iguales, pero si no se parecen se descarta *Igualdad*, pero no *Parentesco*. Finalmente depende del parecido de los nombres si se confirma la *Igualdad* o no.

Tabla 3.4: Tabla de reglas de nombres, con resultados de igualdad.

Apellidos	Nombres	Igualdad
ApellidoPaterno[mp_s] Y ApellidoMaterno[mp_s]	Nombres[mp_s]	Igualdad[mp_s]
	Nombres[p_s]	Igualdad[p_s]
	Nombres[ns]	Igualdad[ns]
	Nombres[p_ns]	Igualdad[p_ns]
	Nombres[mp_ns]	Igualdad[mp_ns]
ApellidoPaterno[p_s] Y ApellidoMaterno[p_s]	Nombres[mp_s ó p_s]	Igualdad[p_s]
ApellidoMaterno[ns] Y ApellidoPaterno[p_s ó mp_s]	Nombres[ns]	Igualdadns
ApellidoPaterno[p_ns]	Cualquier nombre	Igualdad[p_ns]
ApellidoPaterno[mp_ns]	Cualquier nombre	Igualdad[mp_ns]

3.3.2. Semejanza por Tiempos

El siguiente paso del algoritmo es comparar los tiempos en los cuales vivieron las personas. Para poder determinar si dos personas vivieron en tiempos cercanos se usan

diferentes variables difusas que cuantifican distintas distancias entre fechas consideradas importantes. Si todas ellas están en un rango normal se considera que ambas personas se encontraban en un intervalo de tiempo compatible. Si solo una de ellas llegase a fallar se considera que ambos registros son incompatibles temporalmente. Para llegar a esa conclusión, se usa un controlador difuso donde cada regla indica que son compatibles si y solo si todo el resto está conforme. Si falla uno entonces no hay coherencia.

A continuación se describirán los elementos del controlador difuso encargado de los tiempos:

Variables Difusas: en general para el diseño de las variables difusas de tiempo no se consideró un estado intermedio para “no sé” con el objetivo de simplificar la cantidad de reglas. Dependiendo de la naturaleza de la variable se tiene una función de membresía para el estado normal en el cual debería estar la variable, más otras indicando estados anómalos.

Se presentan a continuación las distintas variables difusas usadas por el controlador de tiempos:

- *Años verídicos reales (AVR):* corresponde a cuántos años se cree que vivió una persona en particular. Se obtiene a partir de las fechas de nacimiento y defunción si es que existen. Sus funciones de membresía están hechas pensando en que una persona de la época vive cerca de 80 años en promedio y a medida que se aleja de dicho valor es menor probable que esa persona haya estado viva (ver Figura 3.5a).

Funciones de membresía:

- *Normal:* indica que la persona vivió una cantidad normal de años. Tiene forma trapezoidal. Parámetros iniciales: $[-1, 0, 80, 120]$
- *Muy Viejo:* indica que la persona tiene demasiados años como para ser cierto. Tiene forma trapezoidal. Parámetros iniciales: $[80, 120, 200, 200]$
- *Paradoja:* indica que la persona es imposible que sea real porque implica que nació después de morir. Tiene forma trapezoidal. Parámetros iniciales:

$$[-200, -200, -1, 0]$$

- *Fecha de nacimiento inicial (FNI)*: corresponde a la distancia entre dos fechas de nacimiento de dos registros. Esta variable es necesaria porque nunca se registra un cumpleaños, sino que se tiene la fecha de bautizo y la edad, lo cual puede introducir variaciones en esta variable. Sus funciones de membresía se hicieron bajo la lógica de que *FNI* debe ser siempre cero para un caso ideal (ver Figura 3.5b).

Funciones de membresía:

- *Cerca*: las fechas de nacimiento están razonablemente cerca. Tiene forma triangular. Parámetros iniciales: $[0, 0, 10]$
- *Lejos*: las fechas de nacimiento están muy lejos como para ser consideradas coherentes. Tiene forma trapezoidal. Parámetros iniciales: $[0, 10, 200, 200]$.
- *Murió después de evento (MDE)*: comprueba de que una persona no haya hecho cosas después de morir, tales como asistir un matrimonio. Tiene algo de *slack* porque las fechas no siempre son exactas (ver Figura 3.5c).

Funciones de membresía:

- *Probablemente sí (p_s)*: el sujeto efectivamente murió después de ocurrido el evento. Tiene forma trapezoidal. Parámetros iniciales: $[0, 3, 200, 200]$
- *No se (n_s)*: no se tiene información para decidir. Tiene forma triangular. Parámetros iniciales: $[-3, 0, 3]$
- *Probablemente no (n_s)*: el sujeto murió antes de haber ocurrido el evento. Esto implica que hay una incoherencia. $[-200, -200, -3, 0]$
- *Edad antes de matrimonio (EAM)*: es el intervalo entre que una persona nace y se casa. Se ocupa pensando en que un individuo normalmente se casa siendo adulto, dependiendo de las leyes locales. Por lo tanto, hay un rango de edades donde es más probable que una persona de la época se haya casado, y otros dos donde es particularmente difícil que ocurra, en particular la niñez (ver Figura 3.6a).

Funciones de membresía:

- *Muy viejo*: la persona se considera muy vieja como para que se haya casado en esas fechas. Tiene forma trapezoidal. Parámetros iniciales: $[50, 60, 200, 200]$.
 - *Normal*: la persona se casó a una edad considerada común. Tiene forma trapezoidal. Parámetros iniciales: $[20, 25, 50, 60]$.
 - *Muy joven*: la persona es muy joven para haberse casado de acuerdo a los registros. Tiene forma trapezoidal. Parámetros iniciales: $[-200, -200, 0, 25]$.
- *Edad antes de nacimiento (EAN)*: es el intervalo entre el nacimiento de un hijo y el nacimiento del padre. Se asume que la edad para tener hijos se mueve dentro de un rango en forma similar al matrimonio. Usa las **funciones de membresía** de *EAM* (ver Figura 3.6b).
 - *Distancia entre eventos (DEV)*: es el intervalo entre dos eventos cualquiera que no están cubiertos bajo los casos anteriores. Funciona bajo la lógica de que los eventos que vive una persona tienen que haber ocurrido en un intervalo temporal acotado (ver Figura 3.6c).

Funciones de membresía:

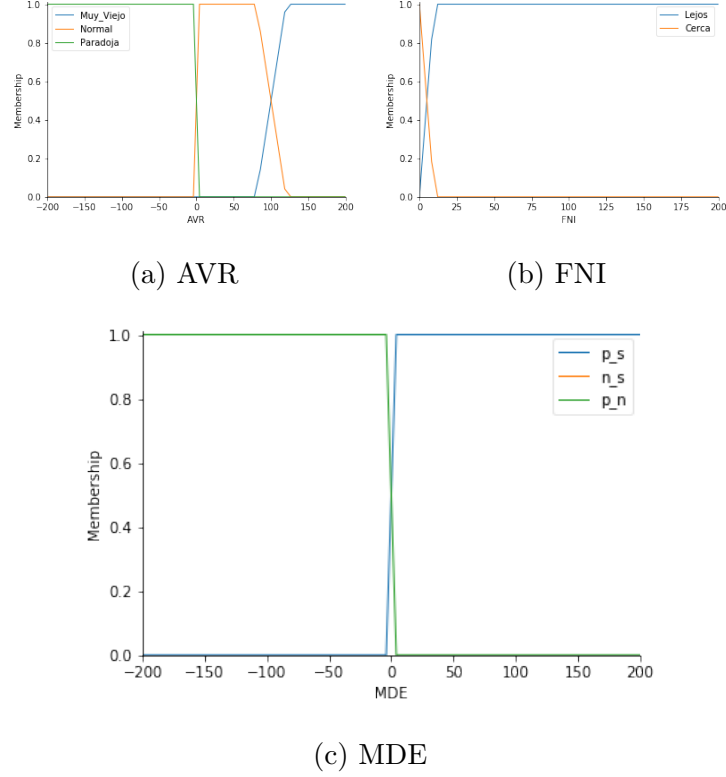
- *Normal*: la distancia entre eventos es razonable para que hayan ocurrido en la misma persona. Tiene forma de z , con parámetros iniciales $a = 25, b = 50$, los cuales indican que la función empieza a decaer en a y se hace cero en b .
 - *Lejos*: la distancia entre eventos es muy lejana para que haya sido atendida por la misma persona. Tiene forma de s , con parámetros iniciales $a = 25, b = 50$, los cuales indican que la función empieza a ascender en a y se hace uno en b .
- *Coherencia Temporal*: consecuencia de aplicar las reglas. Indica si los tiempos son normales de acuerdo a los parámetros o no. Se obtiene inicialmente usando un método de defuzzificación "mínimo de máximos (som)" (ver Figura 3.6d).

Funciones de membresía:

- *Alta*: ninguna de las variables anteriores está fuera de sus rangos normales. Tiene forma triangular. Parámetros iniciales: $[0, 0, 1]$

- *Baja*: hay alguna variable que no es coherente, por lo tanto no se deben unir los registros. Tiene forma triangular. Parámetros iniciales: $[0, 1, 1]$

Figura 3.5: Gráficos de las variables difusas AVR , FNI , MDE

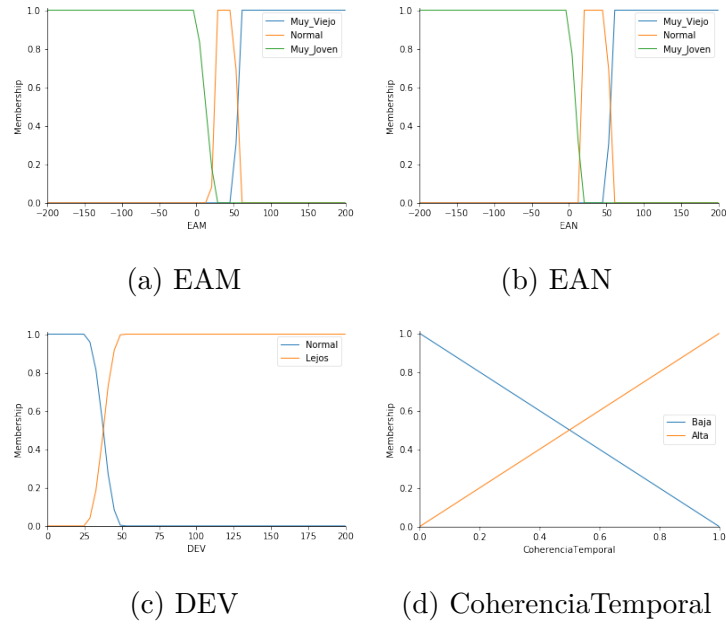


3.3.3. Semejanza por Relaciones de Grafo

Una de las aristas importantes para saber si es posible establecer si dos individuos son similares, y por lo tanto corresponde que se unan en un árbol genealógico a través de los parentescos comunes que ambos comparten. Para este fin se usa la estructura del grafo de similitudes entre dos candidatos a pariente en común, y si se puede establecer un nexo entre ambas opciones se otorga un puntaje.

El puntaje de una relación específica, el cual cuantifica qué tan fácil es llegar de una persona a otra mediante dicha conexión, corresponde al inverso de la cantidad de saltos necesarios para llegar de un pariente a otro. De esta forma, si solo se necesita un salto

Figura 3.6: Gráficos de las variables difusas EAM , EAN , DEV y $CoherenciaTemporal$



se obtiene un puntaje máximo, indicando que ambos parientes son muy similares, y por lo tanto los individuos también puede que lo sean. En caso contrario, el puntaje disminuye a medida que aumenta la cantidad de saltos necesarios.

Existen cuatro tipos de relaciones que se consideran lo suficientemente importantes para dar puntaje en primera instancia:

- Padre
- Madre
- Cónyuge
- Hijo

Se hace la distinción entre un padre y una madre para tomar en cuenta la posibilidad de que una persona sea *natural*. Así mismo se usan las relaciones *padre* $\rightarrow$ *hijo* e *hijo* $\rightarrow$ *padre* ya que como la relación padre/hijo es unidireccional puede ser prudente

investigar ambas direcciones para asegurar que el árbol resultante sea lo suficientemente coherente.

Después de calcular todos los puntajes entre dos personas, se usa un controlador difuso para decidir si, en base a los puntajes de grafo ambas personas merecen ser llamadas similares o no. Se usa una variable *fuzzy* por puntaje, siendo sus funciones de membresía los calificadores de “cerca” y “lejos” dependiendo de cuántos saltos hayan sido necesarios para pasar de un pariente a otro.

Las reglas que gobiernan al controlador corresponden a casos que apelan al “sentido común” de cómo deben ser las relaciones en una familia. Por ejemplo: si el puntaje de ambos padres está calificado como “cerca” entonces se considera más probable que los individuos sean similares. A partir del resultado de combinar las reglas se decide si la relación entre ambos tiene sentido o no de acuerdo al grafo, terminando en la elección de mantener o eliminar el nexo entre ambos.

Algunos de los resultados de las reglas pueden ser calificados como “ambiguos” en el sentido que no necesariamente se sabe *a priori* si son considerables como evidencia a favor o en contra, pero que si se intuye que pueden tener efecto. El genealogista podrá, por lo tanto, seleccionar si considera aquellas reglas como indicativas para mantener la unión o de separarla.

3.4. Visualizaciones

Para que la herramienta creada sea de uso para un genealogista es necesario mostrar la información generada en una forma relativamente fácil de analizar. Para este fin se escogieron tres estilos de visualización que sirven para determinar si los resultados son útiles para los fines del usuario.

Tabla 3.5: Reglas de grafo

Regla	Resultado
<i>Padre[Cerca] Y Madre[Cerca]</i>	<i>Decision[Conservar]</i>
<i>Padre[Lejos] O Madre[Lejos]</i>	<i>Decision[Rechazar]</i>
<i>Hijo[Cerca] Y Cónyuge[Cerca]</i>	<i>Decision[Conservar]</i>
<i>Hijo[Lejos] Y Padre[Cerca] Y Madre[Cerca]</i>	<i>Decision[Conservar]</i>
<i>(Padre[Cerca] Y Madre[Lejos]) O (Ma-</i>	<i>Decision[Default]</i>
<i>dre[Cerca] Y Padre[Lejos])</i>	
<i>Hijo[Lejos] Y Cónyuge[Lejos] Y NO Pa-</i>	<i>Decision[Rechazar]</i>
<i>dre[Cerca] Y NO Madre[Cerca]</i>	
<i>(NO Hijo[Cerca] O NO Cónyuge[Cerca]) Y</i>	<i>Decision[Conservar]</i>
<i>(Padre[Cerca] Y Madre[Cerca])</i>	
<i>(Hijo[Lejos] Y Cónyuge[Cerca]) O (Hi-</i>	<i>Decision[Default]</i>
<i>jo[Cerca] Y Cónyuge[Lejos])</i>	
<i>Cónyuge[Cerca]</i>	<i>Decision[Conservar]</i>
<i>Hijo[Cerca]</i>	<i>Decision[Conservar]</i>
<i>Cónyuge[Lejos]</i>	<i>Decision[Conservar]</i>

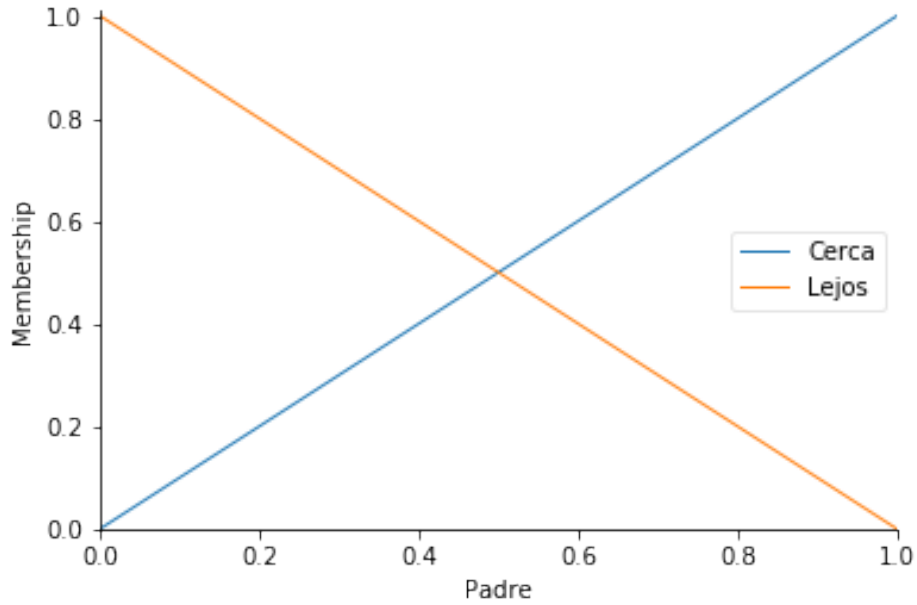


Figura 3.7: Variable difusa de padre. Las otras variables tienen la misma forma.

3.4.1. Árbol Genealógico

Para fines de comprender el legado genealógico de una o varias personas, una de las herramientas más usadas es el árbol genealógico [5]. Permite ver las relaciones de ascendencia y descendencia de una persona (ver Figura 3.8a).

Si bien existen diversos formatos para mostrar la información de la genealogía de una persona [7][8], se decidió por usar el método que está por defecto en `vis.js` para visualización de grafos jerárquicos.

Para decidir qué árbol mostrar se escogen expresiones regulares para nombre y apellidos y se ven cuáles de los nodos resultados calzan. Luego se obtiene el subgrafo al que pertenecen, respetando la cantidad de saltos de relaciones de padre que se determinaron con antelación. De esta forma se tiene un árbol genealógico centrado en el o los resultados buscados, con una cantidad de antecesores y sucesores predefinidos.

3.4.2. Búsqueda de Parientes

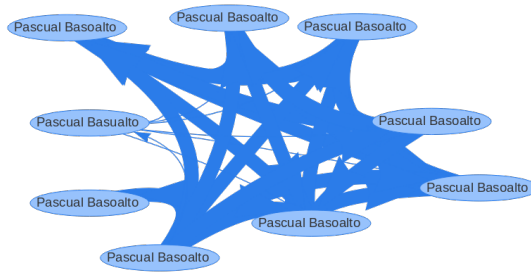
Una de las preguntas que son de interés para un genealogista es "¿Estas dos personas son parientes?". Para resolverla se usó el algoritmo para caminos más cortos que viene por defecto en *Neo4j*. La idea es que si, mediante relaciones de *padre/hijo*, se logra encontrar un camino entre dos personas es porque tienen algún pariente en común. De esta forma se pueden confirmar si dos individuos son primos, por ejemplo, o si hay algún otro tipo de consanguinidad que pueda generar un impedimento en un matrimonio (ver Figura 3.8b).

3.4.3. Particiones

Cada persona dentro de un árbol es en realidad una colección de nodos *Persona* que están unidas por relaciones de similitud y que, mediante detección de comunidades se determinan que se tratan de la misma persona. Para poder diagnosticar problemas en

dicha union, se hace necesario ver cómo son los *links* entre los registros dentro de la partición.

La visualización consiste en mostrar todos los nodos que conforman a la partición escogida dentro de un árbol de interés junto con sus relaciones de similitud. De esta forma se puede confirmar que se unieron dos personas que no deberían haber sido unidas bajo el criterio del usuario, y por lo tanto, hay que cambiar parámetros (ver Figura 3.9a).



(a) Particiones

Figura 3.9: Imágenes de las tres formas de visualizar información de los resultados.

Capítulo 4

Implantación y Validación de la Solución

En el capítulo presente se prueba el sistema usando datos genealógicos obtenidos desde registros eclesiásticos de los siglos XVIII y XIX. Las pruebas consisten en la ejecución de los algoritmos usando distintos parámetros, obteniendo diferentes árboles genealógicos de personas consideradas interesantes. Se analizan comparando los resultados con árboles encontrados manualmente.

4.1. Hardware

Las siguientes pruebas se realizaron sobre un computador de escritorio cuyas características son:

- **CPU:** *AMD Ryzen 5 1400 Quad-Core @ 8x 2.744GHz*
- **RAM:** *4 GB*
- **OS:** *Arch Linux, kernel x86_64 Linux 4.19.4-arch1-1-ARCH*
- **Versión de Neo4j:** 3.4.9 ejecutado desde *docker*.

4.2. Conjunto de datos

Se trabaja sobre un conjunto de datos que consiste en 944 registros eclesiásticos de distintos tipos. Se aprecia una cantidad considerablemente mayor de matrimonios en comparación a otros tipos de eventos. Esto puede causar problemas al momento de calcular cotas inferiores en las fechas de nacimiento. Si se compara la cantidad de padres y madres, se puede deducir que hay madres solteras dentro de los datos, ya que la cantidad de madres es mayor a la de padres. Sin embargo, en total hay más hombres que mujeres. El desglose se presenta en las tablas 4.1, 4.2 y 4.3.

Tabla 4.1: Desglose de registros dentro del *dataset* de acuerdo al tipo de evento

Defunciones	Matrimonios	Bautizos	Total
121	738	57	944

Tabla 4.2: Desglose de registros dentro del *dataset* de acuerdo al tipo de persona

Padre	Madre	Casado	Padrino	Testigo	Bautizado	Fallecidos	Total
234	262	234	65	71	19	48	944

Tabla 4.3: Desglose de registros de acuerdo al sexo de la persona

Hombre	Mujer	No se sabe	Total
482	451	11	944

4.3. Metodología de las pruebas

Se ejecuta el algoritmo completo usando una configuración en específico, el cuál consiste en las opciones relevantes de nombre (ej: elección del algoritmo de comparación de *strings*), tiempo y grafo, más los parámetros de las variables difusas cuando hayan cambios importantes con respecto a su *default*. Luego se unen usando *Louvain* para la

detección de personas. Finalmente se buscan y comentan los árboles resultantes de las siguientes personas:

- Juan Antonio Duval: su árbol genealógico es de tamaño pequeño. Presenta problemas de ambigüedad, ya que hay tres generaciones de personas que tienen el mismo nombre (Juan Antonio) y no se tiene la suficiente información para saber quién es padre de quién.
- Pascual Basoalto: su familia es de tamaño grande en comparación al anterior. Permite mostrar la capacidad del sistema de armar árboles con múltiples generaciones. Por su tamaño, la posibilidad de encontrar anomalías al momento de cambiar parámetros es más grande que con la persona anterior.

4.4. Primera prueba: Parámetros por defecto

Primero se realiza la ejecución del programa usando los parámetros predefinidos, los cuales consisten en:

- **Nombres:**
 - Parámetros *fuzzy*: por defecto (explicados en sección 3.3.1)
 - Coeficiente: *OverlapCoefficient*
 - Tokenizador: *por q-gramas*
 - *Threshold*: 0.65
- **Tiempos:**
 - Parámetros *fuzzy*: por defecto (explicados en sección 3.3.2)
- **Grafos:**
 - Parámetros *fuzzy*: por defecto (explicados en sección 3.3.3)
 - Eliminar sin puntaje: verdadero
 - Aceptar inciertos: verdadero

4.4.1. Pascual Basoalto

Al buscar por “Pascual Basoalto” se encuentran dos árboles (Figura 4.1); el más pequeño de ellos (Figura 4.2) consiste en solo dos integrantes: Pascual, y su madre Bárbara. Se puede asumir que se trata de una madre soltera, considerando que el hijo tiene su apellido.

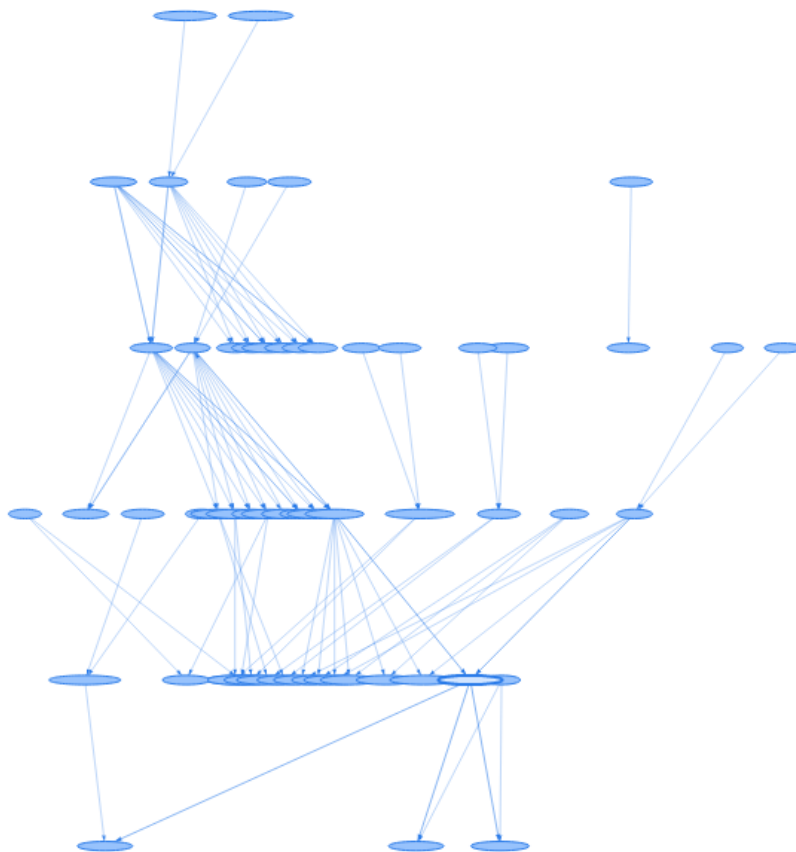


Figura 4.1: Vista general de los árboles genealógicos que contienen a “Pascual Basoalto”

La rama más grande consiste en cinco generaciones de Basoalto (Figura 4.3). Pascual Basoalto es hijo de Francisca (Díaz de) León y Francisco Basoalto. El caso de la madre de Pascual es interesante porque es un ejemplo donde hay que usar “apodos”. Normalmente la similitud entre “Díaz de León” y “León” es prácticamente nula, pero diciéndole al sistema que ambos son equivalentes permite asegurar que exista una



Figura 4.2: Familia de Pascual y Bárbara Basoalto

comparación entre ellos.

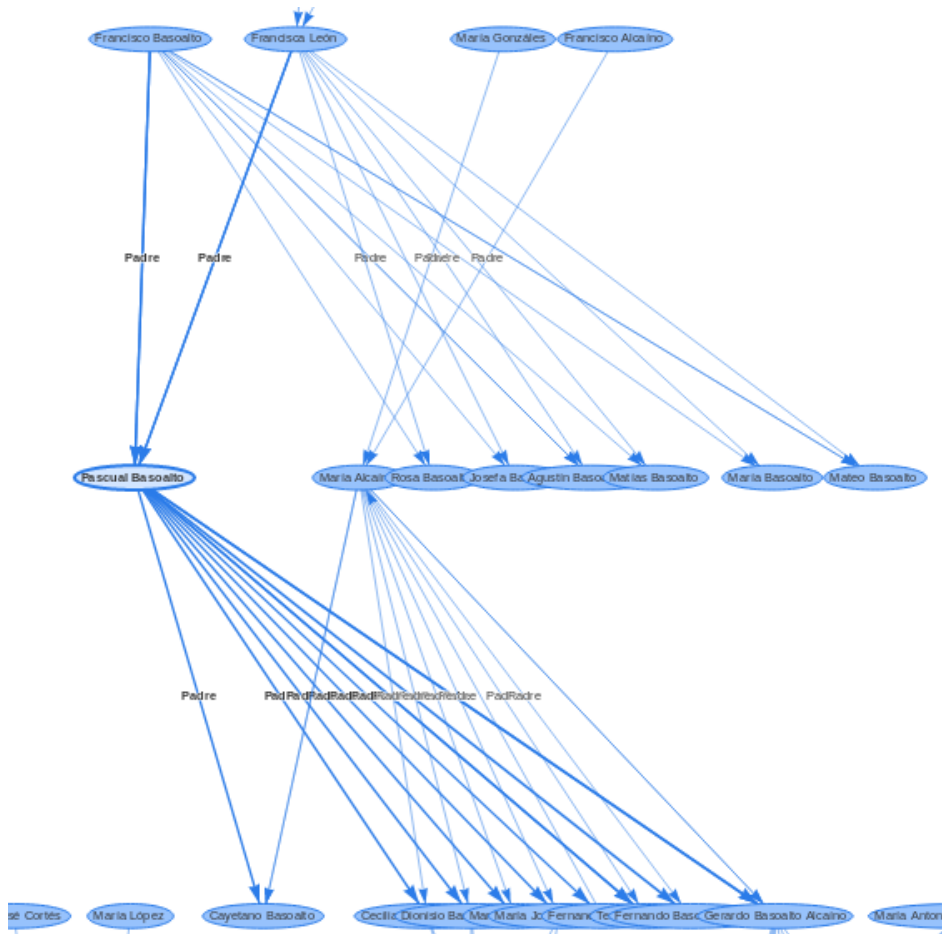


Figura 4.3: Árbol genealógico centrado en Pascual Basoalto León

La partición de Pascual Basoalto León es un ejemplo de registros íntimamente interconectados (Figura 4.4). Es prácticamente de la manera que se debería ver una partición después de la poda, donde todos se parecen entre sí, dando evidencia de que los registros deben ser unidos. La excepción está en un registro que solo se parece a uno de sus pares; en este caso es porque se parecen los padres. Con el resto se presume que no está conectado porque el resto se parece entre sí por sus hijos y matrimonios. Este caso puede ser evidencia de que la restricción, donde solo se conservan aquellos que tengan puntaje de grafo, es muy restrictiva para este caso.

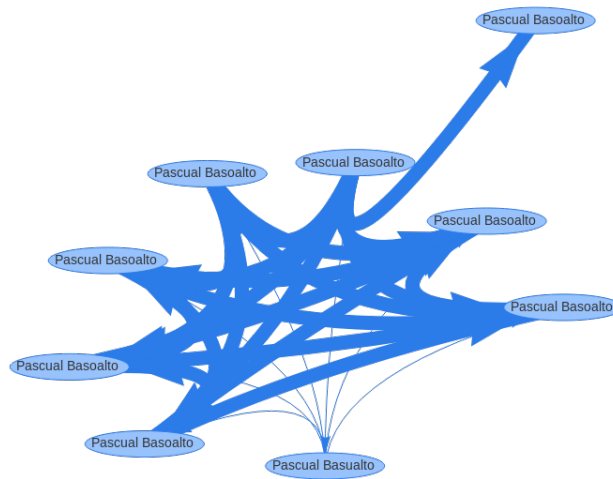


Figura 4.4: Partición de Pascual Basoalto León

4.4.2. Juan Antonio Duval

Cuando se busca por “Juan Antonio Duval” se encuentran tres familias (Figuras 4.5 y 4.6). Todas ellas tienen un patriarca con el nombre buscado y un hijo con el mismo nombre. El hecho de que ambos compartan el nombre puede generar ambigüedades, cosa que no ocurrió en este *test*, ya que se fue estricto en que se compararan solo aquellos en los cuales se está seguro de que hay información de grafo.

Es probable que haya algún nexo entre estas tres ramas, ya que vivieron en fechas relativamente comparables. *A priori* se sabe que uno de ellos es padre del otro, pero para un humano la tarea de saber cómo es realmente la relación entre ellos es muy complicada.

4.5. Segunda prueba: No eliminar sin puntaje de grafo

En esta prueba se verifica el efecto que tiene el no eliminar aquellas similitudes donde no se pudieron comparar por grafo. *A priori* se puede esperar que se mantengan más relaciones de similitud y, por lo tanto, se unan más personas.

4.5.1. Pascual Basoalto

Al buscar por “Pascual Basoalto” se genera un árbol extremadamente complicado como para corresponder a la realidad (Figura 4.7). Si se mira a los ascendientes y descendientes directos de “Pascual Basoalto León” (Figura 4.8) se puede apreciar la presencia de una hija de “Pascual Basoalto” y “Francisca León”, lo cual es incorrecto en este caso.

Otro caso anómalo es el de José María Basoalto (Figura 4.9), el cual tiene múltiples padres y madres. Probablemente al relajar la condición de que solo se mantienen aquellos comparados por grafo se conservan relaciones superfluas que no aportan en nada.

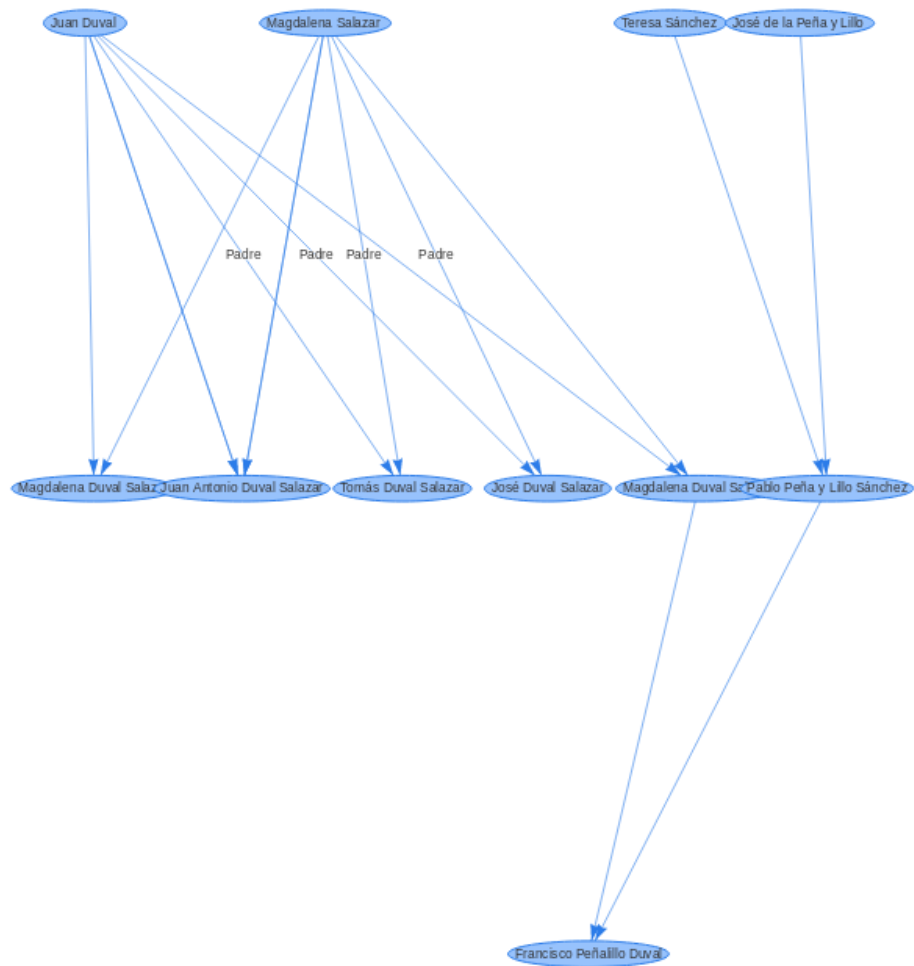


Figura 4.5: Árbol que contiene a Juan Duval Salazar

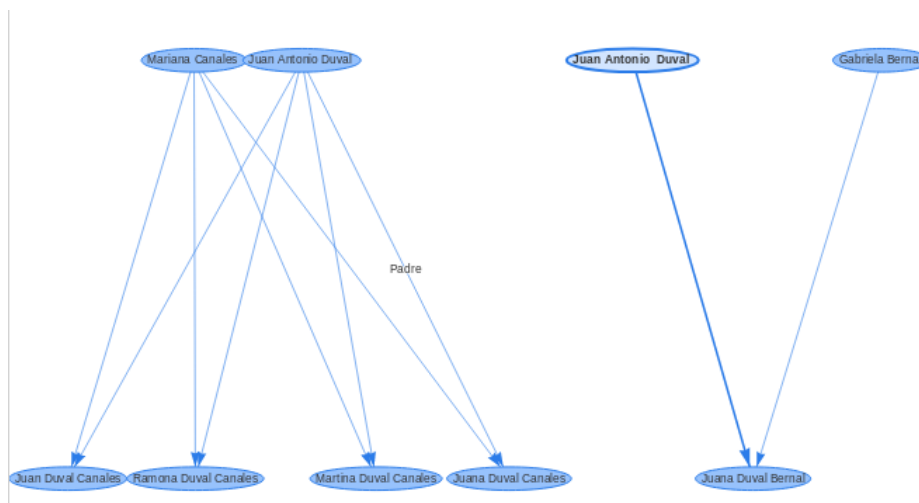


Figura 4.6: Árboles que contienen a Juan Duval Canales y a Juan Duval Bernal

4.5.2. Juan Antonio Duval

Cuando se busca por “Juan Antonio Duval” se encuentra un grafo un poco más compacto que en la primera prueba (Figura 4.10). En este caso sí se considera que hay ambigüedades al momento de decidir que cuál Juan Antonio Duval es cuál, ya que se sabe que en ese grupo están representados tanto los padres como los hijos (ya que hay una flecha que apunta a si mismo), pero no hay suficiente información como para saber quién es quién. A lo menos confirma que ocurrieron en tiempos relativamente comparables. También se puede confirmar que Juan Duval, padre de Juan Antonio Duval Salazar, es el más antiguo de ellos (Magdalena Salazar es la madre más antigua del grupo). Por lo tanto, Juan Antonio Duval Salazar puede que se haya casado con alguien de su misma generación.

Dado los casos anteriores, se asume que es necesario eliminar aquellos registros que no se pudieron comparar mediante relaciones de grafo si se desea tener resultados más certeros. Mantener relaciones sin comparación de grafo puede ser útil en casos donde se sabe que hay ambigüedades que se puedan hacer más evidentes conservando algunas de las relaciones sin evidencia por árbol (resultado de comparar actas de nacimientos con el matrimonio de un hijo, por ejemplo).

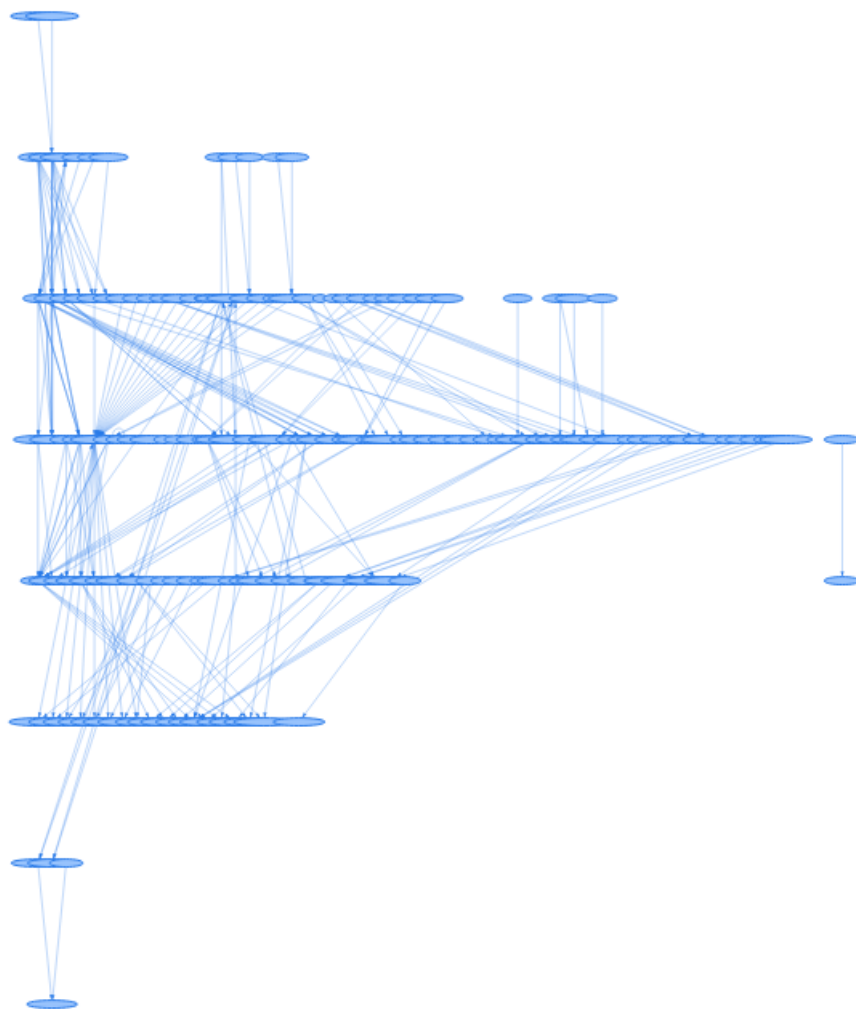


Figura 4.7: Vista general del árbol genealógico de Pascual Basoalto

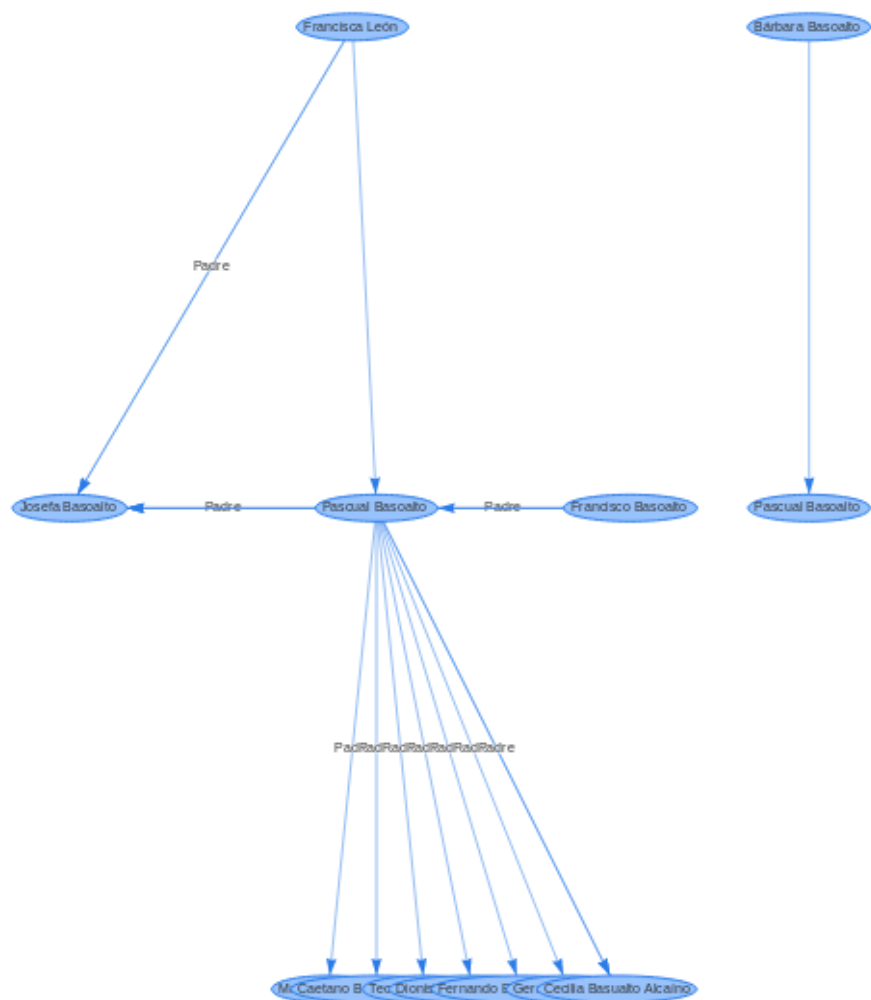


Figura 4.8: Descendientes y ascendientes directos de Pascual Basoalto León

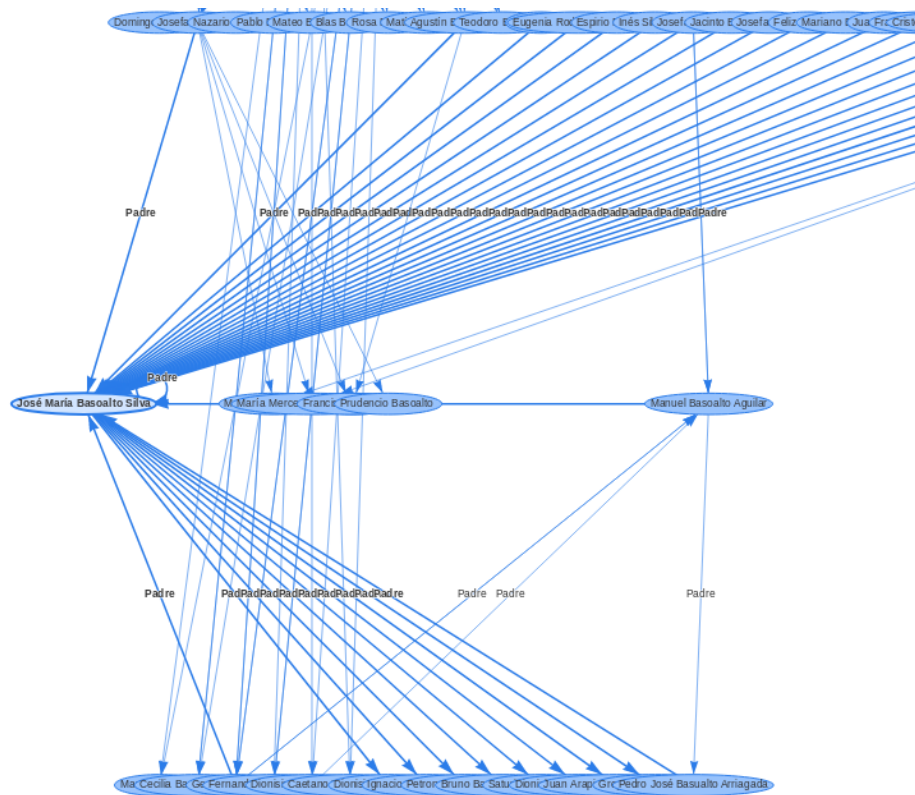


Figura 4.9: Árbol genealógico de José María Basoalto

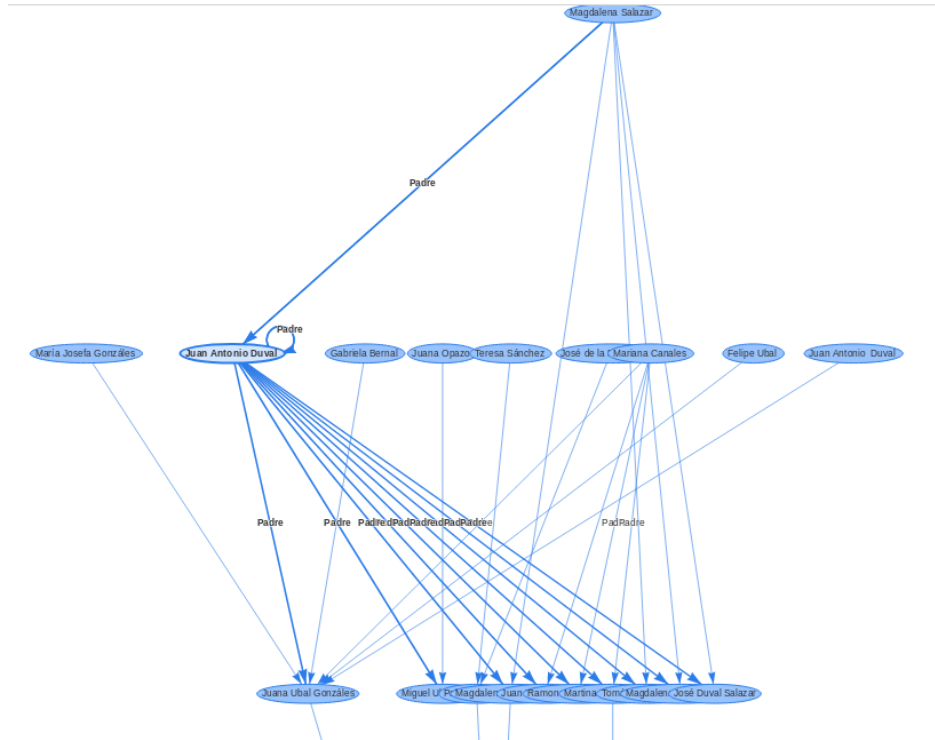


Figura 4.10: Árbol genealógico de Juan Antonio Duval

4.6. Tercera Prueba: Usando *Jaro-Winkler* como comparación de *Strings*

En esta prueba se quiere saber el efecto que tiene cambiar el cálculo de puntajes de nombres desde uno basado en *tokens* a uno basado en *distancia de edición*. Se usó *Jaro-Winkler* donde el puntaje necesario para ser guardado es de 0.7, indicando que el parecido entre dos nombres debe ser de al menos 0.7, sino no se guarda. Después se comparará con el mismo algoritmo pero con un *threshold* de 0.8.

4.6.1. Pascual Basoalto con un *threshold* de 0.7

Se puede observar que al buscar por “Pascual Basoalto” se obtiene una familia con más conexiones (Figura 4.11). Al centrarse en Pascual Basoalto León se puede corroborar que sus familiares se mantienen intactos en comparación con la primera prueba

(Figura 4.1).

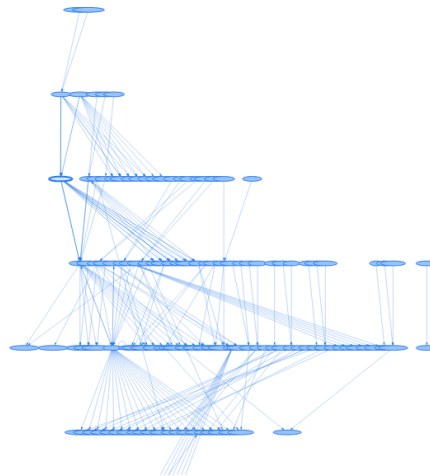


Figura 4.11: Vista general de la familia Basoalto

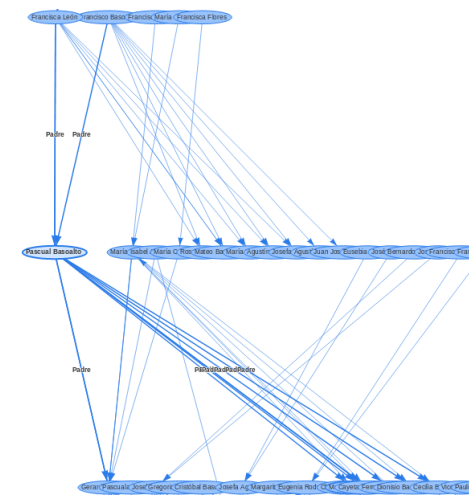


Figura 4.12: Vista a la familia de Pascual Basoalto León

Con Pablo Basoalto se genera una anomalía, ya que tiene muchos padres (Figura 4.13). Como el sistema tolera múltiples cambios a los nombres entonces hace que se generen similitudes con personas con las cuales son bastante diferentes (Figura 4.14).

4.6.2. Juan Antonio Duval con un *threshold* de 0.7

En este caso también se generan conexiones extra, generadas por relajar los requisitos para aceptar un puntaje de nombre (Figura 4.15). Se puede apreciar que el problema de la ambigüedad en la ascendencia de Juan Antonio Duval es solucionado haciendo que uno de ellos tenga hijos con Magdalena Salazar y Mariana Canales a la vez, insinuando la presencia de un segundo matrimonio.

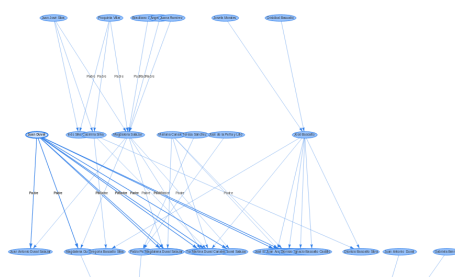


Figura 4.15: Vista a Juan Antonio Duval

4.6.3. *Threshold* en 0.8

Con un puntaje mínimo de 0.8 se obtienen resultados similares a la primera prueba tanto para “Pascual Basoalto” como para “Juan Antonio Duval” (Figura 4.16). Esto implica que el parámetro *threshold* debería tener un valor entre 0.7 y 0.8 para lograr resultados pasables. Probablemente exista un valor óptimo que permita adquirir más información sin introducir ruido en dicho rango. Desafortunadamente, con el hardware usado cada prueba toma cerca de 40 minutos, lo cual hace que buscar exhaustivamente un resultado positivo tome mucho tiempo.

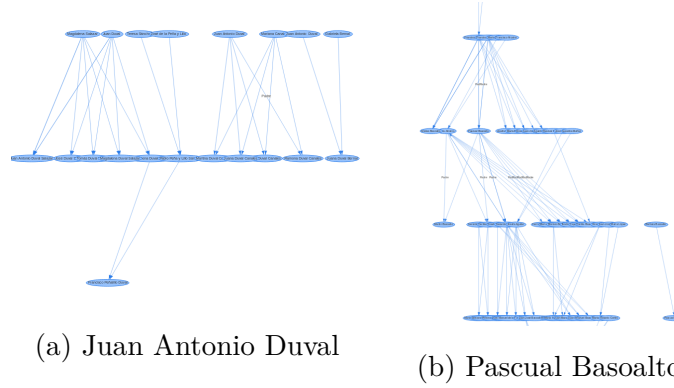


Figura 4.16: Resultados de usar un *threshold* de 0.8

4.7. Prueba cuatro: Cambios a rangos de tiempo (*DEV*)

En esta prueba se cambia la variable *DEV* usando como parámetros $[a : 50, b.120]$, lo que implica que considera hasta unos 50 años para que la *distancia entre eventos* tenga como resultado *normal*, bajando gradualmente a *lejos* desde 50 años a 120 años. Esto amplía el rango donde la distancia entre eventos es normal.

Se escoge *DEV*, ya que existe una gran cantidad de registros que no son “primarios” (personajes centrales de un acta de nacimiento, matrimonio o defunción) y por lo tanto, no son capturados por el resto de las variables.

Haciendo este cambio solo se encontró una diferencia significativa al buscar por “Juan Antonio Duval”, por lo cual solo se muestran sus resultados (Figura 4.17). En dicho caso aparece un hijo nuevo, *Pedro Ubal*, el cual es hijo de un Juan Ubal pero de madre desconocida. Lo anterior puede implicar la presencia de otro matrimonio, o que simplemente hacen falta más datos para poder probar o desmentir que Pedro Ubal fue hijo de Juan Duval y que simplemente hubo un cambio de apellido por parte del sacerdote a cargo.

Se puede concluir que aumentando el rango donde *DEV* es considerado *normal*, efectivamente se conservan más registros como *similares*.

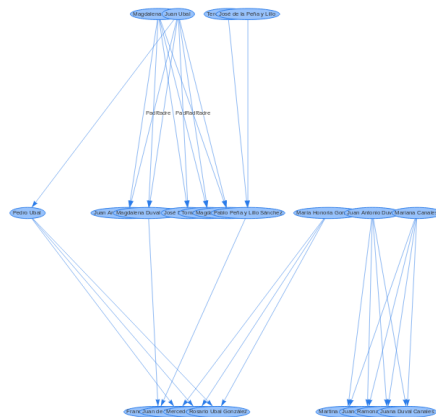


Figura 4.17: Árbol genealógico de Juan Antonio Duval

4.8. Rendimiento de la base de datos

Hay que mencionar que la ejecución completa de los pasos del algoritmo toma un tiempo de aproximadamente 40 minutos, siendo el paso de comparar nombres el que toma más tiempo. La culpa en este caso consiste en que es necesario calcular la similitud de nombres entre todos los pares de personas existentes, lo cual es costoso.

El tiempo hipotéticamente se podría mejorar aplicando alguna estrategia de paralelismo, pero en la práctica se hace complicado porque *Py2neo* no soporta el uso de *multiprocess* [63] [64]. Además *multithreading* no sirve en este caso, ya que el problema de rendimiento está dado por la *CPU* y *Cpython* solo ejecuta un *thread* a la vez [65].

Al probar el programa en el *notebook* de desarrollo, el cual tiene como especificaciones una *CPU AMD A8-3510MX APU* y *8 GB* de *RAM*, se puede notar una grave ralentización del programa, subiendo el tiempo de ejecución a más de 12 horas.

Otra opción para reducir los tiempos es limitar la comparación a nombres o apellidos iguales. De esa forma solo se comparan aquellos que tengan el mismo nombre pero sin necesariamente saber si los apellidos son similares, o aquellos que tienen los mismos apellidos, asegurando que aquellos que sean comparados sean de la misma familia pero que tengan distintas representaciones en su nombre. De esta forma se consiguen resultados en 5 minutos en el computador de prueba. Sin embargo no está exento de

consecuencias la realización de dicha simplificación, la cual incluye la reducción de posibles similitudes de nombres.

Un efecto importante de hacer aún más estricta la selección por nombres es que la estrategia de usar *apodos* no funciona tan bien, ya que si no son el mismo apellido y nombre entonces no hay comparación, por lo tanto no se puede aplicar el apodo. Con los sujetos de prueba usados no ocurrió este suceso, pero puede ser una consideración importante con *datasets* más grandes y/o con más errores tipográficos.

4.9. Resumen de los resultados

Los resultados obtenidos en esta sección en general son alentadores. Para casos simples logra conectar los distintos individuos encontrando árboles genealógicos que tienen sentido. En casos más complicados un genealogista tiene la posibilidad de investigar diversas formas de atacar el problema de la ambigüedad presente mediante el ajuste de parámetros del algoritmo. Esto necesariamente implica relajar algunas condiciones, tales como hacer que la comparación de nombres exija una menor similitud para considerarlos como posibles candidatos a ser las mismas personas, o hacer que el sistema use un rango de tiempo mayor para buscar candidatos; resultando en una mayor posibilidad de encontrar inconsistencias en los árboles.

Desafortunadamente el rendimiento del sistema podría ser mejor, ya que si bien la simplificación usada acelera notablemente la comparación entre los diferentes registros, es posible que el tiempo de ejecución siga siendo alto cuando la cantidad de datos llegue a los millones. Las formas de mejorar esto no solo pasan por buscar más esquemas de *blocking* o de usar *hardware* más poderoso, sino que también implica portar el sistema a algún entorno que permita usar los múltiples núcleos de un procesador en una forma eficiente.

Capítulo 5

Conclusiones

Se logró el objetivo de armar árboles genealógicos a partir de datos sucios y duplicados mediante el uso de una *BDG* más la aplicación de lógica difusa y algunos algoritmos de grafo, particularmente *algoritmos de comunidad*. Junto a lo anterior se proveyeron herramientas para facilitar el análisis, tales como visualización de árboles y de particiones. Todo esto permite obtener resultados coherentes y con menos trabajo que el requerido al hacerlo en forma manual.

La representación como grafo ayuda a encontrar resultados novedosos, aunque siempre hay que tomar en cuenta que dichos resultados vienen con un aumento de incertidumbre, la cual resulta de reducir o eliminar algunas restricciones. Esto implica que la información es menos confiable para juzgar en forma definitiva. A su vez, si se usan restricciones más estrictas los resultados son más confiables, pero también más obvios. Esta representación no solo ayuda al momento de buscar relaciones entre los distintos individuos presentes en los árboles, sino que también facilita el uso de algoritmos de grafo de mayor complejidad, permitiendo obtener información valiosa.

Se probaron e implementaron distintos algoritmos para ayudar a la deduplicación de datos, en particular para los nombres donde hay una gran variedad de métricas a elegir. Dependiendo de los requisitos de cada genealogista se pueden usar algoritmos basados en *tokens*, los cuales son mejores cuando se sabe que no hay errores ortográficos y solo

se quiere saber si un nombre está contenido dentro de otro más grande, por ejemplo. Por su parte, los basados en *distancia de edición* son sensibles a pequeños cambios en los nombres, los cuales los hace muy buenos para errores tipográficos, pero más difíciles de trabajar cuando se trata de comparar un segundo nombre con un nombre completo, por ejemplo.

El rendimiento de la base de datos es competente, siendo el mayor cuello de botella el trabajo en *CPU* por parte del programa. Por esto el programa es usable con *datasets* pequeños, como el descrito en la sección 4.2. Por lo mismo se recomienda separar *datasets* en instancias más pequeñas o usar un procesador lo más poderoso posible.

Una de las extensiones a hacer en el presente trabajo es mejorar el rendimiento del programa en general. Se recomienda refactorizar el programa para poder permitir la ejecución paralela de los algoritmos de analizado. De no ser posible se debería tratar de hacer que la mayor cantidad de cálculos dentro de la *BDG* mediante la creación de *plugins* para *Neo4j*.

Otra extensión a hacer es la creación de más reglas, con el objetivo de capturar casos que ahora no son considerados dentro del sistema. Por ejemplo, usar el estado de viudez al momento de un matrimonio para decidir si está casado por segunda vez o no. En conjunto con esto se pueden usar métodos más inteligentes para generar reglas [25] con la desventaja que necesita una cantidad considerable de datos de entrenamiento, más la experticia suficiente para poder usar dicho enfoque.

Los conocimientos adquiridos en la carrera fueron de particular utilidad para el correcto desarrollo de la memoria. Se destacan los conocimientos de bases de datos, los cuales fueron útiles al momento de contrastar el funcionamiento de una *BDG* con una relacional convencional, además de permitir entender mejor la nomenclatura y conceptos.

Por otro lado, los conocimientos de matemáticas discretas adquiridas durante la carrera fueron de particular uso al momento de trabajar con grafos. Se incluyen en este caso tanto los cursos más básicos de teoría de grafos como electivos donde se aprende sobre redes complejas.

Las habilidades enseñadas en los cursos de desarrollo de software fueron otro pilar importante del trabajo, permitiendo diseñar el programa de la mejor forma posible.

Durante el transcurso de la carrera se dictaron varios electivos relacionados con *big data*, los que permitieron el aprendizaje de distintos paradigmas para trabajar información, incluyendo todo lo que es bases de datos *NoSQL*, dentro de las cuales se incluyen las *BDG*.

Bibliografía

- [1] DB-Engines Ranking - popularity ranking of graph DBMS, . URL <https://db-engines.com/en/ranking/graph+dbms>.
- [2] Artyom Kalinin. : (A, C, E, D, F) A F .English: Shortest path (A, C, E, D, F) between vertices A and F in the weighted directed graph., December 2013. URL https://commons.wikimedia.org/wiki/File:Shortest_path_with_direct_weights.svg.
- [3] j_ham3. English: A sketch of a small network displaying community structure, with three groups of nodes with dense internal connections and sparser connections between groups., October 2011. URL https://commons.wikimedia.org/wiki/File:Network_Community_Structure.svg.
- [4] Donald Bell. The component diagram, December 2004. URL <http://www.ibm.com/developerworks/rational/library/dec04/bell/index.html>.
- [5] Julián Pérez Porto and María Merino. Definición de genealogía — Definicion.de, 2011. URL <https://definicion.de/genealogia/>.
- [6] Michael J McGuffin and Ravin Balakrishnan. Interactive Visualization of Genealogical Graphs. page 8.
- [7] Robert Ball. Visualizing genealogy through a family-centric perspective. *Information Visualization*, 16(1):74–89, 2017.
- [8] Kerstin Keller, Prahalika Reddy, and Shimul Sachdeva. Family Tree Visualization. page 11.

- [9] I. Robinson, J. Webber, J. Webber, and E. Eifrem. *Graph Databases*. O'Reilly, 2013. ISBN 978-1-4493-5626-2. URL <https://books.google.cl/books?id=RTvAmQEACAAJ>.
- [10] Fraud Detection | Neo4j Graph Database Solutions, . URL <https://neo4j.com/use-cases/fraud-detection/>.
- [11] Ben Rund. The Good, The Bad, and the Hype about Graph Databases for MDM, March 2017. URL <https://tdwi.org/articles/2017/03/14/good-bad-and-hype-about-graph-databases-for-mdm.aspx>.
- [12] The Neo4j Graph Platform – The #1 Platform for Connected Data, . URL <https://neo4j.com/>.
- [13] Using Neo4j from Java, . URL <https://neo4j.com/developer/java/>.
- [14] Nigel Small. The Py2neo v4 Handbook — The Py2neo v4 Handbook. URL <https://py2neo.org/v4/>.
- [15] Neo4j Subscriptions, . URL <https://neo4j.com/subscriptions/>.
- [16] Multi-Model Database | Graph Database | OrientDB, . URL <http://orientdb.com/>.
- [17] OrientDB Enterprise - Multi-Model NoSQL Enterprise Database, . URL <http://orientdb.com/orientdb-enterprise/>.
- [18] Titan: Distributed Graph Database, . URL <http://titan.thinkaurelius.com/>.
- [19] Part III. Storage Backends, . URL <http://s3.thinkaurelius.com/docs/titan/1.0.0/storage-backends.html>.
- [20] TinkerPop3 Documentation, . URL <http://tinkerpop.apache.org/docs/3.0.1-incubating/#graph>.
- [21] DB-Engines Ranking - Method, . URL https://db-engines.com/en/ranking_definition.

- [22] Tags - Stack Overflow, . URL <https://stackoverflow.com/tags?tab=popular>).
- [23] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. An Evaluation Methodology and Experimental Comparison of Graph Databases. 2017.
- [24] Mark J Wierman. *An Introduction to the Mathematics of Uncertainty*. August 2010. URL https://www.creighton.edu/fileadmin/user/CCAS/programs/fuzzy_math/docs/MOU.pdf.
- [25] Jyh-Shing Roger Jang. *Neuro-fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence*. Prentice Hall, 1997. ISBN 978-0-13-261066-7. Google-Books-ID: vN5QAAAAMAAJ.
- [26] L. A. Zadeh. Quantitative fuzzy semantics. *Information Sciences*, 3(2):159–176, April 1971. ISSN 0020-0255. doi: 10.1016/S0020-0255(71)80004-X. URL <http://www.sciencedirect.com/science/article/pii/S002002557180004X>.
- [27] Werner Van Leekwijck and Etienne E. Kerre. Defuzzification: criteria and classification. *Fuzzy Sets and Systems*, 108(2):159–178, December 1999. ISSN 0165-0114. doi: 10.1016/S0165-0114(97)00337-0. URL <http://www.sciencedirect.com/science/article/pii/S0165011497003370>.
- [28] Harpreet Singh, Madan M. Gupta, Thomas Meitzler, Zeng-Guang Hou, Kum Kum Garg, Ashu M. G. Solo, and Lotfi A. Zadeh. Real-Life Applications of Fuzzy Logic, 2013. URL <https://www.hindawi.com/journals/afs/2013/581879/>.
- [29] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, July 2009. ISBN 978-0-262-53305-8. Google-Books-ID: aefUBQAAQBAJ.
- [30] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest paths algorithms: theory and experimental evaluation. *Mathematical Programming*, 73 (2, Ser. A):129–174, 1996. ISSN 0025-5610. doi: 10.1016/0025-5610(95)00021-6. URL <https://mathscinet.ams.org/mathscinet-getitem?mr=1392160>.

- [31] Amit Patel. Amit's A* Pages. URL <http://theory.stanford.edu/~amitp/GameProgramming/>.
- [32] Santo Fortunato and Darko Hric. Community detection in networks: A user guide. *Physics Reports*, 659:1–44, November 2016. ISSN 03701573. doi: 10.1016/j.physrep.2016.09.002. URL <http://arxiv.org/abs/1608.00163>. arXiv: 1608.00163.
- [33] Symeon Papadopoulos, Yiannis Kompatsiaris, Athena Vakali, and Ploutarchos Spyridonos. Community detection in Social Media: Performance and application considerations. *Data Mining and Knowledge Discovery*, 24(3):515–554, May 2012. ISSN 1384-5810, 1573-756X. doi: 10.1007/s10618-011-0224-z. URL <http://link.springer.com/10.1007/s10618-011-0224-z>.
- [34] Chapter 5. Community detection algorithms - The Neo4j Graph Algorithms User Guide v3.4, . URL <https://neo4j.com/docs/graph-algorithms/current/algorithms/community/>.
- [35] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate Record Detection: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(1): 1–16, January 2007. ISSN 1041-4347. doi: 10.1109/TKDE.2007.250581.
- [36] Gonzalo Navarro. A Guided Tour to Approximate String Matching. URL http://repositorio.uchile.cl/bitstream/handle/2250/126168/Navarro_Gonzalo_Guided_tour.pdf.
- [37] Wael H.Gomaa and Aly A. Fahmy. A Survey of Text Similarity Approaches. *International Journal of Computer Applications*, 68(13):13–18, April 2013. ISSN 09758887. doi: 10.5120/11638-7118. URL <http://research.ijcaonline.org/volume68/number13/pxc3887118.pdf>.
- [38] Christian S Perone. Machine Learning :: Cosine Similarity for Vector Space Models (Part III) | Terra Incognita. URL <http://blog.christianperone.com/2013/09/machine-learning-cosine-similarity-for-vector-space-models-part-iii/>.

- [39] Peter Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE transactions on knowledge and data engineering*, 24(9):1537–1555, 2012.
- [40] Melanie Weis and Felix Naumann. Relationship-based duplicate detection. 2006.
- [41] Legacy Family Tree Genealogy Software: Features, . URL <https://legacyfamilytree.com/>.
- [42] Family Tree Maker, . URL [/ftm/](https://familytreemaker.com/).
- [43] Free Family Tree, Genealogy and Family History - MyHeritage, . URL <https://www.myheritage.com/>.
- [44] Ancestry® | Genealogy, Family Trees & Family History Records, . URL <https://www.ancestry.com/>.
- [45] Free Family History and Genealogy Records — FamilySearch.org, . URL <https://www.familysearch.org/>.
- [46] Ancestry Company Facts, . URL <https://www.ancestry.com/corporate/about-ancestry/company-facts>.
- [47] FamilySearch.org Genealogy - FamilySearch Wiki, . URL https://www.familysearch.org/wiki/en/FamilySearch.org#Online_Records.
- [48] FamilySearch. GEDCOM - FamilySearch Developer Center — FamilySearch.org. URL <https://www.familysearch.org/developers/docs/gedcom/>.
- [49] Dinastree. Info on GEDCOM - dynastree. URL <http://www.dynastree.com/infocentre/info-on-gedcom>.
- [50] Inc Intellectual Reserve. About GEDCOM X. URL <http://www.gedcomx.org/About.html>.
- [51] Graham Kirby, Conrad de Kerckhove, and Shumailov Ilia. Comparing Relational and Graph Databases for Pedigree Data Sets. URL https://socialhistory.org/sites/default/files/docs/kirby_et_al_-_database_comparison.pdf.

- [52] Lucía Pasarin Perea. *Analysis of alternatives to store genealogical trees using Graph Databases*. PhD thesis, Universitat Politècnica de Catalunya, 2013.
- [53] Javier Lozano Moya. *Repositori d'arbres genealògics en BD NoSQL*. B.S. thesis, Universitat Politècnica de Catalunya, 2015.
- [54] Hampus Lundberg. Fuzzy Matching and Merging of Family Trees using a Graph Database. 2015. URL <http://lup.lub.lu.se/student-papers/record/5154052>.
- [55] Julia Efremova, Bijan Ranjbar-Sahraei, Hossein Rahmani, Frans A. Oliehoek, Toon Calders, Karl Tuyls, and Gerhard Weiss. Multi-source entity resolution for genealogical data. In *Population Reconstruction*, pages 129–154. Springer, 2015.
- [56] 1.2. Graph database concepts - Chapter 1. Introduction, . URL <https://neo4j.com/docs/developer-manual/current/introduction/graphdb-concepts/#graphdb-neo4j-labels>.
- [57] Mark Richards. *Software Architecture Patterns*. O'Reilly Media, Incorporated, 2015. ISBN 978-1-4919-7143-7. Google-Books-ID: AtF2nQAACAAJ.
- [58] APOC User Guide 3.4.0.4, . URL <https://neo4j-contrib.github.io/neo4j-apoc-procedures/>.
- [59] Neo4j team. The Neo4j Graph Algorithms User Guide v3.4. URL <https://neo4j.com/docs/graph-algorithms/current/>.
- [60] Scikit fuzzy development team. skfuzzy 0.2 docs — skfuzzy v0.2 docs. URL <https://pythonhosted.org/scikit-fuzzy/>.
- [61] py_stringmatching Team. User Manual for py_stringmatching — py_stringmatching 0.3 documentation. URL http://anhaidgroup.github.io/py_stringmatching/v0.3.x/index.html.
- [62] Armin Ronacher. Welcome | Flask (A Python Microframework). URL <http://flask.pocoo.org/>.

- [63] technige. Deadlock in multiprocessing setup · Issue #474 · technige/py2neo, . URL <https://github.com/technige/py2neo/issues/474>.
- [64] technige. Using py2neo with multiprocessing lib. · Issue #348 · technige/py2neo, . URL <https://github.com/technige/py2neo/issues/348>.
- [65] Python Software Foundation. threading — Thread-based parallelism — Python 3.7.2rc1 documentation. URL <https://docs.python.org/3/library/threading.html>.