INCORPORACIÓN DE UN PARÁMETRO FRACTAL AL ALGORITMO GENÉTICO EN PROBLEMAS NP

Trabajo de Grado Presentado por
Mayra Alejandra Céspedes Fernández

Como requerimiento parcial para optar
por el título de
Ingeniero de Sistemas

Ph.D. Angel García Baños
Director

UNIVERSIDAD DEL VALLE
Facultad de Ingeniería
Escuela de Ingeniería de Sistemas y Computación
Noviembre 2008

Certifico que he leído este documento y que, en mi opinión, es completamente adecuado en alcance y calidad para optar por el título de Ingeniero de Sistemas.

(Ph.D. Angel García Baños) Director

Certifico que he leído este documento y que, en mi opinión, es completamente adecuado en alcance y calidad para optar por el título de Ingeniero de Sistemas.

(Jurado)

Aprobado por la Escuela de Ingeniería de Sistemas y Computación.

Introducción

Los problemas se pueden clasificar en conjuntos o clases de complejidad [InAlg01]. En este proyecto de grado, se estudiarán los problemas pertenecientes a la clase NP. Un problema pertenece a la clase NP si no es posible generar una solución con un algoritmo determinista en tiempo polinomial. Estos problemas son de gran complejidad y por ello es necesario investigar nuevos parámetros, de manera que se pueda predecir lo difícil que es resolver una instancia concreta de un problema.

En el campo de la Matemática y la Física existe un fenómeno denominado *Percolación* el cual describe, para un sistema, la transición de un estado en otro. [Perc02]

En este proyecto se buscará si hay fenómenos similares a la percolación en los problemas intratables, cuando se intentan resolver con algoritmos genéticos, para entender cuán difícil es el problema concreto.

La idea surge al leer el libro *Investigations* del biólogo Stuart Kauffman [Kau00]; en el cual se plantea una forma de navegar por el espacio de soluciones de un problema complejo - en ese caso Job Shop Scheduling -, basado en transiciones de fase.

El propósito de este proyecto es caracterizar la aplicación de la técnica de Percolación a problemas de tipo NP. Los objetivos del mismo son:

- Aplicar la técnica de Percolación.

- Determinar la dimensión fractal de los problemas Traveling Salesman Problem, K-SAT y Knapsack.
- Determinar que correlación existe entre la dificultad de un problema y el grado de percolación del mismo, aplicándolo a los problemas: Traveling Salesman Problem, K-SAT y Knapsack.
- Desarrollar un algoritmo genético que permita aplicar de forma sencilla la técnica de percolación para controlar la presión selectiva.
- Comparar los resultados obtenidos en los problemas Traveling Salesman Problem, K-SAT y Knapsack aplicando la técnica de percolación y cuando no se aplica.

El documento se estructura de la siguiente forma:

Capítulo 1. Se muestra la definición de la Clase NP y se explican cada uno de los problemas a tratar en este proyecto de grado: Traveling Salesman Problem, K-SAT y Knapsack.

Capítulo 2. Se explica el concepto de Algoritmo Genético, así mismo se muestra su estructura y funcionamiento.

Capítulo 3. Se presenta la definición del concepto de Percolación y su clasificación.

Capítulo 4. Se muestra la definición del concepto Fractal, sus características y algunos ejemplos.

Capítulo 5. Se describen los *Generadores de Problemas* para cada tipo de problema (Traveling Salesman Problem, K-SAT y Knapsack).

Capítulo 6. Se presenta una explicación de cómo se realizaron cada uno de los Algoritmos Genéticos para resolver cada problema; se muestra el cálculo de la percolación

y su aplicación en los Algoritmos Genéticos.

Capítulo 7. Se muestran las pruebas realizadas con cada algoritmo y las conclusiones.

Referencias Bibliográficas.

Anexos.

Índice General

Índice de tablas

Índice de figuras

Capítulo 1

Problemas NP

Los recursos comúnmente estudiados en complejidad computacional son:

- El tiempo: mediante una aproximación al número de pasos de ejecución que un algoritmo emplea para resolver un problema.
- El espacio: mediante una aproximación a la cantidad de memoria utilizada para resolver el problema.

En el análisis computacional, se requiere un modelo de la computadora para la que desea estudiar el requerimiento en términos de tiempo. Normalmente, dichos modelos suponen que la computadora es determinista (dado el estado actual de la computadora y las variables de entrada, existe una única acción posible que la computadora puede tomar) y secuencial (realiza las acciones una después de la otra). Estos supuestos son adecuados para representar el comportamiento de algunas computadoras existentes, incluyendo algunas máquinas con computación en paralelo. [InAlg01]

Los problemas se clasifican en conjuntos o clases de complejidad (L, NL, P, P-Completo,NP, NP-Completo, NP Duro, entre otros).

En este proyecto de grado se trabajará con 3 problemas pertenecientes a la clase NP-Completo: Traveling Salesman, K-SAT y Knapsack. Para definir la clase NP-Completo, es necesario tener la definición de: Problemas de decisión, NP y NP-Hard.

Problema de Decisión. Un problema de decisión es un problema en donde las respuestas posibles son SI o NO. Un ejemplo de problema de decisión es la pregunta: ¿ Es un número entero dado primo? Una instancia de este problema sería: ¿ Es 13 primo? [InAlg01]

NP. En teoría de la complejidad computacional, NP es el acrónimo en inglés de Polinómico no determinista (Non-Deterministic Polynomial-time). Es el conjunto de problemas que pueden ser resueltos en tiempo polinómico por una máquina de Turing no determinista. [InAlg01]

La importancia de esta clase de problemas de decisión es que contiene muchos problemas de búsqueda y de optimización para los que se desea saber si existe una cierta solución o si existe una mejor solución que las conocidas. Los problemas más conocidos son: TSP donde se quiere saber si existe una ruta óptima que pasa por todos los nodos en un cierto grafo y K-SAT en donde se desea saber si una cierta fórmula de lógica proposicional puede ser verdadera para algún conjunto de valores booleanos para las variables.

Dada su importancia, se han hecho muchos esfuerzos para encontrar algoritmos que decidan algún problema de NP en tiempo polinómico. Sin embargo, pareciera que para algunos problemas de NP (los del conjunto NP-completo) no es posible encontrar un algoritmo mejor que simplemente realizar una búsqueda exhaustiva.

NP-Hard. Es el conjunto de los problemas de decisión que contiene los problemas H tales que todo problema L en **NP** puede ser tranformado polinomialmente en H. Esto quiere decir, que los problemas son al menos tan complejos con NP, pero no necesariamente se encuentran en NP. [InAlg01]

NP-Completo. Es el suboconjunto de los problemas de decisión en NP tal que todo problema en NP se puede reducir en cada uno de los problemas de NP-Completo. Es

decir:

Un problema de decisión C es NP-Completo si:

1. C es un problema NP, y
2. Todo problema de NP se puede tranformar polinómicamente en C; es decir, es NP-Hard.

El primer problema natural que se demostró que es NP-completo fue el problema SAT. Este resultado fue demostrado por Stephen Cook en 1971, y se lo llamó el teorema de Cook [Cook71]. La demostración de Cook de que SAT un problema NP-completo es muy complicada. Sin embargo, después de que este problema se demostrara que es NP-Completo, es fácil demostrar que muchos otros problemas pertenecen a esta clase.

A continuación se mostrarán las definiciones de los problemas Traveling Salesman, K-SAT y Knapsack.

1.1 Traveling Salesman Problem

El TSP es uno de los problemas más famosos (y quizás el mejor estudiado) en el campo de la optimización combinatoria computacional. A pesar de la aparente sencillez de su planteamiento, el TSP es uno de los más complejos de resolver y existen demostraciones que equiparan la complejidad de su solución a la de otros problemas aparentemente mucho más complejos que han retado a los matemáticos desde hace siglos.

Se plantea de la siguiente forma:

Sean N ciudades de un territorio. El objetivo es encontrar una ruta que, comenzando y terminando en una ciudad concreta, pase una sola vez por cada una de las ciudades y minimice la distancia recorrida por el viajante. Es decir, encontrar una permutación $P = c_0, c_1, ..., c_{n-1}$ tal que $dP = \sum_{i=0}^{n-1} d[c_i, c_{i+1 mod(N)}]$ sea mínimo. [InAlg01]

1.2 K-SAT

Una variable booleana es una variable que sólo puede tomar los valores "verdadero" y "falso". Una fórmula booleana es una expresión formada con variables booleanas y los operadores $\vee$ (disyunción), $\wedge$ (conjunción) y $\neg$ (negación). Si $x_1,\ x_2\ y\ x_3$ son variables booleanas, la expresión $\neg(x_1 \wedge (x_2 \vee x_3)) \vee \neg x_1$ es una fórmula booleana. Una asignación para un conjunto de valores es una asignación de *verdadero* o *falso* a cada una de las variables. Una asignación satisface una fórmula booleana si al evaluar ésta obtenemos el resultado *verdadero*. La asignación $x_1 = falso$, $x_2 = verdadero$ y $x_3 = falso$, por ejemplo, satisface la anterior fórmula booleana [Alg03].
Un literal es una variable o su negación. Una fórmula booleana está en **Forma Conjuntiva Normal** (FNC) si y sólo si es una conjunción de disyunciones de literales. Cada una de las disyunciones de literales es una *cláusula*. Por ejemplo, la fórmula $(x_1 \vee x_2) \wedge (x_2 \vee \neg x_3)$ es una fórmula booleana en FNC.

Existen 2 casos particulares del K-SAT:

- **2-SAT.** Donde cada cláusula contiene exáctamente $k = 2$ literales. Este tipo de problemas pueden ser resueltos en tiempo polinomial, y de hecho es **NL-Completo**; es decir, puede resolverse en $\log(n)$ para cualquier tamaño de entrada n. [InAlg01]

- **3-SAT.** Donde cada cláusula contiene exáctamente $k = 3$ literales. En este caso la complejidad es **NP-Completo**. [InAlg01]

El problema de decisión **SAT** se plantea de la siguiente manera:

SAT
INSTANCIA: una colección $C_1, C_2, ..., Cm$ de cláusulas de un conjunto finito de variables U.
PREGUNTA: ¿Existe una asignación para U que satisfaga todas las cláusulas de C?

1.3 Knapsack

El *Knapsack* es un problema típico de programación entera.

Supóngase que se tiene n objetos distintos y un recipiente. Cada uno de los objetos tiene asociado un peso positivo w_i y un valor positivo v_i para $i = 1, 2, ..., n$. El recipiente no puede llevar un peso que sobrepase la cantidad W. Teniendo en cuenta estos datos, el objetivo del poblema es llenar el recipiente de tal forma que se maximice el valor de los objetos transportados en este. Se debe tener presente que la suma de los pesos de los objetos seleccionados no debe superar la capacidad máxima W del recipiente. Por lo tanto, para obtener la solución deseada, se debe decidir para cada uno de los objetos, si se introduce o no en el recipiente [Knap].

Existen en la actualidad diferentes métodos para tratar de resolver los problemas anteriormente mecionados, los más reconocidos son: Programación Dinámica, Programación con Restricciones, Algoritmos Probabilísticos, Algoritmos Genéticos.

Programación Dinámica

La programación dinámica es un método para reducir el tiempo de ejecución de un algoritmo mediante la utilización de subproblemas superpuestos y subestructuras óptimas.

El matemático Richard Bellman inventó la programación dinámica en el año de 1953 [Bell57], y propuso lo siguiente:

1. Dividir el problema en subproblemas más pequeños.
2. Resolver estos problemas de manera óptima usando este proceso de tres pasos recursivamente.
3. Usar estas soluciones óptimas para construir una solución óptima al problema original.

Los subproblemas se resuelven a su vez dividiéndolos ellos mismos en subproblemas más pequeños hasta que se alcance el caso fácil, donde la solución al problema es trivial.

La programación dinámica toma normalmente uno de los dos siguientes enfoques:

- **Top-down:** El problema se divide en subproblemas, y éstos se resuelven recordando las soluciones en caso de que sean necesarias nuevamente.
- **Bottom-up:** Todos los subproblemas que puedan ser necesarios se resuelven de antemano y después son usados para resolver las soluciones a problemas mayores. Este enfoque es ligeramente mejor en cuanto a consumo de espacio y llamadas a funciones, pero a veces resulta poco intuitivo encontrar todos los subproblemas necesarios para resolver un problema dado.

Programación con Restricciones

La programación con restricciones es una tecnología software utilizada para la especificación y posterior resolución efectiva de grandes y complejos problemas, particularmente combinatorios, de muchas áreas de la vida real. Muchos de estos problemas pueden modelarse como problemas de satisfacción de restricciones (CSPs) y resolverse usando técnicas de programación de restricciones. Esto incluye problemas de áreas tales como inteligencia artificial, investigación operativa, bases de datos, sistemas expertos. Algunos ejemplos son scheduling, planificación, razonamiento temporal, diseño en la ingeniería, problemas de empaquetamiento, criptografía, toma de decisiones. La complejidad de este tipo de problemas suele ser NP.

Los primeros trabajos relacionados con la programación de restricciones datan de los años 60 y 70 en el campo de la Inteligencia Artificial. La idea de la programación por restricciones es resolver problemas mediante la declaración de restricciones sobre el área del problema y consecuentemente encontrar soluciones que satisfagan todas las restricciones, y en su caso óptimo unos criterios determinados.

Algoritmos Probabilistas

Un algoritmo probabilista (o probabilístico) es un algoritmo que basa su resultado en la toma de algunas decisiones al azar, de tal forma que, en promedio, obtiene una buena solución al problema planteado para cualquier distribución de los datos de entrada. Es decir, al contrario que un algoritmo determinista, a partir de unos mismos datos se pueden obtener distintas soluciones y, en algunos casos, soluciones erróneas.
Existen varios tipos de algoritmos probabilísticos dependiendo de su funcionamiento, pudiéndose distinguir:

- **Algoritmos numéricos**, que proporcionan una solución aproximada del problema.
- **Algoritmos de Monte Carlo**, que pueden dar la respuesta correcta o respuestas erróneas (con probabilidad baja).
- **Algoritmos de Las Vegas**, que nunca dan una respuesta incorrecta: o bien dan la respuesta correcta o informan del fallo.

Se puede optar por la elección aleatoria si se tiene un problema cuya elección óptima es demasiado costosa frente a la decisión aleatoria. Un algoritmo probabilista puede comportarse de distinta forma aplicando la misma entrada.

También se encuentran los **Algoritmos Genéticos**, que para efectos del proyecto se escogió como el método de resolución de los problemas que se han planteado. El algoritmo se explica con más detalles en el siguiente capítulo.

Capítulo 2

Algoritmos Genéticos

A finales de los años 60, un investigador de la Universidad de Michigan, el profesor John Holland, creó los *algoritmos genéticos*. Se les llama así porque se inspiran en la evolución biológica y su base genético-molecular. A la técnica que inventó Holland se le llamó originalmente *planes reproductivos*, pero se hizo popular bajo el nombre de *algoritmo genético* tras la publicación de su libro en 1975 [Holl75]. Una definición bastante completa de un algoritmo genético es la propuesta por John Koza: "Es un algoritmo matemático altamente paralelo que transforma un conjunto de objetos matemáticos individuales con respecto al tiempo usando operaciones modeladas de acuerdo al principio Darwiniano de reproducción y supervivencia del más apto, y tras haberse presentado de forma natural una serie de operaciones genéticas en las que se destaca la recombinación o cruce sexual. Cada uno de estos objetos matemáticos suele ser una cadena de caracteres (letras o números) de longitud fija que se ajusta al modelo de las cadenas de cromosomas, y se les asocia con una cierta función matemática que refleja su aptitud". [Koz90]

Los Algoritmos Genéticos (AGs) son métodos adaptativos que pueden usarse para resolver problemas de búsqueda y optimización. Establecen una analogía entre el conjunto de soluciones de un problema, llamado fenotipo, y el conjunto de individuos de una población natural, codificando la información de cada solución en una cadena, generalmente binaria, llamada *cromosoma*. Los símbolos que forman la cadena son

llamados *genes*. Los cromosomas evolucionan a través de iteraciones, llamadas *generaciones*. En cada generación, los cromosomas son evaluados usando alguna medida de aptitud. Las siguientes generaciones (nuevos cromosomas), llamada *descendencia*, se forman utilizando dos operadores, de *cruzamiento* y de *mutación*. La evolución de dichas soluciones hacia valores óptimos del problema depende en buena medida de una adecuada codificación de las mismas. Una de sus características principales es la de ir perfeccionando su propia heurística en el proceso de ejecución, por lo que no requiere largos períodos de entrenamiento especializado por parte del ser humano, principal defecto de otros métodos para solucionar problemas, como los Sistemas Expertos. [AlgGen]

A continuación se presenta el funcionamiento de un algoritmo genético simple, se definen los operadores genéticos más comunes y se presentan algunas ventajas y desventajas respecto a otros métodos.

2.1 Funcionamiento

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. 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. La combinación de buenas características provenientes de diferentes ancestros, puede a veces producir descendientes ligeramente mejores al resto de indiviuos, y cuya adaptación es mayor que la de cualquiera de sus ancestros. De esta manera, las especies evolucionan logrando unas características cada vez mejor adaptadas al entorno en el que viven.

Los Algoritmos Genéticos usan una analogía directa con el comportamiento natural.

Trabajan con una población de individuos, cada uno de los cuales representa una solución factible a un problema dado. A cada individuo se le asigna un valor o puntuación, relacionado con la bondad de dicha solución. En la naturaleza esto sería equivalente al grado de efectividad de un organismo para competir por unos determinados recursos. 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. [AGWiki]

De esta manera se produce una nueva población de posibles soluciones, la cual reemplaza 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.

2.2 Algoritmo

2.2.1 Codificación

Si bien el alfabeto utilizado para representar los individuos no debe necesariamente estar constituido por el (0,1), buena parte de la teoría en la que se fundamentan los Algoritmos Genéticos utiliza dicho alfabeto. En términos biológicos, el conjunto de parámetros representando un cromosoma particular se denomina fenotipo. El fenotipo contiene la información requerida para construir un organismo, el cual se refiere como genotipo. Los mismos términos se utilizan en el campo de los Algoritmos Genéticos.

```
BEGIN /* Algoritmo Genetico Simple */
    Generar una poblacion inicial.
    Computar la funcion de evaluacion de cada individuo.
    WHILE NOT Terminado DO
    BEGIN /* Producir nueva generacion */
        FOR Tamaño poblacion/2 DO
        BEGIN /*Ciclo Reproductivo */
            Seleccionar dos individuos de la anterior generacion,
            para el cruce (probabilidad de seleccion proporcional
            a la funcion de evaluacion del individuo).
            Cruzar con cierta probabilidad los dos
            individuos obteniendo dos descendientes.
            Mutar los dos descendientes con cierta probabilidad.
            Computar la funcion de evaluacion de los dos
            descendientes mutados.
            Insertar los dos descendientes mutados en la nueva generacion.
        END
        IF  la poblacion ha convergido THEN
            Terminado := TRUE
    END
END
```

Figura 2.1: Pseudocódigo del algoritmo

La adaptación al problema de un individuo depende de la evaluación del genotipo. Esta última puede inferirse a partir del fenotipo, es decir puede ser calculada a partir del cromosoma, usando la función de aptitud.

2.2.2 Función de Aptitud

A cada uno de los cromosomas de esta población se aplicará la función de aptitud para saber qué tan "buena" es la solución que se está codificando. La *Función de Aptitud* debe ser diseñada para cada problema de manera específica. Dado un cromosoma particular, la función de adaptación le asigna un número real, que se supone refleja el nivel de adaptación al problema del individuo representado por el cromosoma.

2.2.3 Selección

Después de saber la aptitud de cada cromosoma se procede a elegir los cromosomas que serán cruzados en la siguiente generación. La selección debe dirigir el proceso de búsqueda a favor de los miembros más aptos. Los métodos de selección más comunes son:

Selección por Sorteo

Se consideran las puntuaciones estrictamente como probabilidades de elección para formar la muestra y se constituye ésta realizando k ensayos de una variable aleatoria con dicha distribución de probabilidades. Dado que habitualmente sólo se dispone de un generador de números aleatorios simples -es decir, uniformemente distribuídos entre 0 y 1-, para simular un ensayo de la distribución $p_1, p_2, ..., p_n$ se hace lo siguiente:

1. Se calculan las puntuaciones acumuladas así:

$$q_0 := 0$$
$$q_i := p_1 + ... + p_i (\forall_i = 1, ..., k)$$

2. Se genera un número aleatorio simple $r \leftarrow URand[0, 1]$

3. Se elige el individuo x_i que verifique

$$q_{i-1} < r < q_i$$

Selección por Restos

A cada individuo x_i se le asigna directamente $\lfloor p_i * k \rfloor$ puestos en la muestra. Seguidamente, los individuos se reparten los puestos vacantes en función de sus permutaciones. El reparto se suele hacer por sorteo y se le llama "muestreo estocástico por restos". A continuación se presenta un ejemplo:

Se tiene la población $P = \{x_1, x_2, x_3, x_4\}$ siendo las puntuaciones asociadas $p_1 = 0.1$, $p_2 = 0.3$, $p_3 = 0.1$ y $p_4 = 0.5$ respectivamente. Suponiendo que se quiere generar 2 hijos ($k = 2$), entonces, a x_4 le corresponde un puesto en la muestra. El individuo que ocupe el otro puesto se elige por sorteo; es decir, se genera un número aleatorio simple $r = 0.39874$ y se compara con las permutaciones acumuladas q_i, en este caso el puesto vacante le corresponde a x_2 dado que $q_1 < r < q_2$.

Selección por Ruleta

Es análogo al muestreo por sorteo sólo que en este caso se genera un único número aleatorio simple r y con él se asignan todas las muestras de modo parecido a como se haría en una ruleta.

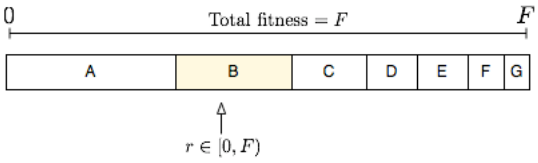


Figura 2.2: Selección Ruleta

Para simular una tirada de ruleta se definen a partir de $r \leftarrow URand[0, 1]$ los siguientes números:

$$r_i := \frac{r+j-1}{k} \ \forall_i = 1, ..., k$$

Selección por Torneo

Cada elemento de la muestra se toma eligiendo el mejor de los individuos de un conjunto de z elementos tomados al azar de la población base. Esto se repite k veces hasta completar la muestra. El parámetro z suele ser un entero pequeño comparado con el tamaño de la población base, normalmente 2 ó 3. En este caso no es necesario hacer la asignación de puntuaciones.

2.2.4 Cruzamiento

El cruzamiento es el principal operador genético, representa la reproducción sexual, opera sobre dos cromosomas a la vez para generar dos descendientes donde se combinan las características de ambos cromosomas padres. Las dos formas más comunes de reproducción por cruce son: uso de un punto único de cruce y uso de dos puntos de cruce.

Cuando se usa un solo punto de cruce, éste se escoge de forma aleatoria sobre la longitud de la cadena que representa el cromosoma, y a partir de él se realiza el intercambio de material de los individuos, cada pareja da origen a dos descendientes para la siguiente generación.

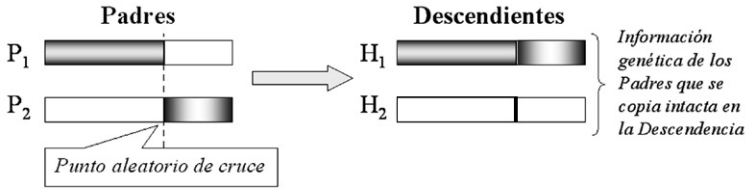


Figura 2.3: Cruce por un solo punto

Cuando se usan dos puntos de cruce, se procede de manera similar, pero en este caso se mantienen los genes de los extremos, y se intercambian los del centro. También aquí existe la posibilidad de que uno o ambos puntos de cruce se encuentren en los extremos de la cadena, en cuyo caso se hará un cruce usando un solo punto, o ningún cruce según corresponda.

2.2.5 Mutación

El operador de mutación se aplica a cada hijo de manera individual, y consiste en la alteración aleatoria (normalmente con probabilidad pequeña) de cada gen que compone el cromosoma.

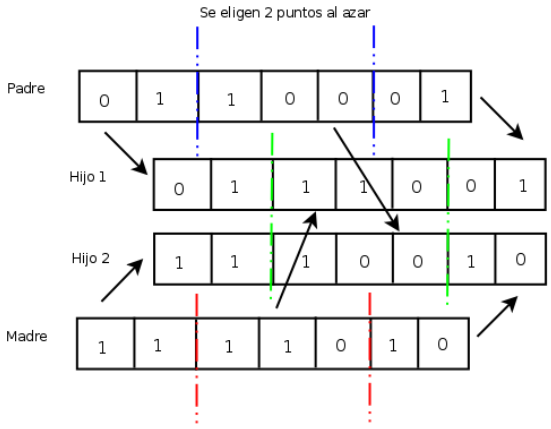


Figura 2.4: Cruce por dos puntos

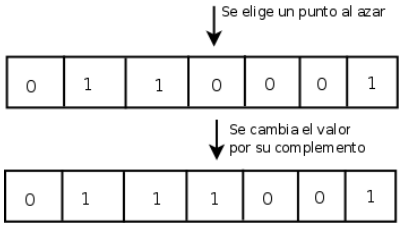


Figura 2.5: Mutación

Cuando se usa una representación binaria, el gen seleccionado se reemplaza por su complemento (un cero se cambia por uno y viceversa).

2.2.6 Reemplazo

Después del proceso de cruce y mutación se genera una nueva población de cromosomas "hijos" que reemplzarán a algunos cromosomas de la población anterior. El *Elitismo* es un método que introduce el mejor indiviuduo (o varios) a la nueva población de cromosomas. Sin embargo, si se usa demasiado, la velocidad del algoritmo aumenta y en consecuencia se cae en óptimos locales.

2.2.7 Finalización

El AG se deberá detener cuando se alcance la solución óptima, pero ésta generalmente se desconoce, por lo que se deben utilizar otros criterios de detención. Normalmente se usan dos criterios: correr el AG un número máximo de iteraciones (generaciones) o detenerlo cuando no haya cambios en la población.

2.3 Ventajas y Desventajas

- Operan de forma simultánea con varias soluciones, en vez de trabajar de forma secuencial como las técnicas tradicionales.
- Cuando se usan para problemas de optimización (maximizar una función objetivo) resultan menos afectados por los máximos locales (falsas soluciones) que las técnicas tradicionales, debido a que utilizan paralelismo implícito.
- Usan operadores probabilísticos, en vez de los típicos operadores determinísticos de las otras técnicas.
- Pueden tardar mucho en converger, o no converger en absoluto, dependiendo en cierta medida de los parámetros que se utilicen: tamaño de la población, número de generaciones.
- Pueden converger prematuramente debido a una serie de problemas de diversa índole.

Capítulo 3

Teoría de los Procesos de Percolación

3.1 Definición

La Teoría de los procesos de percolación comenzó en 1957 cuando S. R. Broadbent y J. M. Hammersley introdujeron el proceso de *Percolación independiente* [BrHa57]. Lo plantearon como un modelo probabilístico para el flujo de un fluido o un gas a través de un medio aleatorio. Los términos fluido y medio se propusieron de manera totalmente general: fluido puede ser un líquido, vapor, flujo de calor, corriente eléctrica, infección, sistema solar o cualquier fluido que pueda moverse a través de un medio. El medio, donde se lleva el fluido puede ser el espacio poroso de una roca, un suelo, un arreglo de árboles o el mismo universo. La dispersión de un fluido a través de un medio desordenado envuelve algunos elementos aleatorios, pero el mecanismo, puede ser uno de dos diferentes tipos. En el primer tipo, la aleatoriedad se atribuye al fluido en un proceso de difusión y en el segundo, se atribuye al medio, en un proceso de percolación [Perc01].

La Teoría de la *Percolación* es una teoría multidisciplinaria, que estudia sistemas desordenados o caóticos, en los cuales los componentes están distribuidos aleatoriamente en una red, permitiendo estudiar fenómenos críticos. Las redes pueden tener tamaños

muy diversos y pueden ser de formas regulares o irregulares. La red tomará la mejor configuración que represente al fenómeno que se va a modelar, así como la dimensión (2D ó 3D). Los fenómenos críticos se presentan en sistemas que se caracterizan por la existencia de un punto crítico en el cual ciertas propiedades del sistema cambian [Perc02].

3.2 Clasificación

La teoría de percolación clásica se centra alrededor de dos problemas. En el problema de percolación enlazada (o consolidada), la red está constituida por lazos. Los lazos de la red están ocupados aleatoriamente e independientemente de los otros con probabilidad *p*, (están abiertos al flujo, difusión y reacción, son elementos conductores microscópicos de un compuesto, etc.), o están vacantes (están cerrados al flujo o corriente, o han sido conectados, son elementos aislantes de un compuesto, etc.) con probabilidad *1-p*. Para una red grande, estas asignaciones son equivalentes a remover una fracción *1-p* para todos los lazos al azar. Dos sitios se llaman conectados si existe al menos una trayectoria entre ellos consistente únicamente de lazos ocupados. Un conjunto de sitios ocupados unidos por lazos vacantes es llamado racimo. Si la red es de extensión muy grande y si p es suficientemente pequeño, el tamaño de cualquier racimo conectado es pequeño. Pero si p es cercano a 1, la red debería estar completamente conectada, separada únicamente de pequeños agujeros ocasionales. En algunos valores de p bien definidos, existe una transición en la estructura topológica de la red aleatoria a partir de una estructura macroscópicamente desconectada a una conectada; este valor se llama umbral de percolación enlazada P_{cb}. Esta es la fracción más grande de lazos ocupados abajo de los cuales no existe racimo percolante de lazos ocupados.

Similarmente en el problema de percolación de sitio, la red está constituida por sitios. Los sitios de la red están ocupados con probabilidad p y las vacantes con probabilidad

1-p. Los dos sitios vecinos más cercanos se llaman conectados si están ambos ocupados, y los racimos conectados en la red son otra vez definidos en la manera obvia. Existe un umbral de percolación de sitio P_{cs} arriba del cual un racimo infinito de sitios ocupados atraviesa la red. Se debe apuntar que, dependiendo de una aplicación específica, se pueden desarrollar muchas variantes de la percolación enlazada o de sitio. Las redes regulares en 3D pueden tomar configuraciones volumétricas similares a las descripciones de arreglos de celdas para estructuras cristalinas. El umbral de percolación de una red 3D ha sido calculado numéricamente por simulaciones Monte Carlo o por otras técnicas. En general, $P_{cb} \leq P_{cs}$.

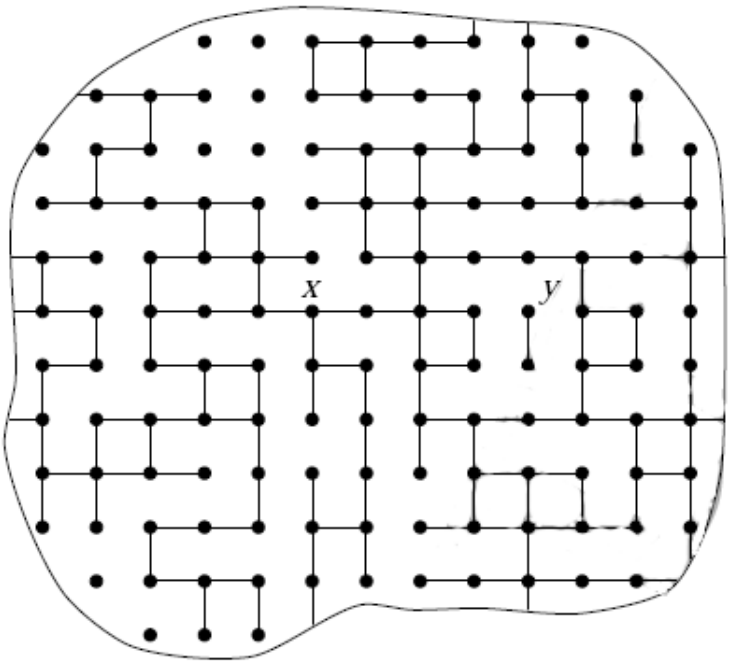


Figura 3.1: Representación en 2D de la Percolación en una piedra porosa.

3.3 Umbral de Percolación

Un cluster es el conjunto de elementos de un mismo componente que se encuentran en contacto. Cuando se extiende por todo el sistema se denomina cluster infinito o percolante. El umbral de percolación es uno de los conceptos más importantes de la Teoría de la Percolación y se define como la concentración de elementos en la que existe la máxima probabilidad de que aparezca un cluster infinito o percolante de un componente.

En el umbral de percolación $p = p_c$, el cluster resultante es un objeto fractal, cuya dimensión puede ser calculada experimentalmente y, en algunos casos, teóricamente.

Cuando un sistema como el que acabamos de describir alcanza el umbral de percolación, se produce una transición de fase geométrica, caracterizada por las propiedades geométricas del cluster infinito de percolación.

El cluster o agregado que aparece en el umbral de percolación, se denomina cluster infinito de percolación, porque su tamaño diverge cuando las dimensiones de la malla se incrementan indefinidamente. Como ya se ha mencionado, en el umbral de percolación, el cluster (infinito de percolación) es un objeto fractal, y se puede estimar su comportamiento geométrico mediante la dimensión fractal.

A continuación se mostrará un breve explicación del concepto Fractal y Dimensión Fractal.

Fractales

A mediados de los años 70's, el matemático Benoit Mandelbrot propuso una nueva herramienta para modelar fenómenos de la naturaleza, esta herramienta conocida como *Fractal* se introdujo rápidamente en la informática, principalmente en la graficación y conforme ha pasado el tiempo se le ha encontrado aplicaciones en muchas otras áreas incluyendo el reconocimiento de formas y la modelación. Un fractal es

un objeto semi geométrico cuya estructura básica, fragmentada o irregular, se repite a diferentes escalas. El término se deriva del Latín fractus, que significa quebrado o fracturado. [Man77]

Los fractales se clasifican en 3 categorías:

- Sistema iterado de funciones. Estos tienen una regla de punto fijo geométrico. Por ejemplo: Conjunto de Cantor, Alfombra de Sierpinski, Curva de Peano.

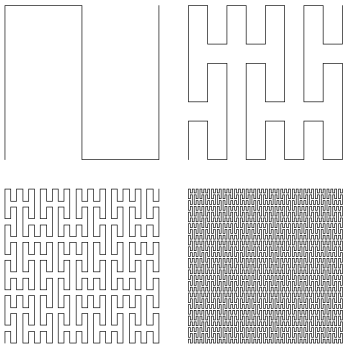


Figura 3.2: Curva de Peano

- Fractales definidos por una relación de recurrencia en cada punto de un espacio. Por ejemplo: Conjuntos de Julia.

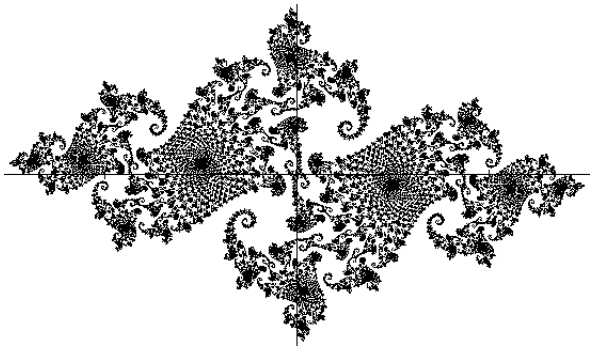


Figura 3.3: Conjuntos de Julia

- Fractales aleatorios. Generados por procesos estocásticos. Por ejemplo: Paisaje de Fractal, Cluster infinito de Percolación.

Dimensiones fractales

Para medir la complejidad de una figura, es decir el grado de irregularidad y de interrupción de los fractales se generaliza el concepto habitual de dimensión espacial, y se define la dimensión fractal como el grado de ocupación del espacio. Un punto tiene dimensión 0; una línea tiene dimensión 1; un plano, dimensión 2; una esfera, dimensión 3.

Sea s un factor de reducción de escala.

Dimensión de autosemejanza: Si se dividen en $1/s$ partes cada una de las dimensiones de un segmento, un cuadrado, un cubo, entre otros; el número N, de elementos análogos, pero a menor escala que resultan, viene dado por la expresin $N = (1/s)^D$, siendo D la dimensión espacial en cada caso. Tomando logaritmos en ambos miembros y despejando D se obtiene:

$D = \frac{lnN}{ln(\frac{1}{s})}$

	Factor de reducción de escala **s**	Nº de elementos **N**	Ley potencial $N=(1/s)^D$	Dimensión $D=\log N/\log(1/s)$
Segmento	1/3	3	$3=3^1$	1=log3/log3
Cuadrado	1/3	9	$9=3^2$	2=log9/log3
Cubo	1/3	27	$27=3^3$	3=log27/log3

Figura 3.4: Dimensiones fractales de figuras geométricas

Aplicando esta misma relación al caso de los fractales autosemejantes, N es el número de piezas que se reemplazan en cada paso de la construcción y s el factor de escala, se obtiene que:

	Factor de escala s	Nº de elementos N	Ley potencial $N=(1/s)^D$	Dimensión $D=\log N/\log s$
Conjunto de Cantor	1/3	2	$2=3^D$	D=log2/log3=0,631
Curva de Koch	1/3	4	$4=3^D$	D=log4/log3=1,262
Curva de Peano	1/3	9	$9=3^D$	D=log9/log3=2
Triángulo de Sierpinski	½	3	$3=2^D$	D=log3/log2=1,585

Figura 3.5: Dimensiones en fractales autosemejantes

Dimensión de caja: Se cubre el fractal con una retícula cuadrada en el plano (cúbica, si es en el espacio) y se cuenta el número N de cuadrados (cubos) que contienen puntos del fractal. Se procede del mismo modo con retículas cuyo lado sea la mitad que las anteriores. Idem con retículas cuyo lado sea la fracción s del de la primera.

Se dibujan los puntos $(lnN, ln(\frac{1}{s}))$, para distintos valores de s, en una hoja doblemente logaritmica, y se ajusta la recta a la nube de puntos resultante. La pendiente de esta recta representa la dimensión de caja 3.6.

No se vió la necesidad de calcular la dimensión fractal de los problemas Traveling Salesman, K-SAT y Knapsack; debido a que este cálculo no presenta ninguna relevancia para cumplir con el objetivo general del proyecto.

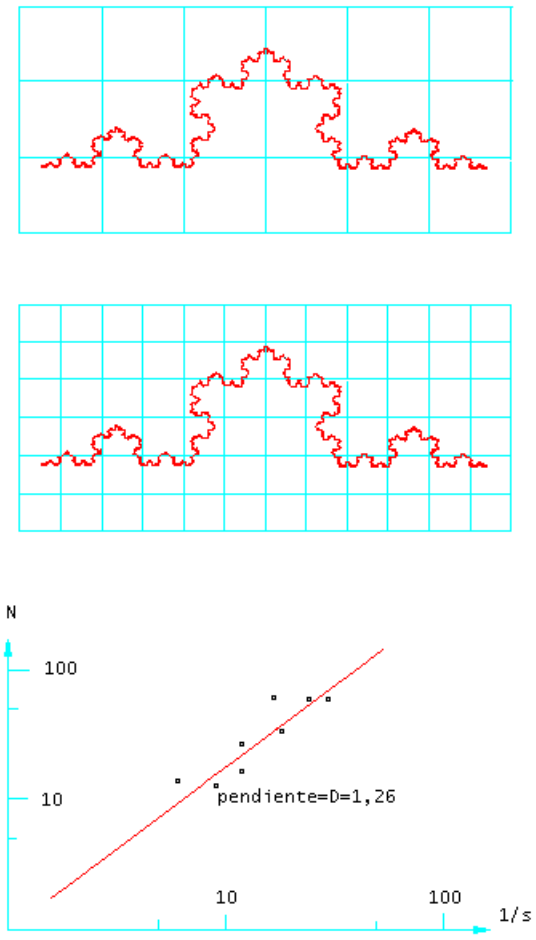


Figura 3.6: Dimensión de Caja

Capítulo 4

Generadores de Problemas

Debido a la naturaleza del proyecto, ésta implica una cantidad considerable de pruebas para conseguir conclusiones válidas, por lo menos para los tres tipos de problemas NP que se trabajarán en el proyecto. En este sentido, se hace necesario automatizar el proceso mediante la generación de problemas de tipo Traveling Salesman, K-SAT y knapsack.

Para realizar los generadores de problemas, se utilizó el lenguaje de programación **Python**.

Python es un lenguaje de programación interpretado, ampliable y orientado a objectos que se distribuye con un amplio conjunto de módulos que soportan acceso a un gran número de servicios del sistema operativo, servicios de internet (como HTML, XML, FTP, etc.), gráficos (incluyendo OpenGL, TK, etc.), funciones de manejo de cadenas, servicios de correo (IMAP, SMTP, POP3, etc.), multimedia (audio, JPEG) y servicios de criptografía. Además hay otros muchos módulos disponibles por terceros que añaden otros muchos servicios. Python es distribuido bajo términos similares a los de la licencia GPL y está disponible para los sistemas operativos Linux, Unix, Windows y Macintosh.

4.1 Estructura General

Inicialmente se definen los módulos que se van a utilizar, en este caso se utilizarán los módulos **random**, **sys**, **math** y una clase padre *GeneradorProblemas*, en la que se define la estructura que debe tener cada uno de los generadores, con métodos en blanco para que se rellenen en cada generador con sus particularidades. Luego se definen los métodos de la clase y un método principal en donde se crea una instancia de la clase generadora y se ejecuta cada método para generar el problema. Cabe anotar que para obtener cada problema, primero se genera la solución, y a partir de ésta se construye el problema en específico. Esto con el objetivo de comparar, mas adelante, la solución que arroja el algoritmo genético con la solución del generador.

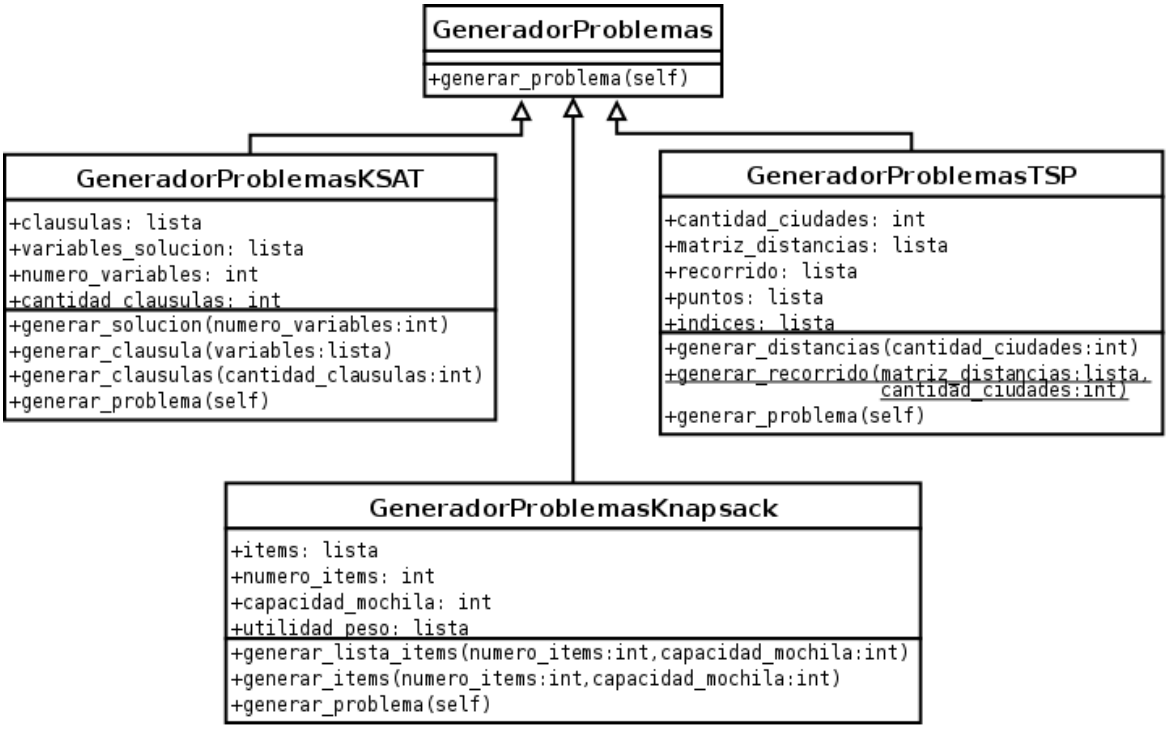


Figura 4.1: Diagrama de Clases

4.2 Generador de problemas tipo Traveling Salesman

Como parámetro de entrada tenemos el número de ciudades que se deben recorrer, a cada ciudad se le genera un punto en el plano (x, y). Para facilitar el proceso de búsqueda de la solución óptima, se ubican los puntos de tal manera que formen figuras geométricas como se muestra en la figura 5.2.

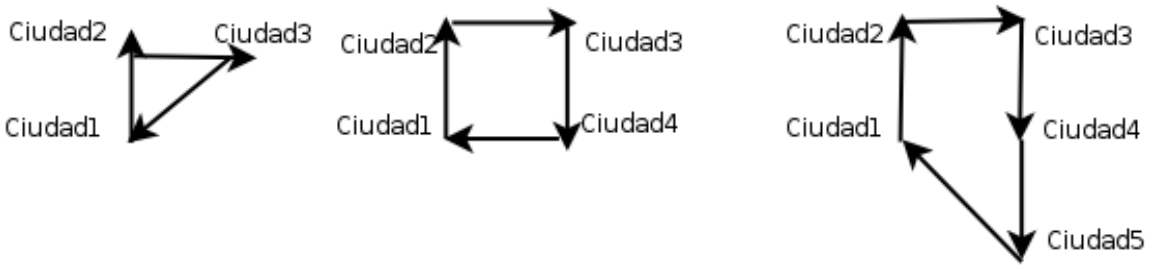


Figura 4.2: Formas Geométricas

Cada punto esta separado equidistantemente para garantizar la optimalidad de la solución. Luego de tener los puntos (x, y) de cada ciudad, se procede a calcular las distancias entre cada par de ciudades, formando una matriz de distancias cuya diagonal se encuentra llena de ceros. Finalmente, se calcula el recorrido, el cual será la secuencia ordenada ascendente o descendentemente de las ciudades -nótese que cualquier secuencia ordenada sirve como solución óptima-.

A continuación se presentan los diagramas de flujo para las funciones: "generar problemas", "generar recorrido" y "generar distancias".

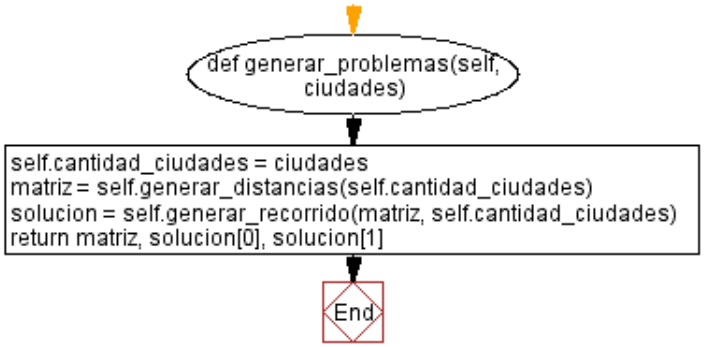


Figura 4.3: Generar Problemas TSP

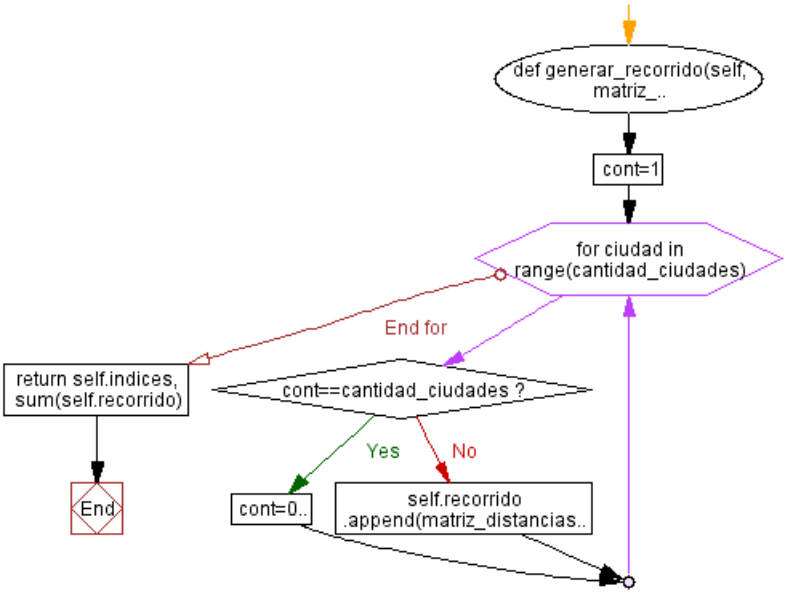


Figura 4.4: Generar Recorrido TSP

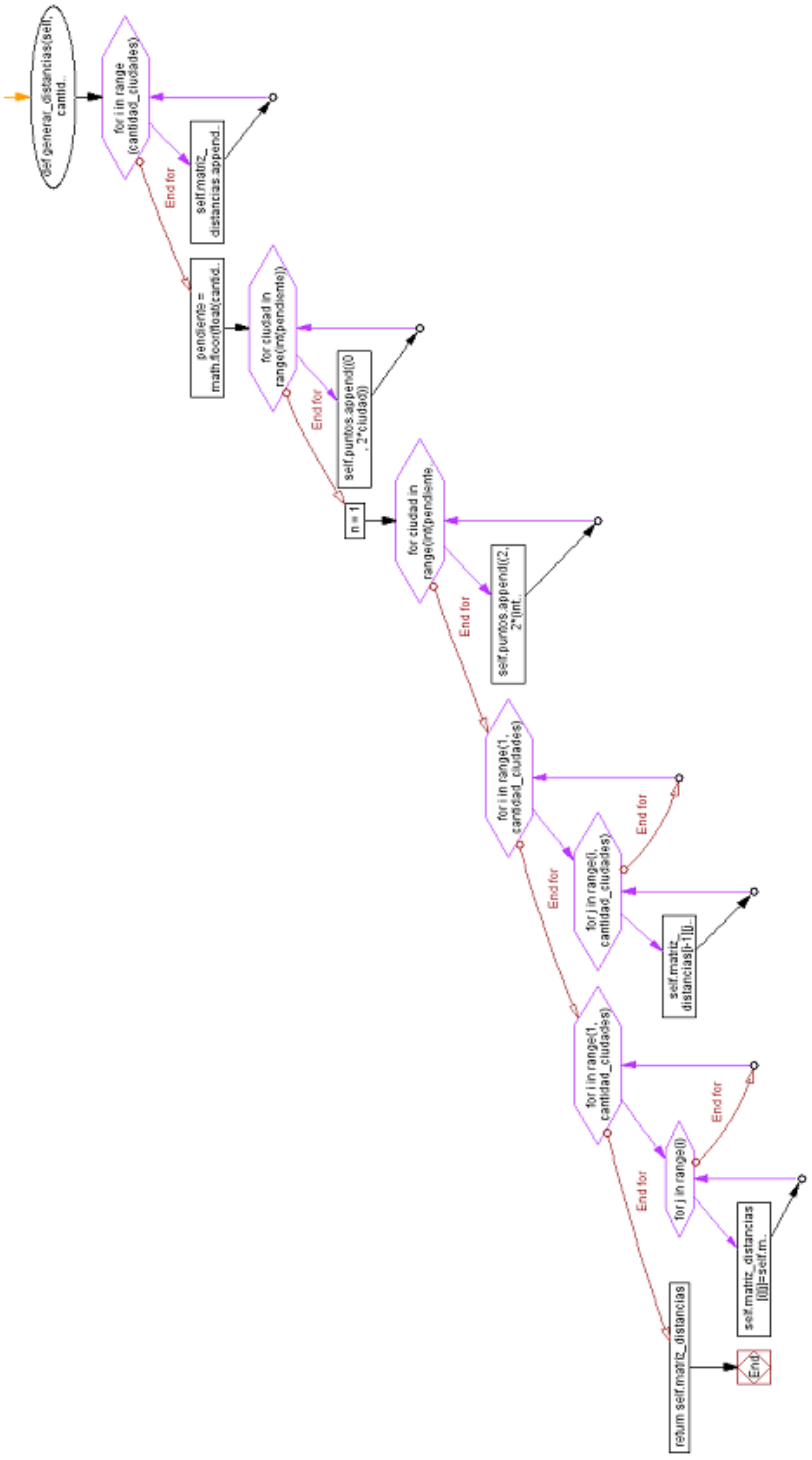


Figura 4.5: Generar Distancias TSP

4.3 Generador de problemas tipo K-SAT

Se decidió implementar el 3-SAT. Esto debido a que los problemas con K literales y n cláusulas que se obtienen del generador son muy fáciles de resolver; mientras que, con el 3-SAT, se obtienen instancias con mayor dificultad.
Para este problema se requieren 2 parámetros de entrada: el número de variables y el número de cláusulas. A continuación se generan valores de 0 y 1 de forma aleatoria y se insertan en una lista de tamaño igual al número de variables, dicha lista será la solución del problema. A partir de la solución se generan las cláusulas del problema, las cuales están codificadas de la siguiente forma:

0: la variable se encuentra negada.
1: la variable se encuentra normal.
2: la variable no se encuentra en la cláusula.

Cada vez que se genera una cláusula aleatoria, se verifica que la solución la pueda satisfacer. Si la cláusula se satisface, se inserta en el problema, en caso contrario, se genera otra cláusula aleatoria.

Un ejemplo de un problema generado con este algoritmo y su representación matemática es el siguiente:

$[1, 2, 2, 1, 0, 2, 2, 2, 2]$ $(x_1 \vee x_4 \vee \neg x_5) \wedge$
$[0, 2, 2, 1, 2, 2, 2, 1, 2]$ $(\neg x_1 \vee x_4 \vee x_8) \wedge$
$[2, 2, 1, 1, 2, 2, 2, 2, 0]$ $(x_3 \vee x_4 \vee \neg x_9) \wedge$
$[2, 2, 1, 2, 2, 2, 1, 0, 2]$ $(x_3 \vee x_7 \vee \neg x_8) \wedge$
$[0, 2, 2, 1, 0, 2, 2, 2, 2]$ $(\neg x_1 \vee x_4 \vee \neg x_5) \wedge$
$[2, 2, 0, 1, 2, 2, 1, 2, 2]$ $(\neg x_3 \vee x_4 \vee x_7)$

A continuación se presentan los diagramas de flujo para las funciones: generar problemas, generar solución, generar cláusulas y generar cláusula.

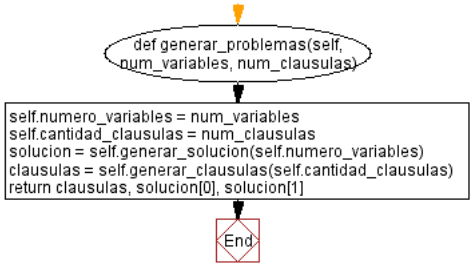


Figura 4.6: Generar Problemas K-SAT

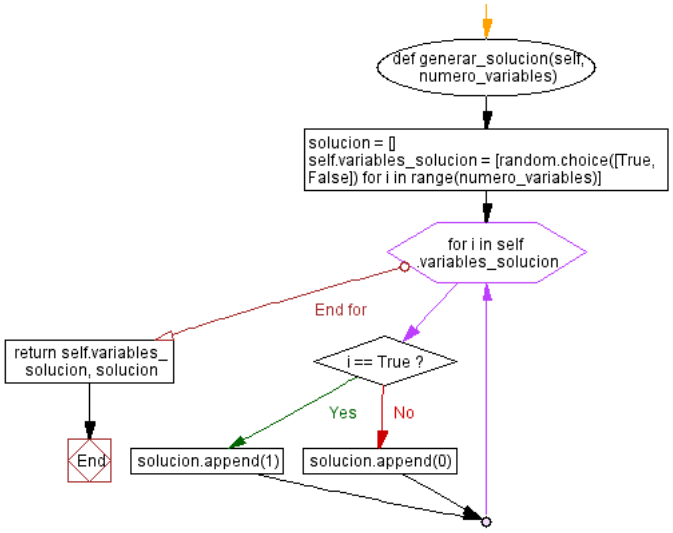


Figura 4.7: Generar Solución K-SAT

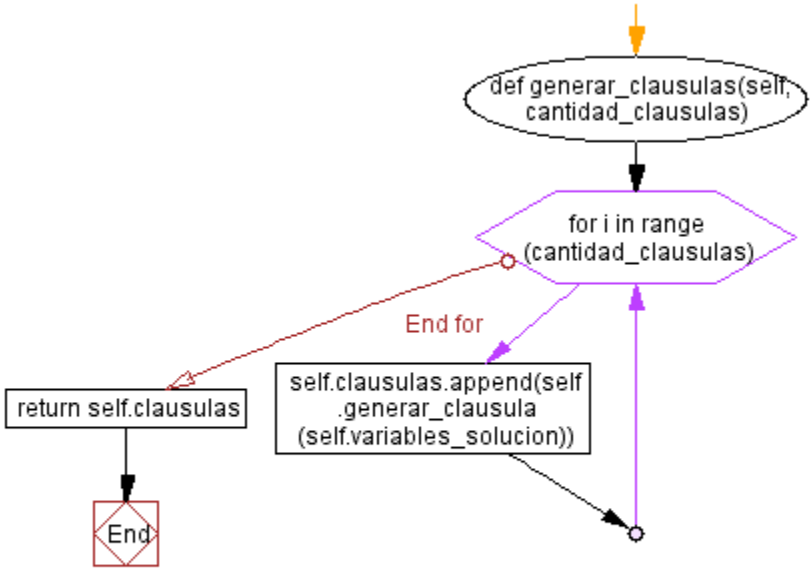


Figura 4.8: Generar Cláusulas K-SAT

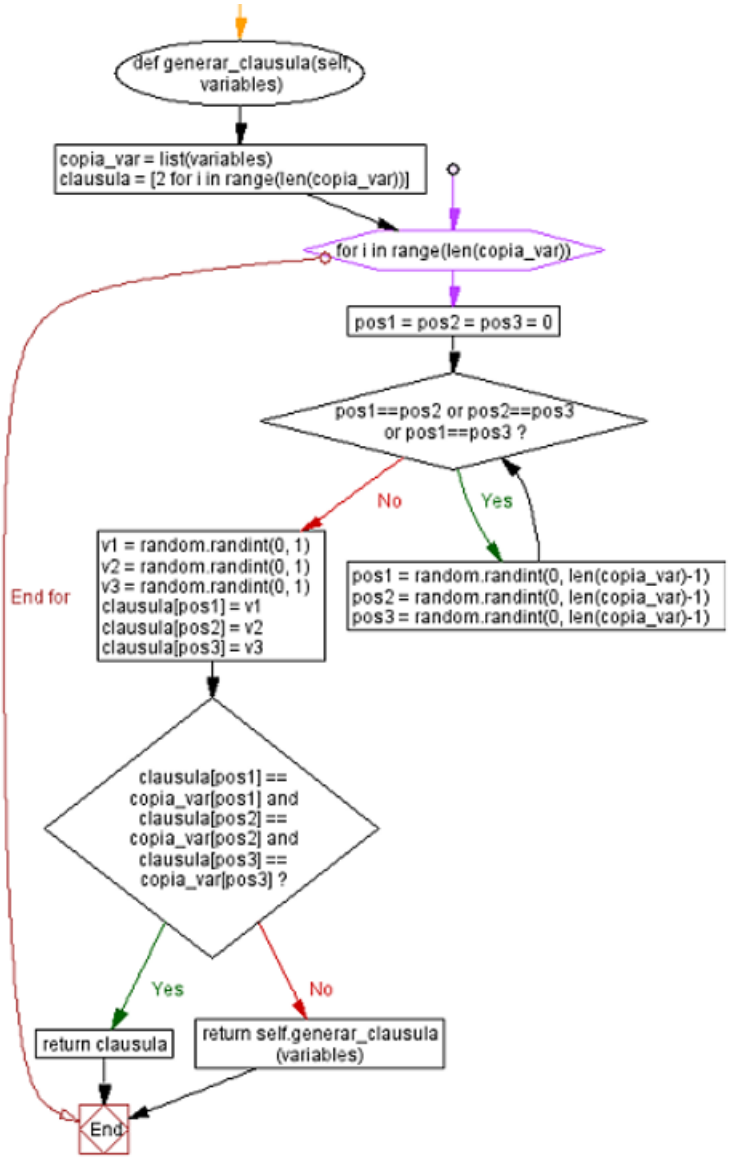


Figura 4.9: Generar Cláusula K-SAT

4.4 Generador de problemas tipo Knapsack

Para este problema se requieren 2 parámetros de entrada: el número de ítems y la capacidad de la mochila, suponiendo que el recipiente es una mochila. Para cada ítem se genera aleatoriamente un peso y se genera un valor potencia de 2 que será su utilidad -se decidió que fuera potencia de 2 porque al escoger la utilidad 2^n se garantiza que es la mayor utilidad, independientemente de si están presentes o no los $n-1$ ítems anteriores, cuya suma siempre da una utilidad menor-. Cada tupla (utilidad, peso), se inserta en una lista que se ordena de acuerdo al valor de la utilidad: $lista = [(2^n, peso_n), (2^{n-1}, peso_{n-1}), ..., (2^0, peso_0)]$
La solución óptima se conformará por los ítems que tengan mayor utilidad y cuyo peso no sobrepase la capacidad de la mochila.

A continuación se presentan los diagramas de flujo para las funciones: generar problemas, generar ítems y generar lista ítems.

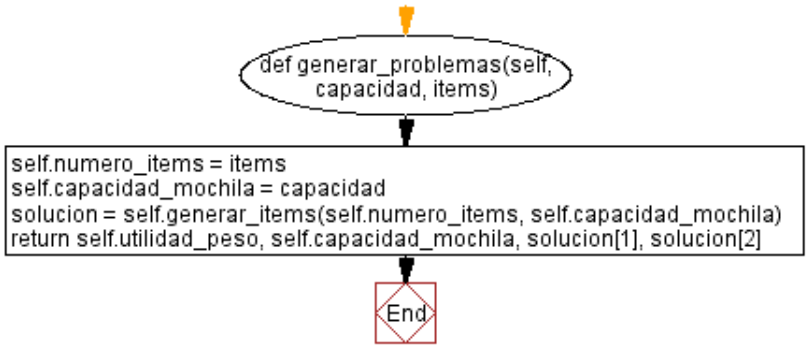


Figura 4.10: Generar Problemas Knapsack

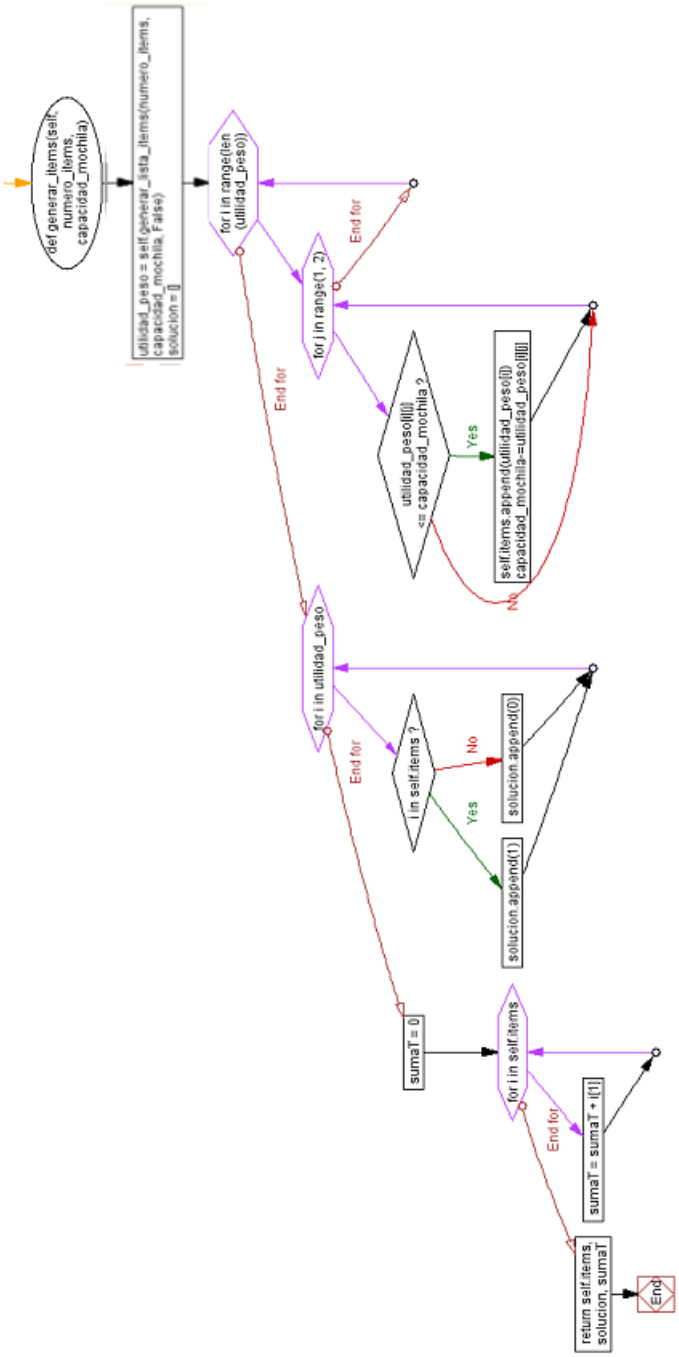


Figura 4.11: Generar Ítems Knapsack

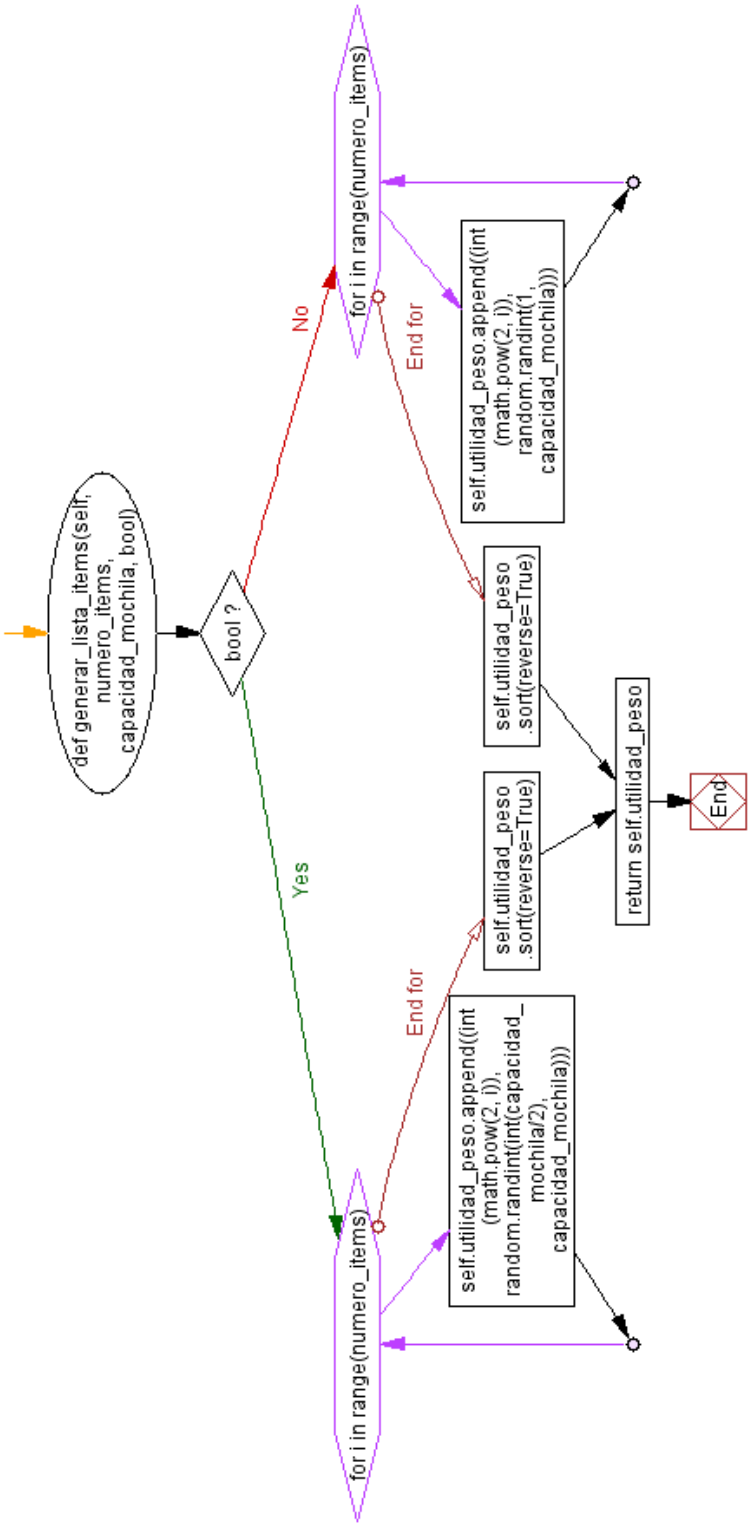


Figura 4.12: Generar Lista Ítems Knapsack

4.5 Automatización

Después de haber realizado y probado por separado cada uno de los generadores, se diseñó una interfaz sencilla, en la cual se pueden generar los problemas de cada tipo y guardarlos en dos archivos; uno para guardar el problema en concreto y el otro para guardar su solución.

Para el problema del viajero (TSP), cada línea del archivo contiene una lista de listas que representa la matriz de distancias de la n ciudades del problema:

[[0, 2.0, 4.0, 6.0, 8.0],
[2.0, 0, 2.0, 4.0, 6.0],
[4.0, 2.0, 0, 2.0, 4.0],
[6.0, 4.0, 2.0, 0, 2.0],
[8.0, 6.0, 4.0, 2.0, 0]]

En este caso se muestra una matriz de distancias de 5 ciudades. La solución se almacena en el archivo de soluciones y se codifica de la siguiente manera:

[1, 2, 3, 4, 5]

En el problema K-SAT, cada línea del archivo contiene una lista de listas, la cual representa el conjunto de cláusulas que se debe satisfacer:

[[1, 2, 1, 2, 2, 0, 2],
[2, 1, 2, 1, 2, 2, 1],
[2, 1, 2, 1, 0, 2, 2],
[2, 2, 1, 2, 2, 0, 1],
[1, 1, 2, 2, 0, 2, 2],
[2, 2, 1, 1, 2, 2, 1],
[2, 2, 2, 1, 0, 2, 1],

$[2, 2, 1, 2, 0, 0, 2]$,
$[2, 1, 1, 2, 2, 2, 1]$,
$[1, 2, 1, 2, 0, 2, 2]$,
$[2, 2, 1, 1, 2, 0, 2]$,
$[2, 1, 1, 2, 2, 2, 1]$,
$[1, 1, 2, 1, 2, 2, 2]$,
$[1, 2, 2, 2, 0, 0, 2]]$

Se presenta un problema de 7 variables y 14 cláusulas. La solución se almacena en el archivo de soluciones y se codifica de la siguiente manera:

$[1, 1, 1, 1, 0, 0, 1]$

Por último, en el problema Knapsack, cada línea del archivo contiene una lista de tuplas, la cual representa la toatlidad de los ítems con sus respectivos pesos y utilidades:

$[(64, 425), (32, 388), (16, 306), (8, 85), (4, 5), (2, 94), (1, 202)]$

Se muestra un problema de 7 ítems y 526 de capacidad. La solución se almacena en el archivo de soluciones y se codifica de la siguiente manera:

$[1, 0, 0, 1, 1, 0, 0]$

4.5.1 Interfaz

La interfaz se realizó con PyGTK. **PyGTK** es un conjunto de módulos que componen una interfaz Python para GTK. **GTK** (GIMP Toolkit) es una biblioteca para crear interfaces de usuario gráficas. Está licenciada usando la licencia LGPL, por lo que se puede desarrollar software abierto, software libre, o incluso software comercial no libre usando GTK sin tener que pagar nada en licencias o derechos.

Se llama el toolkit de GIMP porque originariamente fue escrita para desarrollar el Programa de Manipulación de Imágenes GNU (GIMP), pero GTK se usa ahora en un amplio número de proyectos de software, incluyendo el proyecto de Entorno de Modelo de Objetos orientados a Red (GNOME). GTK está diseñada sobre GDK (Kit de Dibujo de GIMP) que básicamente es una abstracción de las funciones de bajo nivel para acceder a las funciones del sistema de ventanas (Xlib en el caso del sistema de ventanas X).

La interfaz se compone de los siguientes elementos:

- Menú. Contiene el menú de ayuda, en el cual se encuentra un pequeño manual y la información de la versión y el creador de la aplicación.
- Área de dibujo. Permite observar las gráficas obtenidas en la ejecución de los algoritmos genéticos.
- Área de texto. Permite observar las respuestas de la aplicación; es decir, se detalla cada paso que se esté realizando en la ejecución de una acción.

A continuación se presenta una foto de la interfaz y el procedimiento que se debe realizar para generar los problemas de tipo K-SAT.
Para generar los problemas, primero se debe escoger el tipo de problema que se desea generar -en este caso se escoge K-SAT-, luego se escoge la opción "Generar Problemas" y finalmente se oprime el botón "Ejecutar".

Como puede observarse en la gráfica, en la parte inferior se encuentra un área de texto que informa el lugar donde quedaron guardados los archivos con los problemas y su respectiva solución.

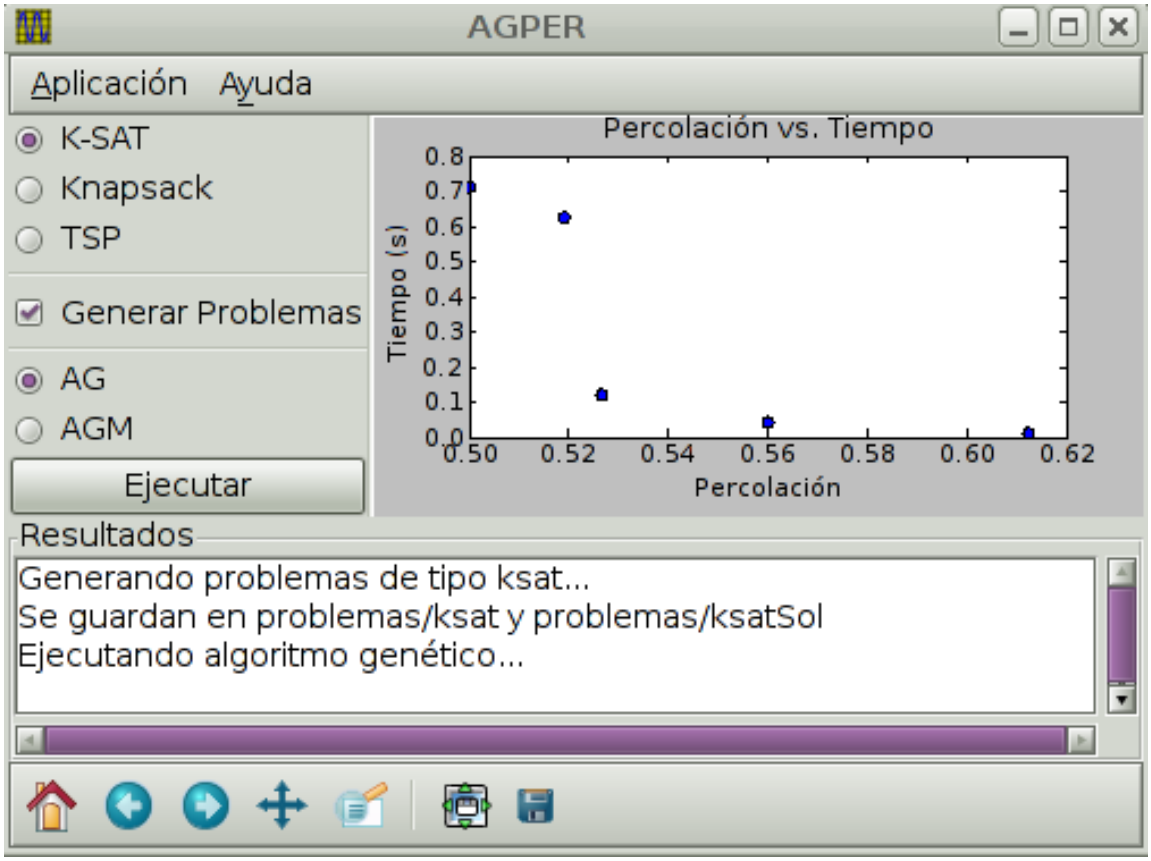


Figura 4.13: Generando problemas tipo K-SAT

Capítulo 5

Diseño de las Funciones de Aptitud

Es preciso codificar los algoritmos en un lenguaje que realice operaciones de manera rápida y eficiente. Python es un lenguaje interpretado, por ello en cálculos complejos o con gran número de iteraciones puede tardar mucho más tiempo que un lenguaje compilado. Sin embargo, Python se construyó sobre lenguaje C y se hará uso de éste último para escribir los módulos que contendrán a los algoritmos genéticos, permitiendo de esta forma extender el interpretador de Python.

Es posible añadir módulos internos a Python programados en C. Tales módulos extienden Python: permitiendo implementar nuevos tipos de objetos internos, hacer llamadas a funciones de C y al sistema. Para facilitar el trabajo se hizo uso del lenguaje Pyrex. [Ash05]

Pyrex es un lenguaje para escribir módulos de Python, desarrollado por Greg Ewing. El lenguaje Pyrex es un gran subconjunto de Python, con una semántica ligera menos dinámica que la de Python, y le añade declaraciones opcionales de tipos de parámetos y variables, que permiten al compilador de Pyrex generar rapidamente código en C. De esta forma, programar los algoritmos genéticos en Pyrex es casi tan sencillo como si lo hiciera con Python, con la ventaja de que el código que se ejecutará estará en lenguaje C y sus operaciones se realizarán mucho más rápido. Por ejemplo, el siguiente fragmento de cdigo es la cabecera de una clase hecha cpn Pyrex:

```
cdef class Agksat:
  """Clase que implementa el Algoritmo Genetico que soluciona problemas
     de tipo K-SAT."""

  cdef int num_variables, valor_variable, contador
  cdef float aptitud

  def __new__(self):
    self.num_variables = 0
    self.valor_variable = 0
    self.aptitud = 0.0
    self.contador = 0

  def iniciar(self, problema, solucion):
    """Metodo que inicia el algoritmo."""
```

El archivo realizado con Pyrex se compila de la siguiente forma:

pyrexc <nombrearchivo>.pyx
gcc -I/usr/include/python2.5 -c <nombrearchivo>.c
gcc <nombrearchivo>.o -fpic -shared -o <nombrearchivo>.so

El primer comando compila el archivo de Pyrex y de esa forma se obtiene una versión en lenguaje C. El segundo comando compila el archivo en lenguaje C para obtener un archivo Objeto. Por último se compila el archivo objeto para generar el archivo que se puede importar en Python.

A continuación se explica de manera general el Algoritmo Genético, con su respectivo diagrama de flujo, que se utiliza para resolver los problemas: Traveling Salesman Problema, K-SAT y Knapsack.

5.1 Algoritmo Genético (AG)

Generar Población. Se define una población de 100 cromosomas.

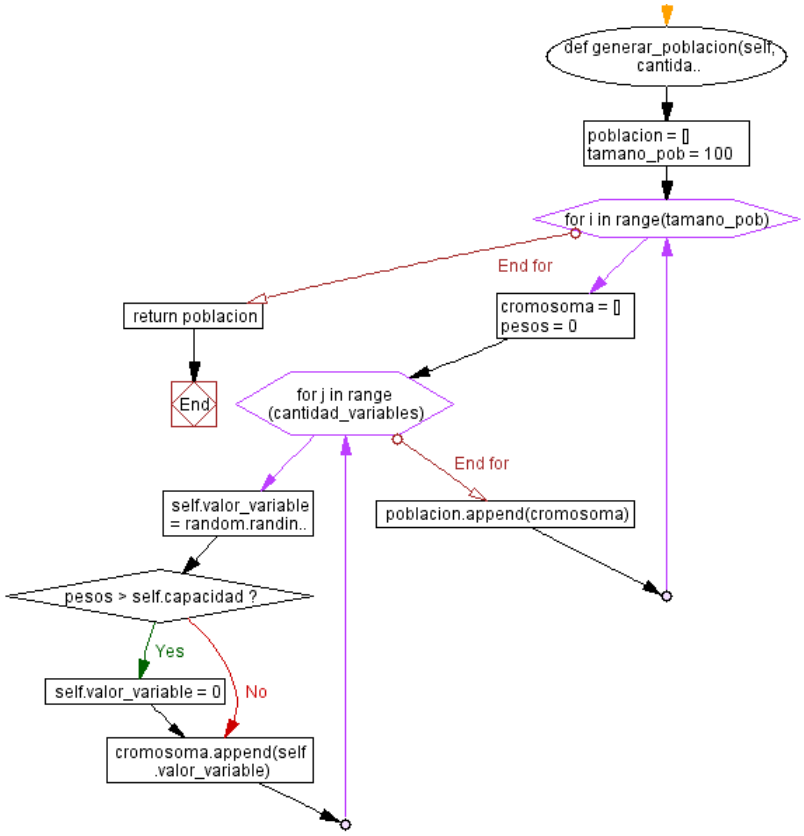


Figura 5.1: Generar Población

Cálculo de Percolación. Se hace el cálculo de la percolación al inicio del algoritmo. La explicación del cálculo se mostrará mas adelante en la sección Algoritmos Genéticos Modificados.

Selección de cromosomas. Se determina el número de cromosomas que se van a seleccionar, en este caso se decidió seleccionar el 60% de la población. Se escoge la forma de selección ruleta, por su fácil implementación y rápidez.

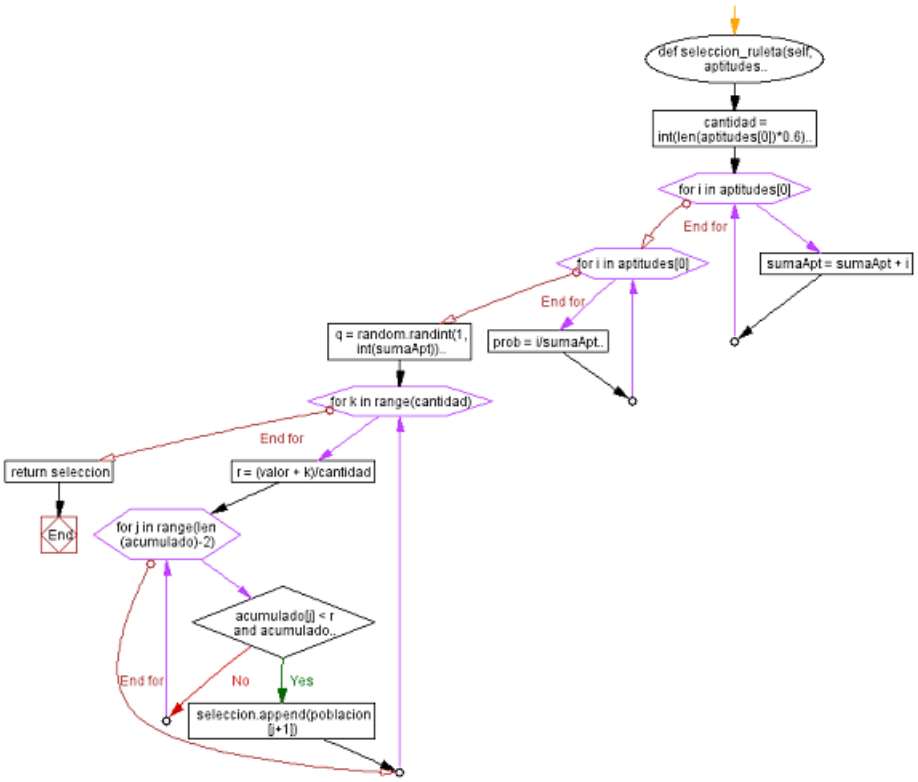


Figura 5.2: Selección de Cromosomas

Reproducción. En la fase de reproducción, se parte la población seleccionada a la mitad, y se toman los cromosomas de la primera mitad para cruzarlos con los de la segunda mitad. Se aplican los dos operadores de reproducción: Cruce y Mutación. Finalmente el grupo de hijos es retornado por la función de reproducción.

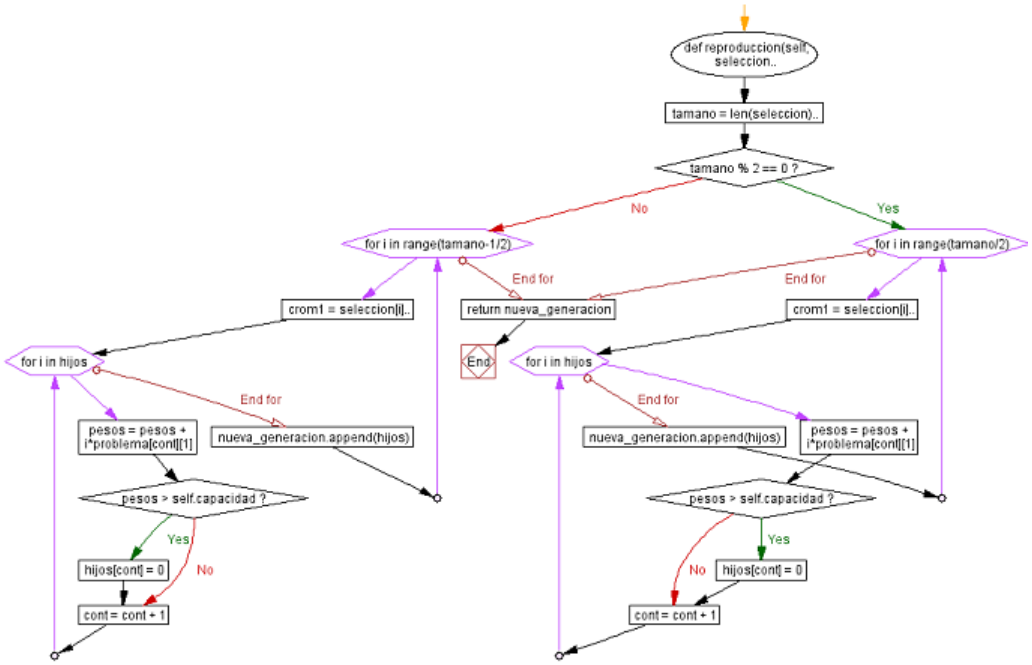


Figura 5.3: Reproducción de Cromosomas

Reemplazo. Luego de haber realizado la reproducción, se realiza el reemplazo de los cromosomas. En este caso se generan k números aleatorios desde 0 hasta el tamaño de la población de cromosomas, dicho número representa la posición en la lista de cromosomas y por lo tanto el cromosoma que se encuentre en esa posición será reemplazado por uno de los de la nueva generación.

Llevado a cabo el reemplazo, se vuelven a realizar todos los pasos desde el cálculo de la aptitud hasta que se encuentre la solución óptima o el algoritmo termine de iterar y devuelva la solución aproximada.

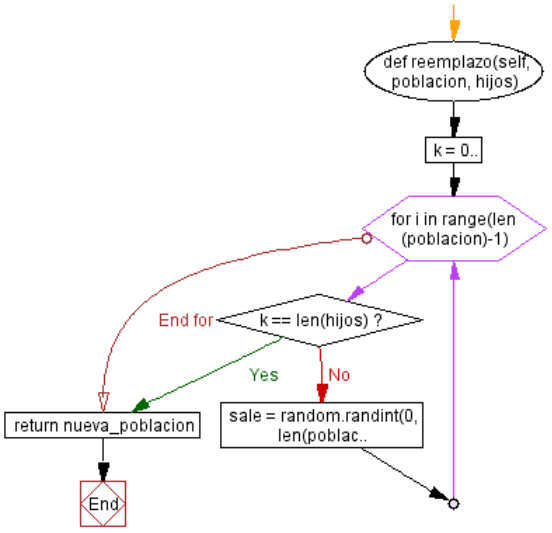


Figura 5.4: Reemplazo de Cromosomas

A continuación se presentan las particularidades del AG para cada problema.

5.2 AG Traveling Salesman

Codificación. Como primera medida se buscó la mejor forma de codificar el cromosoma. Debido a que la solución de este problema en particular debe ser la secuencia de ciudades que debe visitar el vendedor, la mejor forma de codifcarla es con una lista que contenga números enteros. Los números representan a la ciudad que se ha visitado y en qué momento se ha visitado. Por ejemplo si se tienen 5 ciudades y la ruta óptima es primero visitar la tercera ciudad, luego la segunda, la primera, la cuarta y por último la quinta, entonces el cromosoma con la solución debe ser así: [3,2,1,4,5].

Generar Población. Se generan los cromosomas obtieniendo valores aleatorios entre 1 y la cantidad de ciudades, verificando que dichos números no se repitan dentro del cromosoma.

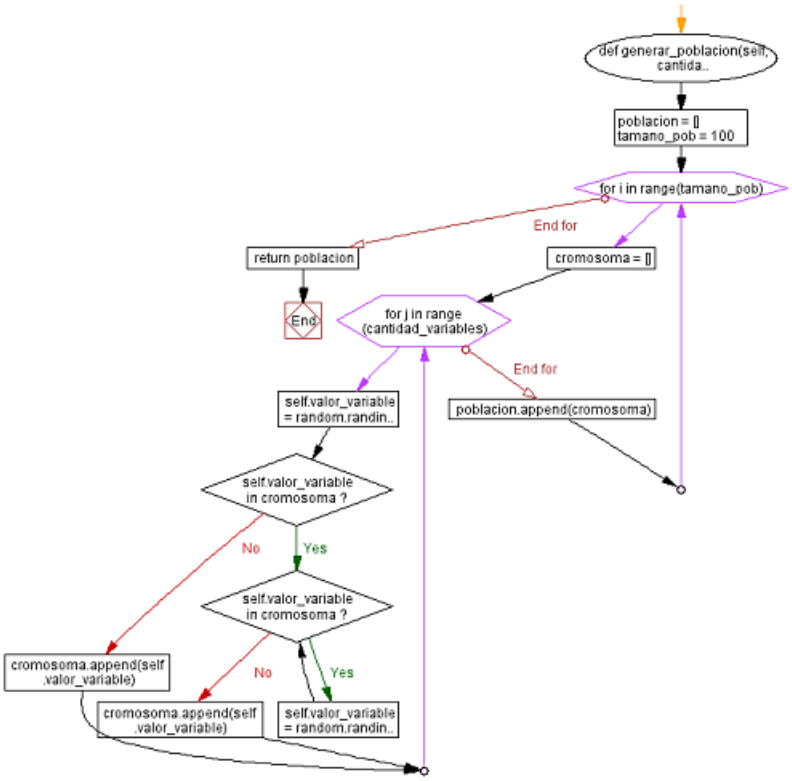


Figura 5.5: Generación de la Población TSP

Función de aptitud. Después de haber generado la población de cromosomas, se procede a evaluar cada uno de ellos, calculando la distancia del recorrido que representan. Luego de calcular las distancias de los recorridos de todos los cromosomas, se obtiene el menor valor y a partir de ese valor se hace la calificación de cada cromosoma de la siguiente forma: $menorValor/distanciaRecorrido[i]$. De esa manera obtenemos valores entre 0 y 1 que determinarán que tan bueno es el cromosoma. La menor distancia encontrada y el cromosoma a la que pertence se guardan en variables temporales y si después de una cantidad de iteraciones considerable este valor

no cambia, el algoritmo termina y retorna el cromosoma cuya distancia es la menor encontrada. Cabe anotar que para problemas de N grande, es muy poco probable que el algoritmo encuentre la solución, en cuyo caso el algoritmo retorna una lista vacía.

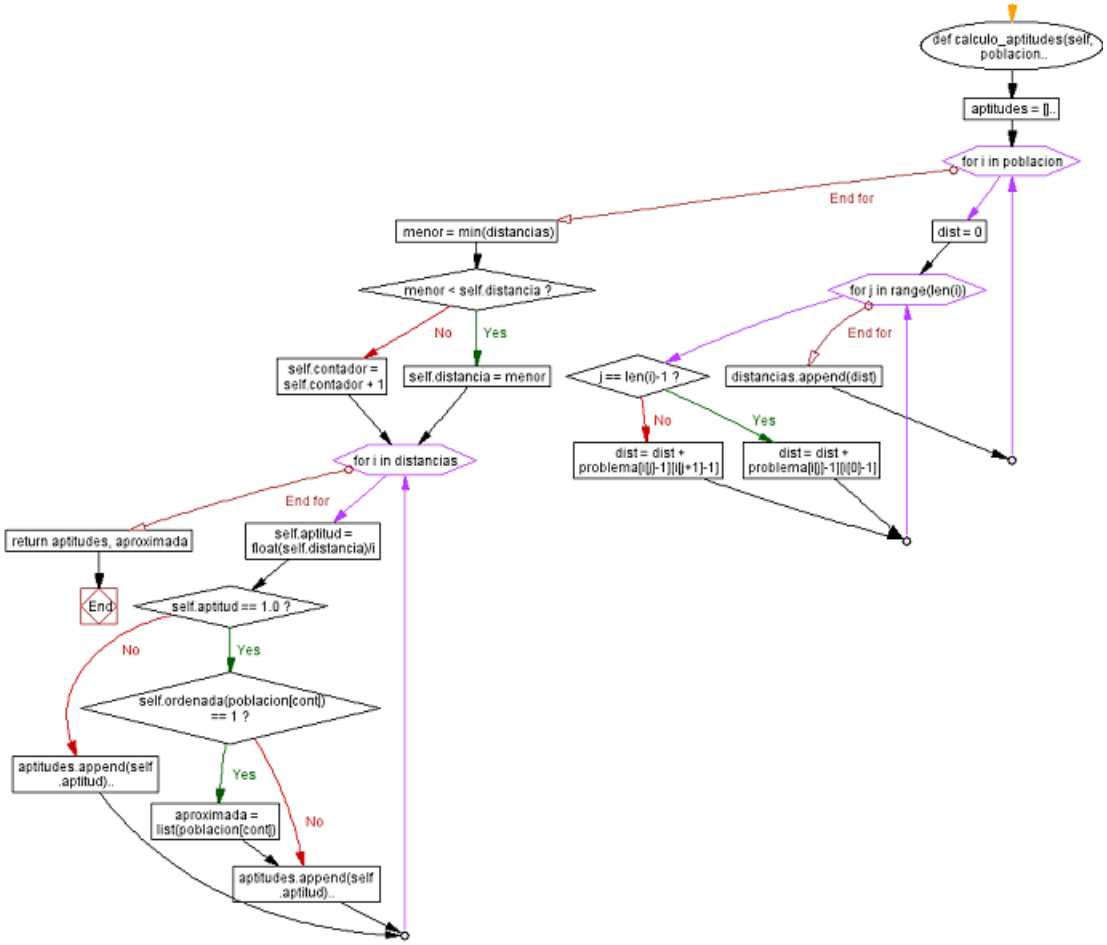


Figura 5.6: Función de Aptitud TSP

Reproducción.

- **Cruce:** para llevar a cabo el cruce de cromosomas con números enteros, fue necesario examinar el cromosoma durante la mezcla para ver si aparecen números

repetidos. En ese caso se genera otro número y se verifica que no esté dentro del cromosoma. Este procedimiento se hace hasta que finalice el cruce, de tal forma que los cromosomas hijos puedan representar una solución válida para el problema.

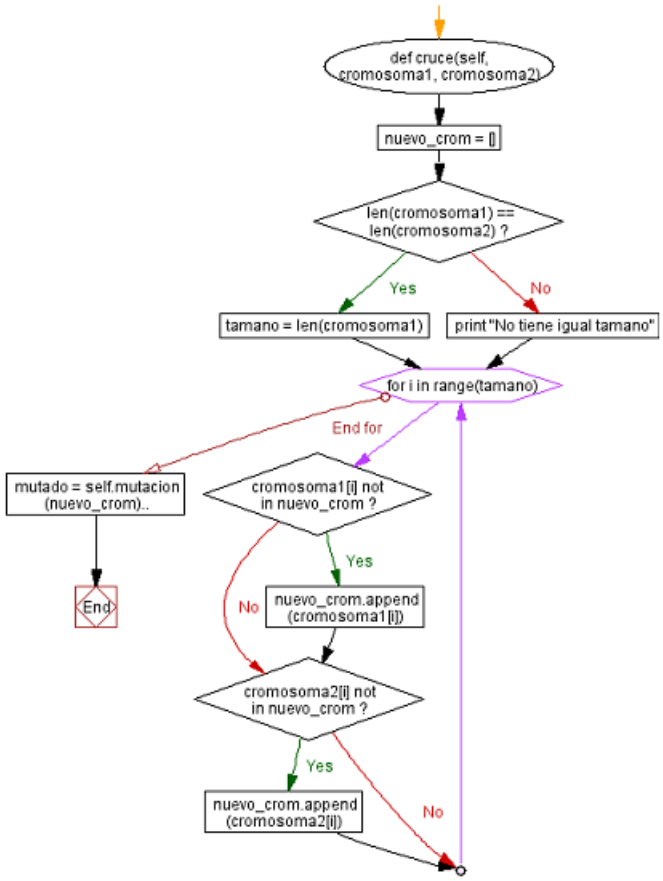


Figura 5.7: Cruce TSP

- **Mutación:** con la mutación se hizo un proceso parecido al del cruce, solo que en este caso es mucho más sencillo porque solo se escogen 2 genes del cromosoma al azar y se intercambian posiciones.

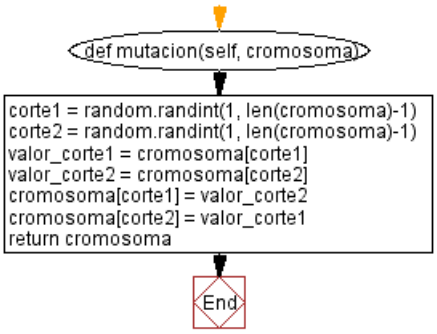


Figura 5.8: Mutación TSP

5.3 AG K-SAT

Codificación. Debido a que la solución de este problema en particular debe ser una lista con los valores que deben tomar las variables booleanas para cumplir con las cláusulas, se decidió utilizar una lista con valores 1 y 0; 1 representa el valor Verdadero y 0 el valor Falso, obteniendo cromosomas de la siguiente forma: [1,0,1,1,0,0].

Generar Población. Se generan los cromosomas con valores aleatorios de 1 y 0.

Función de aptitud. Luego de haber generado la población de cromosomas, se evalúan cada uno de ellos. Primero se toma el cromosoma y se evalúa cada cláusula del problema con la función booleana $\vee$, después se mira cuántas cláusulas fueron satisfechas y ese número se divide entre el total de cláusulas: $clausulasSatisfechas/\#totalClausulas$. Obteniendo de esa forma la calificación o aptitud del cromosoma. En este tipo de problemas suele presentarse que varios cromosomas pueden satisfacer las cláusulas, por ello si la aptitud es igual a 1.0, el algoritmo termina y retorna el cromosoma encontrado a pesar de que no sea el propuesto por el generador.

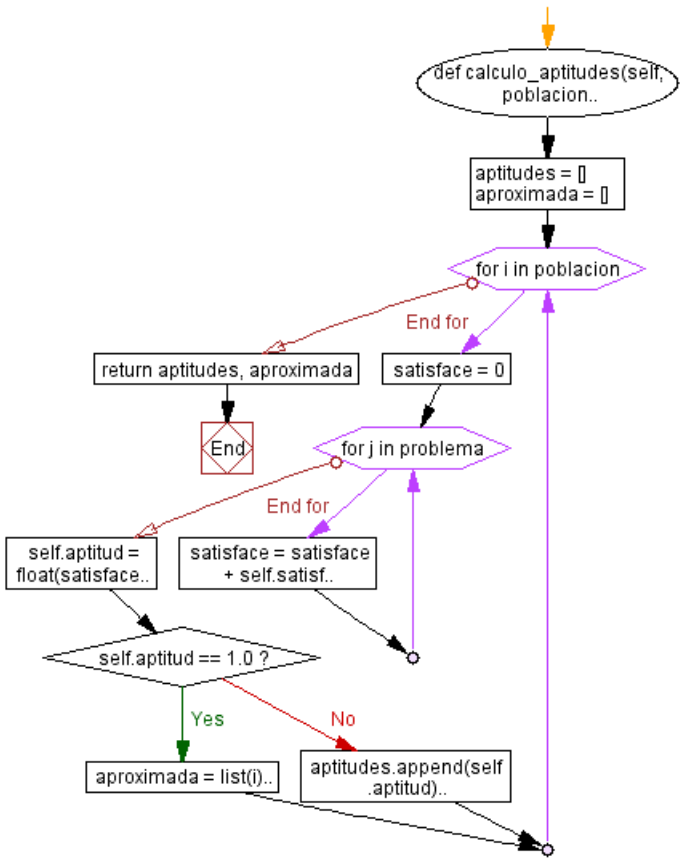


Figura 5.9: Función de Aptitud K-SAT

Reproducción.

- **Cruce:** se escoge un número al azar entre 1 y el número de variables y se parten el par de cromosomas por ese valor, resultando finalmente 2 cromosomas hijos que contendrán la primera parte del padre con la segunda parte de la madre y viceversa.

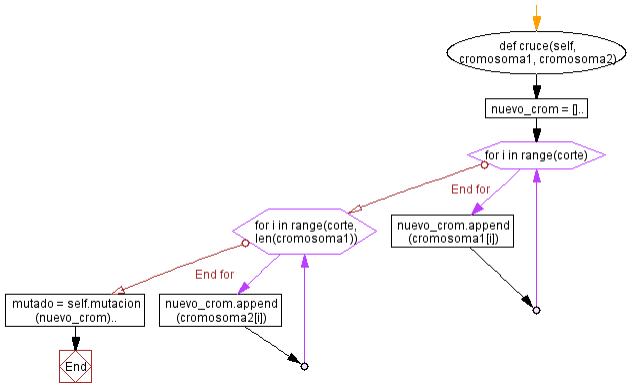


Figura 5.10: Cruce K-SAT

- **Mutación:** se escoge un número al azar entre 1 y el número de variables y se mira el valor del gen en esa posición: si el valor es 1 se le cambia 0 y si es 0 se cambia a 1.

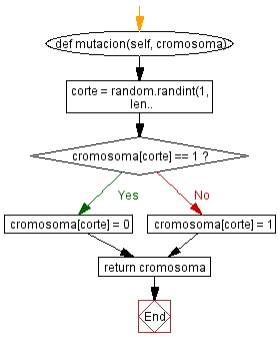


Figura 5.11: Mutación K-SAT

5.4 AG Knapsack

Codificación. En este problema la solución debe representar los ítems que fueron empacados en la mochila, por ello se decidió utilizar una lista con valores 1 y 0: 1 representa que el ítem fue introducido en la mochila y 0 que no, obteniendo cromosomas de la siguiente forma: [1,0,1,1,0,0].

Generar Población. Como en este problema en particular se debe verificar que se cumpla la restricción de no sobrepasar la capacidad de la mochila, se tomó la decisión de generar cromosomas que cumplan con esta condicón, evitando posibles combinaciones que no sean factibles.

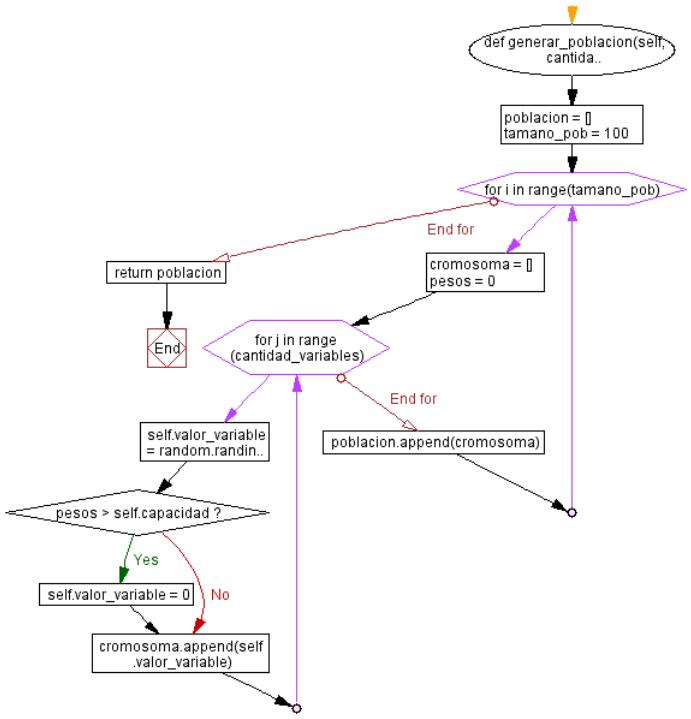


Figura 5.12: Generar Población Knapsack

Función de aptitud. La evaluación de cada cromosoma se hace partiendo del hecho de que ya cumplen con la condición de no sobrepasar la capacidad de la mochila, por lo tanto resta calcular las utilidades que dejan cada uno de los cromosomas y a partir de esas utilidades obtener el mayor valor para hacer el respectivo cálculo de las aptitudes, de la siguiente forma: $utilidad[i]/mayorUtilidad$. De esa manera se obtienen valores entre 0 y 1 que determinarán que tan bueno es el cromosoma. La mayor utilidad encontrada y el cromosoma a la que pertence se guardan en variables temporales y si después de una cantidad de iteraciones considerable este valor no cambia, el algoritmo termina y retorna el cromosoma cuya utilidad es la mayor encontrada.

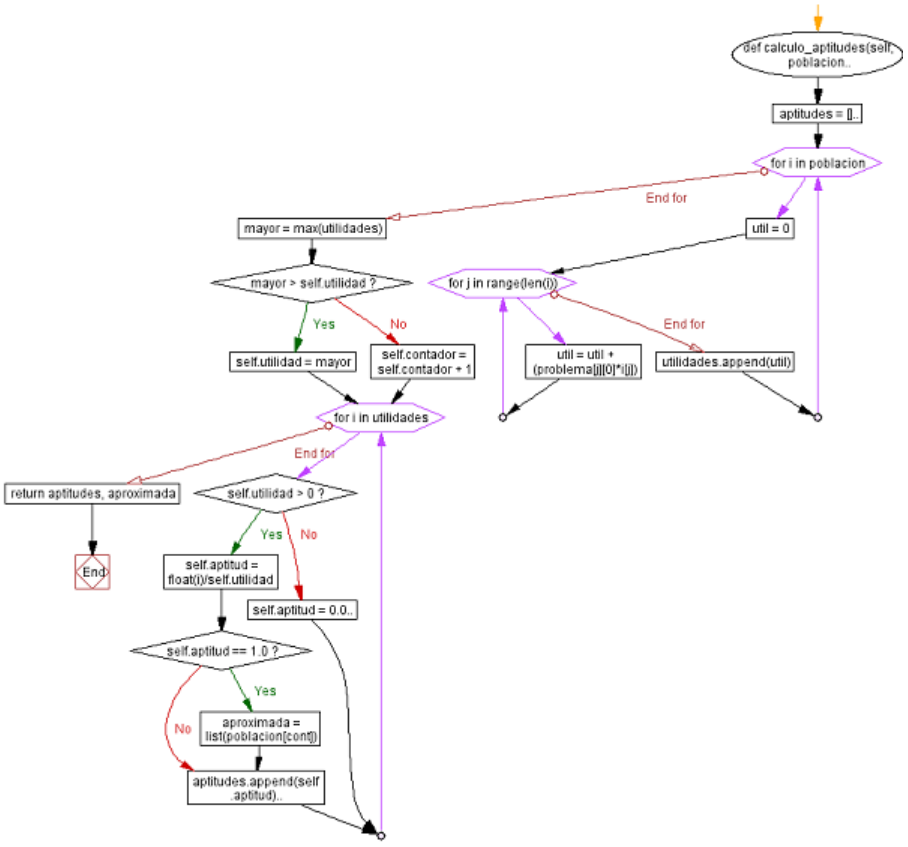


Figura 5.13: Función de Aptitud Knapsack

Reproducción. Por ser un problema con 2 restricciones, para realizar las operaciones de cruce y mutación, se agregó la condición de que los hijos cumplieran con una de las condiciones del problema, la cual es no sobrepasar la capacidad de la mochila. De esta forma se vuelve a obtener una población factible igual que la inicial.

5.5 Algoritmos Genéticos Modificados (AGM)

Generar Población. Al igual que en los AG anteriormente explicados, se genera una población de tamaño 100.

Cálculo Percolación. Se realiza el cálculo del parámetro de percolación, el cual es medido a partir de la población inicial, escogiendo unos cuantos cromosomas y generando pequeños cambios en todas sus dimensiones; es decir, para un cromosoma de tamaño N se deben generar N nuevos cromosomas. De los cromosomas generados se mira cuáles son soluciones factibles F y se calculan sus respectivas aptitudes, luego se comparan con la aptitud del cromosoma original para ver cuántas lo superaron M, de ahí se obtiene que la percolación será igual a $P = M/N$. Este parámetro sirve como indicador para ampliar o disminuir el espacio de soluciones.

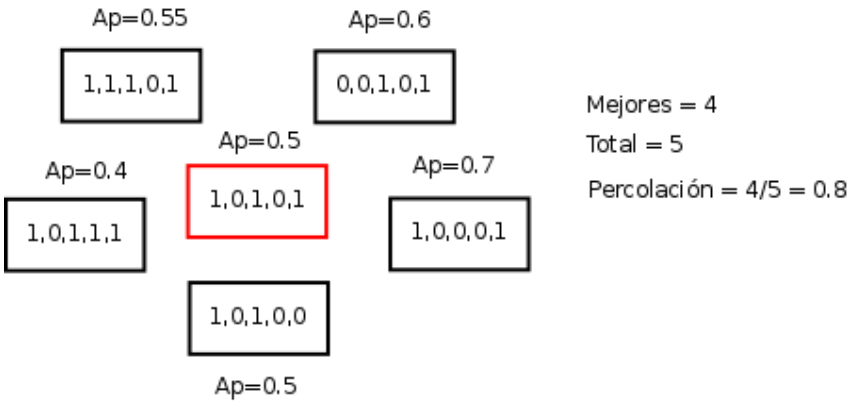


Figura 5.14: Cálculo de Percolación

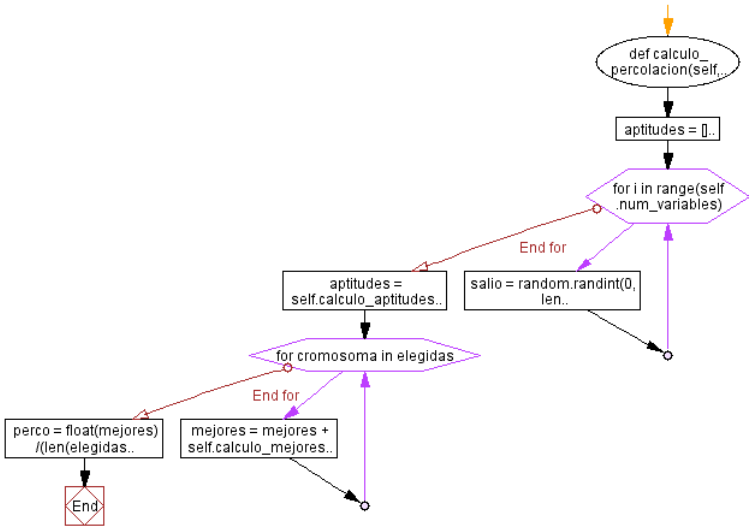


Figura 5.15: Diagrama de Flujo Cálculo de Percolación

Cálculo Aptitudes. Se calculan las aptitudes igual que en los AG.

Selección de cromosomas. Lo primero que se hace es calcular nuevamente la percolación y se compara con el valor de percolación calculada al inicio del algoritmo. Si la nueva percolación es menor, la cantidad de cromosomas para seleccionar se incrementa en un 10%; es decir, se seleccionan 70% de los cromosomas de la población. Si la nueva percolación es mayor, la cantidad de cromosomas para seleccionar disminuye en un 10%, seleccionando el 50% de la población. Si el valor es constante, entonces se selecciona el 60% de la población como en los AG. De esta forma se controla la presión selectiva. Luego se realiza el algoritmo de selección tipo ruleta y finalmente la función devuelve los cromosomas seleccionados.

La Reproducción y el Reemplazo no tienen ningún cambio, se hacen de igual forma que en los AG.

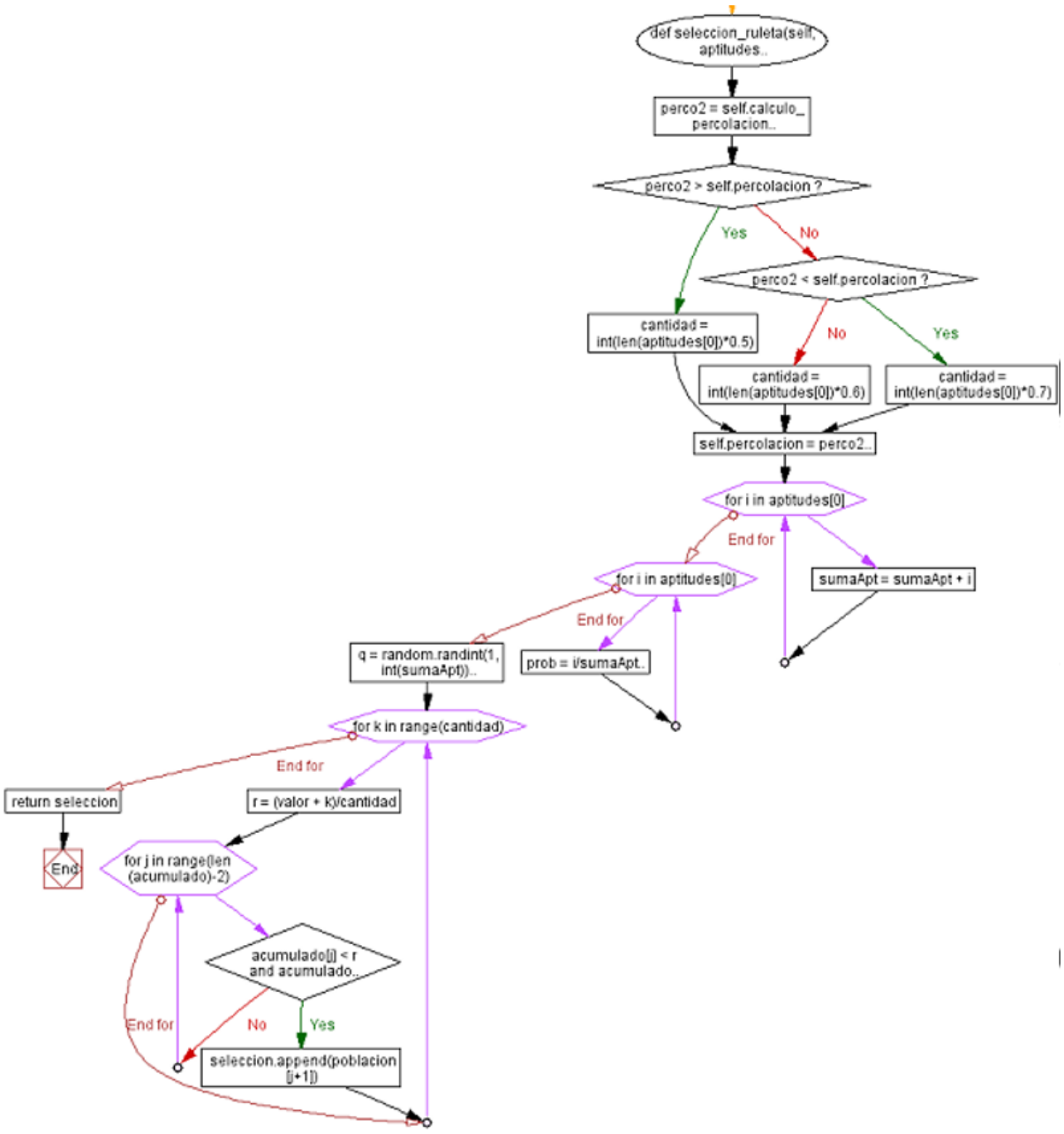


Figura 5.16: Selección AGM

Capítulo 6

Pruebas y Conclusiones

Para llevar a cabo las pruebas, se creó un archivo de pruebas, el cual itera 1000 veces sobre los archivos de problemas y soluciones, obteniendo de esta forma una pareja (problema, solución), la cual sirve como parámetro de entrada del respectivo algoritmo genético. Justo en ese momento se toma un tiempo inicial antes de ejcutar el algoritmo genético y un tiempo final cuando finaliza la ejecución del algoritmo, con la función **time** de Python, para obtener el tiempo que se demora el algoritmo en ejecutarse con los parámetros dados. Finalmente, se escriben en un archivo los siguiente datos: percolación, tiempo de ejecución, número de variables del problema, la solución encontrada en el algoritmo genético y la solución que se propone desde el generador de problemas. Esto con el fin de realizar una gráfica Percolación vs. Tiempo, mirar cuántos problemas fueron solucionados óptimamente y qué dificultad tienen dichos problemas.

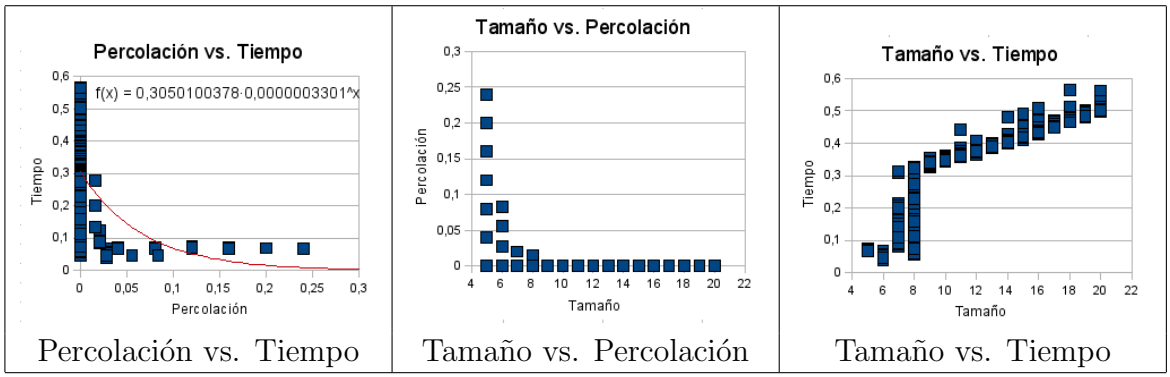
6.1 Pruebas con AG

6.1.1 Traveling Salesman

El archivo de problemas del TSP contiene matrices de tamaños 5 hasta 20. En la tabla 6.2 se muestra un fragmento de la tabla de datos y en la figura 6.1 los gráficos que se

Percolación	Tiempo(s)	Ciudades	Dist. AG	Dist. Gen.
0.12	0.0681579113	5	10.8284271247	10.8284271247
0.04	0.06930089	5	10.8284271247	10.8284271247
0.0204081633	0.0911099911	7	16.1289902045	14.8284271247
0.0204081633	0.1221249104	7	23.313708499	14.8284271247
0.015625	0.2787518501	8	32.4218096754	16.0
0.0	0.1107690334	8	30.2448349897	16.0
0.0	0.4203560352	15	84.265253327	30.8284271247

Tabla 6.1: Datos AG Traveling Salesman



Percolación vs. Tiempo | Tamaño vs. Percolación | Tamaño vs. Tiempo

Figura 6.1: Gráficas AG TSP

obtuvieron al comparar Percolación, Tamaño y Tiempo con la totalidad de problemas.

Como se puede apreciar en la figura 6.1, se observa un comportamiento exponencial, en donde a medida que disminuye la percolación se incrementa el tiempo de ejecución, y también a medida que crece el tamaño del problema aumenta el tiempo de ejecución. Esto implica que el problema es realmente difícil de solucionar. Para comprender mejor el comportamiento del algoritmo y del problema, se han realizado gráficas para los problemas de tamaño 5, 10, 15 y 20, figura 6.4.

Puede observarse en la gráfica de tamaño 5 que el parámetro de percolación es muy cercano a 0. Esto implica que para problemas de pocas ciudades la interconexión entre soluciones candidatas y posible navegación es demasiado pequeña. Sin embargo, los

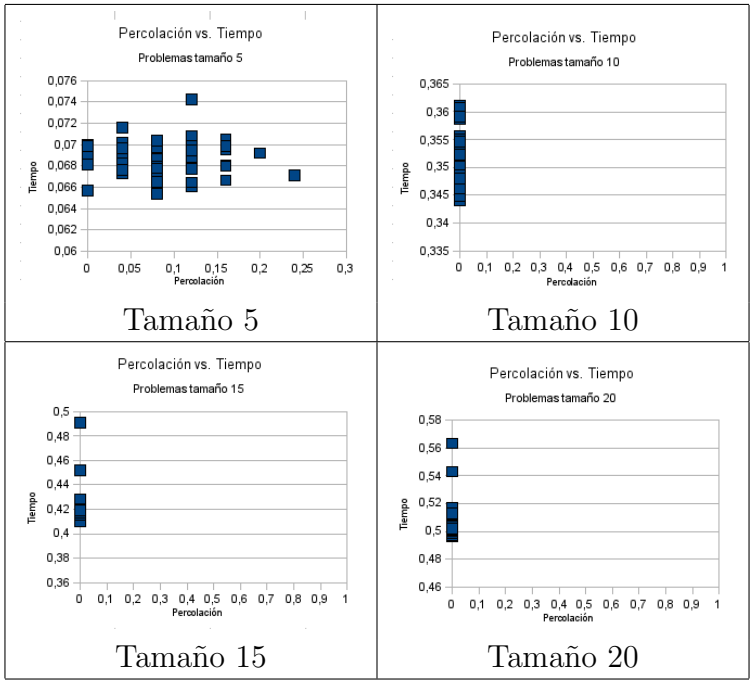


Figura 6.2: Gráficas AG TSP con distinto N

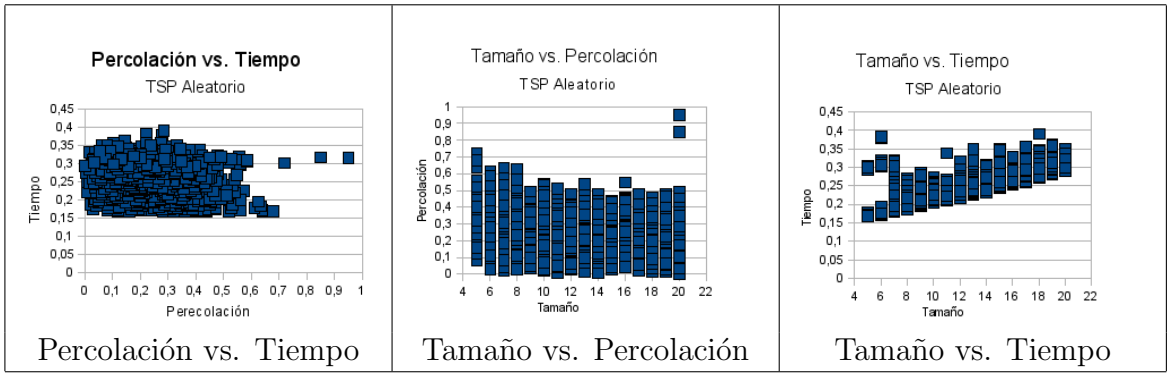
problemas de tamaños 5 hasta 8 tienen mayor probabilidad de ser solucionados por el AG, mientras que para problemas de más de 10 ciudades el AG no puede encontrar ninguna solución. Ésto se debe, a que el TSP es un problema de tipo combinatorio y además se debe presentar un orden específico en las soluciones para que tengan validez. Por ello, en las gráficas de problemas de tamaños 10 a 20 la percolación es 0 y el tiempo de ejecución crece debido a la cantidad de cálculos que el AG realiza para intentar encontrar una ruta óptima al problema de entrada.

Adicionalmente se realizaron pruebas utilizando otra configuración del problema TSP. En este caso se generaron ciudades aleatorias y se le aplicó el AG como en el caso anterior.
Se obtuvieron los siguientes datos y gráficas:

Como puede observarse en la figura 6.3, por ser instancias aleatorias del problema, se presenta mucha dispersión; sin embargo, se puede notar una semejanza en las gráficas

Percolación	Tiempo(s)	Ciudades	Dist. AG	Dist. Gen.
0.388888888889	0.303575992584	6	60.9482056631	68.0900730831
0.387755102041	0.194238901138	7	59	65.3040161241
0.204081632653	0.31929397583	7	63.9593706887	103.790859502
0.413223140496	0.337671041489	11	89	155.05275818

Tabla 6.2: Datos AG Traveling Salesman Aleatorio



Percolación vs. Tiempo | Tamaño vs. Percolación | Tamaño vs. Tiempo

Figura 6.3: Gráficas AG TSP Aleatorios

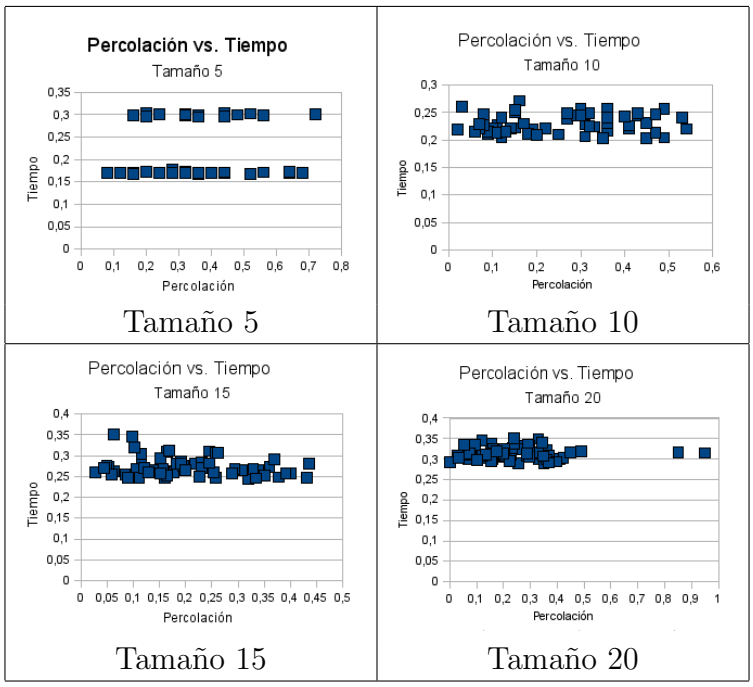


Figura 6.4: Gráficas AG TSP Aleatorio con distinto N

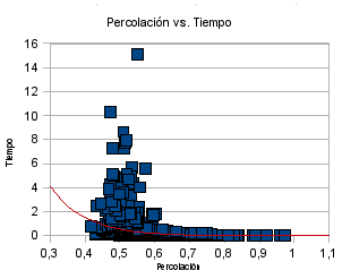
de Tamaño vs. Tiempo y Percolación vs. Tiempo con las de la prueba anterior. A continuación se mostrarán las gráficas por tamaños de problema para ilustrar su comportamiento.

Al igual que en el TSP de formas geométricas, se puede notar la relación entre el valor de percolación, el tamaño del problema y el tiempo que demora el AG en terminar de ejecutarse. A medida que el tamaño del problema aumenta la percolación disminuye e inclusive puede apreciarse que los datos se van agrupando cada vez más.

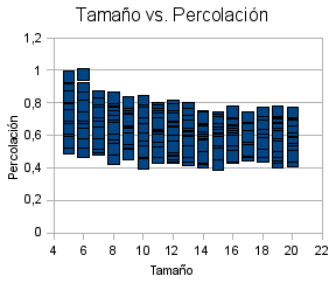
Cabe anotar que, por el hecho de tener instancias totalmente aleatorias del problema, en algunos casos se presentan valores de percolación altos a pesar de que la instancia sea de una gran cantidad de ciudades.

Percolación	Tiempo(s)	Variables	Cláusulas
0.88	0.0168409348	5	5
0.64	0.0604729652	5	17
0.6530612245	0.0600149632	7	17
0.5306122449	0.0843000412	7	45
0.4722222222	0.360986948	12	135
0.475	2.8231761456	20	374

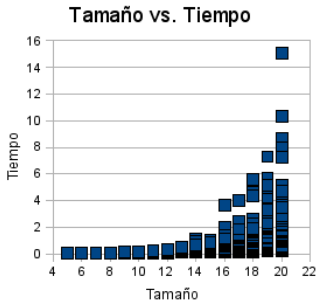
Tabla 6.3: Datos AG K-SAT



Percolación vs. Tiempo



Tamaño vs. Percolación



Tamaño vs. Tiempo

Figura 6.5: Gráficas AG K-SAT

6.1.2 K-SAT

Al igual que en TSP, los problemas de K-SAT contienen variables desde 5 hasta 20 y cláusulas desde 5 hasta 20^2. En la tabla 6.3 se muestra un fragmento de los datos y en el gráfico 6.5 se muestran las figuras que se obtuvieron al comparar Percolación, Tiempo y Tamaño con la totalidad de problemas.

Se puede apreciar en la figura 6.5 que el comportamiento de la curva es exponencial, a medida que la percolación disminuye el tiempo de ejecución aumenta, y entre más grande sea el problema más tiempo de ejecución consume. Para comprender mejor lo que sucede, se muestra a continuación la figura 6.6 con las gráficas para problemas de tamaños 5, 10, 15 y 20.

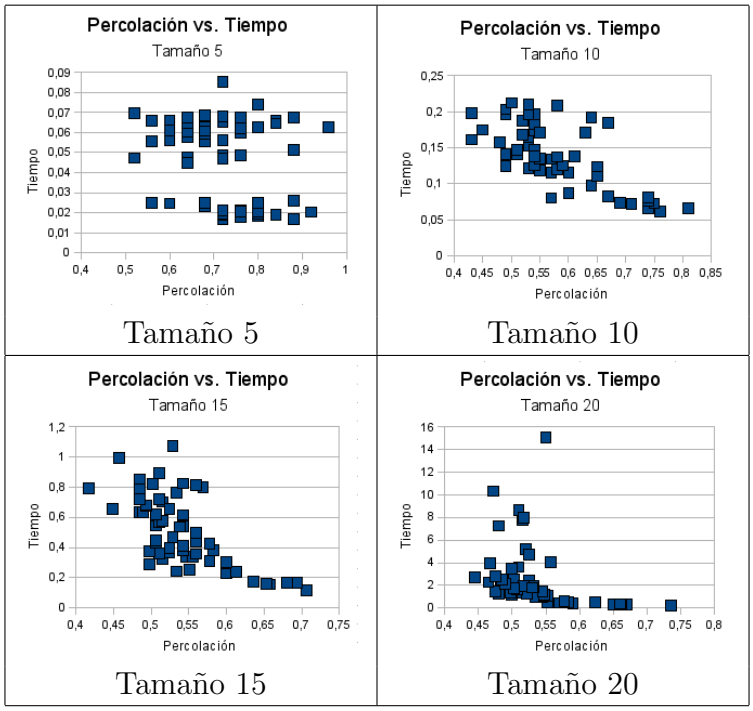


Figura 6.6: Gráficas AG K-SAT con distinto N

Percolación	Tiempo(s)	Ítems	Capacidad Mochila
0.16	0.047011137	5	66
0.1944444444	0.0502409935	6	172
0.125	0.054033041	8	371
0.0625	0.0647909641	12	336
0.1911111111	0.0683169365	15	423
0.07	0.0843670368	20	508

Tabla 6.4: Datos AG Knapsack

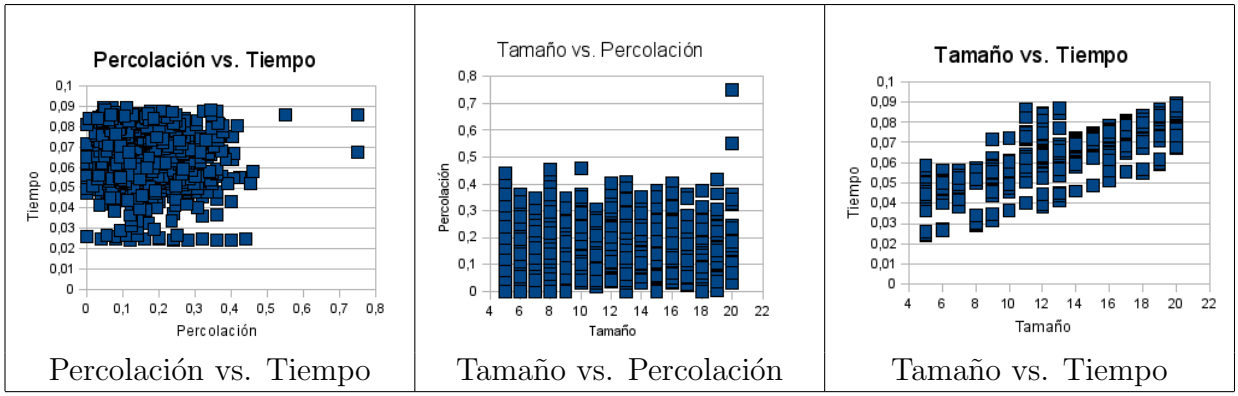
Como se puede observar, a pesar de que para problemas de tamaño pequeño pareciera no haber relación, a medida que crece el problema se comienza a notar la relación entre la dificultad del problema y el parámetro de percolación, observándose un comportamiento exponencial.

6.1.3 Knapsack

Se generaron problemas con una cantidad de ítems que va desde 5 hasta 20 y cuyos pesos se escogen aleatoriamente con valores entre 1 y la capacidad de la mochila. Adicionalmente se tiene una capacidad de la mochila que varía entre 50 y 1000. A continuación se muestran un fragmento de la tabla de datos y los gráficos que se obtuvieron al comparar Percolación, Tamaño y Tiempo con la totalidad de problemas, figuras 6.4 y 6.7 respectivamente.

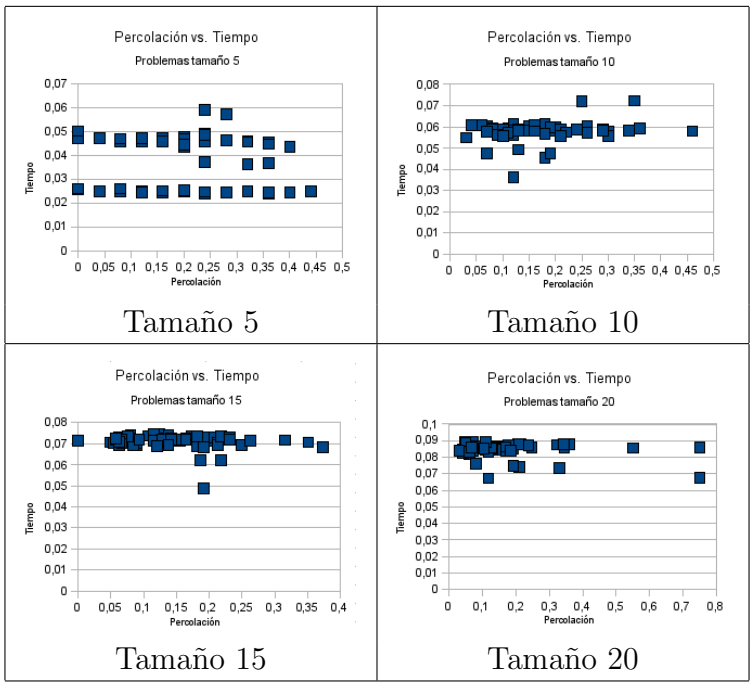
En la gráfica 6.7 puede apreciarse una gran dispersión en los datos; sin embargo, también puede observarse que la mayor parte de los datos se ubican en valores de percolación medio-bajos, por lo tanto, se muestra en la figura 6.8 las gráficas para problemas de tamaño 5, 10, 15 y 20, para mirar el comportamiento en cada una de las condiciones.

Se observa que en problemas de tamaño pequeño hay demasiada dispersión, y a medida que aumenta el tamaño, los datos se comienzan a agrupar mucho más. Para



Percolación vs. Tiempo | Tamaño vs. Percolación | Tamaño vs. Tiempo

Figura 6.7: Gráficas AG Knapsack



Tamaño 5 | Tamaño 10 | Tamaño 15 | Tamaño 20

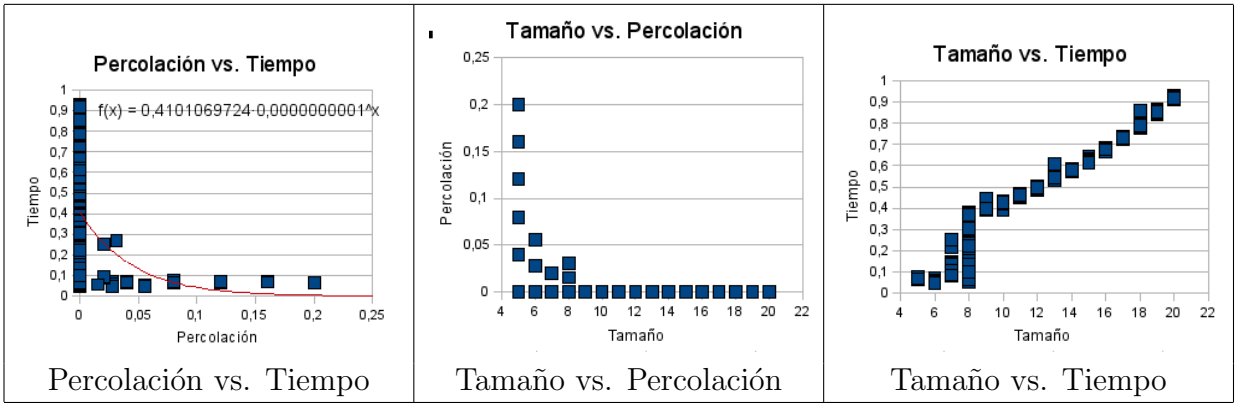
Figura 6.8: Gráficas AG Knapsack con distinto N

problemas pequeños el tiempo de ejecución no sobrepasa los 0.06 segundos y la percolación toma valores entre 0 y 0.45, mientras que para problemas grandes el tiempo de ejecución es cercano a 0.09 segundos y el valor de percolación se concentra entre 0 y 0.25.

A pesar de la gran dispersión en los datos de la figura 6.7, sigue evidenciándose la relación que existe entre el valor de la percolación y la dificultad del problema.

Percolación	Tiempo(s)	Ciudades	Dist. AG	Dist. Gen.
0.439999997616	0.367033004761	5	10.8284271247	10.8284271247
0.244897961617	0.267430067062	7	16.1289902045	14.8284271247
0.25	0.27928519249	10	46.018910286	20.0
0.0532544367015	0.297966957092	13	42.018910286	26.8284271247
0.0830449834466	0.360092878342	17	63.4986021069	34.8284271247

Tabla 6.5: Datos AGM Traveling Salesman



Percolación vs. Tiempo | Tamaño vs. Percolación | Tamaño vs. Tiempo

Figura 6.9: Gráficas AGM TSP

6.2 Pruebas con AGM

6.2.1 Traveling Salesman

Se realizaron las pruebas con los mismas instancias del problema que en las anteriores pruebas de AG. En la tabla 6.5 se pueden apreciar algunos datos de percolación y tiempo, y en la figura 6.9 se aprecian las gráficas que arroja la comparación percolación, tamaño y tiempo.

Al igual que en las pruebas realizadas con los algoritmos genéticos tradicionales, se observa un comportamiento exponencial, en donde a medida que disminuye la percolación se incrementa el tiempo de ejecución y a medida que crece el tamaño del problema aumenta el tiempo de ejecución. Esto implica que el problema es realmente

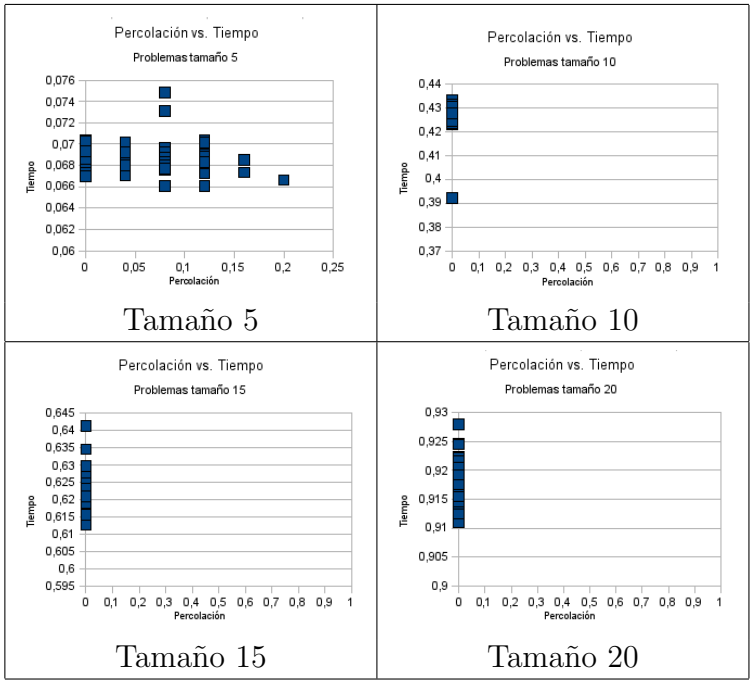


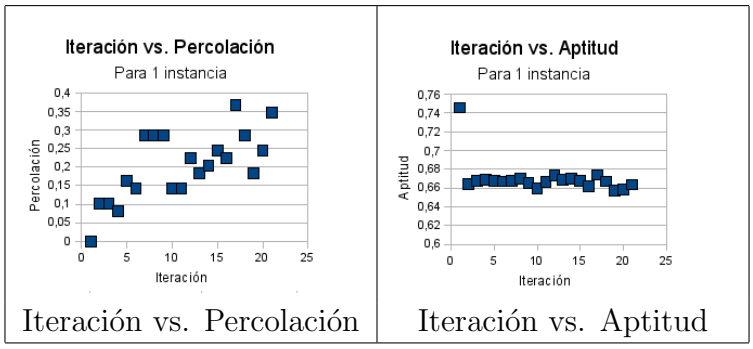
Figura 6.10: Gráficas AGM TSP con distinto N

difícil de solucionar. Sin embargo, el tiempo de ejecución aumentó considerablemente. Para comprender mejor el comportamiento del algoritmo y del problema, se han realizado gráficas para los problemas de tamaño 5, 10, 15 y 20, figura 6.10.

El único cambio con respecto a la prueba AG, es el aumento en el tiempo de ejecución. En la sección AG vs. AGM, se explicará la razón de este suceso.

Dist. AGM	Dist. Gen.
18.6011261595	14.8284271247

Tabla 6.6: Distancias para 1 instancia de TSP



Iteración vs. Percolación | Iteración vs. Aptitud

Figura 6.11: Gráficas AGM TSP para 1 instancia

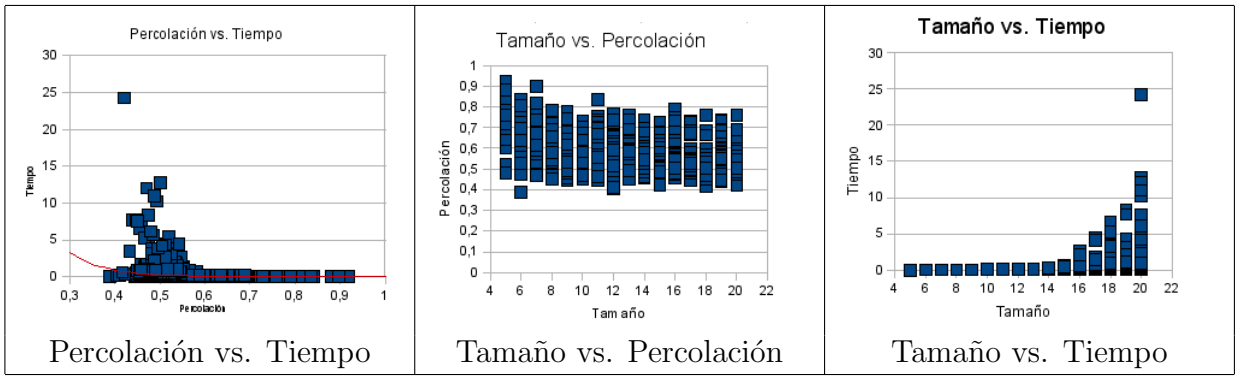
Adicionalmente se realizó el seguimiento en el proceso de ejecución del AGM con una instancia de 7 ciudades y se observa lo siguiente:
Se puede observar que a medida que el AGM se ejecuta, la percolación aumenta; mientras que, la aptitud se mantiene casi constante. Esto implica, que a pesar de que se navegó por el espacio de soluciones, el AGM no pudo encontrar la solución óptima al problema; sin embargo, para este caso, el AGM encontró una solución que se aproxima mucho a la óptima.

6.2.2 K-SAT

Se tomaron los mismos datos de la prueba con AG, y se obtuvieron los siguientes resultados consignados en la tabla 6.7 y la figura 6.12.

Percolación	**Tiempo(s)**	**Variables**	**Cláusulas**
0.8000000119	0.0056869984	5	5
0.6111111045	0.0122888088	6	22
0.5308641791	0.0585551262	9	81
0.506944418	0.1587290764	12	135
0.48046875	0.5255019665	16	247
0.4199999869	24.2304182053	20	377

Tabla 6.7: Datos AGM K-SAT



Percolación vs. Tiempo | Tamaño vs. Percolación | Tamaño vs. Tiempo

Figura 6.12: Gráficas AGM K-SAT

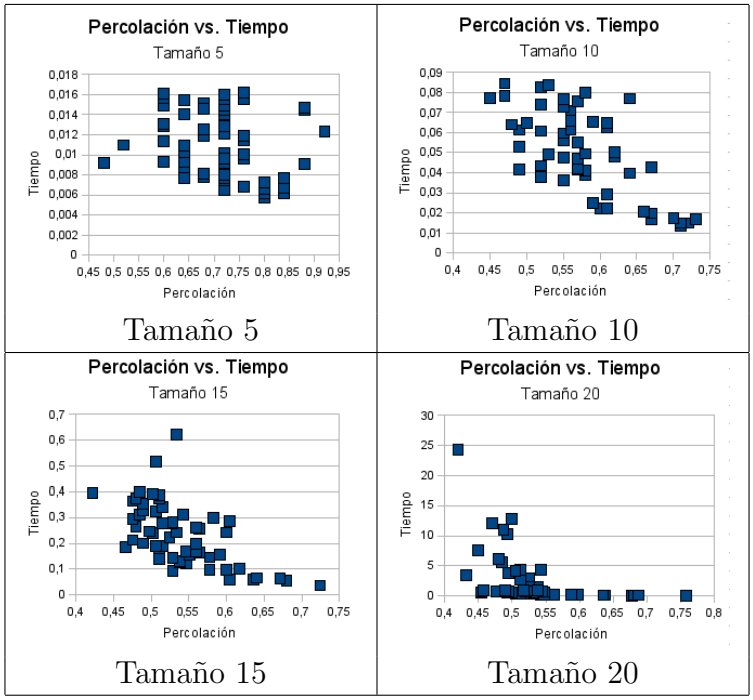
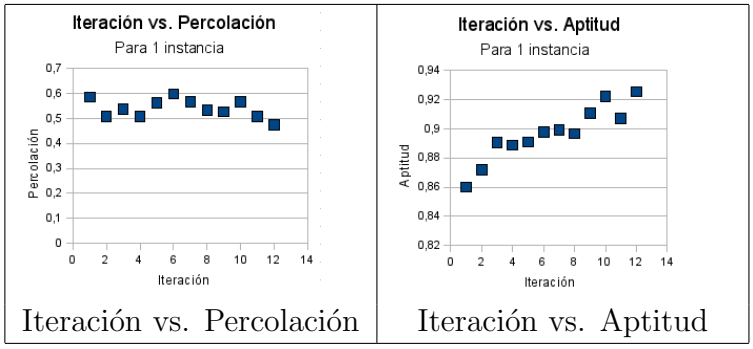


Figura 6.13: Gráficas AGM K-SAT con distinto N

Aparentemente no se nota ningún cambio relevante con respecto a la gráfica obtenida en la prueba de AG. Sin embargo, para comprender mejor lo que sucede, se muestra a continuación la figura 6.13 con las gráficas para problemas de tamaños 5, 10, 15 y 20.

Al analizar las gráficas con más detalle, se puede ver que los problemas de tamaños 5 hasta 15 se solucionan en menos de 1 segundo; solo a partir de problemas de más de 15 variables, el tiempo comienza a crecer. Este comportamiento se debe a que los problemas de gran tamaño se vuelven más difíciles de resolver.



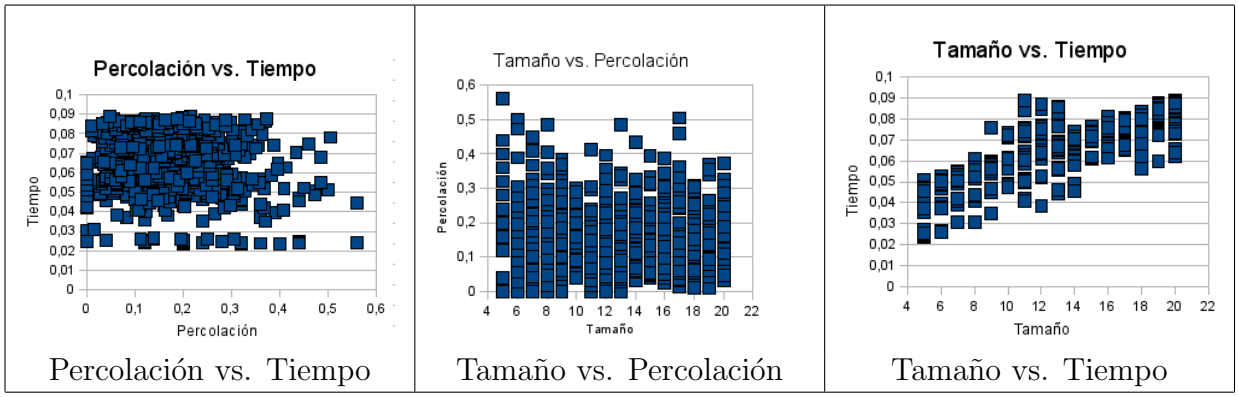
Iteración vs. Percolación | Iteración vs. Aptitud

Figura 6.14: Gráficas AGM K-SAT para 1 instancia

Adicionalmente se realizó el seguimiento en el proceso de ejecución del AGM con una instancia de 13 variables y 85 cláusulas y se observa lo siguiente:
A diferencia del TSP, en este caso se presenta el siguiente fenómeno: mientras que la percolación varía en busca de la solución óptima del problema, la aptitud se va mejorando cada vez más. Por ello, el AGM es capaz de encontrar soluciones óptimas más rápido que el AG tradicional.

Percolación	Tiempo(s)	Ítems	Capacidad Mochila
0.1199999973	0.0361599922	5	66
0.027777778	0.0485239029	6	172
0.2716049254	0.0596561432	9	346
0.2396694273	0.073941946	11	639
0.0666666701	0.0730290413	15	900
0.150000006	0.084717989	20	799

Tabla 6.8: Datos AGM Knapsack



Percolación vs. Tiempo | Tamaño vs. Percolación | Tamaño vs. Tiempo

Figura 6.15: Gráficas AGM Knapsack

6.2.3 Knapsack

Al realizar las pruebas se tomaron los mismos datos de la prueba con AG, y se obtuvieron los siguientes resultados consignados en la tabla 6.8 y la figura 6.15.

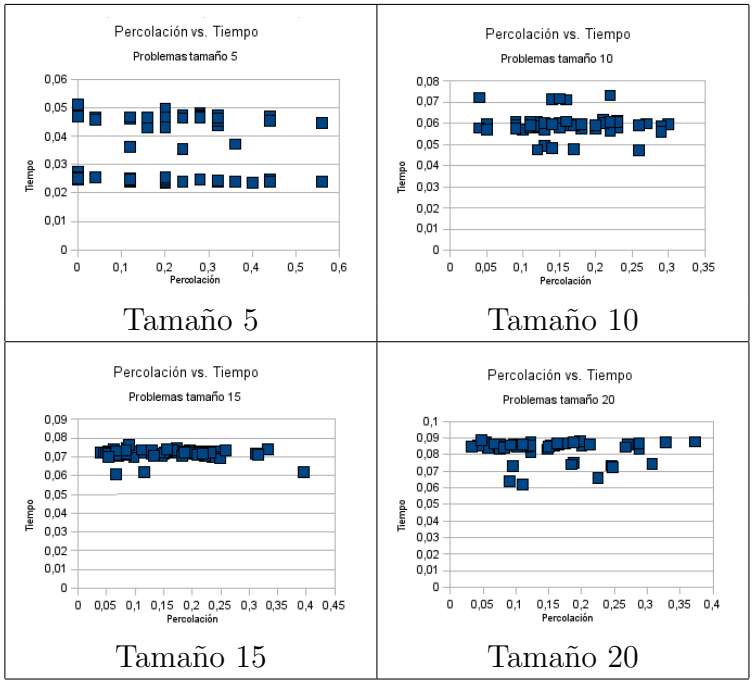
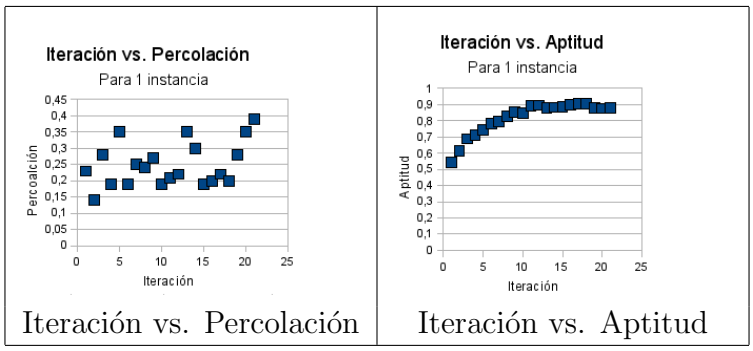


Figura 6.16: Gráficas AGM Knapsack con distinto N

En la gráfica 6.15 se observa un comportamiento similar al de la prueba con AG, en donde se aprecia una gran dispersión en los datos. Por lo tanto, es necesario mostrar las gráficas para problemas de tamaño 5, 10, 15 y 20 (figura 6.16), para mirar el comportamiento en cada una de esas condiciones.

Puede notarse una relación en cuanto a la ubicación de los datos en las gráfica, pudiéndose destacar el hecho de que a mayor número de ítems menor es el valor de percolación y mayor el tiempo de ejecución.



Iteración vs. Percolación | Iteración vs. Aptitud

Figura 6.17: Gráficas AGM Knapsack para 1 instancia

Adicionalmente se realizó el seguimiento en el proceso de ejecución del AGM con una instancia de 10 ítems y una capacidad de 100 y se observa lo siguiente:

Al igual que en K-SAT, la percolación varía en busca de la solución óptima, mientras que la aptitud va subiendo muy lentamente hasta que se estabiliza en valores muy cercanos a 1. Esto implica, que la inclusión del cálculo de percolación en cada iteración y su participación en la función de selección, conlleven a que el AGM obtenga mejores resultados en cuanto a calidad de solución que el AG tradicional.

6.3 AG vs. AGM

En esta sección se compararán los datos obtenidos en cada una de las pruebas ejecutadas, tanto en AG como en AGM, para cada uno de los problemas NP.

6.3.1 TSP

Para la totalidad de datos se obtuvo lo siguiente:

A pesar de que en la figura 6.18 las gráficas son muy parecidas, en el AGM hay un leve aumento en el tiempo de ejecución. Este aumento se debe a que en el AGM se hace el cálculo de percolación en cada iteración y para problemas grandes, este

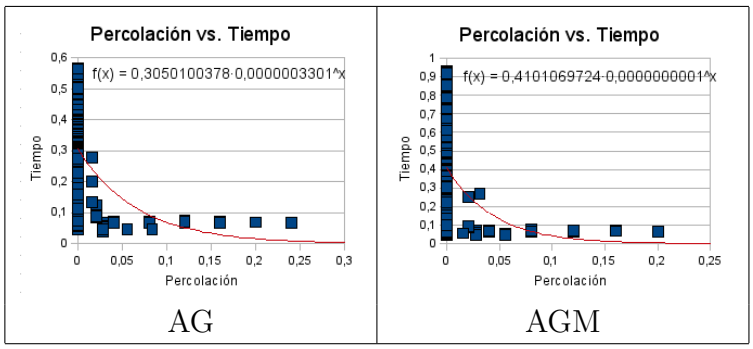


Figura 6.18: TSP AG vs. TSP AGM

cálculo consume un tiempo considerable.

En cuanto a la cantidad de soluciones encontradas se tiene lo siguiente:

- **AG.** Encuentra todas las soluciones para problemas de tamaños 5 y 6, algunas para tamaño 7 y muy pocas para 8. De ahí en adelante el problema se vuelve irresoluble para el AG.
- **AGM.** Encuentra todas las soluciones para problemas de tamaños 5, 6 y 7, algunas para tamaño 8 y muy pocas para 10. De ahí en adelante el problema se vuelve irresoluble para el AGM.

Por lo tanto, el AGM, a pesar de no optimizar el tiempo de ejecución, puede encontrar más soluciones en problemas más complejos que el AG tradicional.

6.3.2 K-SAT

En las gráficas de la figura 6.19 no se puede apreciar ningún cambio evidente, por lo tanto, es necesario hacer la comparación mediante las gráficas por tamaños de problemas.

En la figura 6.20 se observa dispersión en los datos de ambas gráficas; sin embargo, en cuanto a tiempos de ejecución, el AGM obtiene mejores resultados que el AG, observándose tiempos de hasta 0.02 segundos, mientras que en el AG se tienen tiempos

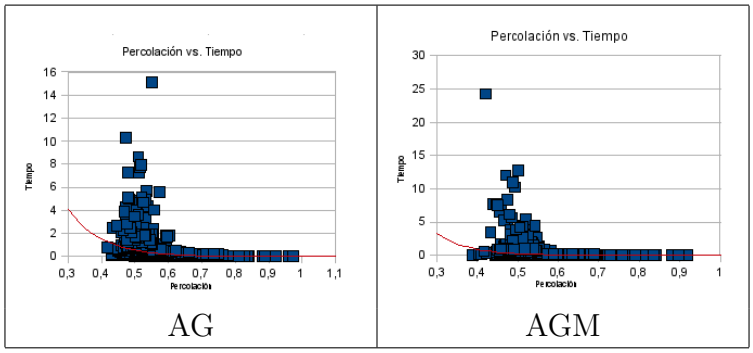


Figura 6.19: K-SAT AG vs. K-SAT AGM

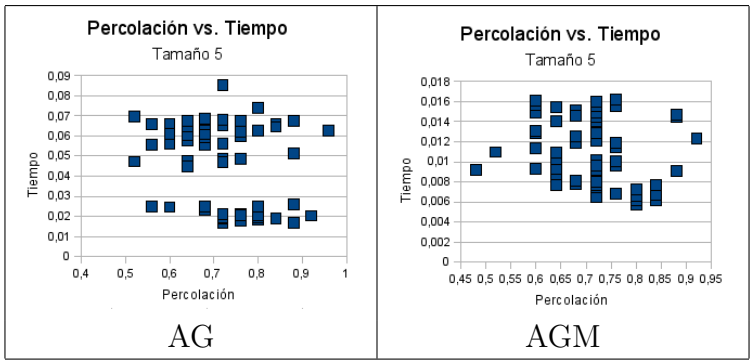


Figura 6.20: K-SAT AG vs. K-SAT AGM Tamaño 5

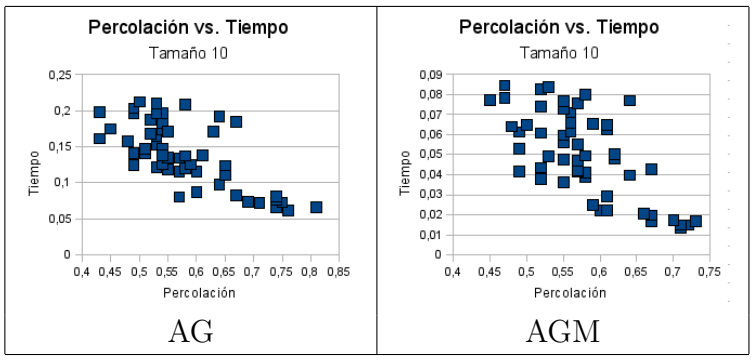


Figura 6.21: K-SAT AG vs. K-SAT AGM Tamaño 10

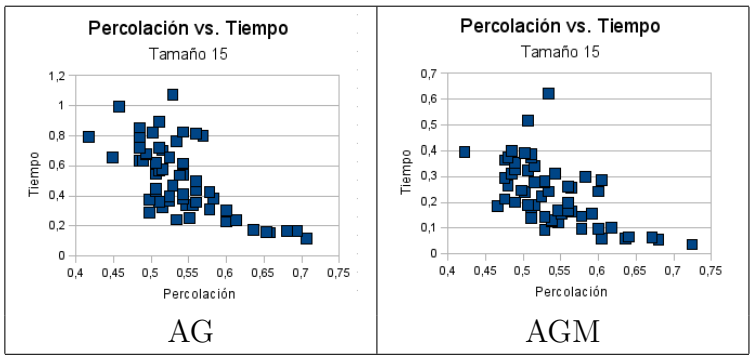


Figura 6.22: K-SAT AG vs. K-SAT AGM Tamaño 15

de hasta 0.09. En cuanto al parámetro de percolación, ambos algoritmos presentan valores parecidos.

En las figuras 6.21 y 6.22 se puede apreciar el mismo comportamiento que en la gráfica de tamaño 5, observándose una mejora considerable en cuanto a tiempos de ejecución por parte del AGM con respecto del AG.

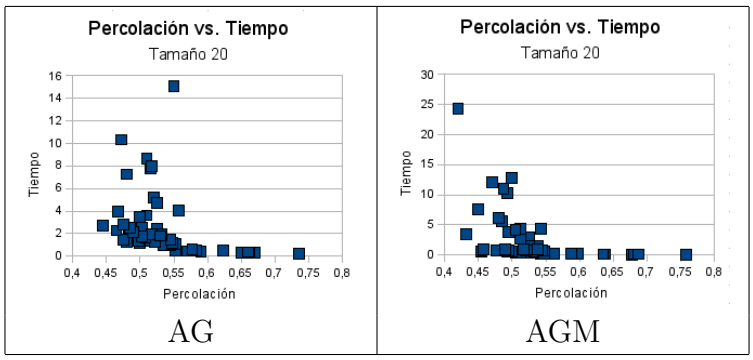


Figura 6.23: K-SAT AG vs. K-SAT AGM Tamaño 20

En ambos casos (figura 6.23) los valores de percolación se concentran entre 0.45 hasta 0.55, lo que implica que los problemas son bastante difíciles de solucionar, por lo tanto es comprensible que en el AGM hayan problemas que demoren 25 segundos.

Finalmente, al analizar los datos obtenidos, se puede concluir que el AGM supera al AG en cuanto a tiempos de ejecución; sin embargo, para algunos problemas de más de 15 variables el AGM puede tardar más que el AG tradicional.

6.3.3 Knapsack

En la figura 6.24 se encuentran los resultados de las pruebas realizadas a los algoritmos y se puede ver notablemente que no existe mayor diferencia en cuanto a tiempos y percolación. Sin embargo, a diferencia de los otros dos problemas, en el problema de la mochila los algoritmos tienen 2 opciones: devolver la solución óptima o la solución aproximada.

Para el AG se obtienen resultados muy buenos para problemas menores a 7 ítems; es decir, los problemas de 5 y 6 ítems se resuelven óptimamente. Para problemas de 7 ítems en adelante, las soluciones que devuelve el algoritmo se aproximan bastante al óptimo, haciéndoles falta 1 ó 2 ítems por incluir en la mochila. En el caso del AGM se obtienen soluciones óptimas en problemas de hasta 8 ítems, a partir de 9 ítems, el

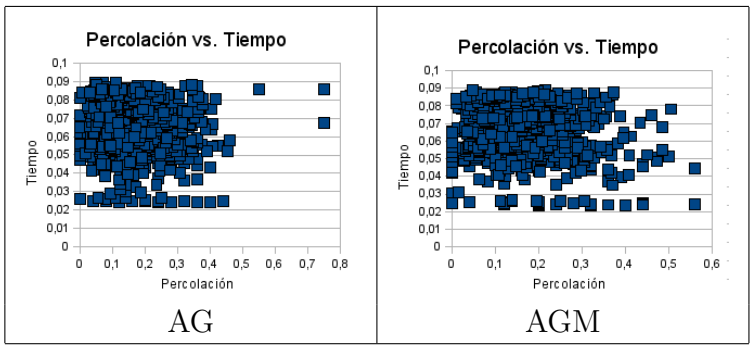


Figura 6.24: Knapsack AG vs. Knapsack AGM

algoritmo devuelve la solución aproximada. Dependiendo del problema, ambos algoritmos devuelven la solución óptima en problemas de orden superior; esto se debe, a que en algunos casos el problema se resuelve introduciendo un solo ítem en la mochila.

Finalmente se observa que hay una leve mejora del AGM con respecto al AG en cuanto a soluciones óptimas, sin afectar el tiempo de ejecución.

6.4 Conclusiones y Recomendaciones

De las pruebas realizadas con ambos algoritmos se llegó a las siguientes conclusiones:

- Existe una relación entre percolación y dificultad. Esta relación toma como base el hecho de que a mayor percolación menor es la dificultad del problema y viceversa. Se analizaron las gráficas por tamaño de problema y se notó la relación Percolación-Dificultad, observándose un comportamiento exponencial en la mayoría de los casos.

- En el problema de TSP es evidente la relación Percolación-Dificultad, debido a que, desde la generación del problema se plantea una forma muy difícil de encontrar la solución a medida que se aumenta la cantidad de ciudades. A causa de la naturaleza combinatoria del problema y el orden que debe tener el recorrido, a partir de 10 ciudades es prácticamente imposible de solucionar -por lo menos con un AG tradicional-.

- Se puede implementar el AGM en la resolución de problemas NP y NP-Completo, ya que, como se observó en las pruebas, el AGM obtuvo buenos resultados en cuanto a tiempo de ejecución y calidad de solución de los problemas que se le plantearon.

- Se pueden investigar otras formas de cambiar el AG en función de la percolación, por ejemplo cambiando probabilidades de mutación o añadiendo otros operadores de reproducción que produzcan cambios más bruscos.

Bibliografía

[Alg03] Marzal, Castro, Albar. *Apuntes de Algorítmica*. `imp.act.uji.es:8080/algoritmica/repo/Tocho8`, 2003.

[AlgGen] *Algoritmos Genéticos*. `http://eddyalfaro.galeon.com/geneticos.html`.

[AGWiki] *Algoritmos Genéticos*. `http://es.wikipedia.org/wiki/Algoritmo_gen%C3%A9tico`.

[Ash05] Ascher, David. *Python Cookbook*. O'Reilly, 2005, 2Ed.

[Bell57] Bellman, Richard. *Dynamic Programming*. Princeton University Press, 1957.

[BrHa57] S. R. Broadbent, J. M. Hammersley. *Percolation Processes I. Crystals and Mazes*. Procedings of the Cambridge Philosophical Society, 1957.

[CompEvo] *Notas de Clase Curso Computación Evolutiva*. `http://eisc.univalle.edu.co/~angarcia/ce/ce.html`.

[Cook71] Cook, Stephen. *The Complexity Of Theorem-Proving Procedures*. ACM, New York, 1971.

[Gol89] Goldberg, David E. *Genetics Algorithms in Search, Optimization and Machien Learning*. Addison Wesley Press, 1989.

[Holl75] Holland, John. *Adaptation in Natural and Artificial Systems*. The Massachusetts Institute Of Technology Press, 1975.

[InAlg01] T. Cormen, C. Leiserson, R. Rivest, C. Stein. *Introduction To Algorithms*. The Massachusetts Institute Of Technology Press, 2001.

[Kau00] Kauffman, Stuart. *Investigations*. Oxford University Press, 2000.

[Knap] *Knapsack Problem*. `http://www.tracer.ull.es/academic/knapsack_html/node1.html`.

[Koz90] Koza, John. *Genetic Programming: A paradigm for genetically breeding population of computers programs to solve problems*. Standford University Computer Science Department Technical Report, 1990.

[Koz92] Koza, John. *Genetic Programming: On the programming of computers by means of natural selection*. The Massachusetts Institute Of Technology Press, 1992.

[Man77] Mandelbrot, Benoit. *Fractals: Form, Chance and Dimension*. WH Freeman and Co., 1977.

[Man82] Mandelbrot, Benoit. *The Fractal Geometry of Nature*. WH Freeman and Co., 1982.

[NPWiki] *Complejidad NP*. `http://es.wikipedia.org/wiki/NP_(Complejidad_computacional)`.

[NPCWiki] *NP-Completo*. `http://es.wikipedia.org/wiki/NP-completo`.

[NPHWiki] *NP-Hard*. `http://es.wikipedia.org/wiki/NP-hard`.

[Pap94] Papadimitriou, Christos. *Computational Complexity*. Addison Wesley Press, 1994.

[Perc01] M. L. Pérez Rea, E. Martínez González. *Modelación de Microestructura de Suelos mediante Teoría de Percolación para el Análisis del comportamiento Esfuerzo-Deformación*. `www.cio.mx/3_enc_mujer/files/extensos/Sesion203/S3-ING06.doc`

[Perc02] Departamento de Farmacia y Tecnología Farmacéutica, Universidad de Sevilla *Aplicación de la Teoría de la Percolación al Estudio de Matrices Hidrófilas de Lobenzarit Disódico y Dextrana*. www.sefig.org/doc/SALAMANCA2005-pdf/BIOFARMACIA/BIOF-020.%20Millan-Jimenez%20y%20col.pdf

[PDWiki] *Problema de Decisión*. http://es.wikipedia.org/wiki/Problema_de_decisi\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{o\global\mathchardef\accent@spacefactor\spacefactor}\accent19o\egroup\spacefactor\accent@spacefactorn.

[Sch90] M. Schroeder. *Fractals, Chaos, Power Laws - Minutes from an Infinite Paradise*. W. H. Freeman and Company, USA, 1990.

[TSPWiki] *Traveling Salesman Problem*. http://es.wikipedia.org/wiki/Problema_del_viajante.