Universidad Central ¨Marta Abreu¨ de Las Villas

Facultad Matemática, Física y Computación

Departamento de Ciencia de la Computación

Trabajo de Diploma

Búsqueda por Palabras Clave en documentos XML utilizando el Sistema de Hormigas Max-Min.

Implementación y Validación.

Autor: Raúl López Orbea.

Tutor: Lic. Beatriz M. Méndez Hernández

Santa Clara, 2013.

"Año 55 de la Revolución"

El que suscribe, Raúl López Orbea, hago constar que el presente trabajo de diploma titulado **Búsqueda por Palabras Clave en documentos XML usando el Sistema de Hormigas Max-Min. Implementación y Validación** fue realizado en la Universidad Central "Marta Abreu" de Las Villas como parte de la culminación de estudios de la especialidad de Licenciatura en Ciencia de la Computación, autorizando a que el mismo sea utilizado por la Institución, para los fines que estime conveniente, tanto de forma parcial como total y que además no podrá ser presentado en eventos, ni publicados sin autorización de la Universidad.

_____

Firma del Autor.

Los abajo firmantes certificamos que el presente trabajo ha sido realizado según acuerdo de la dirección de nuestro centro y el mismo cumple con los requisitos que debe tener un trabajo de esta envergadura referido a la temática señalada.

_____ _____

Firma del Tutor.

Firma del Jefe de Departamento donde se defiende el Trabajo.

"Carpe diem quam minimun credula postero"

"Vive cada momento de tu vida como si fuese el último"

Quinto Horacio Flaco, poeta romano

A toda mi gran familia.

Agradezco sinceramente a todas las personas que han contribuido al desarrollo de este trabajo:

- ✓ *A mis padres por toda su ayuda, confianza y compresión, a mi hermana por soportarme en mis mejores días, y a mi familia por estar conmigo en todo.*
- ✓ *A mi tutora Beatriz Méndez Hernández por su ayuda y paciencia, por apoyarme en todo momento y siempre confiar en mí.*
- ✓ *A todos mis amigos de la Universidad que ya los considero mis hermanos especialmente a Oscar, David, Mario, Sergio, Eduardo Ronal (Piti), y Leandro (Make).*
- ✓ *A Danay y Yaumara por brindarme su amistad, que aunque no la disfrutamos desde un inicio de la carrera espero que perdure para siempre.*
- ✓ *A Roxy y Roxana por brindarme su ayuda y acogerme como si fuera parte de la familia.*
- ✓ *Al profesor Dr. Daniel Gálvez por brindarme su ayuda en los momentos más difíciles.*
- ✓ *A todos los profesores que me formaron profesionalmente a lo largo de la carrera.*

Resumen

Los documentos XML se han manifestado como el estándar por excelencia para la recuperación de datos y el intercambio de información en Internet. Actualmente, la búsqueda por palabras claves ha surgido como un popular paradigma para la recuperación de información sobre estos documentos. Hasta el momento esta forma de búsqueda siempre ha sido a través de algoritmos exhaustivos, solo existen dos métodos meta-heurísticos de la Optimización basada en Colonia de Hormigas para resolver este tipo de problema. En este trabajo se implementa el algoritmo Keyword Search Max-Min Ants System, que realiza la búsqueda por palabras clave en documentos XML basado en el algoritmo Sistema de Hormigas Max-Min, manteniendo la misma calidad de la solución pero mejorando el costo computacional en relación con otros algoritmos existentes. Finalmente se valida el algoritmo en cuanto a calidad de la solución y tiempo de ejecución mediante pruebas estadísticas y se desarrolla un framework que permite visualizar el funcionamiento del algoritmo, haciendo el trabajo con el mismo más fácil y entendible al usuario.

Palabras Clave: búsqueda por palabras clave, XML, MMAS, KSMMAS.

Abstract

XML documents have evolved to be the standard for excellence for data retrieve and exchange of information on the Internet. As such, keyword search has emerged as a popular paradigm for the information retrieval over XML data. Until the actual moment this way of search has always been through exhaustive algorithms, just there are two meta-heuristic methods of Ants Colony Optimization meta-heuristic to solve this kind of problem. This paper implements the Keyword Search Max-Min Ants System algorithm, which performs keyword search in XML documents based on the Max-Min Ants System algorithm, maintaining the same solution quality but improving the computational cost compared to other existing algorithms. Finally, the algorithm is validated in terms of solution quality and computational cost using statistical tests and a framework is developed that allows visualizing the algorithm working, making the work with easier and understandable to the user.

Keyword: keywords search, XML, MMAS, KSMMAS.

Tabla de Contenidos

INTRODUCCIÓN **1**

CAPÍTULO 1. "FUNDAMENTACIÓN TEÓRICA" **6**

1.1 XML Y LA DEFINICIÓN DEL PROBLEMA 6

1.2 LAS COLONIAS DE HORMIGAS NATURALES 7

1.3 LA HORMIGA ARTIFICIAL 9

1.4 DE LAS HORMIGAS NATURALES A LA META-HEURÍSTICA DE OPTIMIZACIÓN BASADA EN COLONIAS DE HORMIGAS 11

1.5 LA META-HEURÍSTICA DE OPTIMIZACIÓN BASADA EN COLONIAS DE HORMIGAS 12

1.6 MODELOS DE OPTIMIZACIÓN BASADA EN COLONIAS DE HORMIGAS 13

1.6.1 Algoritmo de Optimización Sistema de Hormigas *14*

1.6.2 Algoritmo que utiliza Sistema de Hormigas (KSAS) *19*

1.7 SEMÁNTICA BASADA EN EL MENOR ANCESTRO COMÚN MÁS CERCANO 21

1.7.1 Funciones usadas por los diferentes algoritmos *21*

1.7.2 Algoritmos que utilizan la semántica SLCA *23*

CAPÍTULO 2. "ANÁLISIS E IMPLEMENTACIÓN DEL ALGORITMO KSMMAS" **29**

2.1 MAX −MIN ANT SYSTEM (MMAS) 29

2.1.1 Actualización de los rastros de feromona *30*

2.1.2 Límites de rastros de feromona *32*

2.1.3 Inicialización y Reinicialización de rastros de feromona *34*

2.2 RECUPERACIÓN DE INFORMACIÓN EN DOCUMENTOS XML 36

2.3 ALGORITMO KSMMAS 38

2.4 PARÁMETROS PARA EL KSMMAS 40

2.4.1 Número de hormigas (m) *41*

*2.4.2 Tasa de evaporación (**ρ**)* *42*

2.4.3 Valores del exponente (α y β) *43*

2.5 ANÁLISIS DE COMPLEJIDAD 44

CAPÍTULO 3. "VALIDACIÓN DEL ALGORITMO KSMMAS" 49

3.1 MARCO EXPERIMENTAL ESTADÍSTICO 49

3.1.1 Descripción de las pruebas estadísticas utilizadas 51

3.2 ENTORNO COMPUTACIONAL 52

3.3 RESULTADOS EXPERIMENTALES 52

3.3.1 Dimensión 2 53

3.3.2 Dimensión 3 56

3.3.3 Dimensión 10 59

3.4 INTERFAZ DE USUARIO 63

CONCLUSIONES 68

RECOMENDACIONES 69

REFERENCIAS BIBLIOGRÁFICAS 70

ANEXOS 73

ANEXO I: EJECUCIÓN DEL ALGORITMO CON DOS PALABRAS CLAVE 73

ANEXO II: EJECUCIÓN DEL ALGORITMO CON TRES PALABRAS CLAVE 74

ANEXO III: EJECUCIÓN DEL ALGORITMO CON MÁS DE TRES PALABRAS CLAVE 75

ANEXO IV: EJECUCIÓN DEL ALGORITMO CON UNA PALABRA CLAVE ERRÓNEA 76

Índice de Tablas

Tabla 1. Parámetros del KSMMAS..... *44*
Tabla 2. Prueba no paramétrica de Wilcoxon. *52*
Tabla 3. Resultados para la dimensión 2..... *56*
Tabla 4. Resultados de la prueba de Wilcoxon para la dimensión 2..... *56*
Tabla 5. Resultados para la dimensión 3..... *59*
Tabla 6. Resultados de la prueba de Wilcoxon para la dimensión 3..... *59*
Tabla 7. Resultados para la dimensión 10..... *62*
Tabla 8. Resultados de la prueba de Wilcoxon para la dimensión 10..... *62*
Tabla 9. Resultados generales comparativos de la prueba de Wilcoxon..... *63*

Índice de Figuras

Figura 1. Ejemplo de un documento XML. *7*

Figura 2. El comportamiento de la colonia termina por obtener el camino más corto entre dos puntos..... *11*

Figura 3. Estructura del algoritmo AS. *17*

Figura 4. Ejemplo de árbol que representa un XML..... *22*

Figura 5. Pseudocódigo del algoritmo Stack..... *24*

Figura 6. Pseudocódigo del algoritmo Scan Eager. *26*

Figura 7. Pseudocódigo del algoritmo Indexed Lookup Eager. *26*

Figura 8. Un ejemplo de un árbol XML para ilustrar el funcionamiento del algoritmo MSLCA. . *27*

Figura 9. Pseudocódigo del algoritmo Incremental Multiway-SLCA (MSLCA). *28*

Figura 10. Ventana de bienvenida a la aplicación..... *64*

Figura 11. Aplicación desktop para interactuar con el algoritmo. *65*

Figura 12. Ejemplo de ejecución del algoritmo desde la aplicación con tres palabras clave..... *66*

Figura 13. Ejemplo de ejecución del algoritmo desde la aplicación con cuatro palabaras clave..... *67*

Introducción

Introducción

La investigación en el campo de las Reglas de Negocio es muy amplia. Dentro de este campo se encuentra el versionamiento de las reglas. Las reglas y versiones realizadas sobre una base de datos se encuentran en un documento XML cuya raíz solo representa a la base de datos sobre la que están aplicadas las reglas. Los hijos de esa raíz son las reglas y las versiones realizadas a las mismas son sus hijos y así sucesivamente. Cada vez que se necesita agregar una regla, cosa que ocurre con mucha frecuencia, es necesario verificar si la nueva regla no ha estado activa con anterioridad, para esto se hace una búsqueda en el XML, ya que no es óptimo agregar la regla si ella ya existe. Para este problema se necesita un algoritmo que realice en el XML una búsqueda por palabras clave.

Existen problemas de optimización combinatoria complejos en diversos campos de la investigación. A menudo estos problemas son muy difíciles de resolver en la práctica. El estudio de esta dificultad inherente para resolver dichos problemas tiene cabida en el campo de la teoría de la Ciencia de la Computación, ya que muchos de ellos pertenecen a la clase de problemas NP-duros, lo que significa que no existe un algoritmo conocido que los resuelva en un tiempo polinomial (Jonson 1979).

Día tras día, siguen apareciendo nuevos problemas de este tipo, lo que ha dado lugar a que se hayan realizado muchas propuestas de algoritmos para tratar de solucionarlos. Las técnicas existentes se pueden clasificar básicamente en algoritmos *exactos* o *aproximados*.

Los *algoritmos exactos* intentan encontrar una solución óptima y demostrar que la solución obtenida es de hecho la óptima global; estos algoritmos incluyen técnicas como, por ejemplo: procesos de vuelta atrás (Backtracking), ramificación y poda (Branch and Bound), programación dinámica, etc.(Steiglitz 1982, Bratley 1996). Debido a que los algoritmos exactos muestran un rendimiento pobre para muchos problemas se han desarrollado múltiples tipos de algoritmos aproximados que proporcionan soluciones de alta calidad para estos problemas combinatorios (aunque no necesariamente la óptima) en un tiempo computacional breve.

Los *algoritmos aproximados* se pueden clasificar en dos tipos principales: *algoritmos constructivos y algoritmos de búsqueda local*. Los primeros se basan en generar soluciones desde cero añadiendo componentes a cada solución paso a paso. Un ejemplo bien conocido son las heurísticas de construcción voraz o heurísticas greedy (Bratley 1996). Su gran ventaja es la velocidad: normalmente son muy rápidas y, además, a menudo devuelven soluciones razonablemente buenas, la idea más genérica del término heurístico está relacionada con la tarea de resolver inteligentemente problemas reales usando el conocimiento disponible.

Desafortunadamente, los algoritmos de mejora iterativos pueden estancarse en soluciones de baja calidad (óptimos locales muy lejanos al óptimo global). Para permitir una mejora adicional en la calidad de las soluciones, la investigación en este campo en las últimas dos décadas ha centrado su atención en el diseño de técnicas de propósito general para guiar la construcción de soluciones o la búsqueda local en las distintas heurísticas. Estas técnicas se llaman comúnmente *meta-heurísticas* (Kelly 1996, S. Voss 1999, Kochenberger 2003) y consisten en conceptos generales empleados para definir métodos heurísticos.

Dicho de otra manera, una meta-heurística puede verse como un marco de trabajo general referido a algoritmos que puede aplicarse a diversos problemas de optimización (combinatoria) con pocos cambios significativos si ya existe previamente algún método heurístico específico para el problema. De hecho, las meta-heurísticas son ampliamente reconocidas como una de las mejores aproximaciones para atacar los problemas de optimización combinatoria. (Reeves 1995, Lenstra 1997, Fogel 2000).

Entre las meta-heurísticas más recientes se encuentra la Optimización basada en Colonias de Hormigas (ACO, del inglés, Ant Colony Optimization) que fue presentada por Marco Dorigo en su tesis doctoral en 1992 (Dorigo 1992). Básicamente, se trata de una técnica probabilística para resolver problemas computacionalmente complejos que pueden ser reducidos a la búsqueda de buenos caminos en grafos. ACO se inspira directamente en el comportamiento de las colonias reales de hormigas para solucionar problemas de

optimización combinatoria. Se basan en una colonia de hormigas artificiales, esto es, agentes computacionales simples que trabajan de manera cooperativa y se comunican mediante rastros de feromona artificiales.

Un problema donde es aplicada esta meta-heurística es en la investigación en los XML, que hoy en día se han convertido en el estándar para la representación de datos. Debido a la flexibilidad estructural y heterogeneidad de los XML es difícil para el usuario hacer consultas estructuradas que devuelvan resultados que les satisfagan. Por esto, la búsqueda por palabras clave ha surgido como un popular paradigma para la recuperación de información sobre documentos XML. Uno de los méritos más significativos de la búsqueda por palabras clave en documentos XML es la simplicidad con que esta se realiza, ya que los usuarios no necesitan aprender un complejo lenguaje de consulta o conocer la estructura interna del XML. Sin embargo, este formato simple de consultas puede no ser preciso y puede devolver un gran número de resultados que pueden ser clasificados en diferentes tipos y de los cuales muy pocos serían de interés para el usuario.

Para los documentos XML, donde los datos son vistos como un árbol estructurado jerárquicamente, una semántica natural para la búsqueda por palabras clave sería devolver todos los nodos que contengan todas las palabras clave en sus subárboles. Sin embargo, esto traería consigo que se le devolvieran al usuario una gran cantidad de nodos, muchos de los cuales solo servirían de nodos de enlace a los que realmente contienen las palabras clave (Sun, Chan et al. 2007).

Un resultado efectivo de una consulta debería cumplir con dos condiciones:

1. Los subárboles devueltos contienen en sus nodos todas las palabras clave a partir del nodo raíz.
2. No existe ningún nodo descendiente que cumpla la condición anterior.

A cada nodo interno *v* del árbol que representa al XML se le llama *nodo elemento*, y a la etiqueta que le corresponde, *etiq (v)*. A cada nodo hoja le corresponde un dato, y se le llama *nodo valor*.

El **objetivo general** de esta investigación es implementar y validar la búsqueda por palabras clave en documentos XML usando de la meta-heurística ACO el algoritmo Sistema de Hormigas Max-Min (MMAS, del inglés Max-Min Ant System).

Para lograr dicho objetivo dividiremos el trabajo en los siguientes **objetivos específicos:**

- Realizar un estudio de algoritmos anteriores que realizan la búsqueda por palabras clave en documentos XML.
- Implementar un algoritmo que realice la búsqueda por palabras clave en documentos XML basado en el algoritmo MMAS.
- Validar el algoritmo en cuanto a calidad de la solución y tiempo de ejecución mediante pruebas estadísticas.
- Desarrollar un framework que permita visualizar el funcionamiento del algoritmo.

Preguntas de Investigación:

1. ¿Qué ventajas trae utilizar una búsqueda heurística sobre una exhaustiva?
2. ¿Cuál de los algoritmos heurísticos que realizan la búsqueda es mejor?

Luego de haber construido el marco teórico se formuló la siguiente **hipótesis de investigación:**

Aplicando un algoritmo basado en MMAS se logra una mejora en cuanto a tiempo de ejecución en la búsqueda por palabras clave en XML sin dañar de forma estadísticamente significativa la calidad de la solución.

Justificación de la Investigación:

Existen diferentes algoritmos que permiten identificar nodos significativos en documentos XML, donde estos son vistos como árboles y devuelven el mejor subárbol que contiene todas las palabras clave que el usuario busca. La mayoría de estos algoritmos realizan la búsqueda de manera exhaustiva lo que trae consigo que para un XML extenso se demoren mucho por lo que vale la pena intentar realizar esta búsqueda utilizando un algoritmo que

realice una búsqueda heurística que en XML grandes debe reducir el tiempo de la búsqueda.

El trabajo está estructurado de la siguiente manera:

En el capítulo 1 se exponen algunos conceptos, definiciones y funciones para la teoría de árboles y se presentan algoritmos que basados en algunas de las semánticas de teoría de árboles realizan la búsqueda por palabras clave en documentos XML además se describe un algoritmos basado en la meta-heurística ACO que resuelve el problema.

En el capítulo 2 se describe un algoritmo de la meta-heurística ACO, el MMAS, además se implementa el algoritmo KSMMAS, que realiza la búsqueda por palabras clave en documentos XML usando el algoritmo Sistema de Hormigas Max-Min (MMAS) de la meta-heurística ACO.

En el capítulo 3 se valida el algoritmo mencionado en el capítulo 2 comparándolo con el Algoritmo Búsqueda por Palabras Clave basado en Sistema de Hormigas (KSAS, del inglés, Keyword Search Ants System) en cuanto a calidad de la solución y en cuanto a tiempo de ejecución mediante pruebas estadísticas como la prueba de Wilcoxon. También se realiza una aplicación visual para una mejor interacción entre los usuarios y el algoritmo.

Capítulo 1.
"Fundamentación Teórica"

Capítulo 1. "Fundamentación Teórica"

Los documentos XML han llegado a convertirse en el estándar para el intercambio de información en Internet (Hristidis V. 2003, Lin Guo 2003, Yunyao Li 2004, Yu Xu 2005, Guoliang Li 2007, Liu 2007, Sun, Chan et al. 2007, Jianxin Li 2010). La mayoría de los algoritmos para resolver la búsqueda por palabras clave en documentos XML son exhaustivos por lo que requieren demasiado tiempo cuando los XML son muy grandes, por esto se decidió investigar la meta-heurística ACO como una forma alternativa de resolver dicho problema. Por lo antes planteado en este capítulo se hace un breve estudio de los algoritmos de esta meta-heurística y de los algoritmos que ya existen para resolver la búsqueda por palabras clave en documentos XML.

1.1 XML y la definición del problema

Un XML es modelado como un árbol arraigado y etiquetado como el que se muestra en la figura 1. Cada nodo interno de *v* en un árbol corresponde a un elemento XML, llamado nodo elemento, y está etiquetado con una *etiq (v)*. Cada nodo hoja del árbol corresponde a un dato que es un valor, llamado nodo valor. Por ejemplo, en la figura 1 "*Dean*" y "*Title*" son nodos elementos, "*John*" y "*Ben*" son nodos valor. En este modelo los nodos atributos son modelados como hijos de un nodo elemento asociado, y no se distinguen de los nodos elementos (Sun, Chan et al. 2007).

A cada nodo (elemento o valor) en el árbol XML se le asigna un identificador único, *Dewey ID*. Un nodo v_1 precede a otro nodo v_2 en un recorrido en preorden por el árbol, si y solo si $pred\ (v_1) < pred\ (v_2)$. La relación $<$ entre dos identificadores funciona como si comparáramos dos secuencias. Además la información que preserva este identificador también puede ser usada para detectar hermanos y relaciones ancestro-descendiente entre los distintos nodos.

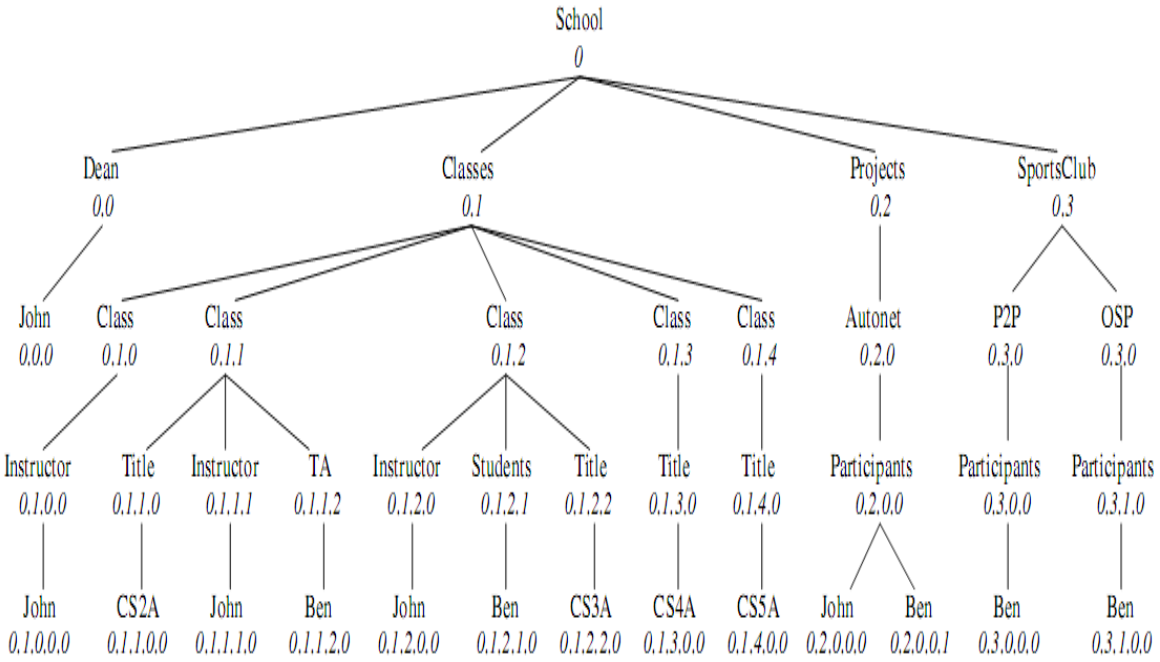


Figura 1. Ejemplo de un documento XML.

- Un nodo *u* es un hermano del nodo *v* si y solo si *pred (u)* difiere del *pred (v)* solo en el último componente. Por ejemplo, *0.1.1.0 (Title)* y *0.1.1.1 (Instructor)* son nodos hermanos, pero *0.1.1 (Class)* y *0.1.1.1 (Instructor)* no son hermanos.
- Un nodo *u* es un ancestro de un nodo *v* si y solo si *pred (u)* es un prefijo del *pred (v)*. Por ejemplo, *0.1 (Classes)* es un ancestro de *0.1.2.0.0 (John).*

Para simplificar, se usa $u < v$ para denotar que *pred (u)* $<$ *pred (v)*. $u \leq v$ denota que $u < v$ o $u = v$. También se usa $u \prec v$ para denotar que *u* es un ancestro de *v*, o lo que es lo mismo, que *v* es un descendiente de *u*. Nótese que si $u \prec v$ entonces $u < v$, pero en sentido contrario no se cumple la implicación.

1.2 Las Colonias de Hormigas Naturales

Las hormigas son insectos sociales que viven en colonias y que, debido a su colaboración mutua, son capaces de mostrar comportamientos complejos y realizar tareas difíciles desde el punto de vista de una hormiga individual. Un aspecto interesante del comportamiento de

muchas especies de hormigas es su habilidad para encontrar los caminos más cortos entre su hormiguero y las fuentes de alimento. Este hecho es especialmente interesante si se tiene en cuenta que muchas de las especies de hormigas son casi ciegas, lo que evita el uso de pistas visuales.

Mientras que se mueven entre el hormiguero y la fuente de alimento, algunas especies de hormigas depositan una sustancia química denominada feromona (una sustancia que puede "olerse"). Si no se encuentra ningún rastro de feromona, las hormigas se mueven de manera básicamente aleatoria, pero cuando existe feromona depositada, tienen mayor tendencia a seguir el rastro. De hecho, los experimentos realizados por biólogos han demostrado que las hormigas prefieren de manera probabilística los caminos marcados con una concentración superior de feromona (J. M. Pasteels 1987, S. Goss 1989).

En la práctica, la elección entre distintos caminos toma lugar cuando varios caminos se cruzan. Entonces, las hormigas eligen el camino a seguir con una decisión probabilística sesgada por la cantidad de feromona: cuanto más fuerte es el rastro de feromona, mayor es la probabilidad de elegirlo. Puesto que las hormigas depositan feromona en el camino que siguen, este comportamiento lleva a un proceso de autorrefuerzo que concluye con la formación de rastros señalados por una concentración de feromona elevada. Este comportamiento permite además a las hormigas encontrar los caminos más cortos entre su hormiguero y la fuente del alimento (S. Goss 1989). (Nótese que las hormigas solo se comunican de manera indirecta, a través de modificaciones del espacio físico que perciben. Esta forma de comunicación se denomina *Estimergia Artificial*.)

En la Figura 2 se ilustra cómo este mecanismo permite a las hormigas encontrar el camino más corto. Inicialmente no existe ningún rastro de feromona en el medio y, cuando una hormiga llega a una intersección, elige de manera aleatoria una de las bifurcaciones posibles. Según transcurre el tiempo y mientras que las hormigas están recorriendo los caminos más prometedores, estos van recibiendo una cantidad superior de feromona. Esto ocurre gracias a que al ser los caminos más cortos, las hormigas que los siguen consiguen

encontrar la comida más rápidamente, por lo que comienzan su viaje de retorno antes. Entonces, en el camino más corto habrá un rastro de feromona ligeramente superior y, por lo tanto, las decisiones de las siguientes hormigas estarán dirigidas en mayor medida a dicho camino. Además, este camino recibirá una proporción mayor de feromona por las hormigas que vuelven por él que por las que vuelven por el camino más largo. Este proceso finaliza haciendo que la probabilidad de que una hormiga escoja el camino más corto aumente progresivamente y que al final el recorrido de la colonia converja al más corto de todos los caminos posibles.

Esta convergencia se complementa con la acción del entorno natural que provoca que la feromona se evapore transcurrido un cierto tiempo. Así, los caminos menos prometedores pierden progresivamente feromona porque son visitados cada vez por menos hormigas. Sin embargo, algunos estudios biológicos han demostrado que los rastros de feromona son muy persistentes, la feromona puede permanecer desde unas pocas horas hasta varios meses dependiendo de varios aspectos distintos, como la especie de las hormigas, el tipo de suelo, etc. (E. Bonabeau 1999). Lo que provoca una menor influencia del efecto de la evaporación en el proceso de búsqueda del camino más corto.

En (E. Bonabeau 1999), numerosos experimentos muestran que, debido a la gran persistencia de feromona, es difícil que las hormigas "olviden" un camino que tiene un alto nivel de feromona aunque hayan encontrado un camino aún más corto. Hay que tener en cuenta que si se traslada este comportamiento directamente al ordenador para diseñar un algoritmo de búsqueda podemos encontrarnos con que se quede rápidamente estancado en un óptimo local.

1.3 La Hormiga Artificial

La hormiga artificial es un agente computacional simple que intenta construir soluciones posibles al problema explotando los rastros de feromona disponibles y la información heurística. Sin embargo, en algunos problemas, puede también construir soluciones no

válidas que podrán ser penalizadas dependiendo de lo inadecuado de la solución. Tiene las siguientes propiedades (Caro 1999; Stützle 2003):

- Busca soluciones válidas de costo mínimo para el problema a solucionar.
- Tiene una memoria *L* que almacena información sobre el camino seguido hasta el momento, esto es, *L* almacena la secuencia generada. Esta memoria puede usarse para:

  (i) Construir soluciones válidas.

  (ii) Evaluar la solución generada.

  (iii) Reconstruir el camino que ha seguido la hormiga.
- Tiene un estado inicial $\sigma_{inicial}$, que normalmente se corresponde con una secuencia unitaria y una o más condiciones *t* de parada asociadas.
- Comienza en el estado inicial y se mueve siguiendo estados válidos, construyendo la solución asociada incrementalmente.
- Cuando está en un estado $\sigma_r =< \sigma_{r-1}, r >$ (es decir, actualmente está localizada en el nodo *r* y ha seguido previamente la secuencia σ_{r-1} , puede moverse a cualquier nodo *s* de su vecindario posible *N(r),* definido como

  $N(r) = \{s \mid (a_{rs} \in A) \text{ y } (<\sigma_r, s> \in \Delta\sim)\}$.
- El movimiento se lleva a cabo aplicando una regla de transición, que es función de los rastros de feromona que están disponibles localmente, de los valores heurísticos de la memoria privada de la hormiga y de las restricciones del problema.

  La regla probabilística de transición más habitual define la probabilidad con la que la hormiga *k,* situada en el nodo *r,* decide moverse hacia el nodo *s*.
- Cuando durante el procedimiento de construcción una hormiga se mueve desde el nodo *r* hasta el *s*, puede actualizar el rastro de feromona t_{rs} asociado al arco a_{rs} . Este proceso se llama *actualización en línea* de los rastros de feromona paso a paso.

- El procedimiento de construcción acaba cuando se satisface alguna condición de parada, normalmente cuando se alcanza un estado objetivo.
- Una vez que la hormiga ha construido la solución puede reconstruir el camino recorrido y actualizar los rastros de feromona de los arcos/componentes visitados/as utilizando un proceso llamado *actualización en línea a posteriori*. Este es el único mecanismo de comunicación entre las hormigas, utilizando la estructura de datos que almacena los niveles de cada arco/componente (memoria compartida).

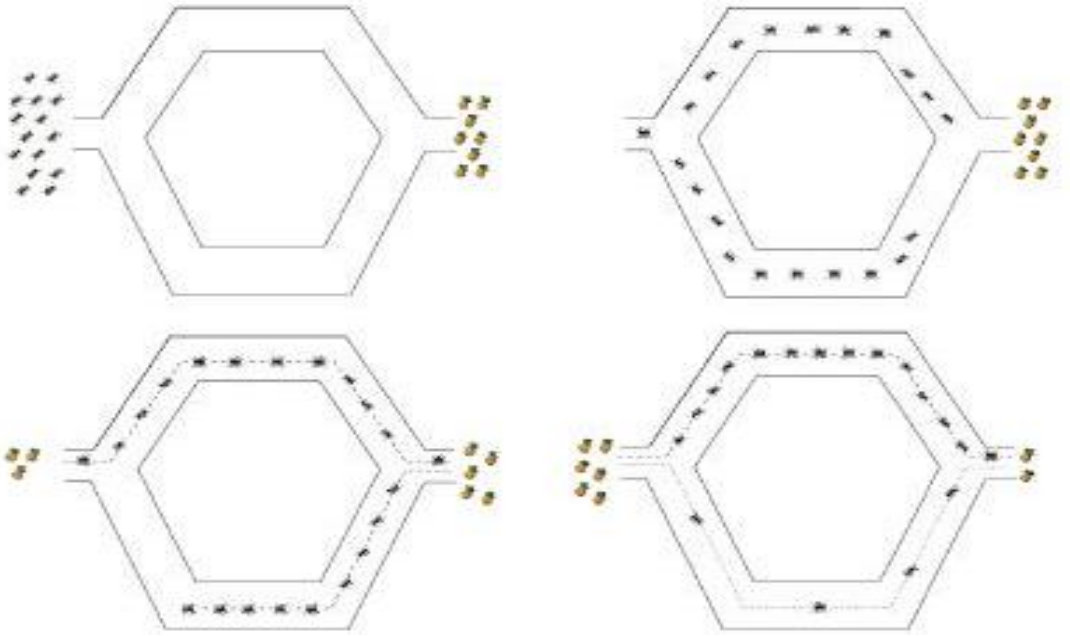


Figura 2. El comportamiento de la colonia termina por obtener el camino más corto entre dos puntos.

1.4 De las Hormigas Naturales a la Meta-heurística de Optimización Basada en Colonias de Hormigas

Los algoritmos de ACO se inspiran directamente en el comportamiento de las colonias reales de hormigas para solucionar problemas de optimización combinatoria.

Se basan en una colonia de hormigas artificiales, esto es, unos agentes computacionales simples que trabajan de manera cooperativa y se comunican mediante rastros de feromona artificiales.

Los algoritmos de ACO son esencialmente *algoritmos constructivos*: que en cada iteración del algoritmo, cada hormiga construye una solución al problema recorriendo un grafo de construcción. Cada arista del grafo, que representa los posibles pasos que la hormiga puede dar, tiene asociada dos tipos de información que guían el movimiento de la hormiga:

- **Información heurística**: Que mide la preferencia heurística de moverse desde el nodo *r* hasta el nodo *s*, o sea, de recorrer la arista a_{rs}. Se nota por n_{rs} Las hormigas no modifican esta información durante la ejecución del algoritmo.
- **Información de los rastros de feromona artificiales**: Que mide la "deseabilidad aprendida" del movimiento de *r* a *s*. Imita a la feromona real que depositan las hormigas naturales. Esta información se modifica durante la ejecución del algoritmo dependiendo de las soluciones encontradas por las hormigas.

1.5 La meta-heurística de Optimización Basada en Colonias de Hormigas

Una colonia de hormigas (artificiales) construye soluciones de forma concurrente y asíncrona para un problema discreto de optimización dado moviéndose en el grafo de representación del problema, donde cada camino factible implica una solución del problema (Dorigo 2001). Las hormigas se mueven aplicando una regla de decisión estocástica que explota los rastros de feromona. Al moverse, las hormigas construyen incrementalmente soluciones al problema de optimización. Una vez que la hormiga ha construido una solución, o mientras la construye la hormiga evalúa su solución y deposita feromona en los componentes del grafo que ha utilizado. Este rastro dirigirá la búsqueda de las hormigas en el futuro.

Además de la actividad de las hormigas, un algoritmo utilizando ACO incluye dos procedimientos adicionales: *evaporación del rastro de feromona* y las *acciones del demonio.*

La *evaporación de la feromona* es el proceso mediante el cual la intensidad del rastro de feromona en los componentes decrece en el tiempo. Desde un punto de vista práctico, la evaporación de feromona es necesaria para evitar una convergencia rápida del algoritmo hacia una región sub-óptima favoreciendo la exploración de nuevas áreas.

Las *acciones del demonio* pueden usarse para implementar acciones centralizadas que no pueden ser realizadas por hormigas individuales.

Ejemplos son la activación de un procedimiento de optimización local, o la colección de información global que puede ser usada para decidir si conviene o no depositar feromona adicional para influir en el proceso de búsqueda desde una perspectiva global. Como un ejemplo práctico, el demonio puede depositar feromona extra en los componentes utilizados por la hormiga que construyo la mejor solución.

A continuación se muestra el pseudocódigo de la meta-heurística ACO.

```
1. Procedimiento Meta-heurística_ACO
     2. ProgramarActividades          {posiblemente en paralelo}
          DirigirActividadHormiga ()
          EvaporarFeromona ()
          AcionesDemonio ()           {opcional}
        FinProgramarActividades
3. Fin Meta-heurística_ACO
```

1.6 Modelos de Optimización Basada en Colonias de Hormigas

Entre los algoritmos de ACO disponibles para problemas de optimización combinatoria NP-duros, se encuentran:

- **Sistema de Hormigas** (AS, del inglés, Ant System) (M. Dorigo 1996).
- Sistema de Colonia de Hormigas (ACS, del inglés, Ant Colony System)(Dorigo 1997).
- **Sistema de Hormigas Max-Min** (MMAS, del inglés, Max-Min Ant System) (Hoos 2000).
- AS con ordenación (RBAS, del inglés, Rank-Based Ant System)(B. Bullnheimer 1999).
- Sistema de la Mejor-Peor Hormiga (BWAS, del inglés, Best-Worst Ant System) (O. Cordón 1999).

Como es de observar haremos énfasis en el algoritmo **Sistema de Hormigas** que no es más que un algoritmo distribuido que consiste en un conjunto de agentes cooperativos que intercambian información de manera indirecta, el paralelismo es inherente al funcionamiento del algoritmo, es decir, el comportamiento de una hormiga es independiente de todas las demás durante la misma iteración. Además analizaremos el algoritmo **Sistema de Hormigas Max-Min**, esta estrategia puede dar origen a una convergencia prematura en un resultado no óptimo, pero se consideraba oportuno que los mejores resultados ejercieran una influencia mayor en el algoritmo utilizado. Para evitar el aumento desmesurado del valor de feromona en rutas no óptimas y por lo tanto la convergencia en estas rutas, se introdujeron los conceptos de máximo y mínimo, que se referían a los valores que podían alcanzar las rutas en cuanto a la cantidad de feromona depositada. El nivel mínimo pone límite al nivel de exploración, para que este siempre tenga posibilidades de ejecución.

1.6.1 Algoritmo de Optimización Sistema de Hormigas

El **Sistema de Hormigas**, desarrollado por Dorigo, Maniezzo y Colorni en 1991, fue el primer algoritmo de ACO. Inicialmente, se presentaron tres variantes distintas: AS-densidad, AS-cantidad y AS-ciclo, que se diferenciaban en la manera en que se

actualizaban los rastros de feromona. En los dos primeros, las hormigas depositaban feromona mientras que construían sus soluciones (esto es, aplicaban una *actualización en-línea paso a paso* de feromona), con la diferencia de que la cantidad de feromona depositada en el AS-densidad es constante, mientras que la depositada en AS-cantidad dependía directamente de la deseabilidad heurística de la transición η_{rs}. Por último, en AS-ciclo, el almacenaje de la feromona se lleva a cabo una vez que la solución está completa (*actualización en línea a posteriori* de feromona). Esta última variante era la que obtenía mejores resultados y es por tanto la que se conoce como **AS** en la literatura.

El AS se caracteriza por el hecho de que la actualización de feromona se realiza una vez que todas las hormigas han completado sus soluciones, y se lleva a cabo como sigue: primero, todos los rastros de feromona se reducen en un factor constante, implementándose de esta manera la evaporación de feromona. A continuación cada hormiga de la colonia deposita una cantidad de feromona en función de la calidad de su solución. Inicialmente, el AS no usaba ninguna acción de mejora de deposición de feromona considerando las trayectorias generadas, sin embargo es relativamente fácil, por ejemplo, añadir un procedimiento de búsqueda local para refinar las soluciones generadas por las hormigas.

Las soluciones en el AS se construyen como sigue. En cada paso de construcción, una hormiga *k* escoge ir al siguiente nodo con una probabilidad que se calcula con la siguiente ecuación:

$$\rho_{rs}^{k} = \begin{cases} \dfrac{[\tau_{rs}]^{\alpha} \cdot [n_{rs}]^{\beta}}{\sum_{s \epsilon N_r^k} [\tau_{rs}]^{\alpha} \cdot [n_{rs}]^{\beta}} & si \;\; s \epsilon N_k(r) \\ 0 & en\ otro\ caso \end{cases} \tag{1.1}$$

Donde $N_k(r)$ es el vecindario alcanzable por la hormiga *k* cuando se encuentra en el nodo *r* (nodos que aún no visita (M. Dorigo 1996), también se puede ver como una lista tabú L_k que salva los nodos previamente visitados para prohibirle a la hormiga visitarlos nuevamente antes de *n* iteraciones (Dorigo 1997), lista que es posteriormente limpiada. Los parámetros α y β ponderan la importancia relativa de los rastros de feromona y la

información heurística, respectivamente. Cada hormiga *k* almacena la secuencia que ha seguido hasta el momento y su memoria L_k, tal como se explicó antes, se utiliza para determinar $N_k(r)$ en cada paso de construcción.

Volviendo a los parámetros α y β, su función es la que sigue: si α = 0, aquellos nodos con una preferencia heurística mejor tienen una mayor probabilidad de ser escogidos, haciendo el algoritmo muy similar a un algoritmo voraz probabilístico clásico (con múltiples puntos de partida en caso de que las hormigas estén situadas en nodos distintos al comienzo de cada iteración). Sin embargo, si β = 0, sólo se tienen en cuenta los rastros de feromona para guiar el proceso constructivo, lo que puede causar un rápido estancamiento, esto es, una situación en la que los rastros de feromona asociados a una solución son ligeramente superiores que el resto, provocando por tanto que las hormigas siempre construyan las mismas soluciones, normalmente óptimos locales. Por tanto es preciso establecer una adecuada proporción entre la información heurística y la información de los rastros de feromona.

Como se ha dicho, el depósito de feromona se realiza una vez que todas las hormigas han acabado de construir sus soluciones y se hace de la siguiente manera: primero, los rastros de feromona asociados a cada arco se evaporan reduciendo todos los rastros de feromona en un factor constante:

$$\tau_{rs} \leftarrow (1 - \rho) \cdot \tau_{rs} \tag{1.2}$$

Donde $\rho \in (0; 1]$ es la tasa de evaporación. El siguiente paso de cada hormiga es recorrer de nuevo el camino que ha seguido (el camino está almacenado en su memoria local L_k) y deposita una cantidad de feromona $\Delta\tau_{rs}^{t}$ en cada conexión por la que ha viajado:

$$\tau_{rs} \leftarrow \Delta\tau_{rs}^{k}, \forall \alpha_{rs} \in S_k \tag{1.3}$$

Donde $\Delta\tau_{rs}^{t} = f(C(S_k))$, es decir, la cantidad de feromona que se deposita depende de la calidad $C(S_k)$ de la solución S_k construida por la hormiga *k*.

En resumen en el algoritmo podemos observar que este posee cuatro pasos básicos a seguir, estos son:

- **Inicialización de parámetros:** En el cual se fijan los valores de feromona iniciales, número de hormigas y los pesos de la información heurística.
- **Programación de actividades:** Que comprende cuatro acciones básicas, correspondientes a la generación y puesta en marcha de las hormigas, evaporación de feromona, las acciones del demonio y el depósito de feromona.
- **Actualización de la memoria de la hormiga:** En el cual se especifica el lugar en el cual la hormiga empieza su camino.
- **Cálculo de la probabilidad de transición y aplicación de la política de decisión:** Este paso se realiza en base al rastro de feromona existente y la información heurística.

Una forma en que se pudiera describir el desarrollo de este algoritmo sería:

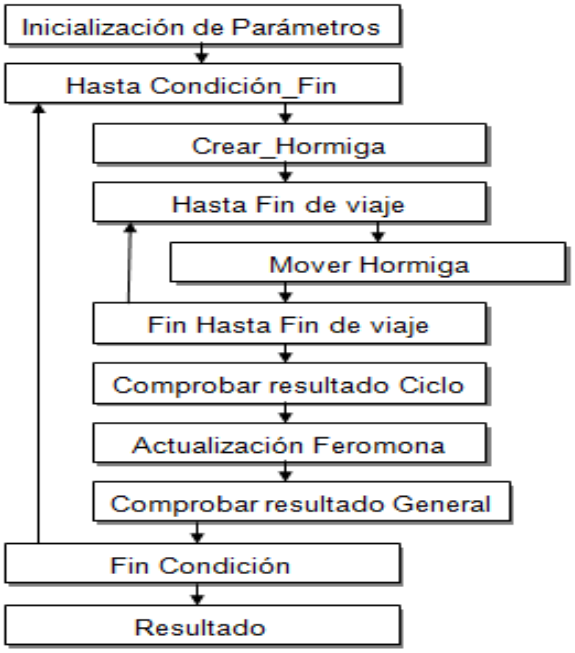


Figura 3. Estructura del algoritmo AS.

Describiendo el comportamiento del algoritmo en palabras, es una ejecución continua hasta cumplir una condición de parada, donde en cada ciclo se crea una hormiga que va cambiando de estado o nodo, de acuerdo a una probabilidad resultante de una función heurística y de la cantidad de feromona detectada en ese recorrido concreto ("**Mover Hormiga**"). Se puede decir, que la hormiga es un elemento simple, que se desplaza basándose en información local, tanto heurística, como la aportada por los elementos de la colonia (feromona). Una vez determinada la probabilidad de las diferentes posibles rutas a tomar, la hormiga decide en función a estos valores ("**Elección del movimiento**"). Una vez elegido el nodo a visitar, se añade este a la lista de nodos visitados (camino de la hormiga), repitiendo este proceso hasta finalizar la visita a todos los nodos. Una vez terminado un ciclo, se procede a realizar la evaporación de feromona depositada en los arcos de la red y el depósito de feromona sobre la solución obtenida ("**Actualización de Feromona**"). El ciclo se repite hasta la condición de fin establecida para el algoritmo.

A continuación mostramos el pseudocódigo de este método.

```
1. Procedimiento AS ()
      2. Inicializar parámetros
      3. Mientras condición hacer
            4. Inicializar_NuevaHormiga ()
            5. Mientras estado != estado fin
               6. Por cada_arco_posible_movimiento
                        Calcular probabilidad_elección
                  Fin Por
               7. Siguiente_posición = política_posición
               8. Lista_Posiciones +=  Siguiente_posición
              Fin Mientras
         9. Realizar_evaporización
```

```
        10. Depósito_Feromona (Lista_Posiciones)
    Fin Mientras
11. Fin Procedimiento
```

1.6.2 Algoritmo que utiliza Sistema de Hormigas (KSAS)

El KSAS, del inglés Keyword Search Ants System está basado en el algoritmo AS, cuya esencia es la búsqueda por palabras clave en documentos XML. El algoritmo AS realiza una búsqueda en un determinado ambiente, generalmente sobre grafos no dirigidos y completos, estos grafos son su espacio de búsqueda que van a explotando hasta encontrar la mejor solución o al menos una bastante buena.

El proceso de cómo se aplica este algoritmo a la búsqueda por palabras clave comienza escogiendo aleatoriamente el nodo inicial para cada hormiga de la colonia, las cuales son posicionadas en ellos para luego empezar a recorrer el árbol virtual formado del XML en cuestión. Los elementos que van explorando se van almacenando en su lista tabú, en caso de que se repita algún elemento pasan hacia una lista temporal, cada elemento del XML puede ser visitado por la hormiga solo ***cantidad de hijos + 1***, evitándose de esta manera la formación de ciclos. Luego se evaporan los rastros de feromona y seguidamente se actualiza del rastro.

Este proceso se ejecuta hasta que se haya recorrido todo el árbol o se encuentren todas las palabras clave que se están buscando.

Cuenta con tres clases de programación en las cuales se comprende el algoritmo; una para tratar todo lo relativo a los recursos de cada hormiga, la ***clase medio***, esta contiene la información del problema además de cómo trabajar con las feromonas, como evaporarlas, inicializarlas, entre otros métodos auxiliares.

Otra es la ***clase KSAS***, considerada como la clase principal que es la encargada de ejecutar el método al igual que imprimir el camino resultante, en esta clase le son pasados los

parámetros de entrada al algoritmo así como el documento XML al que se le realizará la búsqueda.

También tenemos una ***clase ants*** que es donde se define el movimiento de una hormiga, el método más importante con que cuenta es wayAs, el cual recibe como parámetros un objeto de la clase medio (para controlar las feromonas) y las constantes α y β para el cálculo de la función de probabilidad vista anteriormente. Este método devuelve un objeto de tipo Element que la información que nos da es a que nodo se mueve la hormiga en la próxima iteración. La función de visibilidad usada en la probabilidad se trata de dos maneras diferentes teniendo en cuenta de si el nodo candidato al que puede moverme es una palabra clave o no.

Descripción del algoritmo en pseudocódigo.

```
1. Fase de inicialización
        Inicializar contador de ciclos NC
        Inicializar total de Hormigas
        Inicializar solución global L_global = documento.
        Inicializar arreglo de feromona t_(i) = t_0
        Cargar documento XML
2. Repetir NC veces
    Para cada hormiga
       Elegir elemento inicial
    Repetir hasta llenar tabú_k de cada hormiga
    Para cada hormiga:
     Elegir próximo elemento a ser visitado según ecuación (1.1)
     Mover la hormiga al próximo elemento
     Insertar operación seleccionada en tabú_k
```

```
    Evaporar los rastros de feromona según (1.2)
    Repetir para las k hormigas
        Calcular calidad de la solución (Lk)
            Si  (Lk > Lglobal )
                Actualizar Lglobal con Lk
        Incrementar los valores de feromona asociados a Lk según (1.3)
3.  Imprimir subárbol más pequeño Lglobal
```

Lo primero que se obtiene es el vecindario al cual la hormiga puede moverse, después se calcula la probabilidad de acuerdo al tipo de caso en que nos encontremos. Si todo el vecindario alcanzable por la hormiga ya fue visitado nos movemos al padre, teniendo en cuenta que no se vuelve a agregar al camino de la hormiga y ese es el *Element* que devuelve el método, si quedan alguno o algunos por visitar nos movemos utilizando el método tradicional de la ruleta.

1.7 Semántica basada en el Menor Ancestro Común más Cercano

La idea de realizar la búsqueda de palabras clave basada en la semántica del Menor Ancestro Común más Cercano (**SLCA**, por sus siglas en inglés) consiste en que, cada nodo en *T* puede ser visto como una entidad en el mundo. Si *u* es un ancestro de *v*, podemos entender que la entidad representada por *v* pertenece a la entidad que representa *u*. Para una consulta por palabras clave, es más deseable devolver la entidad más específica que posea todas las palabras clave, entre todas las entidades retornadas, no debe existir ninguna relación ancestro-descendiente entre los nodos raíces *t*, que representan las entidades.

1.7.1 Funciones usadas por los diferentes algoritmos

Dados un nodo *v* y un conjunto *S*. La función *first (S)* devuelve el primer nodo $v' \in S$ tal que $v' \prec v_i$ para todo $v_i \in S$. De manera similar, la función *last (S)* devuelve el último nodo $v' \in S$

tal que $v_i \preccurlyeq v'$ para todo $v_i \in S$. Ambas funciones retornan *null* si alguno de los argumentos de entrada es vacío. La función *out (v, S)* devuelve el primer nodo $v' \in S$ tal que $v \prec v'$ no es un descendiente de *v* ni igual a *v*, *out (v, S)= first* $(\{v' \in S \mid v \prec v', v \not\preccurlyeq_a v'\})$. La función devuelve *null* si no existe ningún nodo que cumpla esto o si *v* es *null*.

La función *next (v, S)* devuelve el primer nodo en *S* que sucede a *v* si existe; en otro caso devuelve *null*. La función *pred (v, S)* retorna el predecesor de *v* en *S*, es decir, el último nodo en *S* que precede a *v* si existe, en otro caso devuelve *null*.

La función *closest (v, S)* procesa el nodo de *S* más cercano a *v* de la siguiente manera:

$$closest(v,S) = \begin{cases} pred(v,S) \; si \; lca\big(v, next(v,S)\big) \prec_a lca\big(v, pred(v,S)\big) \\ next\ (v,S) \qquad\qquad\qquad\qquad\qquad\qquad en\ otros\ casos \end{cases} \tag{1.4}$$

Es decir, *closest (v, S)* devuelve *null* si *pred (v, S)* y *next (v, S)* son *null*. La función *lca (S)* busca quien es para *S* el nodo que representa el padre común más cercano y retorna *null* si alguno de sus argumentos es *null*.

Para ver esto más claro pondremos un ejemplo. Dada la siguiente figura.

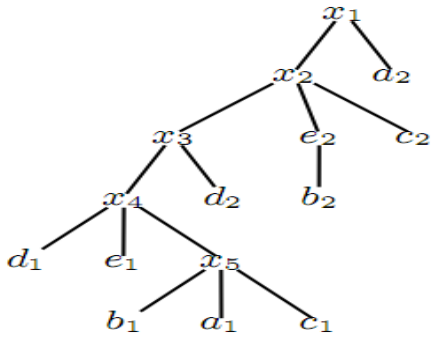


Figura 4. Ejemplo de árbol que representa un XML.

De acuerdo a este árbol y al conjunto de palabras clave $K = \{a, b, c\}$. Una respuesta a esta consulta podría ser $\{a_1, b_1, c_1\}$, pero no lo serían ni $\{a_1, c_1\}$ ni $\{x_1, b_2, c_1\}$. También puede verse que $e_1 \prec b_2$, pero que $e_1 \not\prec x_4$. Además $\{a_1, b_1, c_1\} \prec \{a_2, b_2, c_2\}$. Ahora, si $S=\{d_1, c_1,$

$d_2\}$, entonces el *first* $(S)=d_1$; el *last* $(S)=d_2$; *out* $(x_4, S)=d_2$; *next* $(x_4, S)=c_1$; *next* $(c_1, S)=d_2$; *next* $(e_2, S)=$ *null*; *pred* $(a_1, S)=d_1$; *pred* $(x_4, S)=$ *null*; *closest* $(b_1, S)=c_1$; *closest* $(a_2, S)=d_2$; y el *lca* $(S)=x_3$.

1.7.2 Algoritmos que utilizan la semántica SLCA

Existen diferentes algoritmos que utilizan la semántica Menor Ancestro Común más Cercano (SLCA) para realizar la búsqueda por palabras clave en documentos XML, pero en esta tesis se describirán los tres que mejores resultados muestran según la literatura.

El algoritmo ***Stack*** (Lin Guo 2003) no es más que una variante del algoritmo de mezcla ordenada que utiliza una pila para procesar todos los SLCAs. La idea general del mismo es el uso de una pila para simular el recorrido postorden de un árbol XML virtual formado por la unión de los caminos de la raíz a cada nodo $S_1, \ldots, S_l$ mientras que los nodos son leídos en preorden. Cuando una entrada está en el tope de la pila, lo cual significa que todos sus descendientes en $S_1, \ldots, S_l$ han sido visitados, se sabe si la palabra clave aparece o no en el subárbol. Este algoritmo hace una lista de todas las palabras clave y procesa el prefijo común más largo del nodo que mejor ID tenga.

El pseudocódigo de este algoritmo se muestra en la figura 5. Comienza creando una pila vacía y se tiene una lista de ID de los nodos que no han sido visitados, se lee el próximo nodo con el ID más pequeño y se realizan las operaciones necesarias. En esencia, el orden con que son leídos estos nodos es equivalente a un preorden de un árbol XML ignorando los nodos irrelevantes.

Este algoritmo da como resultado una lista de SLCA por ejemplo, para *slca* $(S_1, \ldots, S_l)$ en un tiempo $O(d \sum_{i=1}^{l} |S_i|)$, o también puede ser visto esto como $O(ld|S|)$*(Yu Xu 2005)*. Podemos ver aquí que no se tiene en cuenta para determinar la complejidad del proceso de mezcla que se desarrolla en el algoritmo, si se tuviera esto en cuenta tendríamos entonces una complejidad de $O(d * lnl * \sum_{i=1}^{l} |S_i|)$.

El algoritmo *Stack* trata todos los conjuntos de entrada $S_1, \ldots, S_l$ como si fueran iguales pero algunas veces la cantidad de elementos de estas listas varía drásticamente. Por esto se crea el algoritmo *Indexed Lookup Eager (Yu Xu 2005)* que trata el caso en que $|S_1|$ es muy diferente del $|S|$.

Propiedad 3. El $slca(\{v\}, S) = lca\big(v, closest(v, S)\big)$ y el $slca(\{v\}, S_2, \ldots, S_l) = slca(slca(\{v\}, S_2, \ldots, S_{l-1}), S_l) = lca\big(v, closest(v, S_2), \ldots, closest(v, S_l)\big)$ para $L>2$.

La complejidad temporal de este algoritmo es $O(|S_l| \sum_{i=2}^{l} d * log|S_i| + |S_l| d * log\,|S_l|)$ o $O(|S_l| ld * log|S|)$.

En la línea 3 del pseudocódigo que se muestra en la figura 7 del algoritmo puede verse que p representa el tamaño del búfer, que puede ser cualquier valor desde 1 hasta $|S_1|$; mientras más pequeña es p, más rápido el algoritmo produce el primer SLCA.

Algorithm 31 StackAlgorithm $(S_1, \cdots, S_l)$

Input: l lists of *Dewey IDs*, S_i is the list of *Dewey IDs* of the nodes containing keyword k_i.
Output: All the SLCAs

```
1:  stack ← ∅
2:  while has not reached the end of all Dewey lists do
3:    v ← getSmallestNode()
4:    p ← lca(stack, v)
5:    while stack.size > p do
6:      en ← stack.pop()
7:      if en.keyword[i] = true, ∀i(1 ≤ i ≤ l) then
8:        output en as a SLCA
9:        mark all the entries in stack so that it can never be SLCA node
10:     else
11:       ∀i(1 ≤ i ≤ l) : stack.top().keyword[i] ← true, if en.keyword[i] = true
12:   ∀i(p < i ≤ v.length) : stack.push(v[i], [])
13:   stack.top().keyword[i] ← true, where v ∈ S_i
14: check entries of the stack and return any SLCA node if exists
```

Figura 5. Pseudocódigo del algoritmo Stack.

Cuando la frecuencia con que aparecen las palabras clave en el conjunto de listas de entrada del algoritmo no tiene diferencias significativas, el costo total de encontrar coincidencias usando búsqueda binaria puede exceder el costo total de encontrar las coincidencias mediante un escaneo de las listas de palabras clave.

El algoritmo ***Scan Eager*** cuyo pseudocódigo se muestra en la figura 6 modifica la línea 15 del algoritmo *Indexed Lookup Eager* utilizando un escaneo lineal para encontrar *lm ()* y *rm ()*. Es decir, se aprovecha de que los accesos a cualquier lista de palabras clave son estrictamente en orden creciente en al algoritmo *Indexed Lookup Eager*.

Scan Eager mantiene un cursor para cada lista de palabras clave y el cursor avanza de S_2 hasta encontrar el nodo que está más cerca de *v* ya sea por la derecha o por la izquierda.

Para asegurar el buen funcionamiento de este algoritmo nótese que *p* tiene que ser no más pequeño que $|S_1|$. La complejidad temporal de *Scan Eager* es un $O(ld|S_1| + d\sum_{i=2}^{l}|S_i|)$, o $O(ld|S|)$*(Yu Xu 2005)*. Como puede verse la complejidad temporal de *Scan Eager* es la misma que la del algoritmo *Indexed Lookup Eager*, pero tiene dos ventajas:

1- *Scan Eager* comienza desde la lista de palabras clave más pequeña y no tiene que escanear hasta el final cada lista de palabras clave, por lo que termina mucho más rápido.

2- El número de operaciones de *lca* de *Scan Eager* es un $O(L|S_1|)$, que es usualmente mucho menos que el de el algoritmo *Stack*, donde el número de operaciones *lca* representan un $O(\sum_{i=1}^{l}|S_i|)$.

Para concluir con algoritmos que realizan la búsqueda basados en la semántica SLCA se describe el algoritmo *Multiway-SLCA* (Sun, Chan et al. 2007) que mejora la actuación del algoritmo *Indexed Lookup Eager* pero mantiene la misma complejidad temporal para el peor caso (Yu Xu 2005).

El funcionamiento de este algoritmo lo ilustramos con el siguiente ejemplo, basándonos en la figura 8.

Algorithm 33 ScanEager $(S_1, \cdots, S_l)$

Input: l lists of *Dewey IDs*, S_i is the list of *Dewey IDs* of the nodes containing keyword k_i.
Output: All the SLCAs

```
1: u ← root with Dewey ID 0
2: for each node v1 ∈ S1 in increasing Dewey ID order do
3:    moving cursors in each list Si to closest(v1, Si), for 1 ≤ i ≤ l
4:    v ← lca(v1, closest(v1, S2), ··· , closest(v1, Sl))
5:    if pre(u) < pre(v) then
6:       if u ≠ v then
7:          output u as a SLCA
8:       u ← v
9: output u as a SLCA
```

Figura 6. Pseudocódigo del algoritmo Scan Eager.

Algorithm 32 IndexedLookupEager $(S_1, \cdots, S_l)$

Input: l lists of *Dewey IDs*, S_i is the list of *Dewey IDs* of the nodes containing keyword k_i.
Output: All the SLCAs

```
1: v ← ⊥
2: while there are more nodes in S1 do
3:    Read p nodes of S1 into buffer B
4:    for i ← 2 to l do
5:       B ← getSLCA(B, Si)
6:    removeFirstNode(B), if v ≠ ⊥ and getFirstNode(B) ⪯ v
7:    output v as a SLCA, if v ≠ ⊥, B ≠ ∅ and v ≠ getFirstNode(B)
8:    if B ≠ ∅ then
9:       v ← removeLastNode(B)
10:      output B as SLCA nodes
11: output v as a SLCA

12: Procedure getSLCA(S1, S2)
13: Result ← ∅; u ← root with Dewey ID 0
14: for each node v ∈ S1 in increasing Dewey ID order do
15:    x ← lca(v, closest(v, S2))
16:    if pre(u) < pre(x) then
17:       Result ← Result ∪ {u}, if u ≠ x
18:       u ← x
19: return Result ∪ {u}
```

Figura 7. Pseudocódigo del algoritmo Indexed Lookup Eager.

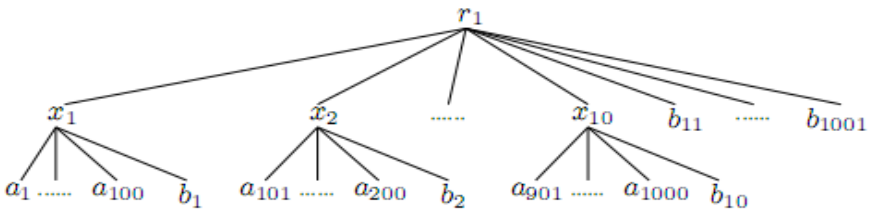


Figura 8. Un ejemplo de un árbol XML para ilustrar el funcionamiento del algoritmo MSLCA.

Ejemplo: Consideremos el conjunto de palabras clave de entrada $\{a, b\}$ y se le aplica el algoritmo al XML de la figura 8. De acuerdo a esto $S_a = \{a_1, \ldots, a_{1000}\}$ y $S_b = \{b_1, \ldots, b_{1001}\}$, *slca* $(S_a, S_b) = \{x_1, \ldots, x_{10}\}$. Como $|S_a| < |S_b|$, el algoritmo *Indexed Lookup Eager* enumeraría cada uno de los nodos de S_a en orden creciente para producir un SLCA potencial. Esto trae consigo realizar 1000 operaciones *slca* para llegar a un resultado de tamaño 10. De esta forma fueron hechas montones de computaciones redundantes, un ejemplo de esto es que el SLCA de a_i y S_b dan el mismo resultado (x_1) para $1 \leq i \leq 100$. Conceptualmente, cada SLCA potencial que se ha computado por *Indexed Lookup Eager* puede ser pensado como siendo conducido por algunos nodos desde S_a (o S_1 más general). El algoritmo *Multiway-SLCA (MSLCA)* selecciona un nodo "*ancla*" entre las listas de palabras clave para a partir de ese nodo conducir las computaciones de SLCAs. En este ejemplo, *MSLCA* primeramente tendrá en consideración el primer nodo de cada lista de palabras clave y de esos selecciona el último que aparezca en el árbol haciendo un recorrido en preorden. Es decir, entre $a_1 \in S_a$ y $b_1 \in S_b$, se escogería b_1 como el nodo "*ancla*". Después, usando b_1 como nodo "*ancla*", seleccionaremos el nodo más cercano (función *closest*) a b_1 de cada uno de las L listas excepto de la lista a la cual pertenece el nodo "*ancla*" y se computará el *lca* de todos los nodos elegidos, por ejemplo el *lca* $(a_{100}, b_1) = x_1$. El próximo nodo "*ancla*" se selecciona de la misma forma pero eliminando todos los nodos que aparezcan antes que el *pred* (b_1) de cada lista de palabras clave. Entonces se selecciona b_2, y el *slca* $(b_2, S_a) = x_2$. Es evidente que el algoritmo MSLCA permite saltar muchas computaciones innecesarias.

De este algoritmo existen dos variantes, *Basic Multiway-SLCA (BMS)* e *Incremental Multiway-SLCA (IMS)* propuestas por (Sun, Chan et al. 2007), el pseudocódigo del segundo se muestra en la figura 9. El algoritmo *BMS* implementa la idea general que se ha visto hasta ahora. IMS introduce además una optimización con el objetivo de reducir el número de computaciones de *lca* que realiza el algoritmo *BMS. BMS* necesita recuperar los nodos en orden desde un conjunto no ordenado y esto trae consigo un gasto de tiempo extra. Por lo que de estas dos variantes analizaremos la variante optimizada, *IMS*. Este algoritmo computa SLCA iterativamente, en cada iteración se selecciona un nodo "*ancla*" v_m y se almacena al igual que su índice *m*. *IMS* se optimiza básicamente en las líneas 7-8. Nótese que cada llamada a la función *closest* requiere dos computaciones *LCA*. *IMS* reduce el número de computaciones *LCA* a un máximo de *L*, es decir como máximo una por cada lista de palabras clave.

BMS e *IMS* dan como salida todos los nodos SLCA en un tiempo *O* ($|S_1|ldlog|S|$).

Algorithm 34 MULTIWAYSLCA $(S_1, \cdots, S_l)$

Input: l lists of *Dewey IDs*, S_i is the list of *Dewey IDs* of the nodes containing keyword k_i.
Output: All the SLCAs

1: $v_m \leftarrow last(\{first(S_i) \mid 1 \le i \le l\})$, where the index m is also recorded
2: $u \leftarrow root$ with *Dewey ID* 0
3: **while** $v_m \neq \perp$ **do**
4: **if** $m \neq 1$ **then**
5: $v_1 \leftarrow closest(v_m, S_1)$
6: $v_m \leftarrow v_1$, if $v_m < v_1$
7: $v_i \leftarrow closest(v_m, S_i)$, for each $1 \le i \le l, i \neq m$
8: $x \leftarrow lca(first(v_1, \cdots, v_l), last(v_1, \cdots, v_l))$
9: **if** $u \le x$ **then**
10: output u as a SLCA, if $u \not\preceq x$
11: $u \leftarrow x$
12: $v_m \leftarrow last(\{rm(v_m, S_i) \mid 1 \le i \le l, v_i \le v_m\})$
13: **if** $v_m \neq \perp$ **and** $u \not\preceq v_m$ **then**
14: $v_m \leftarrow last(\{v_m\} \cup \{out(u, S_i) \mid 1 \le i \le l, i \neq m\})$
15: output u as a SLCA

Figura 9. Pseudocódigo del algoritmo Incremental Multiway-SLCA (MSLCA).

Capítulo 2.

"Análisis e Implementación del Algoritmo KSMMAS"

Capítulo 2. "Análisis e Implementación del Algoritmo KSMMAS"

La buena calidad de los resultados obtenidos al aplicar AS a instancias pequeñas de un problema no escalaba al incrementarse las dimensiones de las instancias para el problema considerado. En 1996, Stützle y Hoos formulan la propuesta Max-Min Ant System (MMAS) (Hoos 1996, Hoos 1997, Hoos 1997, Hoos 2000) que pretende solucionar algunos aspectos no considerados en AS. La idea fundamental detrás de la propuesta es lograr una mayor explotación de las mejores soluciones encontradas, pero incorporando a su vez mecanismos que aseguren que no se llegue a un estado de estancamiento en etapas tempranas de la búsqueda.

2.1 MAX −MIN Ant System (MMAS)

Como su nombre lo indica la esencia de este algoritmo es la búsqueda por palabras clave en documentos XML basada en el algoritmo AS de la meta-heurística ACO.

Para la aplicación de este algoritmo a nuestro problema se hizo necesario hacer una serie de modificaciones, especialmente en los rastros de las feromonas y debido a que el espacio de búsqueda donde se realiza la explotación es la información del documento XML representada en forma de árbol.

El algoritmo de búsqueda por palabras clave usando Sistema de Hormigas Max-Min (KSMMAS, por sus siglas en inglés, Keyword Search Max-Min Ant System)(Hoos 2000) fue implementado en Java y para trabajar con los documentos XML se utilizó la biblioteca JDOM, la cual facilita el manejo de la información del documento.

Una vez comprendida las bases del algoritmo podemos introducirnos en las modificaciones particulares del mismo:

- Solo la mejor hormiga puede actualizar los rastros de feromona (esta puede ser la hormiga que generó el mejor camino de la iteración actual o la hormiga que generó el mejor camino desde el inicio del algoritmo).
- Limita el rango posible de los valores de rastros de feromona al intervalo [τ_{min}, τ_{max}]. Este mecanismo fue ideado para contrarrestar el efecto que produce permitir que solo la mejor hormiga actualice la feromona.
- Los rastros de feromona son inicializados con el valor del límite superior de rastro de feromona, el cual, junto con un pequeño factor de evaporación de feromona, incrementa la exploración de los caminos al comienzo de la búsqueda.
- Los rastros de feromona son reinicializados cada vez que el sistema conduce a la situación de estancamiento o cuando no ha sido generado ningún camino mejor durante un cierto número de iteraciones.

2.1.1 Actualización de los rastros de feromona

Después de que todas las hormigas han construido una solución, se actualizan los rastros de feromona aplicando evaporación de acuerdo a la ecuación que en este caso es la misma que es utilizada por el Sistema de Hormigas.

$$\tau_i \leftarrow (1 - \rho) * \tau_i \tag{2.1}$$

Donde ρ se conoce como constante de evaporación y es la encargada de reducir los rastros de feromona para evitar el estancamiento de las soluciones y τ_i son los valores de la feromona en el arreglo en que se almacenan, seguida por el depósito de nueva feromona de acuerdo a:

$$\tau_i \leftarrow \tau_i + \Delta\tau_i^{best} \tag{2.2}$$

Donde $\Delta\tau_i^{best}$ es la cantidad de feromona que la mejor hormiga deposita en los arcos que ha visitado, el cual se define como:

$$\Delta\tau_i^{best} \begin{cases} \frac{1}{L_{best}} & \text{si la mejor hormiga visitó la palabra } i \text{ en su camino} \\ 0 & \text{en otro caso} \end{cases} \quad (2.3)$$

Donde L_{best} es la longitud del camino de la mejor hormiga. La hormiga a la que se le permite añadir feromona puede ser la que generó la mejor solución de la iteración actual, en cuyo caso $L_{best} = L_{ib}$ donde L_{ib} es la longitud del mejor camino encontrado en la iteración actual; o la que generó la mejor solución desde el inicio del algoritmo, en cuyo caso $L_{best} = L_{bs}$ donde L_{bs} es la longitud del mejor camino encontrado desde el inicio del algoritmo.

Una vez analizado como es que se actualizan las feromonas en este algoritmo podemos observar en el siguiente pseudocódigo cómo es que se evaporan los rastros:

```
1. Procedimiento evaporar_feromona.
2. Para cada feromona τi hacer
      Ejecutar ecuación (2.1)
3. Fin de Procedimiento.
```

Donde la feromona τ_i pertenece al arreglo encargado de almacenar el rastro de feromonas y la ecuación 2.1 vista anteriormente actualiza cada elemento del arreglo. Esta técnica mantiene la variante de AS para la evaporación de los rastros de feromona, una vez evaporado el rastro de feromonas se observa la actualización de la misma, la cual es más agresiva ya que evapora todos los rastros y aporta solamente en los de la mejor solución.

Actualización del rastro de feromonas:

```
1. Procedimiento actualizar_feromona.
2. Para cada palabra  i del documento hacer
```

```
3. Para cada palabra  j del camino de la hormiga hacer
        Si palabra i es igual a palabra j entonces
        Ejecutar ecuación (2.2)
4. Fin de Procedimiento.
```

Comprender que se verifica si el documento contempla alguna de las palabras clave almacenadas en el camino, en caso de que haya sido visitada la palabra, se actualiza el rastro correspondiente mediante la ecuación 2.2.

Los resultados experimentales demuestran que el mejor rendimiento se obtiene incrementando gradualmente la frecuencia de elegir la mejor global para la actualización de feromona respecto de la mejor de la iteración.

2.1.2 Límites de rastros de feromona

Los valores posibles para los rastros de feromona están limitados al rango $[\tau_{min}, \tau_{max}]$. Por lo tanto, la probabilidad de un estancamiento del algoritmo disminuye al darle a cada conexión existente una probabilidad, aunque bastante pequeña, de ser escogida.

Estos límites explícitos se incorporan para la cantidad de feromona que puede haber en una componente $\tau_{min} \leq \tau_i \leq \tau_{max}$. Después de cada iteración se debe asegurar que el rastro de la feromona quede comprendido en el umbral, por lo cual si $\tau_i > \tau_{max}$ se debe realizar el ajuste $\tau_i = \tau_{max}$. Del mismo modo, si $\tau_i < \tau_{min}$ se debe realizar el ajuste $\tau_i = \tau_{min}$. Si adicionalmente se imponen las condiciones naturales $\tau_{min} > 0$ y $\eta_i < \infty$ la probabilidad de elegir un componente específico siempre es > 0.

Podemos observar el tratamiento de las restricciones antes mencionadas en el *procedimiento regular_feromona,* el cual ajusta los límites de la feromona y así los mantiene en el rango especificado. Este procedimiento se muestra en el siguiente pseudocódigo.

```
1. Procedimiento regular_feromona.
```

```
2. Para cada feromona  τi hacer
    3. Si feromona τi > Max_feromona entonces
        feromona  τi = Max_feromona
    sino
    4. Si  feromona τi  < Min_feromona entonces
        feromona  τi = Min_feromona
1. Fin de Procedimiento.
```

Las feromonas son restringidas por los límites ***Max_feromona*** *y* ***Min_feromona*** que son calculados de acuerdo a diferentes parámetros y requisitos que se verán a continuación.

- **Estimación de τ_{max}:**

El valor máximo se calcula adaptando el mayor valor que teóricamente podría tener:

$$\tau_{max}^{teo} = \frac{1}{\rho * Costo(S_{opt})} \tag{2.4}$$

En la práctica, como evidentemente no se dispone de $Costo(S_{opt})$, se utiliza el valor correspondiente a la mejor solución encontrada hasta el momento $Costo(S_{bs})$. Por lo tanto, para cada iteración en la que se obtenga una mejor solución se debe realizar la actualización de τ_{max} .

- **Estimación de τ_{min}:**

La estimación de τ_{min} es bastante más compleja. A continuación se presenta un breve resumen de los conceptos esenciales para lograr la comprensión de la ecuación resultante (2.5).

Se define la convergencia de **KSMMAS** como la distribución de feromona en la cual para cada punto de selección de componentes, una de las posibles componentes tiene τ_{max} y el resto tiene τ_{min} como rastro de feromona. Cuando se ha producido la convergencia, la mejor solución hasta el momento se debe construir con probabilidad $\rho_{best} > 0$. Una hormiga construirá la mejor solución hasta el momento, si en cada selección de componente realiza la elección correcta, es decir si selecciona la componente que tiene asociado como rastro de feromona τ_{max}. Adicionalmente, la hormiga debe realizar *n* decisiones correctas, es decir una por cada componente que selecciona. Finalmente se asume que las opciones entre las que debe elegir en cada paso es constante (*avg*) y que la probabilidad de tomar la decisión correcta en cada paso es constante. Bajo estos supuestos se determina la siguiente estimación:

$$\tau_{min} = \frac{\tau_{max} * \left(1 - \sqrt[n]{\rho_{best}}\right)}{(avg - 1) * \sqrt[n]{\rho_{best}}} \quad (2.5)$$

Mediante estas fórmulas se calculan los límites de la feromona, que luego son utilizados para restringirlas en el intervalo que se vaya generando a medida que se ejecute el algoritmo.

2.1.3 Inicialización y Reinicialización de rastros de feromona

Los rastros son inicializados con el máximo valor posible para todas las componentes (τ_{max}). Antes de la primera iteración no se conoce una solución por lo cual se utiliza un valor alto, para que tras la primera iteración todas las componentes tengan asociado el valor τ_{max}.

Se utilizan tasas de evaporación lenta, logrando aumentar la exploración que se realiza en etapas tempranas.

Esta inicialización de los rastros se pone presente mediante el *procedimiento inicializar_matrix_feromona* en el cual, todo el arreglo de feromonas se inicializa con el

valor máximo del límite (τ_{max}) en que están comprendidas las mismas, en lugar de un valor pequeño τ_0.

```
1. Procedimiento inicializar_matrix_feromona.
     2. Para cada feromona τi hacer
                feromona τi = Max_feromona
3. Fin de Procedimiento.
```

Así, al aplicar la regla de actualización los arcos de las buenas soluciones mantienen valores altos mientras que los de las malas reducen el valor de sus rastros. También da lugar a una mayor exploración al comienzo de la ejecución del algoritmo.

Además, cuando se detecta que la búsqueda está estancada, se aplica una **reinicialización**, volviendo a poner todos los rastros de feromona a τ_{max}. Para entender mejor lo antes planteado centrémonos en el siguiente pseudocódigo:

```
1. Procedimiento reinicializar.
     2. Si costo_solución_local > costo_solución_global entonces
          solución_global = solución_local
          equalPath = false
          contPath = 0
      sino
          equalPath = true
          contPath ++
     3. Si contPath >= reinicio
          Procedimiento inicializar_matrix_feromona
          contPath = 0
```

```
4. Fin de Procedimiento.
```

Una vez que calculamos el costo de la solución óptima hasta el momento en la ejecución del algoritmo, podemos comprobar si se genera alguna con un costo mejor, en caso de que exista la posibilidad asignamos la nueva solución generada a la almacenada para nuevas corridas y ver si es el camino deseado.

Para poder reinicializar el rastro de la feromona en caso de que la búsqueda esté estancada, necesitamos controlar la frecuencia de la iteración, para eso utilizamos la variable booleana *equalPath* la cual nos verifica en que iteración estamos, si toma el valor *false* es que estamos en presencia de una solución mejor, por lo tanto, perdemos la continuidad de las iteraciones y en el caso de que tome *true* es que se mantiene con la misma solución a medida que se itera; así logramos reinicializar si se llega al máximo de iteraciones continuas y no alternadas. Otra de las variables que se utilizará será *contPath,* la cual cuenta las iteraciones en que no se genera una solución mejor a la anterior. Una vez analizadas las variables comprobamos si *contPath* es mayor o igual a una cierta cantidad de iteraciones estáticas previamente incorporadas al algoritmo (variable *reinicio*), si se cumple la condición es que se ha llegado a un estancamiento donde la solución no presenta mejoras por lo que se reinicializa el rastro de feromonas nuevamente con τ_{max} y el contador de las iteraciones (*contPath)* es llevado a cero.

De forma general podemos concluir que la combinación de todo lo anterior establece un buen balance exploración-explotación y reduce la posibilidad de estancamiento de la búsqueda.

2.2 Recuperación de información en documentos XML

La literatura recomienda utilizar como criterio de recuperación de información estructurada, como es la que nos brindan los XML, el siguiente principio:

"Un sistema de recuperación de información deberá siempre recuperar la parte más específica del documento que dé respuesta a la consulta." (Manning 2008)

Este principio es muy difícil de implementar algorítmicamente debido a la ambigüedad que puede apreciarse si dos nodos tienen iguales etiquetas y representan cosas totalmente diferentes. Por ejemplo, un creador y el producto que este creó pueden tener el mismo nombre, por lo tanto serían respuestas aceptadas por el usuario.

Para resolver este problema bastaría con darle prioridades a los nodos, pero ¿por qué darle prioridades a todos los nodos si ese problema solo existiría cuando los nodos que se repitan sean palabras clave?

Para dar solución a este problema en documentos XML existe una fórmula para calcular que tan buena es la solución que se le está brindando al usuario. Esta fórmula se conoce como función de *context resemblance* (C_R) (Manning 2008) y se muestra a continuación:

$$C_R = \begin{cases} \dfrac{1+|C_q|}{1+|C_d|}, & si\ C_q\ coincide\ con\ C_d \\ 0 & en\ otro\ caso \end{cases} \tag{2.6}$$

Donde $|C_q|$ es la cantidad de palabras clave que el usuario está buscando y $|C_d|$ es la cantidad de nodos del documento. El C_R es igual a 1 cuando la cantidad de palabras clave que el usuario está buscando coincide con la cantidad de nodos del documento, esto se considera la respuesta perfecta.

De acuerdo a este principio y a la función expuesta nuestro problema se plantearía de la siguiente manera:

Dado un documento XML *V* (recordar que los XML son vistos como árboles) con elementos (nodos), donde V_i son los distintos fragmentos (subárboles) del documento que

contienen todas las palabras clave, $|C_q|$ es el número de palabras clave de entrada y $|C_{di}|$ la cantidad de elementos (nodos) del fragmento (subárbol) *i*.

Este problema visto matemáticamente quedaría:

$$\forall q \in C_q, \exists d \in C_{di} : q = d \tag{2.7}$$

Después de garantizar que se cumpla esta condición, el problema a resolver no sería más que optimizar la siguiente función:

$$\max \frac{1 + |C_q|}{1 + |C_{di}|} \quad sujeto\ a: |C_{di}| \geq |C_q| \tag{2.8}$$

Con esto se puede garantizar que la solución de este problema va a ser el menor subárbol que contenga todas las palabras clave.

2.3 Algoritmo KSMMAS

El proceso de aplicar el algoritmo KSMMAS a la búsqueda por palabras clave en documentos XML comienza posicionando cada una de las hormigas que conforman la colonia en un elemento escogido aleatoriamente del XML, en este algoritmo se introduce el cálculo de los rangos de feromona (τ_{max} y τ_{min}) donde el arreglo de feromona se inicializa con el valor de τ_{max} en un inicio. Luego se empieza a recorrer el árbol virtual formado del XML en cuestión, para este problema la función heurística está definida por las ecuaciones 2.7 y 2.8, los elementos que van explorando se van almacenando en su lista tabú, en caso de que se repita algún elemento pasan hacia una lista temporal, cada elemento del XML puede ser visitado por la hormiga solo ***cantidad de hijos + 1***, evitándose de esta manera la formación de ciclos. Luego los valores de feromona son evaporados una vez y seguidamente, se actualiza el rastro de feromona, en este caso, actualiza la hormiga que ha obtenido la mejor solución desde el inicio de la ejecución hasta el momento (mejor solución global), tener en cuenta que la cantidad de feromona a depositar tiene que ser mayor o igual

que τ_{min} y menor o igual que τ_{max}. Si pasadas 10 iteraciones se mantiene el mismo costo de la solución global entendemos que se ha llegado a un estancamiento del algoritmo por lo que se reinicia el arreglo de feromona.

Este proceso se repite hasta que cada hormiga haya recorrido todos los nodos del árbol o haya encontrado todas las palabras clave, en otras palabras, ya no tenga más elementos a donde moverse o haya encontrado la solución.

Debemos entender también la heurística de cómo se mueven las hormigas en el árbol a la hora de escoger un nuevo nodo.

En este caso la vecindad alcanzable por el elemento *k* son todos sus hijos y su padre. Para hallar las probabilidades que tiene la hormiga de moverse a cada elemento de su vecindad utilizamos la ecuación 1.1 donde la función heurística a utilizar depende del caso ante el cual nos encontremos.

1. Cuando el nodo no ha sido visitado, es decir, no se encuentra en la lista tabú de la hormiga y es una palabra clave la función heurística a aplicar es:

$$\frac{2 + |C_{qi}|}{2 + |C_{di}|} \tag{2.9}$$

La modificación que encontramos en esta expresión respecto al problema planteado en la expresión 2.8 es que como todavía no hemos insertado el elemento en el tabú de la hormiga simplemente lo sumamos al principio, tanto en el numerador como en el denominador.

2. Cuando el nodo a analizar no ha sido visitado pero no es una palabra clave la función heurística por la cual nos guiamos es la siguiente:

$$\frac{1 + |C_{q}|}{2 + |C_{di}|} \tag{2.10}$$

En este caso solo es necesario agregar uno al denominador debido a que solo vamos a agregar otro elemento al subárbol que se va a devolver, no se va a agregar otra palabra clave.

3. Cuando uno de los elementos de la vecindad ya ha sido visitado no se analiza, es decir, no se le calcula la probabilidad.

Para finalizar este procedimiento se halla por el método de la ruleta a cual elemento va a moverse la hormiga, si este elemento no ha sido visitado se inserta en el tabú de la hormiga y en otro caso se inserta en un temporal. Este temporal tiene almacenado todos los elementos a los que se ha movido esa hormiga, sin importar que se repitan. La hormiga no puede visitar un elemento una cantidad de veces mayor que la longitud de la vecindad de ese elemento. El pseudocódigo de la heurística seguida por las hormigas para este problema es el mismo para el AS y se muestra a continuación.

```
1. Inicializar la vecindad de la hormiga k con sus hijos y su padre
2. Para cada elemento de la vecindad
     Si no está en el tabú y es palabra clave
        Calcular la probabilidad según expresión 1.1 y función
heurística 2.9
     Si no está en el tabú y no es palabra clave
        Calcular la probabilidad según expresión 1.1 y función
heurística 2.10
3. Aplicar la ruleta al arreglo de probabilidades
4. Insertar en la lista temporal el elemento escogido por la ruleta
5. Si no ha sido visitado
     Insértalo en la lista tabú
```

2.4 Parámetros para el KSMMAS

En la Tabla 1 se muestran los parámetros que utiliza esta variante. Si bien pueden parecer una gran cantidad de parámetros, varios son determinados analíticamente y se debe realizar simplemente el cálculo (τ_{min}, τ_{max} y por lo tanto también τ_{init}) y *avg* surge directamente del problema considerado.

Lograr la comprensión de estos valores en este sentido es importante por varias razones, principalmente ya que queremos hacer hincapié en el hecho de que no es posible definir un conjunto óptimo de valores para los parámetros. Si bien en general se acepta que los valores para asignar dependerán del problema y de los casos particulares, no es tan infrecuente observar que algunos de los parámetros se consideran buenos o malos en absoluto, sin ninguna referencia al tiempo de cálculo.

El problema con este enfoque es que la velocidad óptima de la convergencia depende tanto de la instancia y del tiempo de cálculo disponible. Si se requiere la solución muy rápido, se podría preferir una configuración de los parámetros que alcance un óptimo local rápido con respecto a uno que se mantenga explorando el espacio de búsqueda y que probablemente en un tiempo más largo alcanzaría un mejor mínimo local. En este sentido, el análisis de la influencia de cada parámetro en el comportamiento de búsqueda del algoritmo puede ser una manera de enfatizar este elemento.

Es necesario tener en cuenta que algunos de los parámetros recomendados necesitan calibración como cantidad de hormigas (*m*), ρ y las constantes α y β.

En el caso de $\boldsymbol{\rho_{best}}$ es un parámetro introducido por el algoritmo MMAS para que una vez que ha convergido el algoritmo, la mejor solución encontrada se construye con un ρ_{best} de probabilidad, que es significativamente más alto que 0.

2.4.1 Número de hormigas (*m)*

En primer lugar analizamos el efecto del número de hormigas en el comportamiento del algoritmo, dado en este caso por la variable *m.* El valor de *m* afecta al comportamiento del algoritmo esencialmente el nivel de exploración. Teniendo en cuenta la regla de actualización de feromona, el nivel de la exploración se decide principalmente por las soluciones utilizadas para la actualización, esto es debido al hecho de que si se utilizan sólo pocas soluciones después de pocas iteraciones, sólo los arcos pertenecientes a ellos tendrán

una probabilidad significativa de ser elegido. Si la variación de la mejor solución iterada es pequeña, entonces, la convergencia será rápida.

La conclusión de este razonamiento es que cuanto mayor sea el número de hormigas, mayor será la exploración, que a la vez, cuanto mayor sea el número de iteraciones, mayor será la explotación del conocimiento acumulado.

2.4.2 Tasa de evaporación (ρ)

El parámetro ρ está presente en la regla de actualización de la feromona fijando la cantidad de feromona que se evaporará. La relevancia de este parámetro está relacionada con el nivel de exploración del espacio de búsqueda.

Para examinar la influencia de los diferentes valores del parámetro ρ, que determina también la velocidad de convergencia de KSMMAS hacia buenas soluciones, tenemos que tener presente que para un número bajo de caminos construidos los mejores recorridos se encuentran cuando utilizamos los valores más bajos de ρ.

Esto es debido a que para un ρ muy pequeño los rastros de feromona en los arcos en los cuales no se hace el decremento de manera rápida llevan a que la búsqueda se concentre desde temprano alrededor de los mejores caminos encontrados hasta el momento.

En cambio si ρ es alta, muy pocas iteraciones se realizan para alcanzar marcadas diferencias entre los rastros de feromonas en arcos contenidos en caminos de alta calidad y las que no forman parte de los mejores caminos.

Sin embargo, para un número grande de caminos construidos, el uso de un valor de ρ más alto vale la pena, ya que el algoritmo es capaz de explorar más el espacio de búsqueda. Además, es interesante tener en cuenta que con más caminos construidos aumenta el rendimiento general para todos los valores de ρ . Esto se debe principalmente al efecto de los límites más bajos del rastro.

2.4.3 Valores del exponente (α y β)

Los parámetros α y β, representan el exponente del nivel de feromonas y la heurística a medir en la **regla proporcional al azar**, respectivamente.

Su función principal consiste en destacar las diferencias entre los arcos; presentan la restricción de que $\alpha < \beta$, (se considera que $\alpha, \beta \geq 0$) que influye directamente en la función de visibilidad usada para el cálculo de probabilidad, esto se debe a que $\boldsymbol{\alpha}$ condiciona el movimiento de las hormigas y amplifica las diferencias entre los buenos y los malos arcos, y $\boldsymbol{\beta}$ influye en la heurística del algoritmo, heurística que utilizamos en la optimización del algoritmo.

Parámetro	Descripción	Valor recomendado
$\boldsymbol{m}$	Tamaño de la población de Hormigas.	n
$\boldsymbol{\tau_{init}}$	Cantidad inicial de feromona en las componentes de las soluciones.	$\tau_{\max}$
$\boldsymbol{\tau_{min}}$	Cantidad mínima de feromona en las componentes de las soluciones.	$\tau_{min} = \frac{\tau_{max} * \left(1 - \sqrt[n]{\rho_{best}}\right)}{(avg - 1) * \sqrt[n]{\rho_{best}}}$
$\boldsymbol{\tau_{max}}$	Cantidad máxima de feromona en las componentes de las soluciones.	$\tau_{max}^{teo} = \frac{1}{\rho * Costo(S_{opt})}$
$\boldsymbol{\rho_{best}}$	La probabilidad de que una hormiga construya la mejor solución.	$\rho_{\text{best}} \geq 0$

$\boldsymbol{avg}$	El promedio de la cantidad de componentes entre los que debe elegir una hormiga en cada paso para armar la solución completa.	$\frac{n}{Estados\ que\ puede\ elegir}$
α	Influencia relativa de la componente de feromona.	α ≥ 0
β	Influencia relativa del componente heurístico.	β ≥ 0
$\boldsymbol{\rho}$	Tasa de evaporación de la feromona.	$0 \leq \rho \leq 1$

Tabla 1. Parámetros del KSMMAS

2.5Análisis de Complejidad

En el análisis de la complejidad no nos interesa tanto el tiempo que demoran en ejecutarse cada una de ellas, sino la cantidad de operaciones elementales que pudieran aparecer en un determinado algoritmo. Esto significa que asumiremos que las operaciones elementales se ejecutan en una *unidad de tiempo*. En teoría estas operaciones no son elementales ya que el tiempo necesitado para ejecutarlas se incrementa con la longitud de los operandos. En la práctica, sin embargo, podría ser más funcional considerarlas como operaciones elementales teniendo en cuenta que el tamaño de los operandos dentro de las instancias es razonable en relación con el tamaño de las instancias.

En el análisis de este algoritmo podemos observar que cada línea de código se corresponda con un número *variable* de operaciones elementales, de tal manera no modifica la forma interna de ejecución de los algoritmos KSAS pero si mejora el tiempo de ejecución con respecto a la búsqueda exhaustiva, de este modo para analizar la complejidad del algoritmo debemos analizar las variables que definen el tiempo de la exploración (cantidad de hormigas *m*, cantidad de iteraciones *ci* y el costo de cada hormiga, *ch*). Una vez analizado

los aspectos más importantes de la complejidad de un algoritmo, podemos describir el pseudocódigo del KSMMAS y así entender cómo se ponen de manifiesto.

```
1. Fase de inicialización
      Inicializar contador de ciclos NC
      Inicializar total de Hormigas
      Inicializa τmax y τmin
      Inicializar solución global Lglobal = documento.
      Inicializar arreglo de feromona t(i) = tmax
      Inicializar contador de reinicio en 10
      Cargar documento XML
2. Repetir NC veces
       Para cada hormiga
          Elegir elemento inicial
      Repetir hasta llenar tabúk de cada hormiga
       Para cada hormiga:
          Elegir próximo elemento a ser visitado según ecuación (1.1)
          Mover la hormiga al próximo elemento
          Insertar elemento seleccionado en tabúk
      Repetir para las k hormigas
          Calcular calidad de la solución (Lk)
      Evaporar los rastros de feromona según (2.3)
      Si  ( Lk > Lglobal )
            Actualizar Lglobal con Lk
```

Actualizar los valores de feromona (τ) ($\tau_{max} \geq \tau \geq \tau_{min}$) asociados a L_{global} según (2.4)

Si (NC módulo reinicio=0)

Inicializar arreglo de feromona $t_{(i)} = t_{max}$

3. Imprimir subárbol más pequeño L_{global}.

Una vez visto el método podemos analizar la complejidad temporal del algoritmo, teniendo presente que en una sucesión de instrucciones está dada por la regla de la suma $C(S_1;S_2;\ldots;S_n)=C(S_1)+C(S_2)+...+C(S_n)$ además saber que las instrucciones de comparación, operaciones aritméticas y asignación se consideran que son constantes, un O(1), notación utilizada para hacer referencia al orden de la operación elemental (línea de código),y que la complejidad de un ciclo está dada por el total de veces que se evalúa la condición por la complejidad de evaluar la condición (generalmente un O(1)), más el producto del total de veces que se ejecuta el ciclo por la complejidad del cuerpo del ciclo; esto es porque una sentencia dentro de un ciclo se ejecuta tantas veces como se entre al ciclo, es por esto que cada sentencia dentro del ciclo, va tener una complejidad que es igual al producto de la sentencia por la cantidad de veces que se ejecuta.

La función que describe dicho comportamiento es:

$$f_{MMAS}(m, ci, ch) = f_{MMAS}(m) * f_{MMAS}(ci) * f_{MMAS}(ch) \quad (2.11)$$

donde $f_{MMAS}(m)$ está condicionada por la cantidad de hormigas con que cuenta el problema, en este caso *m*, $f_{MMAS}(ci)$ va a depender del número de iteraciones que el algoritmo ejecutará antes que este termine, para esto tomaremos el parámetro como *ci*, y la otra función que se evaluará será $f_{MMAS}(ch)$, donde dependerá de la complejidad del método *wayAs()*, complejidad que analizando las operaciones elementales (asignaciones, operaciones aritméticas, instrucciones condicionales, ciclos y hasta procedimientos que

llaman a otros procedimientos) que describen cada sentencia podemos conformar un polinomio que describirá la misma:

Para comprender la complejidad del polinomio, debemos analizar las sentencias que lo conforman, por ejemplo:

1. Las operaciones elementales de asignación, operaciones aritméticas, entre otras con complejidad constante $\mathbf{O(1)}$.

2. La condición de búsqueda del *for*, que dependerá de la cantidad de hijos que tenga el estado actual y del padre del mismo, además de la cantidad de hormigas que tenga el problema, este hará un recorrido lineal por cada una de ellas con un complejidad de $\mathbf{O(n)}$.

3. Conocer que la *instrucción condicional* es la suma de las complejidades de evaluar la condición y la complejidad de la instrucción a ejecutar. En el caso de una instrucción *if-then-else*, el costo de la instrucción a ejecutar es la complejidad de la instrucción de mayor costo de las dos posibles.

   - *C(IF-Then) = C(condition) + C(Then)*
   - *C(IF-Then-Else)= C(condition)+ max{C(Then), C(Else)}*

Una vez visto esto y teniendo un *if* condicional anidado al *for* que llama a otros procedimientos, podemos conformar:

$$\begin{aligned} O(cond_isHere) + \ [O(cond_isKW) + \max(O(then), O(else))] \\ = O(n) + \ [O(n) + \max(O(1), O(1))] \\ = O(n) + \ [O(n) + \ O(1)] \\ = O(n) + \ [O(n) + \ O(1)] \\ = \mathbf{O(2n)} \end{aligned} \tag{2.12}$$

4. La condición de búsqueda *for*, con complejidad $\mathbf{O(n)}$.

Una vez analizado cada sentencia del método podemos componer el polinomio:

$$\begin{gathered} O(op_{elem}) + O(for_v.size) * [O(if_{isHere}) + O(if_{isKW})] + \\ + O(sent_for) \\ O(op_{elem}) + O(for_v.size) * [O(if_{isHere}) + O(if_{isKW})] + \\ + O(sent_for) \\ = O(1) + O(n) * O(2n) + O(n) \\ = O(1) + O(2n^2) + O(n) \\ = O(2n^2) + O(n) \end{gathered} \tag{2.13}$$

El cual nos da que el polinomio presenta una complejidad de un $\mathbf{O(n^2)}$.

Luego de analizar las diferentes variables que influyen en la complejidad temporal del algoritmo podemos calcular dicha complejidad de acuerdo a la ecuación 2.11.

$$f_{MMAS}(m, ci, ch) = f_{MMAS}(m) * f_{MMAS}(ci) * f_{MMAS}(ch)$$

$$f_{MMAS}(m, ci, ch) = m * ci * 2n^2$$

Este comportamiento está acotado superiormente por la expresión:

$$O(m * ci * 2n^2) \tag{2.14}$$

Debido a esto el algoritmo presenta una complejidad temporal con un orden de $\boldsymbol{O(n^2)}$ que en términos de complejidad presenta la misma que el algoritmo KSAS, pero mejora considerablemente al algoritmo MSLCA antes visto, ya que este presenta una complejidad con un orden de $O(|S_1|ldlog|S|)$, que visto de forma polinomial se podría interpretar como $O(m * n * p * \log c)$, que en el peor de los casos se comporta como $O(n^3 * \log c)$, acotado superiormente con un $O(n^3)$.

Capítulo 3.

"Validación del algoritmo KSMMAS"

Capítulo 3. "Validación del algoritmo KSMMAS"

Un objetivo común para un proyecto de investigación es investigar el impacto de sus resultados, por eso el rápido y sostenido incremento en el poder de cálculo de la computación desde la segunda mitad del siglo XX ha tenido un sustancial impacto en la práctica de la ciencia estadística. Debido a esto una vez analizado los algoritmos para la búsqueda de palabras clave debemos llegar a una conclusión definitiva en que si los resultados obtenidos satisfacen nuestros objetivos planteados. En este capítulo se presentan los resultados obtenidos en las pruebas realizadas a los algoritmos propuestos (KSAS y KSMMAS), en cuanto a tiempo de ejecución, el cual el tiempo de KSMMAS debe presentar mejoras significativas en relación al tiempo de ejecución del KSAS y en la calidad de la solución, la cual queremos que se mantenga con los estándares esperados, y la comparación de estos resultados. También se indica la configuración de los experimentos a realizar y se detallan los resultados de cada una de las ejecuciones del algoritmo. Finalmente se realiza un análisis de los mismos.

3.1 Marco experimental estadístico

No existe un procedimiento establecido y aceptado para comparar algoritmos sobre múltiples conjuntos de datos. Un motivo importante es el comportamiento no determinístico de éstos, por lo que la diferencia detectada entre los resultados de dos algoritmos podría deberse a factores aleatorios, y no a una mejora real (Demsar 2006, García 2008).

Numerosas son las posibles técnicas a aplicar para determinar si un algoritmo difiere de forma significativa respecto a otro, para esto los investigadores pueden aplicar distintas técnicas estadísticas. Principalmente existen dos grupos:

- Las **pruebas paramétricas** (García, Fernández et al. 2009, Luengo, García et al. 2009) son las más empleadas. Se caracterizan en que utilizan por cada algoritmo y

función, el error medio alcanzado de un conjunto de ejecuciones y lo utilizan para identificar si la diferencia entre dos algoritmos es estadísticamente significativa. Un ejemplo de estas pruebas son la prueba de t-Student (para comparar dos algoritmos) y ANOVA (para comparar múltiples algoritmos).

- Las **pruebas no-paramétricas** (Sheskin 2006) utilizan una representación ordinal de los valores basada en el orden de los algoritmos para cada uno de los problemas (no presuponen una distribución de probabilidad para los datos, por ello se conocen también como de distribución libre (distribution free)). Un ejemplo de estas pruebas son, la prueba de Wilcoxon (Wilcoxon 1945)(para comparar los resultados de dos algoritmos), la prueba de Friedman, (análisis de varianza de segunda vía para variables de carácter no paramétrico) y otros como las pruebas de Bonferroni-Dunn o de Holm (Holm 1979).

Para aplicar las pruebas paramétricas es necesario que los datos tengan una distribución normal y una varianza homogénea. Sin embargo, los resultados obtenidos por la meta-heurística no cumplen con estas características, de ahí que surja la necesidad de aplicar pruebas no paramétricas a nuestros algoritmos. También se emplean estadísticos descriptivos para reflejar el comportamiento de los algoritmos.

Para comparar el comportamiento de los nuevos algoritmos se tienen en cuenta dos factores de comparación.

- **Costo computacional (tiempo de ejecución):** Se ejecutan los dos algoritmos (KSAS y KSMMAS) con los mismos parámetros y se comparan entre ellos.
- **Calidad de la solución**: Se ejecutan los algoritmos sobre un conjunto de 60 documentos XML para demostrar que la calidad de la solución que brinda el algoritmo KSAS no difiere de manera significativa de la que brinda el KSMMAS.

3.1.1 Descripción de las pruebas estadísticas utilizadas

Las pruebas estadísticas no paramétricas son las que, a pesar de basarse en determinadas suposiciones, no parten de la base de que los datos analizados adopten una distribución normal (técnica estadística que no presupone ninguna distribución de probabilidad teórica de la distribución de nuestros datos, donde en la mayor parte de ellas los resultados estadísticos se derivan únicamente a partir de procedimientos de ordenación y recuento, por lo que su base lógica es de fácil comprensión). Las pruebas no paramétricas no requieren asumir normalidad de la población y en su mayoría se basan en el ordenamiento de los datos, por eso la población tiene que ser continua. Son técnicas que no presuponen ningún modelo probabilístico teórico.

Una vez analizadas las características de las pruebas estadísticas no paramétricas podemos abordar las principales pruebas que utilizaremos para el desarrollo del trabajo:

- **Porcentaje de desviación (*gap*):** Es la desviación relativa entre una solución, tomada como referencia, y las propuestas de soluciones analizadas. El *gap* entre un par de soluciones puede ser calculado entonces mediante la expresión 3.1.

$$g(SA) = \frac{Z_{ref} - Z_{sa}}{Z_{ref}} * 100 \qquad (3.1)$$

Donde Z_{ref} es considerado como la solución de referencia (los resultados del KSAS), y Z_{sa} el enfoque de solución analizada (la solución de KSMMAS). Un valor positivo del *gap* significa que el enfoque de soluciones es superior a los resultados considerados como referencia.

Es necesario también analizar si existen diferencias significativas en cuanto a los resultados obtenidos en la calidad de la solución y el costo computacional de estos; para esto utilizamos la prueba de Wilcoxon.

- **Prueba de Ranking de Signos de Wilcoxon**: A la hora de hallar las diferencias entre los resultados de dos algoritmos y comparar la media de dos muestras relacionadas puede emplearse esta prueba. Se utiliza como alternativa a la prueba de t-Student cuando no se puede suponer la normalidad de dichas muestras. Se propone una hipótesis H_0 (H_0: no existen diferencias significativas) la cual se rechaza si la significación de la prueba es menor que 0.05 ($sig < 0.05$), en caso contrario la hipótesis propuesta es aceptada.

Prueba	Núm. de variables	Variables	Objetivo
Wilcoxon	2	Continuas, en escala al menos ordinal	Determinar si la diferencia entre la magnitud de las diferencias positivas entre los valores de las dos variables y la magnitud de las diferencias negativas es estadísticamente significativa.

Tabla 2. Prueba no paramétrica de Wilcoxon.

3.2 Entorno Computacional

Todos los resultados experimentales obtenidos en este epígrafe fueron desarrollados en una P5 Intel a 2.6 GHz de velocidad, 2 GB de RAM con sistema operativo Windows, la implementación de tales algoritmos se realizó en el lenguaje Java, para lo cual se utilizó el NetBeans IDE como entorno de desarrollo, en todos los casos se realizaron 10 corridas de cada algoritmo.

3.3 Resultados experimentales

Para el análisis de los resultados obtenidos por los algoritmos se tomó una muestra de 60 documentos XML de la colección de Wikipedia de INEX 2011 a los que se le aplicaron un

grupo de pruebas estadísticas. Se tomaron estos documentos basados en que existiera diversidad en ellos en cuanto a la cantidad de elementos que contienen.

Estas pruebas se realizaron sobre la base de tres dimensiones diferentes para el arreglo de palabras clave de entrada (2, 3, 10 palabras clave). También se calculó el porcentaje de diferencia (*gap*) en todas las dimensiones para **KSAS y KSMMAS**; en el caso de la diferencia en cuanto a *tiempo de ejecución* es preciso minimizar el costo computacional por lo que necesitamos valores positivos y si estamos buscando el porcentaje de diferencia para la *calidad de la solución* necesitamos en el mejor de los casos maximizar los resultados o en cambio que no demuestren diferencias.

Por último se aplicó la prueba de Wilcoxon a la variante de mejor *gap* contrarrestándola con las restantes alternativas para así ver en que algoritmos existe una diferencia relevante.

Los parámetros tomados para los algoritmos después de una serie de pruebas realizadas son los siguientes:

- Cantidad de ciclos *NC* = 1000
- Cantidad de hormigas *m* = 10
- $\alpha = 1$
- $\beta = 2$
- $\rho_{best} = 0.05$
- Constante de evaporación $\rho = 0.02$
- Se decidió que en caso de estancamiento la feromona se reiniciara cada 10 ciclos (iteraciones).

3.3.1 Dimensión 2

A continuación pasaremos a analizar el comportamiento de los algoritmos en cuanto al costo computacional y la calidad de la solución.

Primeramente analizaremos el costo computacional para la búsqueda de 2 palabras clave ya que la búsqueda de solamente una palabra clave no varía la calidad de la solución. Con cualquiera de los algoritmos siempre esta calidad es la mejor (calidad de la solución = 1) y el costo computacional es despreciable. Es necesario aclarar que las palabras clave a buscar fueron elegidas aleatoriamente.

La tabla 3 muestra en la primera columna el nombre del XML, en la segunda y tercera el costo computacional del algoritmo KSAS y KSMMAS respectivamente y el *gap* asociado respecto a cada XML.

	Tpo. 2 Palabras Clave		**Gap (%) Tpo. Ejecución**
Archivo XML	**KSAS**	**KSMMAS**	**KSMMAS/KSAS**
12	102.4	**51.6**	**49.609**
324	15.4	**14.8**	**3.896**
631	16.4	**11.4**	**30.488**
906	25.8	**24.4**	**5.426**
988	**2.4**	4.2	-75.000
1661	12.6	**5.4**	**57.143**
1950	20.4	**19.4**	**4.902**
2003	11.6	**11.4**	**1.724**
2215	26	**22.8**	**12.308**
2503	14.6	**14.4**	**1.370**
2664	**2.2**	4.6	-109.091
2681	8.6	**3.6**	**58.140**
2694	6.8	**3.4**	**50.000**
2695	13	**13**	**0.000**
3070	12.2	**5.8**	**52.459**
3143	**2.2**	3.2	-45.455
3975	28.6	**27.6**	**3.497**
4361	15.4	**15**	**2.597**
4367	12	**5.4**	**55.00**

4446	5.4	**3.4**	**37.037**
4645	12.6	**5.4**	**57.143**
4667	14.2	**12.6**	**11.268**
4770	20.6	**15.4**	**25.243**
4927	14.2	**13.6**	**4.225**
4963	**1.8**	5.6	-211.111
5147	14.8	**11.6**	**21.622**
5649	9.2	**3.4**	**63.043**
5708	4.8	**3.4**	**29.167**
5734	4.4	**3.2**	**27.273**
675	**2.4**	5.6	-133.333
843	12.8	**5.6**	**56.250**
986	3.6	**3.4**	**5.556**
1190	**2.6**	4	-53.846
1213	**2.6**	3.4	-30.769
1837	13	**13**	**0.000**
2226	9.4	**3.6**	**61.702**
2300	12	**5.2**	**56.667**
2447	25.8	**22.8**	**11.628**
3027	12.2	**5.6**	**54.068**
5262	18.6	**17.8**	**4.301**
5725	321.8	**71.8**	**77.688**
5769	7.8	**3.6**	**53.846**
5899	7.8	**3.2**	**58.974**
5956	7.4	**3.4**	**54.054**
6020	**2.8**	5.4	-92.857
6628	5.8	**3.4**	**41.379**
6630	6.8	**3.2**	**52.941**
6697	11.8	**3.4**	**71.186**
6921	10.4	**3.4**	**67.309**
7010	3.2	**3.2**	**0.000**

7148	4.6	**3.2**	**30.435**
7213	4.2	**3.8**	**9.524**
7214	3.8	**3.4**	**10.526**
7427	**2**	3.6	-80.000
7574	**2.2**	3.2	-45.455
7617	4	**3.4**	**15.000**
8880	**1.8**	3.2	-77.778
9080	4.2	**3.2**	**23.810**
12253	**2.8**	3.2	-14.286
12285	3.6	**3.4**	**5.556**
Average			**8.633**

Tabla 3. Resultados para la dimensión 2.

Para demostrar que las diferencias encontradas son estadísticamente significativas se aplicó la prueba de Wilcoxon a los resultados obtenidos. Los resultados se muestran en la tabla 4.

Variables	R^+	R^-	Sig.
KSMMAS-KSAS	1424.00	287.00	0.000

Tabla 4. Resultados de la prueba de Wilcoxon para la dimensión 2.

Como puede verse existen diferencias significativas entre los algoritmos en cuanto al costo computacional, ya que se rechaza la hipótesis nula de que no existen diferencias significativas debido a que la significación tiene un valor de $0.000 < 0.05$.

En este caso 45 de los 60 XML dan menor el costo de KSMMAS con respecto a KSAS, 12 dan un costo mayor y solamente 3 mantienen el mismo costo.

3.3.2 Dimensión 3

Como puede verse los resultados para una dimensión de dos palabras claves presentan diferencias en cuanto al costo computacional, pero no mejora los tiempos de los 60 XML, por eso es necesario también analizar los resultados del costo computacional de los

algoritmos y el porcentaje de desviación para cuando se quieren buscar tres palabras clave, y así comparar los resultados.

	Tpo. 3 Palabras Clave		Gap (%) Tpo. Ejecución
Archivo XML	**KSAS**	**KSMMAS**	**KSMMAS/KSAS**
12	496,4	**55,6**	**88,799**
324	26,4	**11,6**	**56,061**
631	31,2	**11,4**	**63,462**
906	48	**27,7**	**30,000**
988	5,8	**5,4**	**6,897**
1661	26	**4,4**	**59,231**
1950	35,6	**14,6**	**58,989**
2003	21,8	**10,4**	**52,294**
2215	37	**15,6**	**57,838**
2503	28,6	**17,8**	**37,762**
2664	6,8	**5,6**	**17,647**
2681	16	**4,5**	**47,500**
2694	13,8	**5,8**	**50,725**
2695	16,4	**4**	**51,220**
3070	35,8	**3,6**	**67,598**
3143	7,8	**4,6**	**41,026**
3975	59	**18,6**	**68,475**
4361	19,4	**8,6**	**55,670**
4367	24	**5,8**	**57,500**
4446	11	**7,4**	**9,091**
4645	20,6	**7,8**	**62,136**
4667	25,8	**3,4**	**63,566**
4770	38,4	**13,8**	**64,063**

4927	24,4	**10**	**36,066**
4963	9,6	**4,2**	**56,250**
5147	23,4	**13,8**	**41,026**
5649	16,2	**7,6**	**53,086**
5708	8,4	**5**	**40,476**
5734	7,2	**3,4**	**52,778**
675	4,8	**4,6**	**4,167**
843	25	**8,1**	**36,000**
986	6,4	**5**	**21,875**
1190	**5,4**	5,6	-3,704
1213	8,5	**4,6**	**45,882**
1837	23,8	**15,4**	**35,294**
2226	16,8	**7,2**	**57,143**
2300	24,6	**6**	**55,285**
2447	39	**13,3**	**58,462**
3027	24,6	**15,8**	**35,772**
5262	27,4	**14,4**	**47,445**
5725	719,2	**80**	**88,877**
5769	11,2	**6,6**	**41,071**
5899	11,4	**8,8**	**22,807**
5956	11	**5**	**54,545**
6020	4,5	**4,4**	**2,222**
6628	15,6	**6,7**	**46,154**
6630	16,2	**7,2**	**55,556**
6697	16,5	**7,8**	**52,727**
6921	16	**7,4**	**53,750**
7010	5,6	**4,6**	**17,857**

7148	8,2	**8**	**2,439**
7213	7,2	**6,2**	**13,889**
7214	6,4	**6,2**	**3,125**
7427	4,8	**4,4**	**8,333**
7574	4,8	**4,2**	**12,500**
7617	11,8	**4,6**	**61,017**
8880	3,6	**3,4**	**5,556**
9080	11,4	**7,6**	**33,333**
12253	5	**4,6**	**8,000**
12285	11,8	**4,8**	**59,322**
Average			**41,398**

Tabla 5. Resultados para la dimensión 3.

Luego de aplicada la prueba de Wilcoxon para corroborar si estas diferencias son significativas obtenemos los siguientes resultados.

Variables	R^+	R^-	Sig.
KSMMAS-KSAS	1887.00	4.00	0.000

Tabla 6. Resultados de la prueba de Wilcoxon para la dimensión 3.

En este caso el algoritmo KSMMAS presenta un mejor comportamiento que el KSAS, mejorando sustancialmente el costo computacional para 59 de los 60 XML procesados.

3.3.3 Dimensión 10

Ya analizado para dimensiones de dos y tres palabras clave, como una última muestra analizaremos los resultados que presenta para una búsqueda de 10 palabras clave.

	Tpo. 10 Palabras Clave		Gap (%) Tpo. Ejecución
Archivo XML	**KSAS**	**KSMMAS**	**KSMMAS/KSAS**

12	910,0	**59,7**	**93,443**
324	95,7	**18,7**	**80,488**
631	51,0	**12,7**	**75,163**
906	88,0	**33,5**	**68,561**
988	10,7	**7,3**	**31,250**
1661	1591,0	**4,7**	**99,707**
1950	8271,3	**8225,7**	**0,552**
2003	2221,7	**2048,3**	**7,802**
2215	1733,3	**40,7**	**97,654**
2503	5174,3	**5084,3**	**1,739**
2664	2121,3	**1934,7**	**8,799**
2681	1613,3	**4,7**	**99,711**
2694	49,0	**11,0**	**77,551**
2695	1585,3	**4,0**	**99,748**
3070	1619,3	**4,7**	**99,712**
3143	16,3	**7,3**	**55,102**
3975	3412,0	**60,0**	**98,242**
4361	48,3	**16,0**	**66,897**
4367	59,0	**6,0**	**89,831**
4446	28,0	**8,7**	**69,048**
4645	34,7	**10,0**	**71,154**
4667	1600,7	**4,0**	**99,750**
4770	62,0	**15,0**	**75,806**
4927	51,7	**9,3**	**81,935**
4963	9,3	**4,7**	**50,000**
5147	53,3	**14,0**	**73,750**
5649	34,0	**19,0**	**44,118**

5708	36,7	**13,3**	**63,636**
5734	1497,3	**4,0**	**99,733**
675	14,0	**5,3**	**61,905**
843	1562,7	**8,9**	**99,680**
986	1947,0	**1881,7**	**3,356**
1190	16,0	**6,7**	**58,333**
1213	12,7	**8,0**	**36,842**
1837	42,0	**16,7**	**60,317**
2226	44,7	**11,0**	**75,373**
2300	1605,3	**6,7**	**99,585**
2447	1567,7	**16,3**	**99,149**
3027	4881,3	**4819,3**	**1,270**
5262	527,3	**37,3**	**92,920**
5725	2110,7	**1217,7**	**42,309**
5769	1557,3	**8,7**	**99,443**
5899	76,7	**19,0**	**75,217**
5956	2106,3	**1959,3**	**6,979**
6020	11,7	**7,0**	**40,000**
6628	1508,3	**8,4**	**99,558**
6630	1514,0	**9,3**	**99,384**
6697	1579,3	**9,0**	**99,430**
6921	23,3	**12,0**	**48,571**
7010	20,0	**12,0**	**40,000**
7148	25,7	**10,3**	**59,740**
7213	20,0	**10,0**	**50,000**
7214	2139,7	**1960,0**	**8,397**
7427	43,3	**9,7**	**77,692**

7574	14,7	**8,3**	**43,182**
7617	21,7	**12,3**	**43,077**
8880	12,0	**8,0**	**33,333**
9080	1538,3	**10,3**	**99,328**
12253	14,0	**9,3**	**33,333**
12285	20,0	**8,3**	**58,333**
Average			**63,782**

Tabla 7. Resultados para la dimensión 10.

Aplicando la prueba de Wicoxon a los resultados.

Variables	R^+	R^-	Sig.
KSMMAS-KSAS	1891.00	0.00	0.000

Tabla 8. Resultados de la prueba de Wilcoxon para la dimensión 10.

Es posible observar para esta dimensión que los 60 XML dan un menor costo computacional del algoritmo KSMMAS con respecto a KSAS.

Una vez analizado el costo computacional para estas dimensiones (2, 3,10) llegamos a la conclusión de que existen diferencias significativas en cuanto al tiempo de ejecución, pero debemos analizar también la calidad de la solución para verificar que no presente diferencias en relación con los algoritmos ya estudiados.

En la tabla 9 se muestra un resumen de las pruebas realizadas tanto para la calidad que para el tiempo de ejecución.

Se puede observar que para la calidad de la solución no existen diferencias significativas, por lo que mantiene la misma calidad en la solución de los algoritmos anteriores, en cambio para el costo computacional en las tres dimensiones presenta diferencias significativas,

afirmando así, que el algoritmo KSMMAS mejora en tiempo de ejecución al KSAS manteniendo la misma calidad de la solución.

KSMMAS-KSAS		Pruebas Estadísticas	
Aspectos Comparativos	**Dimensiones**	***Wilcoxon***	***Promedio (Gap)***
Costo Computacional	**2**	0.000	8,633
	3	0.000	41,398
	10	0.000	63,782
Calidad de la Solución	**2**	0.102	No es necesario hallar el *gap,* ya que la calidad se mantiene, no se mejora.
	3	0.180	
	10	0.317	

Tabla 9. Resultados generales comparativos de la prueba de Wilcoxon.

3.4 Interfaz de Usuario

Una vez implementado el algoritmo KSMMAS, se creó una aplicación desktop que le permita al usuario interactuar con el algoritmo de manera más sencilla. Al ejecutar el programa lo primero que se muestra es una ventana de bienvenida a la aplicación (figura 10).

Luego de visualizar la ventana de bienvenida, se carga la aplicación desktop, en la que su área principal está dividida en distintos paneles, cada uno con una función diferente que hacen posible la ejecución del algoritmo, estos paneles se muestran en la figura 11 y la función de estos son:

1. Seleccionar el XML al que se le ejecutará la búsqueda de las palabras clave especificando la dirección en donde se encuentra.

Figura 10. Ventana de bienvenida a la aplicación.

2. Introducir las palabras clave que se quieren encontrar en el documento, se especifica que deben estar divididas por coma para una buena ejecución.
3. Introducir la constante de evaporación que utiliza el algoritmo para evaporar el rastro de feromona.
4. Introducir la cantidad de iteraciones que el algoritmo recorrerá mientras se ejecuta.
5. Introducir las constantes alfa (α) y beta (β) respectivamente, utilizadas en el cálculo de la probabilidad de escoger el nuevo estado al que se moverá la hormiga.
6. El botón *Limpiar Parámetros* se utiliza para eliminar los valores existentes y poder actualizarlos con otros para una nueva corrida.
7. El botón *Correr KSMMAS* se encarga de ejecutar el algoritmo con los parámetros estipulados anteriormente para generar la solución.
8. Mostrar la solución resultante luego de la ejecución (camino de la hormiga (solución), calidad de la solución, costo computacional).

9. El botón S*alvar Resultados* salva los resultados de la corrida en un archivo (txt), una vez salvados se le pueden efectuar todas las operaciones deseadas, así como visualizarlos de una manera más sencilla.

10. El botón *Cerrar KSMMAS* cierra la aplicación.

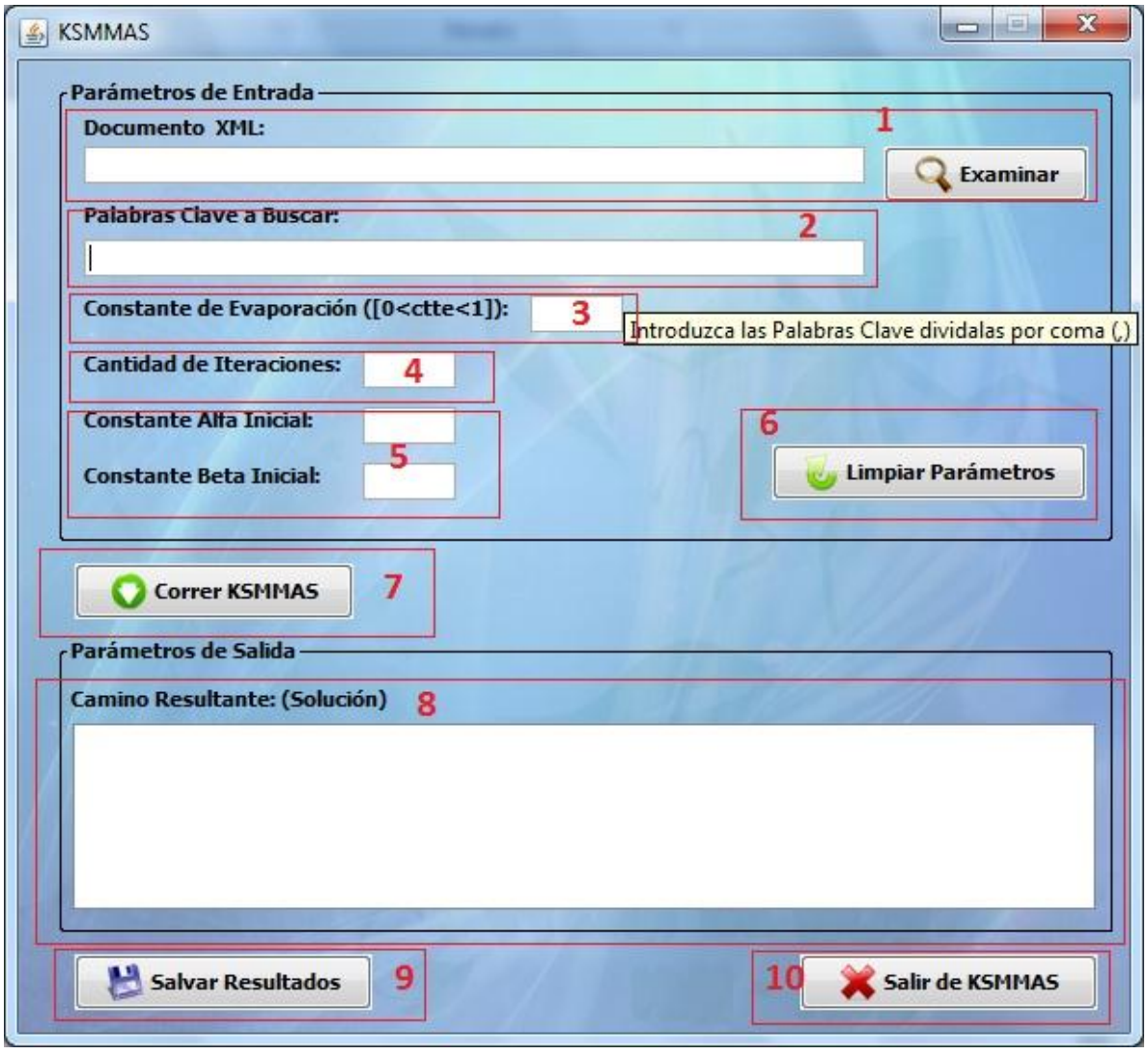


Figura 11. Aplicación desktop para interactuar con el algoritmo.

Para tener un panorama mejor de cómo es que se trabaja con la aplicación, en la figura 12 y 13 se muestra la ejecución del algoritmo, donde, se escoge un documento XML aleatorio,

se le pasan las palabras clave que se quieren buscar, se le pasa la constante de evaporación, la cantidad de iteraciones que realizará y la constante alfa y beta. Ya una vez ejecutado el algoritmo (*Correr KSMMAS*), se muestran los resultados obtenidos.

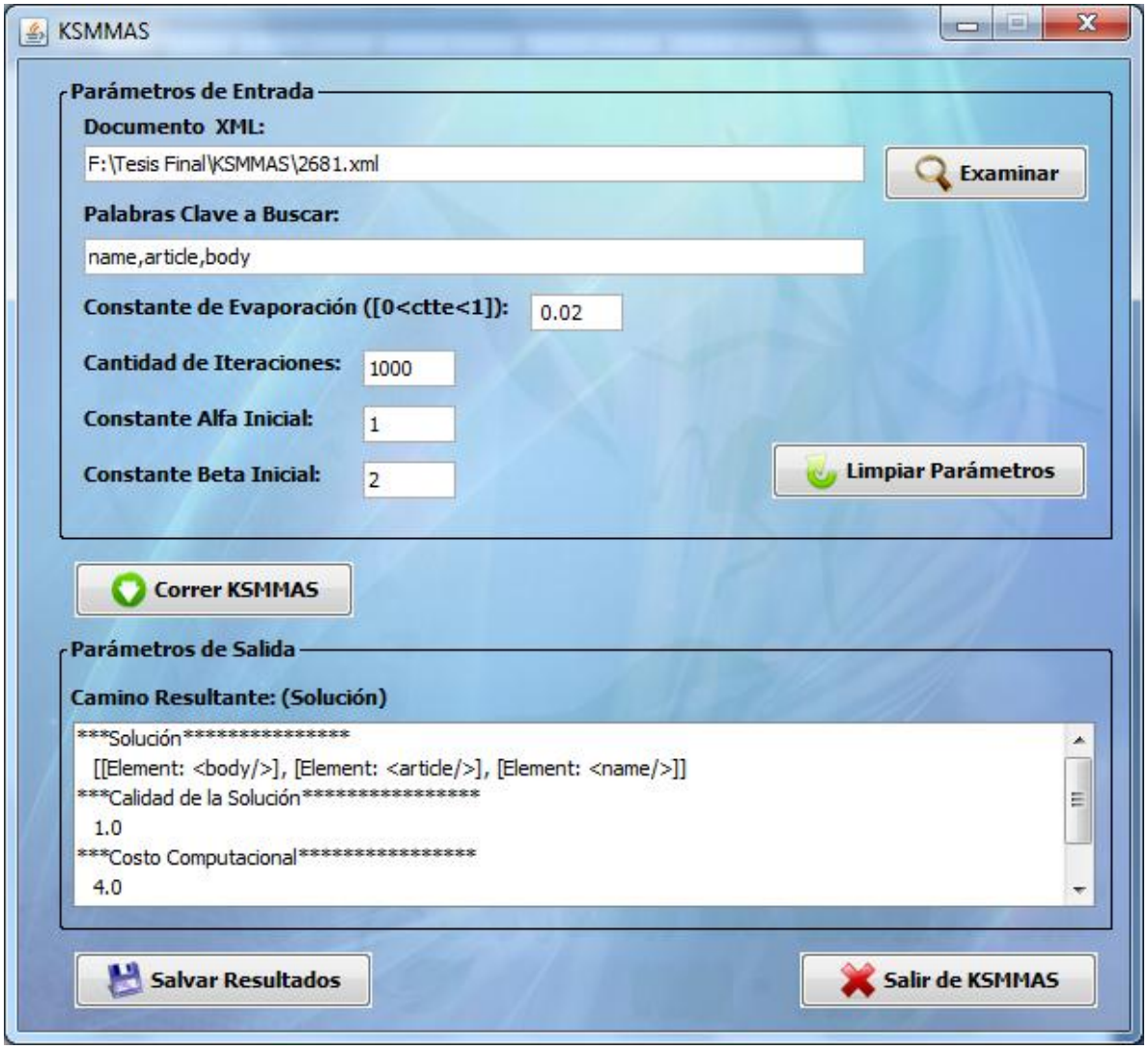


Figura 12. Ejemplo de ejecución del algoritmo desde la aplicación con tres palabras clave.

Ya con esta interfaz el usuario puede establecer una mejor interacción con el algoritmo. Es una manera de recuperar la información de los documentos XML más rápida, pero a la vez más factible ya que presenta mejores condiciones para trabajar.

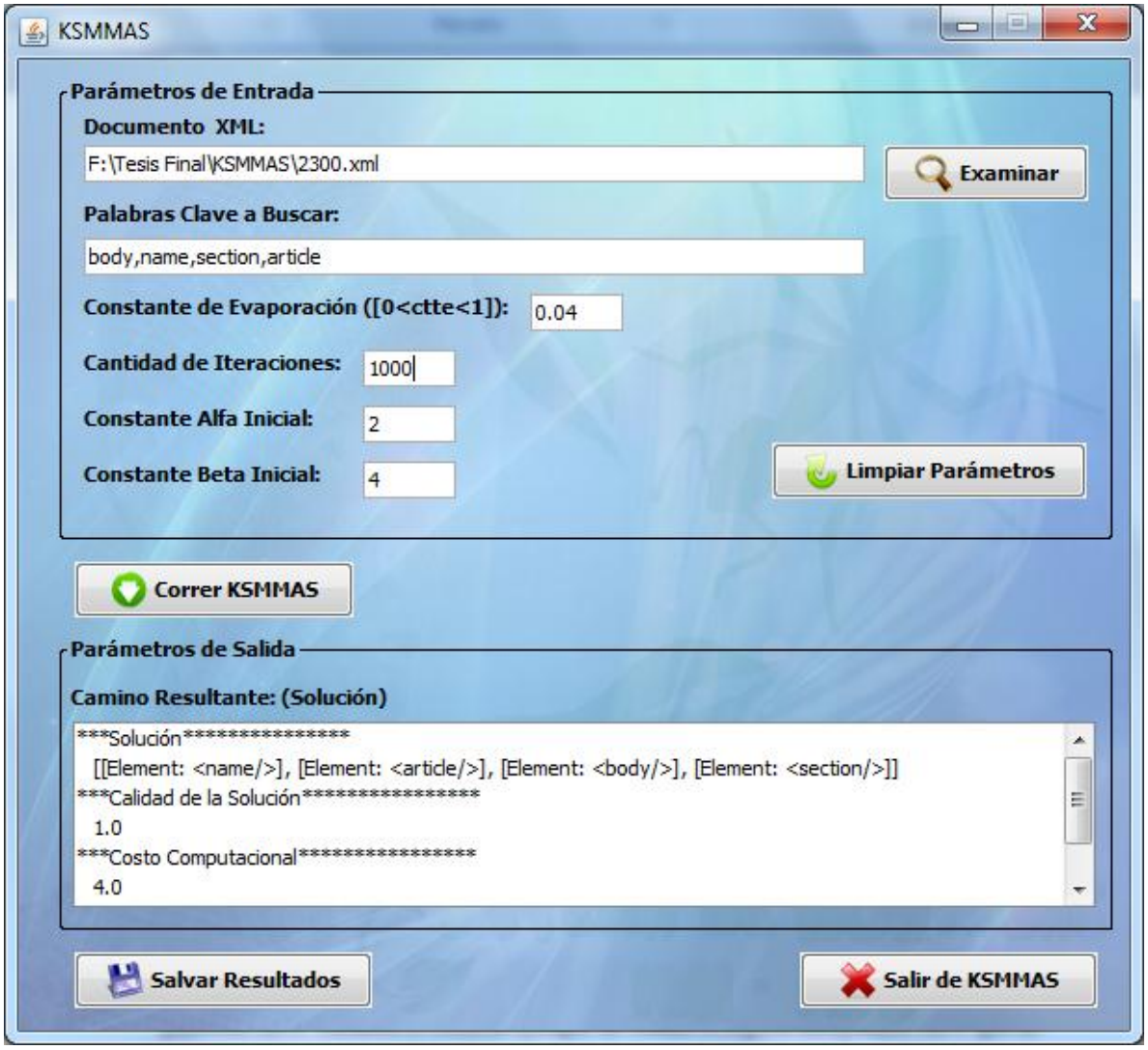


Figura 13. Ejemplo de ejecución del algoritmo desde la aplicación con cuatro palabras clave.

Ya en los anexos se muestran otras ejecuciones del algoritmo variando la cantidad de palabras clave. También se toma un ejemplo de cuando se le pasa una palabra clave que no aparece en el documento XML, y como es tratada esta dificultad.

Conclusiones

Conclusiones

1. Se realizó un estudio de los principales algoritmos que realizan la búsqueda por palabras clave en documentos XML, tomando en cuenta el IMS como algoritmo exhaustivo, pero haciendo énfasis en el algoritmo que utiliza la meta-heurística ACO y que está basado en el AS, siendo este el KSAS.
2. Se implementó el algoritmo KSMMAS, que realiza la búsqueda por palabras clave en documentos XML basado en el algoritmo MMAS, el cual tiene las bases en el AS, pero está sujeto a algunas modificaciones.
3. Se validó el algoritmo en cuanto a calidad de la solución y tiempo de ejecución utilizando la prueba de Wilcoxon como prueba estadística, también se utilizó el porcentaje de desviación para analizar los resultados obtenidos.
4. Se desarrolló un framework que permite visualizar el funcionamiento del algoritmo, haciendo el trabajo con el mismo más fácil y entendible al usuario.

Recomendaciones

Recomendaciones

1. Aplicar este algoritmo de búsqueda por palabras clave al problema del versionamiento de Reglas de Negocio.
2. Aplicar otros algoritmos de la meta-heurística ACO para ver si se obtienen mejores soluciones.
3. Difundir los resultados de esta tesis para futuras investigaciones en el tema.

Referencias bibliográficas

1. B. Bullnheimer, R. F. H., y C. Strauss (1999). "A new rank-based version of the Ant System: A computational study." Central European Journal for Operations Research and Economics: 25-38.
2. Bratley, G. B. y. P. (1996). Fundamentals of Algorithmics. Prentice Hall, Englewood Cliffs.
3. Demsar, J. (2006). "Statistical comparisons of classifiers over multiple data sets." The Journal of Machine Learning Research **7**: 1-30.
4. Dorigo, M. (1992). Optimization, Learning and Natural Algorithms.
5. Dorigo, M. (2001). Ant Algorithms Solve Difficult optimization Problems, Advances in Artificial Lif. Sixth European Conference on Artificial Life. Springer Berlin Heidelberg.**:** 11-12.
6. Dorigo, M. G., L.M (1997). Ant Colony System: A Cooperative learning approach to the traveling salesman problem, Evolutionary Computation. **1:** 53-66.
7. E. Bonabeau, M. D., y G. Theraulaz (1999). Swarm Intelligence: From Natural to Artificial Systems. New York, Oxford University Press.
8. Fogel, Z. M. y. D. B. (2000). How to Solve It: Modern Heuristics. Springer Verlag, Berlín, Alemania.
9. García, S., A. Fernández, J. Luengo and F. Herrera (2009). "A study of statistical techniques and performance measures for genetics-based machine learning: accuracy and interpretability." Soft Computing **13**(10): 959-977.
10. García, S., Herrera, Francisco (2008). "An extension on statistical comparisons of classifiers over multiple data sets for all pairwise comparisons." Journal of Machine Learning Research **9**(2677-2694): 66.
11. Guoliang Li, J. F., Jianyong Wang, Lizhu Zhou (2007). Effective keyword search for valuable lcas over xml documents. Proceedings of the sixteenth ACM conference on Conference on information and knowledge management. Lisbon, Portugal, ACM**:** 31-40.
12. Holm, S. (1979). "A simple sequentially rejective multiple test procedure." Scand Journal Stat **6**: 65-70.
13. Hoos, T. S. a. H. (1996). Improving the Ant System: A Detailed Report on the MAX - MIN Ant System. Technical Report Germany, AIDA**:** 96-12.
14. Hoos, T. S. a. H. (1997). The MAX-MIN Ant System and Local Search for Combinatorial Optimization Problems: Towards Adaptive Tools for Global

Optimization. Proceedings of the second Metaheuristics International Conference (MIC'97), France.

15. Hoos, T. S. a. H. (1997). MAX-MIN Ant System and Local Search for the Traveling Salesman Problem. Proceedings of the fourth International Conference on Evolutionary Computation (ICEC). IEEE**:** 308-313.
16. Hoos, T. S. y. H. H. (2000). MAX-MIN Ant System. Future Generation Computer Systems.
17. Hristidis V., P. Y. a. B., A. (2003). Keyword proximity search on XML graphs Proceedings. 19th International Conference on Data Engineering, IEEE**:** 367-378.
18. J. M. Pasteels, J.-L. D., y S. Goss (1987). Self-organization mechanisms in ant societes.
19. Jianxin Li, C. L., Rui Zhou and A Wei Wang (2010). Suggestion of promising result types for XML keyword search. Proceedings of the 13th International Conference on Extending Database Technology. Lausanne, Switzerland, ACM**:** 561-572.
20. Jonson, M. R. G. y. D. S. (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, San Francisco.
21. Kelly, I. H. O. a. J. P. (1996). Meta-Heuristics: Theory & Applications. Boston, Kluwer Academic Publishers.
22. Kochenberger, F. G. a. G. (2003). Handbook of Metaheuristics, Kluwer Academic Publishers.
23. Lenstra, E. H. L. A. y. J. K. (1997). Local Search in Combinatorial Optimization. Chichester, John Wiley & Sons.
24. Lin Guo, F. S., Chavdar Botev, Jayavel Shanmugasundaram (2003). XRANK: ranked keyword search over XML documents. Proceedings of the 2003 ACM SIGMOD international conference on Management of data. San Diego, California, ACM**:** 16-27.
25. Liu, Z. a. Y. C. (2007). Identifying meaningful return information for xml keyword search. SIGMOD**:** 329-340.
26. Luengo, J., S. García and F. Herrera (2009). "A study on the use of statistical tests for experimentation with neural networks: Analysis of parametric test conditions and non-parametric tests." Expert Systems with Applications **36**(4): 7798-7808.
27. M. Dorigo, V. M., y A. Colorni (1996). The Ant System: Optimization by a colony of cooperating agents. IEEE Transactions on Systems, Man and Cybernetics**:** 29-41.
28. Manning, C. D., P. Raghavan, and H. Schtze (2008). "Introduction to Information Retrieval." Cambridge University Press. 496.

29. O. Cordón, F. H., L. Moreno (1999). Integración de Conceptos de Computación Evolutiva en un Nuevo Modelo de Colonias de Hormigas. VIII Conferencia de la Asociación Española para la Inteligencia Artificial, (Seminario Especializado en Computacion Evolutiva). Murcia, España. **II:** 98-105.

30. Reeves, C. (1995). Modern Heuristic Techniques for Combinatorial Problems. McGraw Hill, London, UK.

31. S. Goss, S. A., J. L. Deneubourg, y J. M. Pasteels (1989). Self-organized shortcuts in the Argentine ant. Naturwissenschaften.

32. S. Voss, S. M., I.H. Osman, y C. Roucairol (1999). Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization. Boston, Kluwer Academic Publishers.

33. Sheskin, D. J. (2006). Handbook of parametric and nonparametric statistical procedures, CRC, Chapman & Hall: London/West Palm Beach.

34. Steiglitz, C. H. P. y. K. (1982). Combinatorial Optimization - Algorithms and Complexity. Prentice Hall, Englewood Cliffs.

35. Sun, C., C.-Y. Chan and A. K. Goenka (2007). Multiway SLCA-based keyword search in XML data. Proceedings of the 16th international conference on World Wide Web. Banff, Alberta, Canada, ACM**:** 1043-1052.

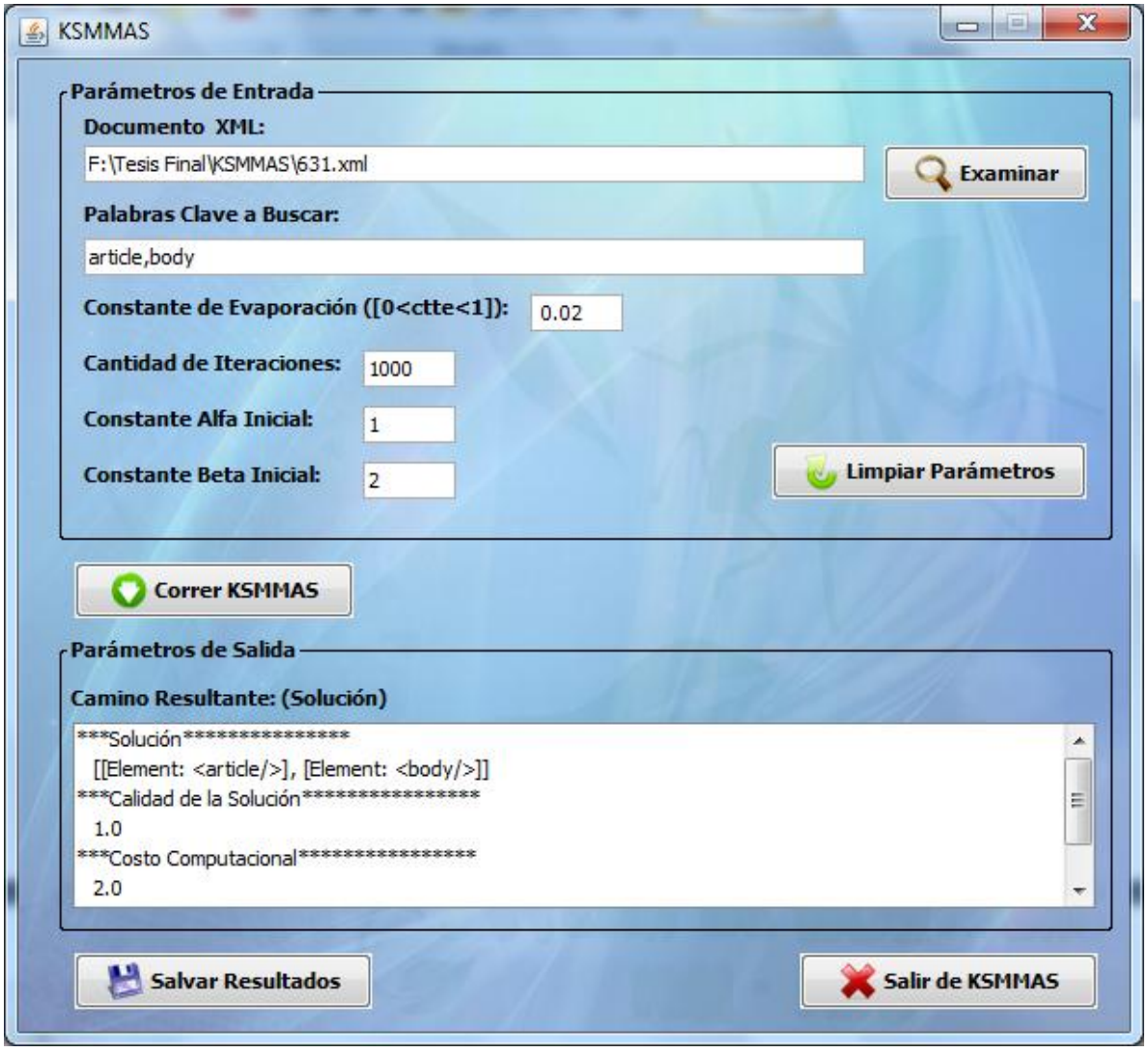
36. Wilcoxon, F. (1945). " Individual comparisons by ranking methods." Biometrics Bulletin **1(6)**: 80-83.

37. Yu Xu, Y. P. (2005). Efficient keyword search for smallest LCAs in XML databases. Proceedings of the 2005 ACM SIGMOD international conference on Management of data. Baltimore, Maryland, ACM**:** 537-538.

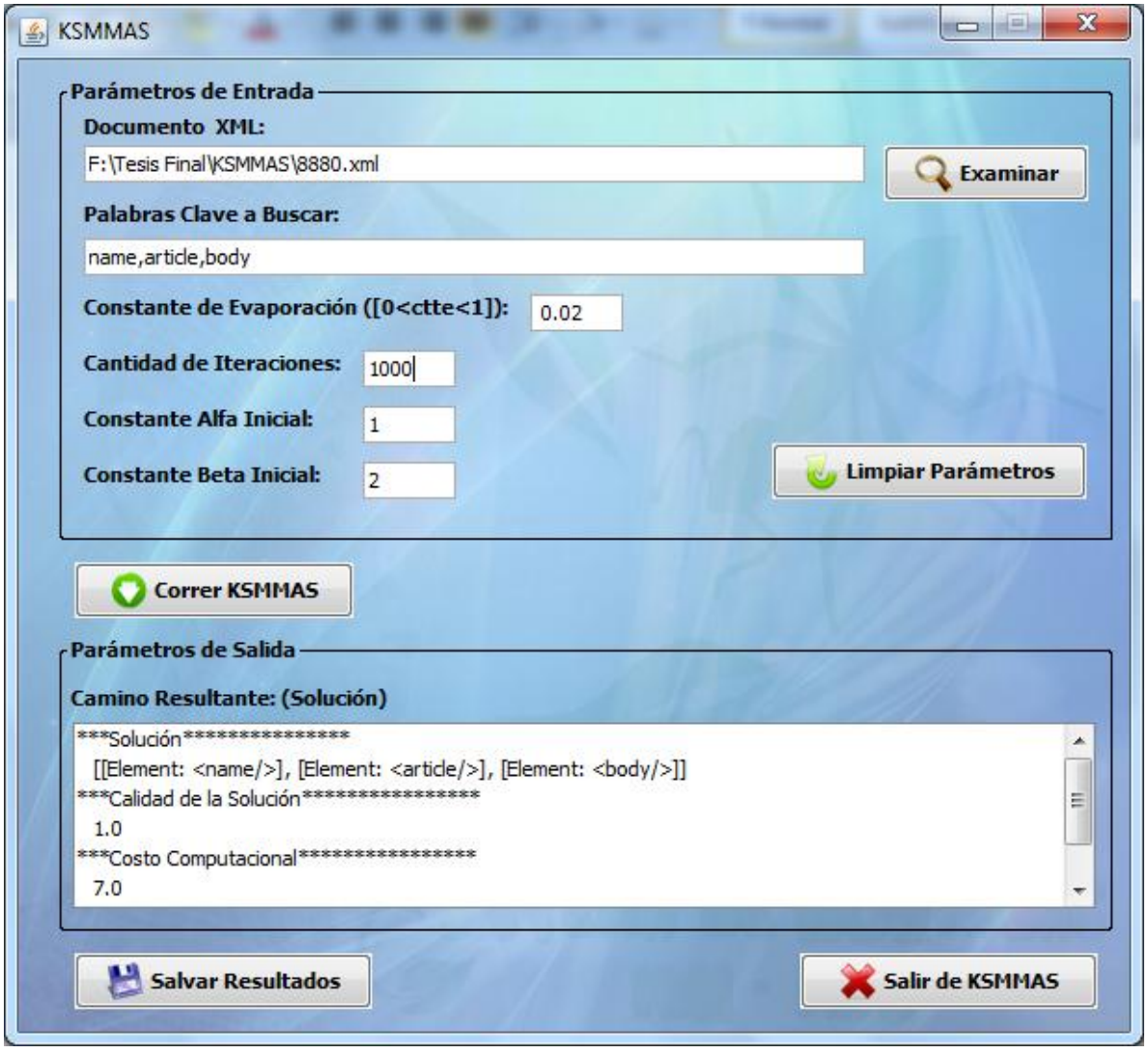
38. Yunyao Li, C. Y., H. V. Jagadish (2004). Schema-free XQuery. Proceedings of the Thirtieth international conference on Very large data bases. Toronto, Canada, VLDB Endowment. **30:** 72-83.

Anexos

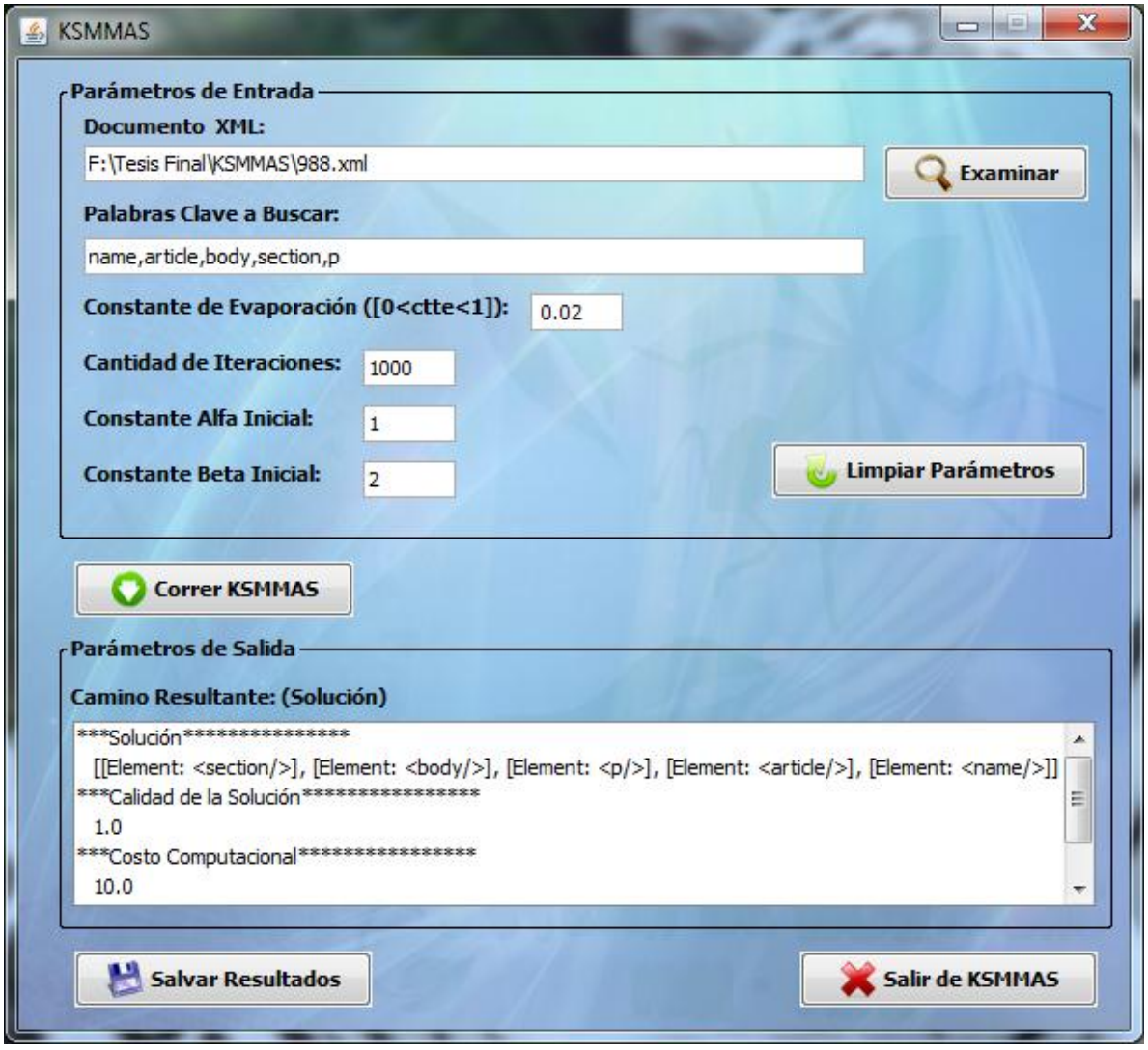
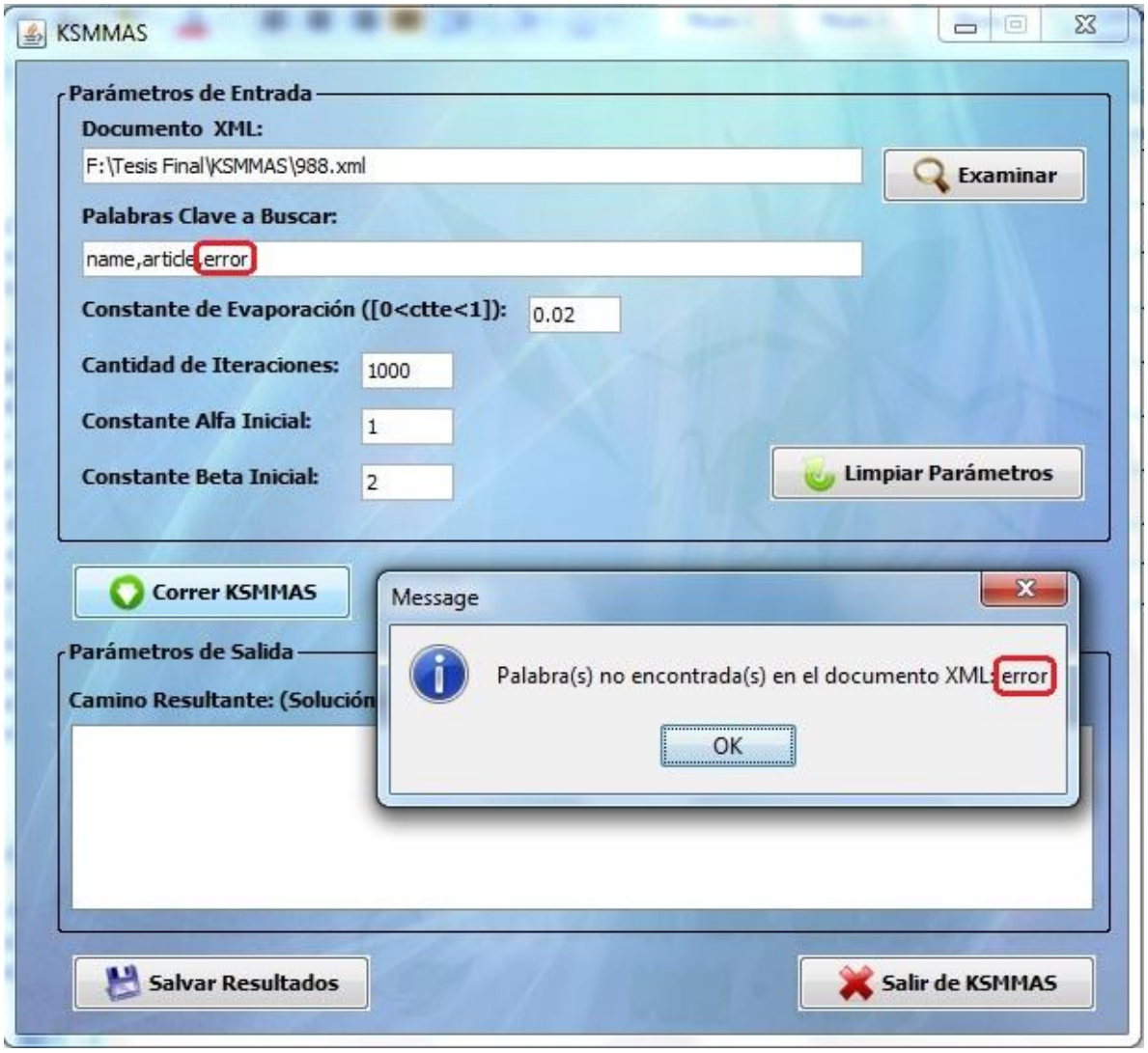
Anexo I: Ejecución del algoritmo con dos palabras clave



Anexo II: Ejecución del algoritmo con tres palabras clave



Anexo III: Ejecución del algoritmo con más de tres palabras clave



Anexo IV: Ejecución del algoritmo con una palabra clave errónea