UNIVERSIDAD NACIONAL MAYOR DE SAN MARCOS

FACULTAD DE INGENIERÍA DE SISTEMAS E INFORMÁTICA

UNIDAD DE POSGRADO

Generación automática de casos de prueba para test de una GUI, usando colonia de hormigas y metaheurística golosa

TESIS

Para optar el Grado Académico de Magíster en Ingeniería de Sistemas e Informática con mención en Ingeniería de Software

AUTOR

José Fernando Rodríguez Valderrama

ASESOR

Glen Rodríguez

Lima – Perú

2013

Dedico este trabajo a Dios, quién supo guiarme por el buen camino, darme las fuerzas para seguir adelante y no desmayar en los problemas.

A mis padres Pedro† y Rosa que han velado cada día por mi bienestar, y a mi familia en general por estar siempre presentes acompañándome para poderme realizar.

A Víctor y Silvia por su contribución ejemplar en la culminación de este trabajo.

A todos ellos se los agradezco desde el fondo de mi alma.

AGRADECIMIENTOS

Al profesor Dr. Glen Rodríguez, por su orientación, paciencia y dedicación para que este trabajo cumpla con los objetivos trazados.

Al profesor Mg. Erick Vicente por su apoyo, consejos y ser parte de este innovador tema de investigación.

A las autoridades responsables del convenio BILIMERI entre Francia, Perú y Colombia, que ha permitido un valioso intercambio académico, cultural y humano entre estos países, y que considero importante en nuestra formación.

A mis amigos y compañeros de estudios con los que compartimos muchas experiencias académicas y personales, y porque en todo momento me incentivaron para que culmine este trabajo. A todas aquellas personas que indirectamente me ayudaron para culminar este trabajo y que muchas veces constituyen un invalorable apoyo.

Antes de culminar quisiera mencionar a dos personas que me proporcionaron confianza, confort y facilidades laborales para culminar con este proyecto que me ha demandado mucho sacrificio, me refiero a Gabriel Vega y Omar Rivas, gerentes y compañeros de la empresa “HIPER S.A.”

Y por encima de todo doy gracias a Dios.

"¿Por qué esta magnífica tecnología científica, que ahorra trabajo y nos hace la vida más fácil, nos aporta tan poca felicidad?
La respuesta es ésta, simplemente porque aún no hemos aprendido a usarla con tino."

Albert Einstein (1879-1955)

Generación Automática de Casos de Prueba para Test de una GUI, usando Colonia de Hormigas y Metaheurística Golosa

RESUMEN

Hoy en día, debido al aumento del tamaño y la complejidad del software, el proceso de pruebas se ha convertido en una tarea vital en el desarrollo de cualquier sistema informático. La calidad de un caso de prueba se mide por los criterios de cobertura, la cual depende de la longitud de los caminos recorridos en el árbol de secuencias de eventos. Encontrar aquellos caminos factibles y que cumplan con los criterios de cobertura es un problema altamente combinatorio. Para este tipo de problemas es bien justificado el uso de algoritmos heurísticos y metaheurísticos.

En esta perspectiva, se presenta una propuesta del uso de dos (2) metaheurísticas: *Optimización basada en Colonia de Hormigas y Metaheurística Golosa,* los que permitirán la **generación automática de casos de prueba para test sobre una GUI (Graphical User Interface)**

Finalmente con los casos de prueba generados, estos puedan ser aplicados al producto final (pruebas funcionales) y que permitan detectar en qué puntos el producto no cumple sus especificaciones; es decir, comprobar su funcionalidad. Esto facilitará a las empresas de software que necesiten modificar algún artefacto o componente del sistema por cambios en el negocio, para que puedan contar con una herramienta de software que permita generar nuevos casos de prueba asociados al cambio realizado.

Palabras Claves: Casos de Prueba, Árbol de Secuencia de Eventos, Criterios de Cobertura, Algoritmos Metaheurísticos, Optimización de Colonia de Hormigas, GUI.

Automatic Generation of GUI Test Cases using Ant Colony Optimization and Greedy algorithm

ABSTRACT

Today, due to the increased size and complexity of the software, the test process has become a vital task in the development of any computer system. The quality of a test case is measured by the coverage criteria, which depend on the length of the paths in the tree of sequences of events. Find those feasible paths and comply the coverage criteria is a highly combinatorial problem. For such problems is well justified the use of heuristics and metaheuristics algorithms.

In this perspective, we present a proposal of using two (2) metaheuristics: *Ant Colony Optimization and Greedy,* which enable the **automatic generation of test cases on a GUI.**

Finally with the generated test cases, they can be applied to the final product (tests) and detect at which points the product does not perform its specifications, i.e. verify its functional qualities. This will enable to change any device or system component by changes in the business in any software companies, so they can have a software tool that can generate new test cases associated with the changes made.

Key words: Test Cases, Tree Sequence of Events, Coverage Criteria, Metaheuristics Algorithms, Ant Colony Optimization, Graphical User Interface.

ÍNDICE

Lista de figuras

Lista de tablas

Capítulo 1: Introducción

1.1 Planteamiento del Problema

1.1.1 Antecedentes

Hoy en día, debido al aumento del tamaño y la complejidad del software, el proceso de prueba se ha convertido en una tarea vital en el desarrollo o mantenimiento de cualquier sistema informático, es por ello que la fase de pruebas es la última oportunidad en el proceso de desarrollo de software para detectar y corregir sus posibles anomalías a un costo razonable, ya que la forma generalizada de trabajo utilizada por los profesionales en el área es la de ejecutar la prueba sobre el producto "terminado" [1].

En sentido estricto, y a pesar de la gran cantidad de esfuerzo e importancia de las tareas de pruebas, se vuelve necesario probar el software en múltiples ocasiones.

El estándar internacional que define los procesos del ciclo de vida del software, la ISO 12207 [2], no define un proceso de pruebas como tal, sino que aconseja durante la ejecución de los procesos principales o de la organización, utilizar los procesos de soporte de Validación y Verificación.

Myers, define el proceso de prueba del software como la búsqueda de fallos mediante la ejecución del sistema o de partes del mismo en un entorno y con valores de entrada bien definidos [3]. La mayor parte de los defectos se introducen en las etapas tempranas del proceso de desarrollo, y de una manera irrefutable se ha comprobado que el costo de su corrección aumenta a medida que el error permanece no detectado [4].

A pesar de constituir una buena práctica el hecho de definir el plan de pruebas desde las primeras etapas, muy pocos se acogen a esta idea y comienzan a pensar en las pruebas solo después de tener el código frente a ellos [5]. La mayoría de los equipos de desarrollo de software cuentan con un tiempo limitado para la creación de pruebas minuciosas y bien planificadas, lo que trae consigo un atropello de pruebas en su mayoría funcionales y sin una planificación previa, lo anterior es equivalente a que se hagan pruebas redundantes, que haya caminos que no sean probados y que las entradas a los casos de prueba no sean adecuadamente escogidas [6].

Referente a nuestro tema de estudio, conocemos la publicación de Atif M. Memon cuyo título es *"An event-flow model of GUI-based applications for testing",* planteándose dos puntos importantes: [7]

- *El Modelo de Flujo de Eventos:* modelamiento de los eventos y la interacción entre ellos.
- *La construcción del Modelo de Flujo de Eventos:* construcción del grafo de flujo de eventos y el árbol de integración.

Otra referencia como aporte a esta investigación, es la tesis de Daniella Rojas Pacheco cuyo título es "Generación de Casos de Pruebas Unitarios para Java basados en la Técnica de McGregor & Sykes", donde se propone el uso de la técnica del *Análisis de Valores Límites,* permitiéndole generar casos más eficaces a la hora de cubrir errores [8].

1.1.2 Formulación del Problema

Los métodos de pruebas funcionales usados para la generación de casos de prueba, que se orientan a usar alguna técnica conocida o que se realizan manualmente, conlleva a que los desarrolladores dediquen mayor tiempo a su construcción. Solo el diseño de los casos de prueba puede significar un esfuerzo considerable y supone el 50% del costo total del desarrollo del software [9].

Aunque la literatura de Ingeniería de Software propone una y otra vez este tipo de pruebas, lo cierto es que las técnicas y herramientas concretas disponibles actualmente aún están muy lejos como para hacer del testing una actividad rentable para la mayoría de los equipos de desarrollo [10].

Se requiere entonces de una herramienta que permita la generación de los casos de prueba, con alta probabilidad de descubrir un error, necesaria para realizar las pruebas funcionales a un programa de software (interfaces gráficas de usuario)

1.2 Objetivos

1.2.1 Objetivo General

- ✓ Desarrollar una herramienta de software para mejorar las técnicas que actualmente se usan para la generación de casos de prueba para test de una GUI

de uso común en aplicaciones de escritorio (como formularios), aplicando para ello Colonia de Hormigas y metaheurística Golosa.

1.2.2 Objetivos Específicos

- ✓ Realizar una investigación bibliográfica en el uso de algoritmos, modelos matemáticos y/o herramientas capaces de encontrar y optimizar buenos casos de prueba.
- ✓ Establecer los requerimientos que debe cumplir una herramienta para generar casos de prueba.
- ✓ Desarrollar un aplicativo computacional que permita la "generación automática de casos de prueba para test de una GUI, basado en colonia de hormigas y metaheurística golosa".
- ✓ Analizar los resultados de la técnica empleada, respecto a su cobertura y efectividad.

1.3 Justificación

Hoy en día, usualmente se entiende que el testing es una actividad que se realiza una vez que los programadores han terminado de codificar. Entendido de esta forma el testing se convierte en una actividad costosa e ineficiente desde varios puntos de vista:

- Los testers estarán ociosos durante la mayor parte del proyecto y estarán sobrecargados de trabajo cuando este esté por finalizar.
- Los errores tienden a ser detectados muy tarde.
- Se descubre un gran número de errores cuando el presupuesto se está terminando.
- Los errores tienden a ser detectados por los usuarios y no por el personal de desarrollo, lo que implica un desprestigio para el grupo de desarrollo.

La motivación de realizar este trabajo surge para mitigar la falta de objetividad al realizar testing que puede ocurrir cuando la organización que desarrolla es la misma que ejecuta las pruebas, esto, sumado a las presiones institucionales por salir al mercado hacen que en muchos casos la calidad del software se conozca recién cuando se pone en producción. Para la búsqueda de la calidad de los productos y el logro de sus objetivos, se vuelve indispensable contar con modelos y técnicas las cuales sean capaces de

asegurar y facilitar el uso de sus aplicaciones. La tarea de las pruebas es una de las más importantes y más costosas, por ello es importante asegurar que el proceso se realice con una cierta calidad.

Quiero puntualizar tres aspectos referidos a la calidad en el software:

- ✓ *En el software, la calidad no es opcional:* No se puede elegir fabricar software de baja calidad y rebajar el precio. Se puede restar funcionalidad, pero no calidad.
- ✓ Nadie recuerda quien hizo un buen software (de calidad), pero nadie olvida el que fallaba constantemente.
- ✓ Y finalmente pasa que "algunas empresas con grandes carteles en recepción del estilo "La calidad es nuestra esencia" o "La calidad al servicio del cliente" y con todas las certificaciones habidas y por haber de 'calidad', no tienen ni un solo especialista en calidad de software, ni un solo especialista en probar aplicaciones. Y no, los desarrolladores, no son expertos en calidad y pruebas."

Por esta razón y con el ánimo de explorar la efectividad de la metaheurística Colonia de Hormigas, se decidió desarrollar el presente trabajo de grado **"Generación Automática de Casos de Prueba para Test de una GUI, usando Colonia de Hormigas y Metaheurística Golosa".**

Se eligió las metaheurísticas ya que ofrecen un equilibrio adecuado entre los algoritmos exactos y los algoritmos heurísticos: son métodos genéricos que ofrecen soluciones de buena calidad (el óptimo global en muchos casos) en un tiempo moderado.

> *"La utilización de algoritmos exactos tiene como principal ventaja que, garantizan encontrar el óptimo global en cualquier problema, pero tienen el grave inconveniente de que en problemas reales (que suelen ser NP-duros en la mayoría de los casos) su tiempo de ejecución crece de forma exponencial con el tamaño del problema. En cambio, los algoritmos heurísticos son normalmente bastante rápidos, pero la calidad de las soluciones encontradas está habitualmente lejos de ser óptima. Otro inconveniente de los heurísticos es que no son fáciles de definir en determinados problemas."*

Con la presente investigación se pretende:

- ✓ Que las Áreas de Desarrollo cuenten con una herramienta de software capaz de reducir el tiempo dedicado en la construcción de los casos de prueba, y así poder

realizar las pruebas del sistema (o funcionales) importantes para las fases posteriores.

- ✓ Cuando el software es modificado, es necesario volver a aplicar algunas pruebas para comprobar que no se han introducido nuevos errores. Esto es lo que se conoce como *pruebas de regresión.* Muchas de estas pruebas pueden reutilizarse de una etapa anterior, cuando el software pasó por primera vez la fase de pruebas.
- ✓ Aumentar el conocimiento en el uso de técnicas y/o herramientas de testeo de software.
- ✓ Nuestra herramienta de software no es la panacea, pero ayuda.
- ✓ La automatización es complementaria, ya que facilita la repetibilidad.

1.4 Alcance

- ✓ En el presente trabajo de investigación se pretende implementar un aplicativo capaz de encontrar "buenos" casos de prueba; es decir, casos de prueba que tengan una probabilidad alta de descubrir un error. Es por ello que el enfoque más popular para conseguir conjuntos de casos de prueba adecuados es *usar criterios de cobertura:* un conjunto de casos de prueba se considera adecuado si consigue cubrir una gran cantidad de eventos (longitud de los caminos recorridos) del objeto de prueba.
- ✓ El modelo a desarrollar tendría como entrada una *Secuencia de Eventos-GUI,* y como salida la generación de los *Casos de prueba.*
- ✓ Los casos de prueba para test de una GUI, son del tipo unitarios.
- ✓ No se realizaría la prueba del software, solo abarcaría la generación automática de los casos de prueba para test en una GUI.
- ✓ Se plantearán algunos casos de ejemplo con interfaces simples y complejas, como formularios, de uso común en aplicaciones de escritorio.

Capítulo 2: Estado del Arte

Para maximizar la calidad de nuestros casos de prueba, es necesario considerar los siguientes aspectos: Que los casos de prueba sean medibles, esto a través de los criterios de cobertura los cuales van a depender de la longitud de los caminos recorridos, y, tratar de encontrar de forma eficiente un conjunto pequeño de casos de prueba.

2.1 Taxonomía según ACM

Clasificación primaria:

D. Software

↳ **D.2** SOFTWARE ENGINEERING

↳ **D.2.4** Software/Program Verification

↳ **Subjects:** Validation

Clasificación secundaria:

D. Software

↳ **D.2** SOFTWARE ENGINEERING

↳ **D.2.5** Testing and Debugging

↳ **Subjects:** Testing tools

2.2 Revisión de la literatura

En la naturaleza, los seres vivos se enfrentan a problemas que deben resolver con éxito para desarrollarse, como obtener más luz del sol, o conseguir alimento para su subsistencia. La sapiencia de la naturaleza para resolver estos problemas ha sido siempre admirada e imitada en muchos campos de la ciencia. Una de ellas, la computación, ha creado técnicas de Inteligencia Artificial con el objeto de plasmar en sistemas computacionales la esencia misma de la naturaleza observable [11].

En el área de Inteligencia Artificial (IA) se pueden observar, dos enfoques diferentes en función del objetivo:

- La concepción de IA como el intento de desarrollar una tecnología capaz de suministrar a la computadora capacidades de razonamiento o discernimiento similares, o aparentemente similares, a las de la inteligencia humana.
- La concepción de IA como investigación relativa a los mecanismos de inteligencia humana (por extensión, investigación relativa a la vida y al universo), que emplea la computadora como herramienta de simulación para la validación de teorías.

Una posible clasificación de los paradigmas de Inteligencia Artificial es presentada a continuación [12]:

<table>
<tr><td rowspan="2">Enfoque Simbólico
“top-down”</td><td>a. Sistemas Expertos (Expert System)

Son llamados así porque emulan el razonamiento de un experto en un dominio concreto y en ocasiones son usados por éstos. Con los sistemas expertos se busca una mejor calidad y rapidez en las respuestas dando así lugar a una mejora de la productividad del experto.</td></tr>
<tr><td>b. Lógica Difusa (Fuzzy Logic)

Ó lógica heurística. Se utiliza para la resolución de una variedad de problemas, principalmente los relacionados con control de procesos industriales complejos y sistemas de decisión en general, la resolución y la compresión de datos.</td></tr>
<tr><td rowspan="2">Enfoque Subsimbólico
“botton-up”</td><td>a. Computación Evolutiva

Rama de la inteligencia artificial que involucra problemas de optimización combinatoria. Se inspira en los mecanismos de Evolución biológica:</td></tr>
<tr><td>✓ Templado Simulado (Simulated Annealing)

Es un algoritmo de búsqueda meta-heurística para problemas de optimización global.</td></tr>
</table>

	✓ *Algoritmos Genéticos (Genetic Algorithms)* Es un método de búsqueda dirigida basada en *probabilidad.* Bajo una condición muy débil (que el algoritmo mantenga elitismo; es decir, guarde siempre al mejor elemento de la población sin hacerle ningún cambio) se puede demostrar que el algoritmo *converge en probabilidad* al óptimo. ✓ *Programación Genética (Genetic Programming)* Es una especialización de los algoritmos genéticos en la que cada individuo de la población es un programa informático. Es una metodología basada en los *algoritmos evolutivos,* y que se inspira en la evolución biológica. ✓ *Estrategias Evolutivas (Evolutionary Strategies)* Esta técnica está básicamente enfocada hacia la optimización paramétrica. Son algoritmos evolutivos enfocados hacia la optimización paramétrica, teniendo como características principales que utilizan una representación a través de vectores reales, una selección determinística y operadores genéticos específicos de cruce y mutación. ✓ *Clasificadores Genéticos (Genetic Clasifiers)* Son algoritmos evolutivos cuya finalidad es obtener un sistema clasificador. Normalmente los clasificadores genéticos hacen evolucionar un conjunto de reglas que serían las encargadas de realizar la clasificación, y en ese caso se pueda considerar como un tipo de *Programación Evolutiva.* ✓ *Programas Evolutivos (Evolutionary Programming)* Se hace evolucionar una población de estructuras de datos sometiéndolas a una serie de transformaciones

	específicas y a un proceso de selección. Los datos utilizados son decimales. ✓ *Algoritmos Miméticos (Memetic Algorithms)* Se definen como la hibridización del algoritmo genético, que se encargará de realizar una búsqueda exploratoria, con un algoritmo exploratorio.
	b. Redes Neuronales Artificiales Son sistemas que intentan simular el cerebro, inspirados en su capacidad de aprender, sea por medio de nuevas instrucciones, o basándose en su experiencia. Las Redes Neuronales Artificiales fueron originalmente una simulación abstracta de los sistemas nerviosos bilógicos, formados por un conjunto de unidades llamadas "neuronas" o "nodos" conectadas unas con otras.
	c. Vida Artificial Es una disciplina que estudia la vida natural, recreando los fenómenos biológicos en computadoras y otros medios artificiales. Complementa el estudio teórico de la biología pero en lugar de tomar organismos aislados y analizar su comportamiento, lo que se intenta es colocar juntos organismos que actúan como los seres vivos. Las publicaciones realizadas sobre Vida Artificial pueden ser clasificadas en las siguientes áreas: ✓ Comportamiento adaptativo y social. ✓ Biología evolucionaria. ✓ Aprendizaje / Robótica / Aplicaciones.

Tabla 2.1 Clasificación de los paradigmas de IA [12].

Nuestra investigación trata sobre una de las más recientes técnicas de inteligencia artificial en el área de *Vida Artificial,* conocida como *Sistema de Hormigas* (AS, Ant System en inglés), algoritmo inspirado en el comportamiento, a nivel de especie, de una colonia de hormigas que optimizan el camino para llegar desde el nido hasta su fuente de comida, utilizando para éste fin, comunicación indirecta por medio de una substancia química denominada *feromonas,* y su capacidad para adaptarse a cambios de ambiente.

2.3 Revisión de Métodos/Algoritmos

A continuación se analizará con mayor detalle los siguientes métodos/algoritmos existentes, los cuales se relacionan a los casos de prueba:

2.3.1 Algoritmos Genéticos para obtener nuevos casos de prueba

Los casos de prueba se combinan aleatoriamente para construir una población inicial de casos de prueba, y es allí donde los *Algoritmos Genéticos* (GA, Genetic Algorithms en inglés) se aplican para mejorar la habilidad de esta población inicial.

Se dice que el objetivo de los algoritmos genéticos, es la optimización de algoritmos basados en la genética natural y mecanismos de selección [13].

Comentaremos que un AG, utiliza tres operadores: Reproducción, Entrecruzamiento y Mutación:

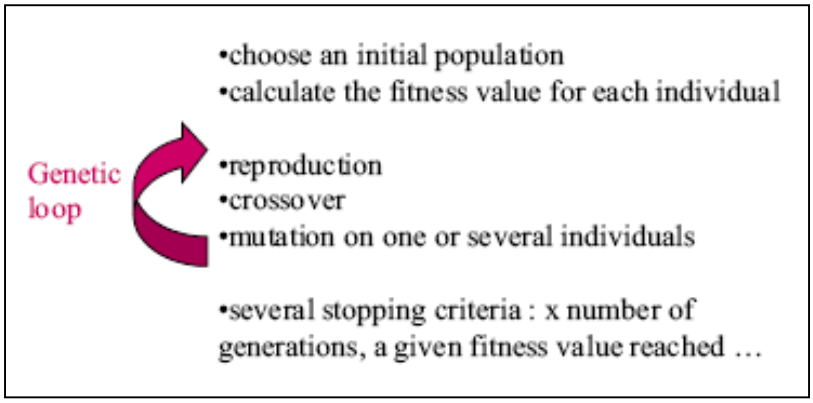


Figura 2.1 Proceso de los GA [13].

2.3.1.1 Análisis de Mutación

El Análisis de Mutación se aplica con éxito con el propósito de evaluar la calidad de los casos de pruebas unitarias para clases orientadas a objetos “OO”, además de estimar

cuantos nuevos casos de prueba son necesarios para probar un mejor componente de software dado [13].

Las Pruebas de Mutación o técnicas de prueba para entornos "OO", inicialmente fueron diseñadas para crear datos de pruebas eficaces. Ésta técnica consiste en crear un conjunto de versiones defectuosas o "mutantes" a un programa, con el objetivo de diseñar un conjunto de casos de prueba que distinga al programa de todos los mutantes.

- ✓ *Proceso de selección de pruebas* → Generación de mutantes a partir de las Pruebas sobre los Componentes (CUT), y aplicaciones de casos de prueba para matar c/mutante. También habla sobre el proceso incremental que se sigue en las pruebas de software:
  - Escribir casos de pruebas iniciales.
  - Optimizar los casos de pruebas iniciales.
  - Se comprueban si existen errores en el programa inicial; de ser el caso, se corrigen y se regresa al paso anterior.
- ✓ *Operadores de mutación* → tienen las siguientes características:
  - Independiente del lenguaje, para ser aplicado a diversos lenguajes OO (Java, C++, etc.)
  - El gasto de recursos computacionales, *es limitado*.
  - Garantizar la cobertura del flujo de control de los métodos.
  - Mutación para unidades de clase y pruebas del sistema.

2.3.1.2 Algoritmos Bacteriológicos para adaptar casos de prueba

Son una adaptación del enfoque genético para la generación de pruebas OO. El proceso comprende:

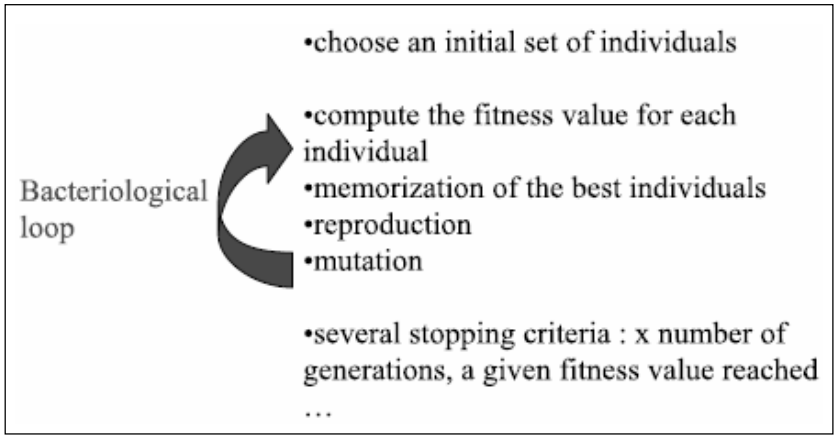


Figura 2.2 Proceso de los BA [13].

El modelo bacteriológico, *es más un enfoque de adaptación* que un *enfoque de optimización* (algoritmos genéticos). Su objetivo es la *mutación en la población inicial,* para adaptarlo a un entorno particular [13]. La adaptación es sobre la base de pequeños cambios en los individuos (llamados también bacterias)

A diferencia del modelo genético, las bacterias NO pueden dividirse. Solo pueden reproducirse y modificarse para mejorar la población. También, al igual que el modelo genético es necesario elegir una población inicial de individuos.

Para el modelo de optimización de las pruebas, cabe resaltar la aparición de la *Memorización.* El enfoque bacteriológico manipula una memoria que es el conjunto de las mejores bacterias que se han salvado de las generaciones anteriores.

2.3.2 Un Modelo de Eventos de flujo de aplicaciones basadas en su GUI para probar

Este modelo propuesto por Atif M. Memon [7], nos habla de la importancia que tienen las GUI como medio de interacción con el software de hoy.

Hace hincapié de la importancia que tienen los GUI cuando de diseñar y construir se trata, ya que en la actualidad los desarrolladores están dedicando cada vez más tiempo a su implementación (por encima del 60% del total del tiempo estimado para la construcción del software). A su vez nos habla que, una vez implementado el SW, las técnicas utilizadas para su validación, requieren una importante cantidad de esfuerzo manual para las Pruebas.

Es por ello que se propone un *Modelo de Flujo de Eventos,* el cual represente los eventos y sus interacciones. Al igual que los *modelos de flujo de control* representan

todas las posibles rutas de ejecución de un programa, y que los *modelos de flujo de datos* representan todas las posibles definiciones y usos de una ubicación de memoria, los *Modelos de Flujo de Eventos* nos representan todas las posibles secuencias de eventos que se pueden ejecutar en una GUI [7].

Actuales técnicas de pruebas sobre una GUI, usadas en la práctica, son incompletas y en gran medida manuales. Entre las herramientas utilizadas para este fin, tenemos el *WinRunner,* que proporciona muy poco de automatización, especialmente en la creación de los casos de prueba. Otras herramientas que requieren programación para cada caso de prueba, son las extensiones de *JUnit* como *JFCUnit, Abbot, Pounder, entre otras*.

Este modelo consta de 2 partes:

- ✓ *Codificar cada caso en términos de condiciones previas,* por ejemplo: El estado en el cual los eventos puedan ser ejecutados, los cambios en el estado después de que los eventos han sido ejecutados.
- ✓ *Representar todas las posibles secuencias de eventos,* que puedan ejecutarse dentro de una GUI como un conjunto de gráficos dirigidos.

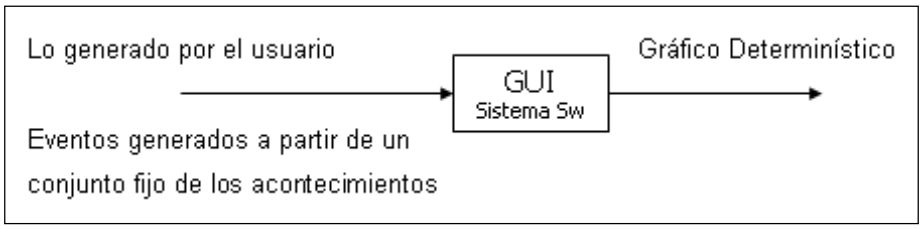


Figura 2.3 Construcción del Modelo de Flujo de Eventos [7].

Dentro del paradigma del *Modelo de Flujo de Eventos,* una importante parte se refiere al *comportamiento de los eventos [7]***.** Cada evento se representa en términos de cómo es que éste modifica el estado de una GUI cuando es ejecutado.

Entonces, podría decirse que el estado de una GUI es el estado colectivo de cada uno de sus objetos (botones, menús, etc.), y contenedores (frames, ventanas, etc.) los cuales a su vez contienen otros objetos. Cada objeto de una GUI está representado por un conjunto de *propiedades* de esos objetos, tales como: el color de fondo, el tipo de letra, texto en barra de título, etc.

El conjunto de objetos, con sus propiedades, son utilizados para crear el Modelo de Estado de las GUI.

La interacción de los eventos, representa todos los posibles eventos que interactúan en la GUI. Tal representación del espacio de interacción de los eventos, permitirá la personalización de varios *ESES's (Estrategia de Exploración del Espacio de Eventos)* los que recorrerán partes de este espacio para pruebas automatizadas: [7]

- ✓ Cada vértice representa un evento (clic en el botón Editar → Pegar). Un borde del vértice-X al vértice-Y, muestra que el evento-Y puede ser realizado inmediatamente después del evento-X.
- ✓ Este gráfico es similar al gráfico de control de flujo, donde los *vértices* representan la declaración del programa (en algunos casos porción de instrucciones básicas) y los *bordes* representan el orden posible de ejecución entre la declaración.
- ✓ En la práctica, *los GUI's son jerárquicos,* y esta jerarquía puede ser explotada para identificar grupos de eventos de la GUI, los que pueden ser modelados aisladamente. Una jerarquía de la GUI, corresponde a examinar la estructura de las *ventanas modales* en la GUI.

En conclusión, el autor hace una propuesta de cómo modelar una GUI en base a los eventos que puedan ocurrir, y que éstas al ser combinadas con ESES's personalizadas, puedan ser utilizados para diversos aspectos de pruebas sobre GUI's.

Con el *Modelo de Flujo de Eventos*, se puede generar rápidamente un gran número de casos de prueba para GUI. El modelo también promueve la reutilización, porque una vez creado el modelo, se puede generar otros casos de prueba si la versión de la GUI fue modificada.

2.3.2.1 Coverage Criteria for GUI Testing

Una vez establecido los modelos de flujo de eventos, propuesto por Atif Memon, se debe definir los criterios de cobertura para las GUIs; es decir, las reglas que proporcionen una medida objetiva de la calidad de las pruebas.

Una definición sería: Representan el conjunto de reglas o roles que nos ayudan a determinar si una GUI ha sido probada adecuadamente. Estos criterios de cobertura se

aplicarán sobre los eventos y la secuencia de eventos, los cuales nos permitirán conocer la medida de la prueba [14].

Dado que el número total de permutaciones de secuencia de eventos en una GUI no trivial, es muy grande, se descompondrá en componentes de GUI, donde cada uno de los cuales se utilizará como unidad básica de prueba.

La idea principal es definir criterios de pruebas en términos de eventos GUI, y la interacción entre éstos.

Un componente GUI se representa por una nueva estructura llamada *event-flow graph* (grafo de flujo de eventos), el cual identifica los eventos dentro de un éste. La interacción entre los componentes GUI se representarán en un *integration tree* (árbol de integración)

Para definir nuestro criterio de cobertura, necesitamos representar nuestros componentes GUI y los eventos dentro del componente.

Los componentes son definidos en términos de "Ventanas modales". Un ejemplo de un componente GUI es la ventana modal "FileOpen" en el MS-Word, donde el usuario interactúa con los eventos dentro de éste componente: seleccionando un archivo, cancelar, etc.

Una clasificación de eventos GUI es usado para identificar componentes. La clasificación de los eventos GUI, es:

- ✓ Eventos con restricción de foco (Restricted-focus events): ventanas modales.
- ✓ Eventos sin restricción de foco (Unrestricted-focus events): ventanas no modales.
- ✓ Eventos de terminación: cerrar una ventana, botones de un MsgBox.

Vamos a explicar el criterio de cobertura dentro del componente GUI "Intra-component Coverage".

Cobertura dentro del Componente (Intra-components Coverage)

Se refiere a definir un criterio de cobertura para los eventos y la interacción entre éstos. Para ello se definirán 3 coberturas, las cuales son:

- ✓ *Cobertura de Eventos,* la cobertura de un evento requiere que cada evento en el componente GUI se realice al menos una vez. Como requisito es necesario verificar si cada evento se ejecuta como se esperaba.
- ✓ *Cobertura de interacción de Eventos,* se refiere a verificar la interacción entre todos los pares posibles de eventos en el componente. Bajo este criterio, se requiere que después de que un evento "e" se realice, todos los eventos que interactúan con "e" serán ejecutados al menos una vez.
- ✓ *Cobertura de Longitud de la Secuencia de Eventos,* Se definirá un criterio el cual capturará el impacto en el contexto. Intuitivamente, el contexto para un evento "e" es la secuencia de eventos realizados antes de "e". Un evento puede ser realizado en un número infinito de contextos.

La definición de éste criterio, es: "Un conjunto P de una secuencia de eventos satisface el criterio de longitud-n de secuencia de eventos, si y solo si, P contiene todas las secuencia de eventos de longitud igual a n"

2.3.3 Ant System

Esta técnica aún no se ha aplicado en la generación automática de casos de prueba, lo cual es razón suficiente su aplicación/uso en esta tesis.

Una de las mayores bondades de ésta técnica, y como se explicará más adelante, es que puede ser aplicado a problemas que acepten una representación vía grafo. Además que soporta pruebas en problemas grandes, alrededor de 100 nodos aproximadamente.

A continuación se describirá la definición y origen de esta novedosa técnica, la cual se aplicará para nuestro tema de tesis.

Es una innovadora técnica basada en agentes muy simples llamados *hormigas,* nació con la tesis doctoral de Marco Dorigo (1992) [15] quien en 1996 [16], publicó tres variantes del algoritmo que se diferencian en el momento y la manera de actualizar una matriz que representa las feromonas de los sistemas biológicos:

- *Ant-density*: con actualización constante de las feromonas por donde pasa una hormiga.
- *Ant-quantity*: con actualización de feromonas inversamente proporcional a la distancia entre 2 aristas recorridas.

- *Ant-cycle*: con actualización de feromonas inversamente proporcional al trayecto completo, al terminar un recorrido. Este último presentó mejores resultados y las siguientes investigaciones se centraron en él.

Como *Ant System* es una clase de 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.

Existen diversas técnicas para resolver problemas de optimización, tales técnicas pueden ser clasificadas como técnicas exactas o técnicas aproximadas. Sin embargo, existen problemas para los cuales no se conoce un algoritmo que pueda resolver todas sus instancias en tiempo polinomial puesto que tienen espacios de búsqueda muy grandes, lo que hace pertinente el uso de técnicas aproximadas que suelen ser conocidas como heurísticas. La palabra *heurística* se deriva del griego *heuriskein,* que significa "encontrar" o "descubrir".

Sabemos que estas técnicas retoman su inspiración de diversas áreas de estudio tales como: matemáticas, física, química, biología, etc. De este último campo, podemos mencionar las heurísticas inspiradas en el principio de selección natural, las cuales se conocen en conjunto como *computación evolutiva* o *algoritmos evolutivos.*

2.3.3.1 Optimización por Colonia de Hormigas (ACO, Ant Colony Optimization en inglés)

Los algoritmos basados en *Colonia de Hormigas* son un tipo de metaheurística que engloba un conjunto de técnicas de optimización inspiradas en el comportamiento colectivo forrajeo de las hormigas, las cuales son capaces de encontrar un camino corto entre el nido y la fuente de alimento: [17]

> *"Estos algoritmos se han aplicado con éxito a problemas de optimización combinatoria que pueden transformarse en una búsqueda dentro de un grafo"*

La idea principal de éste algoritmo, consiste en usar hormigas artificiales que simulan dicho comportamiento en un escenario llámese los grafos. A continuación describiremos el proceso mediante el cual, utilizando la feromona, las hormigas son capaces de construir la ruta más corta entre el nido y el alimento.

Inicialmente, las hormigas desconocen el camino más corto entre el nido y la comida para lo cual, al inicio tendrán una distribución azarosa por todo el espacio explorado, ver Figura 2.4 (a). Al cabo de cierto tiempo, y sabiendo que las hormigas se mueven a una misma velocidad constante, la cantidad de feromona se hará más persistente en aquellos lugares que son más transitados y evitará su rápida evaporación. De tal forma, las distancias más cortas entre el nido y la fuente de alimento tenderán a ser transitadas más frecuentemente por las hormigas, lo que permitirá hacer más fuerte el rastro de los caminos más cortos y éstas tenderán a abandonar con el tiempo los caminos largos, ver Figura 2.4 (b)

Si esta ruta se pierde por el bloqueo de algún objeto o por los cambios ambientales, ver Figura 2.4 (c), una vez que las hormigas se encuentren en el punto que se ha perdido el rastro volverá a elegir aleatoriamente un camino, ver Figura 2.4 (d), algunas hacia arriba y otras hacia abajo. Sin embargo, aquellas que elijan ir por la parte de arriba que es el camino más corto, alcanzarán el otro extremo con mayor rapidez que las que eligieron el camino más largo, de tal forma que, transcurrido cierto tiempo (por retroalimentación de la feromona), volverán a restablecer el camino corto y terminarán abandonando el camino largo, ver Figura 2.4 (e)

De este modo, las colonias de hormigas pueden establecer rutas cortas utilizando la feromona como medio de intercambio de información, lo que se conoce como *stigmergy* [18]

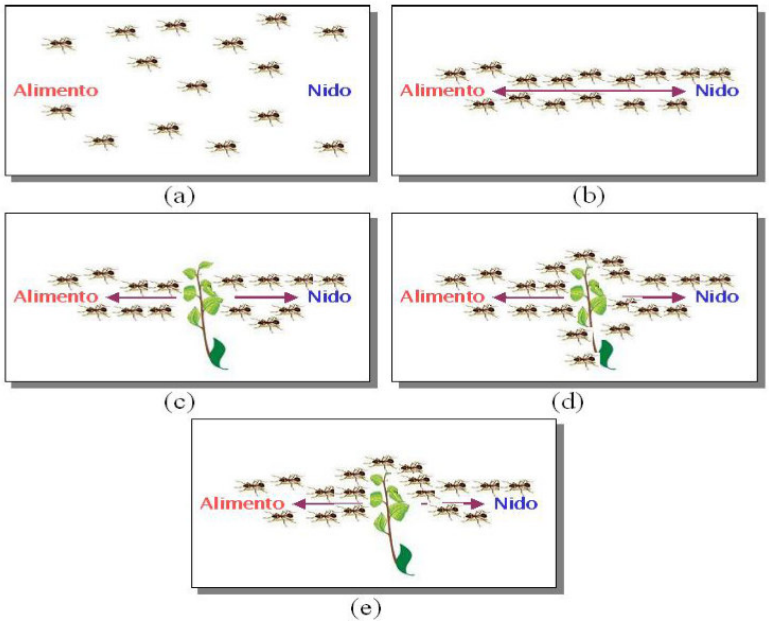


Figura 2.4 Comportamiento adaptativo de las hormigas [17].

De tal forma, aquellos rastros que contengan mayor cantidad de feromona serán seguidos con mayor seguridad por las hormigas, a diferencia de los rastros que son pobres en cantidades de este químico.

En definitiva, puede decirse que el proceso se caracteriza por una retroalimentación positiva, en la que la probabilidad con la que una hormiga escoge un camino aumenta con el número de hormigas que previamente hayan elegido el mismo camino.

El primer algoritmo basado en la optimización mediante colonias de hormigas fue aplicado al Problema del Viajante (Dorigo et al, 1996), obteniéndose unos resultados bastante alentadores. A partir de dicho algoritmo se han desarrollado diversos heurísticos que incluyen varias mejoras, y han sido aplicados no solo al TSP sino también a problemas como el VRP y el QAP entre otros (Dorigo et al, 1999)

El algoritmo sigue la estructura computacional básica siguiente:

Inicio de rutina o ciclos de rastros

Do While *(hasta que criterio no sea satisfecho)*

For *k=1* ***To*** *m*

Do Until *(hormiga "k" complete su recorrido)*

Seleccionar *el siguiente nodo a visitar*

End Do

End For

Calcular *la longitud del recorrido y* ***Actualizar*** *los niveles de feromona*

End Do

Figura 2.5 Algoritmo Ant Colony Optimization [17].

2.3.3.2 Descripción de la metaheurística ACO

Los algoritmos ACO son procesos iterativos. En cada iteración se "lanza" una colonia de *m* hormigas y cada una de las hormigas de la colonia construye una solución al problema. Las hormigas construyen las soluciones de manera probabilística, guiándose

por un rastro de feromona artificial y por una información calculada a priori de manera heurística. La regla probabilística es:

$$P_{[i][j]} = \left[\boldsymbol{\tau}[i][j]^{\alpha} * \eta[i][j]^{\beta} \Big/ \sum \boldsymbol{\tau}[i][j]^{\alpha} * \eta[i][j]^{\beta} \right]; \text{ con } j \notin \text{LT}$$

Siendo:

$$\eta_{[i][j]} = 1 \,/\, d_{[i][j]}$$

Donde:

$P_{[i][j]}$ es la probabilidad con la que la hormiga *k,* situada actualmente en el nodo *i,* elige ir al nodo *j* como próxima parada.

$\tau_{[i][j]}$ es la cantidad de feromona acumulada sobre el arco *(i, j)*

$\eta_{[i][j]}$ es la información heurística; es decir, el conocimiento previo

α parámetro para regular la influencia de $\tau_{[i][j]}$

β parámetro para regular la influencia de $\eta_{[i][j]}$

LT es la lista tabú el cual contiene los nodos visitados por la hormiga *k*

$d_{[i][j]}$ es la distancia entre los nodos *i* y *j*

Cuando todas las hormigas han construido una solución (cuando se termina la iteración) se calcula la longitud del recorrido generado por cada hormiga, y se actualiza el mejor recorrido encontrado hasta el momento.

Finalmente, se actualizan las cantidades de feromona en cada arco. La fórmula a seguir es:

$$\tau_{[i][j]}(t+1) = (1 - p).\, \tau_{[i][j]}(t) + \Delta\tau_{[i][j]}^{best}$$

$$\Delta\tau_{[i][j]}^{best} = \begin{cases} Q/L & si\ el\ arco\ (i,j)\ pertenece\ a\ T \\ 0 & en\ caso\ contrario \end{cases}$$

Donde:

p es el coeficiente de evaporación de la feromona [0, 1]

$\Delta\tau_{[i][j]}$ es el incremento en la cantidad de feromona en el arco (*i, j*)

Q es un valor constante que representa la cantidad de feromona depositada por una hormiga en cada recorrido

T es la mejor solución encontrada hasta el momento o bien la mejor solución encontrada en la iteración

L es la longitud de la solución T

En cada recorrido, cada hormiga deja una cantidad de feromona dada por Q/L_k, donde Q es una constante y L_k la longitud de su recorrido. Por lo tanto, en recorridos más cortos se deposita más feromona.

En el algoritmo se simula la evaporación de feromona que sucede en la realidad. Las cantidades de feromona se reducen a un factor (1 - p) antes de que se deposite una nueva feromona. Esto se hace para evitar convergencia prematura.

Tras la actualización de la feromona puede comenzarse una nueva iteración. El resultado final es la mejor solución encontrada a lo largo de todas las iteraciones realizadas.

2.3.4 Comparación de los Métodos

A continuación se evalúan las técnicas que permitan asegurar la calidad de los casos de prueba, evaluando si cumplen con los siguientes criterios básicos:

Criterio de Evaluación	**Atiff Memon (event-flow model)**	**Ant Colony Optimization (ACO)**
¿La técnica empleada está enfocada para pruebas del tipo caja negra?	1	0
¿Se parte de los eventos que interactúan sobre una GUI?	1	1
¿Es adecuado para ser aplicado a problemas que acepten una representación vía grafo?	1	1
¿Permite construir una amplia variedad de soluciones, con capacidad de exploración mucho mayor?	0	1

¿Es posible ser probado en problemas grandes?	0	1
	3	**4**

Tabla 2.2 Comparación de los métodos.

2.4 Casos de Prueba como un problema de Set Covering

Dentro de los problemas de subconjuntos, se encuentra el Problema de Cobertura de Conjuntos el cual tiene una gran cantidad de aplicaciones prácticas en la vida real, principalmente en lo que se refiere al uso correcto de los recursos de una organización, tanto a recursos materiales como de personas.

Este tipo de resolución de problemas [19] funciona a partir de un subconjunto de elementos que deben cumplir ciertas restricciones y tienen un costo asociado. De manera que al ser seleccionados los elementos sean cubiertas las restricciones y a partir de una matriz donde se representan los recursos y necesidades a resolver por el problema, se deben elegir tales combinaciones que sean cubiertas por al menos una columna con el objetivo de reducir al mínimo el costo de las columnas usadas [20, 21]

El *Set Covering* es un problema de la familia NP. Es por eso que se deben usar algoritmos aproximados o heurísticas, que trabajen en tiempo polinomial, para poder obtener una respuesta lo más aproximado a la exacta en un tiempo polinomial.

Un algoritmo que ofrece resolver el problema del *Set Covering* en tiempo polinomial es el *Greedy,* cuya función es seleccionar el conjunto con la cardinalidad más grande en cada iteración [22]. Existen también los siguientes algoritmos: Grasp y Genéticos, los cuales se describirán a continuación:

2.4.1 Algoritmo Genético para Set Covering

Los Algoritmos Genéticos son atribuidos a Holland [23] y a sus estudiantes en la década de los 70.

Beasley [24] presentó un algoritmo heurístico Lagrangiano e informó que su heurística había dado resultados de mejor calidad que otras heurísticas [25][26] para problemas

que involucran hasta 1000 filas y 10000 columnas. Por otra parte, Jacobs y Brusco [27] desarrollaron una heurística basada en Simulated Annealing e informaron su éxito considerable en problemas de hasta 1000 filas y 10000 columnas. Sen [28] investigó el rendimiento de un algoritmo de simulated annealing y de un algoritmo genético simple en el SCP de costo mínimo, pero dio pocos resultados computacionales. En otro estudio, Beasley [29] desarrolló una heurística para SCP sin costo único basado en algoritmos genéticos. Los resultados computaciones indicaron que dicha heurística era capaz de generar soluciones óptimas para problemas de tamaño pequeño y soluciones de alta calidad para problemas de gran tamaño.

Representación:

Lo primero que hay que definir es la representación a utilizar. Así, tenemos un vector de *n* bits (con *n* el número de columnas del problema) en donde un 1 en el bit *i* indica que esa columna (*i*) es parte de la solución.

Por definición, los AG utilizan representación binaria; sin embargo, esta representación tiene el problema que al aplicarle los operadores genéticos a los individuos, los resultados no siempre serán soluciones factibles. Esto obliga a aplicar una función de penalización en la función de evaluación para que quede representado el grado de no satisfacción de la solución. Otra forma de resolver este problema es aplicando una heurística que "arregle" la solución no factible, convirtiéndola en una factible.

Beasley [29] efectuó pruebas que indicaron que el rendimiento de un Algoritmo Genético utilizando una representación no binaria fue más bajo que el mismo utilizando una representación binaria.

2.4.1.1 El Algoritmo Genético estándar

Los Algoritmos Genéticos (AG) estándar, por su parte, no fueron desarrollados para solucionar problemas de optimización y por lo tanto no consideran las restricciones que pueda tener un problema determinado. Sin embargo, con pequeñas modificaciones, se pueden obtener muy buenos resultados en problemas de optimización.

Los AG copian el proceso evolucionario mediante el procesamiento simultaneo de una población de soluciones. Por lo general, se comienza con una población generada de manera aleatoria, cuyos individuos se van recombinando de acuerdo al valor que tengan en la función de evaluación. Así, mientras mejor sea el valor obtenido por el individuo,

más posibilidades tiene de ser seleccionado para generar nuevas soluciones. De esta forma, las nuevas generaciones de soluciones conservan lo bueno de las generaciones anteriores permitiendo, después de un proceso iterativo, llegar a un valor que eventualmente podría ser el óptimo.

Para hacer las combinaciones, el AG estándar posee dos operadores genéticos que son: el cruzamiento y la mutación. El primero toma dos individuos y forma un tercero mediante la combinación de los componentes de los individuos padres, permitiendo de esta forma que el algoritmo efectúe una explotación del espacio de búsqueda. Por otra parte, la mutación toma un individuo y por medio de un proceso genera otro individuo para la nueva población. Esto permite que el algoritmo efectúe una exploración del espacio de búsqueda.

En líneas generales, un AG estándar se puede ver como un proceso que efectúa lo siguiente:

1. begin

2. Generar una población inicial.

3. Evaluar el fitness de los individuos de la población

4. repetir

5. Seleccionar individuos padres desde la población

6. Aplicar los operadores genéticos para generar nuevos individuos

7. Evaluar el fitness de los individuos hijos

8. Reemplazar algunos o todos los individuos de la población por los hijos

9. hasta condición de término

10. end

Figura 2.6 Procesos de un AG [23].

Implementación:

Para implementar un AG es necesario definir los siguientes puntos:

- ✓ Representación: Este es el punto de partida en la definición de un AG. De esta definición va a depender gran parte de la estructura final del AG. Como comentamos anteriormente, los AG utilizan una representación binaria, pero

existen variaciones como los Algoritmos Evolutivos que permiten otros tipos de representaciones.

- ✓ Tratamiento de los individuos no factibles: Dado que los AG no están diseñados para trabajar con restricciones, es necesario definir un criterio o un mecanismo para tratar los individuos no factibles.
- ✓ Inicialización: Se refiere a cómo se va a generar la población inicial. Generalmente, dicha población se genera de manera aleatoria.
- ✓ Condición de término: Se debe establecer hasta cuándo va a correr el algoritmo. Generalmente se utiliza un número determinado de generaciones (iteraciones) o una medida de convergencia.
- ✓ Funciones de Evaluación o Fitness: Es la función que va a determinar que tan buena es una solución. La función de evaluación puede ser igual o diferente a la función objetivo.
- ✓ Operadores Genéticos: Los operadores genéticos son los encargados de efectuar la reproducción de la población, siendo los más comunes el cruzamiento y la mutación.
- ✓ Selección: Se refiere a como se van a seleccionar los individuos para aplicarles los operadores genéticos. La selección debe dirigir el proceso de búsqueda en favor de los miembros más aptos.
- ✓ Reemplazo: Este punto se refiere a cómo se va a reemplazar la población con los individuos generados, lo cual puede diferir del cómo se seleccionan. Es decir, los criterios de selección y reemplazo no tienen por qué ser iguales.
- ✓ Parámetros de funcionamiento: Un AG necesita que se le proporcionen ciertos parámetros de funcionamiento, tales como el tamaño de la población, la probabilidad de la aplicación de los operadores genéticos y la cantidad de generaciones, entre otros.

2.4.1.2 Algoritmo Genético para resolver el problema de Set Covering

Basándonos en lo anterior descrito y en los trabajos de Beasley [29] y Cormier [30], se definirán las siguientes variables como parte del algoritmo genético para resolver el problema de Set Covering:

I = conjunto de todas las filas

J = conjunto de todas las columnas

a_i = conjunto de columnas que cubren la fila i, $i \in I$

b_i = conjunto de filas cubiertas por la columna j, $i \in J$

S = conjunto de columnas en una solución

U = conjunto de filas no cubiertas

w_i = número de columnas que cubren la fila i, $i \in I$ en S

c_j = costo de la columnas j

A continuación se describe el cómo se desarrollaron los distintos puntos que definen un AG:

A. Representación

Como se indicó anteriormente, lo primero que se debe definir es la representación que se va a utilizar. Para este caso se usará la representación binaria por ser la representación más natural para el problema de set covering. Así, tendremos un vector de igual largo que la cantidad de filas a ser cubiertas:

columna (gen)	1	2	3	4	...	n-1	n
string de bits	0	1	0	0	...	1	0

Figura 2.7 Representación binaria de un individuo [29].

B. Proceso previo

Una vez definida la representación, se debe programar una rutina que tome los datos de un archivo y transforme dicha información a una representación binaria. Mediante una función se debe tomar los datos desde un archivo determinado y crear las matrices a_i,b_j y el vector de costos c, lo que permite que el algoritmo trabaje con esos datos. El nombre del archivo de entrada es indicado al programa a través del teclado.

C. Tratamiento de los individuos no factibles

Las soluciones modificadas por los operadores genéticos no siempre se van a mantener factibles, por lo que hay que aplicar algún método o función que trabaje con las soluciones no factibles. Para considerar lo anterior, se probaron dos estrategias. La primera basada en el trabajo de [29] y consiste en aplicar una heurística que arregla una solución no factible y la convierte en una factible. Dicha heurística se implementa en una función *arregla()* y provee, además, un paso de optimización local con el fin de hacer al AG más eficiente. Los pasos requeridos para hacer factible una solución pasa por determinar cuáles filas aún no han sido cubiertas y escoger las columnas necesarias para su cobertura. La búsqueda de dichas columnas se basa en la proporción:

$$\frac{\text{costo de una columna}}{\text{cantidad de filas no cubiertas que cubre}}$$

Una vez que la solución se ha vuelto factible, se aplica un paso de optimización que elimina aquellas columnas redundantes. Una columna redundante es aquella que si se quita, la solución sigue siendo factible. La heurística efectúa, a grandes rasgos, lo siguiente:

Procedimiento arregla()

begin

Inicializar $w_i = | S \cap a_i | \; \forall i \in I$

Inicializar $U = \{ i | w_i = 0, \forall i \in I \}$

Para cada fila *i* en U (en orden creciente de *i*):

Encontrar la primera columna *j* (en orden creciente de *j*) en ai, que minimice $c_j / (U \cap b_j)$

Incluir *j* en *S* y hacer $w_i = w_i + 1, \; \forall_i \in b_j$. Hacer $U = U - b_j$

Para cada columna *j* en *S* (en orden decreciente de *j*), si $w_i >= 2, \; \forall_i \in b_j$, hacer $S=S-j$ y $w_i = w_i - 1, \; \forall_i \in bj$

Ahora *S* es una solución factible del problema que no contiene columnas redundantes.

end

D. Inicialización

El tamaño de la población, así como la población inicial son elegidos de manera tal que el dominio de las soluciones asociadas con la población este cubierto adecuadamente. El tamaño de la población depende de cómo se genere la población inicial. Dicho método lo implementa la función *inicia()*

E. Condición de término

Los AG poseen básicamente dos condiciones de término: número máximo de generaciones u convergencia en solución. En el primer caso, el algoritmo itera un número determinado de veces, luego se detiene y entrega el mejor valor que tenga hasta ese momento. En el segundo caso, el algoritmo itera que los individuos de la población converjan en un único valor.

F. Función de Evaluación

Como se dijo anteriormente, para trabajar con soluciones no factibles se utilizaron dos estrategias- En la primera, utilizando una heurística que repara las soluciones, las funciones de evaluación y objetivo son iguales y son implementadas a través de la función *evalua():*

$$\sum_{j=1}^{n} cj * xj$$

De la misma manera, en la segunda estrategia, donde se utiliza una función de penalización, a la función objetivo se le agrega un factor proporcional a la cantidad de filas no cubiertas. De esta forma, la función de evaluación queda:

$$\sum_{j=1}^{n} [Cj * Xj + (k * factor\ penzalización)]$$

Donde:

$x_j = \{1 / 0\}$, $\forall j \in N$

k = Determina el grado de severidad con que el valor de penalización es construido.

factor penalización = Es un parámetro para el funcionamiento del algoritmo

Dicho factor de penalización debe tener un valor adecuado de forma tal que afecte al resultado final. Es decir, si el valor de dicho factor es muy pequeño y el problema es de minimización, no habrá diferencia notables en la función de evaluación respecto a una solución válida.

G. Operadores Genéticos

Los operadores genéticos utilizados en el desarrollo del algoritmo fueron el cruzamiento y la mutación. En el caso del cruzamiento, primero se implementó el operador de fusión propuesto por Beasley [29], la cual toma dos individuos padres y genera un solo individuo hijo. Su funcionamiento es el siguiente: Dadas las soluciones *P1* y *P2,* con sus respectivos valores de fitness f_{p1} y f_{p2} y el individuo hijo *C,* se tiene $\forall i = 1,...,n$.

begin

Si $P1[i] = P2[i]$, entonces $C[i] = P1[i] = P2[i]$

Si $P1[i] \neq P2[i]$, entonces:

$C[i] = P1[i]$ con probabilidad: $p = f_{p2} / (f_{p1} + f_{p2})$

$C[i] = P2[i]$ con probabilidad: $1 - p$

end

Lo destacable de este operador es que considera los valores de fitness para determinar qué elementos se copian al hijo, logrando pasar información de mayor valor a las demás descendencias.

H. Selección

Los operadores genéticos necesitan que se les entregue uno o dos individuos para procesarlos. Esto hace necesario que se defina un criterio de selección de los individuos.

Por su parte, dado que el problema de set covering es de minimización, la probabilidad que un individuo sea seleccionado está determinado por:

$$p_i = \frac{1/fi}{\sum_{i=1}^{m} 1/fi}$$

Donde: f_i es el valor obtenido por la solución i en la función de evaluación.

I. Reemplazo

Una vez generada la nueva población con los individuos hijos, se evalúan y pasa completa como población de padres a la nueva generación.

J. Parámetros de funcionamiento

El AG posee varios parámetros de funcionamiento entre los que se encuentran:

- ✓ Tamaño de la población.
- ✓ Número máximo de generaciones.
- ✓ Probabilidad de aplicación del operador de cruzamiento.
- ✓ Probabilidad de aplicación del operador de mutación.
- ✓ Factor de penalización.

K. Funcionamiento general del Algoritmo

En líneas generales el AG desarrollado tiene el siguiente funcionamiento:

Procedimiento AG

begin

Generar una población inicial de N soluciones aleatorias. Hacer $t = 0$

repetir

Seleccionar 2 soluciones *P1* y *P2* de la población utilizando el método de la ruleta

Combinar *P1* y *P2* para formar una nueva solución *C* utilizando el operador de cruzamiento

Muta k columnas seleccionadas de *C* en forma aleatoria, donde k está determinado por la probabilidad de aplicación del operador de mutación.

Hacer *C* válida y remover las columnas redundantes en *C* aplicando el operador de heurística

Hacer $t = t + 1$

hasta que $t = M$ soluciones se hayan generado. La mejor solución encontrada es la con el menor fitness en la población.

end

2.4.2 Algoritmo Greedy para Set Covering

Se llama algoritmo ávido o voraz (Greedy) a un algoritmo de optimización que en cada paso de decisión toma la mejor alternativa en ese momento.

Este esquema algorítmico es el que menos dificultades plantea a la hora de diseñar y comprobar su funcionamiento.

Las principales características/ventajas de usar Greedy, son:

- Se aplican a problemas de optimización.
- Suelen ser rápidos y fáciles de implementar.
- Toma las decisiones en función de la información que está disponible en cada momento.
- Una vez tomada la decisión no vuelve a replantearse en el futuro.
- La velocidad de ejecución es muy rápida además de usar muy poca memoria.
- Seleccionar una solución parcial "localmente óptima", y llevarla a una solución "globalmente óptima".

Es cierto que este algoritmo no puede dar lugar a una solución óptima, pero sí encuentra la "mejor" elección en cada etapa.

Definición del Problema:

Se nos da el universo U, tal que: $|U|=n$, y los conjuntos $S_1, \ldots, S_k \subseteq U$. Un *set cover* es una colección C de algunos de los conjuntos de $S_1, \ldots, S_k$, cuya unión es el universo entero U.

Formalmente, C es un *set cover,* si $\bigcup_{S_i \in C} S_i = U$.

El objetivo es minimizar $|C|$

El algoritmo Greedy para poder cubrir un conjunto ponderado debe construir una cobertura en varias ocasiones, seleccionando de un conjunto s que permita minimizar el peso w_s dividido por el número de elementos en s que no han sido cubiertos por los conjuntos elegidos. Se detiene y devuelve los conjuntos elegidos cuando forman la cobertura.

Greedy-set-cover(S, w)

1. Initialize $C \leftarrow \emptyset$. Define $f(C) = | \cup_{s \in C} S |$
2. Repeat until $f(C) = f(S)$
3. Choose $s \in S$ minimizing the price per element $w_s / [f(C \cup \{s\}) - f(C)]$
4. Let $C \leftarrow C \cup \{s\}$
5. Return $\{C\}$

Figura 2.8 Greedy heuristic for the set-covering problem [31].

2.4.3 Algoritmo Grasp para Set Covering

El nombre viene de su acrónimo en inglés *Greedy Randomized Adaptive Search Procedure* (GRASP), que en español se traduce como *procedimientos de búsqueda basados en funciones ávidas, aleatorias y adaptativas,* es una estrategia metaheurística que construye soluciones usando una aleatoriedad controlada mediante una función voraz.

GRASP se basa en el siguiente principio de operación:

> *"Es un procedimiento multi-arranque en el que cada arranque se corresponde con una iteración. Cada iteración tiene dos fases bien definidas: la fase de construcción, que se encarga de obtener una solución factible de alta calidad; la fase de mejora, que se basa en la optimización (local) de la solución obtenida en la primera fase"*

GRASP fue propuesto originalmente para el set covering problem por T. Feo y M. Resende a finales de los ochenta [32]. No fue hasta el trabajo que se publicó en el año 1995 cuando adquiere una terminología y forma definitiva como metaheurística de propósito general. Una amplia descripción de esta metodología se puede encontrar en Feo y Resende [33] y Pitsoulis y Resende [34]

Los orígenes algorítmicos de GRASP provienen de la *metaheurística semi-constructiva (semi-constructive heuristic).* Esta técnica también se caracteriza por ser un método multi-arranque basado en una construcción miope aleatorizada. La diferencia fundamental con respecto a GRASP es que la primera técnica no utilizaba un procedimiento de mejora (búsqueda local)

Una de las mayores ventajas de la metaheurística GRASP es lo fácil que este esquema general puede ser adecuado para la solución de un problema particular.

Procedimiento para el set covering:

Sea *C* la solución que se va a construir, se tienen los conjuntos $S_1, \ldots, S_k$, un *set cover* es una colección *C* de algunos de los conjuntos de $S_1, \ldots, S_k$, donde además se tienen los siguientes parámetros: *T = tamaño máximo de lista / α = es la tasa de relajación / RCL = restricted candidate list*

El procedimiento se describe de la forma siguiente:

1. Let $C \leftarrow \emptyset$
2. Get $T = \max \{ |S_i| / i=1\ldots m \}$
3. **While** $(T > \emptyset)$
4. Build $RCL = \{ S_i / |S_i| \geq \alpha^*(T), i=1\ldots m \}$
5. Get *index* = random (*RCL*)
6. Set $C \leftarrow C \cup RCL_{index}$
7. Update $S = \{ S_i - RCL_{index} / RCL_{index} \subset S_i , i=1\ldots m \}$
8. Get $T = \max \{ |S_i| / i=1\ldots m \}$
9. **End While**
10. Return {*C*}

Figura 2.9 Grasp for the set-covering problem.

Como se observa se determina el valor para *T* que indica cual es el mayor número de elementos por cada conjunto S_i ($i = 1\ldots m$) (punto 2). Luego se construye un conjunto *RCL* "restricted candidate list" (denominado "lista de candidatos") formado por los de mayor valor "$\alpha * T$" (punto 4), y que luego se elegirá aleatoriamente uno de esa lista (punto 5). La solución *C* irá almacenando en cada iteración el valor encontrado en *RCL* (punto 6). Luego se quitará de *S* aquellos valores de *RCL* que ya están incluidos en *S* (punto 7). Finalmente se re-calcula el valor para *T* (punto 8)

- ✓ Si: *T* es mayor que 0, se repite nuevamente desde el (punto 4),
- ✓ Sino: sale del bucle y devuelve la solución *C* (punto 10) la cual es la mejor solución a la que se llega repitiendo varias veces la ejecución de este procedimiento.

El parámetro α sirve para controlar el grado de aleatoriedad del procedimiento.

- ✓ A mayor valor de α menor aleatoriedad.
- ✓ Si α = 1, en la *RCL* sólo estaría el mejor candidato.
- ✓ Si α = 0, estarían todos los candidatos (función aleatoria pura)

En implementaciones estándares de GRASP el parámetro α se determina de forma aleatoria. Sin embargo, existen diversas estrategias para elegir el valor de α, las mismas que solo mencionaremos ya que no es el propósito de su estudio en este capítulo:

- ✓ Seleccionar su valor aleatoriamente de acuerdo a una distribución uniforme de probabilidad discreta (caso general)
- ✓ Auto-ajustar su valor según la calidad de las soluciones recientes obtenidas (*GRASP reactivo*)
- ✓ Seleccionar su valor de una distribución no uniforme de probabilidad discreta decreciente (mayor probabilidad para los mejores valores)
- ✓ Fijar su valor a un número concreto (próximo a la elección miope pura)

Capítulo 3: Colonia de Hormigas y Metaheurística Golosa para la Generación Automática de Casos de Prueba

En el presente capítulo se presenta el uso de las metaheurísticas Colonia de Hormigas y Goloso, como alternativas para generar automáticamente casos de prueba para test en una GUI. El objetivo está en función a minimizar el número de *rutas parciales.* Una *ruta parcial* es la secuencia de eventos que actúan sobre una GUI. A una ruta parcial también la denominaremos como *Caso de Prueba.*

La automatización se logra a partir de dos elementos fundamentales: el grafo de eventos-GUI, que representa el comportamiento deseado de la interfaz, y del criterio de cobertura seleccionado para la generación de los casos de prueba.

En la primera sección describiremos el concepto de *metaheurística* el cual se seleccionó para nuestro tema de investigación (definiciones, ventajas y desventajas de los distintos métodos metaheurísticos, y la clasificación de las técnicas de optimización). Posteriormente se explicará la adaptación al algoritmo ACO dentro del marco de la metaheurística y los procedimientos que lo componen: definición, fórmulas generales, declaración de variables y el algoritmo adaptado. Finalmente, se muestra como el método Goloso se aplica para resolver el problema de la cobertura para cada una de las rutas parciales.

3.1 Metaheurísticas

La optimización en el sentido de encontrar la mejor solución, o al menos una solución lo suficientemente buena, para un problema es un campo de vital importancia en la vida real y en ingeniería. Constantemente estamos resolviendo pequeños problemas de optimización, como el camino más corto para ir de un lugar a otro, la organización de una agenda, etc. En general, estos problemas son lo suficientemente pequeños y podemos resolverlos sin ayuda adicional. Pero conforme se hacen más grandes y complejos, el uso de las computadoras para su resolución es inevitable.

Comenzaremos este capítulo dando una definición formal del concepto de optimización. Asumiendo el caso de la minimización, podemos definir un *problema de optimización* como sigue:

Definición (Problema de optimización): *Un problema de optimización se formaliza como un par* (S, f), *donde* $S \neq \emptyset$ *representa el espacio de soluciones (o de búsqueda) del problema, mientras que* f *es una función denominada* función objetivo *o* función fitness, *que se define como:*

$$f : S \rightarrow \mathrm{lR} .$$

Así, resolver un problema de optimización consiste en encontrar una solución, $i^* \in S$, *que satisfaga la siguiente desigualdad:*

$$f(i^*) \leq f(i), \qquad \forall\, i \in S .$$

Asumir el caso de maximización o minimización no restringe la generalidad de los resultados, puesto que se puede establecer una igualdad entre tipos de problemas de maximización y minimización de la siguiente forma:

$$\max\{f(i) | i \in S\} \equiv \min\{-f(i) | i \in S\} .$$

Debido a la gran importancia de los problemas de optimización, a lo largo de la historia de la Informática se han desarrollado múltiples métodos para tratar de resolverlos. Una clasificación muy simple de estos métodos se muestra en la Figura 3.1. Inicialmente, las técnicas las podemos clasificar en exactas (o enumerativas, exhaustivas, etc.) y aproximadas. Las técnicas exactas garantizan encontrar la solución óptima para cualquier instancia de cualquier problema en un tiempo acotado. El inconveniente de estos métodos es que el tiempo necesario para llevarlos a cabo, aunque acotado, crece exponencialmente con el tamaño del problema, ya que la mayoría de éstos son NP-duros. Esto supone en muchos de los casos que el uso de estas técnicas sea inviable, ya que se requiere mucho tiempo para la resolución del problema (posiblemente miles de años). Por lo tanto, los algoritmos aproximados para resolver estos problemas están recibiendo una atención cada vez mayor por parte de la comunidad internacional desde hace unas décadas. Estos métodos sacrifican la garantía de encontrar el óptimo a cambio de encontrar una "buena" solución en un tiempo razonable.

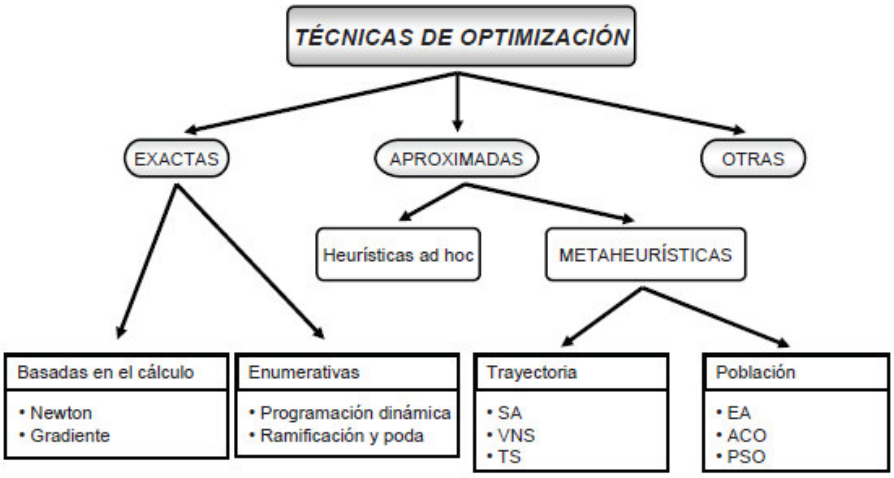


Figura 3.1 Clasificación de las técnicas de optimización.

Dentro de los algoritmos aproximados se pueden encontrar dos tipos: los heurísticos *ad hoc* y las metaheurísticas (en la que nos centraremos en este capítulo). Los heurísticos *ad hoc,* a su vez, pueden dividirse en *heurísticos constructivos* y *métodos de búsqueda local.*

Los heurísticos constructivos suelen ser los métodos más rápidos. Aunque en muchos casos encontrar un heurístico constructivo es relativamente fácil, las soluciones ofrecidas suelen ser de muy baja calidad. Encontrar métodos de esta clase que produzca buenas soluciones es muy difícil, ya que dependen mucho del problema, y para su planteamiento se debe tener un conocimiento muy extenso del mismo.

Los métodos de búsqueda local parten de una solución ya completa y, usando el concepto de *vecindario,* recorren parte del espacio de búsqueda hasta encontrar un *óptimo local.* El vecindario de una solución *s,* que denotamos con $N(s)$, es el conjunto de soluciones que se pueden construir a partir de *s* aplicando un operador específico de modificación (generalmente denominado *movimiento*). Un óptimo local es una solución mejor o igual que cualquier otra solución de su vecindario. Estos métodos, partiendo de una solución inicial, examinan su vecindario y se quedan con el mejor vecino, continuando el proceso hasta que encuentran un óptimo local. En muchos casos, la exploración completa del vecindario es inabordable y se siguen diversas estrategias, dando lugar a diferentes variaciones del esquema genérico. Según el operador de movimiento elegido, el vecindario cambia y el modo de explorar el espacio de búsqueda también, pudiendo simplificarse o complicarse el proceso de búsqueda.

Finalmente, en los años setenta surgió una nueva clase de algoritmos aproximados, cuya idea básica era combinar diferentes métodos heurísticos a un nivel más alto para conseguir una exploración del espacio de búsqueda de forma eficiente y efectiva. Estas técnicas se han denominado *metaheurísticas.* Este término fue introducido por primera vez por Glover [35]. Antes de que el término fuese aceptado completamente por la comunidad científica, estas técnicas eran denominadas heurísticas modernas [36]. Esta clase de algoritmos incluye colonias de hormigas, algoritmos evolutivos, búsqueda local iterada, enfriamiento simulado, y búsqueda tabú. Se pueden encontrar revisiones de metaheurísticas en [37, 38].

De las diferentes descripciones de metaheurísticas existentes, se pueden destacar ciertas propiedades fundamentales que caracterizan a este tipo de métodos:

- ✓ Las metaheurísticas son estrategias o plantillas generales que "guían" el proceso de búsqueda.
- ✓ El objetivo es una exploración eficiente del espacio de búsqueda para encontrar soluciones (casi) óptimas.
- ✓ Las metaheurísticas son algoritmos no exactos y generalmente son no deterministas.
- ✓ Pueden incorporar mecanismos para evitar regiones no prometedoras del espacio de búsqueda.

Resumiendo estos puntos, se puede acordar que una metaheurística es una estrategia de alto nivel que usa diferentes métodos para explorar el espacio de búsqueda. En otras palabras, una metaheurística es una plantilla general no determinista que debe ser rellenada con datos específicos del problema (representación de las soluciones, etc.), y que permiten abordar problemas con espacios de búsqueda de gran tamaño.

Dentro de las metaheurísticas podemos distinguir dos tipos de estrategias de búsqueda. Por un lado, tenemos las extensiones "inteligentes" de los métodos de búsqueda local (evitar de alguna forma los mínimos locales y moverse a otras regiones prometedoras del espacio de búsqueda). Y otro tipo de estrategia es el seguido por las colonias de hormigas (estos incorporan un componente de aprendizaje en el sentido de que, de forma implícita o explícita, intentan aprender la correlación entre las variables del problema para identificar regiones del espacio de búsqueda con soluciones de alta

calidad. Estos métodos realizan, en este sentido, un muestreo sesgado del espacio de búsqueda)

3.2 Aplicando ACO para obtener Rutas Parciales

La idea principal consiste en usar hormigas artificiales que simulan dicho comportamiento en un escenario también artificial: un grafo. Las hormigas artificiales se colocan en nodos iniciales de dicho grafo y lo recorren saltando de un nodo a otro con la meta de encontrar el camino más corto desde du nodo inicial hasta su nodo final u objetivo. Cada hormiga avanza de forma independiente a las demás, pero la decisión del siguiente nodo a visitar depende de ciertos valores numéricos asociados a los arcos o nodos del grafo. Estos valores modelan los *rastros de feromona* que depositan las hormigas reales cuando caminan. Las hormigas artificiales alteran (al igual que las hormigas reales) los rastros de feromona del camino trazado, de modo que el avance de una puede influir en el camino de otra.

En ese sentido se pretende utilizar ACO para obtener las rutas más largas. Para ello se partirá del grafo de secuencia de eventos-GUI, y se maximizará la cantidad de soluciones suficientemente buenas, tal vez próximas a la óptima; es decir, construir una amplia variedad de soluciones parciales con una capacidad de exploración mucho mayor.

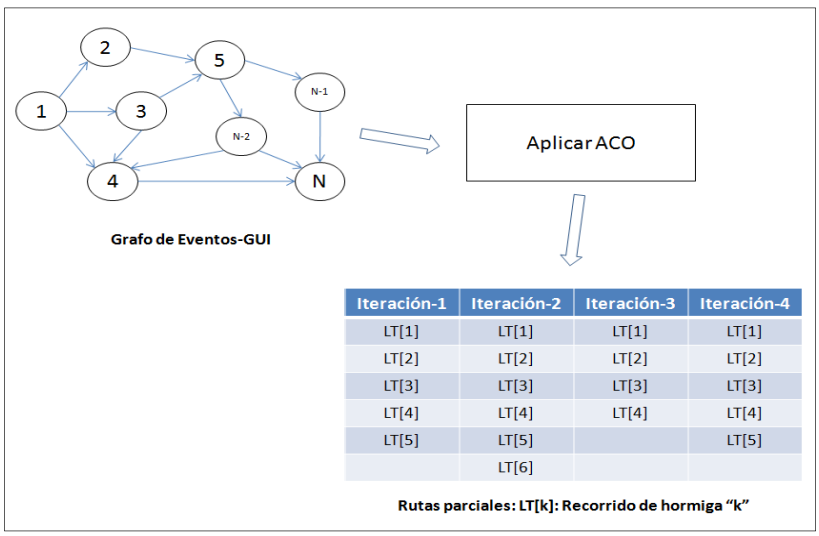


Figura 3.2 Aplicando ACO para hallar las rutas parciales.

De acuerdo al enfoque en esta primera fase, debemos representar el espacio de todas las posibles secuencias de eventos que actúan en una GUI a través de un grafo G.

Definición: G = (V, A), donde *V* es el conjunto de eventos-GUI, y *A* es el conjunto de interacciones entre cada uno de los eventos; este último es un conjunto de pares de la forma (*i, j*) tal que $i, j \in V$, tal que $i \neq j$.

Las características que persigue este grafo son:

- ✓ Contener uno o más eventos de arranque (nodos iniciales); es decir, cuales eventos serán punto de partida para el recorrido de las hormigas.
- ✓ La distancia entre uno y otro evento siempre es uno (1). Es decir, el costo es el mismo de ir de un evento a otro.
- ✓ La interacción entre los eventos es dirigida. Ej.: *evento i* → *evento j.*

Una vez definido el grafo, se deben establecer los parámetros α y β los cuales en el Capítulo 5 serán calibrados.

Finalmente, aplicamos el algoritmo ACO adaptado a nuestro modelo (ver Figura 3.3) para obtener las rutas parciales; esto es, cada una de las hormigas pueda identificar posibles "buenas áreas" en el espacio de búsqueda.

3.2.1 Justificación de ACO

Los modelos de ACO se pueden aplicar (y se han aplicado) a problemas con un número de nodos *n* de varios millares. En estos problemas, el grafo de construcción tiene un número de arcos del orden de n^2*;* es decir, varios millones de arcos, y la matriz de feromonas requiere para su almacenamiento varios megabytes de memoria. Sin embargo, el esquema descrito no es adecuado en problemas en los que el grafo de construcción tiene como mínimo del orden de 10^6 nodos (y 10^{12} arcos). Tampoco lo es cuando la cantidad de nodos no se conoce de antemano y los nodos y arcos del grafo de construcción se generan conforme avanza la búsqueda. Nosotros tratamos de resolver aquí este tipo de problemas. En efecto, el número de estados de un sistema concurrente es normalmente muy alto incluso en pequeños modelos.

Discutamos las cuestiones que impiden a los modelos existentes resolver este tipo de problemas. En primer lugar, en la fase de construcción, las hormigas de un ACO tradicional caminan hasta que construyen una solución completa. Una solución completa es un camino que termine en un estado de aceptación. Si permitimos a las hormigas caminar por el grafo sin repetir nodo hasta que encuentren un nodo objetivo,

podrían llegar a un nodo sin sucesores no visitados (un nodo sin salida para las hormigas). Incluso si encuentran un nodo objetivo, puede ser necesario mucho tiempo y memoria para construir una solución candidata, ya que los nodos objetivo pueden estar muy lejos del nodo inicial. A esto podemos añadir que ni siquiera tenemos asegurado que exista un estado de aceptación en el grafo, porque el modelo puede cumplir la propiedad que se intenta violar. Por tanto, no es viable, en general, trabajar con soluciones completas como hacen los actuales modelos. **Es necesario poder trabajar con soluciones parciales.** No estamos discutiendo un detalle de implementación, sino que tratamos con cuestiones que deben ser tenidas en cuenta en el diseño del nuevo algoritmo para poder trabajar con problemas que tienen un grafo de construcción de gran dimensión.

3.2.2 Características al usar ACO

- ✓ El enfoque ACO es particularmente adecuado para ser aplicado a problemas que acepten una *representación vía grafo,* necesario para imitar la búsqueda de un camino.
- ✓ La *representación del rastro de feromona* puede ser realizado a través de una matriz de número reales (τ) de N x N, donde N = número de eventos-GUI.
- ✓ El aspecto más importante en este algoritmo es la probabilidad de transición P_{ij} de una hormiga *k,* para moverse de *i* a *j.*
- ✓ *Factor Heurística local: 1 / d_{ij};* es decir, un valor inversamente proporcional a la distancia entre *(i, j)*
- ✓ ACO permite generar rutas parciales. Cada hormiga genera una solución parcial.

El algoritmo ACO se basa directamente en el comportamiento de las colonias de hormigas reales, para lo cual proponemos que:

HORMIGAS	**<u>Hormiga artificial</u> {k}** Resuelve el problema con un camino en el grafo.
LUGAR DE DECISION	**<u>Nodo</u>** Representan los Eventos de la GUI. **C = {C_1, C_2, C_3, …, C_n},** conjunto finito de eventos.

VIA	**<u>Aristas del grafo</u>** Representa cada paso que se puede dar en un flujo de Eventos de la GUI: **L = {L_{c1c2} / (C1, C2) ∈ C},** conexiones. C es un subconjunto del producto cartesiano C x C. Tiene asociado 2 tipos de información que guían su movimiento: Factor Heurístico y Feromona.
DECISION DE LA HORMIGA	**<u>Factor Heurístico</u>** Mide la preferencia de que el arco sea seleccionado, y a su vez no pueda ser modificada en la ejecución del algoritmo. El resultado (calculado en cada arco) de la función heurística, se realiza mediante una función de probabilidad.
FEROMONA	**<u>Datos de feromona artificial</u>** Es la cantidad de feromona depositado en un arco, el cual es modificado por las hormigas que ya pasaron por ese arco.
HORMIGUERO	**Nodo inicial del grafo**
COMIDA	**Nodo destino del grafo**
ECOSISTEMA DE LAS HORMIGAS	**Grafo**
COLONIA	**Colonia artificial**

Tabla 3.1 Aproximación de ACO al modelo adaptado.

3.2.3 Fórmulas Generales para ACO

a) Matriz de distancia (d_{ij})

$$d_{ij} = \begin{bmatrix} d[1][1] & d[1][2] & d[1][3] \\ d[2][1] & d[2][2] & d[2][3] \\ d[3][1] & d[3][2] & d[3][3] \end{bmatrix} \quad ; \quad \begin{cases} 1 & , tal\ que\ i \rightarrow j \\ 0 & , en\ otro\ caso \end{cases}$$

b) Visibilidad (η_{ij})

$$\eta_{ij} = \begin{bmatrix} \eta[1][1] & \eta[1][2] & \eta[1][3] \\ \eta[2][1] & \eta[2][2] & \eta[2][3] \\ \eta[3][1] & \eta[3][2] & \eta[3][3] \end{bmatrix} = [\ 1 / d_{ij}]; \begin{cases} 1\ , & tal\ que\ \mathrm{d}ij\ >\ 0 \\ 0\ , & en\ otro\ caso \end{cases}$$

c) Cantidad de feromona depositada ($\Delta\tau_{ij}$)

$$\Delta\tau_{ij} = \sum_{i,j}^{m} \underbrace{\Delta\tau_{ij}}$$

Cantidad de feromona depositada en (i, j) para la hormiga k

d) Matriz de feromonas (τ_{ij}) → *Intensidad del rastro de las feromonas*

$$\tau_{ij} = \begin{bmatrix} \tau[1][1] & \tau[1][2] & \tau[1][3] \\ \tau[2][1] & \tau[2][2] & \tau[2][3] \\ \tau[3][1] & \tau[3][2] & \tau[3][3] \end{bmatrix} = \underbrace{[\rho.\tau_{ij} + \Delta\tau_{ij}\ ]}_{Actualización}$$

ρ → Coeficiente de persistencia de las feromonas.

e) Lista tabú (*LT*), *contiene los nodos que la hormiga ha visitado.*

f) Función de probabilidad (*P*), *la hormiga elige la próxima ciudad a visitar, la cual está en función de la distancia y la cantidad de feromona que hay en (i, j)*

$$P_{ij} = \left[[\tau_{ij}]^{\alpha} * [\eta_{ij}]^{\beta} \Big/ \sum_{j \notin LT} [\tau_{ij}]^{\alpha} * [\eta_{ij}]^{\beta} \right]; con\ j \notin LT$$

α → *importancia relativa del sendero de feromonas*

β → *importancia relativa de la distancia entre (i, j)*

g) L_k → *longitud del recorrido completo realizado por la hormiga k*

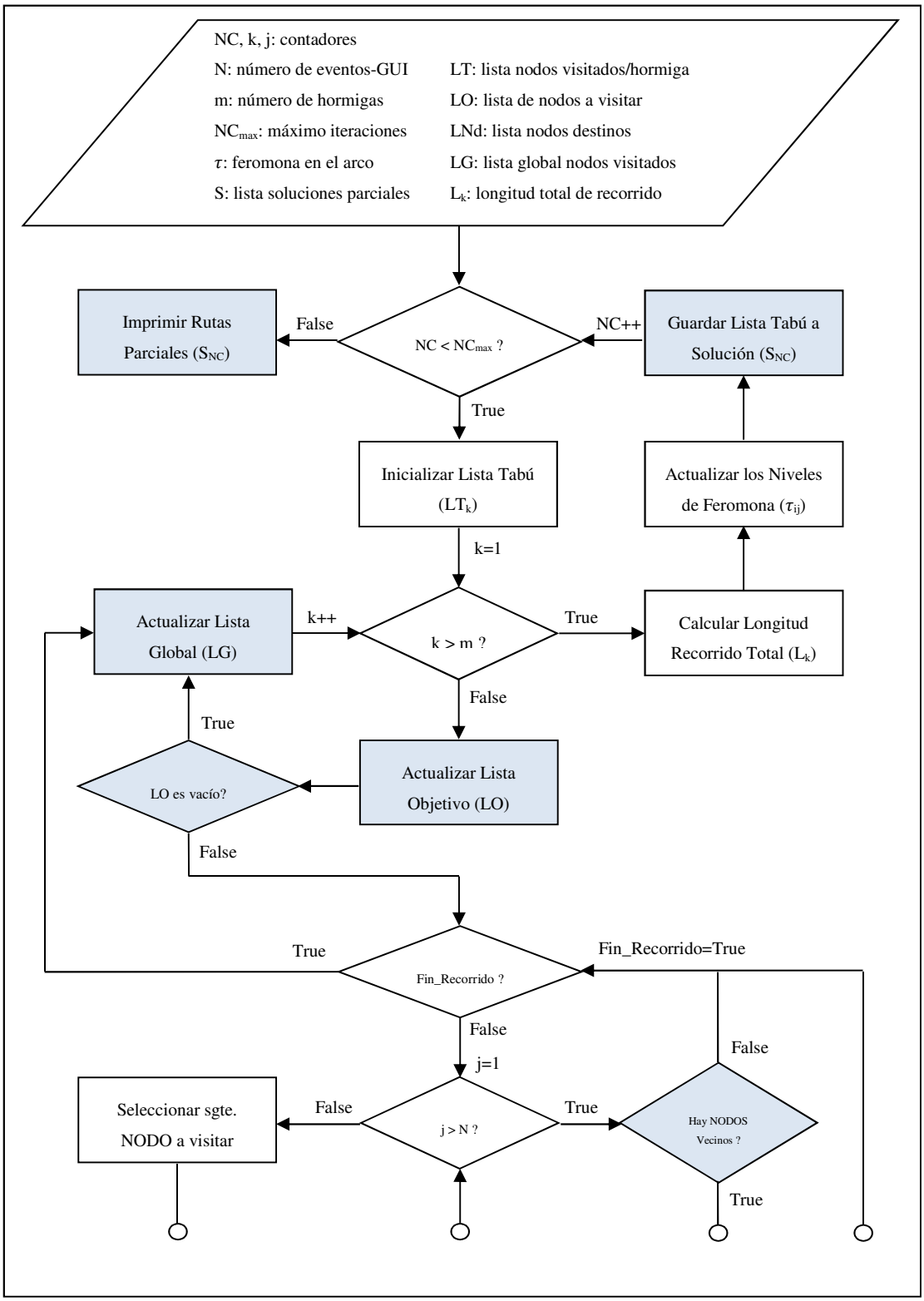
h) Cantidad de feromona que se deposita en cada arco:

$\Delta\tau_{ij} = Q / L_k$ *; si la hormiga "k" camina por el arco (i, j)*

Q → constante

3.2.4 Adaptación del Algoritmo ACO

El algoritmo de la Figura 3.3, muestra ciertas adaptaciones realizadas al algoritmo original ACO, donde las figuras sombreadas representan los procesos adicionales que se han utilizado:



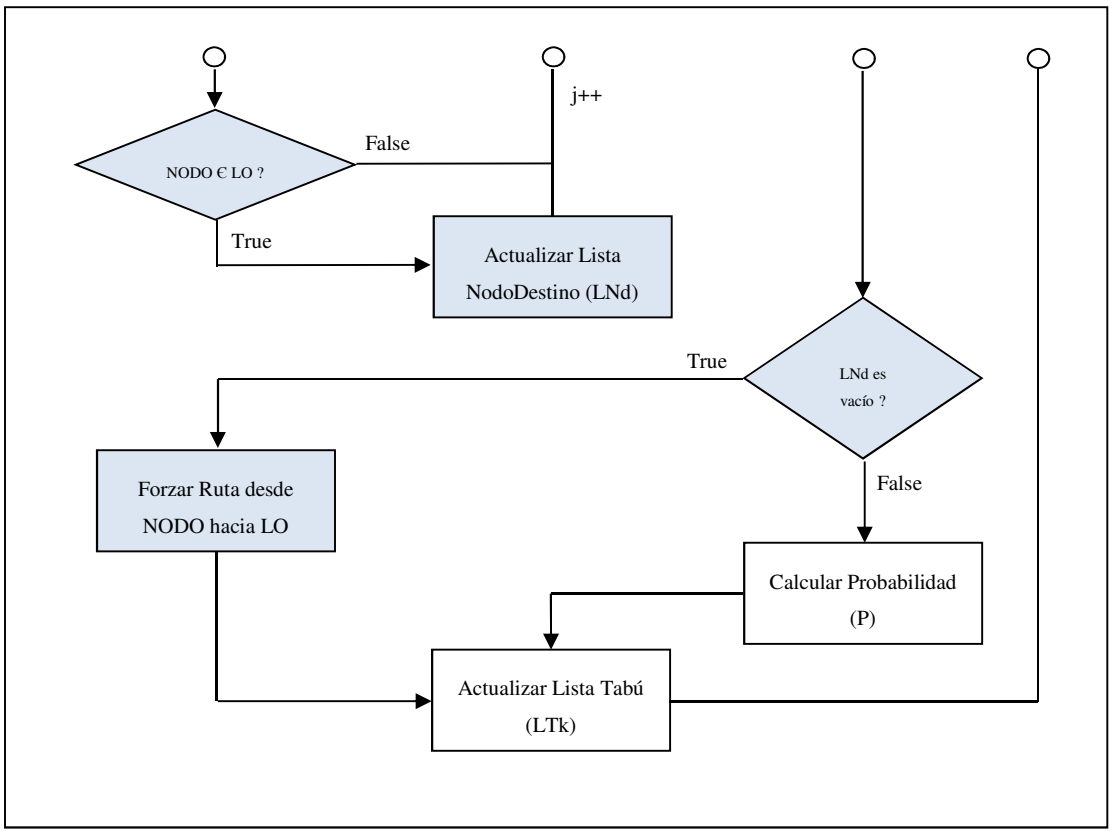


Figura 3.3 Algoritmo ACO aplicado a nuestro modelo.

A continuación la explicación de los procesos adicionales:

a) Actualizar Lista Objetivo (LO): Al inicio de cada iteración de la hormiga k, se agrega a ésta lista aquellos nodos no visitados (diferente del último nodo desde donde la hormiga k iniciará el recorrido). El objetivo principal de esta lista es indicar a las hormigas de los nodos que faltan visitar.

b) LO es vacío?: Permite definir un criterio de parada (STOP). Indica a las demás hormigas que ya no existen más nodos por visitar.

c) Actualizar Lista Global (LG): Se agregan a ésta lista los nodos visitados por cada una de las hormigas al completar su recorrido. El uso de ésta lista NO RESTRINGE la visita de posteriores hormigas que necesitan pasar obligatoriamente por éstos nodos.

d) NODO Є LO?: Permite saber si el siguiente nodo seleccionado está dentro de la lista de nodos a ser visitado por la hormiga (LO). De ser afirmativa la respuesta, se guardará los datos del arco elegido.

e) Actualizar Lista NodoDestino (LNd): Permite almacenar los datos relacionados al arco elegido, tales como: cantidad de feromona e información heurística. Esto será utilizado para calcular la probabilidad de elegir uno u otro nodo.

f) Hay NODOS Vecinos?: Permite saber si la hormiga encontró algún otro nodo para avanzar en su trayecto. De ser negativa la respuesta, se finaliza el recorrido de la hormiga *(Fin_Recorrido=True*), ya que supone que no existen más nodos por visitar. Continúa con la(s) siguiente(s) hormiga(s).

g) LNd es vacío?: Significa que la hormiga tendrá que 'pasar' nuevamente por aquellos nodos ya visitados por hormigas previas. Caso contrario, se calculará la probabilidad de visitar los posibles nodos.

h) Forzar Ruta desde NODO hacia LO: El objetivo que persigue este proceso es 'forzar' a una hormiga a pasar por aquellos nodos ya visitados por hormigas previas, ya que más adelante existen nodos por visitar (pertenecen a la lista objetivo). Para ello se utilizará el criterio de cobertura por longitud, el cual consiste en pasar (visitar) por nodos ya visitados siempre teniendo en cuenta: cubrir la mayor cantidad de *nodos objetivo* al momento de hallar la ruta parcial de la hormiga.

i) Guardar Lista Tabú a Solución (S): Guardar en una lista S los recorridos de las hormigas por iteración.

j) Imprimir Rutas Parciales: Mostrar para cada iteración, las rutas parciales obtenidas por cada una de las hormigas.

3.2.5 Declaración de Variables para ACO Adaptado y Greedy

Símbolo	Nombre	Descripción
d_{ij}	Arco (Distancia)	Conexión entre un origen y destino. Ej.: nodo 'i' → nodo 'j'. Siempre es 1.
i, j	Contadores	Permite recorrer un arreglo y/o matriz.
C	Contador	Permite recorrer una lista o arreglo.
NC	Contador de iteraciones	Permite realizar tantas iteraciones, hasta llegar al máximo posible: NC_{max}
k	Hormiga	Representa la hormiga.
m	Número de hormigas	Las 'm' hormigas se ubicarán aleatoriamente sobre cada nodo perteneciente a una *lista inicio.* *La relación entre m y N, será de 1 a 2.*
N	Número de nodos	Es el número total de nodos o eventos-GUI.

		La relación entre m y N, será de 1 a 2.
α	Peso de la feromona	Estima la importancia relativa de uso del sendero en función de la feromona depositada.
β	Peso del factor heurístico	Cercanía en términos de visibilidad.
ρ	Tasa de Evaporación	Se usa como un mecanismo que evita el estancamiento en la búsqueda, y permite que las hormigas busquen y exploren nuevas regiones del espacio. *Valor: [0 - 1]*
Q	Constante	Cantidad de feromona que se repartirá al final de cada ciclo (iteración), tasada por la distancia total del recorrido.
η_{ij}	Visibilidad	Matriz que representa la información heurística; es decir, el conocimiento previo. Se calcula como la inversa de la distancia: $\eta_{ij} = 1 / d_{ij}$
τ_{ij}	Feromona en el arco	Matriz que indica que tan prometedor es el nodo 'j' visto desde el nodo 'i' en términos de feromona, que a fin de cuentas depende de que tantas hormigas lo hayan usado. Para el primer ciclo (iteración) es un valor aleatorio y único para cada tramo <i, j>
NCmax	Número máximo de ciclos (iteraciones)	En cada iteración se obtendrán para cada una de las hormigas, un conjunto de secuencias de nodos.
LI	Lista Inicio	Lista que contiene los nodos sobre los cuales se establecerán las hormigas para empezar su recorrido. De esta lista se elegirá aleatoriamente el nodo sobre el cual la hormiga *k=1, 2, …, m* empezará.
LT_k	Lista Tabú	Lista utilizada por cada hormiga para llevar un registro de los nodos visitados, y que obliguen a la hormiga a no repetir los nodos que ya visitó; ésta lista se actualiza después de cada paso de tiempo, adicionando el nuevo

		nodo visitado. Se establecerá en la lista tabú 'LT_k' como primer elemento, un nodo aleatorio obtenido de la lista inicio (LI)
LO	Lista Objetivo	Lista usada para llevar un registro de los nodos NO VISITADOS, y que además sean diferentes del último nodo desde donde la hormiga *'k'* iniciará el recorrido: LT_k.lastItem El objetivo de ésta lista es indicar a las hormigas (*k=1, 2, …, m*) los nodos que faltan por visitar. Esta lista se inicializa a vacío, cada vez que una hormiga 'k' inicia un nuevo recorrido. Un criterio de parada (STOP) de una hormiga 'k', se dá cuando ésta lista es vacía.
LG	Lista Global	Lista usada para llevar un registro de los nodos VISITADOS por las hormigas: *k=1, 2, …, m.* Restringe: ya que sólo se consideran los nodos sin repetirse. No Restringe: cuando se requiere pasar por un nodo ya visitado de todas maneras, y la única forma es no restringiendo su visita por dicho nodo. Esta lista se inicializa a vacío, en cada ciclo (iteración)
$\mathbf{L_k}$	Longitud total del recorrido	Para cada LT_k, *k=1,2, …, m,* se calcula la longitud total recorrida, esto con el propósito de calcular el incremento de feromona $\Delta\boldsymbol{\tau}$**,** para LT_k.
$\mathbf{S_{NC}}$	Lista Solución	Lista que contiene las listas tabú de c/hormiga, en la iteración NC. S_{NC} = $\mathbf{\{LT_k, k = 1, 2, \ldots, m\}}$**,**
indexAleatorio	Posición aleatoria de la *lista inicio*	Permite obtener un número aleatorio desde *1* hasta la longitud de la *lista inicio;* éste número aleatorio representará la posición con el cual se procederá a obtener el valor de la *lista inicio.*

flagFin	Flag control de recorrido	Permite saber si la hormiga ya no puede recorrer más, esto debido a que se quedó estancada. Es decir, si LO es vacío. Su valor por defecto es Falso.
sumNumerador	Acumulado de la sumatoria para calcular las probabilidades	Sumatoria de $(\boldsymbol{\tau}[i][j]^{\alpha} * \eta[i][j]^{\beta})$ para cada arco desde 'i' hacia 'j'; donde 'j' no existe en *lista tabú,* y además pertenece a *lista objetivo.*
flagExisteVecino	Flag de control de vecindad	Permite saber si en pleno recorrido, la hormiga (estando en el nodo 'i') tiene por lo menos un vecino disponible (nodo 'j')
P	Arreglo de Probabilidades	Contiene una lista de probabilidades, de los cuales se seleccionará uno al azar.
indexProbab	Posición aleatoria de la lista de probabilidades	Este valor es el índice que se retorna cuando se selecciona un valor probabilidad de P*[ ],* y luego es utilizado para poder obtener el valor del nodo destino (de la lista arrayDestino)
arrayNumerador	Arreglo de Numeradores	Contiene una lista de numeradores $(\boldsymbol{\tau} ij^{\alpha} * \eta ij^{\beta})$, los cuales se utilizan para hallar la probabilidad que le corresponde a cada tramo <i,j>
arrayDestino	Arreglo de Destinos	Contiene una lista de nodos destino 'j' que están en LO, y de los cuales uno es elegido cuando se selecciona el valor al azar.
sumDistancia	Acumulado de la sumatoria para calcular la longitud del recorrido total	Permite hallar la longitud del recorrido de todas las hormigas. Su valor empieza en 0 en cada nueva iteración.
$\mathbf{T}_k$	Lista de Nodos Permanentes	Lista de nodos visitados por la hormiga "k"
$\mathbf{V}_k$	Lista de Distancias Acumuladas	Lista que contiene las distancias acumuladas que permiten ir del nodo "i" al nodo "j". Es decir: $d_{ij} > 0$

V_path_k	Lista de Rutas Propia	Lista que contiene la ruta para ir del nodo "i" al nodo "j". Es decir: $d_{ij} > 0$
MEN	Variable	Calcular la menor distancia acumulada de aquellos nodos NO PERMANENTES, y aquellos nodos que ya tienen distancia acumulada (diferente de -1)

Tabla 3.2 Variables aplicadas a nuestro modelo ACO.

G_{ij}	Matriz Greedy	Esta matriz contendrá tanto las rutas parciales (en las filas), así como los nodos (en las columnas)
GHeader	Lista de Nodo	Lista que contiene el ID de NODO que se seteará después que se actualice la matriz Greedy. Esto es la cabecera de la matriz G (primera fila)
S2	Lista Rutas Parciales (sin repetirse)	Lista que contendrá cada una de las rutas parciales (sin repetirse) obtenidos por ACO, donde los nodos están separados por coma.
arrayColsRemove	Arreglo de columnas a quitar de la matriz Greedy	Contiene las columnas (nodos) que ya NO se considerarán en la matriz Greedy en c/iteración.
arrayNodos	Arreglo de nodos	Contiene la lista de nodos por cada ruta parcial a escribir en la matriz Greedy.
flagExisteX	Flag de control de intersección	Flag booleano que permite verificar si una fila ($Hormiga_w$) intersecta a una columna ($Nodo_z$). En la intersección se comprobará si es "1". *w = 1, 2, 3, .., Y* *z = 1, 2, 3, ... N*
H	Contador	Utilizado para recorrer todas las rutas parciales (que no se repiten) obtenidos por ACO.
Ns	Número de rutas parciales	Contiene la cantidad de rutas parciales (obtenidas por ACO), las cuales **no se repiten.**
countX	Variable	En c/iteración se encuentra la cantidad de nodos que tiene dicha ruta, para

		luego determinar cual tiene la mayor cantidad de nodos.
MAY	Variable	Almacenar la mayor cantidad de nodos que tiene una ruta parcial, la cual se va comparando con las demás rutas parciales.
FILA	Variable	Almacenar la ruta (secuencia de nodos) que tiene mayor cantidad de nodos.
posIMatrix	Posición fila	Variable que almacena la fila (un valor) sobre el cual se establecerá la intersección "1"
posJMatrix	Posición columna	Variable que almacena la columna (un valor) sobre el cual se establecerá la intersección "1"

Tabla 3.3 Variables aplicadas a Greedy.

3.2.6 Pseudocódigo para ACO Adaptado

a. Definir las condiciones iniciales. Matriz de: visibilidad (η) y feromona (τ)

```
1
2
 2.1
   2.1.1
       2.1.1.1
       2.1.1.2
   2.1.2
 2.2
3
```

$\tau_0 = 0.030$ *; para primer ciclo es valor aleatorio y único para cada arco (i, j)*

PARA i = 1 **HASTA** N

  PARA j = 1 **HASTA** N

    SI (d[i][j] > 0)

      $\eta[i][j] = \left[\frac{1}{d[i][j]}\right]$

      $\tau[i][j] = \tau_0$

    Fin SI

  Fin PARA

Fin PARA

b. Definir una lista de nodos iniciales posibles sobre los cuales se situaran a las hormigas de forma aleatoria, y no depende de otros factores debido a que no existe un rastro de feromona preferencial al principio. Cada hormiga de la colonia artificial cuenta con una lista tabú *"LT_k"* que usan para llevar registro de los

nodos visitados. La lista tabú LT_k tendrá como primer elemento el nodo de arranque de cada hormiga "k"

Lista Tabú en t=0, para cada hormiga *"k"*

```
4     NC = 1                                          ; inicializando contador de iteraciones
5     LT[ ] = Empty                                   ; actualizando Lista Tabú 'LT'
6     PARA k = 1 HASTA m
6.1       indexAleatorio = RANDOM(1, LI.length)       ; length: devuelve la cantidad de elementos
                                                        de la lista 'LI'
6.2       LT[k].insert (LI.item[indexAleatorio])      ; item[c]: devuelve el valor del elemento de la
                                                        lista 'LI' en la posición c
                                                      ; insert (elemento): inserta un elemento al
                                                        final de la lista 'LTk'
7     Fin PARA
```

c. Escoger el camino a recorrer: una hormiga *"k"* en el *nodo i* escoge a que *nodo j* moverse, de acuerdo a los nodos contenidos en: *LT[k], LO y LG.*

Función de probabilidad (P_k)

```
8     PARA k = 1 HASTA m
8.1       LO[ ] = Empty                               ; actualizando lista objetivo
8.2       PARA c = 1 HASTA N
8.2.1         SI ( c ∉ LG[ ] ) AND ( c <> LT[k].lastNodo ) ; lastNodo: devuelve el
                                                        último elemento de la lista
                                                        'LTk'
8.2.1.1           LO.insert (c)                       ; insert (c): inserta el elemento 'c' al final de
                                                        la lista 'LO'
8.2.2         Fin SI
8.3       Fin PARA
8.4
8.5       flagFin = False                             ; permite establecer el fin de un recorrido o no
8.6       sumNumerador = 0                            ; acumulado del numerador para probabilidad
8.7
```

```
8.8     SI (LO[ ] == Ø)                 ; criterio de (stop) si lista objetivo es vacía
8.8.1       flagFin = True
8.9     Fin SI
8.10    MIENTRAS (! flagFin)
8.10.1      i = LT[k].lastNodo          ; lastNodo: devuelve el último elemento de la
                                        lista 'LTk'. Desde aquí se recorren
                                        posibles vecinos.
8.10.2
8.10.3      flagExisteVecino = False    ; si desde 'i' puedo avanzar o no
8.10.4      indexProbab = 0             ; devuelve el índice de la probabilidad elegida
                                        al azar, y con ello se puede obtener el
                                        siguiente nodo (de la lista arregloDestino[])
8.10.5      arrayNumerador[ ] = Empty   ; usados en la probabilidad
8.10.6      arrayDestino[ ] = Empty     ; nodos destino usados en la probabilidad
8.10.7
8.10.8      P[ ] = Empty                ; lista de probabilidades de los cuales se elegirá
                                        uno al azar, y me permitirá obtener el
                                        siguiente nodo
8.10.9      PARA j = 1 HASTA N
8.10.9.1        SI ( j ∉ LT[k] ) AND (d[i][j] > 0 )  ; siguiente nodo 'j' NO está en
                                                     lista tabú? y arco (i, j) existe?
8.10.9.1.1          flagExisteVecino = True
8.10.9.1.2
8.10.9.1.3          SI (j ∈ LO[ ])      ; sgte. nodo 'j' pertenece a lista objetivo?
8.10.9.1.3.1            arrayNumerador.insert (τ[i][j]^α * η[i][j]^β)
8.10.9.1.3.2            arrayDestino.insert (j)
8.10.9.1.3.3            sumNumerador += τ[i][j]^α * η[i][j]^β
8.10.9.1.4          Fin SI
8.10.9.2        Fin SI
8.10.10     Fin PARA
8.10.11
8.10.12     SI (flagExisteVecino)       ; existe por lo menos un vecino
```

8.10.12.1

8.10.12.2 **SI** (arrayDestino[ ] == Ø) *; sfca. que desde nodo 'i' no se encontraron nodos vecinos 'j' que pertenezcan a lista objetivo*

8.10.12.2.1

8.10.12.2.2 *; actualizar LT[k] con la mayor cantidad de nodos objetivo*

8.10.12.2.3 arrayPathMayorCover = actualizaListaTabu(k, i, N)

8.10.12.2.4

8.10.12.2.5 *; finalmente guardar c/u de los nodos que conforman la ruta más corta, a su lista tabú*

8.10.12.2.6 **PARA** c = 1 **HASTA** arrayPathMayorCover.length

A **SI** (arrayPathMayorCover[c] <> "")

a.1 **SI** (arrayPathMayorCover[c] ∉ LT[k])

a.1.1 LT[k].insert (arrayPathMayorCover[c])

a.2 **Fin SI**

B **Fin SI**

8.10.12.2.7 **Fin PARA**

8.10.12.2.8

8.10.12.2.9 flagFin = True

8.10.12.3 **SINO**

8.10.12.3.1 *; calcular las probabilidades de visitar los posibles nodos 'j'*

8.10.12.3.2 **PARA** c = 1 **HASTA** arrayDestino.length

8.10.12.3.2.1 $P[c] = \left[{}^{arrayNumerador[c]} / _{sumNumerador} \right]$

8.10.12.3.3 **Fin PARA**

8.10.12.3.4

8.10.12.3.5 *; devolver el índice de la probabilidad elegida al azar*

8.10.12.3.6 indexProbab = selectProbabilidad(P[ ])

8.10.12.3.7

8.10.12.3.8 *; finalmente se guarda el nodo seleccionado a su lista tabú*

8.10.12.3.9 LT[k].insert (arrayDestino[indexProbab])

```
        8.10.12.4          Fin SI
      8.10.13            SINO
        8.10.13.1          flagFin = True
      8.10.14            Fin SI
    8.11               Fin MIENTRAS
    8.12
    8.13               ; actualizamos Lista Global "LG", insertando los nodos de LT[k] sin repetirse
    8.14               PARA c = 1 HASTA LT[k].length
      8.14.1             SI (LT[k].item(c) ∉ LG[ ])
        8.14.1.1           LG.insert (LT[k].item(c))
      8.14.2             Fin SI
    8.15               Fin PARA
  9                  Fin PARA
```

d. Al terminar sus ciclos se obtiene la longitud del viaje de cada hormiga. Cuando todas las hormigas han completado el tour, se actualiza el rastro de feromona adicionando a cada tramo usado la sumatoria de feromona que cada hormiga aporta en los tramos de su recorrido y que es inversamente proporcional a la longitud del recorrido total:

Incremento de Feromona ($\Delta\tau$)

```
10    ; calcular la longitud del recorrido total, para c/hormiga
11    PARA k = 1 HASTA m
  11.1    sumDistancia = 0
  11.2    PARA c = 1 HASTA LT[k].length – 1
    11.2.1      i = LT[k](c)
    11.2.2      j = LT[k](c+1)
    11.2.3      sumDistancia += d[i][j]
  11.3    Fin PARA
```

```
11.4
11.5      L[k] = sumDistancia
12      Fin PARA
13
14      ; calcular el incremento de feromona
15      PARA k = 1 HASTA m
15.1        PARA c = 1 HASTA LT[k].length – 1    ; recorrer los elementos (nodos) de lista tabú 'LTk'
15.1.1          i = LT[k](c)
15.1.2          j = LT[k](c+1)
15.1.3          Δτ[i][j] += Q / L[k]
15.2        Fin PARA
16      Fin PARA
```

e. Para calcular la Feromona que queda en el arco, se debe *evaporar* la feromona inicial $\tau_{ij}(t)$, afectándola por la constante de evaporación *"ρ"* y adicionarle la feromona que acabamos de calcular para cada arco.

Actualización de Feromona (actualización en línea a posteriori)

```
17      PARA i = 1 HASTA N
17.1        PARA j = 1 HASTA N
17.1.1          SI d[i][j] > 0
17.1.1.1            τ[i][j] = (1 – ρ) * τ[i][j] + Δτ[i][j]
17.1.2          Fin SI
17.2        Fin PARA
18      Fin PARA
```

f. Aumentar el contador de ciclos (*NC*) y guardar en una lista *"S"* los recorridos de las hormigas, por iteración.

```
19      SI NC < NCmax
```

```
19.1        NC++
19.2        PARA k = 1 HASTA m
19.2.1          SI (LT[k] <> Ø)
19.2.1.1            S[NC].insert (LT[k])
19.2.2          Fin SI
19.3        Fin PARA
19.4
19.5        IR A (5)
20      SINO
21          ; calcular la mejor solución de acuerdo a: menor número de hormigas y mayor
            cobertura de nodos... APLICAR GREEDY [punto 4.2]

            Ir a 30. (GREEDY)
22      Fin SI
23      END
```

g. Finalmente obtenemos una tabla la cual representa la(s) solución(es) parcial(es): "S" que nos devuelve ACO.

Iteración	Secuencias posibles de recorrido de las hormigas
1	$\| LT_1 \| LT_2 \| LT_3 \| LT_4 \| LT_5 \| LT_6 \|$
2	$\| LT_7 \| LT_8 \| LT_9 \| LT_{10} \| LT_{11} \| LT_{12} \| LT_{13} \|$
3	$\| LT_{14} \| LT_{15} \| LT_{16} \| LT_{17} \| LT_{18} \|$
4	$\| LT_{19} \| LT_{20} \| LT_{21} \| LT_{22} \| LT_{23} \| LT_{24} \|$

Tabla 3.4. Rutas parciales: recorrido de las hormigas

Explicación: Esta lista contiene las rutas parciales encontradas por las hormigas en cada iteración. Por ejemplo, nos muestra que en la iteración 1 con 6 hormigas se logró cubrir todos los nodos, para la iteración 2 con 7 hormigas, la iteración 3 con 5 hormigas, etc.

h. A continuación, describiremos las funciones utilizadas en nuestro pseudocódigo:

```
24      ; calcular la ruta desde el nodo 'i', para la hormiga 'k', hacia los nodos de la lista objetivo (el que cubra la mayor cantidad de nodos objetivo)

25      FUNCION[ ] actualizaListaTabu(k:int, i:int, N:int)

25.1        T[ ] = Empty                ; lista de nodos permanentes

25.2

25.3        ; pasar a nodo permanente aquellos desde donde se empieza a calcular la ruta más corta hacia los demás nodos

25.4        PARA j = 1 HASTA N

25.4.1          SI (j = i)

25.4.1.1            V[j] = 0            ; lista que contiene las distancias acumuladas para llegar al nodo

25.4.1.2            V_path[j] = i + “,”  ; lista que contiene la ruta para llegar al nodo

25.4.2          SINO

25.4.2.1            V[j] = -1

25.2.3          Fin SI

25.5        Fin PARA

25.6

25.7        flagFin = False

25.8        MIENTRAS (! flagFin)

25.8.1          T.insert(i)         ; insertar el elemento ‘i’ al final de la lista permanente ‘T’

25.8.2

25.8.3          PARA j = 1 HASTA N

25.8.3.1            SI (d[i][j] > 0)

A                       SI (V[j] = -1)

a.1                         V[j] = V[i] + d[i][j]

a.2                         V_path[j] += V_path[i] + j + “,”

B                       ELSE

b.1                         SI (V[i] + d) > V[j])

b.1.1                           V[j] = V[i] + d[i][j]

b.1.2                           V_path[j] += V_path[i] + j + “,”

b.2                         Fin SI

C                       Fin SI
```

```
        Fin SI
    Fin PARA

    MEN = 9999
    i = 0
    PARA j = 1 HASTA N
        flagExistePermanente = False

        ; verificar que 'j' no exista en lista permanente 'T'
        PARA c = 1 HASTA T.length
            SI (T[c] == j)
                flagExistePermanente = True
                Exit PARA
            Fin SI
        Fin PARA

        ; calcular la menor distancia acumulada de aquellos nodos NO
          PERMANENTES, y aquellos nodos que ya tienen distancia
          acumulada (diferente de -1)
        SI (!flagExistePermanente) AND (V[j] <> 1)
            SI (V[j] < MEN)
                MEN = V[j]
                i = j
                Exit FOR
            Fin SI
        Fin SI
    Fin PARA

    ; condición de fin (STOP)
    SI (i=0) flagFin = True
Fin MIENTRAS
```

```
25.11       MAY = 0
25.12       i = 0
25.13
25.14       ; para ir hacia c/u de los NODOS de lista objetivo, debe pasar por ciertos
            nodos, de éstos nodos el que cubra la mayor cantidad de nodos (que están en
            lista objetivo), ganará (nodo final de la hormiga)
25.15       PARA c = 1 HASTA LO.length
25.15.1         j = LO.item[c]          ; obtiene el nodo de la lista 'LO' en la posición 'c'
25.15.2
25.15.3         ; calculando cuantos nodos objetivo se cubre para llegar a 'j'
25.15.4         cantidadNodosObjetivo = devuelveNumeroNodosObjetivo(V_path(j))
25.15.5
25.15.6         SI (cantidadNodosObjetivo > MAY)
25.15.6.1           MAY = cantidadNodosObjetivo
25.15.6.2           i = j   ; almacenamos el nodo objetivo ganador (que cubre más nodos
                            objetivos)
25.15.7         Fin SI
25.16       Fin PARA
25.17
25.18       ; en 'V_path(i)' se encuentra la ruta. Devolver como arreglo
25.19       RETURN (SPLIT(V_path(i), ","))
26      Fin FUNCION
27
28      FUNCION selectProbabilidad(P[ ])
28.1        ; ordenando lista de probabilidades de menor a mayor
28.2        PARA x = 1 HASTA P.length
28.2.1          PARA y = x+1 HASTA P.length – 1
28.2.1.1            SI (P[y] < P[x])
28.2.1.1.1              tmpProb = P[y]
28.2.1.1.2              P[y] = P[x]
```

```
28.2.1.1.3          P[x] = P[tmpProb]
28.2.1.1.4
28.2.1.1.5          ; ordenando arrayNumerador…
28.2.1.1.6          tmpNum = arrayNumerador[y]
28.2.1.1.7          arrayNumerador[y] = arrayNumerador[x]
28.2.1.1.8          arrayNumerador[x] = tmpNum
28.2.1.1.9
28.2.1.1.10         ; ordenando arrayDestino…
28.2.1.1.11         tmpDest = arrayDestino[y]
28.2.1.1.12         arrayDestino[y] = arrayDestino[x]
28.2.1.1.13         arrayDestino[x] = tmpDest
28.2.1.2            Fin SI
28.2.2          Fin PARA
28.3        Fin PARA
28.4
28.5    ; calcular las probabilidades acumuladas
28.6    c = 1
28.7    MIENTRAS (c <= P.length)
28.7.1      SI (c == 1)
28.7.1.1            PAcumulada[c] = P[c]
28.7.2      ELSE
28.7.2.1            PAcumulada[c] = PAcumulada[c-1] + P[c]
28.7.3      Fin SI
28.7.4      c = c + 1
28.8    Fin MIENTRAS
28.9
28.10   ; calcular un número aleatorio (entre 0-1)
28.11   numeroRnd = Rnd()
28.12
```

```
28.13   ; recorrer PAcumulada para determinar en qué rango está el número aleatorio (0-1), y determinar su posición en el arreglo
28.14   PARA c = 1 HASTA PAcumulada.length
28.14.1     SI (numeroRnd <= PAcumulada[c])
28.14.1.1       ; la posición 'c' nos permitirá obtener el valor del sgte nodo: arrayDestino(c)
28.14.1.2       Exit PARA
28.14.2     Fin SI
28.15   Fin PARA
28.16
28.17   RETURN (c)
29  Fin FUNCION
```

i. Luego para resolver el problema de la cobertura para cada una de las rutas parciales, aplicamos el algoritmo Greedy.

3.3 Aplicando Greedy para resolver el Set Covering

Aplicar Greedy para obtener la mejor alternativa: Se desea cubrir la mayor cantidad de eventos-GUI, esto conllevará a obtener el mínimo de *rutas parciales*.

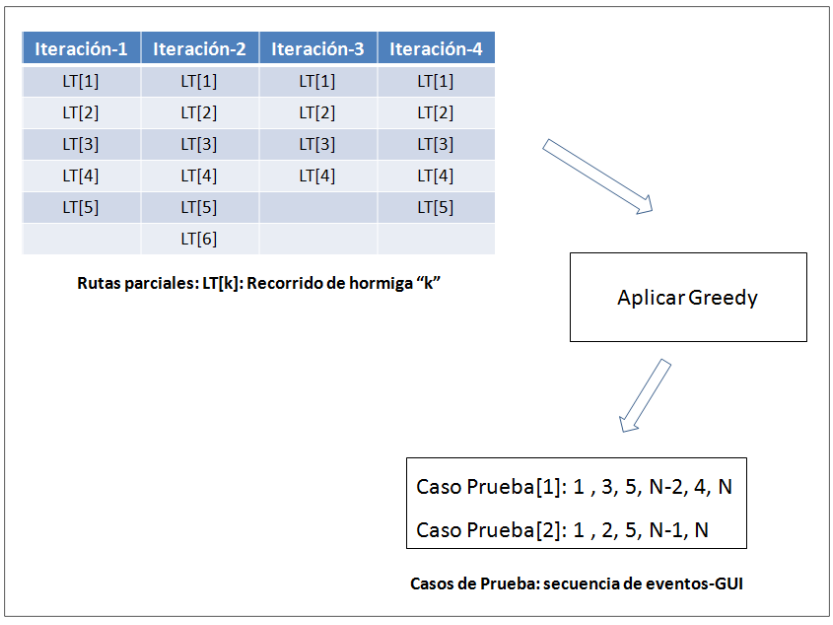


Figura 3.4 Aplicando Greedy para hallar los Casos de Prueba.

Tras las rutas parciales obtenidas con ACO, ahora el objetivo es cubrir la mayor cantidad de nodos a partir de las rutas parciales.

3.3.1 Algoritmo Greedy

Para resolver dicho problema se aplicará el algoritmo Greedy:

- ✓ Partimos del punto de que contamos con las rutas parciales: $LT_1 \rightarrow LT_2 \rightarrow \ldots \rightarrow LT_m$, encontradas por las hormigas: $Hormiga_k$ / $_{k=1, 2,..., m}$.

  Hormiga$_w$: LT_k = Nodo$_1$ → Nodo$_2$ → Nodo$_3$ → … → Nodo$_N$

  Donde:

  k = 1, 2, 3, ..., m

  w = 1, 2, 3, ..., Y. "Y" es el valor máximo en donde cada Hormiga$_w$ no se repitan (sean distintos)

- ✓ Plasmar dichas rutas en una matriz, la cual se denomina "matriz Greedy". La cardinalidad de la matriz es **(w*z)**

	Nodo$_1$	**Nodo$_2$**	**…**	**Nodo$_{z-1}$**	**Nodo$_z$**
Hormiga$_1$	1			1	
Hormiga$_2$		1		1	1
…					
Hormiga$_{w-1}$					1
Hormiga$_w$		1			1

Tabla 3.5 Matriz Greedy.

Donde:

*w = 1, 2, 3, ..., Y (rutas parciales correspondiente a las hormigas sin repetirse). Y = Es un valor que oscila entre 1 y m*NCmax*

z = 1, 2, …, N (cada nodo en grafo)

Explicación: cada fila representa las rutas parciales que ACO genera, las cuales están compuestas por una secuencia de nodos (representado en cada columna). La intersección de una fila Hormiga con una o más columnas Nodo, representa la secuencia de nodos $Nodo_z$ / *z=1, 2, …, N*) recorridos por la hormiga $Hormiga_w$ / *w=fila actual*

✓ Aplicamos Greedy utilizando la siguiente lógica:

a) Seleccionar la ruta (fila) con mayor cantidad de nodos que intersecta. Para el ejemplo de la Tabla 3.5, tenemos $Hormiga_2$ con 3 nodos. Si número de rutas llega a cero, entonces ir a d)
b) Quitar aquellos nodos que intersectan a la ruta (fila) seleccionada.
c) Almacenar la ruta seleccionada, y repetir desde a)
d) Imprimir las rutas que fueron almacenadas.
e) FIN.

✓ Finalmente, Greedy devuelve las siguientes rutas o casos de prueba (secuencias de nodos): **Caso Prueba$_1$: Nodo$_2$ → Nodo$_{N-1}$ → Nodo$_N$**

3.3.2 Pseudocódigo para Greedy

a. Obtenemos la matriz Greedy. Para las filas: se deben considerar las rutas parciales sin repetirse; para las columnas: los nodos.

Obteniendo la matriz Greedy

```
30    S2[ ] = Empty            ; S2 contendrá las secuencias de nodos 'S', sin repetirse
31    PARA c = 1 HASTA S.length
31.1      SI (S[c] ∉ S2[ ])
31.1.1        S2.insert (S[c])
31.2      Fin SI
32    Fin PARA
33
```

```
34    arrayColsRemove[ ] = Empty   ; inicializamos la lista
35    CALL setMatrizGreedy (S2.length, N)
```

b. Finalmente aplicamos el algoritmo Greedy.

Aplicando Greedy

```
36        MIENTRAS (G[ ][ ].rows > 1)
  36.1        MAY = 0                    ; se inicializa con un número pequeño
  36.2
  36.3        ; determinar la fila con mayor intersección
  36.4        PARA i = 1 HASTA G[ ][ ].rows
    36.4.1        countX = cantidadX (G, i)
    36.4.2        SI (countX > MAY)   ; si la cantidad es mayor de una variable 'MAY' que
                                        contiene el último mayor
      36.4.2.1        MAY = countX   ; asignamos a la variable mayor cantidad encontrada
      36.4.2.2        FILA = i              ; guardamos la fila que tiene la mayor cantidad
    36.4.3        Fin SI
  36.5        Fin PARA
  36.6
  36.7        ; determinar las columnas que hacen intersección con ésta FILA (la mayor).
              Luego se irán almacenando en un arreglo las columnas a quitar de la matriz
  36.8        PARA j = 1 HASTA G[ ][ ].cols-1
    36.8.1        SI ( G[FILA][j] == "1" )
      36.8.1.1        arrayColsRemove.insert (GHeader[j])
    36.8.2        Fin SI
  36.9        Fin PARA
  36.10
  36.11       CALL setMatrizGreedy (S2.length, N, FILA)
37        Fin MIENTRAS
```

c. A continuación, describiremos las funciones utilizadas en nuestro pseudocódigo:

```
38      ; establecer(actualizar) la matriz G en cada proceso/iteración
39      FUNCION setMatrizGreedy (Ns:int, N:int, [FILA:int=-1])
  39.1      ; Ns --> número de secuencias posibles de recorrido de las hormigas
  39.2      ; N --> número de nodos
  39.3
  39.4      G[ ][ ] = Empty          ; inicializar la matriz Greedy
  39.5
  39.6      posJMatrix = 1
  39.7      PARA j = 1 HASTA N
    39.7.1      SI (j ∉ arrayColsRemove[ ])
      39.7.1.1          GHeader[posJMatrix] = j  ; estableciendo el encabezado en la
                                                   matriz, sólo con aquellas columnas
                                                   válidas (NO pertenezcan a la lista de
                                                   columnas removidas)
      39.7.1.2          posJMatrix++
    39.7.2      Fin SI
  39.8      Fin PARA
  39.9
  39.10     posIMatrix = 0              ; contador de fila que se va agregando a la
                                          nueva matriz
  39.11     ; recorrer desde la Hormiga[1] hasta H[Ns]: Total de rutas parciales
  39.12     PARA H = 1 HASTA Ns
    39.12.1     SI (H <> FILA)       ; ir agregando filas <> a la FILA MAYOR
      39.12.1.1         G.addRow (H)
      39.12.1.2         posIMatrix += 1
      39.12.1.3
      39.12.1.4         ; obtener los nodos de la solución S2[H-1] a un arreglo. S2
                        contiene las rutas parciales sin repetirse. Ej.: S2[0]=1,2,3 /
                        S2[1]=2,4,5 / S2[2]=1,5,6
      39.12.1.5         arrayNodos = SPLIT(S2[H-1], ",")
```

```
39.12.1.6
39.12.1.7                flagExisteX = False
39.12.1.8                PARA c = 1 HASTA arrayNodos.length
A                            ; para la Hormiga[H], intersecto "1" con la columna
                             "posJMatrix" según corresponda con el nodo[c]
B                            posJMatrix = posicionHeaderColumnaMatrix(G,
                                         arrayNodos[c])
C                            SI (posJMatrix > 0)          ; sí encontró intersección
C.1                              flagExisteX = True
C.2                              G[posIMatrix][posJMatrix] = "1"
D                            Fin SI
39.12.1.9                Fin PARA
39.12.1.10
39.12.1.11               ; si no hay intersección para éste fila, se quita la fila
39.12.1.12               SI (! flagExisteX)
A                            G[ ][ ].rows = G[ ][ ].rows – 1
B                            posIMatrix = posIMatrix – 1
39.12.1.13               Fin SI
39.12.2              Fin SI
39.13            Fin PARA
39.14
39.15            ; como la matriz G se dimensionó hasta la columna N, se debe quitar
                 las últimas columnas las cuales corresponden a las COLUMNAS
                 REMOVIDAS.
39.16            SI (FILA <> -1)
39.16.1              G[ ][ ].cols -=  arrayColsRemove.length
39.17            Fin SI
40           Fin FUNCION
41
42           FUNCION cantidadX (M:FlexGrid, i:int)
42.1             cantidadX = 0
```

```
  42.2          PARA j = 1 HASTA M[ ][ ] – 1
    42.2.1        SI (M[i][j] = “X”)
      42.2.1.1      cantidadX++
    42.2.2        Fin SI
  42.3          Fin PARA
43              Fin FUNCION
44
45              FUNCION posicionHeaderColumnaMatrix (M:FlexGrid, j:int)
  45.1          posicionHeaderColumnaMatrix = 0
  45.2          PARA c = 1 HASTA M[ ][ ].cols-1
    45.2.1        SI (j == M[0][c])
      45.2.1.1      posicionHeaderColumnaMatrix = c
      45.2.1.2      Exit FUNCION
    45.2.2        Fin SI
  45.3          Fin PARA
46              Fin FUNCION
```

Capítulo 4: Diseño e Implementación del Sistema

Para el diseño e implementación de la solución se consideró la generación de un sistema visual que permita tanto la ejecución particular de instancias de prueba, como para la ejecución de los experimentos a los que fueron sujetos los algoritmos, sobre los que tratará el siguiente capítulo.

4.1 Diseño de la interfaz gráfica del Sistema

El diseño de la interfaz gráfica de la solución giró alrededor de dos ejes principales: la facilidad de uso para el usuario y la organización de la información de forma clara y sencilla. Debido a lo anterior, es posible identificar en la interfaz tres secciones bien definidas que se pueden observar en la Figura 4.1.

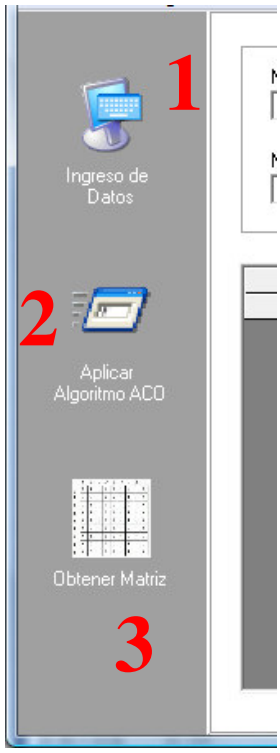


Figura 4.1 Secciones de la interfaz gráfica de la solución.

La primera de estas secciones (marcada con el número 1) presenta la configuración general del algoritmo ACO (número nodos, número hormigas, máximo iteraciones, alfa, beta, matriz de eventos-GUI, etc.). Esta sección es el input para la configuración de ACO.

La segunda sección (2) es de carácter informativo solamente. Muestra al usuario información de las rutas parciales que cada hormiga recorrió por cada iteración ejecutada hasta completar el máximo de iteraciones.

Finalmente la sección marcada con el número (3), cuenta con una tabla que muestra información asociada a la "matriz Greedy" como resultado de las rutas parciales obtenidas en la sección (2). Después el usuario puede ejecutar el algoritmo Goloso.

4.2 Consideraciones sobre el Ambiente de Desarrollo

4.2.1 Entorno de desarrollo utilizado

Para trabajar en entorno Windows se empleó el entorno de desarrollo *Microsoft Visual Studio 6.0 (Visual Basic 6.0),* compatible con las siguientes plataformas de sistema operativo:

- ✓ Microsoft Windows XP Professional SP3 (x86)
- ✓ Microsoft Windows Server 2003 R2 Standard Edition SP2 (x86)
- ✓ Microsoft Windows Vista Home Premium SP2 (x64)
- ✓ Microsoft Windows 7 Professional SP1 (x86)

El motivo de la elección de la herramienta radicó en su rapidez y sencillez para implementar la solución. Dado que se quiso desarrollar una interfase visual amigable y de fácil manejo por parte del usuario final, se prefirió un entorno visual a uno DOS como lo podría ser C++.

Con Visual Basic 6.0 se trató de aprovechar las características RAD (*del inglés Rapid Application Development*) o desarrollo rápido de un sistema en entorno Windows.

4.2.2 Dispositivos de almacenamiento de datos

Se requieren ciertos dispositivos que almacenen la información básica para su ejecución. El sistema implementado empleará *archivos de texto (*.ini)* para el trabajo con los datos.

De la misma forma, se podría efectuar las modificaciones del caso para que el sistema cargue a través de manejadores de base de datos (Microsoft Access, tablas de SQL Server/Express, etc.), los datos necesarios: matriz de eventos-GUI, número nodos y de hormigas, valores para alfa y beta, tasa, número máximo de iteraciones, etc.

4.2.3 Alcances y limitaciones del sistema implementado

- ✓ El sistema presenta un tope máximo de iteraciones para generar las rutas parciales de ACO. Si bien este parámetro es parametrizable, si dentro de esos ciclos no encontrara más rutas parciales, se continuará así con la siguiente iteración hasta completar el máximo configurado.
- ✓ El ingreso de la matriz de eventos-GUI se realizará manualmente.
- ✓ La cantidad máxima de eventos-GUI (nodos) posible, es de 99.
- ✓ La cantidad máxima de hormigas posible, es de 99.
- ✓ Antes de poder aplicar el primer algoritmo (ACO adaptado), se debe indicar al menos un nodo inicial para la partida.

4.3 Módulos del Sistema

En la siguiente figura se presenta la estructura general del sistema:

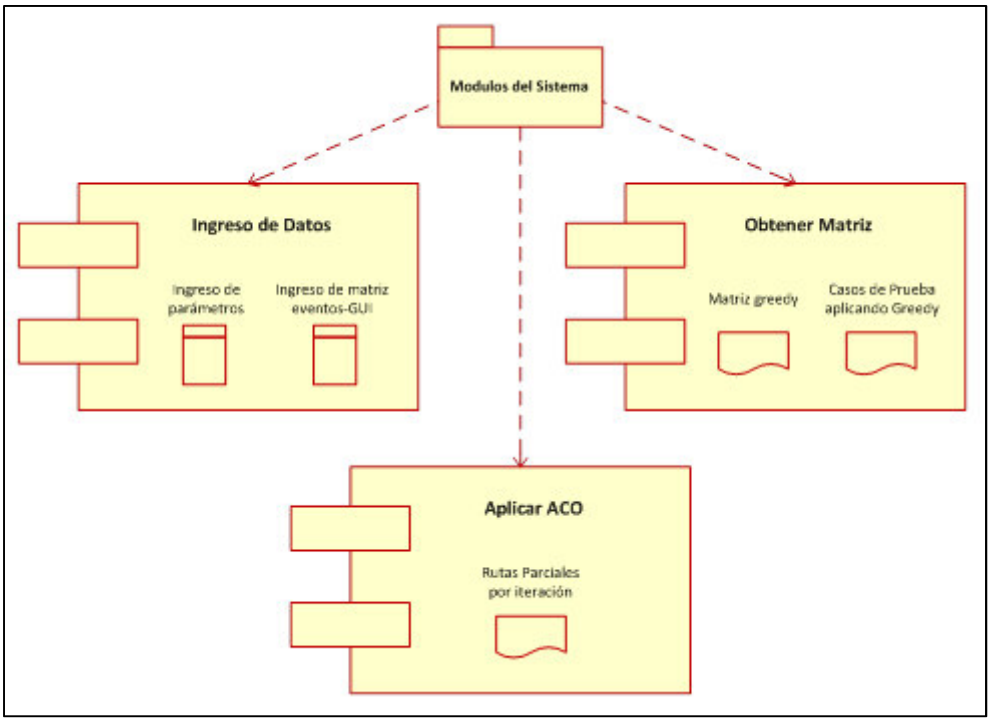


Figura 4.2 Estructura del sistema.

La estructura muestra una división en donde se están separando los procesos de lectura (captura de datos), ejecución de los algoritmos y de los reportes. Los reportes  mostrarán la información producto de la ejecución de los algoritmos.

4.3.1 Ingreso de Datos

El módulo de ingreso de datos permite la configuración de la siguiente información:

- ✓ Ingreso de Parámetros: **(1)**
  - Número nodos (N)
  - Número hormigas (m)
  - alfa (α)
  - Beta (β)
  - Tasa evaporación (ρ)
  - Constante (Q)
  - Máx. iteraciones (NC_{max})
- ✓ Ingreso de Matriz de eventos-GUI: **(2)**
  - Grafo de eventos que ocurren sobre una Interfaz Gráfica de Usuario (GUI)
- ✓ Selección de Nodo Inicial: **(3)**
  - Se debe seleccionar por lo menos un nodo inicial (de partida)

En la siguiente figura se ve la pantalla de ingreso de datos:

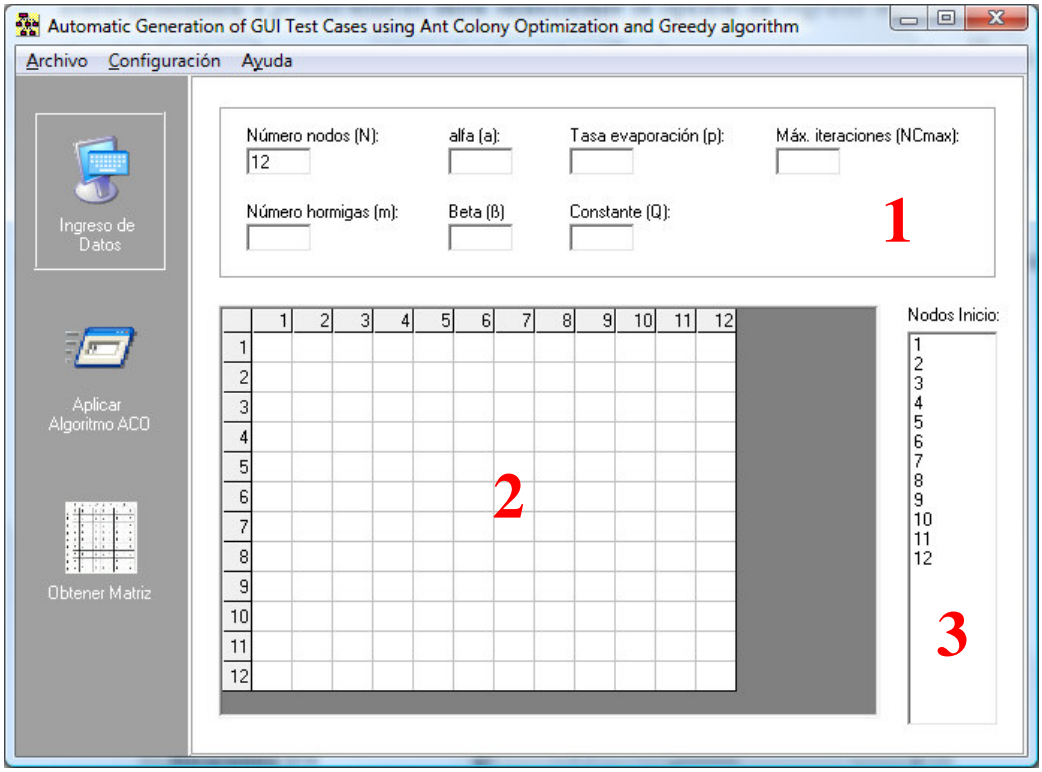


Figura 4.3 Pantalla de ingreso de datos.

4.3.1.1 Ingreso de datos a través de archivo de configuración

Este ingreso automatizado permite indicarle al sistema la ruta donde se encuentra el archivo (extensión *ini*) que contiene la información de los parámetros, así como de la matriz de eventos-GUI.

Para ello el usuario debe seleccionar en el menú del sistema la opción Archivo / Abrir, luego una vez encontrado el archivo *ini,* y dado clic en el botón Abrir, la información contenida en él se mostrará automáticamente sobre la pantalla de ingreso de datos.

El formato que cumple el archivo **.ini,* se puede observar en la siguiente figura:

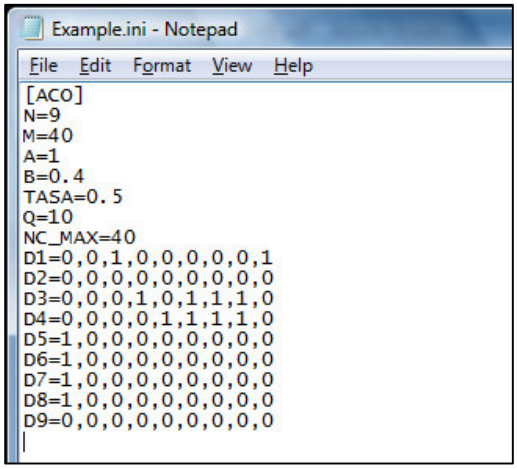


Figura 4.4 Formato del archivo de configuración (.ini*)**

4.3.1.2 Ingreso manual de datos

La otra posibilidad que ofrece el sistema es el ingreso manual de los parámetros y matriz eventos-GUI. A continuación una breve explicación de los parámetros más relevantes:

- ✓ Número nodos (N): Después de ingresar este parámetro, la matriz de eventos-GUI se redimensionará automáticamente a: (N x N)
- ✓ Matriz eventos-GUI: Para poder asociar un nodo origen (i) a un nodo destino (j): $i \rightarrow j$, existe 2 posibilidades:
  - Haciendo clic sobre la celda intersección (i, j), o
  - Digitando el valor "1" sobre la celda intersección.

  Para ambas posibilidades, se puede quitar la asociación del nodo origen a un nodo destino, mediante [backspace] o [Supr]
- ✓ Seleccionar el(los) Nodo(s) inicial(es), situado en la parte derecha de la pantalla.

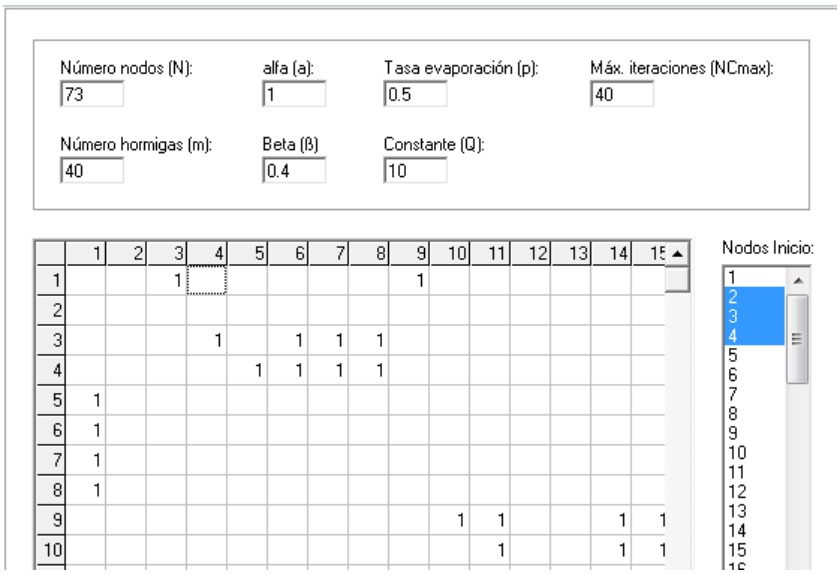


Figura 4.5 Ingreso manual de datos.

4.3.2 Aplicar ACO

Una vez terminado el ingreso de los datos y realizada las validaciones internas necesarias, se procede a la ejecución del primer algoritmo: **"Ant Colony Optimization (ACO adaptado)"**

El resultado obtenido son las *Rutas Parciales de cada hormiga* por cada iteración, hasta completar el máximo iteraciones:

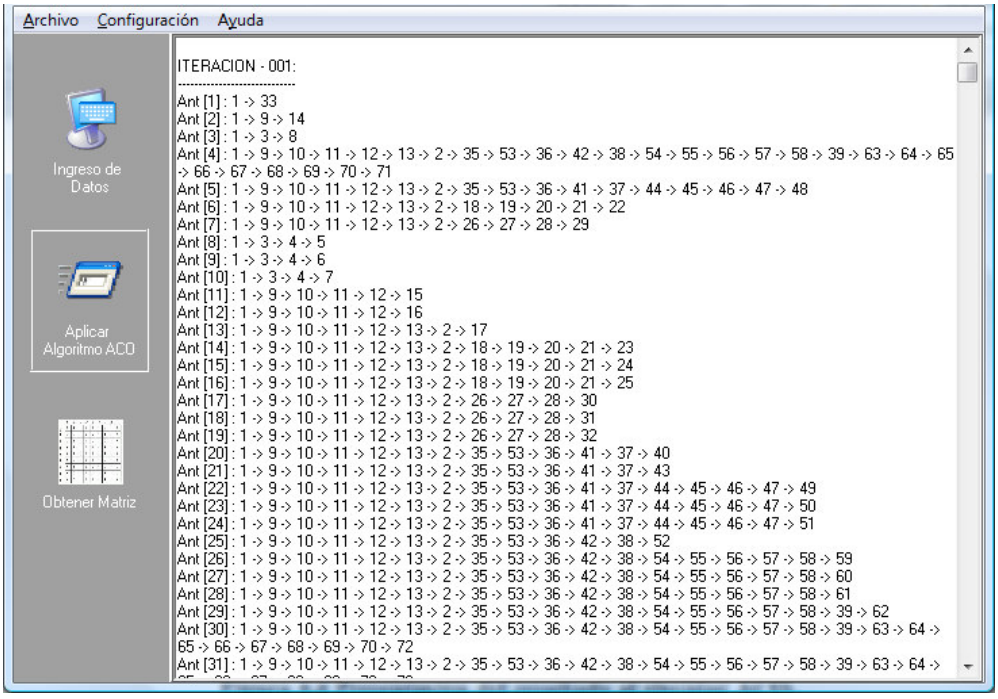


Figura 4.6 Presentación del resultado al ejecutar ACO.

4.3.3 Obtener Matriz

El sistema presenta la "Matriz Greedy" sin repetirse. Luego, se debe marcar el segundo algoritmo a ejecutar: **"Aplicar Set Covering… con Greedy"**

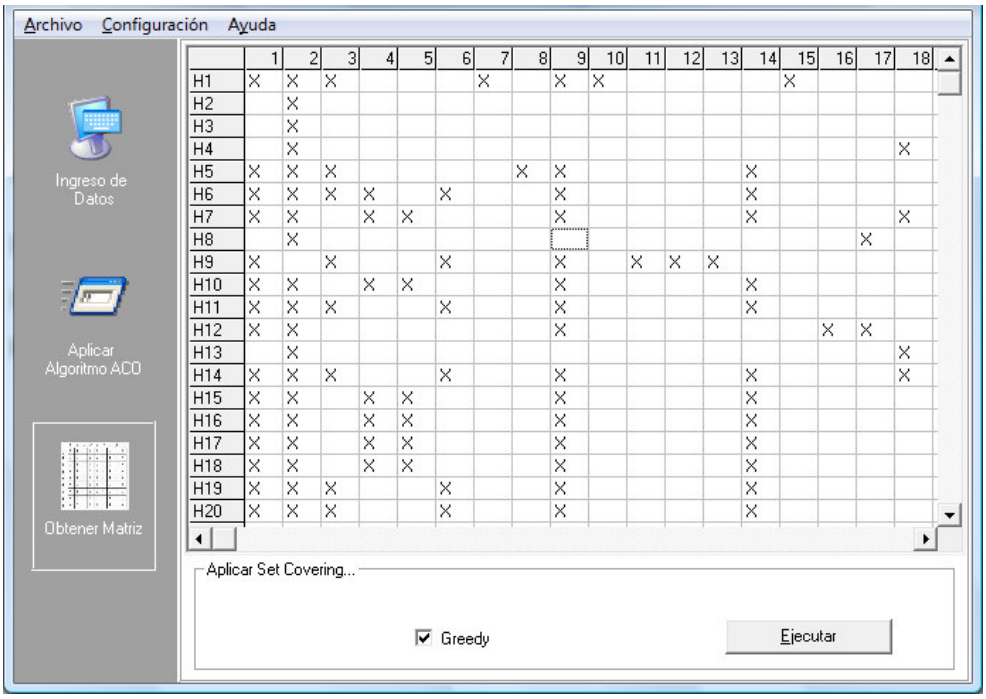


Figura 4.7 Presentación de la Matriz Greedy.

Finalmente, el resultado al hacer clic en el botón Ejecutar es:

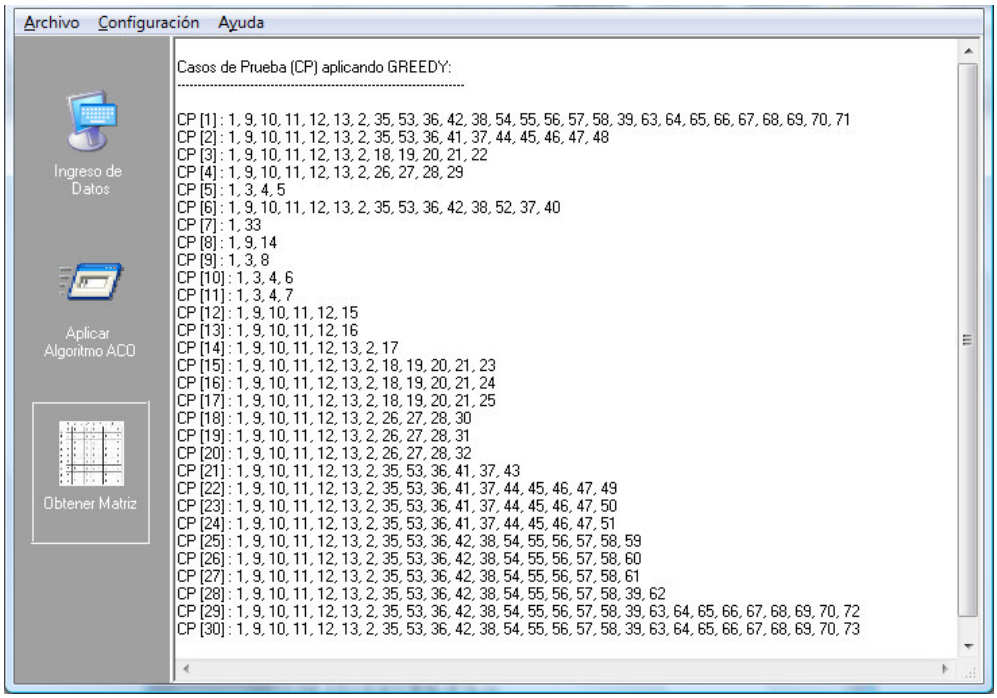


Figura 4.8 Presentación de los Casos de Prueba aplicando Greedy.

4.4 Requerimientos mínimos de hardware y software

A la luz de las pruebas, se necesita un equipo con las siguientes características para un funcionamiento adecuado del sistema implementado:

4.4.1 Configuración de hardware mínimo

- ✓ Procesador Intel o compatible.
- ✓ Velocidad: igual o superior a 2.00 GHz.
- ✓ Espacio libre en disco duro: 20.0 MB.
- ✓ Mouse, teclado.

4.4.2 Configuración de software mínimo

- ✓ Sistemas operativos: Windows XP, Vista, 7.
- ✓ Microsoft Visual Basic 6.0.

Capítulo 5: Experimentos con Caso de Estudio

En este capítulo se toman en cuenta las consideraciones realizadas durante los experimentos a los que fue sometida la aplicación (sistema implementado) con los algoritmos propuestos:

- ✓ Las configuraciones de hardware y de software en las que se llevaron a cabo las pruebas del sistema, el cual implementa los algoritmos propuestos.
- ✓ El desempeño del sistema implementado.
- ✓ El proceso de calibración de las variables de relajación ACO empleadas en los experimentos: valores de α y β.
- ✓ Los resultados en los experimentos: analizaremos la calidad de las soluciones Greedy.
- ✓ Análisis de la calidad de los algoritmos desde la perspectiva del desarrollo de proyectos de Ingeniería de Software.

5.1 Hardware y software empleados

La siguiente es la configuración de hardware y software del equipo empleado para la realización de las pruebas:

- OS Name: Microsoft® Windows Vista™ Home Premium.
- Version: 6.0.6002 Service Pack 2 Build 6002
- System Type: x64-based PC
- Processor: Intel(R) Core(TM)2 Duo CPU - T6400 @ 2.00GHz, 2000 Mhz, 2 Core(s), 2 Logical Processor(s)
- Memory RAM: 4.00 GB

5.2 Descripción del Ambiente del Caso de Estudio

Vamos a generar los casos de prueba de los principales menús del *WordPad 6.0: Archivo y Edición* (y la interacción de cada uno de éstos con sus respectivos submenús), donde aplicando ACO se encontrarán las rutas parciales (las soluciones parciales). Ya luego se aplicará el otro método de optimización "Greedy" para resolver el problema de cobertura, y así obtener los mejores casos de prueba.

A continuación presentamos la ventana principal del aplicativo *WordPad 6.0.*

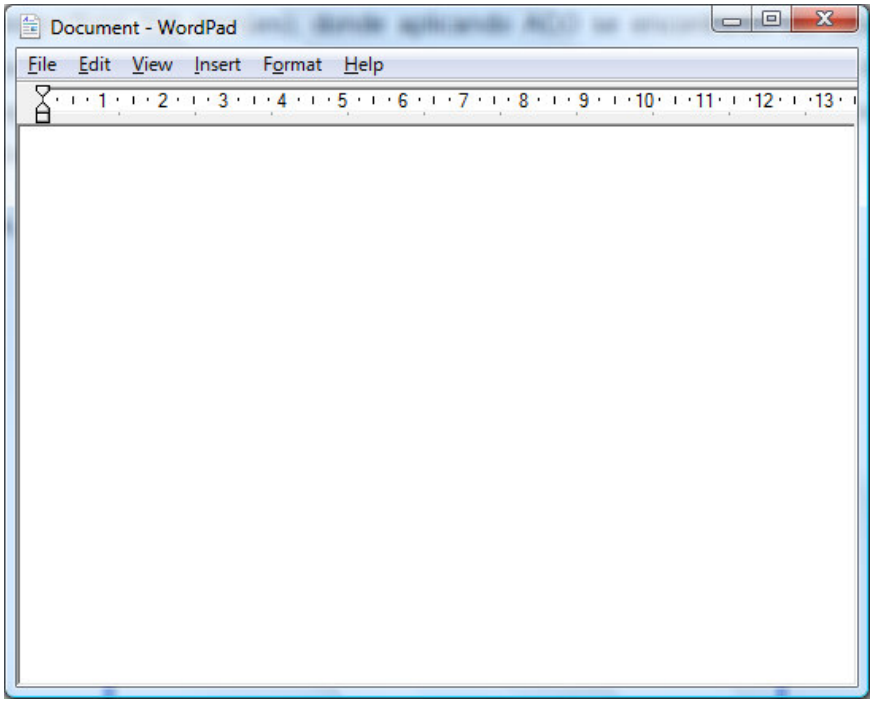


Figura 5.1 GUI del aplicativo "WordPad 6.0".

Los sub-menús sobre los cuales se generarán los casos de prueba, son:

- Para el menú File, los principales submenús elegidos son: New…, Open…, Save, Save As…, Print…, y Exit.

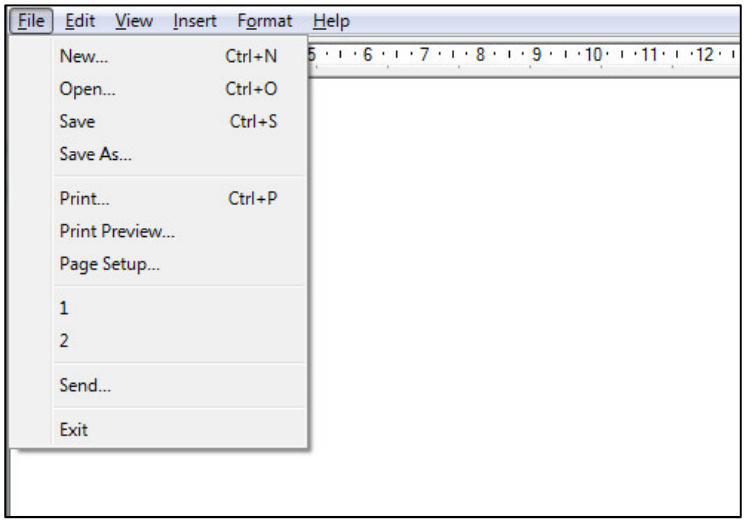


Figura 5.2 Submenú "File".

- Para el menú Edit, los principales submenús elegidos son: Undo, Cut, Copy, Paste, Paste Special…, Clear, Select All, Find…, Find Next, y Replace…

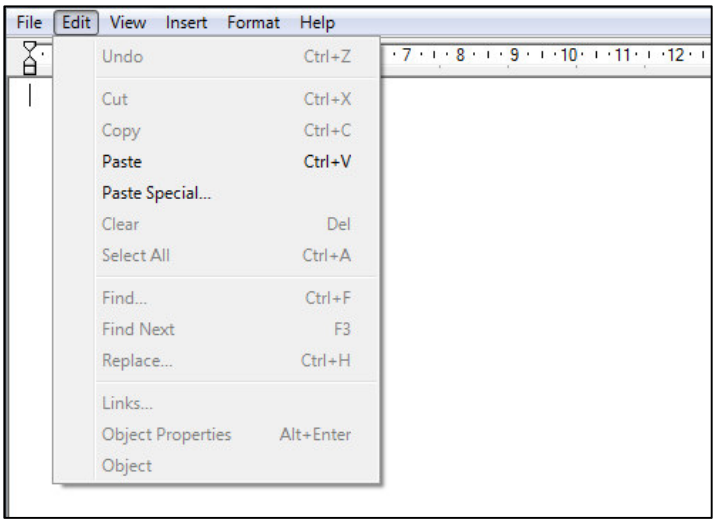


Figura 5.3 Submenú "Edit".

5.3 Flujo de eventos-GUI del WordPad

A continuación se presentan algunos flujos de eventos en el WordPad:

5.3.1 Flujo de Eventos: Nuevo documento

La secuencia inicia el recorrido con el evento clic File (barra de menú), luego ocurre el evento clic New... (aparece una *ventana modal* "New"). Sobre esta *ventana modal* se hace clic sobre un Ítem de la Lista (Rich Text Document). Finalmente clic sobre el botón OK, y nuevamente regresa al menú principal.

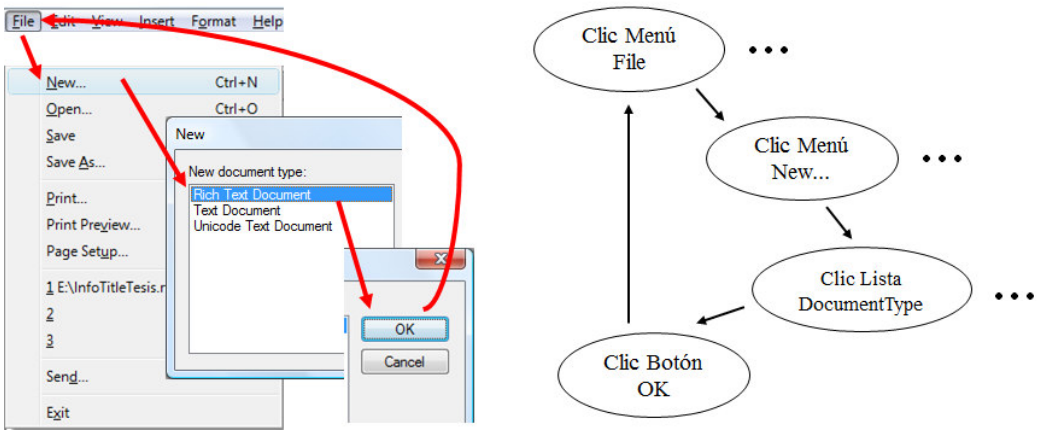


Figura 5.4 Flujo de eventos "Nuevo documento".

5.3.2 Flujo de Eventos: Abrir documento

La secuencia inicia el recorrido con el evento clic File (barra de menú), luego ocurre el evento clic Open… (aparece una *ventana modal* "Open"). Sobre esta *ventana modal* se hace clic sobre el Directorio de la Lista (LookIn) para buscar el archivo a abrir. Luego se hace clic en el Tipo de Filtro (FileType) para filtrar sólo los archivos con esa extensión seleccionada. Posteriormente se hace clic en la Lista de posibles archivos a abrir, y finalmente clic sobre el botón Open, para nuevamente regresar al menú principal.

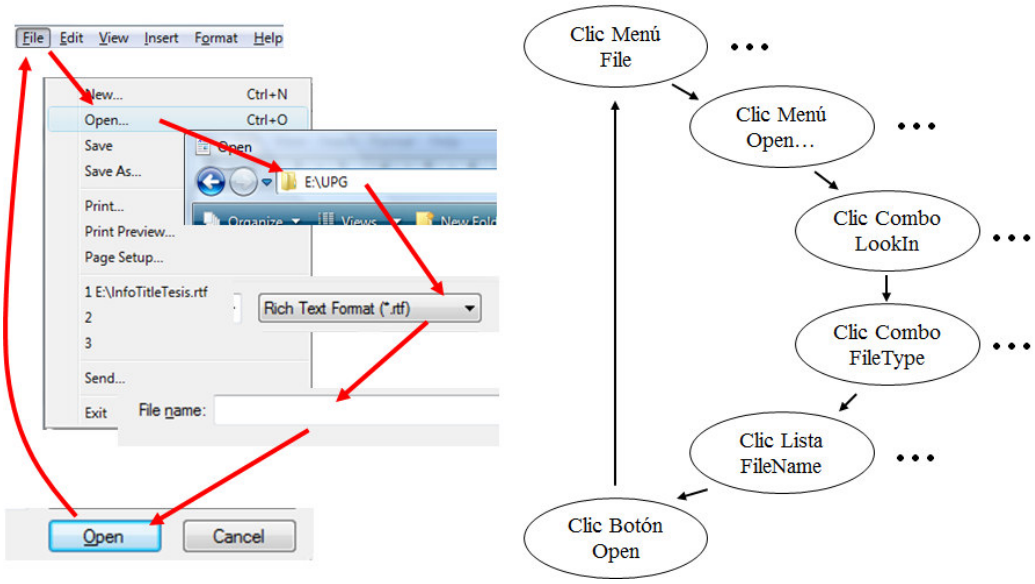


Figura 5.5 Flujo de eventos "Abrir documento".

5.3.3 Flujo de Eventos: Guardar como documento

La secuencia inicia el recorrido con el evento clic File (barra de menú), luego ocurre el evento clic Open… (aparece una *ventana modal* "Open"). Sobre esta *ventana modal* se hace clic sobre el Directorio de la Lista (LookIn) para buscar el archivo a abrir. Luego se hace clic en el Tipo de Filtro (FileType) para filtrar sólo los archivos con esa extensión seleccionada. Posteriormente se hace clic en la Lista de posibles archivos a abrir, y luego clic sobre el botón Open. En el menú principal se hace clic en File (barra de menú), luego clic en el evento Save As… (aparece una *ventana modal* "Save As"). Sobre esta *ventana modal* se hace clic sobre el Directorio de la Lista (SaveIn) para buscar la ruta donde se guardará el archivo. Luego se hace clic en el Tipo de Filtro como se desea guardar el archivo (SaveType). Posteriormente se hace clic en la caja de

texto para colocar el nombre del archivo a guardar (FileName). Finalmente se hace clic en el botón Save, para nuevamente regresar al menú principal.

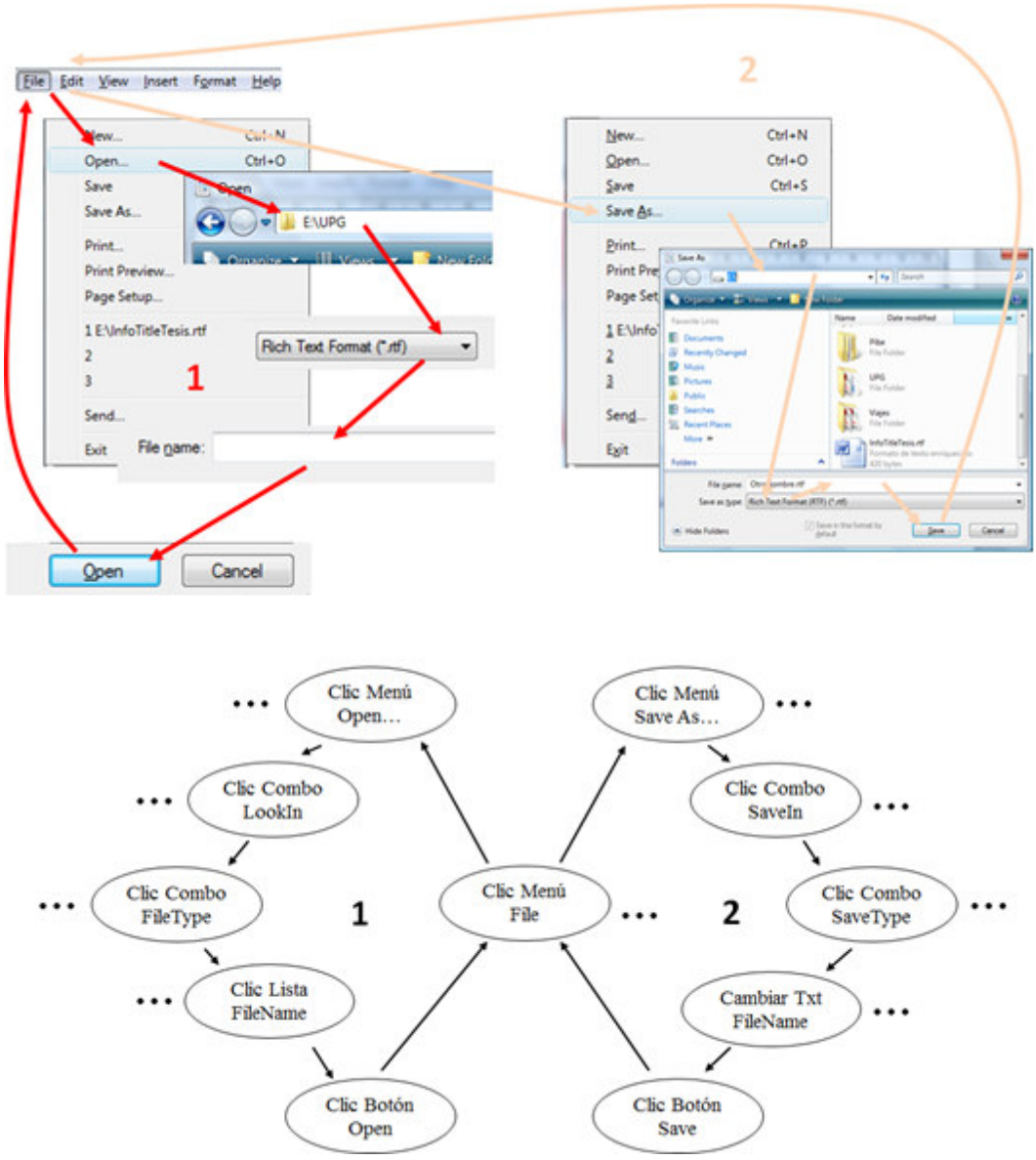


Figura 5.6 Flujo de eventos "Guardar como documento".

Los tres diagramas de flujo de eventos anteriores, representan la secuencia para algunos eventos que ocurren sobre el WordPad. El objetivo es que todas estas secuencias funcionales se modelen de tal manera que al ejecutarse con nuestro Sistema (aplicación de software), se calculen automáticamente los casos de pruebas asociados al aplicativo en estudio "WordPad 6.0"

5.4 Modelamiento de los eventos-GUI

Los objetivos de este caso de estudio, son:

- ✓ Generar las rutas parciales a través de ACO, ya que para generar nuestros casos de prueba partimos de un modelo de eventos-GUI que se relacionan entre sí; dicha interacción de los eventos se realizará a través de un grafo.
- ✓ Luego, una vez obtenidos las rutas parciales, aplicaremos Greedy para obtener los mejores casos de prueba para resolver el problema de cobertura.

Para ello, lo primero a realizar es representar los flujos de eventos (descritos en el punto 5.3) a través de un Grafo. Luego se describirá los principales eventos que interactúan en él, y finalmente se explicará el modo de interrelación entre los eventos-GUI del Grafo.

5.4.1 Grafo

Tiene representación vía una Matriz. Representa lo que vendría ser un grafo de eventos interrelacionados.

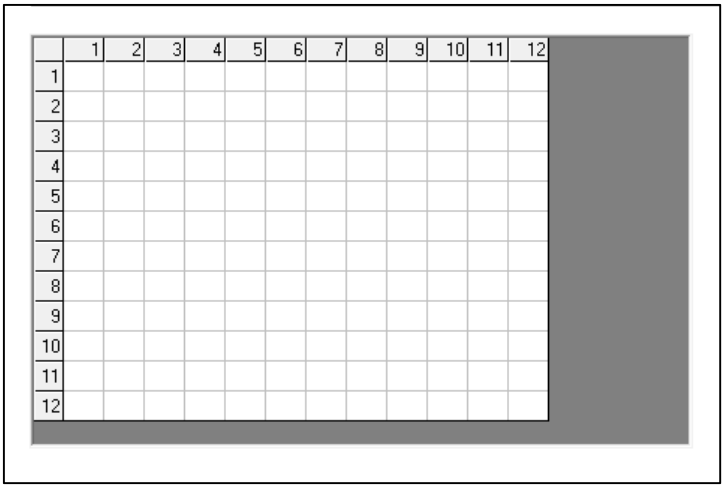


Figura 5.7 Matriz de eventos-GUI.

A continuación, presentamos el Grafo de eventos-GUI, representados a través de una Matriz, para nuestro caso de estudio.

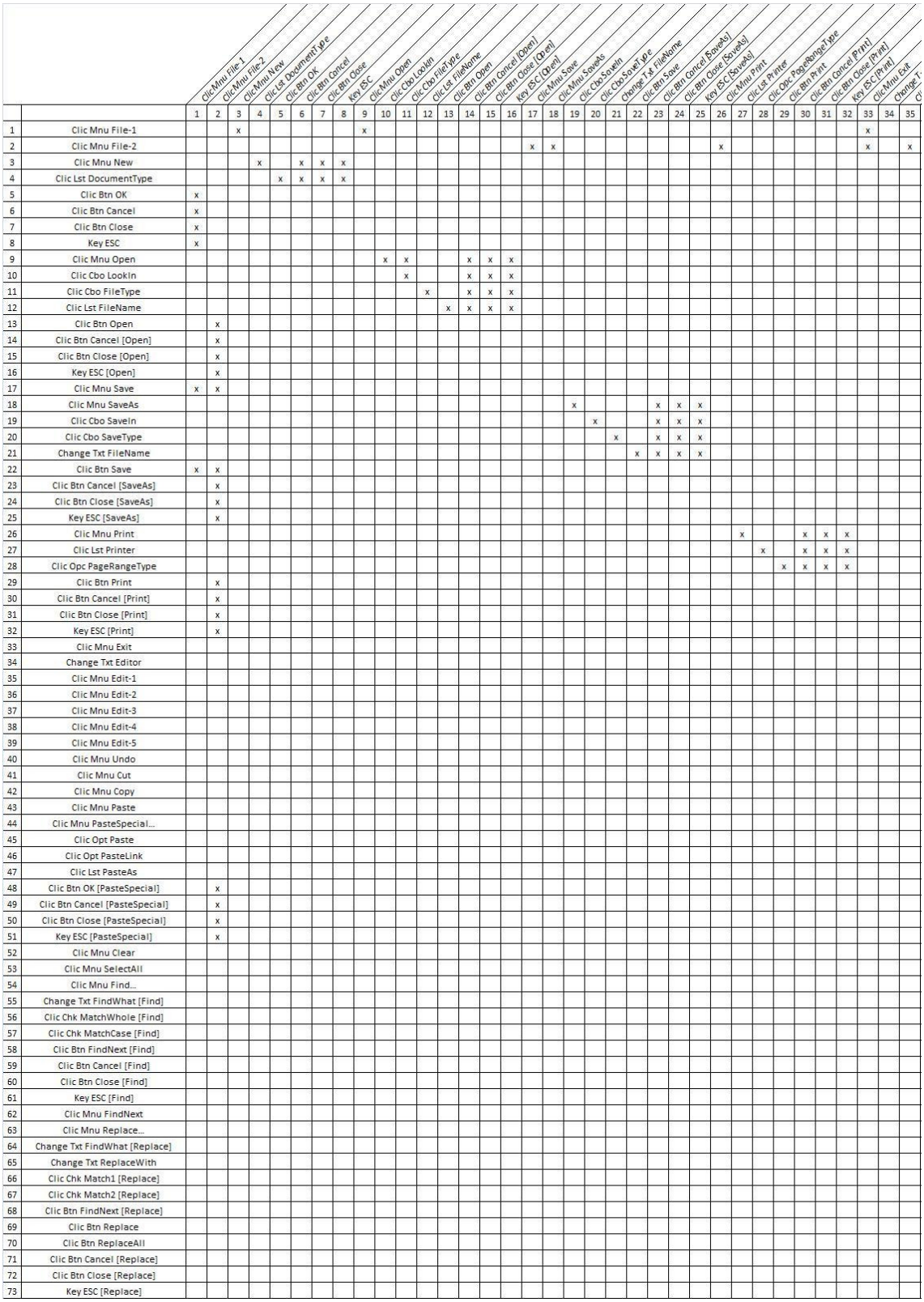


Figura 5.8 Matriz de eventos-GUI (lado izquierdo)

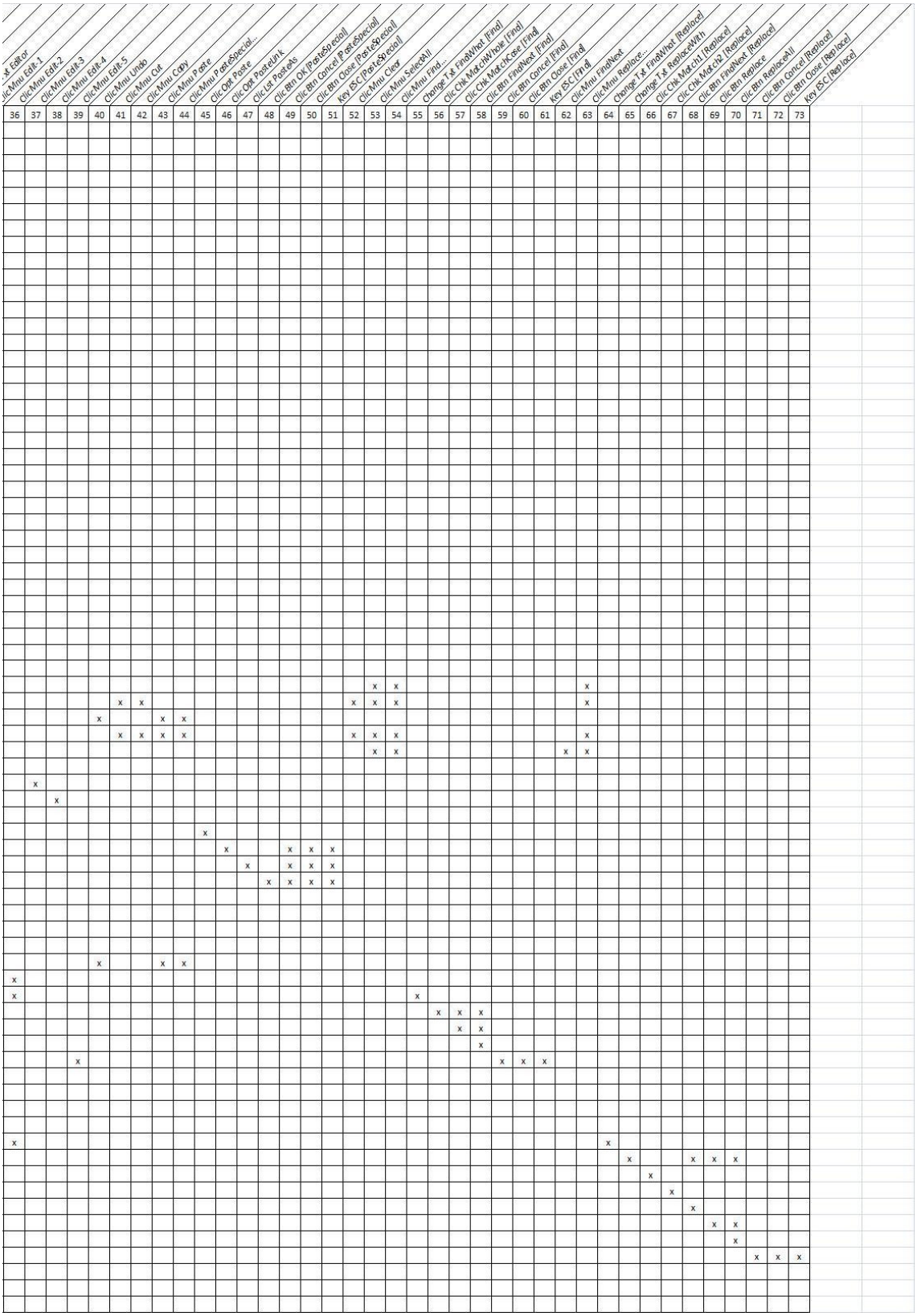


Figura 5.9 Matriz de eventos-GUI (lado derecho)

5.4.2 Eventos-GUI

A continuación se describirán los principales eventos-GUI que actúan sobre el Grafo de flujo de eventos:

- ✓ ***Clic Mnu File-1 (id=1):*** Es el evento de hacer clic en la opción File, en el menú principal del WordPad. Este evento representa el inicio, desde el cual se puede interrelacionar a 1 o más eventos dentro del grafo. A diferencia del evento "Clic Mnu File-2", desde este evento sólo estará disponible ir a los siguientes eventos: "Clic Mnu New", "Clic Mnu Open" o "Clic Mnu Exit".
- ✓ ***Clic Mnu File-2 (id=2):*** Es el evento de hacer clic en la opción File, en el menú principal del WordPad. Desde este evento sólo estará disponible ir a los siguientes eventos: "Clic Mnu Save", "Clic Mnu SaveAs", "Clic Mnu Print" , "Clic Mnu Exit" o "Clic Mnu Edit-1".
- ✓ ***Clic Mnu New (id=3):*** Es el evento de hacer clic sobre la opción New..., dentro de las opciones del menú File. A través de este evento se logra empezar un nuevo documento en el WordPad.
- ✓ ***Clic Mnu Open (id=9):*** Es el evento de hacer clic sobre la opción Open..., dentro de las opciones del menú File. A través de este evento se logra abrir un documento ya existente en el WordPad.
- ✓ ***Clic Mnu Save (id=17):*** Es el evento de hacer clic sobre la opción Save, dentro de las opciones del menú File. Este evento debe estar disponible siempre que se tenga un documento en curso (por ej. un documento que se haya abierto recientemente y se esté modificando su contenido)

  Según nuestro caso de estudio, este evento sólo es 'habilitado', después de ejecutarse el evento "Clic Mnu File-2".
- ✓ ***Clic Mnu SaveAs (id=18):*** Es el evento de hacer clic sobre la opción Save As..., dentro de las opciones del menú File. La diferencia con el evento "Clic Mnu Save", es que desde este evento se puede cambiar el nombre con el que se desea guardar el documento en curso (por ej. un documento que se haya abierto y se esté modificando su contenido)

  Según nuestro caso de estudio, este evento sólo es 'habilitado', después de ejecutarse el evento "Clic Mnu File-2".
- ✓ ***Clic Mnu Print (id=26):*** Es el evento de hacer clic sobre la opción Print..., dentro de las opciones del menú File. Este evento debe estar disponible siempre

que se tenga un documento en curso (por ej. un documento que se haya abierto recientemente) y se requiera imprimir.

Según nuestro caso de estudio, este evento sólo es 'habilitado', después de ejecutarse el evento "Clic Mnu File-2".

- ✓ ***Clic Mnu Exit (id=33):*** Es el evento de hacer clic sobre la opción Exit, dentro de las opciones del menú File. A través de este evento se logra salir del WordPad.
- ✓ ***Clic Mnu Edit-1 (id=35):*** Es el evento de hacer clic en la opción Edit, en el menú principal del WordPad. A través de este evento, sólo estará disponible ir a los siguientes eventos: "Clic Mnu SelectAll", "Clic Mnu Find...", o "Clic Mnu Replace...".

  Según nuestro caso de estudio, este evento sólo es 'habilitado', después de ejecutarse el evento "Clic Mnu File-2".
- ✓ ***Clic Mnu Edit-2 (id=36):*** Es el evento de hacer clic en la opción Edit, en el menú principal del WordPad. A través de este evento, sólo estará disponible ir a los siguientes eventos: "Clic Mnu Cut", "Clic Mnu Copy", "Clic Mnu Clear", "Clic Mnu SelectAll", "Clic Mnu Find...", o "Clic Mnu Replace,...".

  Según nuestro caso de estudio, este evento sólo es 'habilitado', después de ejecutarse cualquiera de los siguientes eventos: "Clic Mnu SelectAll", "Clic Mnu Find...", o "Clic Mnu Replace...".
- ✓ ***Clic Mnu Edit-3 (id=37):*** Es el evento de hacer clic en la opción Edit, en el menú principal del WordPad. A través de este evento, sólo estará disponible ir a los siguientes eventos: "Clic Mnu Undo", "Clic Mnu Paste", o "Clic Mnu PasteSpecial...".

  Según nuestro caso de estudio, este evento sólo es 'habilitado', después de ejecutarse cualquiera de los siguientes eventos: "Clic Mnu Cut", o "Clic Mnu Clear".
- ✓ ***Clic Mnu Edit-4 (id=38):*** Es el evento de hacer clic en la opción Edit, en el menú principal del WordPad. A través de este evento, sólo estará disponible ir a los siguientes eventos: "Clic Mnu Cut", "Clic Mnu Copy", "Clic Mnu Paste", "Clic Mnu PasteSpecial...", "Clic Mnu Clear", "Clic Mnu SelectAll", "Clic Mnu Find...", o "Clic Mnu Replace...".

  Según nuestro caso de estudio, este evento sólo es 'habilitado', después de ejecutarse el evento "Clic Mnu Copy".

- ✓ ***Clic Mnu Edit-5 (id=39):*** Es el evento de hacer clic en la opción Edit, en el menú principal del WordPad. A través de este evento, sólo estará disponible ir a los siguientes eventos: "Clic Mnu SelectAll", "Clic Mnu Find...", "Clic Mnu FindNext...", o "Clic Mnu Replace...".

  Según nuestro caso de estudio, este evento sólo es 'habilitado', después de ejecutarse el evento "Clic Btn FindNext [Find]".
- ✓ ***Clic Mnu Undo (id=40):*** Es el evento de hacer clic sobre la opción Undo, dentro de las opciones del menú Edit.

  Según nuestro caso de estudio, este evento sólo es 'habilitado', después de ejecutarse el evento "Clic Mnu Edit-3"; más específicamente, después de haberse ejecutado: el "Cut" o "Clear".
- ✓ ***Clic Mnu Cut (id=41):*** Es el evento de hacer clic sobre la opción Cut, dentro de las opciones del menú Edit.

  Según nuestro caso de estudio, este evento sólo es 'habilitado', después de ejecutarse cualquiera de los siguientes eventos: "Clic Mnu Edit-2", o "Clic Mnu Edit-4"; más específicamente, después de haberse ejecutado: el "SelectAll", "Find...", "Replace...", o "Copy".
- ✓ ***Clic Mnu PasteSpecial (id=44):*** Es el evento de hacer clic sobre la opción Paste Special..., dentro de las opciones del menú Edit. A través de este evento se logra pegar sobre el documento actual el contenido del Portapapeles.

5.4.3 Interacción con Ventanas Modales

La GUI a probar es el aplicativo *WordPad 6.0* (editor de texto que puede usarse para crear y editar documentos) el cual brinda las bondades de poder interactuar con otras ventanas (modales), además que para ciertos eventos del menú éstos NO están disponibles para todos los casos. Por ejemplo:

- Para que el *Evento Copy* este habilitado, previamente se debe haber seleccionado texto, o
- Para que el *Evento Find Next* este habilitado, previamente se debe haber activado el *Evento Find…,* etc.

Ventana modal:

Permite alternar el foco a otras ventanas del sistema, pero no a la ventana que le da origen ("ventana madre") hasta que se toma una acción sobre ella. Normalmente se utilizan para confirmar una acción del usuario.

Ahora bien, se explicará las opciones para representar la interacción de eventos en Ventanas Modales, y como éstas interactúan con el aplicativo *WordPad.*

- ✓ Una ventana modal puede ser tratada como un EVENTO más dentro del modelo de eventos-GUI (matriz)

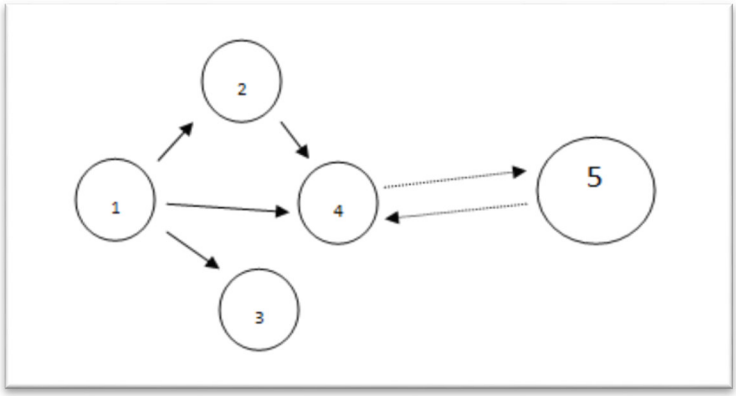


Figura 5.10 Secuencia de eventos que interactúan con una ventana modal, el cual se representa como un solo EVENTO.

- ✓ La ventana modal puede ser tratada como una INTERACCION DE EVENTOS los cuales se interrelacionan con el modelo de eventos-GUI.

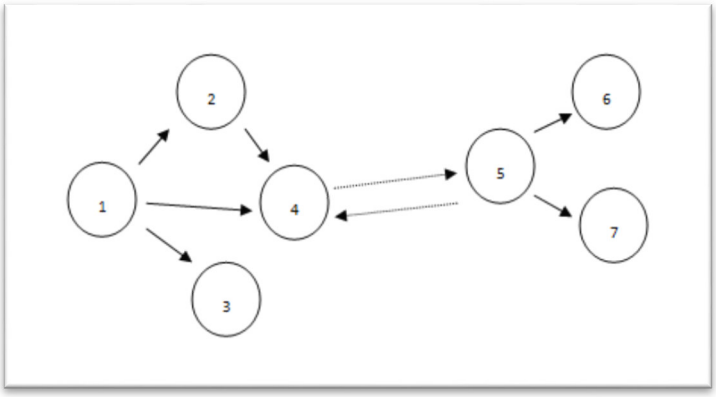


Figura 5.11 Secuencia de eventos que interactúan con una ventana modal, el cual se representa como una INTERACCION DE EVENTOS más.

Para nuestras validaciones con ventanas modales, hemos tomado el caso como si se tratase de Interacción con más Eventos.

5.5 Calibración de Parámetros para la Experimentación

En esta sección se presentarán los diferentes valores para los parámetros a usar en el algoritmo ACO.

En primer lugar, cabe comentar que los valores de los parámetros y pesos mostrados en las fórmulas de la descripción del algoritmo ACO (Capítulo 2) afectan en gran medida al funcionamiento de cada uno de ellos. Por su parte, los pesos fijan la importancia de los miembros de las heurísticas y las funciones de evaluación, a fin de decantar la búsqueda de soluciones hacia las zonas más prometedoras de acuerdo al criterio deseado, y valorarlas según dicho criterio respectivamente.

5.5.1 Parámetros alfa y beta

A continuación se describen dichos parámetros:

- ✓ α: es el *factor de ponderación de feromona.* Fija la importancia de la información memorística respecto a la heurística en la búsqueda, sobretodo en la regla de transición. Si su valor es grande, las concentraciones de feromona tendrán mucha relevancia a la hora de seleccionar el siguiente nodo en la construcción de una solución. Si dicho valor llegara a ser 0, no se tendría en cuenta la información de los rastros de feromona en la búsqueda, lo que significaría que ésta estaría guiada únicamente por heurísticas, convirtiendo el algoritmo en una búsqueda voraz.
- ✓ β: es el *factor de ponderación de heurística*. En este caso, determina la importancia de la información heurística respecto a la feromona en la búsqueda, siendo usando nuevamente en la regla de transición. Valores grandes para este parámetro implicarán una gran relevancia de esta información en la elección del siguiente nodo, seleccionando con mayor probabilidad aquel que mayor valor heurístico tenga asociado. Si tomase un valor igual a 0, no se consideraría ningún conocimiento heurístico en la búsqueda, siguiendo únicamente los rastros existentes, lo que llevaría a un estancamiento del algoritmo al considerar siempre los mismos nodos (pertenecientes a las soluciones obtenidas previamente por otras hormigas)

Para la calibración de los dichos parámetros, se realizaron muchas combinaciones de *alfa* y *beta,* tal que:

$\alpha = \{0, 0.2, 0.4, 0.6, 0.8, 1\}$, y

$\beta = \{0, 0.2, 0.4, 0.6, 0.8, 1, 1.4, 1.8, 2.2, 2.6, 3\}$, obteniendo 66 combinaciones posibles.

Luego, por cada combinación se realizó 5 pruebas consecutivas, esto con el objetivo de encontrar una mejor solución, peor solución y solución promedio para cada caso.

A continuación se mostrarán los resultados de la calibración de *alfa* y *beta,* tanto para la mejor solución, peor solución y solución promedio.

MEJOR SOLUCION											
		BETA									
		0	0.2	0.4	0.6	0.8	1	1.4	1.8	2.2	2.6	3
ALFA	0	28	28	28	27	27	27	27	27	28	27	28
	0.2	27	27	27	27	27	27	27	27	27	27	27
	0.4	27	27	26	27	26	26	27	26	26	27	27
	0.6	26	27	26	27	27	26	26	27	26	28	27
	0.8	27	26	28	26	26	27	26	26	27	27	27
	1	27	28	26	27	27	27	27	27	27	27	27

Figura 5.12 Mejor Solución: con cantidad mínima de Casos de Prueba.

Decimos que los parámetros *alfa* y *beta* nos devuelven la mejor solución: si éstos retornan la **menor cantidad de casos de prueba.** Tal como se muestra en la Figura 5.12, la cantidad mínima obtenida son 26 casos de prueba (sombreado de color amarillo), existiendo por lo menos 16 combinaciones de *alfa* y *beta* que cumplen dicho fin.

Por otro lado, decimos que los parámetros *alfa* y *beta* nos devuelven la peor solución: si éstos retornan la **mayor cantidad de casos de prueba.** Tal como se muestra en la Figura 5.13, la cantidad máxima obtenida es de 33 casos de prueba (sombreado de color rojo), existiendo 2 combinaciones de *alfa* y *beta* que cumplen dicho fin.

PEOR SOLUCION											
		BETA									
		0	0.2	0.4	0.6	0.8	1	1.4	1.8	2.2	2.6	3
ALFA	0	30	30	30	31	32	30	30	29	30	28	30
	0.2	30	28	29	31	29	30	29	30	30	30	29
	0.4	29	29	29	29	29	30	29	28	29	29	29
	0.6	29	33	29	31	29	29	28	30	29	29	29
	0.8	29	30	31	30	30	29	29	30	31	29	29
	1	30	30	29	31	30	30	29	33	31	29	29

Figura 5.13 Peor Solución: con cantidad máxima de Casos de Prueba.

Finalmente, calculamos la solución promedio el cual nos permitirá tener un mejor criterio a la hora de seleccionar los parámetros *alfa* y *beta,* con el propósito de obtener la menor cantidad de casos de prueba. Tal como se muestra en la Figura 5.14, y para fines de tener un resultado más preciso, se optó por mostrar los resultados con decimales (luego se realizará un truncamiento, ya que no existen casos de pruebas fraccionadas); esto garantizará seleccionar los valores correctos tanto para *alfa* como *beta.*

SOLUCION PROMEDIO											
		BETA									
		0	0.2	0.4	0.6	0.8	1	1.4	1.8	2.2	2.6	3
ALFA	0	29	28.6	28.8	29	29	28	28.6	28.4	28.6	27.8	28.8
	0.2	28	27.6	28.2	28.4	28.2	28.6	27.8	28.6	28.8	28	28
	0.4	28.2	28	27.6	28.2	27.6	28.4	28	27.4	27.8	27.8	28.2
	0.6	27.6	29.2	27.8	28.8	27.8	27.2	27.2	28	27.2	28.4	27.8
	0.8	27.6	28	29	28	27.2	27.8	27.8	28	28.2	28	28.2
	1	28.2	28.6	27	28.4	28.2	28.4	28.2	28.6	28.4	28	28

Figura 5.14 Solución Promedio: cantidad promedio de Casos de Prueba.

5.5.2 Análisis de los Resultados

Debemos indicar que cada uno de los valores descritos en la matriz alfa x beta, representa la cantidad de casos de prueba. El objetivo es identificar aquellos valores para alfa y beta que devuelvan la menor cantidad de casos de prueba.

En la Figura 5.12, nos damos cuenta que existen por lo menos 16 casos (combinaciones) que retornan la menor cantidad de casos de prueba (mejores soluciones). La pregunta es *¿Cuál elegir?*

Es por ello que el criterio a considerar para la elección de los valores de alfa y beta, estará basado en la cantidad mínima de casos de prueba que devuelvan ambas soluciones: Mejor Solución (Figura 5.12) y la Solución Promedio (Figura 5.14)

[α, β]	**Mejor Solución**	**Solución Promedio**	**Observaciones**
[0.4, 0.4]	26 casos prueba	27.6 casos prueba	No aplica
[0.4, 0.8]	26 casos prueba	27.6 casos prueba	No aplica
[0.4, 1]	26 casos prueba	28.4 casos prueba	No aplica
[0.4, 1.8]	26 casos prueba	27.4 casos prueba	No aplica
[0.4, 2.2]	26 casos prueba	27.8 casos prueba	No aplica
[0.6, 0]	26 casos prueba	27.6 casos prueba	No aplica
[0.6, 0.4]	26 casos prueba	27.8 casos prueba	No aplica
[0.6, 1]	26 casos prueba	27.2 casos prueba	No aplica
[0.6, 1.4]	26 casos prueba	27.2 casos prueba	No aplica
[0.6, 2.2]	26 casos prueba	27.2 casos prueba	No aplica
[0.8, 0.2]	26 casos prueba	28 casos prueba	No aplica
[0.8, 0.6]	26 casos prueba	28 casos prueba	No aplica
[0.8, 0.8]	26 casos prueba	27.2 casos prueba	No aplica
[0.8, 1.4]	26 casos prueba	28.2 casos prueba	No aplica
[0.8, 1.8]	26 casos prueba	28.8 casos prueba	No aplica
[1, 0.4]	**26 casos prueba**	**27 casos prueba (menor valor)**	**Aplica**

Tabla 5.1 Análisis de resultados respecto a alfa y beta.

Por lo tanto, se concluye en que para obtener *26 casos de prueba óptimos* (según la definición de nuestro problema a resolver), los valores para *alfa* y *beta* deben ser:

$$\alpha = 1 \text{ y } \beta = 0.4$$

5.6 Pantallas Principales del Aplicativo de software

- ✓ **Rutas Parciales:** Generado al aplicar ACO al GUI WordPad. *Mayor referencia, ver ítem 4.3.2.*

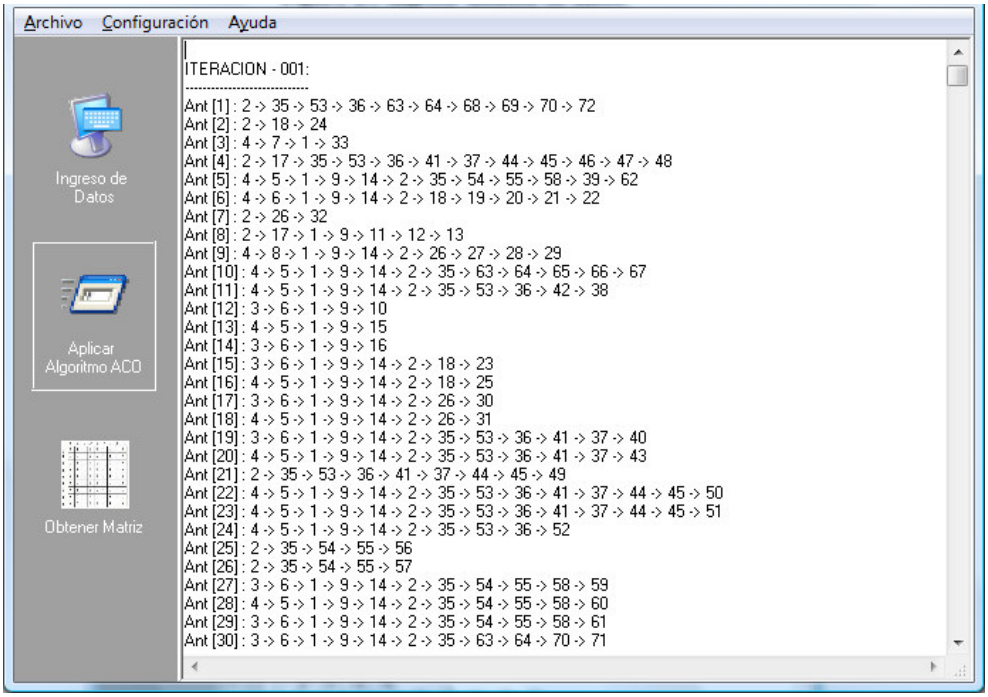


Figura 5.15 Generación de Rutas Parciales (ACO)

- ✓ **Casos de Prueba:** Resultado generado al aplicar Greedy sobre las rutas parciales. *Mayor referencia, ver ítem 4.3.3.*

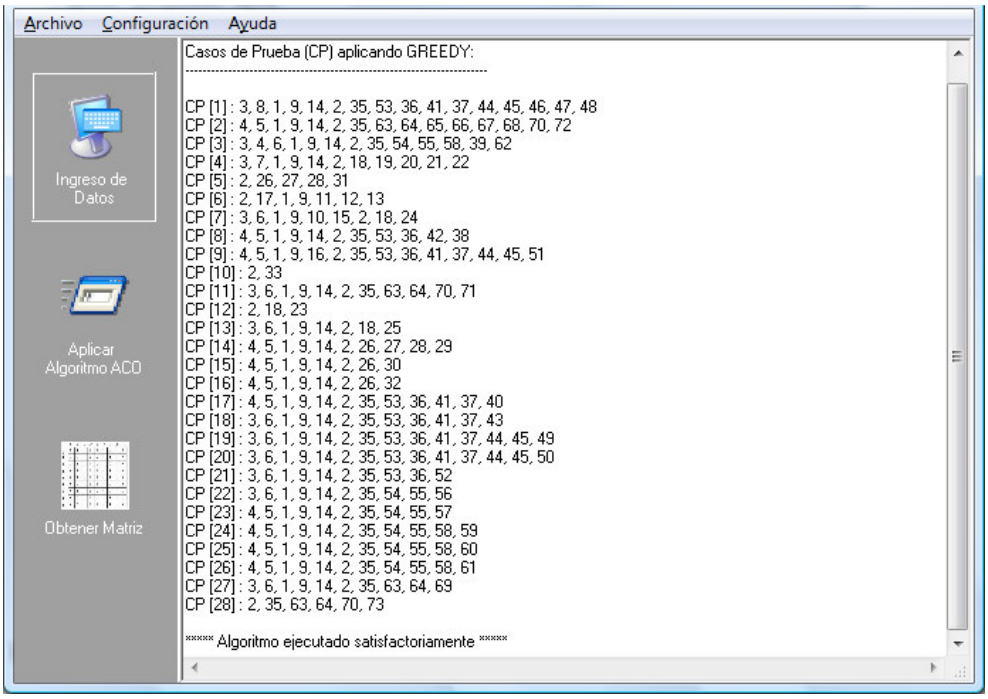


Figura 5.16 Generación de Casos de Prueba (Greedy)

5.7 Comparación con Método manual

Se realizará la comparación de nuestro modelo vs. el método tradicional realizado de forma manual:

Para ello se solicitó a un tester del área de control de calidad de la empresa "HIPER S.A" para que realice los casos de prueba aplicados a las 2 primeras opciones del WordPad: *menú Archivo* y *menú Edición.* El tester del área de calidad, elaboró en una plantilla Excel la definición de los casos de prueba posibles al WordPad. Definió 11 casos de pruebas, relevantes para el tester desde su punto de vista, y donde cada uno en promedio tuvo un tiempo de esfuerzo de 3 minutos. En total, el esfuerzo dedicado para elaborar los 11 casos de pruebas tomó un poco más de 30 minutos.

Nuestro modelo, que genera automáticamente los casos de prueba de las dos primeras opciones del WordPad: File (Archivo) y Edit (Edición), tomó 3 minutos y 46 segundos.

Para mayor detalle de los casos de prueba definidos por el tester del área de calidad, ver Anexo A.

Capítulo 6: Conclusiones y Trabajos Futuros

6.1 Conclusiones

- ✓ La Optimización de Colonia de Hormigas demuestra ser una potente herramienta para crear un conjunto de candidatos para la generación automática de casos de prueba para una GUI. En un conjunto manejable cubre el problema, porque los caminos candidatos son buenos caminos con menos visitas de los eventos.
- ✓ La implementación y resultados obtenidos de nuestro modelo de Casos de Prueba, fue realmente sencilla y dejan ver la eficacia del programa.
- ✓ El número de hormigas deben ser por lo menos la mitad del número de eventos, y la mejor combinación de los parámetros α y β se encontró experimentalmente.
- ✓ La velocidad del algoritmo de nuestro modelo es muy bueno y es mucho más rápido que en el caso del humano para generar los casos de prueba.

6.2 Trabajos Futuros

- ✓ Otros trabajos sugieren que cada evento se debe cubrir por lo menos por 5 diferentes casos de prueba. Nuestro modelo puede adaptarse a ese problema, y es una oportunidad de investigación.
- ✓ El uso de otro algoritmo metaheurístico y un experimento más complejo debe hacerse en el futuro para una mejor validación de este enfoque.
- ✓ Por ahora, nuestra herramienta de software lee datos de un archivo de texto que representa la matriz de adyacencia. En el futuro, de alguna manera gráfica se puede realizar el ingreso de los eventos en el grafo.
- ✓ La dependencia de un evento en otros eventos (estados) no se ha estudiado aquí, y es otro de los retos de investigación por delante.

Referencias Bibliográficas

[1] Kamde, P.M.; Nandavadekar, V.D. and Pawar, R.G., *Value of Test Cases in Software Testing,* 2006 IEEE International Conference on Management of Innovation and Technology, vol.2, pp.668-672.

[2] ISO/IEC 12207, *Software Life Cycle Processes,* 1995.

[3] Myers, G.J., *The Art of Software Testing,* Second Edition, 2004.

[4] McEwen, S., *Requirements: An introduction,* IBM Rational, IBM Corporation, Abril 2004.

[5] Sanz, L.F., *Tutorial: pruebas funcionales y trabajo en equipo,* Universidad Europea de Madrid, Grupo de calidad de software de ATI., 2007.

[6] Indira Chávez Valiente; Yucely López Trujillo y Martha Dunia Delgado Dapena. *Propuesta de generación de casos de prueba teniendo en cuenta indicadores de cubrimiento.* 2009, <http://www.laccei.org/LACCEI2009-Venezuela/p58.pdf>

[7] Atif M. Memon, *An event-flow model of GUI-based applications for testing,* vol.17, nº 3, 2007, pp. 137-157, <http://www.cs.umd.edu/~atif/papers/MemonSTVR2007.pdf>

[8] Daniella Anddrea Rojas Pacheco, *Generación de Casos de Pruebas Unitarios para Java basados en la Técnica de McGregor & Sykes,* Diciembre 2005, <http://cybertesis.ucv.cl:81/tesis/production/pucv/2005/rojas_da/html/index-frames.html>

[9] Beizer, B., *Software Testing Techniques,* 2°ed, Van Nostrand Reinhold, 1990.

[10] Maximiliano Cristiá. *Introducción al Testing de Software.* 2009, <www.fceia.unr.edu.ar/ingsoft/testing-intro-a.pdf>

[11] Marta Almirón, *Ant System,* Trabajo presentado como requisito para optar al título de Máster en Ingeniería de Sistemas, Universidad Nacional de Asunción, pag.8, Dic. 2000.

[12] De la Herrán Gascón M., *Notas sobre Computación Evolutiva,* 1999, <http://www.redcientifica.com/gaia/ia/ia_c.htm>

[13] Benoit Baudry; Franck Fleurey; Jean-Marc Jézéquel y Yves Le Traon, *From genetic to bacteriological algorithms for mutation-based testing,* vol.15, 2005, <http://portal.acm.org/citation.cfm?id=1077305>

[14] Atif M. Memon; Mary Lou Soffa & Martha E. Pollack, *Coverage Criteria for GUI Testing,* Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, ACM New York, USA, Pages: 256-267, ©2001.

[15] Dorigo M., *Ant Colony System,* <http://iridia.ulb.ac.be/dorigo/ACO/ACO.html>

[16] Dorigo M.; Maniezzo V. y Colorni A., *The Ant System: Optimization by a colony of cooperating agents,* IEEE Transaction on Systems, Man, and Cybernetics, vol.26, No.1, pp.1-13, 1996.

[17] M. Dorigo, Optimization, learning and natural algorithms [in Italian]. PhD thesis, Dipartimento di Elettronica, Politecnico di Milano, Milan, 1992.

[18] P. P. Grass´e, La reconstruction du nid et les coordinations inter-individuelles chez bellicositermes natalensis et cubitermes sp. La th´eorie de la Stigmergie: essai d interpretation du Comportement des Termites Constructeurs, pages 41–81, 1959.

[19] U. Aickelin, "A New Genetic Algorithm for Set Covering Problems", Annual Operational Research Conference 42, Swansea, UK, 2000.

[20] B. Crawford, C. Castro, "Combination of Constraint Programming Techniques and ACO for the Set Partitioning and Covering Problems" I Workshop in Constraint Programming. Facultad Ingeniería, Universidad de Concepción, Concepción, Chile. 2006.

[21] M. Dorigo, T. Stutzle, "The ant colony optimization metaheuristic: Algorithms, applications and advances". In F. Glover and G. Kochenberger, editors, Handbook of Metaheuristics, pages 251–285. Kluwer, 2002. Also available as Technical Report IRIDIA/2000-32, IRIDIA Universite Libre de Bruxells, Belgica. <ftp://iridia.ulb.ac.be/pub/mdorigo/tec.reps/TR.11-MetaHandBook.pdf>

[22] Guillermo Didier Bravo Ariza, Desarrollo y modelado de algoritmos para la genotipificación de secuencias: el caso de los transplantes, alelos HLA, Cap.5, 2005, <http://catarina.udlap.mx/u_dl_a/tales/documentos/lis/bravo_a_gd/capitulo5.pdf>

[23] Holland, J. H., "Adaptation in Natural and Artificial Systems", Ann Arbor: The University of Michigan Press, 1975.

[24] J. E. Beasley, "A lagrangean heuristic for set covering problems", Naval Research Logistics, 37:151-164, 1990.

[25] E. Balas and A. Ho., "Set covering algoritms using cutting planes, heuristics and subgradients optimization a computacional study", *Mathematical Programming Study,* 12:37-60, 1980.

[26] F. J. Vasko and G.R. Wilson, "An efficient heuristic for large set covering problems", *Naval Research Logistics Quarterly,* 31:163-171, 1984.

[27] L.W, Jacobs and M.J. Brusco, "A simulated annealing-based heuristic for the set covering problem", *Working paper, Operation Management and Information Systems Departament, Dekalb, IL 60115, USA,* 1993.

[28] S. Sen, "Minimal cost set covering using probabilistic methods", *ACM/SIGAPP Symposium on Applied computting,* 157-164, 1993.

[29] J.E. Beasley and P.C. Chu, "A genetic algorithm for the set covering problem", 1994.

[30] D. Cormier and S.S.Raghavan, "A very simple real-coded genetic algorithm", *North Carolina State University and University of North Carolina at Charlotte.*

[31] Vasek Chvátal., "A greedy heuristic for the set-covering problem", Mathematics of Operations Research, 4(3): 233-235, 1979.

[32] T.A. Feo and M.G.C. Resende, "A Probabilistic heuristic for a computationally difficult Set Covering Problem", *Oper Res Lett,* 8(1989), 67-71.

[33] T.A. Feo and M.G.C. Resende, "Greedy Randomized Adaptive Search Procedures", *Journal of Global Optimization,* 2(1995), 1-27.

[34] L.S. Pitsoulis and M.G.C. Resende, "Greedy Randomized Adaptive Search Procedures in *Handbook of Applied Optimization*", P. M. Pardalos and M. G. C. Resende Eds. (Oxford University Press, 2002), 168-182.

[35] F. Glover, *Future paths for integer programming and links to artificial intelligence.* Computers and Operations Research, 13:533-549*,* 1986.

[36] C.R. Reeves. *Modern Heuristic Techniques for Combinatorial Problems.* Blackwell Scientific Publishing, Oxford, UK, 1993.

[37] E. Alba. *Parallel Metaheuristics: A New Class of Algorithms.* John Wiley & Sons, October 2005.

[38] C. Blum and A. Roli. *Metaheuristics in combinatorial optimization: Overview and conceptual comparison.* ACM Computing Surveys, 35(3):268-308, 2003.

Anexo A

Plantilla de Casos de Prueba utilizado por los tester del área de calidad de la empresa "HIPER S.A". Sobre esta plantilla, se realizó la definición de los casos de prueba sobre las dos primeras opciones del WordPad: *Archivo y Edición.*

NRO. DE CASO DE PRUEBA	TIPO	MÓDULO	SUB MÓDULO 1	SUB MÓDULO 2	PROPÓSITO	DESCRIPCIÓN	RESULTADO ESPERADO
1	Principal	ARCHIVO	NUEVO		Obtener un nuevo documento de WordPad.	1. Seleccionar la opción Archivo del Menú Principal. 2. Seleccionar la opción Nuevo. 3. Seleccionar el Tipo de Nuevo Documento. 4. Dar Aceptar.	Se obtiene un Nuevo documento en Blanco.
2	Principal	ARCHIVO	ABRIR		Abrir un Documento WordPad existente.	1. Seleccionar la opción Archivo del Menú Principal. 2. Seleccionar la opción Abrir. 3. Seleccionar la Unidad de Disco. 4. Seleccionar el Documento a Abrir. 5. Dar clic en Abrir.	Se obtiene el Documento que se quiso Abrir.

3	Principal	ARCHIVO	GUARDAR		Validar la opción de Guardar en un Documento WordPad.	1. Seleccionar la opción Archivo del Menú Principal. 2. Seleccionar la opción Guardar. 3. Seleccionar la parte del Disco donde se guardará. 4. Escribir el Nombre del Documento. 5. Seleccionar el Tipo de documento. 6. Dar clic en Guardar.	Se Guarda el Documento de WordPad.
4	Principal	ARCHIVO	GUARDAR COMO		Validar la opción Guardar Como en un Documento WordPad.	1. Seleccionar la opción Archivo del Menú Principal. 2. Seleccionar la opción Guardar Como. 3. Seleccionar la parte del Disco donde se Guardar el Documento. 4. Asignarle Nombre al Documento. 5. Seleccionar el tipo de Archivo. 6. Dar clic en Guardar.	Se Guarda el Documento de WordPad.
5	Principal	ARCHIVO	IMPRIMIR		Validar la opción Imprimir.	1. Seleccionar la opción Archivo del Menú Principal. 2. Seleccionar la opción Imprimir. 3. Seleccionar la impresora. 4. Editar las Preferencias.. 5. Editar el intervalo de páginas a imprimir. 6. Editar el número de copias. 7. Dar imprimir.	Se obtiene la impresión del Documento.
6	Principal	ARCHIVO	SALIR		Validar que el WordPad se cierre correctamente.	1. Seleccionar la opción Archivo del Menú Principal. 2. Seleccionar la opción Salir. 3. El WordPad se cierra correctamente.	Se cierra correctamente el WordPad.

7	Principal	EDICION	COPIAR	PEGAR	Validar que la opción Copiar y Pegar se realice correctamente.	1. Seleccionar la opción Archivo del Menú Principal. 2. Seleccionar la opción Nuevo. 3. Seleccionar el Tipo de Nuevo Documento. 4. Dar Aceptar. 5. Escribir algún texto. 6. Seleccionar todo o parte del texto escrito. 7. Seleccionar la opción Edición del Menú Principal. 8. Seleccionar la opción Copiar. 9. En el área de escritura, ir a una nueva línea. 10. Seleccionar la opción Edición del Menú Principal. 11. Seleccionar la opción Pegar.	Después de Copiar, el pegado de un texto se realiza correctamente.
8	Principal	EDICION	CORTAR	PEGAR	Validar que la opción Cortar y Pegar se realice correctamente.	1. Seleccionar la opción Archivo del Menú Principal. 2. Seleccionar la opción Nuevo. 3. Seleccionar el Tipo de Nuevo Documento. 4. Dar Aceptar. 5. Escribir algún texto. 6. Seleccionar todo o parte del texto escrito. 7. Seleccionar la opción Edición del Menú Principal. 8. Seleccionar la opción Cortar. 9. En el área de escritura, ir a una nueva línea. 10. Seleccionar la opción Edición del Menú Principal. 11. Seleccionar la opción Pegar.	Después de Cortar, el pegado de un texto se realiza correctamente.

9	Principal	EDICION	SELECCIONAR TODO		Validar que el texto se seleccione todo.	1. Seleccionar la opción Archivo del Menú Principal. 2. Seleccionar la opción Nuevo. 3. Seleccionar el Tipo de Nuevo Documento. 4. Dar Aceptar. 5. Escribir algún texto. 6. Seleccionar la opción Edición del Menú Principal. 7. Seleccionar la opción Seleccionar todo.	Se selecciona todo el texto en el WordPad.
10	Principal	EDICION	BUSCAR		Validar que el texto se seleccione después de buscar.	1. Seleccionar la opción Archivo del Menú Principal. 2. Seleccionar la opción Nuevo. 3. Seleccionar el Tipo de Nuevo Documento. 4. Dar Aceptar. 5. Escribir algún texto. 6. Seleccionar la opción Edición del Menú Principal. 7. Seleccionar la opción Buscar. 8. Ingresar el texto a buscar. 9. Clic en el botón Buscar siguiente.	Se selecciona sólo el texto a buscar.
11	Principal	EDICION	REEMPLAZAR		Validar que un texto se reemplace por otro.	1. Seleccionar la opción Archivo del Menú Principal. 2. Seleccionar la opción Nuevo. 3. Seleccionar el Tipo de Nuevo Documento. 4. Dar Aceptar. 5. Escribir algún texto. 6. Seleccionar la opción Edición del Menú Principal. 7. Seleccionar la opción Reemplazar. 8. Ingresar el texto a buscar. 9. Ingresar el texto a reemplazar por. 10. Clic en el botón Reemplazar.	Se reemplaza el texto buscado por el texto a ser reemplazar.