# Universidad Nacional de La Pampa

## Facultad de Ingeniería

**Trabajo final presentado para obtener el grado de**

**Ingeniero en Sistemas**

# Optimización basada en colonias de hormigas: un análisis paramétrico y comparativo

por

**Sanz Troiani, Sergio Fernando**


**La Pampa - General Pico**

**Octubre de 2011**

# Optimización basada en colonias de hormigas: un análisis paramétrico y comparativo

Sanz Troiani, Sergio Fernando

12 de diciembre de 2011

Quiero dedicar ésta tesis, a Carolina, mi gran amor.
A mi hijo Uciel que representa una batería en mi vida.
A mis padres que me ayudaron y acompañaron en todo momento.
Al grupo de investigación LISI por brindarme un lugar y ayudarme a crecer,
especialmente a Gabriela, mi directora de tesis, por depositar su confianza en mí
y guiar mis pasos en el aprendizaje de la investigación y la docencia.

# Optimización basada en colonias de hormigas: un análisis paramétrico y comparativo

Sanz Troiani, Sergio Fernando

**Facultad de Ingeniería, 2011**

**Asesor de Tesis: Mag. Minetti, Gabriela**

Las variantes algorítmicas de la optimización basada en colonias de hormigas son herramientas eficientes, a la hora de resolver problemas de optimización combinatoria NP-completos. Pero su principal desventaja radica en el gran número de parámetros a configurar previo a su ejecución. Dado que, la configuración de los mismos afecta significativamente el desempeño de estos algoritmos se realiza un análisis paramétrico y comparativo. A partir del cual se identifican los parámetros susceptibles a cambios y sus valores más apropiados según la situación a resolver.

# Índice general

# Índice de cuadros

# Índice de figuras

# Introducción

Cada día surgen nuevas variantes de algoritmos metaheurísticos en investigación, donde la optimización basada en Colonias de Hormigas (Ant Colony Optimization - ACO) [8] está teniendo cada vez mayor importancia. ACO es una metaheurística de aproximación inspirada en el comportamiento de las hormigas reales. Este comportamiento permite a las hormigas encontrar el camino más corto entre las fuentes de comida y su nido. Los problemas atacados por ACO en su mayoría son problemas de optimización combinatoria, donde generalmente el tiempo para encontrar la mejor solución es exponencial dado que son NP-completos.

Uno de ellos es el problema del viajante de comercio asimétrico (Asimetric Travelling Salesman Problem - ATSP). Este problema es muy utilizado ya que su formulación permite representar diversos problemas en el campo de las telecomunicaciones [5], planificación y programación de tareas [17], problemas de asignación cuadrática [23], problemas de balance de tareas en la línea de ensamblaje automotriz [2], entre otros.

Una de los principales desventajas de los algoritmos ACO es el gran número de parámetros a configurar previo a la ejecución del mismo, siendo el desempeño de los mismos susceptible a dicha configuración. Motivo por el cual en este trabajo analizamos y comparamos estadísticamente las principales variantes de ACO: Sistema de Colonia de Hormigas (Ant Colony System - ACS) [7], Sistema de Hormiga Max-Min (Max-Min Ant System - MMAS) [25, 24, 26], Sistema de Hormigas con Ordenación (Rank-Based Ant System - ASrank) [3]. Esto lo realizamos considerando diferentes configuraciones paramétricas sobre un conjunto de instancias de variada complejidad, correspondientes a ATSP. El objetivo es encontrar configuraciones que permitan un buen desempeño de cada variante algorítmica para esta clase de problemas combinatorios.

El resto del trabajo se organiza de la siguiente forma: en la sección 2 presentamos una descripción del problema del viajante de comercio, en la sección 3 resumimos las distintas metaheurísticas de optimización, en la sección 4 introducimos los principales conceptos de la optimización basada en colonias de hormigas. Continuamos con un análisis de los resultados obtenidos durante la experimentación en la sección 5 y culminamos con las conclusiones en la sección 6.

# Capítulo 1

# Problema del viajante de comercio

En el siglo XVII era común proveer de productos a las ciudades, pueblos y aldeas. En ese momento, la figura *agente de ventas* se lo llamaba *viajante de comercio* por la naturaleza de su trabajo. Este llevaba un catálogo de productos para mostrar a sus clientes. La idea era realizar un recorrido pasando por las distintas ubicaciones tratando de no volver a pasar por una misma ubicación y luego volver al lugar de origen intentando de lograr un recorrido mínimo [4].

El problema del viajante de comercio fue definido en 1800 de forma matemática por W.R. Hamilton y Thomas Kirkman. El 5 de Febrero de 1930, el matemático austriaco Karl Menger plantea por primera vez el problema en lenguaje matemático. Estableció la relación con los ciclos hamiltonianos de distancia mínima. Luego, Hassler Whitney le dió el nombre al problema: *"travelling salesman problem (TSP)"*. Richard M. Karp demostró en 1972 que los problemas de ciclos de hamiltonianos son NP-Completos, que implico que TSP también lo es. Hoy en dia, el problema del viajante de comercio es clasificado como un problema de optimización combinatoria y es NP-Completo.

## 1.1. Problema del viajante de comercio asimétrico

El problema del viajante de comercio asimétrico es conocido en inglés como *Asymmetric Traveling Salesman Problem (ATSP)*. Dado un conjunto de ciudades (Nodos o Componentes) y distancias para cada par de ciudades, se debe encontrar una ruta de longitud total mínima que visite cada uno de las ciudades exactamente una vez y vuelva al nodo donde comenzó. Se dice asimétrico porque la distancia entre dos ciudades *i,j* puede ser diferente de *i* a *j* que de *j* a *i* como muestra la figura 1.1, por ejemplo, se observa que para viajar de Santa Rosa a General Pico se puede ir por el este (ruta 5) con distancia $d_{ij}$ y luego volver por el oeste (ruta 35 y ruta 102) con distancia $d_{ji}$, donde $d_{ij}$ distinto a $d_{ji}$.

Existen muchas formulaciones matemática de este problema, la siguiente usa relativamente pocas variables y define la variable binaria $x_{rs}$ $\forall r, s$ [6]:

$$x_{rs} = \begin{cases} 1, & si\ un\ tour\ incluye\ el\ viaje\ de\ la\ ciudad\ r\ a\ la\ s, \\ 0, & otro\ caso \end{cases}$$

El objetivo es minimizar:

$$\sum_{r=1}^{n}\sum_{s=1}^{n} d_{rs}x_{rs} \qquad donde\ d_{rr} = \infty\ for\ r = 1, \ldots, n \tag{1.1}$$

sujeto a:

$$\sum_{s=1}^{n} x_{rs} = 1\ for\ r = 1, \ldots, n\ (salida) \tag{1.2}$$

$$\sum_{r=1}^{n} x_{rs} = 1\ for\ s = 1, \ldots, n\ (llegada) \tag{1.3}$$

$$x_{rs} \in \mathbb{Z}^{+} \forall r, s \tag{1.4}$$

Las restricciones 1.2, 1.3 y 1.4 aseguran que cada $x_{rs}$ sea cero o uno. La restricción 1.2 requiere que un *tour* o recorrido incluya una salida de cada ciudad, mientras que la restricción 1.3 garantiza un arribo en cada ciudad.

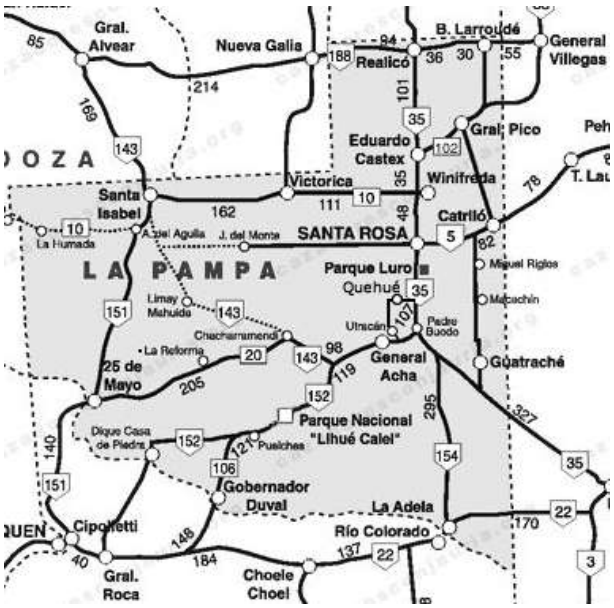


Figura 1.1: Ejemplo de una instancia del problema del viajante de comercio - ATSP.

# Capítulo 2

# Metaheurísticas de optimización

En optimización generalmente se busca la mejor solución o una que cumpla con ciertos criterios de aceptación para un problema dado. Los problemas de optimización se pueden encontrar en forma redundante en la vida cotidiana y laboral, por ejemplo cuando se pretende encontrar el camino más corto para llegar al trabajo, o también cuando se desea organizar la agenda personal.

Los problemas de optimización se pueden modelar a través de un conjunto de variables de decisión, en las cuales se debe configurar sus dominios y restricciones. Estas restricciones se dividen en tres categorías: i) Las que poseen variables discretas ii) Las que poseen variables continuas iii) Las que poseen ambas, discretas y continuas. Este trabajo se centra en las de clase i) llamadas también *Problemas de Optimización Combinatoria* (POC).

Un problema de optimización combinatoria $P = (S, f)$, de acuerdo con Papadimitriou y K. Steiglitz [20], es un problema de optimización en el cual se dan un conjunto de objetos $S$ y una función objetiva $f : S \mapsto \mathbb{R}^+$ que asigna un costo de valor positivo a cada objeto $s \in S$. El objetivo es encontrar un objeto de costo mínimo. Los objetos generalmente son números enteros, subconjuntos de un conjunto de elementos, permutaciones de un conjunto de elementos, o estructuras de grafos.

Muchos algoritmos desarrollados para resolver este tipo de problemas, han sido clasificados en [1]: algoritmos *completos* o *aproximados*. Los algoritmos *completos* garantizan encontrar, para una instancia de POC de tamaño finito, una solución óptima en un tiempo limitado. Hasta el momento, para problemas NP-completo, no existe algoritmo que en tiempo polinomial encuentre una solución óptima. Por lo tanto, los algoritmos completos necesitan de un tiempo exponencial en el peor de los casos. Por esta razón, el uso de métodos *aproximados* ha tenido gran auge en los últimos 30 años. En dichos métodos se sacrifica la garantía de encontrar una solución óptima para lograr buenas soluciones en un tiempo significativamente reducido. En los diferentes métodos básicos de aproximación generalmente se hace una distinción entre *Heurísticas Constructivas* y *Métodos de Búsqueda Local* [1]. Ambos dieron origen a lo que hoy se denomina Metaheurísticas.

## 2.1. Heurísticas constructivas

Las Heurísticas constructivas son generalmente los métodos de aproximación más rápidos. Estos generan soluciones a través del agregado oportuno de componentes a una solución parcial inicialmente vacía. Este agregado oportuno se realiza hasta que la solución esté completa o hasta satisfacer otros criterios de finalización. Se asume que la construcción de una solución termina en el caso que la solución (parcial) actual no puede ser extendida. Esto ocurre cuando la solución obtenida sea factible, es decir, que satisfaga las restricciones del problema. En el contexto de heurísticas constructivas las soluciones (completas) y soluciones parciales son secuencias $< c_i, ..., c_k >$ compuestas por componentes $c_j$ de un conjunto finito de componentes C (donde $|C| = n$). Este tipo de soluciones se denota como $s$ para una solución (completa) y $s^p$ para una solución parcial.

Las heurísticas constructivas tendrán que especificar primero el conjunto de posibles extensiones para cada posible solución (parcial) $s^p$. Este conjunto, denotado por $N(s^p)$, es un subconjunto de $C \setminus \{c \mid c \in s^p\}$. En cada paso de la construcción se escoge una posible extensión hasta que $N(s^p) = \emptyset$, esto significa que $s^p$ es la solución o que $s^p$ es una solución parcial que no puede ser extendida a una solución factible. El marco algorítmico de las heurísticas constructivas se muestra en el Algoritmo 1.

**Algoritmo 1** Heurística Constructiva

$s^p =<>$
Determinar $N(s^p)$
**mientras** $N(s^p) \neq \emptyset$ **hacer**
  $c \leftarrow EscojerDe(N(s^p))$
  $s^p \leftarrow Expandir\ s^p\ agregando\ el\ componente\ c$
  Determinar $N(s^p)$
**fin mientras**
**Salida:** solución construida

Un ejemplo notable de heurísticas constructivas es la heurística *greedy*, que implementa el procedimiento $EscojerDe(N(s^p))$ aplicando una función de peso. Una función de peso es una función que, algunas veces dependiendo de la solución actual (parcial), asigna a cada paso de la construcción un valor heurístico $\eta(c)$ para cada componente $c \in N(s^p)$. Las heurísticas *greedy* eligen a cada paso uno de los componentes con el valor más alto.

Por ejemplo, una heurística *greedy* para el problema TSP es la heurística del vecino más cercano. El conjunto de componentes es el conjunto de nodos (ciudades) en $G = (V, E)$ (donde $G$ es el grafo formado por el conjunto de ciudades $V$, junto con las distancias entre ciudades denotado por el conjunto $E$) . El algoritmo comienza seleccionando una ciudad *i* al azar. Entonces, la solución parcial actual $s^p$ es extendida en cada *n - 1* pasos de la construcción agregando la ciudad $j \in N(s^p) = V \setminus s^p$. Hay que notar que los valores heurísticos para esta heurística se eligen como la inversa de la distancia entre ciudades, no dependen de la solución parcial actual. Por lo tanto, la

función de peso se la denomina como *estática*. En los casos donde el valor heurístico depende de la solución parcial actual, la función se denomina *dinámica*.

## 2.2. Métodos de búsqueda locales

Como se ha mencionado, las heurísticas constructivas generalmente son muy rápidas, sin embargo, ofrecen soluciones de menor calidad comparadas con los algoritmos de búsqueda local. Estos comienzan con una solución inicial y de forma iterativa tratan de reemplazar la solución actual por una solución mejor dentro del vecindario de la solución actual. Donde el vecindario se define de la siguiente manera:

**Definición 1:** Una estructura de vecindario es una función $N : S \rightarrow 2^s$ que asigna a cada $s \in S$ un conjunto de vecinos $N(s) \subseteq S$. $N(s)$ es llamado vecindario de $s$. La aplicación que produce un vecino $s^{'} \in N(s)$ de una solución $s$ se denomina comúnmente *movimiento*.

Una estructura de vecindario junto con la instancia del problema definen la topología del espacio de búsqueda. En un espacio de búsqueda se puede visualizar como un grafo con etiquetas donde cada nodo es una solución (la etiqueta indica el valor de la función objetivo) y los arcos representan la relación del vecindario entre las soluciones. Una solución $s^* \in S$ es llamada *solución mínima global* (o mínimo global) si para todo $s \in S$ se cumple que $f(s^*) \leq f(s)$. El conjunto de todas las soluciones globales mínimas se define como $S^*$. La introducción a una estructura de vecindario nos permite definir el concepto de *soluciones mínimas locales*.

**Definición 2:** Una solución mínima local (o mínimo local) respecto a una estructura de vecinos $N$ es una solución ŝ que $\forall s \in N(\hat{s}) : f(\hat{s}) \leq f(s)$.

El método local más básico de búsqueda es usualmente llamado *búsqueda local de mejora iterativa*, porque cada movimiento se realiza si la solución resultante es mejor que la actual. El algoritmo termina tan pronto como encuentre un mínimo local. El esqueleto de alto nivel del algoritmo se presenta en el algoritmo 2.

**Algoritmo 2** Búsqueda local de mejora iterativa

$s \leftarrow GenerarSolucionInicial()$
**mientras** $\exists s^{'} \in N(s)\ tal\ que\ f(s^{'}) < f(s)$ **hacer**
  $s \leftarrow EscojerMejorVecino(N(s))$
**fin mientras**
**Salida:** s

Hay dos grandes formas de implementar la función $EscojerMejorVecino(N(s))$. La primera forma es una función llamada *primer-mejora* que consiste en escanear los vecinos $N(s)$ y devolver

la primera solución que es mejor que $s$. En contraste, la segunda forma es una función llamada *mejor-mejora* que explora exhaustivamente el vecindario y devuelve una de las soluciones con el mejor valor de función objetivo. Los métodos que utilizan estas funciones se llaman *búsqueda local de primer-mejora* y *búsqueda local de mejor-mejora*. Ambos métodos terminan en un mínimo local. Por lo tanto, el rendimiento depende fuertemente de la definición de la estructura de vecindario $N$.

## 2.3. Metaheurísticas

En 1970, un nuevo tipo de algoritmos de aproximación emergió llamado "Metaheurísticas", básicamente trata de combinar métodos heurísticos básicos en un marco de más alto nivel, de manera que explore el espacio de una forma eficiente y efectiva. La clase de algoritmos metaheurísticos incluyen *optimización basada en colonias de hormigas (ant colony optimization ACO), computación evolutiva (evolutionary computation EC), búsqueda local iterativa (iterated local search ILS), enfriamiento simulado (simulated annealing SA), y búsqueda tabú (tabu search TS).*

Las diferentes descripciones de metaheurística que se encuentran en la literatura permiten extraer algunas propiedades fundamentales con las cuales las metaheurísticas son caracterizadas [1]:

- Las metaheurísticas son estrategias que "guían" al proceso de búsqueda.
- El objetivo consiste en explorar eficientemente el espacio de búsqueda para encontrar soluciones óptimas.
- Las técnicas que constituyen los algoritmos de metaheurísticas van desde una simple búsqueda local a complejos procesos de aprendizaje.
- Los algoritmos de metaheurísticas son aproximados y usualmente no deterministas.
- Pueden incorporar mecanismos para evitar quedar atrapados en áreas confinadas del espacio de búsqueda.
- El concepto básico de metaheurística puede ser descripto sobre un nivel abstracto. Es decir, no está atado a un problema específico.
- Las metaheurísticas no son específicas al problema.
- Las metaheurísticas pueden hacer uso de un conocimiento de dominio especifico en forma de heurística que son controladas por la estrategia de un nivel superior.
- Hoy en día, metaheurísticas más avanzadas utilizan experiencia en la búsqueda para guiar la misma.

El balance entre diversificación e intensificación (exploración/explotación) es de gran importancia. El término de diversificación refiere generalmente hace referencia a la exploración del espacio de búsqueda, mientras que el término intensificación se vincula a la explotación de la experiencia acumulada durante la búsqueda. Es muy importante el balance entre diversificación e intensificación, por un lado, porque identifica rápidamente regiones en el espacio de búsqueda con soluciones de calidad y por el otro lado no emplea demasiado tiempo en regiones del espacio de búsqueda que ya han sido exploradas.Existen muchas formas de clasificar y describir los algoritmos de metaheurísticas. Depende de la característica seleccionada para diferenciarlas entre ellas, se pueden encontrar gran cantidad de clasificaciones posibles, cada una es el resultado de distintos puntos de vista. Se clasifican en metaheurísticas de *inspiración natural vs inspiración no natural*, en métodos *basados en memoria vs sin memoria*, en métodos que utilizan una función objetiva *dinámica vs estática*. Esta sección describe las más importantes metaheurísticas de acuerdo a la clasificación de *un solo punto vs búsqueda basada en población*, que dividen las metaheurísticas en *métodos basados en trayectoria y métodos basados en población*.

## 2.4. Métodos basados en trayectoria

El término métodos basados en trayectoria se utiliza porque el proceso de búsqueda, aplicado por estos métodos, se caracteriza por una trayectoria en el espacio de búsqueda. La mayoría son extensiónes de simples procedimientos de mejora iterativa, con un rendimiento usualmente insatisfactorio. Poseen técnicas para escapar de mínimos locales, lo cual implica que necesitan un criterio de terminación diferente a la simple meta de alcanzar un mínimo local. Los criterios de terminación más usados son tiempo máximo de CPU, un número máximo de iteraciones, una solución $s$ de calidad suficiente, o alcanzar un número máximo de iteraciones sin obtener mejores soluciones.

### 2.4.1. Enfriamiento simulado (Simulated Annealing SA)

Enfriamiento Simulado [22] fue uno de los primeros y más antiguos algoritmos que posee una estrategia para escapar de un mínimo local. La idea de SA fue provista por el enfriamiento del proceso del metal y el vidrio, que asume una configuración baja de energía cuando se enfría con un esquema apropiado de enfriamiento. Para poder escapar de los mínimos locales, la idea fundamental es dar permiso al movimiento hacia soluciones con valores de función objetiva que sean inferiores al valor de la función objetiva de la solución actual. En cada iteración una solución $s^{'} \in N(s)$ se elige al azar. Si $s^{'}$ es mejor que $s$, entonces $s^{'}$ se acepta como la solución actual. De otra manera, $s^{'}$ se acepta con una probabilidad en función de un parámetro de temperatura $T_k$ y $f(s^{'}) - f(s)$. Usualmente esta probabilidad se calcula siguiendo la distribución de Boltzmann ecuación 2.1.

$$p(s^{'}|T_k, s) = e^{-\frac{f(s^{'})-f(s)}{T_k}} \tag{2.1}$$

### 2.4.2. Búsqueda tabú (Tabu Search TS)

La idea básica de la búsqueda tabú [12] es el uso explícito de la historia de la búsqueda, que permite escapar de los mínimos locales e implementar una estrategia de exploración.

Un algoritmo simple de TS está basado en la búsqueda local del mejor candidato y usa memoria a corto plazo para escapar de los mínimos locales y evitar ciclos. La memoria a corto plazo se implementa como una lista tabú que mantiene una pista de las soluciones recientemente visitadas y las excluye de los vecinos de la solución actual. En cada iteración, la mejor solución de las permitidas se elige como la nueva solución actual. Esta solución se agrega a la lista tabú.

### 2.4.3. Métodos de búsqueda local explorativa

**Procedimiento de búsqueda voraz aleatoria y adaptativa (Greedy Randomized Adaptative Search Procedure - GRASP**

GRASP [11, 21] se define como una simple metaheurística que combina heurísticas constructivas y búsquedas locales. Consiste en un procedimiento iterativo, compuesto por dos fases: construcción de soluciones y mejoramiento de soluciones. Donde, la mejor solución es devuelta al terminar el proceso de búsqueda.

El mecanismo de construcción de la solución se constituye como una heurística constructiva aleatoria. Ésta va agregando componentes a la solución parcial actual $s^p$. El componente que es agregado en cada paso se elige al azar de una lista llamada *lista candidata restringida*. Esta lista es un subconjunto de $N(s^p)$, que es el conjunto de componentes permitidos.

La segunda fase del algoritmo es un método de búsqueda local que puede ser un algoritmo de búsqueda local básico como mejoramiento iterativo o más avanzado como SA o TS.

**Búsqueda con vecindario variable (Variable Neighborhood Search VNS)**

VNS [15] es una metaheurística que aplica estrategias de intercambio entre diferentes estructuras de vecindario desde un conjunto finito predefinido. El algoritmo es muy genérico y con mucho grado de libertad para designar variables e instancias particulares.

Al comienzo del algoritmo se definen los conjuntos de estructuras de vecindario. Estas estructuras de vecindario pueden ser elegidas arbitrariamente. Luego, se genera una solución inicial, el índice de vecindario es inicializado, y el algoritmo continúa hasta alcanzar alguna condición de terminación. Cada iteración consiste de tres fases: *agitación, búsqueda local y movimiento*. En la primera fase se selecciona una solución $s^{'}$ de forma aleatoria dentro del vecindario $k$ de la solución actual $s$. La solución $s^{'}$ se utiliza como punto de partida para el procedimiento de búsqueda local,

que tal vez utilice cualquier vecindario y no está restringido al conjunto de estructuras de vecindario $N_k$, $k = 1, ..., k_{max}$. Al finalizar la búsqueda local, la nueva solución $s''$ se compara con $s$, si la solución es mejor, el algoritmo reemplaza a $s$ y procede con $k = 1$. Sino, $k$ se incrementa y una nueva fase de agitación comienza utilizando un vecindario diferente.

### Búsqueda local guiada (Guided Local Search - GLS)

GLS [28] aplica una estrategia para escapar de los mínimos locales muy diferente a las de TS y VNS. Ésta estrategia consiste en cambiar dinámicamente la función objetivo, que resulta en el cambio del espacio de búsqueda. El objetivo es hacer que el mínimo local actual gradualmente sea "menos deseable" con el tiempo.

El cambio dinámico de la función objetivo ésta basado en un conjunto de $m$ soluciones candidatas: $sf_i$, donde $i = 1, ..., m$. La solución candidata puede ser cualquier tipo de propiedad o característica que puede ser usada para discriminar entre las distintas soluciones.

### Búsqueda local iterada (Interated Local Search - ILS)

Esta metaheurística está basada en un simple pero poderoso concepto [27, 16, 18], en cada iteración la solución actual (que es un mínimo local) es perturbada y luego se le aplica a la solución perturbada un método de búsqueda local. Luego, el mínimo local obtenido al aplicar la búsqueda puede ser aceptada como la nueva solución o no. Intuitivamente, ILS realiza una trayectoria a través de mínimos locales $\hat{s_1}, \hat{s_2}, ..., \hat{s_t}$ sin introducir explícitamente una estructura de vecindario sobre $\hat{S}$, aplicando el esquema que muestra el algoritmo 3.

**Algoritmo 3** Iterated Local Search (ILS)

$s \leftarrow GenerarSolucionInicial()$
$\hat{s} \leftarrow BusquedaLocal(s)$
**mientras** condición de terminación no alcanzado **hacer**
  $s' \leftarrow Perturbacion(\hat{s}, historia)$
  $\hat{s}' \leftarrow BusquedaLocal(s')$
  $\hat{s} \leftarrow AplicarCriterioDeAceptacion(\hat{s}', \hat{s}, historia)$
**fin mientras**
**Salida:** Mejor solución encontrada

La importancia de la perturbación es clara: una perturbación pequeña tal vez no permita al sistema escapar del mínimo local encontrado. Por otro lado, una perturbación grande convertiría al algoritmo en un algoritmo similar al de búsqueda local con reinicio aleatorio.

## 2.5. Métodos basados en población

Los métodos basados en población trabajan en cada iteración con un conjunto de soluciones (población) en vez de hacerlo con una única solución. De esta manera, proveen de forma natural e intrínseca una manera de explorar el espacio de búsqueda. Aún, el rendimiento final depende fuertemente de la manera en que la población es manipulada. Los métodos basados en población más estudiados son Algoritmos Evolutivos (Evolutionary Algorithms EA) y Optimización en Colonias de Hormigas (Ant Colony Optimization ACO). En algoritmos EC, una población de individuos es modificada por operadores de recombinación y mutación, y en ACO una colonia de hormigas es usada para construir soluciones guiadas por el rastro de feromona e información heurística.

### 2.5.1. Optimización mediante colonias de hormigas (Ant Colony Optimization - ACO)

La optimización mediante colonia de hormigas es una metaheurística de aproximación que fue inspirada en la observación del comportamiento real de las hormigas [10, 8, 9]. Este comportamiento permite a las hormigas encontrar la trayectoria más corta entre las fuentes de comida y su nido. Inicialmente las hormigas exploran el área que rodea su nido de forma aleatoria, tan pronto una hormiga encuentra una fuente de comida traslada una parte a su nido. Las hormigas depositan un rastro químico de feromonas sobre el suelo (la cantidad de feromona depositada depende de la cantidad y calidad de la comida, esto guiará a otras hormigas hacia la fuente de comida. La comunicación indirecta entre las hormigas a través del rastro de feromonas, les permite encontrar la trayectoria más corta entre su nido y las fuentes de comida). Esta funcionalidad de las colonias de hormigas reales es explotada a través de colonias de hormigas artificiales de manera de poder resolver problemas de optimización combinatorios.

En el próximo capítulo se extiende en profundidad la metaheurística ACO, con sus principales variantes, que fue la seleccionada por el presente trabajo.

### 2.5.2. Algoritmos evolutivos (Evolutionary Algorithms - EA)

Los algoritmos evolutivos (EA) están inspirados por la capacidad natural de evolución de los seres vivos para adaptarse a su entorno. En cada iteración un número de operadores son aplicados a los individuos de la población actual para generar los individuos de la población de la próxima generación (iteración). Usualmente, estos algoritmos utilizan operadores llamados **recombinación** o **cruce** para recombinar dos o más individuos para crear un nuevo individuo. También utilizan los operadores de **mutación** que causan una auto adaptación del individuo. La fuerza motora en algoritmos evolutivos es la **selección** de individuos basada en su **aptitud** (*fitness*) (que puede estar basada en su función objetivo, el resultado de simulaciones experimentales o cualquier otra medida de calidad). Individuos con mayor aptitud tienen mayor probabilidad de ser elegidos como

miembros de la población de la próxima iteración (o como padres para la generación de nuevos individuos). Esto corresponde al principio de **supervivencia del más apto** en la evolución natural. La capacidad natural de adaptarse a sí mismo en un entorno cambiante fue la que inspiró a los algoritmos EC.

**Algoritmo 4** Evolutionary Algorithms (EA)

$P \leftarrow GenerarPoblacionInicial()$
$Evaluar(P)$
**mientras** condición de terminación no alcanzado **hacer**
  $P' \leftarrow Recombinar(P)$
  $P'' \leftarrow Mutacion(P')$
  $Evaluar(P'')$
  $P \leftarrow Seleccion(P, P'')$
**fin mientras**
**Salida:** Mejor solución encontrada

Existe una gran variedad de algoritmos EA ligeramente diferentes propuestos a través de los años. Básicamente caen dentro de tres diferentes categorías que han sido desarrolladas independientemente una de otras. Estas son: **Programación Evolutiva** (Evolutionary Programming EP), **Estrategias Evolutivas** (Evolution Strategies ES) y **Algoritmos Genéticos** (Genetic Algorithms GAs).

El algoritmo 4 contiene la estructura básica de los algoritmos EC. En este algoritmo $P$ representa la población de individuos. En cada iteración un conjunto de individuos hijos $P^{'}$ es generado por la aplicación de la función **Recombinar(**$P$**)**, estos miembros pueden ser mutados por la función **Mutación(**$P^{'}$**)**, produciendo un conjunto de individuos hijos $P''$. Los individuos para la próxima población son seleccionados por la función **Selección(**$P$**,**$P^{''}$**)** a través de la unión de la población anterior $P$ y el conjunto de individuos hijos mutados $P^{''}$. Los individuos de los algoritmos EC no son necesariamente soluciones al problema considerado, pueden ser soluciones parciales, o un conjunto de soluciones, o cualquier objeto que puede ser transformado en una o muchas soluciones en forma estructurada.

### Búsqueda dispersa y reencadenamiento de trayectorias (Scatter Search - SS and Path Relinking - PR)

La búsqueda dispersa [13, 14] y su forma general llamada reencadenamiento de trayectoria difieren de los algoritmos EC por la recombinación de sus soluciones sobre su trayectoria general, construidas sobre espacios euclidianos o vecindarios. Las estrategias de búsqueda de estos algoritmos operan sobre un conjunto de soluciones de referencia ($S_{ref}$) que corresponden a soluciones posibles bajo el problema en consideración. Este conjunto de soluciones de referencia esta constituido por una población de individuos. El algoritmo comienza generando un conjunto $S_{semilla}$

también llamado soluciones semillas, encontradas a través de un método heurístico. Luego, se aplica un método llamado *GeneradorDeDiversificacion($S_{semilla}$)* iterativamente, con el objetivo de crear soluciones lo más diversas posibles con respecto a las existentes. El cual consiste en elegir una de las soluciones semillas y generar nuevas soluciones. Partiendo del conjunto de soluciones $S_{div}$ se elige el primer conjunto de soluciones de referencia $S_{ref}$ de manera que contengan una alta calidad, como así también diversificación. Luego comienza la repetición principal del algoritmo, con un ciclo interno que se repite un número de veces. En el ciclo interno, primero un subconjunto de las soluciones de referencia $S_{sub}$ es seleccionado en la función *GenerarSubConjunto($S_{ref}$)* para construir una o más soluciones de prueba $S_{prueba}$, alguna de éstas soluciones pueden no ser admisibles y por lo tanto modificadas por un procedimiento reparador que las transforma en soluciones admisibles. Un mecanismo de mejoramiento *Mejorar($S_{prueba}$)* es aplicado para tratar de obtener del conjunto de soluciones de prueba (usualmente es una búsqueda local) soluciones dispersas denotadas por $S_{disp}$. Finalmente el conjunto de soluciones de referencia $S_{ref}$ es actualizado con las soluciones de $S_{disp}$ dentro de la función *ActualizarConjuntoDeReferencia($S_{ref}$, $S_{disp}$)*, nuevamente teniendo en cuenta el criterio de calidad y diversidad. Luego, volviendo al ciclo principal, que se repite hasta alcanzar algún criterio de terminación, un conjunto de soluciones $S_{elite}$ es elegido del conjunto de soluciones de referencia, el generador de diversificación es aplicado, y un nuevo conjunto de soluciones de referencia es elegido del resultado del conjunto de soluciones.

**Algoritmo 5** Scatter Search and Path Relinking

$S_{semilla} \leftarrow GeneradorSemilla()$
$S_{div} \leftarrow GeneradorDeDiversificacion(S_{semilla})$
$S_{ref} \leftarrow ElegirConjuntoReferencia(S_{div})$
**mientras** condición de terminación no alcanzado **hacer**
  **mientras** condición de terminación para bucle interno no alcanzado **hacer**
    $S_{sub} \leftarrow GenerarSubConjunto(S_{ref})$
    $S_{prueba} \leftarrow CombinarSoluciones(S_{sub})$
    $S_{disp} \leftarrow Mejorar(S_{prueba})$
    $S_{ref} \leftarrow ActualizarConjuntoDeReferencia(S_{ref}, S_{disp})$
  **fin mientras**
  $S_{elite} \leftarrow ElegirMejores(S_{ref})$
  $S_{div} \leftarrow GeneradorDeDiversificacion(S_{elite})$
  $S_{ref} \leftarrow ElegirConjuntoReferencia(S_{div})$
**fin mientras**
**Salida:** Mejor solución encontrada

*CombinarSoluciones($S_{sub}$)*: En la búsqueda dispersa, que se introduce como soluciones codificadas como puntos del espacio euclidiano, nuevas soluciones son creadas a través de combinaciones lineales de soluciones de referencia, utilizando pesos positivos y negativos. Esto quiere decir que las soluciones de prueba pueden ser ambas, es decir dentro y fuera de la región convexa expandida. En Reencadenamiento de trayectorias, el concepto de combinar soluciones a través de combinaciones lineales de puntos de referencia es generalizado por el espacio de vecindad. Com-

binaciones lineales de puntos del espacio euclidiano pueden ser interpretadas como trayectorias entre soluciones del espacio de vecinos y más allá. Para generar trayectorias deseadas es necesario seleccionar movimientos que satisfagan la siguiente condición: a partir de una solución inicial seleccionada de $S_{sub}$, los movimientos progresivamente deben introducir atributos introducidos por una solución guía que también es seleccionada de $S_{sub}$.

**Algoritmos de Distribución Estimativa (Estimation of Distribution Algorithms - EDAs)**

En la última década muchas investigaciones han intentado contrarrestar los inconvenientes o desventajas de los operadores de recombinación de los algoritmos EC. Con este objetivo han sido desarrollados un número de algoritmos que son algunas veces llamados algoritmos de distribución estimativa (EDAs) [19]. Estos algoritmos, que se fundamentan en la teoría de la probabilidad, son basados en poblaciones que evolucionan como un progreso de búsqueda como los algoritmos EC. Funcionan de la siguiente manera: primero, una población inicial $P$ de soluciones es generada aleatoriamente o heurísticamente, luego el ciclo siguiente es repetido hasta cumplir un criterio de terminación. En la función *ElegirDe(P)* se selecciona una fracción de la mejor solución proveniente de la población actual (denotada por $P_{sel}$).Luego, en la función *DistribucionEstimativaProbabilistica($P_{sel}$)* se deriva una distribución probabilística sobre el espacio de búsqueda desde la solución $P_{sel}$. La muestra de esta distribución probabilística en la función *MuestraDistribucionProbabilistica(p(x))* produce la nueva población para la próxima iteración. En el algoritmo 6 se puede observar el marco general de funcionamiento.

**Algoritmo 6** Estimation of Distribution Algorithms

$P \leftarrow GenerarPoblacionInicial()$
**mientras** condición de terminación no alcanzado **hacer**
  $P_{sel} \leftarrow ElegirDe(P) \quad \{P_{sel} \subseteq P\}$
  $p(x) = p(x|P_{sel}) \leftarrow DistribucionEstimativaProbabilistica(S_{sel})$
  $P \leftarrow Mejorar(S_{prueba})$
  $S_{ref} \leftarrow MuestraDistribucionProbabilistica(p(x))$
**fin mientras**
**Salida:** Mejor solución encontrada

# Capítulo 3

# Optimización mediante colonias de hormigas

La optimización basada en colonias de hormigas (ACO) es una metaheurística de aproximación que fue inspirada por el comportamiento de las hormigas reales. Este comportamiento permite a las hormigas encontrar el camino más corto entre las fuentes de comida y su nido. Inicialmente, las hormigas exploran el área de los alrededores de su nido al azar. Tan pronto, una hormiga encuentre una fuente de comida, ésta lleva algo de comida al nido. Mientras camina, la hormiga deposita un rastro de feromona química en el suelo. La cantidad de feromona depositada, que puede depender de la cantidad y calidad de la comida, guiará a otras hormigas a la fuente de comida. La comunicación indirecta entre las hormigas por el rastro de feromona les permite encontrar el camino más corto entre el nido y las fuentes de comida. Esta funcionalidad de las colonias de hormigas reales es explotada por las colonias de hormigas artificiales para resolver problemas POC.

Están basados en una colonia de hormigas artificiales, que simplemente son agentes computacionales que trabajan cooperativamente y se comunican a través de rastros de feromona artificiales.

## 3.1. Colonia de hormigas naturales

Las hormigas son insectos sociables que viven en colonias, y como poseen acciones colaborativas, son capaces de mostrar comportamientos complejos y realizar tareas difíciles desde la perspectiva de una hormiga. Uno de los aspectos interesantes es su habilidad de encontrar el camino más corto desde la colonia a las fuentes de comida. Otro aspecto importante es que las hormigas son ciegas y al volver de una fuente de comida van dejando un rastro de feromona (Sustancia Excretada) sobre el ambiente.

Inicialmente no existe rastro de feromona en el ambiente, cuando una hormiga llega a una intersección, ella elige un camino al azar. Una vez encontrada la fuente de alimentos esta vuelve con

la comida y va liberando feromona hasta llegar al hormiguero.

Este último proceso es complementado por el entorno natural, dado que la feromona se evapora después de un tiempo. De esta manera, los caminos menos prometedores, progresivamente perderán feromona porque serán visitadas por menos hormigas cada vez.

Cuando ya existe un rastro de feromona las hormigas siguen este rastro. Cuando se encuentran en una intersección éstas eligen el rastro de mayor concentración de feromona. Por lo que de a poco el otro camino irá perdiendo su concentración de feromona. De esta manera el camino más corto será paulatinamente el que poseerá mayor concentración de feromona. La figura 3.1 muestra la forma en que este mecanismo permite a la hormiga alcanzar el camino más corto.

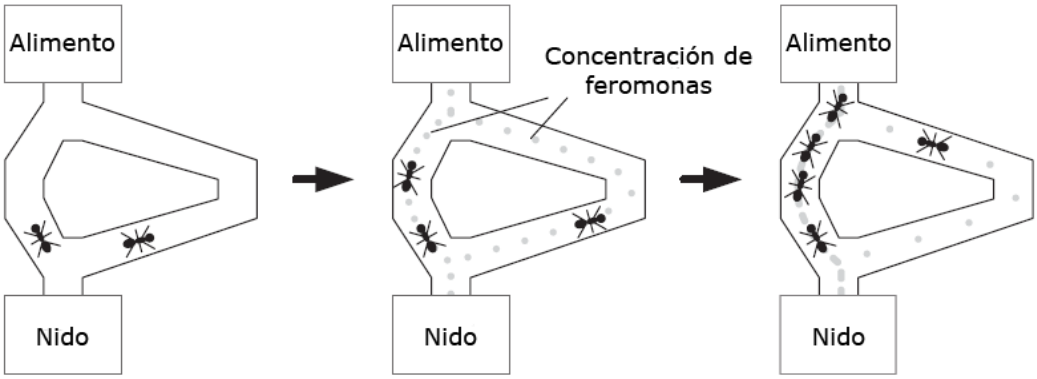


Figura 3.1: Camino más Corto seleccionado por las Hormigas

## 3.2. Metaheurísticas de optimización basada en colonias de hormigas - ACO

Los algoritmos ACO tienen la inspiración del comportamiento de las colonias de hormigas reales para resolver problemas de optimización combinatoria. En los algoritmos ACO el rastro de feromona químico es simulado por un modelo probabilístico parametrizado llamado: **modelo de feromona**. Este modelo consiste en un conjunto de parámetros denominados **valores de feromona**. El ingrediente básico de los algoritmos ACO es una heurística constructiva que es usada para ensamblar soluciones en forma probabilística. En general, la aproximación ACO intenta resolver problemas POC por la iteración de los siguientes pasos:

- Las soluciones son construidas usando el modelo de feromona, que es una distribución probabilística parametrizada sobre el espacio de soluciones.

- Luego, las soluciones son utilizadas para modificar los valores de feromona de manera que las futuras muestras se encaminen hacia soluciones de alta calidad.

**Algoritmo 7** Ant Colony Optimization (ACO)

**mientras** condición de terminación no alcanzado **hacer**
$PlanDeActividades$
$ConstruccionBasadaEnSolucionesHormigas()$
$ActualizacionFeromona()$
$AccionDemonio()$ {opcional}
$Fin\ PlanDeActividades$
**fin mientras**

**Salida:** Mejor solución encontrada

El marco general de la metaheurística ACO se muestra en el algoritmo 7. Éste considera tres componentes que se encuentran reunidos en la construcción del *PlanDeActividades*. La construcción del *PlanDeActividades* no especifica como estas actividades son organizadas y sincronizadas. Esto es realizado por el diseñador del algoritmo.

Los algoritmos ACO son esencialmente algoritmos constructivos: en cada iteracción del algoritmo, cada hormiga construye una solución al problema a través de la construcción de un grafo. Cada arista del grafo, representa los posibles caminos que la hormiga puede tomar, tiene asociado dos tipos de información que guía al movimiento:

- **Información Heurística:** es denotada por $\eta_{rs}$ y mide la preferencia del movimiento del nodo $r$ al nodo $s$ a través de la arista $a_{rs}$.
- **Información sobre el rastro de feromona (Artificial):** es denotada por $\tau_{rs}$ y mide "la deseabilidad aprendida" de movimiento e imita la feromona real que las hormigas depositan. Esta información es modificada durante la ejecución del algoritmo dependiendo de las soluciones encontradas por las hormigas.

## 3.3. Tipos de problemas resueltos por ACO

El tipo de problemas que puede ser resuelto por hormigas artificiales pertenecen al grupo "problemas del camino más corto" que pueden ser categorizados por los siguientes aspectos:

- Hay un conjunto de restricciones $\Omega$ definidos por el problema a solucionar.
- Hay un conjunto finito de componentes $N = \{n_1, n_2, ..., n_3\}$

- El problema presenta estados definidos a través de una secuencia ordenada de componentes $\delta =< n_r, n_s, ..., n_u, ... >$ (<r,s, ...,u, ...> para simplificar). Sobre los elementos de $N$. Si $\Delta$ es el conjunto de todas las posibles secuencias, denotamos con $\Lambda$ al conjunto de (sub)secuencias realizables con respecto a las restricciones $\Omega$. Los elementos de $\Lambda$ definen estados realizables. $|\delta|$ es la longitud de la secuencia $\delta$, y es el número de componentes de la secuencia.
- Existe una estructura de vecino definida como sigue: $\delta_2$ es el vecino de $\delta_1$ si 1) ambos pertenecen a $\Delta$. 2) el estado de $\delta_2$ puede ser alcanzado por $\delta_1$ en un solo movimiento lógico.
- Una solución S es un elemento de $\Lambda$ verificando todos los requerimientos del problema.
- Hay un costo C(S) asociado a cada solución S.
- Algunas veces, un costo o costo estimado puede estar asociado a estados.

Como se ha dicho, todas las características previas sujetas a problemas de optimización combinatoria que pueden ser representados por un grafo con pesos G = (N, A) donde A es el conjunto de aristas que conectan el conjunto de componentes $N$. El grafo se llama generalmente "Grafo de construcción G".

- El componente $n_r$ son los nodos del grafo.
- Los estados $\delta$ (y por lo tanto las soluciones $S$) corresponden al camino en el grafo.
- La arista $a_{rs}$ son conexiones/transiciones que definen la estructura del vecino. $\delta_2 =< \delta_1, S >$ es un vecino de $\delta_1$ si el nodo $r$ es el último componente de $\delta_1$ y la arista $a_{rs}$ existe el en grafo.
- Puede haber costos explícitos $c_{rs}$ sobre las transiciones asociadas a cada arista.
- Los componentes y conexiones pueden tener asociado un rastro de feromona $\tau$ que representa alguna forma indirecta de memoria a largo plazo el proceso de búsqueda, y también posee valores heurísticos $n$ que representan información heurística disponible sobre el problema bajo solución.

## 3.4. La hormiga artificial

La hormiga artificial es un simple agente computacional que trata de construir soluciones posibles al problema explotando la feromona disponible y la información heurística. Sin embargo, si es necesario, también puede construir soluciones no factibles que pueden ser penalizadas. Posee las siguientes propiedades:

- Busca costo mínimo en las soluciones factibles.

- Posee una memoria $L$ que guarda información sobre el camino hasta el momento. $L$ guarda la secuencia generada. Esta memoria puede ser usada para: i) construir soluciones factibles. ii) evaluar las soluciones generadas, iii) remarcar el camino para las hormigas que la siguen.
- Posee un estado inicial $\delta_{initial}$ que usualmente corresponde a una secuencia unitaria, y una o más condiciones de finalización asociada.
- Comienza con el estado inicial $\delta_{initial}$ y se mueve hacia estados posibles, construyendo una solución asociada.
- Cuando se está en un estado $\delta_r =< \delta_{r-1}, r >$ (es decir que está en el nodo $r$ y anteriormente siguió la secuencia $\delta_{r-1}$), puede moverse a cualquier nodo $s$ de los vecinos posibles $N(r)$, definido como $N(r) = \{s \mid (a_{rs} \in \Lambda)\ and\ (< \delta_r, s > \in \Lambda)\}$
- El movimiento es realizado aplicando una regla de transición, que es una función local que tiene en cuenta: el rastro de feromona y valores heurísticos, la memoria privada de la hormiga, y las restricciones del problema.
- Cuando una hormiga se mueve del nodo $r$ al $s$, durante el procedimiento de construcción, puede actualizar el rastro de feromona $\tau_{rs}$ asociado a la arista $a_{rs}$. Este proceso es llamado *actualización en línea paso a paso del rastro de feromona* (*online step-by-step pheromone trail update*).
- El procedimiento de construcción termina cuando cualquier condición de terminación es alcanzada, usualmente cuando el estado objetivo es alcanzado.

De esta manera, el único mecanismo de comunicación entre las hormigas es la estructura de datos que almacena los distintos niveles de feromona de cada arista o componente (memoria compartida).

## 3.5. Modo de operación y estructura genérica de un algoritmo ACO

El modo básico de operación de un ACO es el siguiente: las $m$ hormigas (artificiales) de la colonia se mueven, concurrentemente y asincrónicamente, a través de estados adyacentes del problema. Este movimiento es realizado de acuerdo a una regla de transición que está basada en información local disponible en los componentes (nodos). Esta información local contiene datos heurísticos y memoriales para guiar la búsqueda. Moviéndose sobre el grafo en construcción, las hormigas incrementalmente construyen soluciones. Opcionalmente, las hormigas pueden liberar feromona cada vez que recorren una arista (conexión) mientras construyen soluciones. Una vez que todas las hormigas han generado una solución, está es evaluada y luego se puede depositar una cantidad de feromona en función de la calidad de la solución. Esta información guiará la búsqueda de las otras

hormigas en el futuro.

El modo de operación genérico de ACO también incluye dos operaciones adicionales, **evaporación del rastro de feromona** (*pheromone trail evaporation*) y **acciones del demonio** (*daemon action*). La evaporación de la feromona es disparada por el entorno y es usada como un mecanismo para evitar el estancamiento y permitir a la hormiga explorar nuevas regiones.

Las acciones del demonio son opcionales (las colonias de hormigas reales no aplican este tipo de acciones) para implementar tareas desde una perceptiva global que están desprovistas en la perspectiva local de la hormiga. Por ejemplo, observar la calidad de todas las soluciones generadas y liberar una cantidad de feromona adicional sólo en las transiciones asociadas a algunas soluciones, o aplicar un proceso de búsqueda local a las soluciones generadas por las hormigas antes que actualicen el rastro de feromona. En ambos casos, el demonio reemplaza el proceso llamado "**actualización retrasada en línea de feromona**" (*online delayed pheromone update*) por el proceso llamado "**actualización offline del rastro de feromona**" (*offline pheromone trail update*).

Estructura de un algoritmo ACO:

```
Procedimiento ACO_Metaheurística
    parámetros_inicialización
    mientras (criterio_terminación_no_satisfecha)
        planificar_actividades
            generación_y_actividad_hormiga()
            evaporación_feromona()
            acciones_demonio()
        fin planificar_actividades
    fin mientras
fin procedimiento
```

```
Procedimiento generación_y_actividad_hormiga()
    repetir en paralelo para k=1 hasta m (n\'{u}mero de hormigas)
        nueva_hormiga()
    fin repetir en paralelo
fin procedimiento
```

```
Procedimiento nueva_hormiga(id_hormiga)
    inicializar_hormiga(id_hormiga)
    L = actualizar_memoria_hormiga()
    mientras (estado_actual!=estado_objetivo)
        P = calculo_probabilidad_transición(A,L,Ω)
        proximo_estado = aplicar_política_decisión_hormiga(P,Ω)
```

```
        mover_al_nuevo_estado()
        si (actualizar_feromona_online_paso_a_paso)
            depositar_feromona_sobre_aristas_visitadas()
        fin si
        L = actualizar_estado_interno()
    fin mientras
    si (actualizar_retardo_feromona_online)
        para cada arista visitada
            depositar_feromona_sobre_aristas_visitadas()
        fin para
    fin si
    liberar_recursos_hormiga(id_hormiga)
fin procedimiento
```

El primer paso involucra la inicialización de los valores de los parámetros considerados por el algoritmo. Entre otros, el valor inicial del rastro de feromona asociado a cada transición $\tau_0$, que es un pequeño valor positivo que es generalmente el mismo para todas las conexiones/componentes, el número de hormigas en la colonia $m$, y el peso que define el balance entre información metaheurística y memorial en las reglas de transición probabilísticas.

El procedimiento principal del administrador metaheurístico ACO, *planificar_actividades*, consiste en la planificación de los tres componentes mencionados en ésta sección: i) la generación y operación de las hormigas artificiales, ii) la evaporación de la feromona, y iii) las actividades del demonio. La implementación de este constructor define la sincronización existente entre los tres componentes. Mientras que la aplicación "clásica" a problemas NP-Completo (no distribuida) utiliza preferentemente una planificación secuencial, en problemas distribuidos como en redes de ruteo en paralelo la implementación del algoritmo puede resultar sencillo y eficientemente explotado.

Varios de los componentes son opcionales o dependen del algoritmo especifico ACO, como las acciones del demonio donde se puede decidir donde y cuando se realizara el deposito de feromona. Generalmente, la actualización online paso a paso y online retrasada del rastro de feromona son mutuamente excluyentes y usualmente no están presentes (si ambas faltan, generalmente el demonio actualiza el rastro de feromona).

## 3.6. Algoritmos

En ésta sección se presenta un resumen de los primeros algoritmos ACO, utilizado en el presente trabajo: **Sistema de Hormigas** (*Ant System - AS*), **Sistema de Hormigas Elitista** (*Elitist Ant System - EAS*), **Sistema de Colonias de Hormigas** (*Ant Colony System - ACS*), **Sistema de Hormiga Max-Min** (*Max-Min Ant System - MMAS*) y por último **Sistema de Hormigas Basado en Jerarquización** (*Rank-based Ant System - $AS_{rank}$*).

### 3.6.1. Sistema de hormigas (Ant System - AS)

Desarrollado por Dorigo, Maniezzo y Colorni en 1991 [8], fue el primer algoritmo ACO. Inicialmente, posee tres diferentes variantes AS-density, AS-quantity y AS-cycle, diferenciándose en la forma en que actualizan el rastro de feromona. En los primeros dos, la hormiga libera feromona mientras construye sus soluciones (*online step-by-step pheromone update*), estos se diferencian en que AS-density deposita una cantidad constante de feromona mientras que en AS-quantity depende de la deseabilidad de la heurística de transición $\eta_{ij}$. Por último, en AS-cycle la feromona se deposita una vez que la solución se completa (*online delayed pheromone update*). Esta última variante fue la de mejor rendimiento y actualmente es la que refieren los libros como AS.

AS se caracteriza por la forma en que la actualización de la feromona es activada una vez que todas las hormigas han completado sus soluciones. Primero, todos los caminos de feromonas son reducidos por un factor constante, implementando de esta manera la evaporación de feromona. Segundo, toda hormiga de la colonia deposita una cantidad de feromona en función de la calidad de su solución. Inicialmente, AS no usa acciones de demonio, pero es muy aconsejable hacerlo, por ejemplo: agregar una búsqueda local para refinar las soluciones generadas por las hormigas.

Las soluciones en AS son construidas de la siguiente manera. A cada paso de la construcción, una hormiga $k$ en AS elige ir al siguiente nodo con una probabilidad que es computada como se muestra en la ecuación 3.1.

$$p_{rs}^{k} = \begin{cases} \frac{[\tau_{rs}]^{\alpha}.[\eta_{rs}]^{\beta}}{\sum_{u \in N_k^(r)} [\tau_{ru}]^{\alpha}.[\eta_{ru}]^{\beta}}, & \text{si } s \,\epsilon\, N_k(r) \\ 0, & \text{otro caso} \end{cases} \tag{3.1}$$

Donde $N_k(r)$ es el vecindario factible de la hormiga $k$ cuando está en el nodo $r$, en tanto que, $\alpha$, $\beta \in \mathbb{R}$ son dos parámetros que pesan la importancia relativa del rastro de feromona y la información heurística. Cada hormiga $k$ almacena la secuencia (memoria $L_k$) que ha seguido hasta el momento, esta secuencia es explotada para determinar $N_k(r)$ en cada paso de la construcción.

Con respeto a los parámetros $\alpha$ y $\beta$ su rol es como sigue: si $\alpha$ = 0, los nodos con mejor heurística tienen una mayor probabilidad de ser elegidos, convirtiéndolo en un algoritmo *greedy*. Sin embargo, si $\beta$ = 0, sólo los rastros de feromona son considerados para guiar al proceso de construcción, que pueden causar un rápido estancamiento, situación donde el rastro de feromona asociado a una transición es significativamente más alto que el resto. Esto provoca que la hormiga siempre construya la misma solución, usualmente un mínimo local. El depósito de feromona es realizado una vez que todas las hormigas han terminado de construir sus soluciones. Primero, el rastro de feromona asociado a cada arco se evapora reduciendo todas las feromonas por un factor constante como se muestra en la ecuación 3.2.

$$\tau_{rs} \leftarrow (1-\rho).\tau_{rs} \tag{3.2}$$

Donde, $\rho \in (0, 1]$ es la tasa de evaporación. Luego, cada hormiga remarca el camino que ha seguido (este camino es almacenado en su memoria local $L_k$) y deposita una cantidad de feromona por cada conexión recorrida de acuerdo a la ecuación 3.3.

$$\tau_{rs} \leftarrow \tau_{rs} + f(C(S_k)),\ \ \forall a_{rs} \in S_k \tag{3.3}$$

Donde, $f(s)$ es la función que devuelve el valor a minimizar (o maximizar) relacionada al costo y la función $C(s)$ devuelve el costo de una solución. Para ATSP $f(s) = \frac{1}{C(s)}$ y $C(s)$ es igual a la función objetivo (ecuación 1.1).

Ahora veremos la composición del procedimiento *nueva_hormiga* para este algoritmo:

```
1  Procedimiento nueva_hormiga (ant_id)
2      k = ant_id; r = generate_initial_state; S_k = r
3      L_k = r
4      Mientras (estado_actual != estado_objetivo)
5          Para cada s ∈ N_k(r) hacer p_rs^k = [τ_rs]^α.[η_rs]^β / Σ_{u∈N_r^k} [τ_ru]^α.[η_ru]^β
6          estado_siguiente = aplicar_política_decisión_hormiga(P, N_k(r))
7          r = estado_siguiente; S_k = <S_k, r>
8          ---
9          L_k = L_k ∪ r
10     Fin Mientras
11     \*  El procedimiento evaporacion_feromona() se activa y evapora
           feromona en cada arista a_rs: τ_rs = (1 − ρ).τ_rs  */
12     Para cada arista a_rs ∈ S_k hacer
13         τ_rs = τ_rs + f(C(S_k))
14     Fin Para
15     liberar_recursos_hormiga(ant_id)
16 Fin Procedimiento
```

Es necesario aclarar que la línea 8 vacía está incluida para remarcar que no hay actualización de feromona paso a paso *online* y lo que se encuentra en la línea 11 es para marcar que la evaporación de feromona es realizada por el demonio.

### 3.6.2. Sistema de hormigas elitista (Elitist Ant System - EAS)

Desarrollado por Dorigo, Maniezzo y Colorni en 1991 [8]. Para un mejor rendimiento del AS se propone un algoritmo llamado *Sistema de Hormigas Elitista (Elitist Ant System - EAS)*. En EAS, una vez que las hormigas han liberado feromona sobre las conexiones asociadas a sus soluciones generadas, el demonio realiza un depósito adicional de feromona sobre las aristas pertenecientes a la mejor solución encontrada hasta el momento en el proceso de búsqueda (esta solución es también llamada "mejor solución global"). La cantidad de feromona depositada, dependerá de la calidad de la mejor solución encontrada, es pesada por la cantidad de hormigas elitistas $e$ consideradas como se muestra en la ecuación 3.4.

$$\tau_{rs} \leftarrow \tau_{rs} + e.f(C(S_{global-best})), \ \ \forall a_{rs} \in S_{global-best} \tag{3.4}$$

### 3.6.3. Sistema de colonias de hormigas (Ant Colony System - ACS)

El sistema de colonia de hormigas [7] fue uno de los primeros sucesores de AS. Se introdujeron 3 grandes cambios en AS:

1. ACS usa una diferente regla de transición, que es llamada regla proporcional pseudo-aleatoria (*pseudo-random proportional rule*): si $k$ es una hormiga posicionada en el nodo $r$, $q_0 \in [0,1]$ es un parámetro y $q$ un número aleatorio entre [0,1]. El próximo nodo $s$ es accedido aleatoriamente de acuerdo a la siguiente distribución de probabilidades:

   - si $q \leq q_0$ se realiza una explotación del espacio de búsqueda de acuerdo a la ecuación 3.5.

$$p_{rs}^{k} = \begin{cases} 1 & \text{si s = arg } \underbrace{max}_{u \in N_k(r)} \{\tau_{ru}.\eta_{ru}^{\beta}\} \\ 0, & \text{otro caso} \end{cases} \tag{3.5}$$

   - En el caso de $q > q_0$ se realiza una exploración controlada del espacio de soluciones de acuerdo a la ecuación 3.6.

$$p_{rs}^{k} = \begin{cases} \frac{[\tau_{rs}]^{\alpha}.[\eta_{rs}]^{\beta}}{\sum_{u \in N_r(k)} [\tau_{ru}]^{\alpha}.[\eta_{ru}]^{\beta}}, & \text{si } s \, \epsilon \, N_k(r) \\ 0, & \text{otro caso} \end{cases} \tag{3.6}$$

Como podemos observar, la regla tiene un objetivo doble: cuando $q \leq q_0$, el algoritmo explota el conocimiento disponible, eligiendo la mejor opción con respecto a la información heurística y el rastro de feromona. Sin embargo, si $q > q_0$, éste aplica una exploración controlada, como en AS. En conclusión la regla establece una compensación entre la exploración de nuevas conexiones y explotación de la información disponible hasta el momento.

2. Sólo el demonio (y no la hormiga individual) activa la actualización de feromona, una actualización *offline* del rastro de feromona es realizada (*offline pheromone trail update*). Para hacer esto, ACS considera una sola hormiga, la que generó la mejor solución global $S_{global-best}$.

   La actualización de la feromona se realiza primero evaporando el rastro de feromona en todas las conexiones usadas por la hormiga de mejor solución ecuación 3.7. Está actualización junto con la nueva regla de transición intentan realizar una búsqueda mas dirigida.

$$\tau_{rs} \leftarrow (1-\rho).\tau_{rs}, \ \ \forall a_{rs} \in S_{global-best} \tag{3.7}$$

   Luego, el demonio deposita feromona de acuerdo a la ecuación. 3.8.

$$\tau_{rs} \leftarrow \tau_{rs} + \rho.f(C(S_{global-best})), \ \ \forall a_{rs} \in S_{global-best} \tag{3.8}$$

   Adicionalmente, el demonio puede aplicar una búsqueda local para mejorar la solución de la hormiga antes de actualizar el rastro de feromona.

3. Las hormigas aplican la actualización *online* paso a paso del rastro de feromona (*online step-by-step pheromone trail update*) que alienta la generación de diferentes soluciones a aquellas recién encontradas. Cada vez que una hormiga recorre una arista $a_{rs}$, se le aplica la ecuación 3.9.

$$\tau_{rs} \leftarrow (1-\varphi).\tau_{rs} + \varphi.\tau_0 \tag{3.9}$$

   donde $\varphi \in (0,1]$ es un parámetro de decaimiento. Como se puede observar, la regla de actualización *online* paso a paso del rastro de feromona incluye ambos, evaporación y depósito de feromona. Dado que la cantidad de feromona depositada es muy pequeña, la aplicación de esta regla hace que el rastro de feromona sobre las conexiones recorridas por la hormiga se decremente. Por lo tanto, esto resulta en una técnica adicional de exploración de ACS

haciendo que las conexiones recorridas por una hormiga sean menos atractivas para las siguientes hormigas y también para ayudar a evitar que todas las hormigas sigan el mismo camino.

El procedimiento *nueva_hormiga* y *acciones_demonio* para ACS son los siguientes:

```
Procedimiento nueva_hormiga (ant_id)
    k = ant_id; r = generate_initial_state; S_k = r
    L_k = r
    Mientras (estado_actual != estado_objetivo)
        Para cada s ∈ N_k(r) hacer calculo b_rs = τ_rs.η_rs^β
        q = generar_valor_aleatorio_entre[0,1]
        Si (q <= q0)
            estado_siguiente = max(b_rs, N_k(r))
        sino
            Para cada s ∈ N_k(r) hacer
                p_rs^k = b_rs / Σ_{u∈N_r^k} b_rs
            End Para
            estado_siguiente = aplicar_política_decisión_hormiga(P,N_{k}(r))
        End Si
        r = estado_siguiente; S_k =< S_k, r >

        L_k = L_k ∪ r
    Fin Mientras
    ---
    ---
    ---
    liberar_recursos_hormiga(ant_id)
Fin Procedimiento

Procedimiento acciones_demonio
    Para cada S_k hacer búsqueda_local(S_k) {opcional}
    S_mejor−actual = mejor_solución(S_k)
    Si (mejor(S_mejor−actual, S_mejor−global))
        S_mejor−global = S_mejor−actual
    Fin Si
    Para cada arista a_rs ∈ S_mejor−global hacer
        {El procedimiento evaporación_feromona() se activa y evapora
           feromona en la arista a_rs : τ_rs = (1 − ρ).τ_rs}

        τ_rs = τ_rs + ρ.f(C(S_mejor−global))
```

```
35      Fin Para
36  Fin Procedimiento
```

### 3.6.4. Sistema de hormiga max-min (Max-Min Ant System - MMAS)

El sistema de hormiga Max-Min (MMAS) [25, 24, 26], desarrollado por Stützle y Hoos en 1996, es una de las mejores extensiones de rendimiento de AS. Se incorporaron los siguientes aspectos:

1. Se aplica una actualización *offline* del rastro de feromona (*offline pheromone trail update*), similar al ACS. Después de que todas las hormigas han construido una solución, primero todo rastro de feromona es evaporado según la ecuación 3.10.

$$\tau_{rs} \leftarrow (1-\rho).\tau_{rs} \tag{3.10}$$

   y luego la feromona es depositada de acuerdo a la ecuación 3.11.

$$\tau_{rs} \leftarrow \tau_{rs} + f(C(S_{mejor})), \forall a_{rs} \in S_{best} \tag{3.11}$$

   La mejor hormiga que tiene permiso para agregar feromona puede ser la solución de mejor-iteración o la mejor-global. Resultados experimentales muestran que el mejor rendimiento fue obtenido, si gradualmente se incrementa la frecuencia de elección de la solución mejor-global para la actualización del rastro de feromona. [24, 26]. Por otro lado, en MMAS las soluciones de hormigas son mejoradas, usando frecuentemente optimizadores locales antes de la actualización del rastro de feromona.

2. El posible valor para el rastro de feromona es limitado por el rango $[\tau_{min}, \tau_{max}]$. La posibilidad de que se produzca un estancamiento del algoritmo se reduce dando a cada conexión una probabilidad pequeña de ser elegida. La evaporación de feromona y el máximo nivel posible del rastro de feromona son limitados por la ecuación 3.12; donde $S^*$ es la solución óptima. Reemplazando $S^*$ con $S_{mejor}$ en la ecuación $\tau^*_{max}$, la mejor solución global puede utilizarse para estimar $\tau_{max}$. Para calcular $\tau_{min}$, a menudo, es suficiente con elegir algún factor pequeño ($\tau_r$) y constante de $\tau_{max}$.

$$\tau^*_{max} = 1/(\rho.C(S^*)) \tag{3.12}$$

3. En lugar de iniciar los valores de feromonas con un valor pequeño, en MMAS los caminos de feromonas son inicializados usando un valor estimado como máximo valor permitido (El valor estimado puede obtenerse primero generando alguna solución $S$ por un algoritmo *greedy* construyendo heurísticas y luego reemplazando $S$ en la ecuación por $\tau^*_{max}$). Esto nos lleva a un componente de diversificación adicional en el algoritmo, porque en el inicio las diferencias relativas de los rastros de feromona no son muy marcadas, lo cual difiere a la inicializacion de $\tau_{rs}$ con valores pequeños.

La estructura del procedimiento *acciones_demonio* en MMAS es:

```
1  Procedimiento acciones_demonio
2      Para cada S_k hacer búsqueda_local(S_k)
3      S_mejor−actual = mejor_solucion(S_k)
4      Si (mejor(S_mejor−actual, S_mejor−global))
5          S_mejor−global = S_mejor−actual
6      Fin Si
7      S_best = decision(S_mejor−global, S_mejor−actual)
8      Para cada arista a_rs ∈ S_mejor−global hacer
9          τ_rs = τ_rs + ρ.f(C(S_best))
10         si (τ_rs < τ_min)τ_rs = τ_min
11     Fin Para
12     si (condición_stagnation)
13         Para cada arista a_rs hacer τ_rs = τ_max
14     fin si
15 Fin Procedimiento
```

### 3.6.5. Sistema de hormigas basado en jerarquización (Rank-based Ant System - $AS_{rank}$)

El sistema de hormigas basado en *ranking* o jerarquización [3] es otra extensión de AS propuesto por Bullenheimer, Hartl y Strauss en 1997. Éste incorpora la idea de jerarquización dentro de la actualización de feromona, que es otra vez desarrollada globalmente por el demonio como sigue:

1. Se seleccionan las $m$ hormigas para la jerarquización de acuerdo a la calidad de sus soluciones: $(S'_1, .., S'_m)$, siendo $S'_1$ la mejor solución generada hasta el momento.
2. El demonio deposita feromona en las conexiones recorridas por las mejores hormigas (hormigas elitistas). La cantidad de feromona depositada depende directamente de la jerarquización de la hormiga y de la calidad de la solución.
3. Las conexiones recorridas por la mejor solución global reciben una cantidad adicional de feromona que depende de la calidad de la solución. Este depósito de feromona es considerado el más importante, por lo tanto, recibe un peso de $\sigma$.

Este modo de operación es puesto en efecto por medio de la ecuación 3.13, que se aplica a cada arista una vez que se ha evaporado todo el rastro de feromona.

$$\tau_{rs} \leftarrow \tau_{rs} + \sigma.\Delta_{\tau_{rs}^{gb}} + \Delta_{\tau_{rs}^{rank}} \tag{3.13}$$

$$Donde\ \Delta_{\tau_{rs}^{gb}} = \begin{cases} f(C(S_{mejor-global})), & si\ a_{rs} \in S_{mejor-global}, \\ 0, & otro\ caso \end{cases}$$

$$\Delta_{\tau_{rs}^{rank}} = \begin{cases} \sum_{\mu=1}^{\sigma-1}(\sigma-\mu).f(C(S_{\mu}^{'})), & si\ a_{rs} \in S_{\mu}^{'} \\ 0, & otro\ caso \end{cases}$$

Por lo tanto, el procedimiento demonio $AS_{rank}$ presenta la siguiente estructura:

```
Procedimiento acciones_demonio
    Para cada S_k hacer búsqueda_local(S_k) { opcional }
    rank (S_1, ..., S_m) en orden decreciente de la soluci\'{o}n
    calidad dentro (S'_1, ..., S'_m)
    Si (mejor(S'_1, S_mejor-global))
        S_mejor-global = S'_1
    Fin si
    Para μ = 1 to (σ-1) hacer
        Para cada arista a_rs ∈ S'_u hacer
            τ_rs = τ_rs + (σ - μ).f(C(S'_u))
        Fin para
    Fin para
    Para cada arista a_rs ∈ S_mejor-global hacer
        τ_rs = τ_rs + σ.f(C(S_mejor-global))
    Fin para
Fin procedimiento
```

# Capítulo 4

# Experimentación y análisis de resultados

Con el propósito de llevar a cabo el análisis y comparación estadística de las distintas variantes ACO (AS, EAS, ACS, MMAS y ASrank), considerando diferentes configuraciones de los principales parámetros en cada una de ellas, se resolvieron las instancias de ATSP mostradas en la tabla 4.1 y extraídas de la librería TSPLIB[1]. Para ello se analiza a la variación aquellos parámetros que caracterizan a cada algoritmo como por ejemplo $\rho$ y $\beta$. Es importante aclarar que los experimentos sobre los distintos parámetros se realizaron en el orden en que se ubican en la tabla 4.2. Por ende, cuando se experimenta con un nuevo parámetro, las variables precedentes son configuradas con el valor que por lo general permite obtener mejores resultados. Por ejemplo: cuando se estudia la variación de $\beta$ en ASrank, el valor de $\rho$ usado es 0.1. En cada caso se llevan a cabo 30 ejecuciones con 1000 iteraciones sobre una población de 450 individuos, con un $\alpha = 1$, obtenido de las distintas bibliografías. Para esto se utilizan maquinas con procesadores AMD Phenom 8450 de 3 núcleos de 2.1 MHz con 2 GByte de memoria, sistema operativo Slackware 12.2.0 con el kernel 2.6.27.7.

| | ♯ Ciudades | Costo Mínimo |
|---|---|---|
| br17 | 17 | 39 |
| ftv35 | 36 | 1473 |
| ftv55 | 56 | 1608 |
| ftv70 | 71 | 1950 |
| kro124p | 100 | 36230 |
| rbg358 | 358 | 1163 |
| rbg443 | 443 | 2720 |

Tabla 4.1: Instancias extraídas de la librería TSPLIB.

El comportamiento de cada algoritmo se analizó desde diferentes puntos de vista. Estos son: el error porcentual del mejor costo obtenido respecto al mejor costo encontrado en la literatura (Eb),

[1] http://www2.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/

el error porcentual del mejor costo promedio respecto al mejor costo encontrado (Em) y la iteración promedio donde se encontró el mejor costo (Ei). Para llevar a cabo dicho estudio, se realizan tests estadísticos de varianza no paramétricos (test Kruskal-Wallis); ya que el conjunto de datos no pasó la prueba de normalidad y homogeneidad de la varianza, condiciones necesarias para aplicar ANOVA. El nivel de confianza $\alpha_r$ es igual a 0.05 para todos los tests realizados y el valor de probabilidad de dichas pruebas ($p$) es la probabilidad de rechazar erróneamente la hipótesis nula cuando esta sea verdadera. El valor $p$ es comparado con el nivel de confianza y, si este es menor, entonces los métodos analizados son notablemente distintos. Para este último caso se debe realizar una comparación múltiple para determinar qué algoritmo o qué grupos algorítmicos son mejores o peores.

| Parámetro | AS | EAS | ACS | MMAS | ASrank |
|---|---|---|---|---|---|
| $\rho$ | {0.1, 0.3, 0.5, 0.7, 0.9} | {0.1, 0.3, 0.5, 0.7, 0.9} | {0.1, 0.3, 0.5, 0.7, 0.9} | {0.1, 0.3, 0.5, 0.7, 0.9} | {0.1, 0.3, 0.5, 0.7, 0.9} |
| $\beta$ | {1,3,5,7,9} | {1,3,5,7,9} | {1,3,5,7,9} | {1,3,5,7,9} | {1,3,5,7,9} |
| $e$ | | {1,3,5,7,9} | | | |
| $\varphi$ | | | {0.1, 0.3, 0.5, 0.7, 0.9} | | |
| $q_0$ | | | {0.1, 0.3, 0.5, 0.7, 0.9} | | |
| $\tau_r$ | | | | $\{\frac{1}{40}, \frac{1}{80}, \frac{1}{120}, \frac{1}{160}, \frac{1}{200}\}$ | |
| $\sigma$ | | | | | {1,3,5,7,9} |

Tabla 4.2: Parámetros utilizados en las distintas pruebas realizadas.

La representación de una solución esta dada por una lista de ciudades, donde la primer ciudad de la lista es el origen del *tour*, y las siguientes ciudad de la lista son las ciudades por las que va pasando, donde la ultima es la misma de donde partió. Por ejemplo, para 8 ciudades una representación valida se muestra en la tabla 4.3. En cuanto a la información heurística para el problema ATSP se calcula según la ecuación 4.1, donde se puede ver que a la distancia se le suma el valor 1, realizado por la instancia br17 que posee distancias de valor 0.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 1 |

Tabla 4.3: Representación de una solución para 8 ciudades.

$$\eta_{rs} \leftarrow \frac{1}{d_{rs} + 1} \tag{4.1}$$

La cantidad de experimentos realizados fueron de 525. Donde un conjunto de pruebas consistió en tomar para un tipo de algoritmo un parámetro junto con un valor de la tabla 4.2 y realizar el experimento para las 7 instancias.

En las tablas 4.4, 4.5, 4.6, 4.7 y 4.8 se resumen los resultados del test de varianza mostrando aquellos conjuntos de valores paramétricos donde AS, EAS, ACS, MMAS y ASrank logran su mejor comportamiento para cada instancia del problema. En estas tablas también se muestran los

valores promedios de Eb, Em y Ei asociados a cada conjunto de valores paramétricos previamente presentados. En las siguientes secciones se analizan los datos obtenidos por las distintas variantes algorítmicas (AS, EAS, ACS, MMAS, ASrank).

## 4.1. Análisis de los resultados obtenidos por AS

En primer lugar, se estudia el comportamiento de AS bajo distintos valores paramétricos según se muestra en la tabla 4.2. Se inicia el análisis con el test de varianza presentado en la tabla 4.4 y los valores Eb de la figura 4.1, considerando cada uno de los parámetros separadamente:

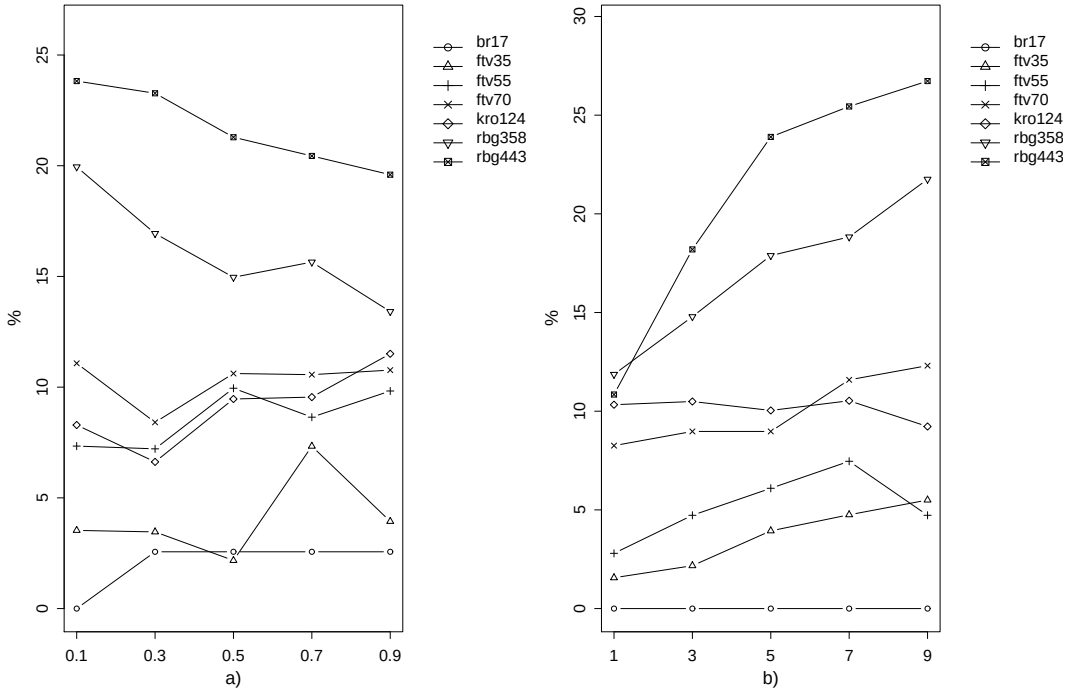


Figura 4.1: a) Valores de Eb considerando a $\rho$ b) Valores de Eb considerando a $\beta$.

**$\rho$.** Para realizar los experimentos con el parámetro $\rho$ se mantuvieron constantes los demás parámetros ($\beta = 5$). Se pueden distinguir dos grupos de instancias: uno constituido por aquellas cuyo número de ciudades es menor o igual a 100 y, un segundo grupo formado por las instancias de mayor complejidad (rbg358 y rbg443). En el primero, los valores de $\rho$ que permiten hallar los mejores resultados y que son comunes a todos los casos pertenecen al intervalo [0.1,..,0.3]. En cambio, para el segundo se observa que una alto valor de $\rho$ (0.9) permite encontrar los mejores resultados en esas tres instancias. En resumen, la diversifica-

ción y la intensificación son puntos críticos en AS cuando resuelve instancias mayores a 100 ciudades, donde es necesario considerar valores crecientes de $\rho$.

**$\beta$.** Se mantuvo constante $\rho$ = 0.3 para los experimentos con éste parámetro. Al analizar la importancia dada a la información heurística, se ha detectado que cuando $\beta$=1 (igual a $\alpha$), AS encuentra los mejores resultados en 6 de los 7 casos de prueba. Por lo cual, se recomienda que $\alpha$ y $\beta$ sean iguales a 1.

| Instancias | AS | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | $\rho$ | | | | $\beta$ | | | |
| | Valor | Eb | Em | Ei | Valor | Eb | Em | Ei |
| br17 | [0.1,...,0.9] | 2.05 | 58.74 | 0 | [1,...,9] | 0.00 | 11.59 | 0 |
| ftv35 | [0.1,...,0.5] | 3.05 | 11.45 | 4 | {1} | 1.56 | 1.79 | 57 |
| ftv55 | [0.1,...,0.7] | 8.29 | 14.53 | 19 | {1} | 2.80 | 5.02 | 56 |
| ftv70 | [0.1,...,0.9] | 10.29 | 17.77 | 12 | [1,...,3] | 8.62 | 11.48 | 49 |
| kro124p | [0.1,...,0.3] | 7.46 | 11.98 | 26 | [5,...,9] | 9.93 | 11.66 | 19 |
| rbg358 | {0.9} | 13.41 | 15.94 | 554 | {1} | 11.87 | 12.90 | 599 |
| rbg443 | {0.9} | 19.60 | 21.60 | 703 | {1} | 10.85 | 11.95 | 671 |

Tabla 4.4: Valores paramétricos con los cuales AS logra una mayor calidad en las soluciones de cada instancia y los valores promedio de Eb, Em y Ei correspondientes. (Eb y Em son errores porcentuales).

A partir de lo expuesto anteriormente, se infirió que $\rho$ y $\beta$ son parámetros sensibles al cambio de configuración cuando AS resuelve esta clase de problemas combinatorios. En cuanto al número promedio de iteraciones necesarias para obtener la mejor solución, se detecto que, independientemente del parámetro analizado, en todos los casos se necesita un número mayor de iteraciones si el número de ciudades se incrementa. En otras palabras, si el tamaño del espacio de búsqueda crece, entonces AS incrementa su esfuerzo para hallar su mejor solución. Aunque, este esfuerzo extra no alcance para mantener la calidad de las soluciones cercana al óptimo.

## 4.2. Análisis de resultados obtenidos por EAS

En este apartado se analizo el comportamiento de EAS bajo distintos valores paramétricos según se muestra en la tabla 4.2. Para ello estudiamos los resultados del test de varianza presentados en la tabla 4.5 y los valores Eb de las figuras 4.2 y 4.3, considerando por separado cada parámetro:

**$\rho$.** Se mantuvieron constantes $\beta$ = 1 y $e$ = 1 para los experimentos con éste parámetro. Como la evaporación del AS y EAS es similar, se puede observar la similitud en sus resultados respecto al parámetro $\rho$, igual que en AS se pueden distinguir dos grandes grupos definidos por la cantidad de ciudades. El primer grupo formado por las instancias menores o iguales a 100 ciudades; donde EAS encuentra los mejores resultados cuando se utilizan valores pequeños de $\rho$ ([0.1,...,0.3]), mientras que para el segundo grupo se obtiene los mejores resultados con valores altos (cercanos a 0.9).

$\beta$**.** Se mantuvieron constantes $\rho$ = 0.5 y $e$ = 1 para los experimentos con éste parámetro. También en el análisis del parámetro $\beta$ se hallan similitudes con AS. Es decir, se puede observar que los mejores resultados se encuentran cuando se utilizan valores de $\beta$ cercanos a 1, y un $\alpha = 1$. Quedando como excepción la instancia kro124p, donde el valor de $\beta$ varía entre [3,...,9].

$e$**.** Se mantuvieron constantes $\rho$ = 0.5 y $\beta$ = 1 para los experimentos con éste parámetro. En la todas las instancias se puede observar que aumentando la cantidad de hormigas elitistas (parámetro $e$) a valores entre [7,...,9] se obtienen las mejores soluciones.

EAS, al igual que AS, utiliza valores pequeños de $\rho$ para instancias con cantidad de ciudades menores o iguales a 100. Mientras que para instancias mayores requiere valores de $\rho$ grandes. En tanto que $\beta$ necesita ser igual a 1. Por último, se infiere que cuando el parámetro $e$ varía entre [7,...,9] se obtienen los mejores resultados para la mayoría de las instancias.

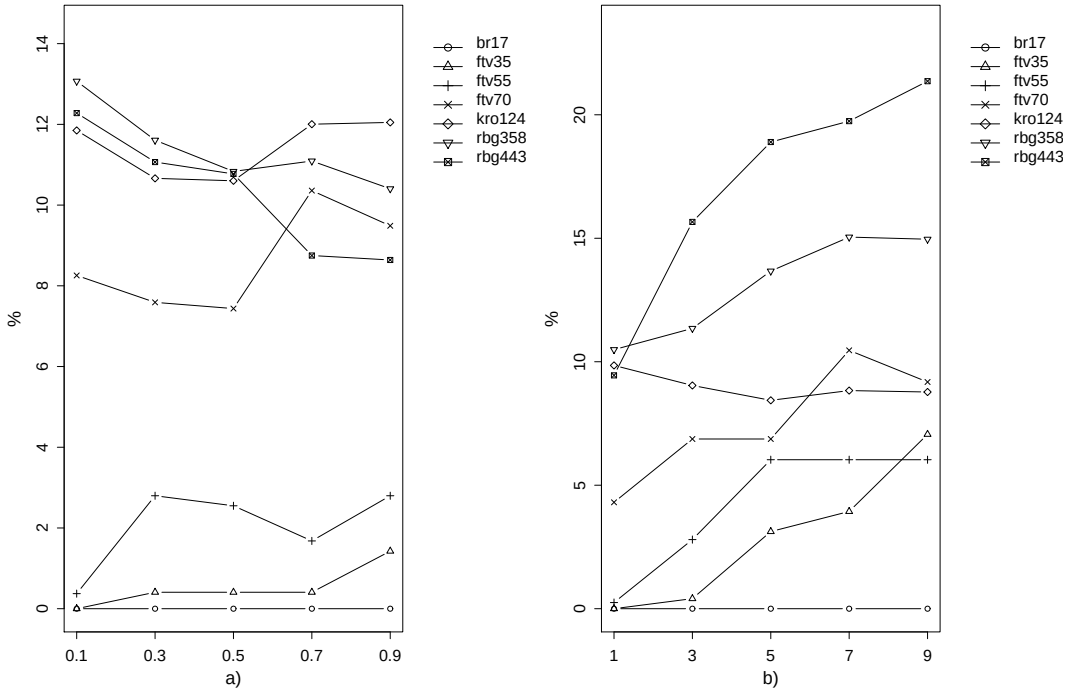


Figura 4.2: a) Valores de Eb considerando a $\rho$ b) Valores de Eb considerando a $\beta$.

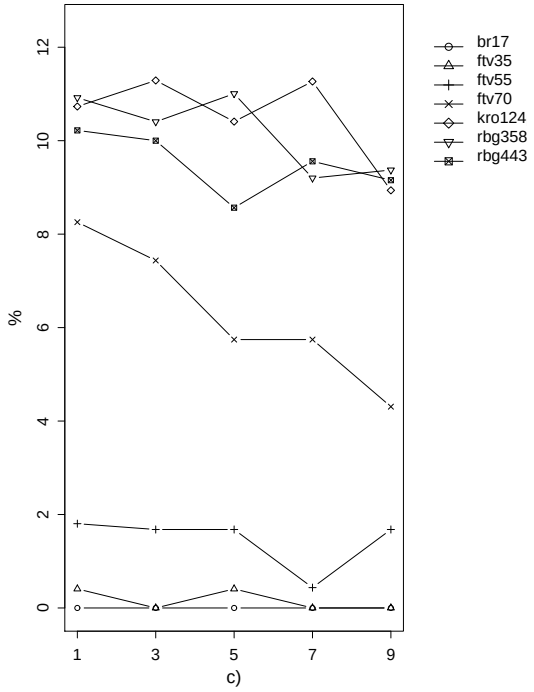


Figura 4.3: c) Valores de Eb considerando $e$.

| Instancias | EAS | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $\rho$ | | | | $\beta$ | | | | $e$ | | | |
| | Valor | Eb | Em | Ei | Valor | Eb | Em | Ei | Valor | Eb | Em | Ei |
| br17 | [0.1,...,0.9] | 0,00 | 0,00 | 0 | [1,...,9] | 0,00 | 13,23 | 0 | [1,...,9] | 0,00 | 0,00 | 0 |
| ftv35 | [0.1,...,0.3] | 0,20 | 1,43 | 68 | 1 | 0,00 | 1,65 | 85 | [1,...,9] | 0,16 | 1,64 | 39 |
| ftv55 | 0.1 | 1,85 | 4,86 | 52 | 1 | 0,25 | 3,59 | 40 | [5,...,9] | 1,26 | 3,66 | 377 |
| ftv70 | [0.1,...,0.5] | 7,76 | 10,61 | 142 | 1 | 4,31 | 8,63 | 73 | [5,...,9] | 5,26 | 9,26 | 352 |
| kro124p | [0.3,...,0.7] | 11,09 | 13,50 | 392 | [5,...,9] | 8,68 | 10,50 | 296 | [3,...,9] | 10,48 | 13,05 | 540 |
| rbg358 | 0.9 | 10,40 | 11,67 | 831 | 1 | 10,49 | 11,79 | 616 | [7,...,9] | 9,29 | 11,50 | 693 |
| rbg443 | 0.9 | 8,64 | 9,73 | 691 | 1 | 9,45 | 10,61 | 419 | [3,...,9] | 9,32 | 10,59 | 495 |

Tabla 4.5: Valores paramétricos con los cuales EAS logra una mayor calidad en las soluciones de cada instancia y los valores promedio de Eb, Em y Ei correspondientes. (Eb y Em son errores porcentuales).

## 4.3. Análisis de los resultados obtenidos por ACS

En esta sección, se estudia el comportamiento de ACS bajo distintos valores paramétricos según se muestra en la tabla 4.2. Se ha comenzado con el análisis del resumen del test de varianza presentado en la tabla 4.6 y los valores Eb de la figura 4.4, considerando cada uno de los parámetros separadamente:

**$\rho$.** Se mantuvieron constantes $\beta$ = 1, $\varphi$ = 0.1 y $q_0$ = 0.1 para los experimentos con éste parámetro. Se observan dos grupos de instancias: uno constituido por aquellas cuyo número de ciudades es menor o igual a 70 y, un segundo grupo formado por las instancias de mayor complejidad (kro124p, rbg358 y rbg443). En el primero, los valores de $\rho$ que permiten hallar los mejores resultados y que son comunes a todos los casos pertenecen al intervalo [0.1,..,0.3]; mientras que las dos instancias más pequeñas amplían este rango a 0.5 y a 0.9. En cambio, para el segundo se observa que una alto valor de $\rho$ (0.9) permite encontrar los mejores resultados en esas tres instancias. En resumen, la diversificación y la intensificación son puntos críticos en ACS cuando resuelve instancias con 124 o más ciudades, donde es necesario considerar valores crecientes de $\rho$.

**$\beta$.** Se mantuvieron constantes $\rho$ = 0.3, $\varphi$ = 0.1 y $q_0$ = 0.1 para los experimentos con éste parámetro. Al analizar la importancia dada a la información heurística, detectamos que cuando $\beta$=1 (igual a $\alpha$), ACS encuentra los mejores resultados en 6 de los 7 casos de prueba. Por lo cual, se recomienda que $\alpha$ y $\beta$ sean iguales a 1.

**$\varphi$.** Se mantuvieron constantes $\rho$ = 0.3, $\beta$ = 1 y $q_0$ = 0.1 para los experimentos con éste parámetro. En 6 de las 7 instancias, soluciones de similar calidad son obtenidas independientemente del valor asignado a esta variable. Por ende, la influencia de $\varphi$ en la actualización local no juega un rol determinante en ACS al resolver ATSP.

**$q_0$.** Se mantuvieron constantes $\rho$ = 0.3, $\beta$ = 1 y $\varphi$ = 0.1 para los experimentos con éste parámetro. Nuevamente, se observan que resultados similares se obtienen cuando se utilizan cualquiera de los valores determinados para este parámetro. Por ende, el control sobre la exploración y la explotación realizado con la aplicación de esta variable no juega un rol preponderante en la solución de ATSP.

A partir de lo expuesto anteriormente, se infirió que $\rho$ y $\beta$ son los parámetros más sensibles al cambio de configuración cuando ACS resuelve esta clase de problemas combinatorios. En tanto que, los valores asignados a las variables $\varphi$ y $q_0$ no causan cambios significativos en la la calidad de los resultados. Esto puede observarse en la similitud de las medias de los errores porcentuales, $Eb$ y $Em$, medidos en los distintos experimentos para una misma instancia. Por ejemplo: para la instancia br17, los resultados de todos los experimentos presentados en la tabla 4.6 arrojan un $Eb = 0\,\%$, o el valor de $Em$ para la instancia más grande (rbg443) varía entre un 10.35 % y un 12.14 %. En cuanto al número promedio de iteraciones necesarias para obtener la mejor solución, detectamos que, independientemente del parámetro analizado, en todos los casos se necesita un número mayor de iteraciones si el número de ciudades se incrementa. En otras palabras, si el tamaño del espacio de búsqueda crece entonces ACS incrementa su esfuerzo para hallar su mejor solución. Aunque este esfuerzo extra no alcance para mantener la calidad de las soluciones cercana al óptimo.

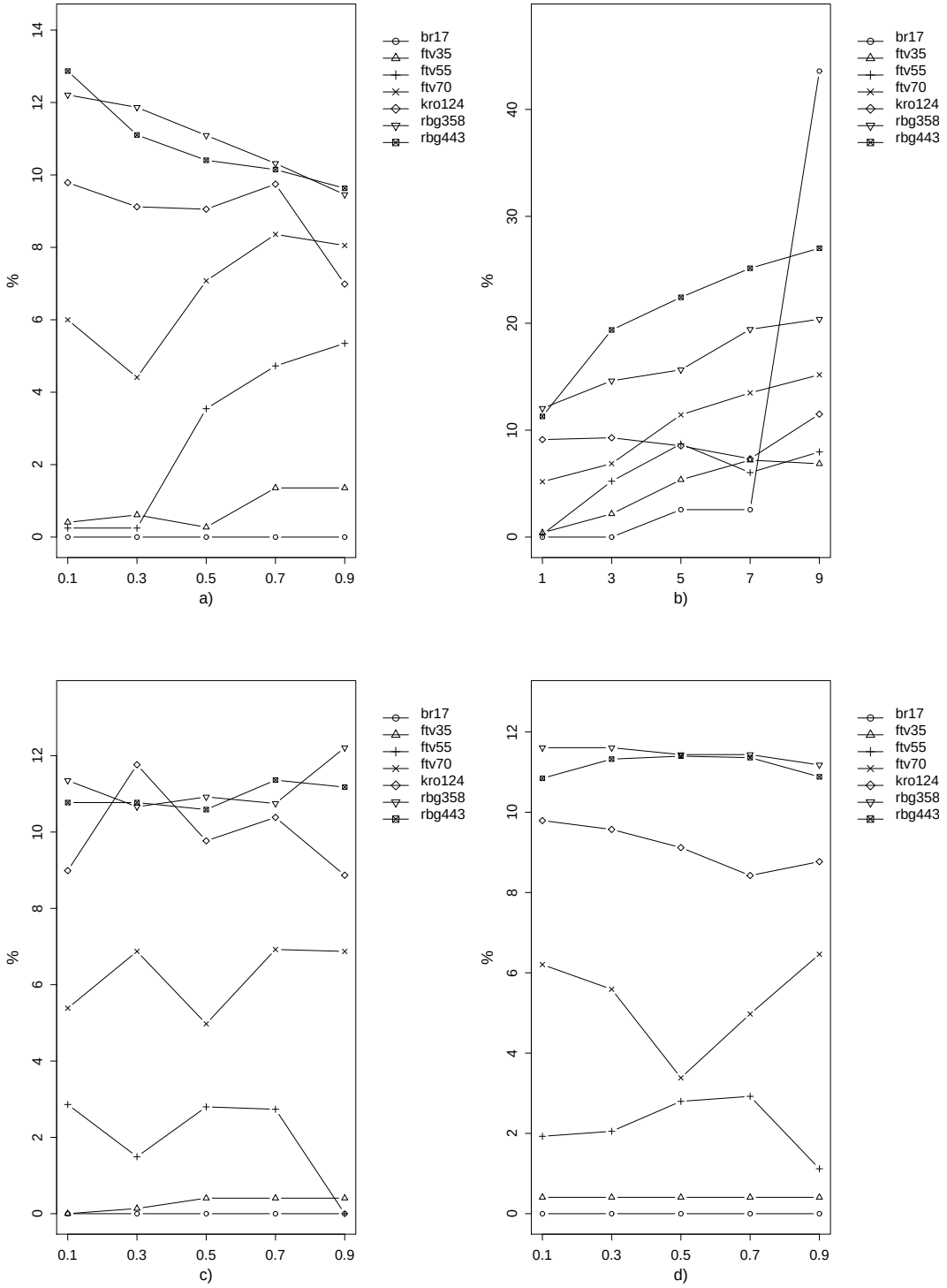


Figura 4.4: a) Valores de Eb considerando a $\rho$ b) Valores de Eb considerando a $\beta$ c) Valores de Eb considerando a $\varphi$ d) Valores de Eb considerando a $q_0$.

| Instancias | ACS | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $\rho$ | | | | $\beta$ | | | | $\varphi$ | | | | $q_0$ | | | |
| | Valor | Eb | Em | Ei | Valor | Eb | Em | Ei | Valor | Eb | Em | Ei | Valor | Eb | Em | Ei |
| br17 | [0.1,..,0.9] | 0.00 | 15.09 | 0 | [1,..,3] | 0.00 | 20.98 | 1 | [0.1,..,0.9] | 0.00 | 10.46 | 0 | [0.1,..,0.9] | 0.00 | 10.50 | 0 |
| ftv35 | [0.1,..,0.5] | 0.43 | 3.35 | 51 | {1} | 0.41 | 3.37 | 43 | {0.1,0.3,0.9} | 0.18 | 2.91 | 44 | [0.1,..,0.9] | 0.41 | 3.05 | 29 |
| ftv55 | [0.1,..,0.3] | 0.25 | 6.67 | 108 | {1} | 0.25 | 7.68 | 37 | [0.1,..,0.9] | 1.98 | 7.55 | 55 | [0.1,..,0.9] | 2.16 | 8.05 | 66 |
| ftv70 | [0.1,..,0.3] | 5.21 | 10.33 | 135 | [1,..,3] | 6.03 | 12.09 | 95 | [0.1,..,0.9] | 6.21 | 10.85 | 149 | [0.1,..,0.9] | 5.32 | 11.05 | 67 |
| kro124p | [0.3,..,0.9] | 8.73 | 13.73 | 269 | [3,..,5] | 8.91 | 12.19 | 85 | [0.1,..,0.9] | 9.95 | 13.44 | 417 | [0.1,..,0.9] | 9.14 | 13.38 | 416 |
| rbg358 | [0.5,..,0.9] | 10.29 | 12.26 | 338 | {1} | 12.04 | 13.13 | 597 | [0.1,..,0.9] | 11.18 | 12.97 | 753 | [0.1,..,0.9] | 11.45 | 12.99 | 734 |
| rbg443 | {0.9} | 9.63 | 10.35 | 115 | {1} | 11.29 | 12.02 | 748 | [0.1,..,0.9] | 10.93 | 12.02 | 781 | [0.1,..,0.9] | 11.16 | 12.14 | 843 |

Tabla 4.6: Valores paramétricos con los cuales ACS logra una mayor calidad en las soluciones de cada instancia y los valores promedio de Eb, Em y Ei correspondientes. (Eb y Em son errores porcentuales).

## 4.4. Análisis de resultados obtenidos por MMAS

En los siguientes parrafos, se analizo el comportamiento de MMAS bajo distintos valores paramétricos según se muestra en la tabla 4.2 y los valores Eb de la figura 4.5 y 4.6. Para ello estudiamos los resultados del test de varianza presentados en la tabla 4.7, considerando por separado cada parámetro:

**$\rho$.** Se mantuvieron constantes $\beta$ = 7 y $\tau_r$ = 1/100 para los experimentos con éste parámetro. A diferencia de ACS, este parámetro en MMAS sólo afecta la cantidad de feromana evaporada por el demonio en cada iteración. Se observa que éste no es un factor dominante en la búsqueda de soluciones de ATSP ya que soluciones de calidad similar son encontradas bajo todos los valores asignados a dicho parámetro.

**$\beta$.** Se mantuvieron constantes $\rho$ = 0.7 y $\tau_r$ = 1/100 para los experimentos con éste parámetro. Dado un $\alpha = 1$, en general es necesario que el grado de importancia de la información heurística sea superior al dado a la información sobre el rastro de feromona. Las instancias con 70 o menos ciudades encuentran soluciones cercanas al óptimo (Eb $< 1\,\%$) con la mayoría de los valores asignados a $\beta$. En cambio para las instancias de 124 y 358 ciudades, MMAS comete errores menores si $\beta \geqslant 5$. En tanto que para la instancia rbg443, MMAS con $\beta = 3$ obtiene los mejores resultados. Resumiendo, si bien se recomienda usar un valor de $\beta > 1$ para resolver cualquier instancia, MMAS es sensible a los valores que pueda tomar este parámetro cuando las instancias consideran 124 o más ciudades.

**$\tau_r$.** Se mantuvieron constantes $\rho$ = 0.7 y $\beta$ = 5 para los experimentos con éste parámetro. Cuando el número de ciudades varía entre 17 y 124, MMAS obtiene soluciones de similar calidad para todos los valores asignados a $\tau_r$. Pero cuando la cantidad de ciudades es mayor a 300, las mejores soluciones las halla cuando $\tau_r \geqslant \frac{1}{160}$. Por ende, se deduce que MMAS es sensible a este parámetro en las instancias de mayor tamaño.

Sintetizando el análisis previo, MMAS generalmente necesita que $\beta$ sea mayor a 1, pero también requiere adecuar los valors de $\beta$ y $\tau_r$, cuando resuelve instancias grandes. Por otra parte, en

función a las 3 medidas de desempeño mostradas en la tabla 4.7, se detectan dos grupos de instancias: uno de ellos formado por br17, ftv35, ftv55, ftv70 y kro124p, mientras que el segundo lo conforman las dos instancias más grandes (rgb358 y rgb443). En el primero MMAS comete errores menores al 4.65 % y en general el número promedio de iteraciones se incrementa cuando el número de ciudades crece. Pero, en el segundo grupo el desempeño de esta variante algorítmica disminuye significativamente al cometer errores mayores al 21 %; además, de no existir una relación entre la complejidad de las instancia y el esfuerzo realizado (Ei).

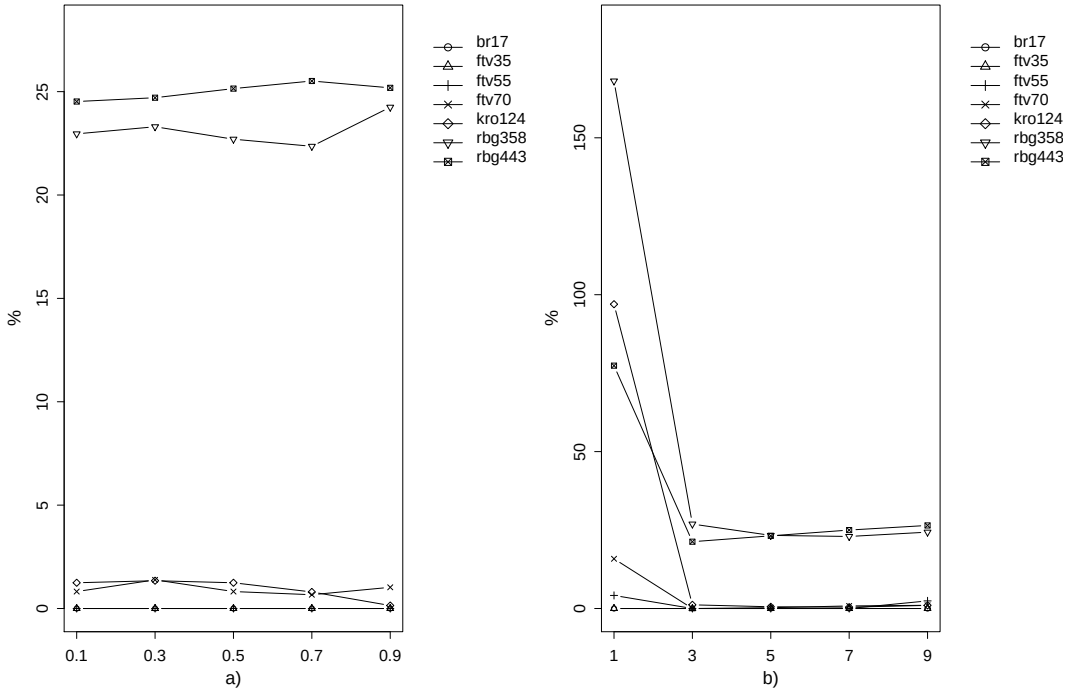


Figura 4.5: a) Valores de Eb considerando a $\rho$ b) Valores de Eb considerando a $\beta$.

| Instancias | MMAS | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $\rho$ | | | | $\beta$ | | | | $\tau_r$ | | | |
| | Valor | Eb | Em | Ei | Valor | Eb | Em | Ei | Valor | Eb | Em | Ei |
| br17 | [0.1,..,0.9] | 0.00 | 0.00 | 0 | [1,..,9] | 0.00 | 0.00 | 0 | $[\frac{1}{40},...,\frac{1}{200}]$ | 0.00 | 0.00 | 0 |
| ftv35 | [0.1,..,0.9] | 0.00 | 1.29 | 98 | [3,..,9] | 0.00 | 1.39 | 155 | $[\frac{1}{40},...,\frac{1}{200}]$ | 0.00 | 1.03 | 10 |
| ftv55 | [0.1,..,0.9] | 0.00 | 2.45 | 343 | [3,..,7] | 0.00 | 1.80 | 74 | $[\frac{1}{40},...,\frac{1}{200}$ | 0.00 | 1.28 | 147 |
| ftv70 | [0.1,..,0.9] | 0.94 | 4.63 | 320 | [3,..,9] | 0.53 | 4.33 | 205 | $[\frac{1}{40},...,\frac{1}{200}]$ | 0.05 | 3.45 | 596 |
| kro124p | [0.1,..,0.9] | 0.96 | 3.06 | 600 | [5,..,9] | 0.66 | 3.01 | 770 | $[\frac{1}{40},...,\frac{1}{200}]$ | 0.28 | 3.30 | 698 |
| rbg358 | [0.1,..,0.9] | 23.11 | 25.70 | 579 | [5,..,9] | 23.53 | 25.90 | 500 | $[\frac{1}{160},...,\frac{1}{200}]$ | 21.45 | 23.92 | 550 |
| rbg443 | [0.1,..,0.9] | 25.01 | 26.62 | 174 | {3} | 21.29 | 22.80 | 85 | $\{\frac{1}{200}\}$ | 22.94 | 24.07 | 914 |

Tabla 4.7: Valores paramétricos con los cuales MMAS logra una mayor calidad en las soluciones de cada instancia y los valores promedio de Eb, Em y Ei correspondientes. (Eb y Em son errores porcentuales).

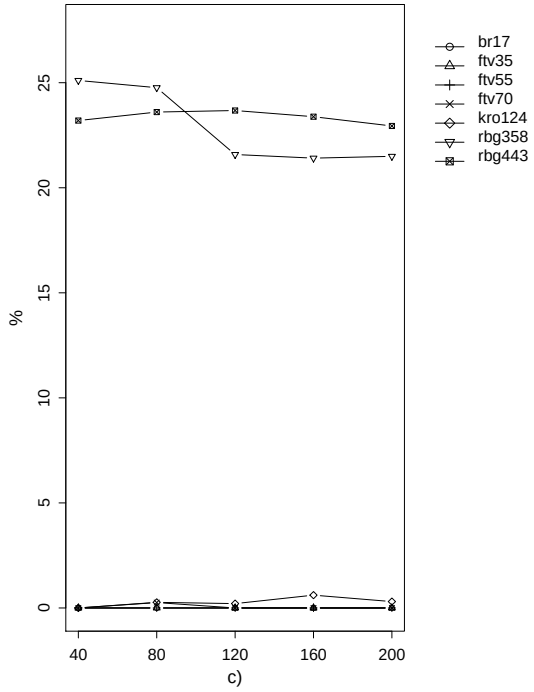


Figura 4.6: c) Valores de Eb considerando a $\tau_r$.

## 4.5. Análisis de resultados obtenidos por ASrank

En esta sección, se estudiara el comportamiento de ASrank bajo distintos valores paramétricos de la variante ASrank. Se comienza analizando los resultados del test estadístico presentado en la tabla 4.8 y los valores Eb de las figuras 4.7 y 4.8, contemplando cada parámetro por separado:

**$\rho$.** Se mantuvieron constantes $\beta$ = 1 y $\sigma$ = 1 para los experimentos con éste parámetro. Al igual que en MMAS este parámetro únicamente incide en la evaporación de feromona realizada por el demonio. Pero a diferencia de MMAS, el comportamiento de ASrank es sensible a la variación del mismo; ya que: en todas las instancias, la mayor calidad de los resultados se obtiene con $\rho$ =0.1. Esto indica que ASrank necesita un porcentaje bajo de evaporación para resolver el ATSP, independientemente del tamaño del caso de prueba.

**$\beta$.** Se mantuvieron constantes $\rho$ = 0.1 y $\sigma$ = 1 para los experimentos con éste parámetro. Como en MMAS, en general es necesario que $\beta$ sea mayor que $\alpha$. Para las instancias con 70 o menos ciudades, ASrank encuentra las soluciones de mayor calidad cuando $\beta \in [5, .., 9]$. En cambio, para la instancias de 124 ciudades comete errores menores si $\beta \geqslant 7$. En tanto que para las instancias rbg358 y rbg443, ASrank con $\beta = 3$ obtiene los mejores resultados.

Resumiendo, para instancias con 124 o menos ciudades se recomienda usar un valor de $\beta \geqslant 7$; para un número mayor de ciudades el valor más apropiado para este parámetro es 3.

**$\sigma$.** Se mantuvieron constantes $\rho = 0.1$ y $\beta = 7$ para los experimentos con éste parámetro. En general, este algoritmo encuentra sus mejores soluciones con $\sigma > 1$; pero no es posible determinar una relación entre este parámetro y el tamaño de las instancias.

Por lo expuesto arriba se infiere que ASrank es susceptible a cambios en cualquiera de estos tres parámetros. Particularmente para $\beta$ los valores dependen de la complejidad de la instancia. En cuanto a la calidad de las soluciones obtenidas por ASrank en la resolución de ATSP, se observan: errores altos (entre el 12 y 45 %) en las instancias chicas cuando únicamente se optimiza el valor de $\rho$. En cambio cuando más de un parámetro es optimizado, ASrank presenta un desempeño uniforme, cuyos errores son al 11 %, aún en las 2 instancias más grandes. En cuanto al esfuerzo computacional, en general, se detecto un incremento en la cantidad de iteraciones (Ei) en relación al aumento del número de ciudades; con la particularidad que para las instancias con 124 ciudades o menos Ei $< 80$. Es importante notar que para ACS y ASrank el límite superior de Ei en esas circunstancias es 417 y 770 respectivamente.

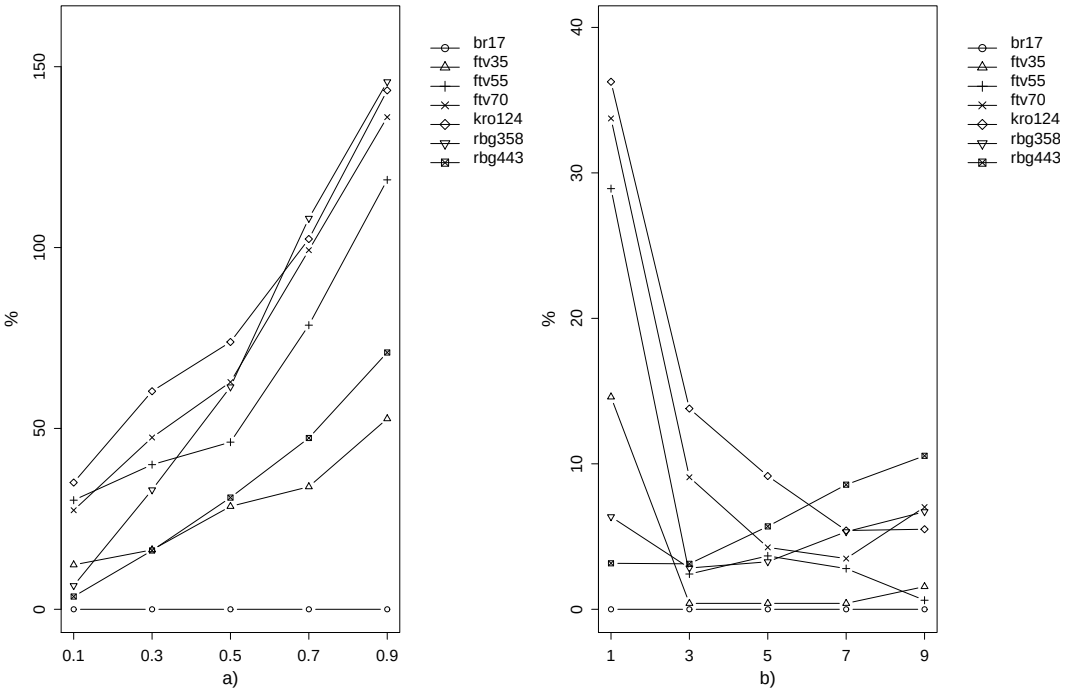


Figura 4.7: a) Valores de Eb considerando a $\rho$ b) Valores de Eb considerando a $\beta$ .

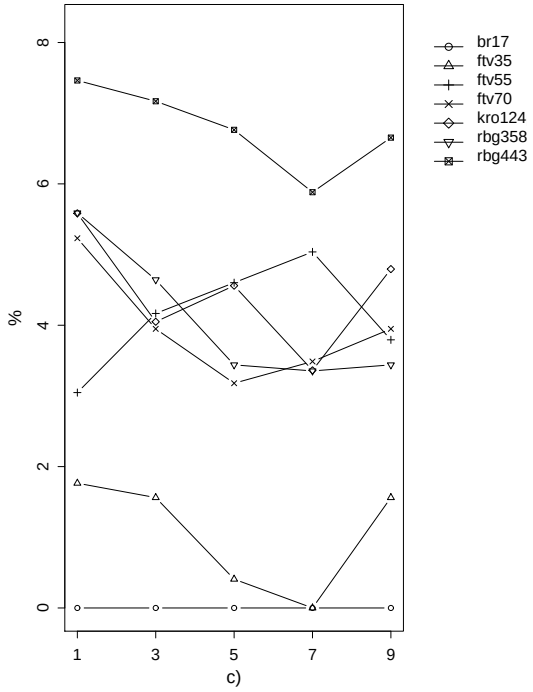


Figura 4.8: c) Valores de Eb considerando a $\sigma$.

| Instancias | ASrank | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $\rho$ | | | | $\beta$ | | | | $\sigma$ | | | |
| | Valor | Eb | Em | Ei | Valor | Eb | Em | Ei | Valor | Eb | Em | Ei |
| br17 | [0.1,..,0.9] | 0.00 | 1.59 | 0 | [1,..,9] | 0.00 | 2.05 | 0 | [3,..,9] | 0.00 | 0.99 | 0 |
| ftv35 | {0.1} | 12.29 | 24.64 | 31 | [5,..,9] | 0.79 | 3.88 | 7 | [5,..,9] | 0.66 | 2.13 | 10 |
| ftv55 | {0.1} | 30.16 | 39.41 | 51 | [5,..,9] | 2.36 | 7.50 | 21 | [1,..,9] | 4.13 | 6.61 | 9 |
| ftv70 | {0.1} | 27.38 | 43.89 | 67 | [5,..,9] | 4.92 | 10.81 | 35 | [3,..,9] | 3.64 | 8.23 | 50 |
| kro124p | {0.1} | 35.04 | 44.57 | 79 | [7,..,9] | 5.46 | 9.28 | 10 | [3,..,9] | 4.19 | 7.68 | 26 |
| rbg358 | {0.1} | 6.53 | 9.66 | 304 | {3} | 2.84 | 4.10 | 274 | [7,..,9] | 3.40 | 4.17 | 679 |
| rbg443 | {0.1} | 3.35 | 4.72 | 407 | {3} | 3.13 | 4.45 | 321 | [5,..,9] | 6.43 | 7.73 | 866 |

Tabla 4.8: Valores paramétricos con los cuales ASrank logra una mayor calidad en las soluciones de cada instancia y los valores promedio de Eb, Em y Ei correspondientes. (Eb y Em son errores porcentuales).

## 4.6. Comparación de AS, EAS, ACS, MMAS y ASrank

Finalmente, se comparan las cinco variantes de ACO usadas en este trabajo. Para lo cual se contrastan los resultados de las tablas 4.4, 4.5, 4.6, 4.7 y 4.8. Por un lado, la comparación se realiza con respecto a los parámetros que tienen en común: $\rho$ y $\beta$. MMAS es la única variante que no es susceptible a las variaciones de $\rho$. En cuanto a la importancia de la información heurística los

cinco algoritmos son sensibles a los cambios de $\beta$: AS, EAS y ACS aseguran que con $\beta = \alpha = 1$ obtienen las mejores soluciones en cualquiera de las instancias, MMAS y ASrank generalmente necesitan que $\beta$ sea mayor a 1 pero además, se recomienda configurar este parámetro según el tamaño de la instancia (ver secciones 4.4 y 4.5). Por otra parte, se analizan a los algoritmos desde el punto de vista de su desempeño medido por Eb, Em y Ei. Los algoritmos AS y EAS para instancias simples, es decir, menores a 70 ciudades, no tienen gran diferencia con los demás algoritmos y dada su sencillez son preferentes para estás instancias. Aunque MMAS obtiene mejores resultados para las instancias con ciudades menores a 300. Con respecto a la calidad detectamos que MMAS es la opción más apropiada para resolver instancias con 124 o menos ciudades, dado que los errores cometidos son menores a los de AS, EAS, ACS y ASrank (Eb $< 1\,\%$ y Em $< 5\,\%$) en todos estos casos. Pero, no sucede lo mismo con el esfuerzo computacional (Ei) requerido en esas instancias, ya que ASrank es él que menos iteraciones necesita para encontrar sus mejores soluciones. Sin embargo, al analizar las instancias con más de 300 ciudades observamos que ASrank obtiene los mejores resultados con el menor esfuerzo, lo sigue EAS (con el menor esfuerzo), ACS, AS y por último se encuentra MMAS.

Por último, como se muestra en la tabla 4.9, se tomaron los mejores valores para los distintos parámetros entre los diferentes algoritmos con las que se realizo la comparativa entre las distintas instancia del problema. Se tuvieron en cuenta los valores Eb, Em e Im de las figuras 4.9.

| Parámetro | AS | EAS | ACS | MMAS | ASrank |
|---|---|---|---|---|---|
| $\rho$ | {0.3} | {0.5} | {0.3} | {0.7} | {0.1} |
| $\beta$ | {1} | {1} | {1} | {5} | {7} |
| $e$ | | {9} | | | |
| $\varphi$ | | | {0.1} | | |
| $q_0$ | | | {0.5} | | |
| $\tau_r$ | | | | $\{\frac{1}{160}\}$ | |
| $\sigma$ | | | | | {7} |

Tabla 4.9: Parámetros utilizados para comparar los algoritmos.

El algoritmo MMAS obtiene los mejores resultados para las instancias inferiores o iguales a 100 ciudades. Mientras que ASrank realiza un menor esfuerzo computacional, con una diferencia del 5 % en Eb respecto al MMAS para las instancias mencionadas. Para instancias que superen las 100 ciudades el algoritmo que mejor se comporta respecto a la mejor solución es el ASrank, aunque MMAS supera al ASrank respecto a esfuerzo computacional para estas instancias pero con una diferencia entre 20 % y 25 % de Eb.

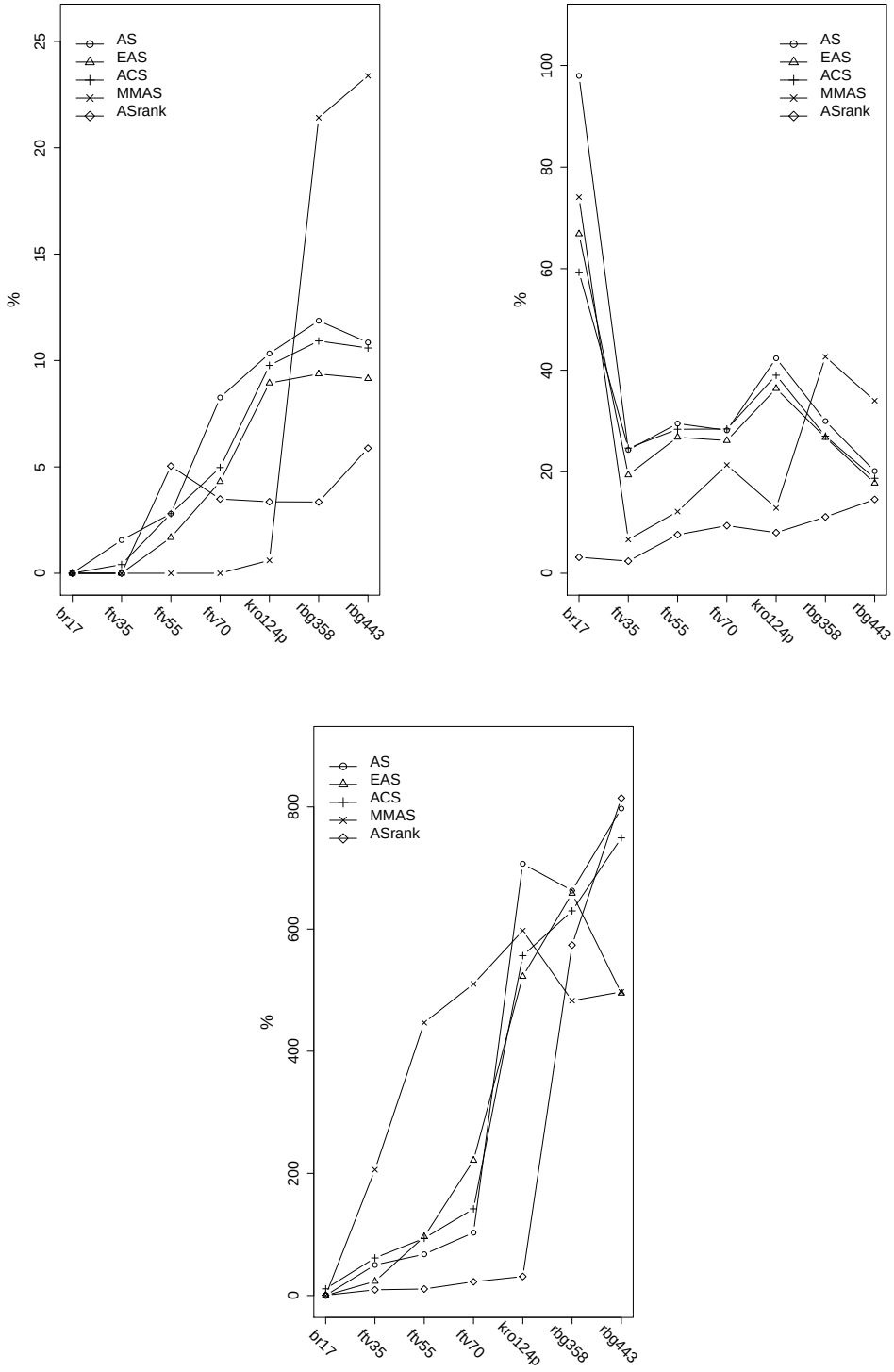


Figura 4.9: a) Valores de Eb b) Valores de Em c) Valores de Im .

# Capítulo 5

# Conclusiones

## 5.1. Conclusiones

En este trabajo se realizó un análisis paramétrico y comparativo de las variantes de AS más importantes: AS, EAS, ACS, MMAS y ASrank. Para esto, se han estudiado los parámetros que las caracterizan bajo diferentes configuraciones experimentando con un problema de optimización combinatoria NP-duro (ATSP), representativo de muchos problemas reales.

A partir de este estudio se puede observar que el comportamiento de:

- AS, EAS y ACS es sensible al grado de evaporación del rastro de feromona (parámetro $\rho$) y a la información del rastro de feromona (parámetro $\beta$), es decir que los valores de estos parámetros alteran el desempeño de estos 3 algoritmos.
- ACS no es sensible respecto de la evaporación online del rastro de feromona (parámetro $\varphi$) y el balance sobre exploración y explotación (parámetro $q_0$), ya que modificándolos no se obtienen grandes cambios en su desempeño. Necesita mayor evaporación del rastro de feromona (parámetro $\rho$) en las instancias más complejas, es decir las instancias donde la cantidad de ciudades es superior o igual a 100. Por otra parte, se requiere que la importancia de la información heurística (parámetro $\eta$) y de la información sobre el rastro de feromona (parámetro $\tau$) sean iguales en todas las instancias, lo cual significa que $\alpha$ y $\beta$ deben ser iguales (generalmente el valor 1).
- MMAS es susceptible al parámetro $\eta$. Por lo tanto, se recomienda dar siempre, más importancia a la información heurística (parámetro $\eta$) que a la información sobre el rastro de feromona (parámetro $\tau_r$), aunque es conveniente adecuar ambas variables en las instancias más grandes, generalmente superiores a 100 ciudades.
- ASrank no es indiferente a cambios en $\rho$, $\beta$ y $\sigma$. Se busca un balance equitativo entre la exploración (parámetro $\sigma$ mayor a 1) y la explotación (parámetro $\rho$ con valor pequeño) del espacio de búsqueda, y se recomienda que la importancia de la información heurística sea superior al del rastro de feromona (parámetro $\beta$ mayor a 1).

Como trabajo futuro se prevee utilizar los resultados obtenidos aquí en la aplicación de un caso real, como es el problema de localización de antenas de radio.

# Apéndice A

# Implementación

En esta sección se adjunta el código fuente de las distintas implementaciones de los algoritmos ACO.

## A.1. Código fuente - poblacion.h

```
#include "time.hh"
#import "solucion.h"

class Poblacion
{
    public:
        Poblacion();
        ~Poblacion();

        float comienzo;
        Solucion *soluciones;
        unsigned int poblacion;
        double mejor_costo;
        double peor_costo;
        Solucion solucion_mc;
        Solucion *current_best;
        double iteracion_mc;
        float tiempo_mc;

        unsigned int inicializar(unsigned int p, unsigned int d);
        unsigned int tam();
        void asignar(unsigned int p, unsigned int d, unsigned int el);
        unsigned int elemento(unsigned int p, unsigned int d);
        Solucion *solucion(unsigned int p);
```

```
25          void calculos(unsigned int ciclo, unsigned int p, double costo)
               ;
26          void imprimir(unsigned int p);
27          void reset();
28          void actualizar_mejor_solucion(unsigned int ciclo);
29  };
30
31  Poblacion::Poblacion()
32  {
33      soluciones = NULL;
34      current_best = NULL;
35      poblacion = 0;
36  }
37
38  Poblacion::~Poblacion()
39  {
40      free(soluciones);
41  }
42
43  unsigned int Poblacion::inicializar(unsigned int p, unsigned int d)
44  {
45      unsigned int i;
46      poblacion = p;
47      soluciones = (Solucion *)malloc(poblacion * sizeof(Solucion));
48      for(i=0;i<poblacion;i++){
49          soluciones[i] = Solucion();
50          soluciones[i].inicializar(d);
51      }
52      solucion_mc = Solucion();
53      solucion_mc.inicializar(d);
54  }
55
56  unsigned int Poblacion::tam()
57  {
58      return poblacion;
59  }
60
61  void Poblacion::asignar(unsigned int p, unsigned int d, unsigned int el
       )
62  {
63      soluciones[p].asignar(d, el);
64  }
65
66  unsigned int Poblacion::elemento(unsigned int p, unsigned int d)
67  {
```

```
68      return soluciones[p].elemento(d);
69  }
70
71  Solucion *Poblacion::solucion(unsigned int p)
72  {
73      return &soluciones[p];
74  }
75
76  void Poblacion::imprimir(unsigned int p)
77  {
78      soluciones[p].imprimir();
79  }
80
81  void Poblacion::calculos(unsigned int ciclo, unsigned int p, double
        costo)
82  {
83      soluciones[p].set_costo(costo);
84      if(current_best==NULL || soluciones[p].costo<(*current_best).costo)
85          current_best = &(soluciones[p]);
86      if(peor_costo<0 || costo>peor_costo){
87          peor_costo = costo;
88      }
89  }
90
91  void Poblacion::actualizar_mejor_solucion(unsigned int ciclo){
92      unsigned int i;
93      if(mejor_costo<0 || (*current_best).costo<mejor_costo){
94          mejor_costo = (*current_best).costo;
95          iteracion_mc = ciclo;
96          tiempo_mc = _used_time(comienzo)/1000000.0;
97          for(i=0;i<(*current_best).dimension;i++){
98              solucion_mc.asignar(i,(*current_best).elemento(i));
99              solucion_mc.set_costo((*current_best).costo);
100         }
101     }
102 }
103
104 void Poblacion::reset()
105 {
106     comienzo = _used_time();
107     mejor_costo = -1;
108     peor_costo = -1;
109 }
```

## A.2. Código fuente - lista_ordenada.h

```
#include <stdio.h>
#include "solucion.h"

using namespace std;

class SolucionEnlazada
{
    public:
        Solucion *solucion;
        SolucionEnlazada *siguiente;

        SolucionEnlazada();
};

SolucionEnlazada::SolucionEnlazada()
{
    solucion = NULL;
    siguiente = NULL;
}

class ListaOrdenada
{
    public:
        ListaOrdenada();
        ~ListaOrdenada();

        SolucionEnlazada *primero;
        unsigned int dimension;
        void agregar(Solucion *sol);
        void reset();
        void imprimir();
};

ListaOrdenada::ListaOrdenada()
{
    primero = NULL;
    dimension = 0;
}

ListaOrdenada::~ListaOrdenada(){
    reset();
}
```

```
43
44  void ListaOrdenada::reset()
45  {
46      if(primero!=NULL){
47          SolucionEnlazada *anterior = primero;
48          while(anterior!=NULL){
49              SolucionEnlazada *siguiente = (*anterior).siguiente;
50              free(anterior);
51              anterior = siguiente;
52          }
53      }
54      primero = NULL;
55      dimension = 0;
56  }
57
58  void ListaOrdenada::agregar(Solucion *sol)
59  {
60      if(primero==NULL){
61          primero = (SolucionEnlazada *)malloc(sizeof(SolucionEnlazada));
62          (*primero) = SolucionEnlazada();
63          (*primero).solucion = sol;
64          (*primero).siguiente = NULL;
65      }else{
66          SolucionEnlazada *agregar = (SolucionEnlazada *)malloc(sizeof(
               SolucionEnlazada));
67          (*agregar) = SolucionEnlazada();
68          (*agregar).solucion = sol;
69          SolucionEnlazada *anterior = NULL;
70          SolucionEnlazada *siguiente = primero;
71          while(siguiente!=NULL && (*((*siguiente).solucion)).costo<(*sol
               ).costo){
72              anterior = siguiente;
73              siguiente = (*siguiente).siguiente;
74          }
75          (*agregar).siguiente = siguiente;
76          if(anterior!=NULL)
77              (*anterior).siguiente = agregar;
78          else
79              primero = agregar;
80      }
81      dimension++;
82  }
83
84  void ListaOrdenada::imprimir()
85  {
```

```
86      SolucionEnlazada *actual = primero;
87      while(actual!=NULL){
88          cout << (*((*actual).solucion)).costo << endl;
89          actual = (*actual).siguiente;
90      }
91      cout << endl;
92  }
```

## A.3. Código fuente - aco.h

```
1   #include <stdio.h>
2   #include <iostream>
3   #include <fstream>
4   #include <fstream.h>
5   #include <sstream>
6   #include <strings.h>
7   #include <math.h>
8   #include "time.hh"
9   #include "vecinos.h"
10  #include "solucion.h"
11  #include "poblacion.h"
12  #include "lista_ordenada.h"
13
14  using namespace std;
15
16  #ifndef MAX_BUFFER
17  #define MAX_BUFFER 200
18  #endif
19
20  inline double random_double(double _min, double _max) // [_min,_max)
21  {
22      return _min + (_max-_min) * drand48();
23  }
24
25  inline int random_int   (int _min, int _max)  // [_min,_max)
26  {
27      return (int) (_min + (_max-_min) * drand48());
28  }
29
30  class Problema
31  {
32      public:
33          Problema();
```

```
34          ~Problema();
35
36          unsigned int ejecuciones;
37          unsigned int ciclos;
38          unsigned int poblacion;
39          unsigned int alpha;
40          unsigned int beta;
41          double deposito;
42          double rho;
43          double costo_promedio;
44          unsigned int elitistas;
45          double q0;
46          double varphi;
47          double max_min;
48          unsigned int rank;
49
50          unsigned int ejecucion, ciclo;
51
52          unsigned int dimension;
53          double **costo;
54          double **eta;
55          double **tau;
56          Poblacion sociedad;
57          Vecino vecino;
58          double costo_promedio_cargado;
59          double tau0;
60          double t_max;
61          double t_min;
62          ListaOrdenada lista;
63
64          double *b;
65
66          void cargar_costos(char* nombre_archivo);
67          void cargar_configuracion(char* nombre_archivo);
68
69          void calcular_eta();
70          void calcular_tau();
71
72          void inicializar();
73          void avanzar();
74          void demonio();
75          void generar_solucion(unsigned int c);
76          void correr();
77          void reset();
78          void evaporar();
```

```
79          double costo_solucion(unsigned int *sol);
80          void actualizacion_retrasada(unsigned int s);
81  };
82
83  Problema::Problema()
84  {
85      srand48( time(NULL) );
86
87      costo = NULL;
88      eta = NULL;
89      tau = NULL;
90      dimension = 0;
91      vecino = Vecino();
92  }
93
94  Problema::~Problema()
95  {
96      unsigned int i;
97
98      free(b);
99
100     for(i=0;i<dimension;i++)
101         free(costo[i]);
102     free(costo);
103
104     for(i=0;i<dimension;i++)
105         free(eta[i]);
106     free(eta);
107
108     for(i=0;i<dimension;i++)
109         free(tau[i]);
110     free(tau);
111 }
112
113 double Problema::costo_solucion(unsigned int *sol)
114 {
115     unsigned int i, j;
116     double sum_costo = 0;
117     for(i=1;i<=dimension;i++){
118         sum_costo+=costo[sol[i-1]][sol[i]];
119     }
120     return sum_costo;
121 }
122
123 void Problema::calcular_eta()
```

```
124 {
125     unsigned int i, j;
126     for(i=0;i<dimension;i++)
127     {
128         for(j=0;j<dimension;j++)
129         {
130             if(costo[i][j] == 0)
131                 eta[i][j] = 1.0;
132             else
133                 eta[i][j] = 1.0/costo[i][j];
134         }
135     }
136 }
137
138 void Problema::calcular_tau()
139 {
140     unsigned int i, j;
141     for(i=0;i<dimension;i++)
142     {
143         for(j=0;j<dimension;j++)
144         {
145             tau[i][j] = 1.0 / (poblacion*costo_promedio_cargado);
146         }
147     }
148 }
149
150 void Problema::cargar_configuracion(char* nombre_archivo)
151 {
152     char salida[MAX_BUFFER];
153     char comando[50];
154     unsigned int i, j;
155
156     ifstream filestr ( nombre_archivo, ifstream::in );
157
158     filestr.getline(salida, MAX_BUFFER, '\n');
159     filestr.getline(salida, MAX_BUFFER, '\n');
160     sscanf(salida,"%d", &ejecuciones);
161     filestr.getline(salida, MAX_BUFFER, '\n');
162     sscanf(salida,"%d", &ciclos);
163     filestr.getline(salida, MAX_BUFFER, '\n');
164     sscanf(salida,"%d", &poblacion);
165     filestr.getline(salida, MAX_BUFFER, '\n');
166     sscanf(salida,"%d", &alpha);
167     filestr.getline(salida, MAX_BUFFER, '\n');
168     sscanf(salida,"%d", &beta);
```

```
169     filestr.getline(salida, MAX_BUFFER, '\n');
170     sscanf(salida,"%lf", &deposito);
171     filestr.getline(salida, MAX_BUFFER, '\n');
172     sscanf(salida,"%lf", &rho);
173     filestr.getline(salida, MAX_BUFFER, '\n');
174     sscanf(salida,"%d", &elitistas);
175     filestr.getline(salida, MAX_BUFFER, '\n');
176     sscanf(salida,"%lf", &q0);
177     filestr.getline(salida, MAX_BUFFER, '\n');
178     sscanf(salida,"%lf", &varphi);
179     filestr.getline(salida, MAX_BUFFER, '\n');
180     sscanf(salida,"%lf", &max_min);
181     filestr.getline(salida, MAX_BUFFER, '\n');
182     sscanf(salida,"%d", &rank);
183
184     filestr.close();
185 }
186
187
188 void Problema::cargar_costos(char* nombre_archivo)
189 {
190     char salida[MAX_BUFFER];
191     char comando[50];
192     unsigned int i, j;
193
194     ifstream filestr ( nombre_archivo, ifstream::in );
195
196     for(i=0;i<4;i++) filestr.getline(salida, MAX_BUFFER, '\n');
197     sscanf(salida,"DIMENSION: %d", &dimension);
198
199     for(i=0;i<3;i++) filestr.getline(salida, MAX_BUFFER, '\n');
200
201     costo = (double **)malloc (dimension * sizeof(double *));
202     eta = (double **)malloc (dimension * sizeof(double *));
203     tau = (double **)malloc (dimension * sizeof(double *));
204
205     costo_promedio_cargado = 0;
206     for(i=0;i<dimension;i++)
207     {
208         costo[i] = (double *)malloc (dimension * sizeof(double));
209         eta[i] = (double *)malloc (dimension * sizeof(double));
210         tau[i] = (double *)malloc (dimension * sizeof(double));
211         for(j=0;j<dimension;j++)
212         {
213             filestr >> costo[i][j];
```

```
214                 if(i!=j)
215                     costo_promedio_cargado += costo[i][j];
216             }
217         }
218         filestr.close();
219         costo_promedio_cargado = costo_promedio_cargado/(dimension*
               dimension-dimension);
220 
221     }
222 
223     void Problema::inicializar(){
224         b = new double[dimension];
225         calcular_eta();
226         sociedad.inicializar(poblacion, (dimension+1));
227         vecino.iniciar(dimension-1);
228     }
229 
230     void Problema::reset(){
231         calcular_tau();
232         sociedad.reset();
233         costo_promedio = 0;
234     }
235 
236     void Problema::avanzar(){
237         unsigned int i;
238         for(i=0;i<poblacion;i++){
239             generar_solucion(i);
240             actualizacion_retrasada(i);
241         }
242     }
243 
244     void Problema::actualizacion_retrasada(unsigned int s)
245     {
246         unsigned int i;
247         double Delta = 1.0 / ((*sociedad.solucion(s)).get_costo()+1.0);
248         for(i=1;i<=dimension;i++)
249         {
250             tau[sociedad.elemento(s, i-1)][sociedad.elemento(s,i)]+=
                   deposito*Delta;
251         }
252     }
253 
254 
255     void Problema::demonio(){
256         evaporar();
```

```
257 }
258
259 void Problema::evaporar(){
260     unsigned int i, j;
261     for(i=0;i<dimension;i++)
262     {
263         for(j=0;j<dimension;j++){
264             tau[i][j] = tau[i][j] * (1.0-rho);
265         }
266     }
267 }
268
269
270 void Problema::generar_solucion(unsigned int c){
271     unsigned int cant = 0;
272     unsigned int s;
273     unsigned int i;
274     double *b;
275
276     vecino.reset();
277
278     sociedad.asignar(c, 0, 0);
279
280     while (vecino.tam()>0){
281         double sum = 0;
282         unsigned int s,i;
283         unsigned int anterior = sociedad.elemento(c, cant);
284         b = new double[vecino.tam()];
285         for (i=0;i<vecino.tam();i++)
286         {
287             double x= pow(tau[anterior][vecino.elemento(i)],alpha)*
288                       pow(eta[anterior][vecino.elemento(i)],beta);
289             sum = sum + x;
290             b[i] = sum;
291         }
292         double R = random_double(0, sum);
293         i=0;
294         while(i<vecino.tam()&&b[i]<R){
295             i++;
296         }
297         s = i;
298         cant++;
299         sociedad.asignar(c, cant, vecino.elemento(s));
300         vecino.eliminar(s);
301         free(b);
```

```
302     }
303     cant++;
304     sociedad.asignar(c, cant, 0);
305     sociedad.calculos(ciclo, c, costo_solucion((*(sociedad.solucion(c))
            ).soluciones));
306     costo_promedio += sociedad.soluciones[c].costo;
307 }
308
309 void Problema::correr(){
310     inicializar();
311     ejecucion = 0;
312     while(ejecucion<ejecuciones){
313         reset();
314         ciclo = 0;
315         while(ciclo<ciclos){
316             avanzar();
317             demonio();
318             ciclo++;
319         }
320         cout << sociedad.mejor_costo << ";";
321         cout << sociedad.peor_costo << ";";
322         cout << costo_promedio/(poblacion*ciclos) << ";";
323         cout << sociedad.iteracion_mc << ";";
324         cout << sociedad.tiempo_mc << ";";
325         cout << _used_time(sociedad.comienzo)/1000000.0 << ";";
326         sociedad.solucion_mc.imprimir();
327         cout << endl;
328
329         ejecucion++;
330     }
331 }
```

## A.4. Código fuente - as.cc

```
1 #include "aco.h"
2
3 using namespace std;
4
5 class Problema_AS: public Problema
6 {
7     public:
8         Problema_AS();
9         ~Problema_AS();
```

```
10
11
12          void avanzar();
13          void demonio();
14          void generar_solucion(unsigned int c);
15          void correr();
16          void reset();
17          void evaporar();
18          void actualizacion_retrasada(unsigned int s);
19  };
20
21  Problema_AS::Problema_AS()
22  {
23  }
24
25  Problema_AS::~Problema_AS()
26  {
27  }
28
29  void Problema_AS::avanzar(){
30      unsigned int i;
31      for(i=0;i<poblacion;i++){
32          generar_solucion(i);
33          actualizacion_retrasada(i);
34      }
35  }
36
37  void Problema_AS::actualizacion_retrasada(unsigned int s)
38  {
39      unsigned int i;
40      double Delta = 1.0 / ((*sociedad.solucion(s)).get_costo()+1.0);
41      for(i=1;i<=dimension;i++)
42      {
43          tau[sociedad.elemento(s, i-1)][sociedad.elemento(s,i)]+=
               deposito*Delta;
44      }
45  }
46
47
48  void Problema_AS::demonio(){
49      sociedad.actualizar_mejor_solucion(ciclo);
50      evaporar();
51  }
52
53  void Problema_AS::evaporar(){
```

```
54      unsigned int i, j;
55      for(i=0;i<dimension;i++)
56      {
57          for(j=0;j<dimension;j++){
58              tau[i][j] = tau[i][j] * (1.0-rho);
59          }
60      }
61  }
62
63  void Problema_AS::generar_solucion(unsigned int c){
64      unsigned int cant = 0;
65      unsigned int s;
66      unsigned int i;
67
68      vecino.reset();
69
70      sociedad.asignar(c, 0, 0);
71
72      while (vecino.tam()>0){
73          double sum = 0;
74          unsigned int s,i;
75          unsigned int anterior = sociedad.elemento(c, cant);
76          for (i=0;i<vecino.tam();i++)
77          {
78              double x= pow(tau[anterior][vecino.elemento(i)],alpha)*
79                        pow(eta[anterior][vecino.elemento(i)],beta);
80              sum = sum + x;
81              b[i] = sum;
82          }
83          double R = random_double(0, sum);
84          i=0;
85          while(i<vecino.tam()&&b[i]<R){
86              i++;
87          }
88          s = i;
89          cant++;
90          sociedad.asignar(c, cant, vecino.elemento(s));
91          vecino.eliminar(s);
92          free(b);
93      }
94      cant++;
95      sociedad.asignar(c, cant, 0);
96      sociedad.calculos(ciclo, c, costo_solucion((*(sociedad.solucion(c))
           ).soluciones));
97      costo_promedio += sociedad.soluciones[c].costo;
```

```
98  }
99
100 void Problema_AS::reset(){
101     calcular_tau();
102     sociedad.reset();
103     costo_promedio = 0;
104 }
105
106 void Problema_AS::correr(){
107     inicializar();
108     ejecucion = 0;
109     while(ejecucion<ejecuciones){
110         reset();
111         ciclo = 0;
112         while(ciclo<ciclos){
113             sociedad.current_best = NULL;
114             avanzar();
115             demonio();
116             ciclo++;
117         }
118         cout << sociedad.mejor_costo << ";";
119         cout << sociedad.peor_costo << ";";
120         cout << costo_promedio/(poblacion*ciclos) << ";";
121         cout << sociedad.iteracion_mc << ";";
122         cout << sociedad.tiempo_mc << ";";
123         cout << _used_time(sociedad.comienzo)/1000000.0 << ";";
124         sociedad.solucion_mc.imprimir();
125         cout << endl;
126
127         ejecucion++;
128     }
129 }
130
131 int main(int argc, char** argv)
132 {
133     Problema_AS p = Problema_AS();
134     p.cargar_costos(argv[2]);
135     p.cargar_configuracion(argv[1]);
136     p.correr();
137
138     return(0);
139 }
```

## A.5. Código fuente - eas.cc

```
#include "aco.h"

class Problema_EAS: public Problema
{
    public:
        Problema_EAS();
        ~Problema_EAS();

        void inicializar();
        void reset();
        void avanzar();
        void demonio();
        void generar_solucion(unsigned int c);
        void correr();
        void evaporar();
        void actualizacion_retrasada(unsigned int s);
        void deposito_elitista();
};

Problema_EAS::Problema_EAS()
{
}

Problema_EAS::~Problema_EAS()
{
}

void Problema_EAS::inicializar(){
    calcular_eta();
    sociedad.inicializar(poblacion, (dimension+1));
    vecino.iniciar(dimension-1);
}

void Problema_EAS::reset(){
    calcular_tau();
    sociedad.reset();
    costo_promedio = 0;
}

void Problema_EAS::avanzar(){
    unsigned int i;
    for(i=0;i<poblacion;i++){
```

```
43          generar_solucion(i);
44          actualizacion_retrasada(i);
45      }
46  }
47
48  void Problema_EAS::actualizacion_retrasada(unsigned int s)
49  {
50      unsigned int i;
51      double Delta = 1.0 / ((*sociedad.solucion(s)).get_costo()+1.0);
52      for(i=1;i<=dimension;i++)
53      {
54          tau[sociedad.elemento(s, i-1)][sociedad.elemento(s,i)]+=
                deposito*Delta;
55      }
56  }
57
58
59  void Problema_EAS::demonio(){
60      sociedad.actualizar_mejor_solucion(ciclo);
61      evaporar();
62      deposito_elitista();
63  }
64
65  void Problema_EAS::deposito_elitista()
66  {
67      unsigned int i, origen, destino;
68      double Delta = 1.0 / (sociedad.solucion_mc.get_costo()+1.0);
69      for(i=1;i<=dimension;i++){
70          tau[sociedad.solucion_mc.elemento(i-1)][sociedad.solucion_mc.
                elemento(i)]+=elitistas*deposito*Delta;
71      }
72  }
73
74  void Problema_EAS::evaporar(){
75      unsigned int i, j;
76      for(i=0;i<dimension;i++)
77      {
78          for(j=0;j<dimension;j++){
79              tau[i][j] = tau[i][j] * (1.0-rho);
80          }
81      }
82  }
83
84
85  void Problema_EAS::generar_solucion(unsigned int c){
```

```
86      unsigned int cant = 0;
87      unsigned int s;
88      unsigned int i;
89      double *b;
90
91      vecino.reset();
92
93      sociedad.asignar(c, 0, 0);
94
95      while (vecino.tam()>0){
96          double sum = 0;
97          unsigned int s,i;
98          unsigned int anterior = sociedad.elemento(c, cant);
99          b = new double[vecino.tam()];
100         for (i=0;i<vecino.tam();i++)
101         {
102             double x= pow(tau[anterior][vecino.elemento(i)],alpha)*
103                       pow(eta[anterior][vecino.elemento(i)],beta);
104             sum = sum + x;
105             b[i] = sum;
106         }
107         double R = random_double(0, sum);
108         i=0;
109         while(i<vecino.tam()&&b[i]<R){
110             i++;
111         }
112         s = i;
113         cant++;
114         sociedad.asignar(c, cant, vecino.elemento(s));
115         vecino.eliminar(s);
116         free(b);
117     }
118     cant++;
119     sociedad.asignar(c, cant, 0);
120     sociedad.calculos(ciclo, c, costo_solucion((*(sociedad.solucion(c))
            ).soluciones));
121     costo_promedio += sociedad.soluciones[c].costo;
122 }
123
124 void Problema_EAS::correr(){
125     inicializar();
126     ejecucion = 0;
127     while(ejecucion<ejecuciones){
128         reset();
129         ciclo = 0;
```

```
130         while(ciclo<ciclos){
131             sociedad.current_best = NULL;
132             avanzar();
133             demonio();
134             ciclo++;
135         }
136         cout << sociedad.mejor_costo << ";";
137         cout << sociedad.peor_costo << ";";
138         cout << costo_promedio/(poblacion*ciclos) << ";";
139         cout << sociedad.iteracion_mc << ";";
140         cout << sociedad.tiempo_mc << ";";
141         cout << _used_time(sociedad.comienzo)/1000000.0 << ";";
142         sociedad.solucion_mc.imprimir();
143         cout << endl;
144 
145         ejecucion++;
146     }
147 }
148 
149 int main(int argc, char** argv)
150 {
151     Problema_EAS p = Problema_EAS();
152     p.cargar_costos(argv[2]);
153     p.cargar_configuracion(argv[1]);
154     p.correr();
155 
156     return(0);
157 }
```

## A.6. Código fuente - acs.cc

```
1 
2 #include "aco.h"
3 
4 class Problema_ACS: public Problema
5 {
6     public:
7         Problema_ACS();
8         ~Problema_ACS();
9 
10         void inicializar();
11         void reset();
12         void avanzar();
```

```
13          void demonio();
14          void generar_solucion(unsigned int c);
15          void correr();
16          void deposito_mejor_global();
17          void actualizacion_online_paso_a_paso(unsigned int origen,
               unsigned int destino);
18  };
19
20  Problema_ACS::Problema_ACS()
21  {
22  }
23
24  Problema_ACS::~Problema_ACS()
25  {
26  }
27
28  void Problema_ACS::inicializar(){
29      calcular_eta();
30      sociedad.inicializar(poblacion, (dimension+1));
31      vecino.iniciar(dimension-1);
32  }
33
34  void Problema_ACS::reset(){
35      calcular_tau();
36      sociedad.reset();
37      costo_promedio = 0;
38  }
39
40  void Problema_ACS::avanzar(){
41      unsigned int i;
42      for(i=0;i<poblacion;i++){
43          generar_solucion(i);
44      }
45  }
46
47  void Problema_ACS::demonio(){
48      sociedad.actualizar_mejor_solucion(ciclo);
49      deposito_mejor_global();
50  }
51
52  void Problema_ACS::deposito_mejor_global()
53  {
54      unsigned int i, origen, destino;
55      double Delta = 1.0 / (sociedad.solucion_mc.get_costo()+1.0);
56      for(i=1;i<=dimension;i++){
```

```
57            tau[sociedad.solucion_mc.elemento(i-1)][sociedad.solucion_mc.
                  elemento(i)]=(1.0-rho)*tau[sociedad.solucion_mc.elemento(i
                  -1)][sociedad.solucion_mc.elemento(i)]+rho*deposito*Delta;
58      }
59  }
60
61  void Problema_ACS::actualizacion_online_paso_a_paso(unsigned int origen
        , unsigned int destino)
62  {
63      tau[origen][destino]=(1.0 - varphi) * tau[origen][destino] + varphi
            * deposito * tau0;
64  }
65
66  void Problema_ACS::generar_solucion(unsigned int c){
67      unsigned int cant = 0;
68      unsigned int s;
69      unsigned int i;
70      double *b;
71
72      vecino.reset();
73
74      sociedad.asignar(c, 0, 0);
75
76      while (vecino.tam()>0){
77          double sum = 0;
78          unsigned int s,i;
79          unsigned int anterior = sociedad.elemento(c, cant);
80          b = new double[vecino.tam()];
81          double q = random_double(0, 1);
82          if(q<=q0){
83              for (i=0;i<vecino.tam();i++)
84              {
85                  double x= pow(tau[anterior][vecino.elemento(i)],alpha)*
86                            pow(eta[anterior][vecino.elemento(i)],beta);
87                  b[i] = x;
88              }
89              s=0;
90              for(i=1;i<vecino.tam();i++){
91                  if(b[i]>b[s])
92                      s = i;
93              }
94          }else{
95              for (i=0;i<vecino.tam();i++)
96              {
97                  double x= pow(tau[anterior][vecino.elemento(i)],alpha)*
```

```
98                              pow(eta[anterior][vecino.elemento(i)],beta);
99                  sum = sum + x;
100                 b[i] = sum;
101             }
102
103             double R = random_double(0, sum);
104             i=0;
105             while(i<vecino.tam()&&b[i]<R){
106                 i++;
107             }
108             s = i;
109         }
110         cant++;
111         sociedad.asignar(c, cant, vecino.elemento(s));
112         actualizacion_online_paso_a_paso(anterior, vecino.elemento(s));
113         vecino.eliminar(s);
114         free(b);
115     }
116     cant++;
117     sociedad.asignar(c, cant, 0);
118     actualizacion_online_paso_a_paso(sociedad.elemento(c, cant-1),
          sociedad.elemento(c, cant));
119     sociedad.calculos(ciclo, c, costo_solucion((*(sociedad.solucion(c))
          ).soluciones));
120     costo_promedio += sociedad.soluciones[c].costo;
121 }
122
123 void Problema_ACS::correr(){
124     inicializar();
125     reset();
126     avanzar();
127     demonio();
128     tau0 = 1.0/(poblacion*(sociedad.peor_costo+1));
129     cout << "tau0: " << tau0 << endl;
130     reset();
131     ejecucion = 0;
132     while(ejecucion<ejecuciones){
133         reset();
134         ciclo = 0;
135         while(ciclo<ciclos){
136             sociedad.current_best = NULL;
137             avanzar();
138             demonio();
139             ciclo++;
140         }
```

```
141             cout << sociedad.mejor_costo << ";";
142             cout << sociedad.peor_costo << ";";
143             cout << costo_promedio/(poblacion*ciclos) << ";";
144             cout << sociedad.iteracion_mc << ";";
145             cout << sociedad.tiempo_mc << ";";
146             cout << _used_time(sociedad.comienzo)/1000000.0 << ";";
147             sociedad.solucion_mc.imprimir();
148             cout << endl;
149 
150             ejecucion++;
151         }
152     }
153 
154     int main(int argc, char** argv)
155     {
156         Problema p = Problema_ACS();
157         p.cargar_costos(argv[2]);
158         p.cargar_configuracion(argv[1]);
159         p.correr();
160 
161         return(0);
162     }
```

## A.7. Código fuente - mmas.cc

```
1   #include "aco.h"
2   
3   class Problema_MMAS: public Problema
4   {
5       public:
6           Problema_MMAS();
7           ~Problema_MMAS();
8   
9           void calcular_tau();
10  
11          void inicializar();
12          void reset();
13          void avanzar();
14          void demonio();
15          void generar_solucion(unsigned int c);
16          void correr();
17          void evaporar();
18          void deposito_mejor_global();
```

```
19  };
20
21  Problema_MMAS::Problema_MMAS()
22  {
23  }
24
25  Problema_MMAS::~Problema_MMAS()
26  {
27  }
28
29  void Problema_MMAS::calcular_tau()
30  {
31      unsigned int i, j;
32      for(i=0;i<dimension;i++)
33      {
34          for(j=0;j<dimension;j++)
35          {
36              tau[i][j] = t_max;
37          }
38      }
39  }
40
41  void Problema_MMAS::inicializar(){
42      calcular_eta();
43      sociedad.inicializar(poblacion, (dimension+1));
44      vecino.iniciar(dimension-1);
45  }
46
47  void Problema_MMAS::reset(){
48      calcular_tau();
49      sociedad.reset();
50      costo_promedio = 0;
51  }
52
53  void Problema_MMAS::avanzar(){
54      unsigned int i;
55      for(i=0;i<poblacion;i++){
56          generar_solucion(i);
57      }
58  }
59
60  void Problema_MMAS::demonio(){
61      sociedad.actualizar_mejor_solucion(ciclo);
62      evaporar();
63      deposito_mejor_global();
```

```
64  }
65
66  void Problema_MMAS::evaporar(){
67      unsigned int i, j;
68      for(i=0;i<dimension;i++)
69      {
70          for(j=0;j<dimension;j++){
71              tau[i][j] = tau[i][j] * (1.0-rho);
72              if(tau[i][j]<t_min)
73                  tau[i][j]=t_min;
74          }
75      }
76  }
77
78  void Problema_MMAS::deposito_mejor_global()
79  {
80      unsigned int i, origen, destino;
81      double Delta;
82      double r = random_double(0,1);
83      double r1 = (double)ciclo/(double)(ciclos-1);
84      t_max = 1.0/(rho*deposito*(sociedad.mejor_costo+1));
85      t_min = t_max/max_min;
86      if(r > r1){
87          Delta = 1.0 / ((*(sociedad.current_best)).get_costo()+1.0);
88          for(i=1;i<=dimension;i++){
89              tau[(*(sociedad.current_best)).elemento(i-1)][(*(sociedad.
                    current_best)).elemento(i)]+=deposito*Delta;
90              if(tau[(*(sociedad.current_best)).elemento(i-1)][(*(
                    sociedad.current_best)).elemento(i)]>t_max)
91                  tau[(*(sociedad.current_best)).elemento(i-1)][(*(
                        sociedad.current_best)).elemento(i)]=t_max;
92          }
93      }else{
94          Delta = 1.0 / (sociedad.solucion_mc.get_costo()+1.0);
95          for(i=1;i<=dimension;i++){
96              tau[sociedad.solucion_mc.elemento(i-1)][sociedad.
                    solucion_mc.elemento(i)]+=deposito*Delta;
97              if(tau[sociedad.solucion_mc.elemento(i-1)][sociedad.
                    solucion_mc.elemento(i)]>t_max)
98                  tau[sociedad.solucion_mc.elemento(i-1)][sociedad.
                        solucion_mc.elemento(i)]=t_max;
99          }
100     }
101 }
102
```

```
103 void Problema_MMAS::generar_solucion(unsigned int c){
104     unsigned int cant = 0;
105     unsigned int s;
106     unsigned int i;
107     double *b;
108 
109     vecino.reset();
110 
111     sociedad.asignar(c, 0, 0);
112 
113     while (vecino.tam()>0){
114         double sum = 0;
115         unsigned int s,i;
116         unsigned int anterior = sociedad.elemento(c, cant);
117         b = new double[vecino.tam()];
118         for (i=0;i<vecino.tam();i++)
119         {
120             double x= pow(tau[anterior][vecino.elemento(i)],alpha)*
121                       pow(eta[anterior][vecino.elemento(i)],beta);
122             sum = sum + x;
123             b[i] = sum;
124         }
125         double R = random_double(0, sum);
126         i=0;
127         while(i<vecino.tam()&&b[i]<R){
128             i++;
129         }
130         s = i;
131         cant++;
132         sociedad.asignar(c, cant, vecino.elemento(s));
133         vecino.eliminar(s);
134         free(b);
135     }
136     cant++;
137     sociedad.asignar(c, cant, 0);
138     sociedad.calculos(ciclo, c, costo_solucion((*(sociedad.solucion(c))
           ).soluciones));
139     costo_promedio += sociedad.soluciones[c].costo;
140 }
141 
142 void Problema_MMAS::correr(){
143     inicializar();
144     ejecucion = 0;
145     while(ejecucion<ejecuciones){
146         ciclo = 0;
```

```
147             t_max = 1.0 / (rho*costo_promedio_cargado);
148             reset();
149             avanzar();
150             demonio();
151             t_max = 1.0/(poblacion*deposito*(sociedad.mejor_costo+1));
152             t_min = t_max/max_min;
153             reset();
154             ciclo = 0;
155             while(ciclo<ciclos){
156                 sociedad.current_best = NULL;
157                 avanzar();
158                 demonio();
159                 ciclo++;
160             }
161             cout << sociedad.mejor_costo << ";";
162             cout << sociedad.peor_costo << ";";
163             cout << costo_promedio/(poblacion*ciclos) << ";";
164             cout << sociedad.iteracion_mc << ";";
165             cout << sociedad.tiempo_mc << ";";
166             cout << _used_time(sociedad.comienzo)/1000000.0 << ";";
167             sociedad.solucion_mc.imprimir();
168             cout << endl;
169 
170             ejecucion++;
171         }
172     }
173 
174     int main(int argc, char** argv)
175     {
176         Problema_MMAS p = Problema_MMAS();
177         p.cargar_costos(argv[2]);
178         p.cargar_configuracion(argv[1]);
179         p.correr();
180 
181         return(0);
182     }
```

## A.8. Código fuente - asrank.cc

```
1   #include "aco.h"
2   
3   class Problema_ASrank: public Problema
4   {
```

```
5       public:
6           Problema_ASrank();
7           ~Problema_ASrank();
8
9           void inicializar();
10          void reset();
11          void avanzar();
12          void demonio();
13          void generar_solucion(unsigned int c);
14          void correr();
15          void evaporar();
16          void deposito_mejor_global();
17          void deposito_ranking();
18  };
19
20  Problema_ASrank::Problema_ASrank()
21  {
22  }
23
24  Problema_ASrank::~Problema_ASrank()
25  {
26  }
27
28  void Problema_ASrank::inicializar(){
29      calcular_eta();
30      sociedad.inicializar(poblacion, (dimension+1));
31      vecino.iniciar(dimension-1);
32      lista = ListaOrdenada();
33  }
34
35  void Problema_ASrank::reset(){
36      calcular_tau();
37      sociedad.reset();
38      costo_promedio = 0;
39  }
40
41  void Problema_ASrank::avanzar(){
42      unsigned int i;
43      for(i=0;i<poblacion;i++){
44          generar_solucion(i);
45      }
46  }
47
48  void Problema_ASrank::demonio(){
49      sociedad.actualizar_mejor_solucion(ciclo);
```

```
50      evaporar();
51      deposito_mejor_global();
52      deposito_mejor_global();
53  }
54
55  void Problema_ASrank::evaporar(){
56      unsigned int i, j;
57      for(i=0;i<dimension;i++)
58      {
59          for(j=0;j<dimension;j++){
60              tau[i][j] = tau[i][j] * (1.0-rho);
61          }
62      }
63  }
64
65  void Problema_ASrank::deposito_ranking()
66  {
67      unsigned int i, j, origen, destino;
68      SolucionEnlazada *actual = lista.primero;
69      j=0;
70      while(actual!=NULL&&j<rank){
71          double Delta = 1.0 / ((*((*actual).solucion)).get_costo()+1.0);
72          for(i=1;i<=dimension;i++){
73              tau[(*((*actual).solucion)).elemento(i-1)][(*((*actual).
                    solucion)).elemento(i)]+=(rank-j)*deposito*Delta;
74          }
75          actual = (*actual).siguiente;
76          j++;
77      }
78  }
79
80  void Problema_ASrank::deposito_mejor_global()
81  {
82      unsigned int i, origen, destino;
83      double Delta = 1.0 / (sociedad.solucion_mc.get_costo()+1.0);
84      for(i=1;i<=dimension;i++){
85          tau[sociedad.solucion_mc.elemento(i-1)][sociedad.solucion_mc.
                elemento(i)]+=rank*deposito*Delta;
86      }
87  }
88
89  void Problema_ASrank::generar_solucion(unsigned int c){
90      unsigned int cant = 0;
91      unsigned int s;
92      unsigned int i;
```

```
93      double *b;
94
95      vecino.reset();
96
97      sociedad.asignar(c, 0, 0);
98
99      while (vecino.tam()>0){
100         double sum = 0;
101         unsigned int s,i;
102         unsigned int anterior = sociedad.elemento(c, cant);
103         b = new double[vecino.tam()];
104         for (i=0;i<vecino.tam();i++)
105         {
106             double x= 100000*pow(tau[anterior][vecino.elemento(i)],
                    alpha)*
107                     pow(eta[anterior][vecino.elemento(i)],beta);
108             sum = sum + x;
109             b[i] = sum;
110         }
111         double R = random_double(0, sum);
112         i=0;
113         while(i<vecino.tam()&&b[i]<R){
114             i++;
115         }
116         s = i;
117         cant++;
118         sociedad.asignar(c, cant, vecino.elemento(s));
119         vecino.eliminar(s);
120         free(b);
121     }
122     cant++;
123     sociedad.asignar(c, cant, 0);
124     sociedad.calculos(ciclo, c, costo_solucion((*(sociedad.solucion(c))
          ).soluciones));
125     costo_promedio += sociedad.soluciones[c].costo;
126     lista.agregar(&sociedad.soluciones[c]);
127 }
128
129 void Problema_ASrank::correr(){
130     inicializar();
131     ejecucion = 0;
132     while(ejecucion<ejecuciones){
133         reset();
134         ciclo = 0;
135         while(ciclo<ciclos){
```

```
136             lista.reset();
137             sociedad.current_best = NULL;
138             avanzar();
139             demonio();
140             ciclo++;
141         }
142         cout << sociedad.mejor_costo << ";";
143         cout << sociedad.peor_costo << ";";
144         cout << costo_promedio/(poblacion*ciclos) << ";";
145         cout << sociedad.iteracion_mc << ";";
146         cout << sociedad.tiempo_mc << ";";
147         cout << _used_time(sociedad.comienzo)/1000000.0 << ";";
148         sociedad.solucion_mc.imprimir();
149         cout << endl;
150 
151         ejecucion++;
152     }
153 }
154 
155 int main(int argc, char** argv)
156 {
157     Problema_ASrank p = Problema_ASrank();
158     p.cargar_costos(argv[2]);
159     p.cargar_configuracion(argv[1]);
160     p.correr();
161 
162     return(0);
163 }
```

# Bibliografía


[1] Enrique Alba. *Parallel Metaheuristics: A New Class of Algorithms*. John Wiley & Sons, NJ, USA, August 2005.

[2] J. Bautista and J. Pereira. Ant algorithms for a time and space constrained assembly line balancing problem. *European Journal of Operational Research*, 177, 2007. HB: 0.33, HBTot: 20.6.

[3] B. Bullnheimer, R. F. Hartl, and C. Strauß. A new rank based version of the ant system – a computational study, 1997.

[4] by a high commis vouyager. The travelling salesman - how he must be and what he should de in order to be sure to perform his tasks and have success in his business, 1832.

[5] G. Di Caro and M. Dorigo. Extending AntNet for best-effort Quality-of-Service routing. Unpublished presentation at *ANTS'98 – From Ant Colonies to Artificial Ants: First International Workshop on Ant Colony Op, October 1998.*

*[6] Jill Cirasella, David S. Johnson, Lyle A. Mcgeoch, and Weixiong Zhang. The asymmetric traveling salesman problem: Algorithms, instance generators, and tests, 2001.*

*[7] M. Dorigo and L. M. Gambardella. Ant colony system: A cooperative learning approach of cooperating agents.* IEEE Transactions on Evolutionary Computation*, 1(1):54–66, 1997.*

*[8] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperative agents.* IEEE Transactions on Systems, Man, and Cybernetics*, Part B, 26(1):29–41, 1996.*

*[9] M. Dorigo and T. Stützle.* Ant colony optimization*. Bradford Books. MIT Press, 2004.*

*[10] Marco Dorigo, Gianni Di Caro, and Luca M. Gambardella. Ant algorithms for discrete optimization.* Artificial Life*, 5:137–172, 1999.*

*[11] T. A. Feo and M. G. C. Resende. Greedy randomized adaptive search procedures. Journal of Global Optimization, 6:109-133, 1995.*

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

*[13] Fred Glover. Scatter search and path relinking, 1999.*

*[14] Fred Glover, Manuel Laguna, and Rafael Martí. Fundamentals of scatter search and path relinking.* CONTROL AND CYBERNETICS*, 39:653–684, 2000.*

*[15] Pierre Hansen and Nenad Mladenovic. Variable neighborhood search: Principles and applications.* European Journal of Operational Research*, 130(3):449–467, 2001.*

*[16] Helena R. Lourenço, Olivier Martin, and Thomas Stützle. Iterated local search. In Fred Glover and Gary Kochenberger, editors,* Handbook of Metaheuristics*, volume 57 of* International Series in Operations Research & Management Science*, pages 321–353. Kluwer Academic Publishers, Norwell, MA, 2002.*

*[17] D. Martens, M. De Backer, R. Haesen, J. Vanthienen, M. Snoeck, and B. Baesens. Classification with ant colony optimization.* IEEE Trans. Evolutionary Computation*, 11(5):651–665, 2007.*

*[18] Olivier Martin, Steve W. Otto, and Edward W. Felten. Large-step markov chains for the traveling salesman problem.* Complex Systems*, 5:299–326, 1991.*

*[19] H. Muhlenbein and G. Paaß. From recombination of genes to the estimation of distributions i. binary parameters.* Lecture Notes in Computer Science*, 1141:178–187, 1996.*

*[20] C.H. Papadimitriou and K. Steiglitz. Combinatorial optimization - algorithms and complexity.* Dover Publications, Inc., New York*, 1982.*

*[21] L. S. Pitsoulis and M. G. C. Resende. Greedy randomized adaptive search procedure, 2002.*

*[22] C. D. Gelatt S. Kirkpatrick and M. P. Vecchi. Optimization by simulated anneling. Science, 220(4598):671-680, 1983.*

*[23] T. Stützle. Max-min ant system for quadratic assignment problems, 1997.*

*[24] T. Stützle. Local search algorithms for combinatorial problems: Analysis, improvemnts, and new applications, volume 220 of diski.* Infix, Sankt Agustin, Germany*, 1999.*

*[25] T. Stützle and H.H. Hoss. Improving the ant system: A detailed report on the max-min ant system. Technical Report AIDA-96-12, FG Intellektik, FB Informatik, TU Darmstadt, Germany, Aug 1996.*

*[26] T. Stützle and H.H. Hoss. Max-min ant system.* Future Generation Computer Systems*, 16(8):889–914, 2000.*

*[27] T.G. Stutzle.* Local Search Algorithms for Combinatorial Problems - Analysis, Improvements and New Applications*. Dissertations in Artificial Intelligence-Infix, 220. Ios Pr Inc, 1999.*

*[28] C. Voudouris and E. Tsang. Guided local search and its application to the traveling salesman problem.* European Journal of Operational Research*, 113(2):469–499, 1999.*