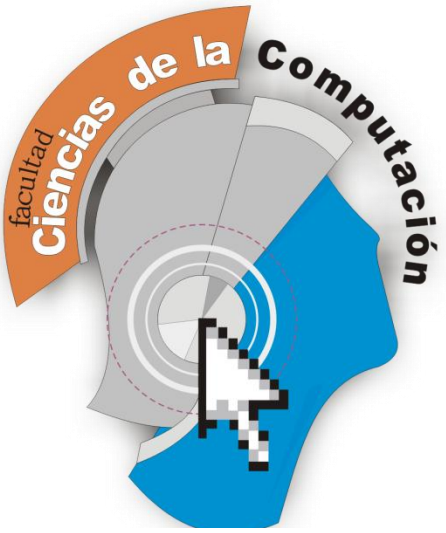
2012



Bémerita Universidad Autónoma de Puebla

TESIS PROFESIONAL

Para Obtener el Título de:

Licenciatura en Ciencias de la Computación

IMPLEMENTACIÓN DE UN ALGORITMO GENÉTICO EN PARALELO PARA RESOLVER EL PROBLEMA DE LA MOCHILA MULTIDIMENSIONAL CON CONDICIONES ADICIONALES

Presenta: Yareli Aburto Sánchez

Asesor: Dra. Darnes Vilariño Ayala

Coasesor: Mc. Mireya Tovar Vidal

AGRADECIMIENTOS

Quiero expresar mi más sincero agradecimiento a la Dra. Darnes Vilariño Ayala y a la MC. Mireya Tovar Vidal por su colaboración en la preparación de esta tesis, ya que sin su apoyo incondicional no me habría sido posible culminar este trabajo de investigación.

Le dedico este trabajo a mis Padres José Armando Aburto Sánchez y Ma. Esther Sánchez Lima que siempre han estado apoyándome y brindándome todo su amor, a mi hijo Daniel Andrés Alcántara Aburto quien es mi motor de vida y a mi esposo José Andrés Alcántara Alonso quien siempre ha estado conmigo pese a las adversidades que hemos pasado.

RESUMEN

Uno de los objetivos más importante del presente trabajo es estudiar el problema de la mochila multidimensional con condiciones adicionales generando una estrategia eficiente obteniendo un algoritmo genético paralelo.

Además se desarrolló un algoritmo genético secuencial para resolver el MKP con condiciones adicionales y un algoritmo genético paralelo para ofrecer la solución a problemas de este tipo. Se validaron los resultados obtenidos con los *"test problems"* expuestos [1]. Aunque no para todos los problemas expuestos se obtuvo el óptimo resultado reportado, los algoritmos llegaron a ser convergentes a este.

TABLA DE CONTENIDO

LISTA DE FIGURAS

LISTA DE ALGORITMOS

LISTA DE TABLAS

INTRODUCCIÓN

Los algoritmos genéticos (AG) son métodos adaptativos para optimizar funciones que se usan para resolver problemas de búsqueda y optimización, están basados en los principios que sustentan la evolución de las especies. A lo largo de las generaciones, las poblaciones evolucionan en la naturaleza, acorde con los principios de la selección natural y la supervivencia de los más fuertes, postulados por Darwin [2].

Por imitación de este proceso, los AG son capaces de ir creando soluciones para problemas del mundo real. Para esto, dentro de su implementación se establecen métodos análogos a la sobrevivencia de los individuos más aptos y los procedimientos genéticos de cruza y mutación. La evolución de dichas soluciones hacia valores óptimos del problema depende en buena medida de una adecuada codificación de las mismas.

El poder de los AG proviene del hecho de que se trata de una técnica robusta, que pueden tratar con éxito una gran variedad de problemas provenientes de diferentes áreas, si bien no se garantiza que el AG encuentre la solución óptima del problema, existe evidencia empírica de que se encuentran soluciones de un nivel aceptable, en un tiempo competitivo con el resto de los algoritmos de optimización combinatoria [3] .La principal característica de los AG es que se pueden aplicar a funciones sumamente complejas, además tienen la ventaja adicional que los códigos de los algoritmos genéticos son sumamente simples.

El objetivo general del presente trabajo es desarrollar un algoritmo genético paralelo para resolver el problema de asignación cuadrática con condiciones adicionales, es decir, donde pares de variables sólo tomen el valor uno al mismo tiempo [4].

La Ec. 1 muestra el problema de asignación cuadrática:

$$\begin{aligned} &\max\ z = c^t x \\ &sa: Ax \le b \\ &A \in M_{mxn}, b \in R^n, x \in (0,1)^n, c \in R^n, c_j < 0 \\ &x_i + x_j \le 1, i, j \subset \{1,..n\} \end{aligned} \qquad \textbf{(1)}$$

Los objetivos específicos son:

1- Estudiar el problema de la Mochila multidimensional con condiciones adicionales.

2- Diseñar un algoritmo genético simple para resolver este tipo de problemas, utilizando la librería GALIB [5].

3- Diseñar e implementar un algoritmo genético paralelo utilizando la librería, estableciendo los mecanismos adecuados de migración de las mejores soluciones de un nodo a otro.

4- Probar el sistema con los *test problems*.

La estructura de la tesis es la siguiente: en el Capítulo 1 se presenta el Marco Teórico sobre el cual se fundamenta el desarrollo de nuestros programas, se mencionan los conceptos fundamentales así como los antecedentes y origen de los algoritmos genéticos así como su estructura básica y conceptos, explicaremos también el problema de la Mochila multidimensional con condiciones adicionales. En el Capítulo 2 se presenta el planteamiento del problema, la elaboración de los casos de uso, los algoritmos genético paralelo y secuencial. El Capítulo 3 contiene las herramientas necesarias con las que realizamos el sistema (C++, Librería GALib), los requisitos básicos que se necesitan para ejecutar los programas así como la explicación de los parámetros y funciones que se utilizaron. En el Capítulo 4 se presentan las pruebas de los programas utilizando los *Test Problems* que hasta ahora se han publicado evaluándolos en tiempo y obteniendo el resultado más óptimo o más cercano al publicado. Por último tenemos las conclusiones donde se apreciaran los resultados obtenidos del presente trabajo y el alcance de nuestros objetivos presentados y la bibliografía de la cual se ha obtenido información así como los autores y artículos destacados.

CAPÍTULO 1: MARCO TEÓRICO

La incursión de la Inteligencia Artificial y los avances que ésta ha tenido vienen de la mano con nuevas técnicas en diversas áreas, una de esas técnicas es conocida como algoritmos genéticos los cuales son usados para resolver problemas de optimización, en este capítulo se presentan descripciones generales, definiciones y métodos, que nos permiten entender las ideas que sirven de base al diseño del algoritmo genético. Así como los conceptos que darán un preámbulo para el estudio del problema de la mochila multidimensional con condiciones adicionales.

1.1 Antecedentes

Los conocimientos sobre evolución se pueden aplicar en la resolución de problemas de optimización. La idea era "*evolucionar*" una población de candidatos a ser solución de un problema conocido, utilizando operadores inspirados en la selección natural y la variación genética natural. Fue Rechenber [6] quien introdujo las "*estrategias evolutivas*", método que empleó para optimizar parámetros reales para ciertos dispositivos. La misma idea fue desarrollada posteriormente por Schwefe.

El campo de las estrategias evolutivas ha permanecido como un área de investigación activa, cuyo desarrollo se produce, en su mayor parte de manera independiente al de los algoritmos genéticos (aunque recientemente se ha visto como ambas comunidades han comenzado a colaborar). Fogel, Owens y Walsh, fueron los creadores de la *"programación evolutiva"*, una técnica en la cual los candidatos a soluciones a tareas determinadas, eran representados por máquinas de estados finitos, cuyos diagramas de estados de transición evolucionaban mediante mutación aleatoria, seleccionándose el que mejor aproximará.

Estas tres áreas, estrategias evolutivas, algoritmos genéticos y programación evolutiva, son las que forman la columna vertebral de computación evolutiva, y de ellas parten los caminos hacia todos los campos de investigación inspirados en nuestros conocimientos sobre evolución.

Otros investigadores desarrollaron su trabajo en los algoritmos para la optimización y el aprendizaje inspirados en la evolución. Cabe resaltar nombres como los de Box, Friedman, Bledsoe, Bremermann, y Reed, Toombs y Baricelli [7]. Sin embargo, su trabajo no ha tenido, la atención que han recibido las estrategias evolutivas, programación evolutiva y los algoritmos genéticos. La primera mención del término de algoritmos genéticos y la primera ubicación sobre una aplicación del mismo, se deben a Bagley, que diseñó algoritmos genéticos para buscar conjuntos de parámetros en funciones de evaluación de juegos, y los comparó con los algoritmos de correlación, procedimientos de aprendizaje modelados después de los algoritmos de pesos variantes de ese periodo. Pero es otro científico el considerado creador de los algoritmos genéticos: John Holland, que los desarrolló, junto a sus alumnos y colega [8].

En contraste con las estrategias evolutivas y la programación evolutiva, el propósito original de Holland no era diseñar algoritmos para resolver problemas concretos, sino estudiar, de un modo formal, el fenómeno de la adaptación tal y como ocurre en la naturaleza, y desarrollar vías para extrapolar esos mecanismos de adaptación natural a los sistemas computacionales.

El libro que Holland [9] escribió, *Adaptación en Sistemas Naturales y Artificiales* presentaba el algoritmo genético como una abstracción de la evolución biológica, y proporcionaba el entramado teórico para la adaptación bajo el algoritmo genético. El algoritmo genético de Holland era un método para desplazarse, de una población de cromosomas (bits) a una nueva población, utilizando un sistema similar a la *"selección natural"* junto con los operadores de *cruce*, *mutación* e *inversión* inspirados en la genética. Los principios básicos de los algoritmos genéticos fueron establecidos por Holland y se encuentran bien descritos en varios textos Goldberg [10], Davis [11], Michalewicz [12]. En la naturaleza los individuos de una población compiten entre sí, en la búsqueda de recursos tales como comida, agua y refugio. Incluso los miembros de una misma especie compiten a menudo en la búsqueda de un compañero, aquellos individuos que tienen más éxito en sobrevivir y en atraer compañeros tienen mayor probabilidad de generar un gran número de descendientes y por el contrario individuos poco dotados producirán un menor número de descendientes.

Esto significa que los genes de los individuos mejor adaptados se propagarán en sucesivas generaciones hacia un número de individuos creciente. El conocimiento que las especies adquieren se codifica en sus cromosomas y se transmite a la siguiente generación de especies, las cuales tendrán mayores probabilidades de sobrevivir. Los algoritmos genéticos pueden ser clasificados como una metaheurística A/S/P Goldberg [6] Schacher [10] y Eshelman reconocen que los AG no son la mejor alternativa en el caso de problemas combinatorios, esto se debe a que las representaciones que inducen a la obtención de buenos esquemas de cruzamiento son difíciles de encontrar. Este problema es más por el hecho de que todavía no existe una clara teoría de esquemas de cruzamiento. La posibilidad de encontrar muy buenas soluciones cercanas al óptimo depende del tamaño del problema y de la calidad de la población inicial.

1.1.1 Evolución Biológica: Conceptos Biológicos a Algoritmos Genéticos

Todo organismo vivo consiste de células y en cada célula hay el mismo conjunto de cromosomas. Los cromosomas son cadenas de ADN y sirven como modelo del organismo completo.

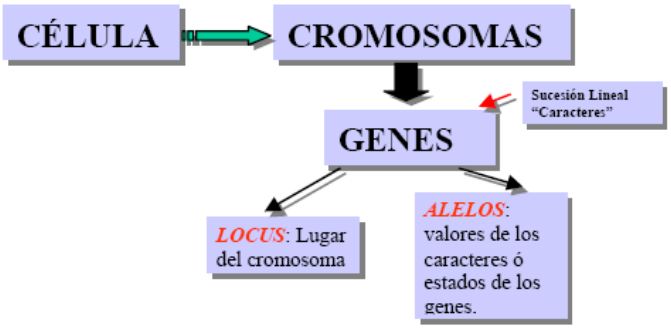


Figura 1.1 Composición de Cromosomas

En la Figura 1.1 se muestra una descripción general de cómo se componen los cromosomas, un cromosoma consiste de genes, bloques de ADN.

Cada gen codifica una proteína particular. Básicamente, podría decirse que cada gen codifica una característica. Cada gen tiene una propia posición en el cromosoma, esta posición se denomina locus [12].

El conjunto completo del material genético (todos los cromosomas) se llama genoma. Un conjunto particular de genes en el genoma es llamado genotipo. El genotipo va con el desarrollo posterior como la base del nacimiento para el fenotipo del organismo, el cual es la característica (física o mental), como el color de los ojos, la inteligencia etc. Durante la reproducción, lo primero que ocurre es la recombinación (crossover). Los genes padres se combinan para formar un nuevo cromosoma. El retoño generado puede mutar. Las mutaciones son pequeñas variaciones en los elementos del ADN. La aptitud de un organismo se mide por el éxito de dicho organismo en su vida (sobrevivir) [13].

Los algoritmos genéticos usan una analogía directa con el comportamiento natural. Trabajan con una población de individuos, cada uno de los cuales representan una solución factible a un problema dado. A cada individuo se le asigna un valor ó puntuación, relacionado con la bondad de dicha solución. En la naturaleza esto equivaldría al grado de efectividad de un organismo para competir por un determinado curso. Cuanto mayor sea la adaptación de un individuo al problema, mayor será la probabilidad de que el mismo sea seleccionado para reproducirse, cruzando su material genético con otro individuo seleccionado de igual forma.

Este cruce producirá nuevos individuos (descendientes de los anteriores) los cuales comparten algunas de las características de sus padres. Cuanto menor sea la adaptación de un individuo, menor será la probabilidad de que dicho individuo sea seleccionado para la reproducción, y por tanto de que su material genético se propague en sucesivas generaciones. De esta manera se produce una nueva población de posibles soluciones, la cual remplaza a la anterior y verifica la interesante propiedad de que contiene una mayor proporción de buenas características en comparación con la población anterior. Así a lo largo de las generaciones, las buenas características se propagan a través de la población.

Favoreciendo el cruce de los individuos mejor adaptados, van siendo exploradas las áreas más prometedoras del espacio de búsqueda. Si el Algoritmo Genético ha sido bien diseñado, la, población convergerá hacia una solución óptima del problema.

I.1.2 Definiciones Utilizadas en Algoritmos Genéticos

En la Tabla 1.1 se muestra la analogía entre los algoritmos genéticos y el sistema natural.

Algoritmo Genético	**Significado**
Cromosomas (cadena, individuo)	Solución (código)
Genes(bits)	Parte de la solución
Locus	Posición del Gen
Alelos	Valor del Gen
Fenotipo	Solución Decodificada (Apariencia Externa)
Genotipo	Solución Codificada (Estructura Interna)

Tabla 1.1 Descripción básica del Algoritmo Genético

Los conceptos básicos que se manejan en la teoría de los algoritmos genéticos son los siguientes:

- *Individuo o cromosoma*: Un individuo determina una potencial solución del problema que se pretende resolver mediante el algoritmo genético.

- *Población:* Conjunto de individuos con los que se trabaja en el algoritmo genético. En un algoritmo genético los individuos que constituyen la población van cambiando, pero generalmente el tamaño de la misma permanece constante.

- *Función fitness:* Se trata de una función evaluadora de la calidad de un individuo como solución a nuestro problema. Permite la ordenación de los individuos de la población en cuanto a bondad de los mismos.

- *Cruce:* es una de las operaciones fundamentales que intervienen en todo algoritmo genético. Como norma general se aplica después de un proceso de selección de dos individuos y consiste en una combinación de los mismos para obtener como resultado otros dos nuevos individuos.

- *Mutación:* Constituye otra operación fundamental en un algoritmo genético. En este caso se selecciona un individuo, el cual sufre una pequeña modificación aleatoria en su codificación obteniéndose otro individuo nuevo.

- *Gen:* Partícula de los cromosomas que producen la aparición de caracteres hereditarios.

- *Evolución:* Serie de transformaciones sucesivas que han sufrido los seres vivos desde los tiempos geológicos.

- *Aptitud:* Medida de la aptitud de un individuo para su supervivencia.

- *Generación:* Iteración de la medida de aptitud y la creación de una nueva población por medio de operaciones genéticas.

- *Reproducción:* Operación genética que origina la creación de una copia exacta de la representación genética de un individuo de la población.

- *Función de Evaluación*: Es la función que elige la próxima generación sobreviviente.

- *Muchedumbre*: Fenómeno donde las mejores hipótesis se reproducen siempre y proliferan, reduciendo la diversidad de la población y las posibilidades de evolución.

A continuación se explicarán los términos o parámetros más importantes manejados en los algoritmos genéticos.

Función de Evaluación (Fitness)

La evaluación es la unión entre el AG y el mundo externo. La evaluación se realiza a través de una función que representa de forma adecuada al problema y tiene como objetivo suministrar una medida de aptitud de cada individuo en la población actual. La función de evaluación es para un AG lo que el medio ambiente es para los seres humanos. Las funciones de evaluación son específicas de cada problema.

Un buen diseño de la función de evaluación (también conocida como función objetivo o función de adaptación) resulta extremadamente importante para el correcto funcionamiento de un AG. Esta función determina el grado de adaptación o aproximación de cada individuo al problema y por lo tanto permite distinguir a los mejores individuos de los peores.

Tamaño de la Población

Una cuestión que se puede plantear es la relacionada con el tamaño idóneo de la población. Parece intuitivo que las poblaciones pequeñas corren el riesgo de no cubrir adecuadamente el espacio de búsqueda, mientras que el trabajar con poblaciones de gran tamaño puede acarrear problemas relacionados con el excesivo costo computacional.

Alander [14], basándose en evidencia empírica sugiere que un tamaño de población comprendida entre l y 21 es suficiente para atacar con éxito los problemas de manera general.

Población Inicial

Habitualmente la población inicial se escoge generando ristras al azar, pudiendo contener cada gen uno de los posibles valores del alfabeto con probabilidad uniforme. Se podría preguntar qué es lo que sucedería si los individuos de la población inicial se obtuviesen como resultado de alguna técnica heurística o de optimización local. En los pocos trabajos que existen sobre este aspecto, se constata que esta inicialización no aleatoria de la población inicial, puede acelerar la convergencia del AG. Sin embargo en algunos casos la desventaja resulta ser la prematura convergencia del algoritmo, queriendo indicar con esto la convergencia hacia óptimos locales.

La población inicial de un AG puede ser creada de muy diversas formas, desde generar aleatoriamente el valor de cada gen para cada individuo, utilizar una función ávida o generar alguna parte de cada individuo y luego aplicar una búsqueda local.

Función Objetivo

Dos aspectos que resultan cruciales en el comportamiento de los AG son la determinación de una adecuada función de adaptación o función objetivo, así como la codificación utilizada. Idealmente es conveniente construir funciones objetivo con "ciertas regularidades", es decir, funciones objetivo que verifiquen que para dos individuos que se encuentren cercanos en el espacio de búsqueda, sus respectivos valores en las funciones objetivo sean similares. Por otra parte una dificultad en el comportamiento del AG puede ser la existencia de gran cantidad de óptimos locales, así como el hecho de que el óptimo global se encuentre muy aislado.

La regla general para construir una buena función objetivo es que ésta debe reflejar el valor del individuo de una manera "real", pero en muchos problemas de optimización combinatoria, donde existe gran cantidad de condiciones, buena parte de los puntos del espacio de búsqueda representan individuos no válidos. Para este planteamiento en el que los individuos están sometidos a restricciones, se han propuesto varias soluciones. La primera sería la que se podría denominar absolutista, en la que aquellos individuos que no verifican las restricciones, no son considerados como tales, y se siguen efectuando cruces y mutaciones hasta obtener individuos válidos, o bien, a dichos individuos se les asigna una función objetivo igual a cero.

Otra posibilidad consiste en reconstruir aquellos individuos que no verifican las condiciones. Dicha reconstrucción suele llevarse a cabo por medio de un nuevo operador que se acostumbra a denominar reparador. La idea general consiste en dividir la función objetivo del individuo por una cantidad (la penalización) que guarda relación con las condiciones que dicho individuo viola.

Dicha cantidad puede simplemente tener en cuenta el número de condiciones violadas ó bien el denominado costo esperado de reconstrucción, es decir el coste asociado a la conversión de dicho individuo en otro que no viole ninguna restricción. Otra técnica que se ha venido utilizando en el caso de que la evaluación de la función objetivo, es la denominada evaluación aproximada de la función objetivo.

En algunos casos la obtención de n funciones objetivo aproximadas puede resultar mejor que la evaluación exacta de una única función objetivo (supuesto el caso de que la evaluación aproximada resulte como mínimo, *n* veces más rápida que la evaluación exacta). Un problema habitual en las ejecuciones de los AG surge debido a la velocidad con la que el algoritmo converge.

En algunos casos la convergencia es muy rápida, lo que suele denominarse convergencia prematura, en la cual el algoritmo converge hacia óptimos locales, mientras que en otros casos el problema es justo el contrario, es decir se produce una convergencia lenta del algoritmo. Una posible solución a estos problemas pasa por efectuar transformaciones en la función objetivo.

El problema de la convergencia prematura, surge a menudo cuando la selección de individuos se realiza de manera proporcional a su función objetivo. En tal caso, pueden existir individuos con una adaptación al problema muy superior al resto, que a medida que avanza el algoritmo "dominan" a la población. Por medio de una transformación de la función objetivo, en este caso una comprensión del rango de variación de la función objetivo, se pretende que dichos "superindividuos" no lleguen a dominar a la población.

El problema de la lenta convergencia del algoritmo, se resolvería de manera análoga, pero en este caso efectuando una expansión del rango de la función objetivo. La idea de especies de organismos, ha sido imitada en el diseño de los AG en un método propuesto por Goldberg [6] y Richardson, utilizando una modificación de la función objetivo de cada individuo, de tal manera que individuos que estén muy cercanos entre sí devalúen su función objetivo, con objeto de que la población gane en diversidad.

Operadores Genéticos

Un algoritmo genético puede utilizar muchas técnicas diferentes para seleccionar a los individuos que deben copiarse hacia la siguiente generación, a continuación se explican los más comunes, algunos de estos métodos son mutuamente exclusivos, pero otros pueden utilizarse en combinación, algo que se hace a menudo [15].

Selección

A continuación se explican brevemente algunas de las técnicas de Selección:

- *Selección elitista*: se garantiza la selección de los miembros más aptos de cada generación. (La mayoría de los AGs no utilizan elitismo puro, sino que usan una forma modificada por la que el individuo mejor, o algunos de los mejores, son copiados hacia la siguiente generación en caso de que no surja nada mejor).

- *Selección proporcional a la aptitud:* los individuos más aptos tienen más probabilidad de ser seleccionados, pero no la certeza.

- *Selección escalada:* al incrementarse la aptitud media de la población, la fuerza de la presión selectiva también aumenta y la función de aptitud se hace más discriminadora. Este método puede ser útil para seleccionar más tarde, cuando todos los individuos tengan una aptitud relativamente alta y sólo les distingan pequeñas diferencias en la aptitud.

- *Selección por torneo:* se eligen subgrupos de individuos de la población, y los miembros de cada subgrupo compiten entre ellos. Sólo se elige a un individuo de cada subgrupo para la reproducción.

- *Selección por rango:* a cada individuo de la población se le asigna un rango numérico basado en su aptitud, y la selección se basa en este ranking, en lugar de las diferencias absolutas en aptitud. La ventaja de este método es que puede evitar que individuos muy aptos ganen dominancia al principio a expensas de los menos aptos, lo que reduciría la diversidad genética de la población y podría obstaculizar la búsqueda de una solución aceptable.

- *Selección generacional:* la descendencia de los individuos seleccionados en cada generación se convierte en toda la siguiente generación. No se conservan individuos entre las generaciones.

- *Selección por estado estacionario*: la descendencia de los individuos seleccionados en cada generación vuelven al acervo genético prexistente, remplazando a algunos de los miembros menos aptos de la siguiente generación. Se conservan algunos individuos entre generaciones.

- *Selección jerárquica:* los individuos atraviesan múltiples rondas de selección en cada generación. Las evaluaciones de los primeros niveles son más rápidas y menos discriminatorias, mientras que los que sobreviven hasta niveles más altos son evaluados más rigurosamente. La ventaja de este método es que reduce el tiempo total de cálculo al utilizar una evaluación más rápida y menos selectiva para eliminar a la mayoría de los individuos que se muestran poco o nada prometedores, y sometiendo a una evaluación de aptitud más rigurosa y computacionalmente más costosa sólo a los que sobreviven a esta prueba inicial.

Mutación

Se define mutación como una variación de la información contenida en el código genético habitualmente, un cambio de un gen a otro producido por algún factor exterior al algoritmo genético [16]. Algunas de las razones que pueden motivar a incorporar mutaciones en nuestro algoritmo son:

- *Desbloqueo del algoritmo*. Si el algoritmo se bloqueó en un mínimo parcial, una mutación puede sacarlo al incorporar nuevos fenotipos de otras zonas del espacio.

- *Acabar con poblaciones degeneradas.* Puede ocurrir que, bien por haber un cuasi-mínimo, bien porque en pasos iniciales apareció un individuo demasiado bueno que acabó con la diversidad genética, la población tenga los mismos fenotipos. A priori se pueden plantear algunas soluciones, como el escalamiento de la función de adaptación; mas, si ya se ha llegado a una población degenerada, es preciso que las mutaciones introduzcan nuevos genomas. Como analizaremos en el operador de cruce, esto se hace implícitamente en cada cruce.

- *Incrementar el número de saltos evolutivos. Los* saltos evolutivos -aparición de un fenotipo especialmente valioso, o, dicho de otra forma, salida de un mínimo local- son muy poco probables en un genético puro para un problema genérico. La mutación permite explorar nuevos subespacios de soluciones, por lo que, si el subespacio es bueno en términos de adaptación, se producirá un salto evolutivo después de la mutación que se expenderá de forma exponencial por la población.

- *Enriquecer la diversidad genética.* Es un caso más suave que el de una población degenerada -por ejemplo, que la población tenga una diversidad genética pobre-, la mutación es un mecanismo de prevención de las poblaciones degeneradas. Sin embargo, si la tasa de mutación es excesivamente alta tendremos la ya conocida deriva genética. Una estrategia muy empleada es una tasa de mutación alta al inicio del algoritmo, para aumentar la diversidad genética, y una tasa de mutación baja al final del algoritmo, para conseguir que converja

- No es conveniente abusar del operador de mutación si no queremos que el AG se convierta en un algoritmo de búsqueda al azar. Igual que sucede en la fase de cruce (reproducción), el proceso de mutación suele implementarse como un valor porcentual y se ha comprobado que el ajuste correcto del porcentaje de mutación es de vital importancia para el correcto funcionamiento del AG. En la mayoría de las implementaciones de AGs se suele emplear unos porcentajes de cruce y mutación estáticos que no varían durante el proceso. Sin embargo, se han obtenido buenos resultados modificando la probabilidad de mutación a medida que aumenta el número de generaciones.

Cruza

Se denomina técnica de cruzamiento a la forma de calcular el genoma del nuevo individuo en función de los genomas padres. El operador de cruza es fuertemente responsable de las propiedades del algoritmo genético, y determinará en gran medida la evolución de la población.

Después de seleccionar a los individuos que engendrarán a la siguiente generación, llega el momento de cruzarlos (crossover) entre ellos, y al igual que en la naturaleza, el proceso consiste en crear a los descendientes a partir del intercambio de material genético de sus padres. Existen dos técnicas principales; intercambio a partir de un sólo punto de cruce y a partir de dos puntos de cruce. Para ciertos problemas es necesario emplear una técnica de cruzamiento especializada que controle el proceso de intercambio genético evitando que se codifiquen en él soluciones inválidas, un ejemplo de problema que necesita un cruce controlado es el del problema del viajante que se tratará más adelante.

Independientemente de la técnica de cruce que se emplee, se suele implementar la reproducción en un AG como un valor porcentual que indica la frecuencia con la que se deben realizar los cruces, y los que no se crucen pasarán como réplicas de sí mismos a la siguiente generación (hay que tener también en cuenta que existe la posibilidad de que se produzca una mutación durante la replicación del código).

Esto nos lleva a una técnica muy importante desarrollada hace unos cuantos años conocida como elitismo [17]. Consiste en que el mejor individuo de la población permanece inalterable generación tras generación (ni siquiera se le aplica ningún tipo de mutación) hasta que aparece un individuo con un fitness mejor que lo sustituye. Su importancia reside en que de esta forma nunca se pierde la mejor solución encontrada hasta el momento. Existen diversas técnicas de cruzamiento [18]:

- *Cruza de un Punto:* Generar número aleatorio entero entre [2, n° de bit-1] para determinar el punto de cruce para cada par de padres a cruzar.

- *Cruza de Dos Puntos:* Se seleccionan aleatoriamente 2 puntos de cruce. Se copian los bits desde el inicio del padre 1 hasta el 1° punto de cruce. Los bits desde el 1° punto hasta el 2° punto del segundo padre. El resto es copiado del primer padre.

- *Cruza de n-puntos:* Los dos cromosomas se cortan por *n* puntos, y el material genético situado entre ellos se intercambia. Lo más habitual es un crossover de un punto o de dos puntos.

- *Cruza uniforme:* Se genera un patrón aleatorio de 1's y 0s, y se intercambian los bits de los dos cromosomas que coincidan donde hay un 1 en el patrón. O bien se genera un número aleatorio para cada bit, y si supera una determinada probabilidad se intercambia ese bit entre los dos cromosomas.

- *Cruzamiento especializado:* En algunos problemas, aplicar aleatoriamente la cruza crea cromosomas que codifican soluciones inválidas; en este caso hay que aplicar un cruzamiento que genere siempre soluciones válidas. Donde se utilizan operadores de cruzamiento por ejemplo el problema del viajante.

I.2 Algoritmo Genético Simple

Cuando se va a aplicar un Algoritmo Genético (AG) para resolver un problema, el primer movimiento que se debe hacer es codificar dicho problema en un cromosoma artificial. Los cromosomas artificiales pueden ser cadenas de unos y ceros, listas de parámetros o hasta códigos complejos, pero la clave que debemos tener en mente es que la maquinaria genética manipulará una representación finita de soluciones, no las soluciones por sí mismas. Otro componente que se debe tener en cuenta cuando intentamos resolver un problema es el procedimiento o los medios para discriminar las soluciones buenas de las malas. La idea es que algo debe determinar la aptitud hacia el propósito de una solución. Esa determinación la logra la función de evaluación o función de Fitness, que será usada por el AG para guiar la evolución de las generaciones futuras.

Una vez codificado el problema de manera cromosómica y teniendo los medios para discriminar las buenas soluciones de las malas, se está en condiciones de evolucionar soluciones para nuestro problema mediante la creación de una población inicial de soluciones posibles. Esta población puede ser creada aleatoriamente o utilizando algún conocimiento previo de buenas soluciones posibles, pero la idea principal es que se partirá la búsqueda desde una población y no desde un punto.

Con la población en su lugar, los operadores genéticos pueden procesar la población iterativamente para crear una secuencia de poblaciones que esperanzadamente contendrán más y mejores soluciones para el problema en cuestión.

Existe una amplia variedad de operadores utilizados en un AG, pero en general son selección, recombinación (o crossover) y mutación. En el Algoritmo 1.1 se ilustra la funcionalidad del algoritmo genético simple, los pasos más importantes son la generación de una población de soluciones, la búsqueda de la función objetivo y la aplicación de operadores genéticos.

Algoritmo 1.1 Algoritmo Genético Simple

```
BEGIN
    WHILE NOT Terminado DO
    BEGIN /* Producir generación*/
        FOR Tamaño población/2 DO
        BEGIN /* Ciclo Reproductivo*/
            Seleccionar dos individuos de la anterior generación,
            Para el cruce (probabilidad de selección proporcional a la
            Función de evaluación del individuo).
            Cruzar con cierta probabilidad los dos individuos
            Obteniendo dos descendientes.
            Mutar los dos descendientes con cierta probabilidad.
            Computar la función de evaluación de los dos
            Descendientes mutados.
            Insertar los dos descendientes mutados en la nueva
            Generación.
        END
        IF la población ha convergido THEN
            Terminado := TRUE
    END
END
```

I.3 Algoritmo Genético Paralelo

Un programa es paralelo si en cualquier momento de su ejecución puede ejecutar más de un proceso. Para crear programas paralelos eficientes hay que crear, destruir y especificar procesos así como la interacción entre ellos. Básicamente existen tres formas de paralelizar un programa [19]:

- *Paralelización de grano fino:* la paralelización del programa se realiza a nivel de instrucción. Cada procesador hace una parte de cada paso del algoritmo (selección, cruza y mutación) sobre la población común.

- *Paralelización de grano medio:* los programas se paralelizan a nivel de bucle. Esta paralelización se realiza habitualmente de una forma automática en los compiladores.

- *Paralelización de grano grueso:* se basan en la descomposición del dominio de datos entre los procesadores, siendo cada uno de ellos el responsable de realizar los cálculos sobre sus datos locales.

La paralelización de grano grueso tiene como atractivo la portabilidad, ya que se adapta perfectamente tanto a multiprocesadores de memoria distribuida como de memoria compartida. Existen dentro de la paralelización de grano grueso tres esquemas generales [20]:

- *Paralelismo en datos:* El compilador se encarga de la distribución de los datos guiado por un conjunto de directivas que introduce el programador. Estas directivas hacen que cuando se compila el programa las funciones se distribuyan entre los procesadores disponibles. Como principal ventaja presenta su facilidad de programación. Los lenguajes de paralelismo de datos más utilizados son el estándar *HPF (High Performance Fortran)* y el OpenMP [21].

- *Programación por paso de mensajes:* El método más utilizado para programar sistemas de memoria distribuida es el paso de mensajes o alguna variante del

mismo. La forma más básica consiste en que los procesos coordinan sus actividades mediante el envío y la recepción de mensajes.

Las librerías más utilizadas son por este orden la estándar MPI (*Message Passing Interface*) y PVM (*Parallel Virtual Machine*) [22].

- *Programación por paso de datos:* A diferencia del modelo de paso de mensajes, la transferencia de datos entre los procesadores se realiza con primitivas unilaterales tipo *put* - *get*, lo que evita la necesidad de sincronización entre los procesadores emisor y receptor. Es un modelo de programación de muy bajo nivel pero muy eficiente, aunque en la actualidad son muy pocos los fabricantes que los soportan.

Una de las principales ventajas de los AG es que permiten que sus operaciones se puedan ejecutar en paralelo. Debido a que la evolución natural trata con una población entera y no con individuos particulares, excepto para la fase de selección, durante la cual existe una competencia entre los individuos y en la fase de reproducción, en donde se presentan iteraciones entre los miembros de la población, cualquier otra operación de la población, en particular la evaluación de cada uno de los miembros de la población, pueden hacerse separadamente. Por lo tanto, casi todas las operaciones en los AG son implícitamente paralelas.

Se ha establecido que la eficiencia de los AG para encontrar una solución óptima, está determinada por el tamaño de la población. Por lo tanto, una población grande requiere de más memoria para ser almacenada. También se ha probado que toma mayor cantidad de tiempo para converger. Si n es el tamaño de la población, la convergencia esperada es $nlog(n)$.Al utilizar computadores en paralelo, no solamente se provee de más espacio de almacenamiento, sino también con el uso de ellos se podrán producir y evaluar más soluciones en una cantidad de tiempo más pequeño.

Debido al paralelismo, es posible incrementar el tamaño de la población, reducir el costo computacional y mejorar el desempeño de los AG. Probablemente el primer intento que se hizo para implementar los AG en arquitecturas en paralelo fue en 1985, por John Grefenstette [23].

Los primeros ensayos consistían en un paralelismo global. Esta aproximación trataba por paralelizar explícitamente las tareas paralelas implícitas de los AG secuenciales, por lo tanto la naturaleza de los problemas permanece invariable.

El algoritmo simplemente maneja una sencilla población en donde cada individuo podía combinarse con cualquiera de los otros, pero la generación de los nuevos hijos y/o su evaluación en paralelo. La idea básica es que los diferentes procesadores puedan crear nuevos individuos y computar sus aptitudes en paralelo, sin tener que comunicarse con los otros.

En este apartado se introducen tres maneras diferentes de explotar el paralelismo de los Algoritmos Genéticos, por medio de los denominados modelos de islas.

- *Modelos de islas:* La idea básica consiste en dividir la población total en varias subpoblaciones en cada una de las cuales se aplica un Algoritmo Genético. Cada cierto número de generaciones, se efectúa un intercambio de información entre las subpoblaciones, proceso que se denomina migración. La introducción de la migración hace que los modelos de islas sean capaces de explotar las diferencias entre las diversas subpoblaciones, obteniéndose de esta manera una fuente de diversidad genética. Cada subpoblación es una "isla", definiéndose un procedimiento por medio del cual se mueve el material genético de una "isla" a otra. La determinación de la, tasa de migración, es un asunto de capital importancia, ya que de ella puede depender la convergencia prematura de la búsqueda. Se pueden distinguir diferentes modelos de islas en función de la comunicación entre las subpoblaciones, ha sido utilizado por varios autores Whitley y Starkweather, Gorges-Schleuter, Tanese.

  Algunas comunicaciones típicas son las siguientes:

  - *Comunicación en estrella,* en la cual existe una subpoblación que es seleccionada como maestra en la Figura 1.2 se muestra esta comunicación (aquella que tiene mejor media en el valor de la función objetivo), siendo las demás consideradas como esclavas. Todas las subpoblaciones esclavas mandan sus h_1 mejores individuos ($h_1, > 1$) a la subpoblación maestra la cual a su vez

manda sus h_2 mejores individuos ($h_2 > 1$) a cada una de las subpoblaciones esclavas.

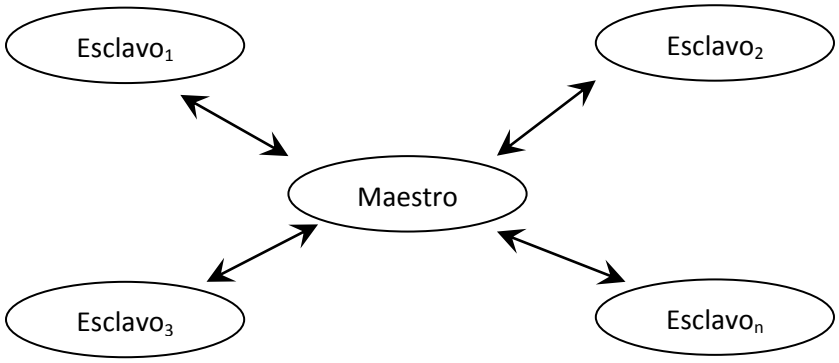


Figura 1.2 Modelo de Islas. Comunicación Estrella

- Comunicación en red, en la cual no existe una jerarquía entre las subpoblaciones en la Figura 1.3 se muestra este tipo de comunicación, mandando todas y cada una de ellas sus h_3 ($h_3 > 1$) mejores individuos al resto de las subpoblaciones.

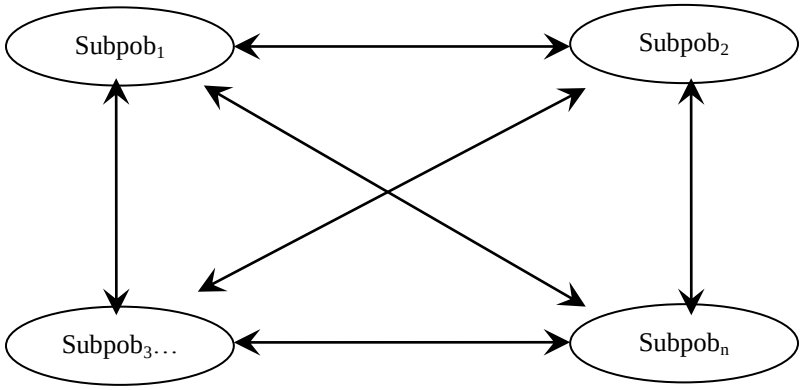


Figura 1.3 Modelo de Islas. Comunicación en Red.

- *Comunicación en anillo,* en la cual cada subpoblación envía sus h_4 mejores individuos ($h_4 > 1$), a una población vecina, efectuándose la migración en un

único sentido de flujo. En la Figura 1.4 se muestra el proceso de comunicación que existe entre subpoblaciones.

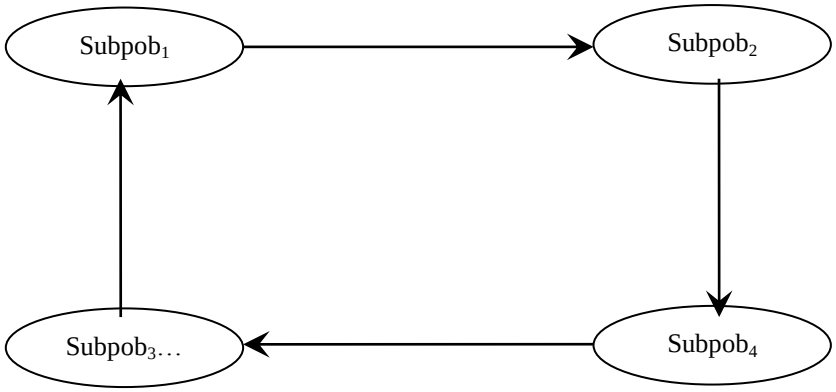


Figura 1.4 Modelo de Islas. Comunicación en Anillo.

La evaluación de la población en paralelo es simple de implementar. A cada procesador se le asigna un subconjunto de individuos para ser evaluados. Por ejemplo, en un computador de memoria compartida, los individuos pueden estar almacenados en la memoria, y cada uno de los procesadores puede leer los cromosomas asignados y puede grabar los resultados del cómputo de las aptitudes. Este método solamente supone que los AG trabajan con una generación actualizada de la población. Se necesita además, alguna sincronización entre las generaciones.

Generalmente, la mayor parte del tiempo de cómputo en un AG se gasta en la función objetivo. El tiempo que se gasta en el manejo de los cromosomas durante las fases de selección y recombinación es despreciable. En un computador de memoria distribuida se puede almacenar la población en un procesador "maestro", el cual es responsable de enviar los individuos a los otros procesadores “esclavos”. El "maestro", también es responsable por guardar los resultados de la evaluación. Una desventaja de esta implementación es que se pueden presentar cuellos de botella, cuando los esclavos están desocupados y sólo el maestro está trabajando.

Pero si se hace un buen uso del procesador maestro, se puede mejorar el factor de balance, distribuyendo dinámicamente los individuos a los procesadores esclavos, cuando ellos terminen sus trabajos.

Una segunda clase de AG paralelos consiste en dividir la población en subpoblaciones, y cada una de ellas ejecutarlas en un procesador. El intercambio entre subpoblaciones es posible por medio de un operador de "migración".

Se emplea el modelo de islas para mostrar como los AG se comportan como si el mundo fuera constituido por islas que se desarrollaran en forma independiente, unas de las otras. En cada una de las islas, la población es libre de converger hacia un óptimo diferente. El operador de migración permite extraer de las diferentes subpoblaciones las buenas características, para luego mezclarlas.

1.4 Mochila Multidimensional

El problema de la mochila multidimensional (MKP) es muy conocido, esencialmente, este problema consiste de una mochila con m condiciones limitadas por b_1, b_2, b_3,..., b_m, y n objetos, cada uno de ellos posee cierta utilidad c_i. El i-ésimo objeto pesa a_{ij} cuando es considerado para una posible inclusión en la restricción j-ésima de capacidad b_j. En términos formales, el problema puede ser matemáticamente formulado como se muestra en la Ec. 2

$$\text{Maximizar} \quad \sum_{i=1}^{n} c_i x_i$$

$$\text{Sujeto a} \quad \sum_{i=1}^{n} a_{ij}\, x_i \le b_j \qquad \forall j = 1,\ldots,m$$

$$x_i \in \{0,1\} \qquad \forall i = 1,\ldots,n$$

$$\text{donde} \quad x_i = \begin{cases} 1 & \textit{si el objeto i es incluido en la mochila} \\ 0 & \textit{de otro modo} \end{cases}$$

(2)

La importancia del problema MKP emerge tanto desde un punto de vista práctico como teórico. Este problema puede ser aplicado en muchas situaciones prácticas, algunas de ellas incluyen problema de patrones de cortes [24], reparto de abarrotes en vehículos con múltiples compartimentos [25], selección de proyectos [26] [27],presupuesto de capital [28], y asignación de procesadores y datos en sistemas computacionales distribuidos [29].

Adicionalmente, problemas tales como cobertura de conjuntos pueden ser reformulados como un problema MKP equivalente, mediante complementación de variables [30], en tanto que el conocido problema de ruteo de vehículos puede ser reducido al problema de la mochila 0-1 para algunas condiciones especiales [3].

1.5 Herramientas de Desarrollo

Para realizar nuestros programas nos apoyaremos en una serie de Herramientas como lo es las librerías GAlib y OOMPI que contienen las funciones y elementos necesarios para llegar a nuestro objetivo, así como la utilización de lenguaje UML para el diseño del programa.

1.5.1 GAlib y OOMPI

Para la implementación del sistema se ha utilizado el entorno de programación C++, este lenguaje fue elegido por su potencia y por ser orientado a objetos. Esta última característica

es fundamental, por la facilidad que brinda este paradigma de programación para escribir, extender, mantener y reutilizar código y las librerías GALib [5] y OOMPI [21] están escritas bajo este lenguaje.

GAlib es una biblioteca de funciones en C++ que proporciona al programador de aplicaciones un conjunto de objetos para el desarrollo de algoritmos genéticos. Usando Galib es posible resolver problemas de optimización mediante la construcción de una algoritmo genético, usando estructuras de datos y operadores estándar o específicos de selección, cruce y mutación, escalado y criterios de finalización

La biblioteca funciona principalmente con dos clases: un genoma y un algoritmo genético, cada instancia del genoma representa una única solución a su problema y el algoritmo genético objeto define la forma en que la evolución debería tener lugar.

El algoritmo genético utiliza una función objetivo que hemos definido para nuestro problema específico y esto es para determinar en cada genoma la supervivencia del mejor. Se utiliza el genoma operadores (incorporado en el genoma) selección y las estrategias de sustitución (incorporado en el algoritmo genético) para generar nuevos individuos.

Hay tres cosas para resolver un problema utilizando un algoritmo genético:

- Definir una representación
- Definir los operadores genéticos
- Definir la función objetivo

La librería nos ayuda con los primeros dos puntos suministrando diversos ejemplos y a partir de las cuales se puede construir nuestra representación y los operadores necesarios. La función objetivo es la propuesta para la solución del problema. Cuando se tiene la representación, operadores, y medición de los objetivos, se puede aplicar cualquier algoritmo genético para encontrar mejores soluciones al problema.

Cuando se utiliza un algoritmo genético para resolver un problema de optimización, se debe ser capaz de representar una única solución a su problema en una sola estructura de datos. El algoritmo genético de una población debe crear soluciones basadas en una muestra la

estructura de los datos que se proporcione. La biblioteca contiene cuatro tipos de genomas: GAListGenome, GATreeGenome, GAArrayGenome, y GABinaryStringGenome, derivadas de la base GAGenome.

La biblioteca contiene cuatro tipos de algoritmos genéticos. El primero es la norma «simple algoritmo genético" descrito por Goldberg [6]. Este algoritmo no utiliza la superposición de las poblaciones y el elitismo opcional. Cada generación del algoritmo crea una nueva población de individuos.

El segundo es un "estado de equilibrio algoritmo genético" que utiliza la superposición de las poblaciones. En esta variación, se puede especificar qué parte de la población deben ser sustituidos en cada generación. La tercera variación es el incremental algoritmo genético, en la que cada generación se compone de sólo uno o dos hijos. El cuarto tipo es el 'Deme' algoritmo genético. Este algoritmo se desarrolla en paralelo, se establece la forma en que algunos de los individuos de una población migran a otras.

Para las comunicaciones paralelas se eligió a la Interfaz de Paso de Mensajes (**MPI**, por sus siglas en inglés *Message Passing Interface*), la cual es el protocolo de comunicaciones paralelas más utilizado en la actualidad. MPI es un conjunto de rutinas que pueden ser utilizadas en programas escritos en C++, Fortran y Ada, para implementar comunicaciones de procesamiento paralelo.

Object Oriented MPI [21] (**OOMPI**) es una librería de la clase de especificación que encapsula la funcionalidad de MPI dentro de una clase funcional jerárquica, para proporcionar una simple, flexible e intuitiva interfaz.

Una de las implementaciones más extendidas de MPI es **LAM** (*Local Area Multicomputer*), desarrollado en sus orígenes por el Ohio Supercomputer Center y mantenido en la actualidad por el Open Systems Laboratory (OSL) en la Universidad de Indiana, Estados Unidos. Para el análisis y diseño del sistema se utilizó el proceso unificado de desarrollo de software, y el lenguaje de modelado UML.

1.5.2 UML (*Unified Modeling Language*)

Hoy en día, UML ("*Unified Modeling Language*") está consolidado como el lenguaje estándar en el análisis y diseño de sistemas de cómputo que permite modelar, construir y documentar los elementos que lo conforman. Mediante UML es posible establecer la serie de requerimientos y estructuras necesarias para plasmar un sistema de software previo al proceso intensivo de escribir código.

En la Figura 1.5 se muestra como ha venido evolucionando el lenguaje UML quien ha puesto fin a las llamadas "guerras de métodos" que se han mantenido a lo largo de los 90, en las que los principales métodos sacaban nuevas versiones que incorporaban las técnicas de los demás. Con UML se fusiona la notación de estas técnicas para formar una herramienta compartida entre todos los ingenieros de software que trabajan en el desarrollo orientado a objetos.

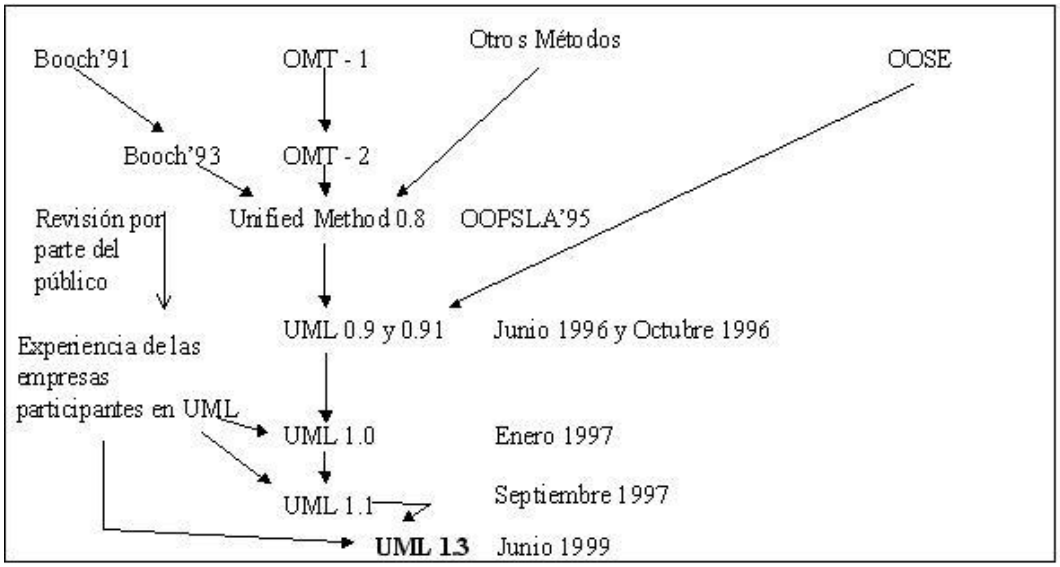


Figura 1.5 Evolución de UML

CAPÍTULO 2 ANÁLISIS DEL PROBLEMA A RESOLVER

En este capítulo se presentará un resumen de los diferentes métodos y estrategias que se han implementado para dar solución al problema de la mochila multidimensional con condiciones adicionales, se dará a conocer las diferentes herramientas que se utilizan para el planteamiento de los métodos de solución.

2.1 Planteamiento del Problema

Los problemas que pueden ser modelados usando variables binarias son muy frecuentes, por ejemplo, problemas de asignación de recursos, ubicación, etc. Cuando los coeficientes de las condiciones son todos positivos, a este problema se le conoce como el *Multidimensional Knapsack Problem (MKP)*, en particular, en el presente trabajo al modelo general se le agregan ciertas condiciones adicionales que obligan a que determinados pares de variables no tomen simultáneamente el valor 1, a este tipo de problema se le conoce como el ***Problema de la Mochila Multidimensional con Variables de Condición***.

Matemáticamente se enuncia en la Ec. 1:

$$\begin{aligned} &\max\ z = c^t x \\ &sa: Ax \le b \\ &A \in M_{mxn},\ b \in R^n,\ x \in (0,1)^n, c \in R^n, c_j < 0 \\ &x_i + x_j \le 1, i, j \subset \{1,..n\} \end{aligned} \quad \textbf{(1)}$$

El objetivo del presente trabajo es el desarrollo de un algoritmo genético secuencial y un algoritmo genético en paralelo para resolver este tipo de problemas, que aproveche al máximo la estructura del mismo, evitando con esto caer en soluciones parciales que no satisfacen el sistema de condiciones adicionales.

Si es posible obtener una solución factible, el algoritmo genera un esquema de la solución, es decir, coloca 1's en los alelos del cromosoma que llevan al algoritmo a una solución factible y 0's en caso contrario.

Cada restricción del Problema puede separarse de la forma que muestra la ecuación Ec. (2).

$$\sum_{j \in N^+} a_j x_j + \sum_{j \in N^-} a_j \leq a_0 \qquad (2)$$

Dónde:

N^+ denota el conjunto de índices con variables que tienen coeficiente positivo.

N^- denota el conjunto de índices de variables que tienen coeficiente negativo en cada restricción.

En el Algoritmo 2.1 se muestra el algoritmo que se utilizo para realizar la reducción de los genomas, eliminando los que no llegaban a una solución factible.

Algoritmo 2.1 Algoritmo de Simplificación.

BEGIN

Inicializar solución f_j = false // Paso 1

/*Análisis de factibilidad Paso 2*/

IF para alguna restricción se cumple que $\sum a_j > a_0, j \in N^-$ **THEN**

El problema no tiene solución factible. Fin del Algoritmo

ELSE

Análisis de Inactividad

/*Análisis de inactividad: Paso 3*/

$$\sum a_j \leq a_0, j \in N^+$$

IF para alguna restricción se cumple **THEN**

Eliminar la restricción.

IF f_j = false, **THEN**

Fijar Solución

ELSE

Fin del Algoritmo

/*Fijar Solución Paso 4 */

IF se cumple $a_j > a_0 - \sum a_k$ **THEN**

$x_j = 0$ f_j= true

IF Si se cumple que $-a_j > a_0 - \sum a_k$ **THEN**

$x_j = 1$ f_j= true.

IF f_j = true **THEN**

Análisis de factibilidad

ELSE

Fin del Algoritmo

El problema no tiene solución factible. Fin del Algoritmo

END

A partir de esto ya estamos en condiciones de formular tanto el algoritmo genético secuencial el cual se muestra en Algoritmo 2.2 que resuelve el problema MKP con condiciones adicionales

Algoritmo 2.2 Algoritmo Genético Secuencial.

BEGIN

Leer Archivo que contiene el problema MKP.

Aplicar el algoritmo de simplificación.

Fijar los parámetros necesarios para la aplicación del algoritmo genético: tamaño de la población, número de generaciones, porcentaje de mutación y el tipo de cruza.

Generar la población inicial.

WHILE NOT concluya algoritmo genético **DO**

Combinar genoma actual de la evolución y el esquema de la solución obtenida, mediante el algoritmo de simplificación.

Evaluar condiciones adicionales, si no las satisface descartar genoma, en caso contrario evaluar genoma con la función fitness.

Aplicar mutación y cruzamiento de acuerdo a los parámetros establecidos.

END

Mostrar los resultados obtenidos.

En el Algoritmo 2.3 se muestra Algoritmo genético paralelo que se utilizo para resolver el problema MKP con condiciones adicionales.

Algoritmo 2.3 Algoritmo Genético Paralelo

BEGIN

FOR Tamaño de nodos **DO**

BEGIN

Leer Archivo que contiene el problema MKP.

Aplicar algoritmo el algoritmo de simplificación.

Fijar los parámetros necesarios para la aplicación del algoritmo genético: tamaño de la población, número de generaciones, porcentaje de mutación y el tipo de cruza.

Generar la población inicial.

WHILE NOT Algoritmo genético no concluya **DO**

BEGIN

FOR Tamaño población **DO**

BEGIN

Combinar genoma actual de la evolución y el esquema de la solución obtenida, mediante el algoritmo de simplificación.

Evaluar genoma en función de las condiciones adicionales, si las satisface, evaluar el resto de las condiciones por medio de la evaluación de la función fitness en el algoritmo genético. Si no satisface las condiciones adicionales descartar genoma.

Cada 50 iteraciones (migrar):

Cada nodo esclavo *i* envía el mejor genoma factible al nodo esclavo *i+1*.

Cada nodo esclavo *i* recibe el genoma del *i-1*. Una vez recibido el genoma, lo mezcla entre su población, y continúa con la aplicación del algoritmo genético.

END

END

Enviar mejor solución al nodo 0.

END

Nodo maestro recibe las soluciones de los nodos esclavos. Reportar mejor solución.

Mostrar el resultado del problema MKP

END

2.2 Diagramas de Casos de Uso

Para la captura de los requerimientos del sistema se desarrollaron los diagramas de casos de uso requeridos, en la Tabla 2.1 se muestran los requerimientos generales para desarrollar el algoritmo genético secuencial y el algoritmo genético paralelo.

MKP	Resolver el problema de la mochila multidimensional con condiciones adicionales
Comentarios	Formas en las que se podrá resolver el problema: • Ejecutar el Programa que contiene el Algoritmo Genético Básico para resolver el problema de MKP con condiciones adicionales. • Ejecutar el Programa que contiene el Algoritmo Genético Paralelo que resuelve el MKP con condiciones Adicionales (ejecución bajo Linux con recursos de Red, LAM/MPI y OOMPI).

Tabla 2.1 Descripción del Problema MKP con condiciones adicionales

En la Figura 2.1 se muestra el caso de uso general

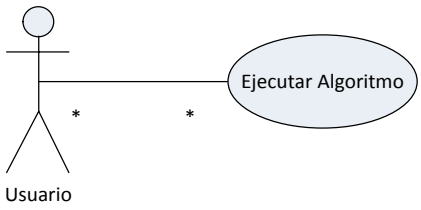


Figura 2.1 Caso de Uso General.

Diagramas de Casos de Uso Específico

En la Figura 2.2 se representa el Caso de Uso del algoritmo genético secuencial que se tomo como base para desarrollar el programa, como se ve es necesario instalar la librería GALib que se requieren para su ejecución como modificación del código base. Como entrada el usuario siempre deberá de indicar un documento de texto que deberá contener los datos que utilizará en su problema y es pasado como un argumento para la ejecución del programa.

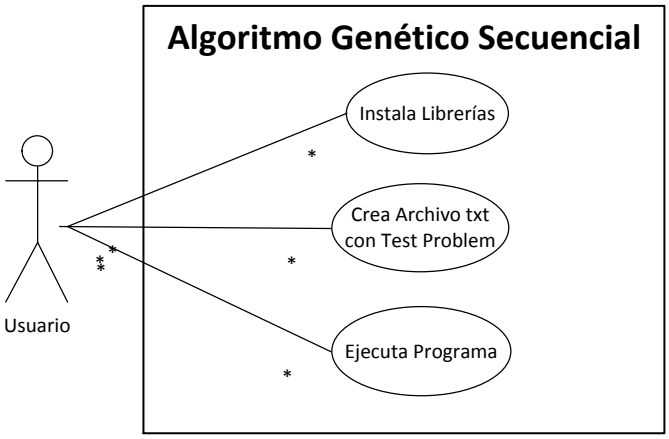


Figura 2.2 Caso de Uso: Ejecución del Algoritmo Genético Secuencial.

En la Figura 2.3 se muestra el caso de uso para la ejecución del algoritmo genético paralelo, para ejecutar el programa en la plataforma Linux es necesario instalar LAMP/MPI, para modificar el código se deberán tener instaladas las librerías: OOMPI, MPI y GALib.

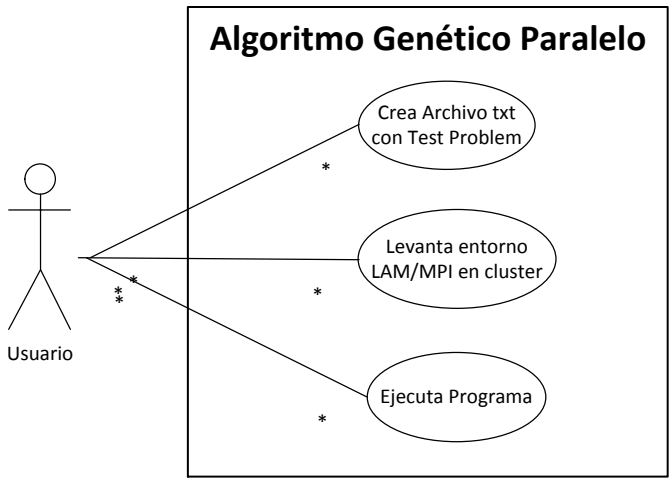


Figura 2.3 Caso de Uso: Ejecución del Algoritmo Genético Paralelo

Para el Algoritmo 2.2 y el Algoritmo 2.3 el usuario deberá crear en ambos casos un documento con los datos que utilizará para ejecutar el sistema que será envía en forma de parámetro y así generar la solución a dicho problema.

2.3 Diagrama de Clases General

Un diagrama de clases es un tipo de diagrama estático que describe la estructura de un sistema mostrando sus clases, atributos y las relaciones entre ellos. Los diagramas de clases son utilizados durante el proceso de análisis y diseño de los sistemas, donde se crea el diseño conceptual de la información que se manejará en el sistema, y los componentes que se encargaran del funcionamiento y la relación entre uno y otro.

En el Figura 2.4 se muestra el diagrama de clases del Algoritmos Genético Secuencial que resuelve el problema MKP con condiciones adicionales representando todas las clases que se generaron en el desarrollo del programa, así mismo se muestra la asociación con la clase GAGenome que es la clase que contiene todas las funciones necesarias para crear la evaluación de cada genoma y su factibilidad.

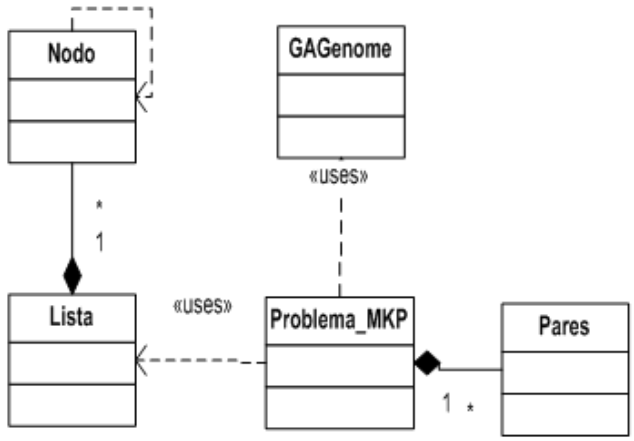


Figura 2.4 Diagrama General de Clases

En el Figura 2.5 se muestran las clases que contiene la librería OOMPI donde se visualizan una gran cantidad de recursos que se utiliza para programar el algoritmo.

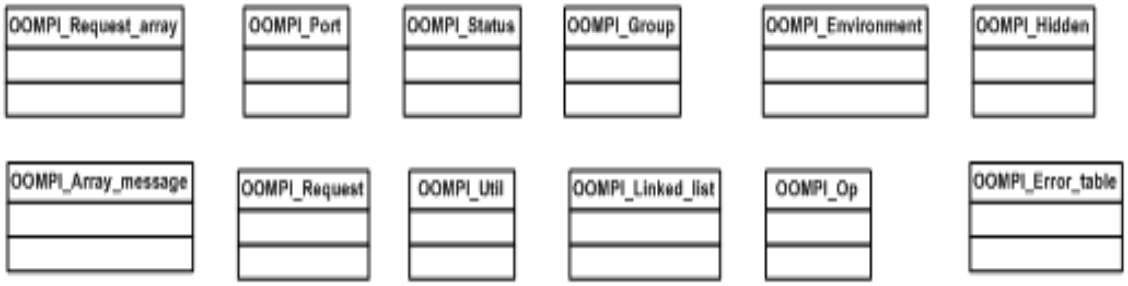


Figura 2.5 Diagrama de Clases librería OOMPI

En la Figura 2.6 se muestran las relaciones entre clases de la librería OOMPI para programar nuestro algoritmo.

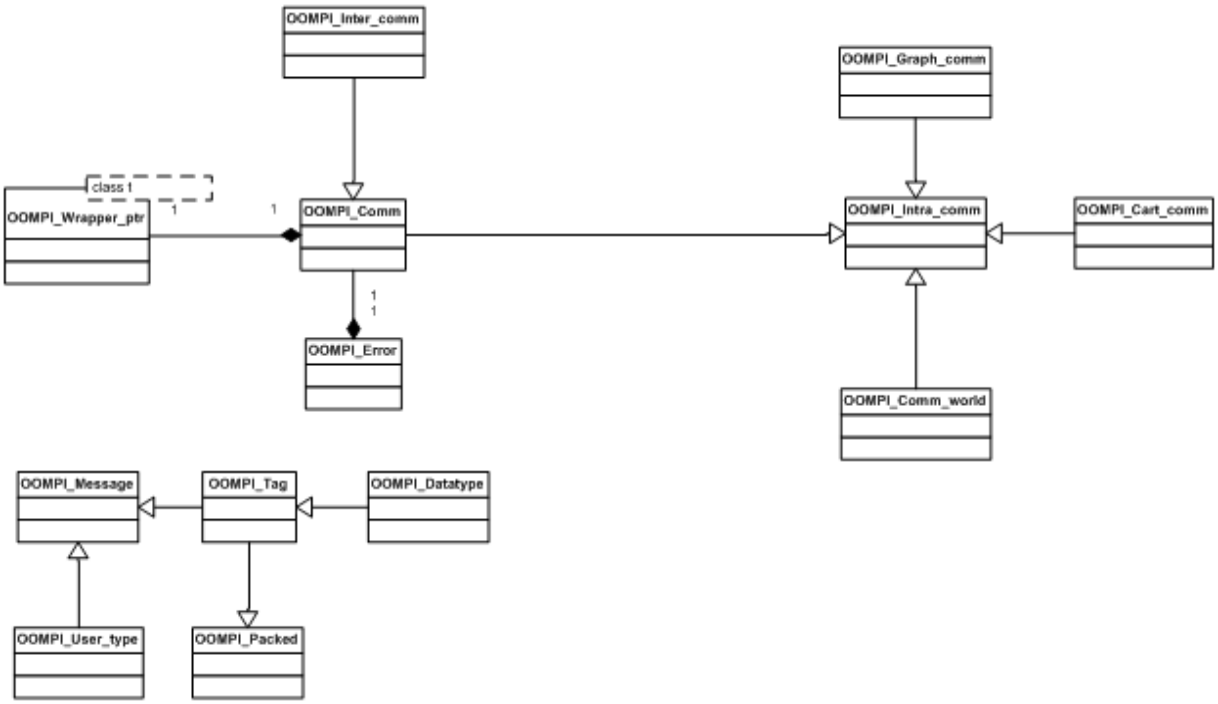


Figura 2.6 Diagrama de Clases Librería OOMPI

CAPÍTULO 3: DISEÑO DEL SISTEMA

En este capitulo se muestra una la estructura estática en términos de clases y relaciones así como una descripción de las clases utilizadas para generar nuestros algoritmos.

3.1 Diagrama de Clases Entidad

En el Figura 3.1 se muestran todas las clases generadas para nuestro algoritmo y resolver MKP con condiciones adicionales.

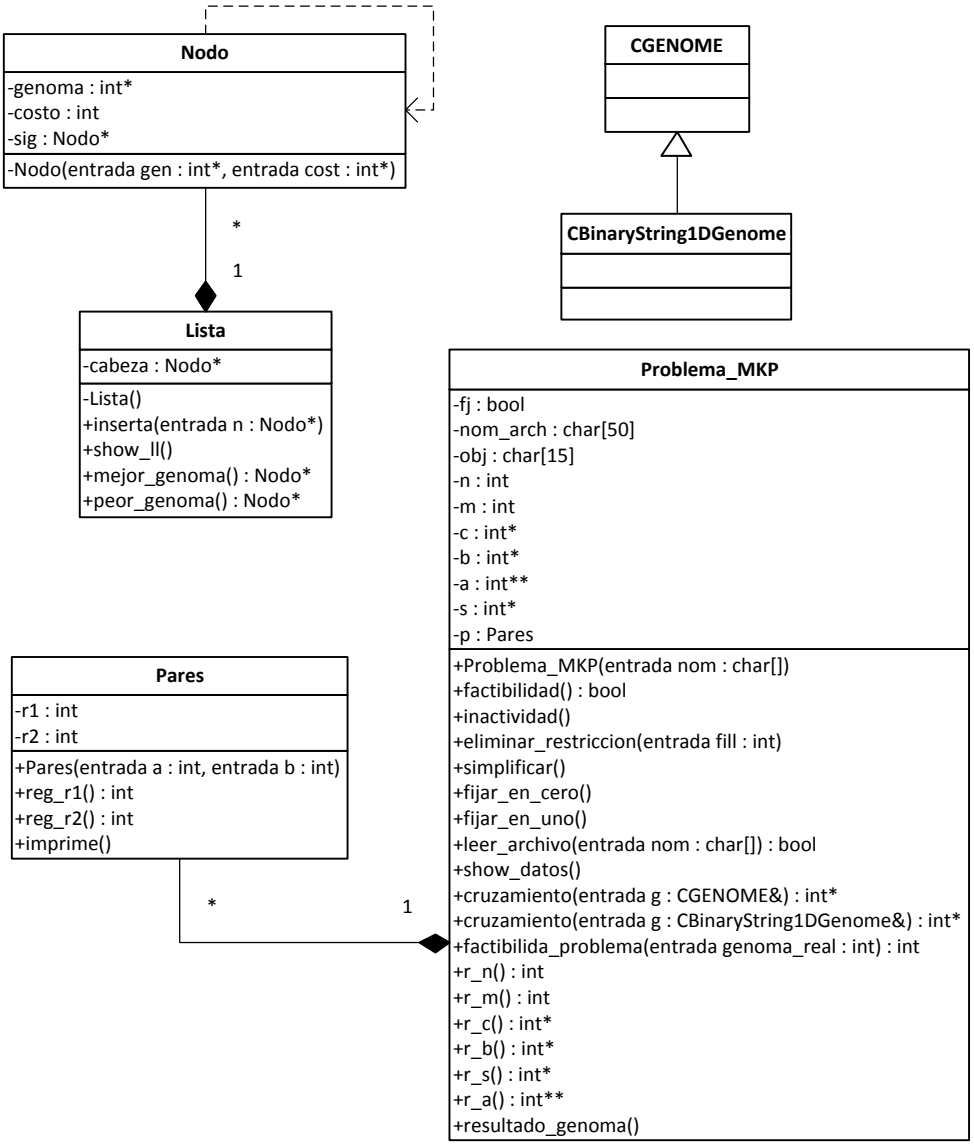


Figura 3.1 Diagrama de Clases Entidad del Programa.

A continuación se describen cada una de las clases que se realizarón.

Clase Nodo: Encargada de crear objetos tipo nodo que almacenara cada uno de los resultados *obtenidos* en la ejecución del programa.

Variables:

- *genoma*: variable tipo apuntador (int *), se utiliza para representar el mejor genoma factible para la solución de nuestro problema, los valores que se le pueden asignar son 1 y 0.

- *costo:* variable tipo entero, se utiliza para almacenar el costo real de nuestro genoma, mide la factibilidad de la solución.

- *sig*: variable de tipo apuntador a Nodo, se utiliza para almacenar la dirección del siguiente Nodo.

Procedimientos:

- *Nodo(int* gen, int cost):* Constructor de la clase, inicializa la variable y objetos con los parámetros que recibe, los cuales representan a un genoma y el costo del genoma.

Clase Lista: es la clase que representa a todos los elementos que son factibles al problema MKP con condiciones adicionales.

Variables:

- *cabeza:* variable apuntador tipo Nodo.

Procedimientos:

- *Lista():* constructor de la clase, además inicializa la variable cabeza (en NULL).
- *Inserta(Nodo n):* al recibir como parámetro a un nuevo Nodo va agregándolo a la lista generando una lista ligada, Si es el mejor costo lo almacena más cerca a la cabeza, de lo contrario lo almacena al final.

- *show_ll():* Muestra el contenido de la lista ligada generada.
- *mejor_genoma ():* función que regresa el mejor genoma factible para la solución del problema (el primer elemento de la lista). Se utilizara para obtener el resultado maximizado del problema MKP con condiciones adicionales, ofreciendo un tratamiento diferente a las condiciones adicionales.
- *peor_ genoma ():* función que regresa el peor genoma factible para la solución del problema (el último elemento de la lista). Se utilizará para obtener el resultado minimizado del problema MKP con condiciones adicionales.

Clase Pares: Las condiciones del problema están dadas por pares por eso fue preciso crear una clase que almacena aquellas variables y la restricción es que no pueden tomar el valor 1 simultáneamente.

Variables:

- *r1*: variable de tipo entero, que deberá tomar valor de 1 o 0.
- *r2*: variable de tipo entero, que deberá tomar valor de 1 o 0.

Procedimientos:

- *Pares(int a, int b):* Constructor de la clase, inicializa los valores r1=a y r2=b.
- *reg_r1():* Función que regresa el valor r1.
- *reg_r2():* Función que regresa el valor r2.
- *imprime()*: Función que muestra el contenido de las variables almacenadas.

Clase Problema_MKP: Clase encargada de leer el archivo que contiene el problema resolver y guardar los datos en las variables correspondientes, el objetivo del problema a resolver (minimizar o maximizar), es la encargada de mostrar el resultado del problema.

Esta clase necesita de un objeto Lista, para poder guardar las soluciones factibles del problema y en una de sus funciones recibe un objeto de tipo GAGenome o CBinaryString1DGenome, el cual lleva el genoma actual del proceso evolutivo, que es verificado para saber si ese objeto (genoma) lleva una solución factible del problema.

Variables

- *fj*: variable de tipo booleana, utilizada en la función de simplificación para detener el procedimiento de simplificación.
- *nom_arch*: variable de tipo carácter de tamaño 50, almacena el nombre del problema.
- *obj*: variable de tipo carácter de tamaño 15, almacena el tipo objetivo que se desea alcanzar puede contener los valores Max para Maximizar el problema o Min en caso de que se desee Minimizar.
- *n*: variable de tipo entero, almacena el número de variables del problema MKP con condiciones adicionales.
- *m*: variable de tipo entero, almacena el número de condiciones que tiene el problema MKP con condiciones adicionales.
- *c*: variable de tipo apuntador a entero, almacena la función objetivo del problema MKP con condiciones adicionales (tamaño n).
- *b*: variable de tipo apuntador a entero, almacena las variables libres de restricciones del problema MKP con condiciones adicionales (tamaño m).
- *a*: variable de tipo apuntador a un entero de tipo apuntador, almacena las funciones de restricción del problema MKP con condiciones adicionales, (tamaño m x n).
- *s*: variable de tipo apuntador a entero, almacena un esquema del genoma solución del problema. (tamaño n).
- *p*: variable de tipo apuntador, almacena los pares de condiciones.

Procedimientos:

- *Problema_MKP (char nom[50]):* Constructor de la clase, recibe como parámetro el nombre del archivo a leer y crea una lista ligada.
- *leer_archivo (char nom[50]):* función que lee el archivo que contiene al problema MKP con condiciones adicionales y guarda cada parte del problema en las variables correspondientes, esta función regresa *true* si fue posible leer correctamente el archivo *false* en caso contrario.
- *show_datos ():* muestra la información leída del archivo correspondiente al problema MKP con condiciones adicionales que se resolverá.
- *simplifica ():* función que realiza la simplificación del problema, llama a las funciones que componen el proceso de simplificación como son: factibilidad, inactividad, fijar_en_cero y fijar_en_uno.
- *factibilida ():* función que verifica si el problema MKP con condiciones adicionales tiene una solución posible. Regresa *true* si existe tal solución en caso contrario regresa *false.*
- *inactividad ():* función que evalúa si una condición es inactiva para la solución del problema, si es inactiva la restricción es eliminada de la matriz de restricción por medio de la función elimina_restriccion.
- *elimina_restriccion (int fil):* función que elimina la restricción y recibe como parámetro la fila de la matriz a eliminar.
- *fijar_en_cero()* y *fijar_en_uno():* funciones que fija a un esquema de solución del problema MKP con condiciones adicionales, este esquema es guardado en la variable *s*.
- *cruzamiento (CGenome& g)* y *cruzamiento (CBinaryString1DGenome& g):* funciones que reciben un genoma enviado desde la función fitness, cruza ese genoma con el esquema solución del problema y regresa el vector que contiene el resultado.
- *factibilidad_problema(int * genoma_real):* función recibe el genoma que se obtuvo como resultado de la función cruzamiento. Se encarga de verificar si ese genoma es una solución factible, es decir, una posible solución, para el problema MKP con condiciones adicionales, si lo es, almacena ese genoma junto con su costo en lista

ligada, cuando termina este procedimiento regresa el costo del genoma factible en caso contrario regresa el valor de la constante NO.

- *resultado_genoma():* esta función obtiene de la lista ligada el nodo que contiene la mejor solución al problema, esto depende de si el objetivo es Maximizar (el nodo más cercano a la cabeza) o Minimizar (el nodo que se encuentra al final de la lista).
- *r_n():* regresa el valor de la variable n.
- *r_m():* regresa el valor de la variable m.
- *r_c():* regresa el valor del arreglo c.
- *r_b():* regresa el valor del arreglo b.
- *r_s():* regresa el valor del arreglo s.
- *r_a():* regresa la matriz a.

3.2 Diagrama de Secuencia

Los diagramas de secuencia muestran el intercambio de mensajes, es decir, la forma en que se invocan en un momento dado. Los diagramas de secuencia ponen especial énfasis en el orden y el momento en que se envían los mensajes a los objetos.

En los diagramas de secuencia, los objetos están representados por líneas intermitentes verticales, con el nombre del objeto en la parte más alta. El eje de tiempo también es vertical, incrementándose hacia abajo, de forma que los mensajes son enviados de un objeto a otro en forma de flechas con los nombres de la operación y los parámetros.

En la Figura 3.2 y Figura 3.3 se tienen los diagramas de secuencia para los algoritmos que siguen los programas, a continuación se describirán los objetos que se utilizan.

- *p1*: Objeto de tipo *Problema_MKP*, se crea para resolver nuestro problema y contiene las funciones necesarias para leer y simplificar el problema, así como las funciones que procesan la información como cruzar los genomas y mostrar el resultado de este procedimiento.

- *ll*: Objeto de tipo Lista, guarda la lista y contiene los genomas factibles para ser solución del problema MKP con condiciones adicionales a resolver.

- *genome*: Objeto de tipo GA1DBinaryStringGenome, es el genoma que lleva en cada paso la evolución de un genoma diferente, usando la función fitness, donde se realiza el cruzamiento con el esquema solución.

- *ga*: Objeto de tipo GADemeGA, contiene el algoritmo genético encargado de evolucionar al genoma en cada paso, es decir, es un algoritmo evolutivo, emula varias poblaciones de genomas, contiene los métodos necesarios para el cruzamiento y mutación de genomas.

- *Nodo*: Objeto de tipo Nodo, este objeto es creado cada vez que se encuentre un genoma factible para la solución del problema y es enlazado en el objeto tipo lista ll.

La Figura 3.2 se muestra el diagrama de secuencia donde se muestra los pasos que lleva acabo nuestro algoritmo cabe mencionar que el resultado que genere depende de lo que se requiera si Maximizar (devolverá el mejor genoma) o Minimizar (devolverá el peor genoma).

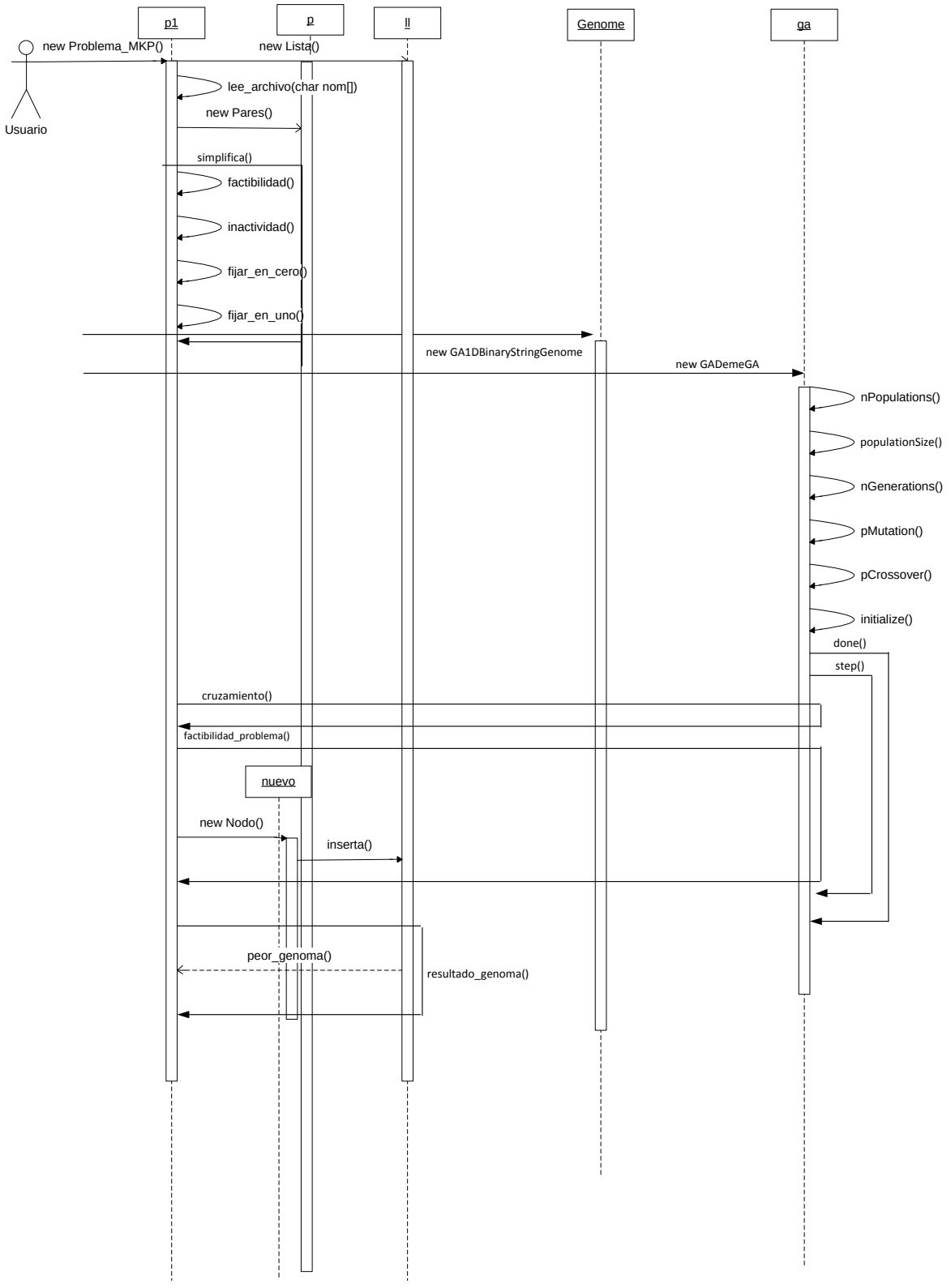


Figura 3.2 Diagrama de Secuencia del Algoritmo Genético Secuencial.

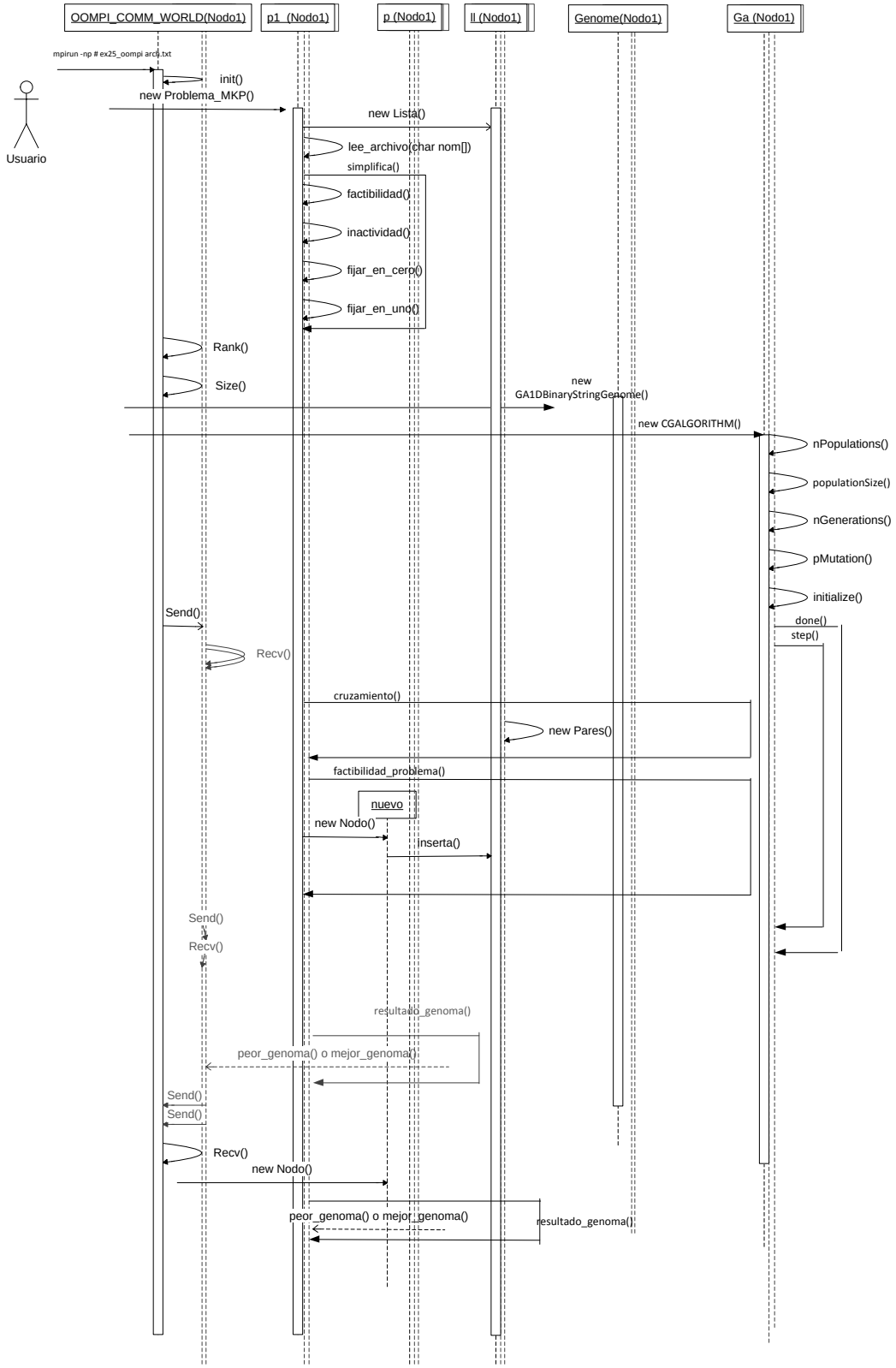


Figura 3.3 Diagrama de Secuencia del Algoritmo Genético Paralelo.

En la Figura 3.3 se agrega un nuevo objeto para usar las variables y funciones de la clase OOMPI y de la clase OOMPI_COMM_WORLD se usaran variables y funciones estáticas de clase, para inicializar el entorno MPI, y así enviar y recibir genomas entre nodos que ejecutan al programa paralelo. Además se muestra la iteración entre nodos, la función *Send()* envía entre nodos a los genomas y la función *Recv()* recibe a los genomas, y funcionan de la siguiente manera:

- Si se ejecuta por primera vez el algoritmo paralelo:
    - El Nodo1 envía su Genoma a cada uno de los nodos.
    - Los *n* Nodos reciben el genoma del Nodo1.
- Si se llevan *x* iteraciones del algoritmo, los nodos del 2 al *n* envían sus genomas de como se muestra en la Figura 3.4.

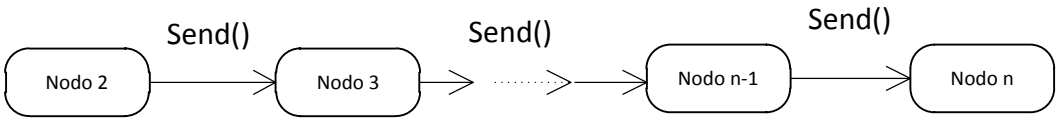


Figura 3.4 Envío de Genomas entre nodos

- Si llevan *x* iteraciones del algoritmo, los nodos del Nodo 3 al Nodo *n*-1 reciben los genomas como se ve en la Figura 3.5.

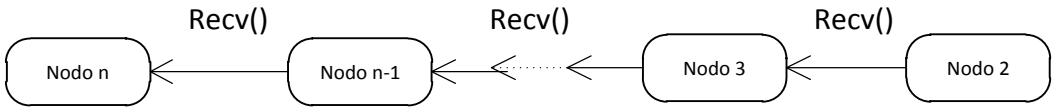


Figura 3.5 Reciben los Genomas entre nodos.

- Del Nodo 2 al Nodo *n* envían al Nodo 1 el mejor genoma.

3.3 Diagrama de Actividad

Los diagramas de actividad describen la secuencia de las actividades en un sistema. Los diagramas de actividad son una forma especial de los diagramas de estado, que únicamente (o mayormente) contienen actividades.

En la Figura 3.6 se muestra el diagrama que muestra los pasos que sigue el programa para construir un objeto *Problema_MKP*, el cual siempre requiere leer la información desde un archivo el cual es enviado como parámetro, en caso de no leer el archivo correctamente el sistema termina.

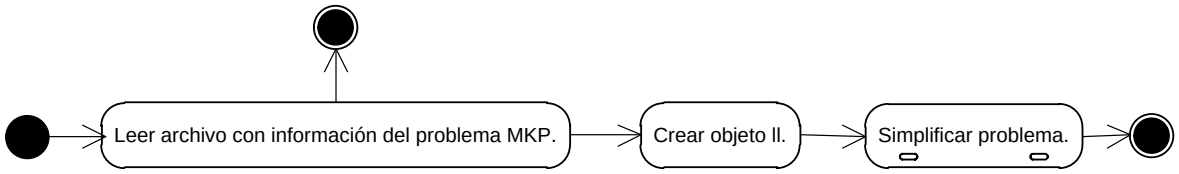


Figura 3.6 Diagrama de Actividad para Crear un objeto Problema_MKP

Cada vez que se requiera resolver un problema de MKP con condiciones adicionales el sistema sigue los pasos correspondientes, es importante definir el problema correctamente en el archivo para que se pueda dar una solución esperada. La solución del problema que estamos definiendo desde el archivo es dada por la Figura 3.7.

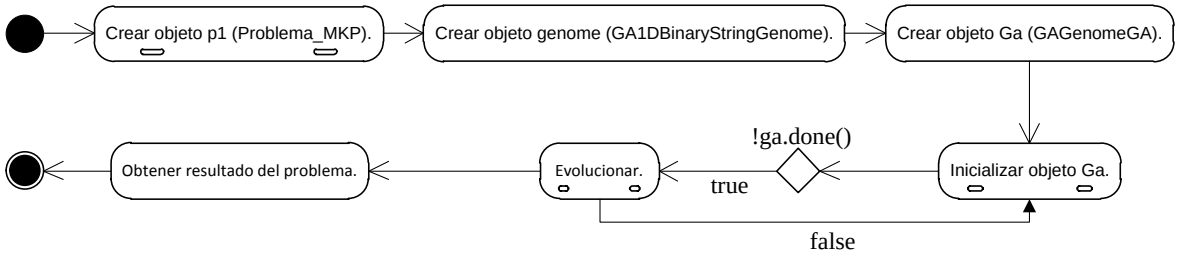


Figura 3.7 Diagrama de Actividad del Algoritmo Genético Secuencial.

En la Figura 3.8 se muestra la secuencia que sigue el Programa para establecer los valores del objeto GA obtenidos del archivo y así generar la solución esperada.

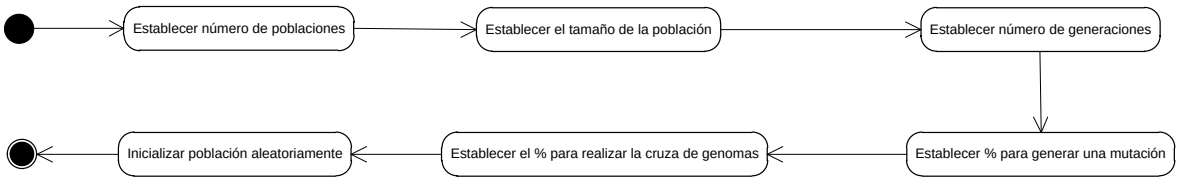


Figura 3.8 Diagrama de Actividad para Inicializar valores del objeto GA.

En la Figura 3.9 se muestra la secuencia que sigue al evolucionar el genoma y se deberá realizar en cada paso y se define por la función *step()*.

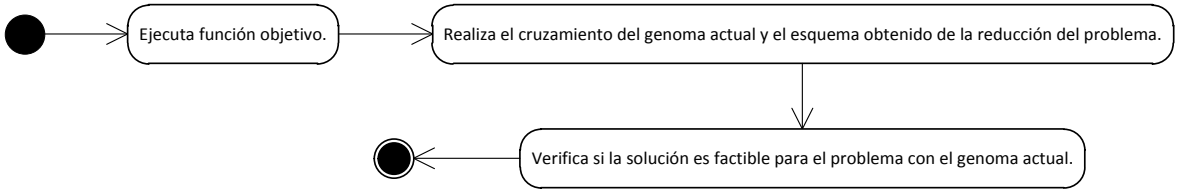


Figura 3.9 Diagrama de Actividad para Evolucionar.

En la Figura 3.10 se muestra los pasos que sigue el programa para realizar la simplificación de los genomas, debe obtener genomas factibles, si no se encuentra una solución factible el sistema termina.

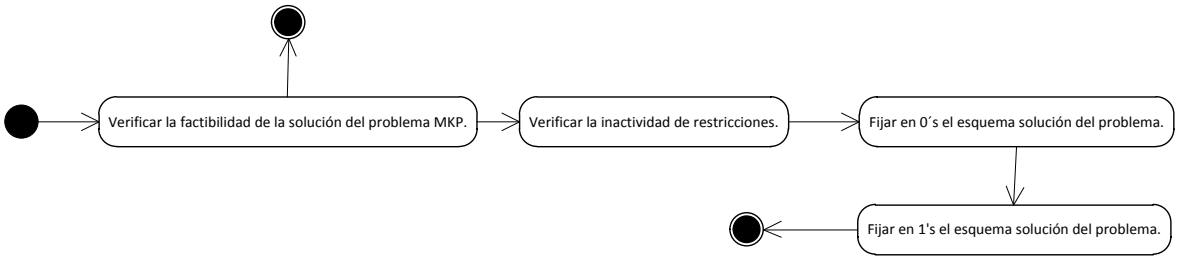


Figura 3.10 Diagrama de Actividad para Simplificar el problema.

En la Figura 3.11se muestra la secuencia que el algoritmo genético paralelo sigue en cada proceso y la comunicación entre nodos.

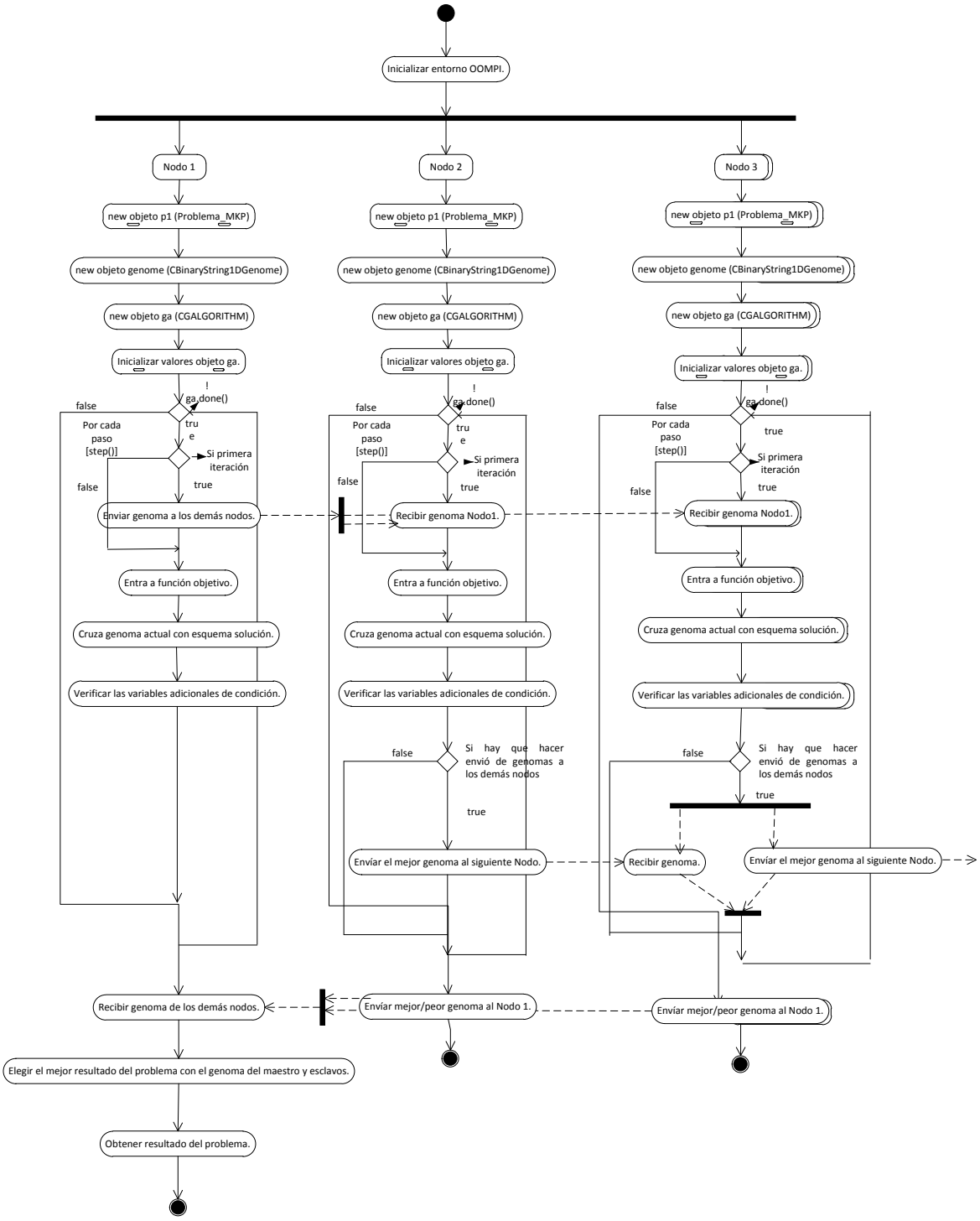


Figura 3.11 Diagrama de Actividad para Inicializar el entorno OOMPI.

CAPÍTULO 4: IMPLEMENTACIÓN Y ANÁLISIS DE RESULTADOS

Un diagrama de componentes representa cómo un sistema de software es dividido en componentes y muestra las dependencias entre estos componentes. Los componentes físicos incluyen archivos, cabeceras, bibliotecas compartidas, módulos, ejecutables, o paquetes. Los diagramas de Componentes prevalecen en el campo de la arquitectura de software pero pueden ser usados para modelar y documentar cualquier arquitectura de sistema.

En la Figura 4.1 se muestran el diagrama de componentes del algoritmo secuencial, el cual puede ser ejecutado en entorno Linux como Windows. En este trabajo solo se esta tomando la opción de ser ejecutado bajo el entorno Linux.

El programa principal lleva el nombre de **ex25y.C**, así mismo se generaron 2 archivos cabecera con el nombre de lee_datos_reducey.h y lista_ligada_genoma.h.

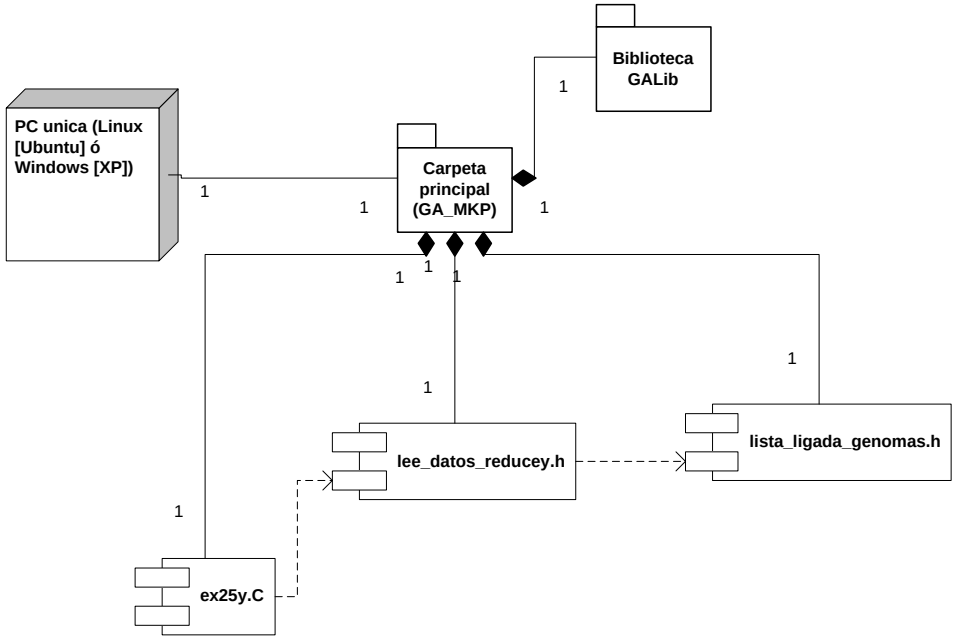


Figura 4.1 Diagrama de Componentes del Algoritmo Genético Secuencial

En la Figura 4.2 se muestra como se comportan los diferentes componentes del entorno OOMPI y como tienen relación con nuestro programa generado.

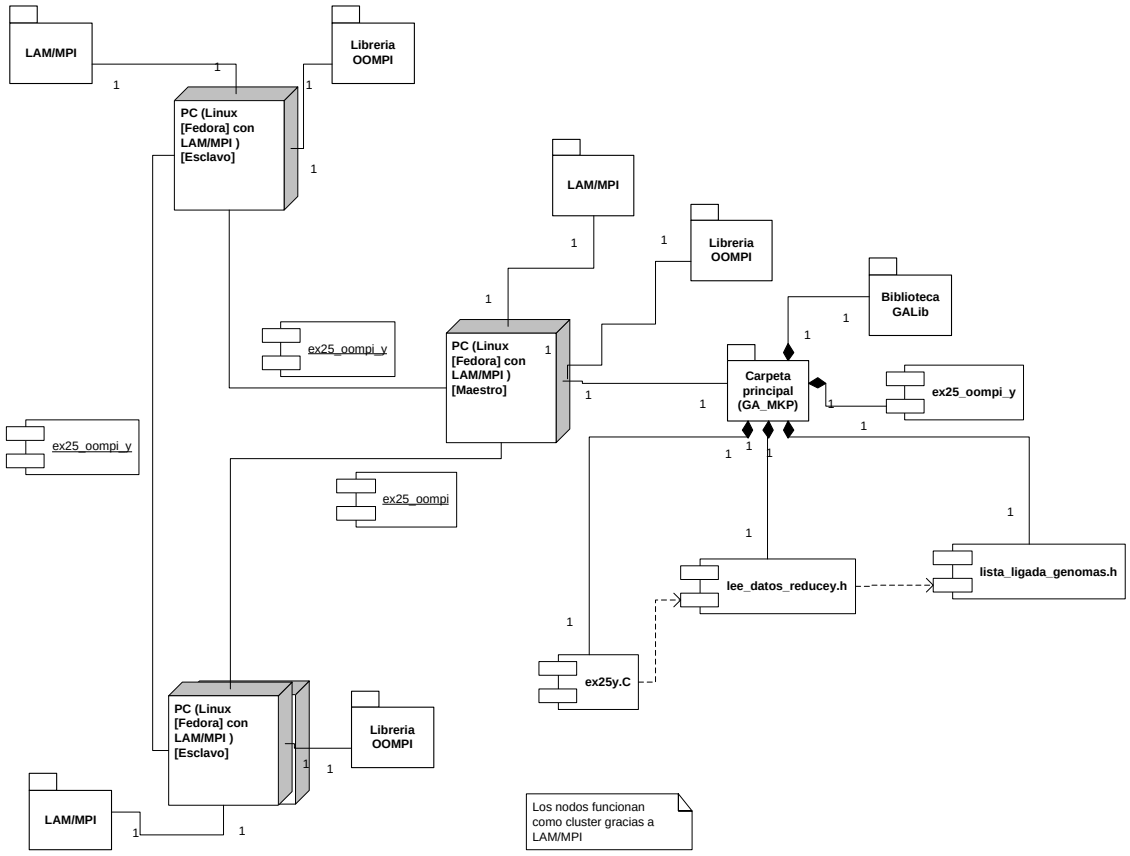


Figura 4.2 Diagrama de Componentes del Algoritmo Genético Paralelo.

A continuación se analizan y muestran los resultados obtenidos en algunos de los *test problems* propuestos [38], a los cuales se les han agregado un conjunto de condiciones adicionales para poder ser utilizados y probar el programa desarrollado.

Las pruebas, se realizaron bajo plataforma Linux como se comento anteriormente ya que es más eficiente, la característica principal que se debe tener si se requiere ejecutar el programa que contiene al Algoritmo Genético Secuencial es:

- Tener instalado el paquete *galib246.rar* y el compilador *GNU C/C++* (en caso de que se desee modificar partes del código).

Para instalar deberá seguir las instrucciones de la distribución de Linux pertinente para instalar un archivo **make.**

Algoritmo Genético Paralelo:

- Tener instalado la plataforma *LAM/MPI*, la biblioteca *OOMPI*, el compilador GNU C/C++ (en caso de que se desee modificar partes del código) y el paquete *Mibrary.rar*.

- Para instalar deberá seguir las instrucciones de la distribución de Linux pertinente para instalar un archivo **make**.

En Plataforma Windows 2000/XP/Vista sólo se podrá ejecutar el Algoritmo Genético Secuencial y se debe:

- Tener instalado el paquete *galib-2.4.6-gcc3.4.4-2siomek*.

- Para instalar deberá contar con el compilador *GNU C/C++*, el software *Dev-C++* (en caso de que se desee modificar partes del código) y tener actualizado el *path* de Windows con el compilador GNU para usar gcc/g++ en MS-DOS.

Cada vez que se requiere resolver un problema es necesario agregarlo a un archivo de texto plano.

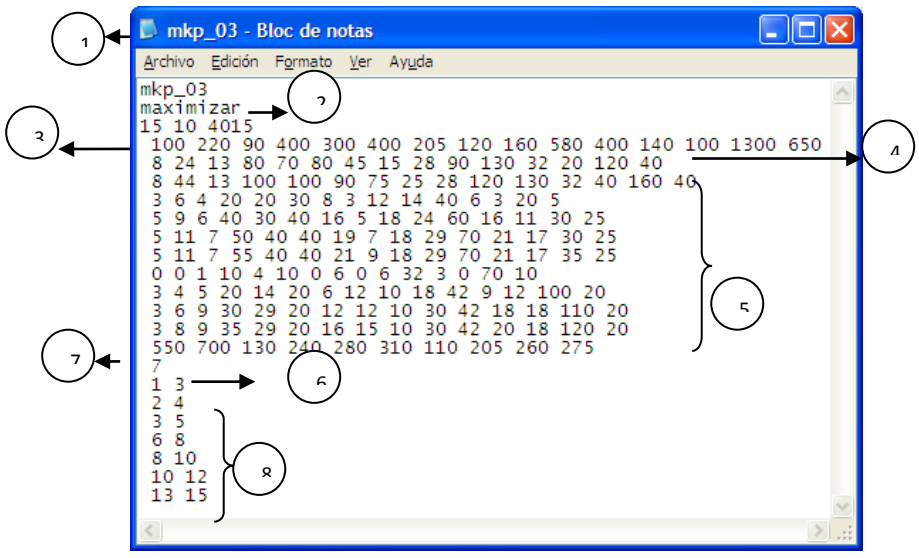


Figura 4.3 Sintaxis de Archivos .txt para generar solución.

En la Figura 4.1 se muestra la sintaxis que se debe de seguir para generar un archivo de entrada con nuestros test problema, dónde:

1. *Nombre problema*: Nombre del problema a resolver.
2. *Objetivo*: Indica cual es el objetivo se desea alcanzar en la solución del problema MKP. Existen los siguientes tipos:
    a. *maximizar:* indica que se debe de maximizar la función objetivo.
    b. *minimizar:* indica que se debe de minimizar la función objetivo.
3. *#n #m #optimo : donde*
    a. *#n:* tamaño del problema.
    b. *#m:* número de condiciones del problema.
    c. *#optimo:* un valor diferente de cero (0) si se conoce un valor óptimo al problema, en caso de no ser así, el valor será cero (0).
4. *Coeficientes:* función objetivo.
5. *Condiciones:* matriz de condiciones.
6. *Términos Independientes*.
    a. #de Pares de Condiciones adicionales

Pares de Condiciones: las condiciones adicionales siempre tendrán esta forma.

Ejecución del programa con el Algoritmo Genético

Para ejecutar el programa que contiene al Algoritmo Genético Secuencial:

Linux:

1. Entrar a línea de comandos de Linux (BASH).

2. Ir a la carpeta *galib246/*, su ubicación depende de cómo y en dónde fue instalada.

3. Ir a la carpeta *examples/*.

4. Ejecutar el programa de la siguiente forma:

./ags-mkp <nombre_archivo>.<ext> (./ags-mkp nombre_archivo.txt)

Cuando se ejecute el programa que contiene al Algoritmo Genético Paralelo:

1. Entrar a línea de comandos de Linux (BASH).

2. Ir a la carpeta *Mibrary/*, su ubicación depende de cómo y en dónde fue instalada.

3. Ir a la carpeta *ejemplos/*.

4. Iniciar la plataforma LAM/MPI a través del comando *lamboot* con los parámetros necesarios para crear el entorno paralelo.

5. Ejecutar el programa de la siguiente forma:

./mpirun –np # agp-mkp < nombre_archivo>.<ext> Donde –np # indica cuantos nodos son los que usará la aplicación en paralelo. (./mpirun –np 5 agp-mkp nombre_archivo.txt)

4.1 Interpretación de Resultados

Los problemas que sirvieron para validar el programa desarrollado son los *test problem* [1] en donde se muestra sólo la mejor solución reportada hasta el momento. Cabe mencionar que las pruebas realizadas con cada archivo que tenía un problema MKP fue resuelto 10 veces con cada uno de los algoritmos AGS y AGP, se midió el tiempo consumido para cada uno para reportar el mejor tiempo promedio obtenido y el valor de la función objetivo para cada uno de ellos.

Una vez que el programa termine su ejecución ya sea con el algoritmo genético secuencial o el algoritmo genético paralelo se arroja en la consola el vector solución del genoma resultante (con dominio de ceros y/o unos), el valor resultante de la maximización/minimización del problema MKP con Condiciones Adicionales, y el tiempo que le tomó al programa encontrar la solución mostrada.

En la Tabla 4.1 se muestra el tiempo promedio obtenido para cada *test problem* aplicando el algoritmo genético secuencial, así como el numero y las condiciones adicionales que se utilizaron para la solución (en la tabla se muestran como CA), los test problema mk_gk01 al mk_gk11 no pudieron ser resueltos en la PC, por falta de capacidad de memoria, provocando que se aborte la ejecución del programa. El número de poblaciones tomada fue de 100 elementos, así mismo se tomaron 100 individuos y un total de 50 generaciones.

Nombre del Problema	**N**	**M**	**Núm. de CA**	**Condiciones Adicionales (CA)**	**Mejor Solución**	**Tiempo promedio**
mkp_01	6	10	3	{[1,3][2,4][3,5]}	3800	0.020
mkp_03	15	10	7	{[3,5], [6,8], [8,10], [10,12], [13,15]}	3810	0.03
mkp_04	20	10	7	{[1,3], [2,4], [5,7], [8,10], [12,14], [13,15]}	5910	0.039

mkp_05	28	10	8	{[2,4], [5,7], [8,10], [11,13]}	12060	0.050
mkp_06	39	5	14	{[1,3], [3,5], [5,7], [20,22], [24,26]}	10171	0.052
mkp_07	50	5	15	{[1,3], [2,4], [5,7], [8,10], [9,11], [14,16], [18,20], [21,23], [27,29], [37,39] }	15580	0.06
mk_gk01	100	15	10	{[1,3], [2,4], [6,8], [7,9], [15,19], [20,22], [25,27], [25,27], [30,32], [39,41],[60,62] }		
mk_gk02	100	25	10	{[3,5], [7,9], [10,12], [15,17], [20,22], [30,32], [40,42], [43,45], [70,72], [79,81]}		
mk_gk03	150	25	10	{[65,67], [70,72], [76,78], [80,82], [92,94], [98,100], [112,114], [115,117], [124,126], [131,133]}		
mk_gk04	150	50	10	{[40,42], [45,47], [57,59], [60,62], [72,74], [90,92], [100,102], [114,116], [120,122],[124,126]}		
mk_gk05	200	25	10	{[2,4], [22,24], [30,32], [60,62], [95,97], [100,102], [135,137], [140,142], [170,172],[193,195]}		
mk_gk06	200	50	10	{[2,4], [11,13], [25,27], [42,44], [50,52], [60,52], [100,102], [112,114], [122,124],[190,192]}		
mk_gk07	500	25	10	{[1,3], [10,12], [50,60], [70,72], [90,92], [100,102],		

				[202,204], [305,307], [402,404], [490,492]}		
mk_gk08	500	50	10	{[10,12], [43,45], [100,102], [152,154], [200,202], [244,246], [300,302], [380,382], [400,402], [498,500]}		
mk_gk09	1500	25	10	{[1,3], [90,92], [120,122], [200,202], [433,437], [809,811], [1000,1002], [1300,1302],[1456,1458],[1498,1500]}		
mk_gk10	1500	50	10	{[100,102], [233,235], [456,458], [573,575], [678,680], [784,786], [900,902], [1004,1006],[1111,1113],[1452,1454]}		
mk_gk11	2500	100	10	{[300 302], [410 412], [521 523], [800 802], [940 942], [1000 1002], [1340 1342], [1800 1802], [2000 2002], [2400 2402]}		

Tabla 4.1 Resultados obtenidos para los *Tests Problems*, aplicando el algoritmo genético secuencial.

En la Tabla 4.2 se muestran el genoma con la solución que se obtuvo al aplicar el algoritmo genético secuencial, se encuentran marcados con * las soluciones que no se obtuvieron.

Nombre del Problema	**Resultado Genoma**
mkp_01	0 1 1 0 0 1
mkp_03	1 1 0 0 1 1 0 0 1 1 0 0 1 1 1
mkp_04	1 1 1 0 0 0 0 1 1 1 0 0 0 1 1 0 0 1 1 1
mkp_05	1 0 0 1 1 0 0 1 1 1 0 0 1 1 1 1 0 0 1 0 1 1 1 0 0 1 0 1
mkp_06	1 0 1 1 0 0 0 1 1 0 1 1 0 0 1 1 1 0 0 0 1 0 1 1 0 1 1 0 0 0 1 1 1 1 1 0 1 1 1
mkp_07	1 1 0 0 1 0 0 1 0 1 1 0 1 0 0 0 1 0 1 0 0 0 1 1 1 1 0 0 1 1 1 1 1 0 0 1 1 1 0 0 1 1 1 0 0 0 1 0 0 1
mk_gk01	*
mk_gk02	*
mk_gk03	*
mk_gk04	*
mk_gk05	*
mk_gk06	*
mk_gk07	*
mk_gk08	*
mk_gk09	*
mk_gk10	*
mk_gk11	*

Tabla 4.2 Solución obtenida, aplicando el algoritmo genético secuencial.

En el Algoritmo genético paralelo la representación de los datos se tiene de la siguiente manera:

de Individuos por Población:

Si n <= 100→ 250 elementos, o si n > 100 → n*4 elementos

de Generaciones:

Si n<= 100 → 180 generaciones, o si n > 100 → ((n*3)/2) generaciones

de nodos de *cluster*: 2, 4,6

En la Tabla 4.3 se muestran los resultados del Genoma en el AGP, desde el problema mkp_01, hasta el mk_g06, es esta ocasión si fue posible llegar a un genoma solución.

Nombre del Problema	**Mejor Genoma**
mkp_01	0 1 1 0 0 1
mkp_03	1 1 0 1 0 1 1 0 1 1 0 0 0 1 1
mkp_04	1 0 0 0 0 1 0 0 1 0 0 0 0 1 1 1 1 1 1 1
mkp_05	1 1 1 0 0 0 0 0 1 0 0 0 0 1 1 1 1 1 0 1 1 1 1 0 1 1 1 1
mkp_06	1 1 0 1 0 1 0 0 0 0 1 1 0 0 1 1 1 0 1 0 0 1 1 0 1 0 1 1 1 1 1 1 1 0 0 1 0 1 1 1
mkp_07	0 0 1 1 0 1 1 1 1 0 0 1 1 0 1 1 1 0 0 1 0 1 1 0 1 0 1 1 0 1 0 1 0 0 1 1 0 1 1 1 1 1 1 1 0 1 0 1 1 0 0
mk_gk01	0 1 1 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 1 0 1 1 0 0 1 0 1 1 0 0 0 1 0 0 0 0 1 1 0 0 0 0 0 1 0 0 1 0 1 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 0 0 1 1 1 1 0 0 1 0 0 1
mk_gk02	1 1 0 0 1 1 0 1 0 1 1 0 0 1 0 0 0 0 1 0 1 0 1 1 1 1 0 0 1 0 0 1 0 0 1 1 1 1 0 0 1 0 0 0 0 1 1 1 1 0 0 0 1 1 0 0 1 1 0 0 0 1 1 0 0 1 1 0 1 0 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 0 0 0 1 0 0 0 1 0 0 1 1 0 0 0
mk_gk03	0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 0 0 1 0 1 1 0 0 0 1 1 1 1 1 0 1 1 0 1 1 1 0 0 0 1 1 1 0 1 1 0 1 1 0 0 0 1 0 1 1 1 0 0 1 1 0 1 0 1 0 0 0 0 0 1 0 1 0 1 0 1 0 0 1 1 0 0 1 0 0 0 1 1 1 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 1 0 1 1 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0
mk_gk04	1 1 1 0 0 1 0 0 0 1 0 0 1 1 0 0 0 1 1 1 0 0 1 0 1 1 1 0 1 0 1 0 1 1 1 1 0 1 1 1 0 0 1 1 1 0 0 0 0 0 1 1 1 0 0 1 1 1 0 0 0 0 0 0 0 0 1 1 1 0 0 1 1 1 0 1 0 0 1 0 0 0 1 0 1 1 0 1 1 1 0 1 1 1 1 0 1 0 0 1 0 0 0 0 1 0 1 1 0 1 1 0 0 1 0 1 1 1 0 1 1 0 0 0 1 0 0 1 0 0 1 0 1 0 0 0 1 1 1 1 1 0 0 0 1 1 1 1 0

mk_gk05	1 0 0 0 1 1 1 1 0 1 0 0 0 0 1 1 0 1 0 0 0 0 0 0 0 1 1 1 0 1 1 1 1 1 1 1 1 1 0 0 0 1 0 1 1 1 0 1 1 0 0 0 0 1 1 0 1 1 0 0 0 0 1 0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 1 1 0 1 0 1 0 1 1 1 0 1 0 0 0 1 0 0 1 1 1 1 0 1 1 0 0 1 1 1 1 1 0 1 0 0 1 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 0 0 1 1 0 0 1 1 0 1 1 0 1 0 1 0 1 1 0 0 1 0 0 0 1 1 0 1 1 1 0 0 1 0 0 1 1 0 0 0 1 0 1 1 1 1 1 1 1 0 0 0 0 1 0 1 1 0 1 0 1 0 1
mk_gk06	0 0 1 0 1 0 1 1 0 1 0 1 0 0 1 1 1 0 0 1 0 0 1 1 0 1 1 1 1 0 0 1 0 1 1 1 0 0 1 0 1 0 0 0 0 0 0 1 0 0 1 0 0 0 1 1 1 0 1 0 1 0 0 0 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 1 1 0 0 1 0 1 0 1 0 1 1 1 1 0 1 0 1 0 0 0 1 0 1 0 1 1 0 1 0 1 1 1 1 0 1 1 1 0 0 1 0 0 1 0 0 1 0 1 1 0 1 0 1 1 0 1 0 1 0 1 0 0 1 1 0 1 1 1 1 0 1 0 0 0 0 1 0 0 0 1 1 0 0 1 0 1 1 1 0 0 0 1 0 0 0 1 1 0 0 0 0 0 0 0 1 1 1 0 1 1 1 0 0 1 1 1 1 1 1 0

Tabla 4.3 Solución de los test problema, aplicando el algoritmo genético paralelo.

En la tabla 4.4 se muestran los resultados obtenidos tanto por al algoritmo genético secuencial, como con el algoritmo genético paralelo, para los *tests problems*, de los cuales se muestra la mejor solución reportada hasta el momento.

Nombre del Problema	**Mejor Solución Reportada**	**Mejor Solución AGS**	**Tiempo promedio**	**Mejor Solución AGP**	**Tiempo promedio**
mkp_01	3800	3800	0.02	3800	0.083
mkp_03	4015	3810	0.03	4015	0.143
mkp_04	6120	5910	0.039	6120	0.203
mkp_05	12400	12060	0.05	12400	0.296
mkp_06	10551	10171	0.052	10551	0.353
mkp_07	16226	15580	0.06	16226	0.456
mk_gk01	3641	-	-	3641	0.84
mk_gk02	3818	-	-	3818	0.74

mk_gk03	5486	-	-	5486	1.636
mk_gk04	5604	-	-	5604	2.323
mk_gk05	7424	-	-	7424	2.596
mk_gk06	7450	-	-	7450	2.576
mk_gk07	18521	-	-	18521	4.616
mk_gk08	18285	-	-	18285	9.356
mk_gk09	56040	-	-	56040	6.916
mk_gk10	55693	-	-	55693	17.393
mk_gk11	93197	-	-	93197	43.146

Tabla 4.4 Tiempo promedio obtenido con AGP y AGS.

En la Figura 4.4 se muestra una grafica comparativa de los tiempos promedios obtenidos de ambos algoritmos, cuando se trato de evaluar los problemas mk_gk01 al mk_gk11 el algoritmo genético secuencial ya no encontró solución al problema MKP con condiciones adicionales y el AGP pudo encontrar una solución si no la mejor solución reportada hasta el momento una solución muy cercana a ella, cabe señalar que los resultados son variables y depende de la capacidad de procesamiento del equipo de computó.

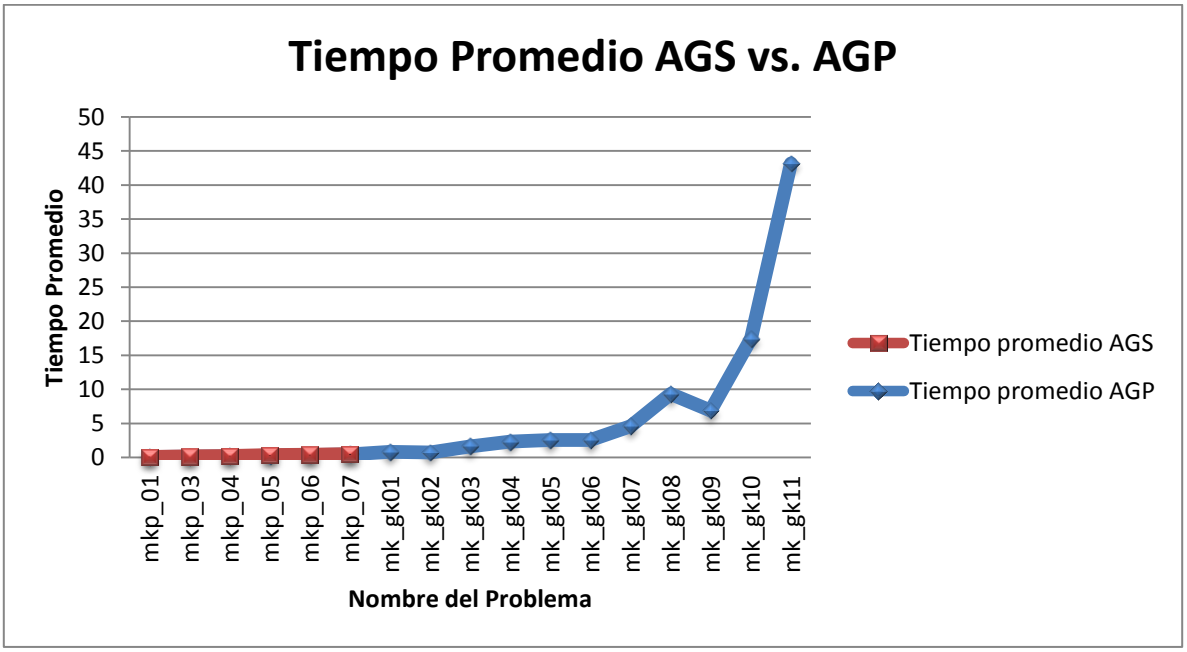


Figura 4.4 Grafica que muestra el Tiempo Promedio AGS vs. AGP

En la tabla 4.5 se muestran los tiempos promedio utilizando 2, 4 y 6 nodos del *cluster*, para la ejecución del algoritmo genético paralelo.

Nombre del Problema	**Mejor Solución**	**2 Nodos**	**4 Nodos**	**6 Nodos**
mkp_01	3800	0.09	0.08	0.08
mkp_03	4015	0.14	0.14	0.15
mkp_04	6120	0.2	0.22	0.19
mkp_05	12400	0.29	0.3	0.3
mkp_06	10551	0.32	0.37	0.37
mkp_07	16226	0.46	0.5	0.41
mk_gk01	3641	0.82	0.9	0.8
mk_gk02	3818	1.17	0.1	0.95
mk_gk03	5486	1.64	1.64	1.63
mk_gk04	5604	2.39	2.33	2.25
mk_gk05	7424	3.18	2.71	1.9
mk_gk06	7450	2.82	2.78	2.13
mk_gk07	18521	3.94	4.76	5.15
mk_gk08	18285	9.51	9.4	9.16
mk_gk09	56040	7.74	6.27	6.74
mk_gk10	55693	17.99	15.51	18.68
mk_gk11	93197	63.88	35.84	29.72

Tabla 4.5 Tiempos promedios obtenidos utilizando 2,4 y 6 Nodos.

En la Figura 4.5 se muestran los tiempos obtenidos utilizados con el algoritmo genético secuencial y el algoritmo genético paralelo con 2, 4 y 6 nodos.

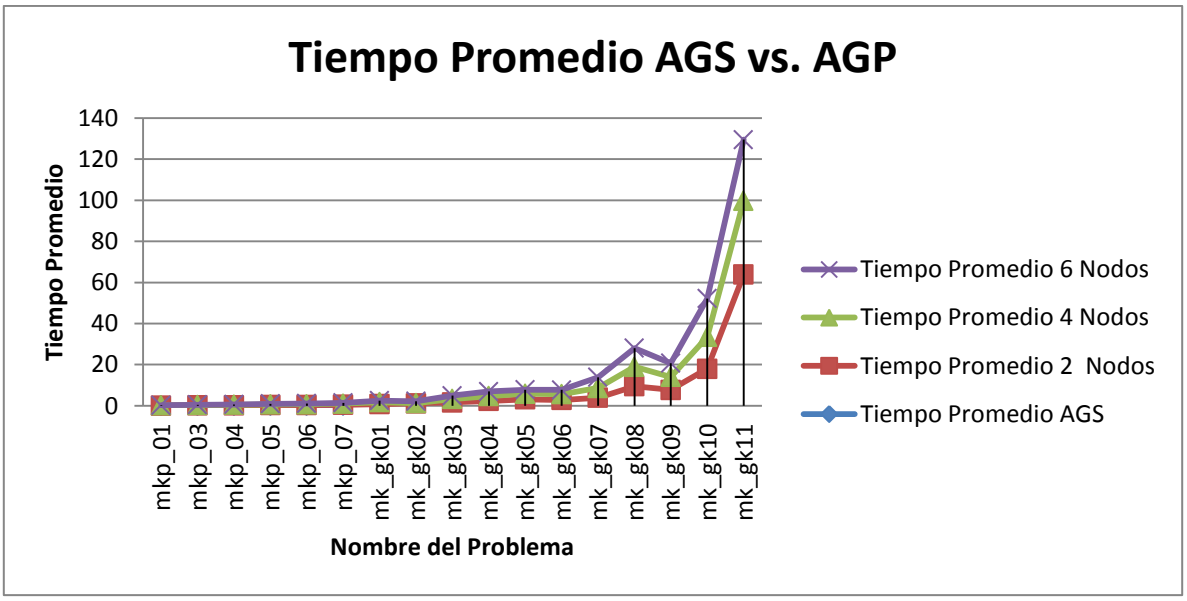


Figura 4.5 Grafica que muestra el comparativo de tiempo promedio.

Los programas generados fueron probados con los *test problems* propuestos y se validaron con la mejor solución publicada hasta el momento. Por los resultados obtenidos puede apreciarse que el algoritmo genético paralelo puede aumentar el tiempo por el intercambio de mensajes entre nodos pero puede obtenerse una mejor aproximación o incluso obtener el resultado hasta ahora publicado con una mayor población generada ya que al realizar la migración y mezclar los mejores individuos de una población a otra, permite enriquecer el espacio de búsqueda y por lo tanto una mayor rapidez en obtener la solución final.

CONCLUSIONES

Los objetivos de la presente tesis se cumplieron.

1. Se implementó un algoritmo genético secuencial y un algoritmo genético paralelo utilizando la librería GALIB.

2. Se validaron ambos sistemas desarrollados con los *tests problems* reportados en la literatura. A cada problema se le agregaron ciertas condiciones que no afectaban las soluciones finales, pero si podían validar el módulo de simplificación.

3. El algoritmo genético secuencial se comportó como era de esperarse, ya que no se lograron ejecutar problemas ni de mediana restricciones.

4. La implementación del algoritmo genético paralelo, consideró un esquema de migración muy simple, sólo se migran los mejores individuos a los vecinos más cercanos. La potencialidad de estos resultados no puede medirse, ya que al tener las condiciones adicionales se restringe mucho el espacio de factibilidad del mismo.

5. Se ejecutó el algoritmo paralelo con 2, 4 y 6 nodos del *cluster*, permitiendo corroborar que para esta clase de problemas lo más conveniente es usar 4 nodos, pues ya el tiempo de comunicación con el esquema de migración utilizado aumenta considerablemente.

Como trabajo futuro se extenderá la programación del algoritmo genético paralelo a una arquitectura multi-kernel utilizando Open Computing Language (OpenCL).

REFERENCIAS

[1] «Librería de Test problems para el MKP,» [En línea]. Available: http://people.brunel.ac.uk/~mastjjb/jeb/orlib/mknapinfo.html. [Último acceso: 2012].

[2] C. R. Darwin, On the Origin of Species by Means of Natural Selection, London: Murray, 1859.

[3] H. Christos, Papadimitriou y S. Kenneth, Optimización combinatoria: Algoritmos y complejidad, Englewood Cliffs, NJ: Prentice Hall, 1982.

[4] R. E. Burkard, S. E. Karisch y F. Rendl, «Pen Engineering,» [En línea]. Available: http://www.seas.upenn.edu/qaplib/. [Último acceso: 2011].

[5] L. GALib., «versión 2.4.6,» [En línea]. Available: http://lancet.mit.edu/ga/. [Último acceso: 2011].

[6] D. E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley, 1989.

[7] J. E. Beasley, «Brunel University West London,» [En línea]. Available: http://people.brunel.ac.uk/~mastjjb/jeb/orlib/mknapinfo.html. [Último acceso: 2009].

[8] M. Mitchell, An Introduction to Genetic Algorithms, Cambridge, Massachusett: MIT Press, 1996.

[9] J. . H. Holland , Adaptation in Natural and Artificial Systems, Ann Arbor Michigan: Press, 1975.

[10] J. d. Schacher, L. J. Caruna y R. Eshelman, «A study of control parameters affecting online performance of genetic algorithms for function optimization.,» de *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann, 1989, pp. 51-60.

[11] L. Davis , Handbook of Genetic Algorithms, Van Nostrand Reinhold, New York, 1991.

[12] E. Alfaro. [En línea]. Available: http://eddyalfaro.galeon.com/geneticos.html. [Último acceso: 2012].

[13] C. C. A. Coello, «Introducción a los Algoritmos Genéticos,» Red, Cientifica, [En línea]. Available: http://www.redcientifica.com/doc/doc199904260011.html. [Último acceso: 2011].

[14] J. T. Alander, «On optimal population size of genetic algorithms,» de *Computer Systems and Software Engineering*, Proceedings CompEuro, 1992, pp. 65-70.

[15] R. Reeves, R. Colin, E. Rowe y E. Jonathan , Genetic Algorithms, Principales and Perspectives, 2002.

[16] A. Marczyk, 2004. [En línea]. Available: http://the-geek.org/docs/algen/. [Último acceso: 2011].

[17] J. Monge F., «2010,» [En línea]. Available: http://www.tec-digital.itcr.ac.cr/revistamatematica/contribuciones-v6-n1-may2005/Geneticos/Index.htm.

[18] A. genético, «Eman,» [En línea]. Available: http://www.sc.ehu.es/ccwbayes/docencia/mmcc/docs/. [Último acceso: 2012].

[19] W. Shih, «A branch and bound method for the multiconstraint zero one knapsack problem,» de *Journal of Operation Research Society*, 1979, pp. 369-378.

[20] U. d. Valencia. [En línea]. Available: http://informatica.uv.es/iiguia/AAC/AA/apuntes/aic_rendimiento.pdf. [Último acceso: 2011].

[21] O. 1.0.4, «OOMPI,» [En línea]. Available: http://lancet.mit.edu/ga/. [Último acceso: 2011].

[22] . R. Matti, J. Westerholm y J. Dongarra, Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer, 2009.

[23] D. Van Gucht, J. Grefenstette, D. Gopal, B. Rosmaita y D. Van Gucht, «Proc. 1st Int. Conf. Genetic Algorithms and Their Applications,» de *Genetic Algorithms for the Traveling Salesman Problem*, Lawrence Erlbaum Ass, 1985, pp. 160-168.

[24] P. C. Gilmore y R. E. Gomory, «The theory and computation of knapsack functions,» de *Operations Research*, pp. 1045-1074.

[25] E. Chajakis y M. Guignard, «A model for delivery of groceries in vehicle with multiple compartments and Lagrangean approximation schemes,» de *Proceedings of Congreso Latino Ibero- Americano de Investigación de Operaciones e Ingeniería de Sistemas*, Mexico, 1992.

[26] C. C. Petersen, «Computational experience with variants of the Balas algorithm applied to the selection of R&D projects,» de *Management Science*, 1967, pp. 736-750.

[27] G. Kyparisis, S. Gupta y C. Ip, «Project selection with discount returns and multiple constraints,» de *European Journal of Operational Research*, 1996, pp. 87-96.

[28] H. M. Weingartner, Mathematical Programming and the Analysis of Capital Budgeting Problems, Chicago: Markham Publishing, 1967.

[29] B. Gavish y H. Pirkul, «Allocation of databases and processors in a distributed data processing,» de *Management of Distributed Data Processin*, Amsterdam, North-Holland, 1982, pp. 215-231.

[30] D. S. Hochbaum, Approximation Algorithms for NP-hard Problems, New York: PWS Publishing Company, 1996.

[31] S. Eilon, C. Watson, C. Gabdy y N. Christofides, «Mathematical modeling and practical analysis,» de *Distribution Management*, Hafner Publishing Company, 1971.

[32] S. Martello y Toth, «Knapsack Problems,» de *Algorithms and Computer Implementations*, Sons, 1990.

[33] O. H. Ibarra y C. E. Kim, «Fast Approximate Algorithms for the 0/1 Knapsack Problem and Sum of Subset Problems,» de *Journal of the Association for Computing Machinery*, 1975, pp. 463-468.

[34] R. Bellman, Dynamic Programming, Princeton University Press: Press NJ, 1957.

[35] L. B. Booker, D. E. Goldberg y J. H. Holland, «Classifier Systems and Genetic Algorithms,» de *Artificial Intelligence*, 1989, pp. 235-282.

[36] L. Davis, «Applying adaptive algorithms to epistatic domains,» de *Proceedings of the International Joint Conference on Arti_cial Intelligence*, 1985, pp. 162-164.

[37] D. Bertsimas y R. Demir, «An approximate dynamic programming approach to multidimensional knapsack problem,» de *Management Science*, 2002, pp. 550-565.

[38] T. E. Davis y J. C. , «A Markov chain framework for the simple genetic algorithm.,» de *Evolutionary Computation*, Principe, 1993, pp. 269-288.

[39] M. Gestal, «S.A.B.I.A. Group,» Operadores genéticos, [En línea]. Available: http://sabia.tic.udc.es/mgestal/cv/AAGGtutorial/node8.html. [Último acceso: 2011].

[40] M. Guervos y J. Julian, «Informática evolutiva,» [En línea]. Available: http://geneura.ugr.es/~jmerelo/ie/ags.htm. [Último acceso: 2010].