



# UNIVERSIDAD DE GRANADA

---

## PRÁCTICA 2: SNIMP

---

METAHEURÍSTICAS

**Estudiante:** CARMEN AZORÍN MARTÍ

**DNI:** 48768328W

**Curso:** 5º DGIIM

**Correo:** CARMENAZORIN@CORREO.UGR.ES

**Prácticas:** LUNES 15.30 - 17.30

# Índice

<b>1. Introducción</b>	<b>3</b>
<b>2. Aplicación de algoritmos</b>	<b>4</b>
2.1. Representación de las soluciones . . . . .	4
2.2. Descripción del problema y función objetivo . . . . .	4
2.3. Creación de soluciones aleatorias . . . . .	5
2.4. Heurística asociada a cada nodo . . . . .	6
2.5. Mutación de soluciones . . . . .	6
2.6. Cruce en dos puntos con reparación . . . . .	7
2.7. Cruce con orden . . . . .	8
<b>3. La clase Graph</b>	<b>8</b>
<b>4. Algoritmo Genético Generacional (AGG)</b>	<b>9</b>
<b>5. Algoritmo Genético Estacionario (AGE)</b>	<b>12</b>
<b>6. Algoritmo Memético (AM)</b>	<b>14</b>
6.1. AM-(10,1.0) . . . . .	16
6.2. AM-(10,0.1) . . . . .	17
6.3. AM-(10,0.1mej) . . . . .	17
<b>7. Estructura del código</b>	<b>18</b>
7.1. Compilación y ejecución . . . . .	19
<b>8. Experimentos y análisis de resultados</b>	<b>20</b>
8.1. Configuración de los algoritmos . . . . .	20
8.2. Análisis de los cruces . . . . .	21
8.3. Análisis de cada caso . . . . .	21
8.4. Análisis Final . . . . .	28

# Metaheurísticas

## Práctica 2: SNIMP

### 1. Introducción

El problema SNIMP (Social Network Influence Maximization Problem) busca encontrar un conjunto de nodos (usuarios) en una red social que maximicen la influencia en la red. Es decir, dados unos usuarios "semilla", queremos que la propagación de la información a través de la red sea la mayor posible.

Formalmente, el problema se define a partir de un conjunto de números enteros y un tamaño de solución predefinido  $k$ . El objetivo es seleccionar exactamente  $k$  elementos del conjunto original de manera que el valor de la función objetivo (fitness) se maximice.

En esta práctica, SNIMP se presenta a través de un grafo dirigido, definido en archivos de texto con el siguiente formato:

```
# Directed graph (each unordered pair of nodes is saved once): CA-GrQc.txt
# Collaboration network of Arxiv General Relativity category
# Nodes: 5242 Edges: 28980
# FromNodeId    ToNodeId
0    1
0    2
0    3
0    4
0    5
0    6
...
```

Cada nodo representa un perfil de la red social y las aristas indican que un perfil ha influenciado a otro. Queremos seleccionar los  $k$  perfiles que más gente influncian.

El número total de soluciones posibles es combinatorial, es decir, todas las combinaciones de  $n$  nodos tomados de  $k$  en  $k$ , lo que hace inviable recorrer el espacio de soluciones completo para instancias de tamaño grande. Por ese motivo, el problema SNIMP utiliza heurísticas y metahuerísticas que calculen soluciones buenas.

## 2. Aplicación de algoritmos

Una metaheurística es un algoritmo de optimización diseñado para encontrar soluciones cercanas a las óptimas en problemas complejos donde la búsqueda exhaustiva es impracticable. En nuestro caso, usaremos las siguientes metaheurísticas para el problema SNIMP, que es NP-completo:

- **Algoritmos Genéticos Generacionales (AGG).** Basados en evolución poblacional completa en cada generación, con operadores de selección, cruce y mutación, y elitismo para conservar las mejores soluciones.
- **Algoritmos Genéticos Estacionarios (AGE).** Variante de los algoritmos genéticos donde solo se reemplaza una parte de la población por generación, lo que permite una evolución más progresiva.
- **Algoritmos Meméticos (AM).** Extienden los algoritmos genéticos incorporando búsqueda local sobre los cromosomas, mejorando la explotación del espacio de soluciones.

Estos algoritmos permiten comparar el rendimiento, tiempo de cómputo y calidad de la solución encontrada para un problema tan complejo y útil como el que nos respecta.

### 2.1. Representación de las soluciones

Las soluciones se representan mediante un vector de enteros, donde cada entero se corresponde con el identificador de un nodo seleccionado del grafo. Este vector tiene un tamaño fijo de 10 nodos, especificado en el constructor de la clase del problema:

```
typedef std::vector<int> tSolution;
```

### 2.2. Descripción del problema y función objetivo

El problema se representa mediante una clase `Snimp` derivada de `Problem` que contiene el grafo como una lista de adyacencia, junto con información sobre el número de nodos y el tamaño de la solución. La función objetivo (fitness) evalúa una solución simulando un proceso de propagación de infecciones.

Para cada solución candidata:

- Se realizan 10 simulaciones independientes (entornos).
- En cada simulación, se parte de los nodos seleccionados como infectados iniciales.
- En cada iteración, los nodos infectados intentan contagiar a sus vecinos una probabilidad del 1 %.

- El proceso se repite hasta que no aparezcan nuevos infectados.
- El fitness final es la media del número total de nodos infectados tras las 10 simulaciones.

Función Fitness(Solución S):

```

fitness_total <- 0
Para i = 1 hasta 10 hacer:
    infectados_iniciales <- S
    infectados <- S
    Mientras infectados_iniciales no vacío hacer:
        infectados_nuevos <- []
        Para cada nodo j en infectados_iniciales hacer:
            Para cada vecino v de j hacer:
                Si v no está infectado y Random() < 0.01 entonces:
                    infectados_nuevos <- infectados_nuevos {v}
            infectados <- infectados infectados_nuevos
            infectados_iniciales <- infectados_nuevos
        fitness_total <- fitness_total + tamaño(infectados)
    devolver fitness_total / 10

```

### 2.3. Creación de soluciones aleatorias

El operador de generación de soluciones aleatorias selecciona `solSize` nodos aleatorios distintos del grafo:

Función CreateSolution():

```

infectados <- []
nodosDisponibles <- []

Para cada par (nodo, vecinos) en graph:
    nodosDisponible <- nodosDisponibles + nodo

nodosBarajados <- barajar(nodosDisponibles)

Para i desde 0 hasta solSize-1:
    Si i < tamaño(nodosBarajados):
        infectados[i] <- nodosBarajados[i]
    Sino:
        terminar
return infectados

```

Lo que hacemos es crear dos listas vacías: una para la solución final y otra para los nodos disponibles.

A continuación, extraemos todos los nodos existentes del grafo (las keys del `graph`) y los barajamos para obtener un orden aleatorio. Finalmente, tomamos los primeros `solSize` nodos del conjunto barajado.

## 2.4. Heurística asociada a cada nodo

Para facilitar el algoritmo voraz (Greedy), se define una función heurística que evalúa la importancia de un nodo. Esta heurística es la suma del número de vecinos del nodo más la suma de los grados de esos vecinos:

Función Heurística(Nodo n):

```
heuristica ← 0
```

```
vecinos ← grafo[nodo]
```

Para cada vecino en vecinos:

```
    heuristica ← heuristica + tamaño(grafo[vecinos])
```

```
return número_de_vecinos(nodo) + sumaGradosVecinos
```

## 2.5. Mutación de soluciones

La función de mutación implementa un operador que reemplaza un nodo de la solución actual por otro nodo que no esté ya incluido en ella.

Función Mutate(solution):

```
mutated ← copia de solution
```

```
used ← conjunto de nodos en solution
```

```
unused ← nodos del grafo que no están en used
```

```
if unused está vacío:
```

```
    return mutated
```

```
pos ← posición aleatoria de 0 a solSize - 1
```

```
newNode ← nodo aleatorio de unused
```

```
mutated[pos] ← newNode
```

```
return mutated
```

Lo que hace el método es copiar la solución actual para no modificarla directamente y construye una lista de nodos no presentes en la solución `unusedNodes`. Posteriormente, elige una posición aleatoria de la solución y se selecciona un nuevo nodo de entre los no usados. Finalmente, sustituye el nuevo nodo en la posición elegida.

## 2.6. Cruce en dos puntos con reparación

La función del cruce sin orden genera dos hijos válidos a partir de dos padres. Además, cada hijo debe tener exactamente `solSize` nodos únicos.

```
Función crossover2Puntos(p1, p2)
  n ← tamaño de la solución
  punto1 ← valor aleatorio entre 0 y n-2
  punto2 ← valor aleatorio entre punto1+1 y n-1

  hijo1 ← vector de tamaño n inicializado a -1
  hijo2 ← vector de tamaño n inicializado a -1

  Para i desde 0 hasta tamaño de p1 - 1 hacer
    Si i está entre punto1 y punto2 entonces
      hijo1[i] ← p2[i]
      hijo2[i] ← p1[i]
    Sino
      hijo1[i] ← p1[i]
      hijo2[i] ← p2[i]
  Fin Si
Fin Para

  todosNodos ← lista de todos los nodos posibles

  repararHijo(hijo1, todosNodos)
  repararHijo(hijo2, todosNodos)

  return (hijo1, hijo2)
```

El método elige dos puntos aleatorios que delimitan el segmento de los genes. Y se copia ese segmento del otro padre en cada hijo:

- El hijo 1 recibe el segmento del padre 2
- El hijo 2 recibe el segmento del padre 1

El resto de los genes se completa desde el padre original, permitiendo que se repitan algunos nodos que son comunes a ambos padres. Para solucionar el problema, existe la función auxiliar `reparar_hijo` que busca nodos que no estén en el hijo todavía y los inserta en las posiciones con nodos repetidos.

## 2.7. Cruce con orden

La función de cruce con orden genera dos hijos distintos a partir de dos padres, manteniendo nodos sin repetir. Para ello, combina los nodos de ambos padres en una lista ordenada crecientemente, y luego reparte los nodos entre los hijos de forma alterna.

```

Función crossoverOrden(p1, p2)
    conjunto ← concatenación de p1 y p2

    ordenado ← copia de conjunto
    ordenar ordenado de menor a mayor

    h1, h2 ← listas vacías

    Para i desde 0 hasta tamaño de ordenado - 1 hacer
        Si i es par y tamaño de h1 < solSize entonces
            añadir ordenado[i] a h1
        Sino si tamaño de h2 < solSize entonces
            añadir ordenado[i] a h2
        Fin Si
    Fin Para

    return (h1, h2)

```

El método forma un conjunto con todos los nodos presentes en los padres **p1** y **p2** y los ordena crecientemente. A continuación, se reparten los hijos:

- El hijo 1 recibe los nodos de posiciones pares
- El hijo 2 recibe los nodos de las posiciones impares

## 3. La clase Graph

La clase **Graph** implementa una estructura de datos para representar grafos dirigidos mediante listas de adyacencia, donde cada nodo se asocia con un vector de sus nodos vecinos. Esta representación es eficiente en memoria y tiempo para algoritmos que requieren frecuentes accesos a los vecinos de un nodo.

La clase tiene un único atributo: `unordered_map<int, vector<int>> adjList`, un mapa no ordenado que asocia el identificador de un nodo con una lista de nodos adyacentes. Además tiene los siguientes métodos clave:



- `addEdge(int from, int to)` que añade una arista entre dos nodos.
- `printGraph()` que imprime el grafo en formato legible.
- `readGraphFromFile(const string &filename)` que lee un archivo de texto que define las aristas del grafo (los archivos incluidos en el enunciado de la práctica) y contruye el objeto `Graph`.

Función `readGraphFromFile(texto nombre_archivo):`

```
archivo <- abrir nombre_archivo
grafo <- nuevo Graph()
```

Mientras `leer_linea(archivo):`

```
Si linea empieza con '#':
    continuar
from, to <- extraer enteros de linea
grafo.addEdge(from, to)
```

```
cerrar archivo
devolver grafo
```

Al principio esta función verifica que el archivo exista y si falla, muestra un error y termina el programa. A continuación procesa línea por línea: si la línea está comentada, la ignora; si la línea es válida, extrae los nodos y añade la arista al grafo. Finalmente, devuelve el grafo construido.

En el `main` se crea la estructura del grafo a partir de un archivo de texto y luego se inicializa el problema SNIMP con estos datos.

## 4. Algoritmo Genético Generacional (AGG)

Las clases `AGG_conorden` y `AGG_sinorden` representan algoritmos genéticos generacionales que parten de una población de soluciones, las mejora generación tras generación aplicando selección, cruce y mutación, y mantiene la mejor solución encontrada.

La diferencia entre ambas clases es: `AGG_sinorden` usa el operador de cruce en dos puntos con reparación y `AGG_conorden` usa el operador de cruce por orden. La base de ambas variantes es la misma, solamente se diferencian en cómo combinan los padres.

Función `AGG_sinorden_optimize(problem, maxevals):`

```
popSize <- 30
Pc <- 0.7      // probabilidad de cruce
Pm <- 0.1      // probabilidad de mutación
```

```
cromSize <- tamaño del cromosoma
evals <- 0

// Inicializar población aleatoria
poblacion <- []
fitnesses <- []
para i = 1 hasta popSize hacer:
    sol <- solución aleatoria
    fit <- fitness(sol)
    añadir sol a poblacion
    añadir fit a fitnesses
    evals++

bestSol <- mejor solución en poblacion
bestFit <- fitness correspondiente

mientras evals < maxevals hacer:
    // Selección por torneo (k = 3)
    padres <- []
    para i = 1 hasta popSize hacer:
        seleccionar 3 individuos aleatorios
        elegir el mejor según fitness
        añadirlo a padres

    // Cruzar parejas con probabilidad Pc
    numCruces <- ceil(Pc * (popSize / 2))
    hijos <- []
    para i = 0 hasta popSize paso 2 hacer:
        si (i / 2) < numCruces entonces:
            (h1, h2) <- crossover2Puntos(padres[i], padres[i+1])
            añadir h1 y h2 a hijos
        si no:
            añadir padres[i] y padres[i+1] a hijos

    // Mutación
    numMutaciones <- round(Pm * popSize)
    elegir al azar numMutaciones índices únicos
    para cada índice i:
        hijos[i] <- mutación(hijos[i])

    // Evaluar nueva población
```

```
newFitnesses <- []
para cada hijo h en hijos mientras evals < maxevals:
    fit <- fitness(h)
    añadir fit a newFitnesses
    evals++
    si fit > bestFit:
        bestFit <- fit
        bestSol <- h

// Elitismo: reemplazar peor hijo si es necesario
idxWorst <- índice del peor fitness en newFitnesses
hijos[idxWorst] <- bestSol
newFitnesses[idxWorst] <- bestFit

poblacion <- hijos
fitnesses <- newFitnesses

return ResultMH(bestSol, bestFit, evals)
```

Se generan 30 soluciones aleatorias y cada una se evalúa mediante la función `fitness` del problema y se guarda el mejor individuo global. A continuación, en cada generación:

1. Se selecciona por torneo de 3 los padres a cruzar, lo que favorece los individuos con mejor fitness, pero manteniendo la diversidad
2. Se cruzan aproximadamente el 70 % de las parejas y el resto se copian tal cual
3. Se mutan aleatoriamente el 10 % de la población
4. Se evalúan todos los nuevos hijos (hasta llegar a `maxevals`)
5. Si el mejor global no está entre los nuevos hijos, se introduce en lugar del peor
6. Reemplazo total a la anterior generación por la nueva

Finalmente, se detiene el algoritmo cuando se alcanza el límite de evaluaciones.

Este algoritmo genético trabaja generando muchas soluciones aleatorias y luego las va cruzando, mutando y evaluando a lo largo del tiempo para intentar encontrar una solución óptima. Lo bueno que tiene es que empieza con una población diversa y va combinando las mejores soluciones para mejorar poco a poco. Además, el cruce y la mutación permiten que no todas las soluciones sean iguales, así que el algoritmo sigue explorando nuevas posibilidades en cada generación. Aun así, gracias al elitismo siempre se asegura de conservar la mejor solución encontrada hasta el momento,

evitando perder calidad si una generación sale peor.

Sin embargo, una desventaja clara es que en teoría tarda mucho más que otros métodos como Greedy o BLsmall, ya que evalúa muchas soluciones en cada iteración.

## 5. Algoritmo Genético Estacionario (AGE)

El Algoritmo Genético Estacionario es la variante de algoritmos genéticos donde la población se modifica poco a poco: en cada iteración se seleccionan sólo 2 padres y se generan 2 hijos. De esos dos hijos, se escoge el mejor y solo se reemplaza el individuo de la población si el hijo es mejor que el peor de la población.

Función AGE\_conorden\_optimize(problem, maxevals):

```
popSize <- 30
Pm <- 0.1
evals <- 0

// Inicializar población aleatoria
poblacion <- []
fitnesses <- []
para i = 1 hasta popSize hacer:
    sol <- solución aleatoria
    fit <- fitness(sol)
    añadir sol a poblacion
    añadir fit a fitnesses
    evals++

bestSol <- mejor solución en poblacion
bestFit <- fitness correspondiente

mientras evals < maxevals hacer:
    // Selección por torneo (k = 3) de 2 padres
    padres <- []
    para i = 1 hasta 2 hacer:
        seleccionar 3 individuos aleatorios
        elegir el mejor según fitness
        añadirlo a padres

    // Cruce con orden (Pc = 1)
    (h1, h2) <- crossoverOrden(padres[0], padres[1])
```

```
// Mutación con probabilidad Pm
si random() < Pm entonces:
    h1 <- mutación(h1)
si random() < Pm entonces:
    h2 <- mutación(h2)

// Evaluar hijos
fit1 <- fitness(h1)
evals++
fit2 <- fitness(h2)
evals++

mejorHijo <- el de mayor fitness entre h1 y h2
fitMejorHijo <- su fitness

// Reemplazo: sustituir el peor si el hijo es mejor
idxWorst <- índice del peor fitness en fitnesses
si fitMejorHijo > fitnesses[idxWorst] entonces:
    poblacion[idxWorst] <- mejorHijo
    fitnesses[idxWorst] <- fitMejorHijo

// Actualizar mejor solución global si mejora
si fitMejorHijo > bestFit entonces:
    bestFit <- fitMejorHijo
    bestSol <- mejorHijo

return ResultMH(bestSol, bestFit, evals)
```

Este método usa cruce con orden, aunque la clase `AGE_sinorden` llamaría al cruce sin orden.

El proceso de este algoritmo se basa en generar una población fija de 30 soluciones. En cada generación:

1. Se hace selección de dos padres por torneo de 3
2. Se hace un cruce del tipo determinado de ambos padres
3. Se hace mutación con probabilidad 0.1 de ambas soluciones
4. Se hace un reemplazo elitista donde, si alguno de los hijos supera al peor individuo de la población, lo sustituye

Esta estrategia hace que el algoritmo evolucione de forma más estable y lenta, refinando poco a

poco la población. En lugar de explorar muchas soluciones nuevas, como en AGG, se centra en una mejora más local y progresiva. Esto provoca una convergencia más lenta, pero más consistente, evitando grandes oscilaciones. Es bueno para mejorar soluciones ya buenas, pero es menos potente en la búsqueda de soluciones radicalmente nuevas, como es nuestro caso.

## 6. Algoritmo Memético (AM)

Un Algoritmo Memético es una combinación del algoritmo genético generacional con un componente de búsqueda local. La idea es mezclar soluciones (cruce y mutación) y mejorarlas localmente.

La idea general del algoritmo es la siguiente, aunque se han realizado 3 variantes diferentes:

Función `AM_optimize(problem, maxevals):`

```
popSize <- 30
Pc <- 0.7           // probabilidad de cruce
Pm <- 0.1           // probabilidad de mutación
generacionesBL <- 10 // cada cuántas generaciones aplicar BL
evals <- 0
generation <- 0

// Inicializar población
poblacion <- []
fitnesses <- []
para i = 1 hasta popSize hacer:
    sol <- solución aleatoria
    fit <- fitness(sol)
    añadir sol a poblacion
    añadir fit a fitnesses
    evals++

bestSol <- mejor solución en poblacion
bestFit <- su fitness

mientras evals < maxevals hacer:
    generation++

    // Selección por torneo (k = 3)
    padres <- []
    para i = 1 hasta popSize hacer:
        seleccionar 3 individuos aleatorios
        añadir el mejor a padres
```

```
// Cruzar parejas con probabilidad Pc
numCruces <- ceil(Pc * (popSize / 2))
hijos <- []
para i = 0 hasta popSize paso 2 hacer:
  si (i / 2) < numCruces:
    (h1, h2) <- crossoverOrden(padres[i], padres[i+1])
  sino:
    h1 <- padres[i], h2 <- padres[i+1]
  añadir h1 y h2 a hijos

// Mutación
numMutaciones <- round(Pm * popSize)
seleccionar al azar numMutaciones índices únicos
para cada índice i:
  hijos[i] <- mutación(hijos[i])

// Evaluación
newFitnesses <- []
para cada hijo h mientras evals < maxevals:
  fit <- fitness(h)
  añadir fit a newFitnesses
  si fit > bestFit:
    bestFit <- fit
    bestSol <- h
  evals++

// Búsqueda Local cada N generaciones
...

// Elitismo: insertar el mejor en lugar del peor
idxWorst <- índice del peor fitness en newFitnesses
hijos[idxWorst] <- bestSol
newFitnesses[idxWorst] <- bestFit

poblacion <- hijos
fitnesses <- newFitnesses

retornar ResultMH(bestSol, bestFit, evals)
```

Este algoritmo genera diversidad mediante cruces y mutaciones, pero también refina de vez en cuando

las soluciones usando una búsqueda local.

Primero inicializa una población aleatoria de soluciones y las evalúa, guardando la mejor solución encontrada. En cada generación, al igual que en AGG, se seleccionan los padres por torneo de 3 y se realizan cruces por parejas, generando dos hijos por cruce. Los que no se cruzan se copian tal cual a la siguiente generación. Y se aplican mutaciones aleatorias sobre algunos individuos.

Lo interesante es que cada 10 generaciones, se aplica la búsqueda local sobre los hijos según alguna restricción de cada versión. Esto permite refinar esas soluciones a nivel individual y aumentar su calidad.

Finalmente, se aplica elitismo: si el mejor individuo global no está en la nueva población, se asegura que lo esté, reemplazando al peor.

El algoritmo debería ir mejorando progresivamente, combinando la variabilidad de las soluciones nuevas con las mejoras puntuales gracias a la búsqueda local. Se espera que alcance mejores resultados que un algoritmo genético o una búsqueda local sola, ya que esta equilibrando exploración y explotación. Por otro lado, puede pasar que el tiempo de ejecución aumente, ya que las búsquedas locales son costosas.

## 6.1. AM-(10,1.0)

En esta versión cada 10 generaciones, se aplica búsqueda local a todos los individuos de la población. Para ello, usamos BLsmall con un máximo de 20 evaluaciones por individuo.

```
// Aplicar búsqueda local cada N generaciones
si (generation módulo generacionesBL == 0) entonces:
  para i = 0 hasta popSize - 1 hacer:
    si evals >= maxevals entonces salir del bucle

    // Aplicar BLsmall sobre el hijo i con un máximo de 20 evaluaciones
    resultBL <- BLsmall.optimize(problem, hijos[i], newFitnesses[i], mínimo(maxevals - e

    // Si mejora, actualizar la solución y su fitness
    si resultBL.fitness > newFitnesses[i] entonces:
      hijos[i] <- resultBL.solution
      newFitnesses[i] <- resultBL.fitness

    evals <- evals + resultBL.evaluations
```

Esta variante es la más intensa, ya que modifica todas las soluciones de la población cada 10 gene-



raciones, lo que teóricamente hace que aumente la calidad de la solución. Sin embargo, esto requiere un coste computacional muy alto.

## 6.2. AM-(10,0.1)

En esta versión, cada 10 generaciones, se selecciona cada individuo con probabilidad 0.1 y se les aplica búsqueda local.

```
// Aplicar BL cada N generaciones sobre un subconjunto aleatorio de hijos
si (generation módulo generacionesBL == 0) entonces:
  para i = 0 hasta popSize - 1 mientras evals < maxevals hacer:
    si número aleatorio entre 0 y 1 <= pLS entonces:
      resultBL <- BLsmall.optimize(problem, hijos[i], newFitnesses[i], mínimo(maxevals

      si resultBL.fitness > newFitnesses[i] entonces:
        hijos[i] <- resultBL.solution
        newFitnesses[i] <- resultBL.fitness

    evals <- evals + resultBL.evaluations
```

Esta versión tiene un coste más controlado, ya que no se hace la búsqueda local para toda la población, sino para un 10 % de ella. Pero mantiene la diversidad de la población, aunque con un efecto de mejora menos consistente que en la versión anterior.

## 6.3. AM-(10,0.1mej)

En esta versión, cada 10 generaciones, se ordena la población por fitness y se aplica la búsqueda local a los  $0.1 \times M$  mejores individuos.

```
// Aplicar BL cada N generaciones sobre los mejores pLS * N hijos
si (generation módulo generacionesBL == 0) entonces:
  numMejores <- máximo(1, redondear(pLS * popSize))

  // Crear lista de índices ordenados por fitness descendente
  indices <- [0, 1, ..., popSize - 1]
  ordenar indices de forma que newFitnesses[indices[i]] > newFitnesses[indices[i+1]]

  para i = 0 hasta numMejores - 1 mientras evals < maxevals hacer:
    idx <- indices[i]
    resultBL <- BLsmall.optimize(problem, hijos[idx], newFitnesses[idx], mínimo(20, maxevals
```

```
si resultBL.fitness > newFitnesses[idx] entonces:
    hijos[idx] <- resultBL.solution
    newFitnesses[idx] <- resultBL.fitness

evals <- evals + resultBL.evaluations
```

Esta versión mejora la anterior, ya que se concentra en las mejores soluciones, evitando malgastar evaluaciones no tan útiles. Sin embargo, si los mejores se estancan, no mejora al resto.

## 7. Estructura del código

El código está estructurado como la plantilla proporcionada por el profesor en [https://github.com/dmolina/template\\_mh](https://github.com/dmolina/template_mh). A continuación se describe la estructura general del proyecto:

- Carpeta `build/` se genera cuando se ejecuta el script de compilación (usando `cmake` y `make`), y contiene los archivos compilados necesarios para ejecutar el programa. Además, se guardan los resultados de las ejecuciones en archivos con nombre `resultados_conjuntoDatos.txt`.
- El archivo `inc/snimp_problem.h` contiene la declaración de la clase `Snimp`, que es una subclase de la clase `Problem`. Esta clase define los métodos de creación de soluciones aleatorias, la evaluación del fitness y la heurística de cada nodo.
- Los archivos `inc/graph.h` y `src/graph.cpp` contienen la declaración e implementación de la clase `Graph`. Esta clase se encarga de leer el archivo de texto que contiene las aristas del grafo y convertirlo a listas de adyacencia, donde cada nodo se asocia con un vector de sus nodos vecinos.
- Los archivos `inc/greedy.h` y `inc/randomsearch.h` definen las clases `GreedySearch` y `RandomSearch`, que implementan los algoritmos correspondientes. Las clases se implementan en los archivos `src/greedy.cpp` y `src/randomsearch.cpp`.
- Los archivos `inc/blsmall.h` y `inc/lsall.h` definen las clases `BLsmall` y `LSall`, que implementan los algoritmos de búsqueda local. Las funciones `optimize` de ambas clases se implementan en el archivo `src/localsearch.cpp`, junto a la función auxiliar `buscar_vecinos_aleatorios` para generar soluciones vecinas aleatorias.
  - En esta práctica se ha modificado la clase `BLsmall` para heredar de `MHTrayectory`, permitiendo aplicar la búsqueda local desde una solución completa.
- En la carpeta `inc/` se han añadido los siguientes archivos:
  - `AGG1.h` algoritmo genético generacional con cruce de dos puntos y reparación.

- AGG2.h algoritmo genético generacional con cruce por orden.
  - AGE1.h algoritmo genético estacionario con cruce por orden.
  - AGE2.h algoritmo genético estacionario con cruce de dos puntos y reparación.
  - AM1.h, AM2.h, AM3.h las trtes variantes de algoritmos meméticos.
- En la carpeta `src/` se han añadido los archivos que implementan las clases definidas en los `.h`.
  - En el archivo `main.cpp` se inicializan los objetos de los diferentes algoritmos y se ejecutan.
  - El archivo `script.sh` automatiza el proceso de compilación y ejecución del programa. Se le puede llamar con o sin parámetro de semilla.

## 7.1. Compilación y ejecución

El archivo `script.sh` es un script en bash que automatiza el proceso de compilación y ejecución del programa. Funciona en varios pasos:

Primero, configura las rutas importantes:

```
BUILD_DIR=~/.Documentos/Quinto-DGIIM/MH/template_mh/build
OUTPUT_FILE=resultados_p2p-Gnutella25.txt
```

Luego compila el proyecto:

```
cd $BUILD_DIR || exit 1
cmake -DCMAKE_BUILD_TYPE=Debug .
make
```

Cuando ejecuta el programa principal (`./main "$SEED"`), va leyendo línea por línea la salida que generan los algoritmos. Cada vez que detecta el nombre de un algoritmo (`AGG_conorden`, `AGG_sinorden`, `AGE_conorden`, `AGE_sinorden`, `AM1`, `AM2`, `AM3`), guarda sus resultados:

```
42 AGG\_conorden "2741 786 4605 1796" 11.5 1000 209
42 AGG\_sinorden "3002 2156 3125 2985" 12.3 1000 12
```

El script hace esto:

- Extrae la solución, fitness, evaluaciones y tiempo de cada algoritmo
- Los formatea correctamente quitando texto sobrante

- Usa semillas que van aumentando (42, 43, 44...) cada vez que prueba los 4 algoritmos
- Todo lo guarda en el archivo `resultados_p2p-Gnutella25.txt` o en el que se indique

Para usarlo simplemente:

1. Poner el script en la carpeta correcta
2. Darle permisos con `chmod +x script.sh`
3. Ejecutarlo con `./script.sh` o `./script.sh 25` si se quisiese ejecutar con la semilla 25, por ejemplo

Cabe destacar que si se quiere ejecutar un conjunto de datos concreto, como `p2pGnutella25.txt`, debemos cambiar el nombre del archivo en el `main.cpp`. Además, para guardar los resultados en el archivo de salida correspondiente, en el script debemos indicar el `OUTPUT_FILE` al nombre que se le quiera dar. El archivo de salida se guardará en la carpeta autogenerada `build`.

## 8. Experimentos y análisis de resultados

### 8.1. Configuración de los algoritmos

Los algoritmos se han ejecutado bajo los siguientes parámetros:

Todos usan las mismas semillas (43 a 47) para ser comparables. Además, el fitness se calcula igual para todos (semilla 10), permitiendo una comparación justa. Los algoritmos genéticos tienen los siguientes parámetros:

- Evaluaciones: 1000
- Población: 30 cromosomas (soluciones diferentes)
- Probabilidad de cruce: 0.7 en el generacional y 1 en el estacionario
- Probabilidad de mutación: 0.1

Los algoritmos meméticos tienen los siguientes parámetros:

- Evaluaciones: 1000
- Población: 30 cromosomas (soluciones diferentes)
- Probabilidad de cruce: 0.7

- Probabilidad de mutación: 0.1

## 8.2. Análisis de los cruces

En los casos del problema que veremos en el siguiente apartado vamos a ver una diferencia de tiempos entre los algoritmos que usan el cruce con orden y los del cruce sin orden.

El `crossover2Puntos` toma un segmento aleatorio del padre 1 y lo reemplaza con el segmento correspondiente del padre 2, y viceversa. Tras eso, tiene que detectar duplicados, reparar los hijos en caso de tener nodos repetidos. Esto implica búsqueda, comparación y edición de vectores, que no tarda poco y se tiene que ejecutar en cada cruce. Todo esto incrementa el coste computacional del cruce.

Por otro lado, `crossoverOrden` simplemente concatena ambos padres, ordena el conjunto y reparte elementos alternos entre hijos. La única operación costosa es la ordenación, pero el resto es muy eficiente. Al no necesitar reparación ni verificación de duplicados, se reduce considerablemente el tiempo de ejecución.

## 8.3. Análisis de cada caso

Se han utilizado cuatro conjuntos de datos reales extraídos de la colección Stanford Large Network Dataset Collection. Estos incluyen una red de colaboración académica y tres instancias de una red P2P en distintos días. A continuación, se analizan los resultados obtenidos en cada caso.

Tabla 1: Resultados promedio de ca-GrQc.txt

Algoritmo	Posición	Fitness	Tiempo (segs)	Evaluaciones
RandomSearch	10	12.5	$5.72 \times 10^{-1}$	1000
Greedy	1	18.8	$1.10 \times 10^{-2}$	1
LSall	2	15.1	$7.08 \times 10^{-1}$	1000
BLsmall	11	11.3	$2.58 \times 10^{-2}$	28.6
AGG_sinorden	6	13.56	$1.18 \times 10^0$	1000
AGG_conorden	8	13.48	$5.78 \times 10^{-1}$	1000
AGE_sinorden	3	13.7	$1.38 \times 10^0$	1000
AGE_conorden	9	12.98	$5.61 \times 10^{-1}$	1000
AM1	5	13.56	$8.47 \times 10^{-1}$	1000
AM2	7	13.5	$6.33 \times 10^{-1}$	1000
AM3	4	13.66	$6.20 \times 10^{-1}$	1000

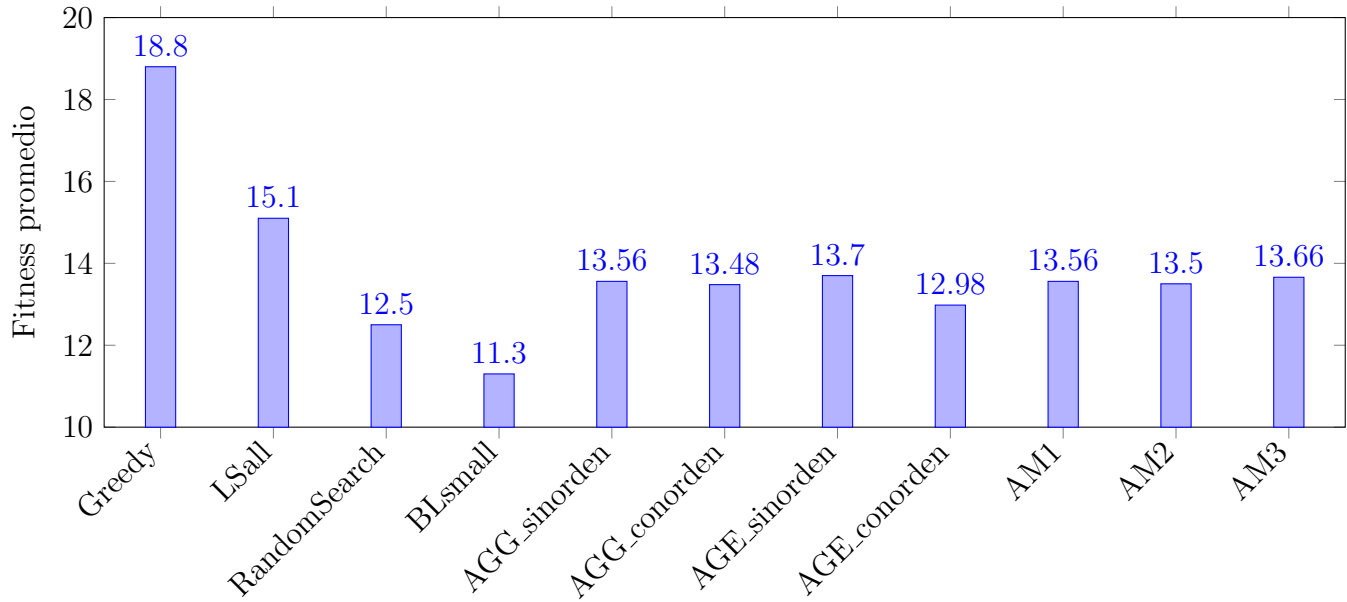


Figura 1: Fitness promedio por algoritmo en ca-GrQc.txt

Este conjunto representa una red de colaboración de autores en el campo de la relatividad general y la cosmología cuántica. Con 5242 nodos y 14496 enlaces, es una red relativamente pequeña y estructurada. Observamos que Greedy sigue destacando como el mejor algoritmo, alcanzando un fitness medio de 18.8 en una sola evaluación. Entre los algoritmos nuevos, los algoritmos meméticos y los genéticos obtienen valores de fitness más homogéneos, en torno a 13.5. Entre ellos, AM3 es el mejor, seguido de AGE\_sinorden. Sin embargo, el tiempo de ejecución varía notablemente: AGE\_sinorden es el más lento, mientras que AGG\_conorden y AGE\_conorden son los más rápidos.

Tabla 2: Resultados promedio de p2p-Gnutella05.txt

Algoritmo	Posición	Fitness	Tiempo (segs)	Evaluaciones
RandomSearch	9	11.38	$6.80 \times 10^{-1}$	1000
Greedy	1	13.7	$1.48 \times 10^{-2}$	1
LSall	10	10.98	$2.18 \times 10^{-1}$	1000
BLsmall	11	10.8	$3.48 \times 10^{-2}$	25.4
AGG_sinorden	6	11.5	$1.21 \times 10^0$	1000
AGG_conorden	5	11.52	$4.38 \times 10^{-1}$	1000
AGE_sinorden	4	11.52	$1.47 \times 10^0$	1000
AGE_conorden	8	11.46	$3.83 \times 10^{-1}$	1000
AM1	2	11.72	$6.99 \times 10^{-1}$	1000
AM2	7	11.48	$4.77 \times 10^{-1}$	1000
AM3	3	11.52	$4.48 \times 10^{-1}$	1000

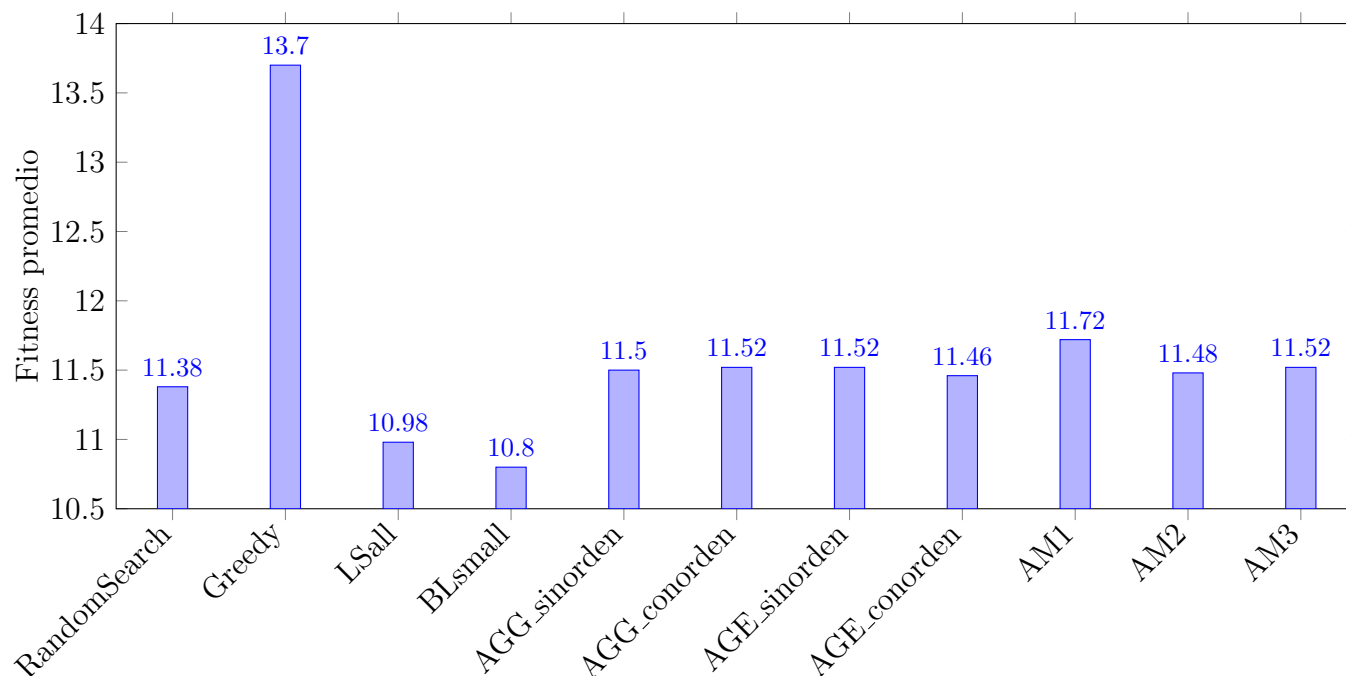


Figura 2: Comparación de fitness promedio por algoritmo en p2p-Gnutella05.txt

Este conjunto representa una red de intercambio de archivos P2P con 8846 nodos y 31839 enlaces. Esta estructura es más caótica, lo que podría afectar a la eficacia de algunos algoritmos. En este conjunto, Greedy vuelve a destacar claramente con fitness 13.7 y lo hace en una única evaluación. En segundo lugar se sitúa AM1 seguido de AM3, demostrando que los meméticos logran soluciones consistentes y de calidad, aunque a costa de tiempos de ejecución bastante superiores. Los algoritmos generacionales también tienen un rendimiento similar entre ellos.

Tabla 3: Resultados promedio de p2p-Gnutella08.txt

Algoritmo	Posición	Fitness	Tiempo (segs)	Evaluaciones
RandomSearch	9	11.36	$5.09 \times 10^{-1}$	1000
Greedy	1	12.60	$1.02 \times 10^{-2}$	1
LSall	10	11.36	$2.76 \times 10^{-1}$	1000
BLsmall	11	10.80	$1.64 \times 10^{-2}$	22.2
AGG_sinorden	7	11.42	$7.73 \times 10^{-1}$	1000
AGG_conorden	5	11.48	$3.28 \times 10^{-1}$	1000
AGE_sinorden	8	11.44	$9.42 \times 10^{-1}$	1000
AGE_conorden	4	11.50	$3.41 \times 10^{-1}$	1000
AM1	2	11.56	$4.54 \times 10^{-1}$	1000
AM2	3	11.44	$3.34 \times 10^{-1}$	1000
AM3	6	11.40	$3.36 \times 10^{-1}$	1000

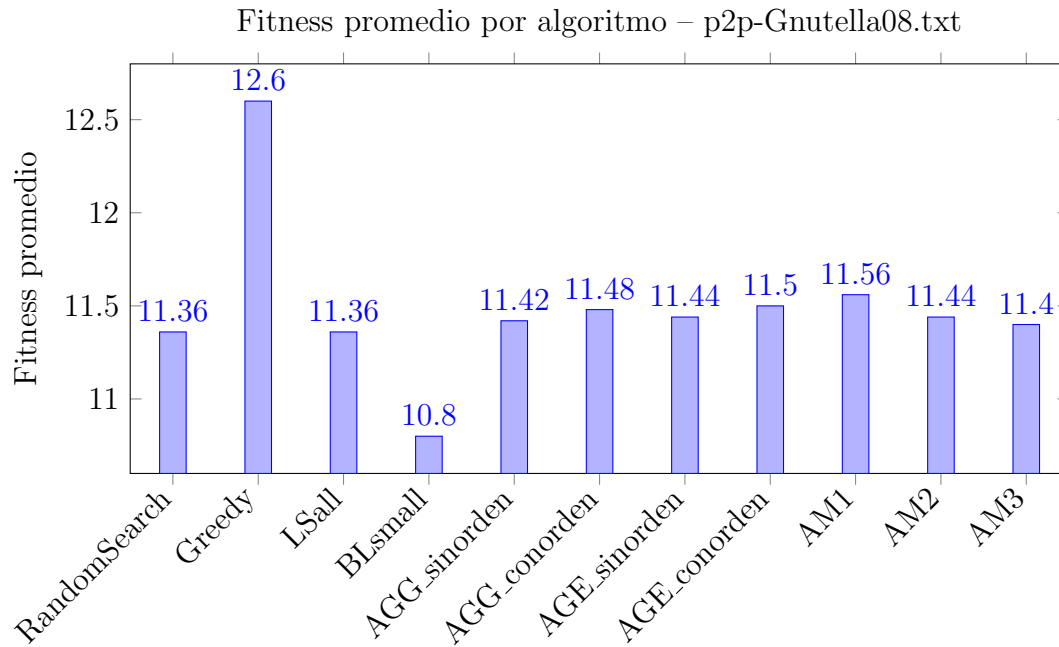


Figura 3: Comparación del fitness promedio de los algoritmos en el dataset p2p-Gnutella08.txt

Este conjunto corresponde a otra instancia de la red Gnutella, con 6301 nodos y 20777 enlaces. Es ligeramente más pequeña que la anterior. De nuevo Greedy es el algoritmo más eficaz en cuanto a calidad de la solución y eficiencia temporal. En cuanto a los algoritmos generacionales y meméticos: AM1 logra el segundo mejor resultado, superando a AM2 y AM3, sugiriendo que en este caso aplicar la búsqueda local a toda la población tiene el mejor impacto. AGE\_conorden y AGG\_sinorden destacan



entre los genéticos, sugiriendo que el uso de cruce con orden mejora ligeramente la calidad con respecto al sin orden.

Tabla 4: Resultados promedio de p2p-Gnutella25.txt

Algoritmo	Posición	Fitness	Tiempo (segs)	Evaluaciones
RandomSearch	7	11.14	$1.38 \times 10^0$	1000
Greedy	1	13.4	$4.62 \times 10^{-2}$	1
LSall	9	10.92	$2.84 \times 10^{-1}$	1000
BLsmall	10	10.8	$1.11 \times 10^{-1}$	36.2
AGG_sinorden	5	11.48	$1.65 \times 10^0$	1000
AGG_conorden	4	11.52	$4.45 \times 10^{-1}$	1000
AGE_sinorden	6	11.46	$2.14 \times 10^0$	1000
AGE_conorden	6	11.46	$4.77 \times 10^{-1}$	1000
AM1	2	11.54	$8.71 \times 10^{-1}$	1000
AM2	8	11.38	$5.70 \times 10^{-1}$	1000
AM3	6	11.44	$5.64 \times 10^{-1}$	1000

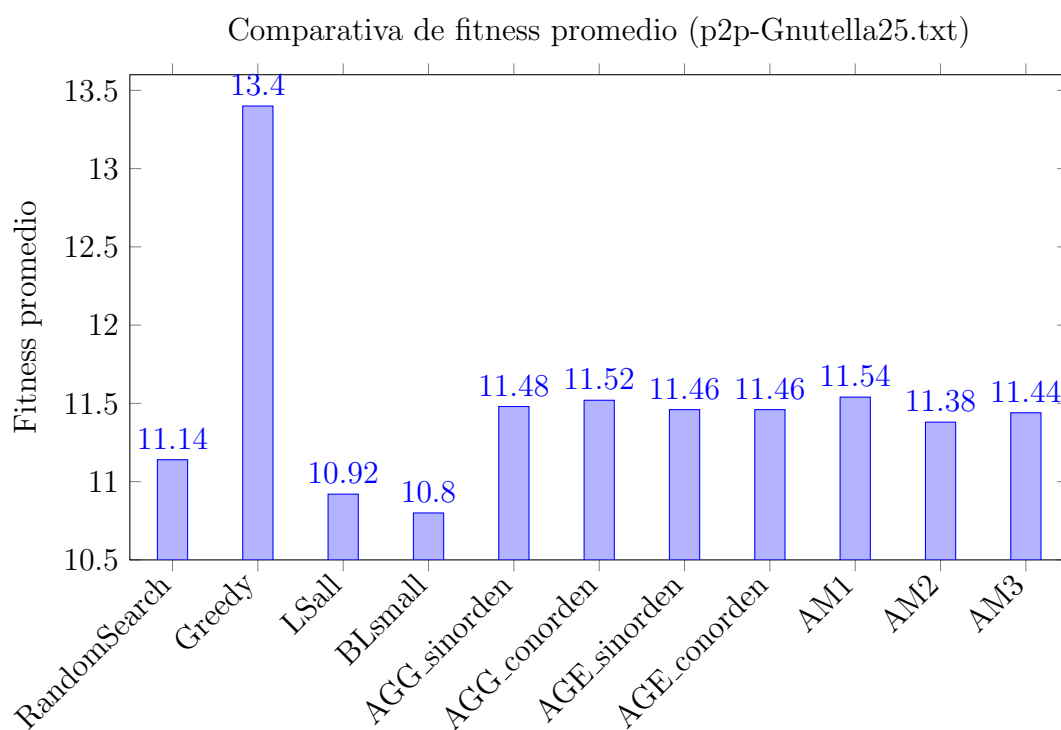


Figura 4: Comparación de fitness promedio por algoritmo en p2p-Gnutella25.txt

Esta instancia es la más grande analizada, con 22687 nodos y 54705 enlaces. Pero mantiene resultados similares a los anteriores casos, donde Greedy se consolida como la mejor estrategia para este tipo de red. Los algoritmos meméticos mantienen un rendimiento bastante bueno, especialmente AM1 que obtiene la segunda mejor posición, a pesar de requerir más tiempo. Los algoritmos genéticos también tienen buenos resultados. En particular, AGG\_conorden supera a su versión sin orden y a los estacionarios. Los algoritmos estacionarios tienen tiempos mucho más altos, lo que implican un mayor carga computacional sin una mejora significativa del fitness.

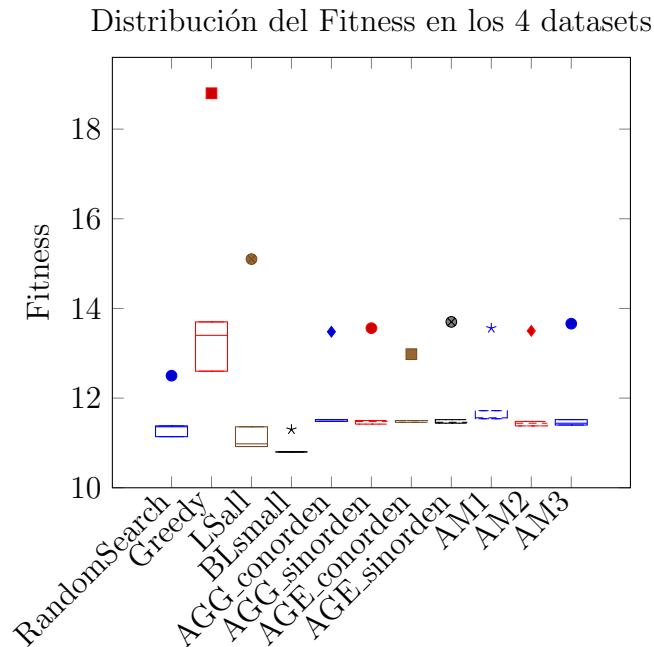


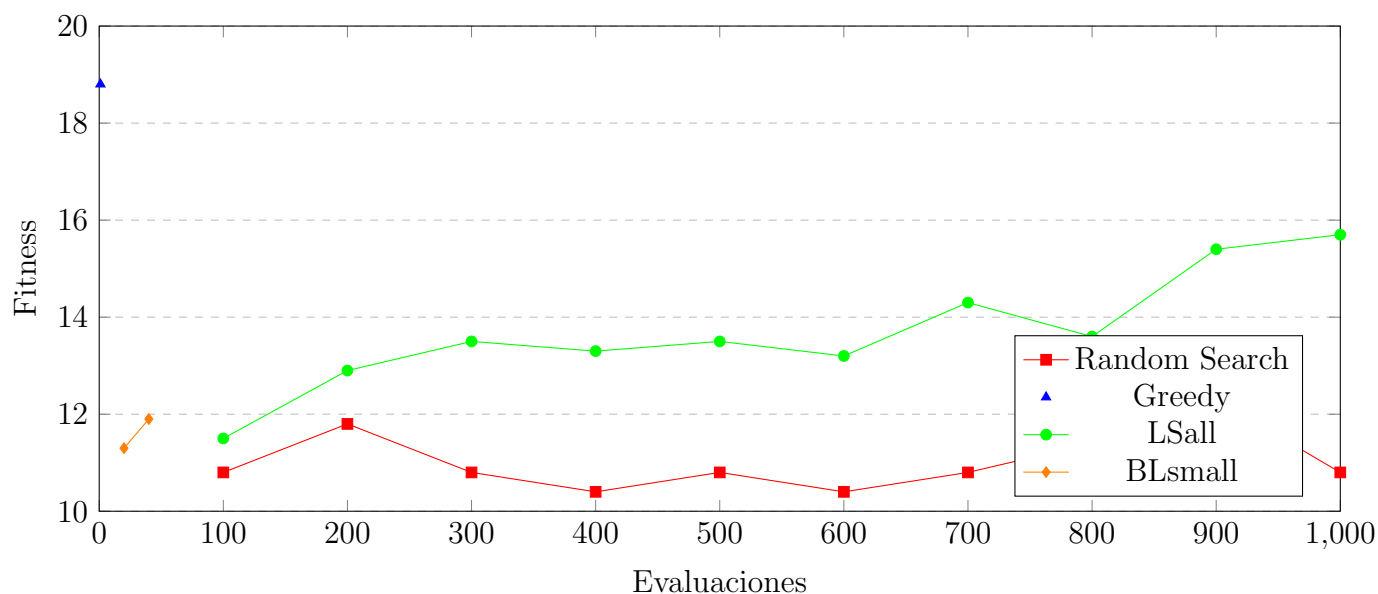
Figura 5: Boxplot del Fitness de todos los algoritmos en los 4 conjuntos de datos

El diagrama de caja muestra la distribución de los valores del fitness obtenidos por los 11 algoritmos. Observamos lo siguiente: Greedy sigue destacando como el algoritmo con el mayor valor de fitness, pero también presenta una dispersión muy amplia, incluyendo un outlier extremadamente alto. Esto sugiere que en algunos datasets puede encontrar soluciones muy buenas, aunque su rendimiento no es estable.

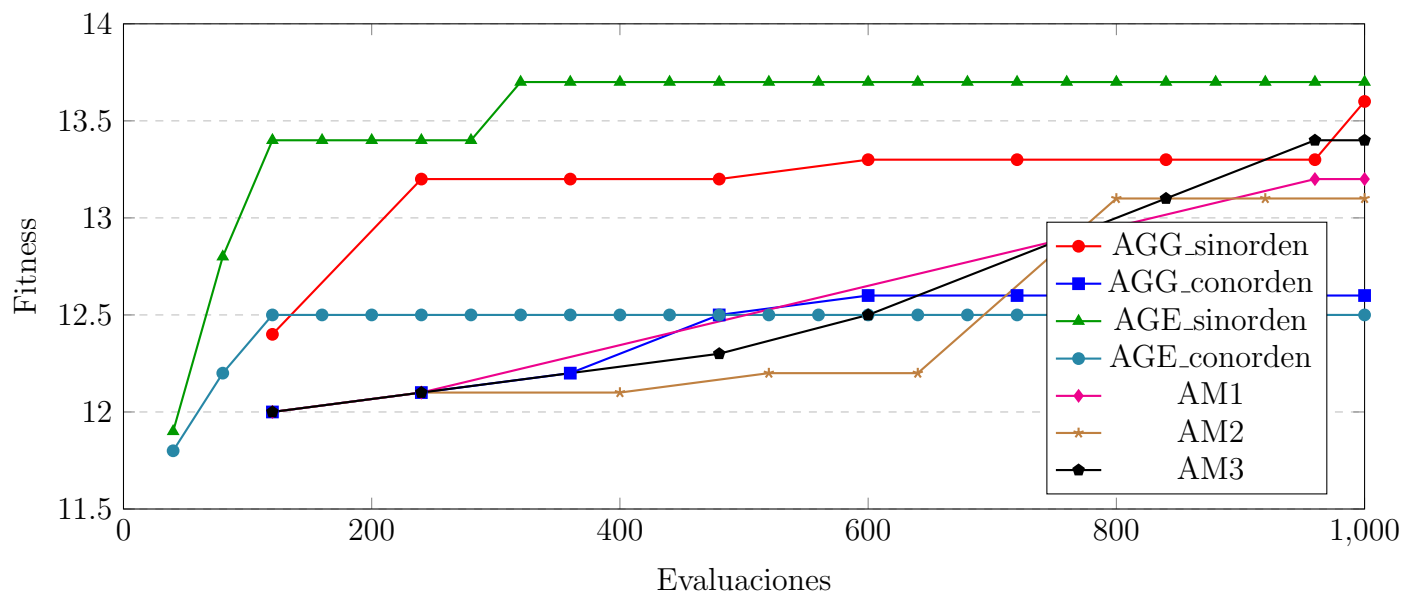
Los algoritmos nuevos presentan un rango de valores mucho más agrupado. AM1 logra una mediana elevada, siendo la mejor después de Greedy, pero con una dispersión mucho menor. Esto lo convierte en un método más fiable y eficaz.

AGE\_conorden tiene valores ligeramente superiores a los métodos simples de la práctica anterior, pero debajo del resto de algoritmos evolutivos.

Convergencia de Algoritmos (ca-GrQc.txt, Semilla 42)



Convergencia de algoritmos genéticos y meméticos (Semilla 43)



La gráfica muestra la convergencia de los algoritmos genéticos y meméticos aplicados al conjunto `ca-GRQc.txt` usando la semilla 43, donde el eje horizontal representa las evaluaciones realizadas y el eje vertical el valor del fitness.

La línea verde de `AGE_sinorden` destaca como la más efectiva, ya que alcanza rápidamente un fitness alto de 13.4 en solo 120 evaluaciones y sigue mejorando hasta estabilizarse en 13.7. Esta convergencia

tan rápida sugiere que la estrategia estacionaria es muy efectiva en ese conjunto de nodos. Este comportamiento tiene sentido ya que al ser un algoritmo estacionario, solo se generan dos descendientes por iteración y se selecciona el mejor entre padre e hijos. Esta estrategia evita la pérdida de buenas soluciones.

El AGG\_sinorden también muestra un buen rendimiento, alcanzando un fitness alto de 13.3 en unas 600 evaluaciones y manteniéndolo estable hasta el final. Al ser generacional, actualiza toda la población en cada iteración, aunque no se pierde el mejor gracias al elitismo, que permite una convergencia estable.

Los algoritmos genéticos con orden se estacan en valores mucho más bajos sin apenas mejora después de las primeras evaluaciones. Esto podría deberse a estar generando hijos demasiado similares.

Los algoritmos meméticos no llegan a converger tan claramente, van consiguiendo mejoras tras varias evaluaciones continuamente. Esto tiene sentido, ya que aplican búsqueda local cada 10 generaciones. AM3 es el que mejor resultado final alcanza, como era de esperar, ya que la búsqueda local sobre los mejores individuos refina aún más las buenas soluciones. AM2, al aplicar búsqueda local aleatoriamente, ofrece un resultado intermedio. Y, finalmente, AM1 aplica búsqueda sobre toda la población, lo cual es costoso y no parece tener la eficiencia que debería.

## 8.4. Análisis Final

Tabla 5: Tabla final de resultados

Algoritmo	Fitness Promedio	Tiempo Promedio (secs)	Evaluaciones Promedio
RandomSearch	11.595	$7.85 \times 10^{-1}$	1000
Greedy	14.625	$2.06 \times 10^{-2}$	1
LSall	11.5075	$2.28 \times 10^{-1}$	514.05
BLsmall	10.99	$2.94 \times 10^{-2}$	28.2
AGG_sinorden	11.99	$1.20 \times 10^0$	1000
AGG_conorden	12.00	$4.47 \times 10^{-1}$	1000
AGE_sinorden	12.03	$1.48 \times 10^0$	1000
AGE_conorden	11.85	$4.40 \times 10^{-1}$	1000
AM1	12.10	$7.18 \times 10^{-1}$	1000
AM2	11.95	$5.04 \times 10^{-1}$	1000
AM3	12.01	$4.92 \times 10^{-1}$	1000

Los resultados muestran claramente que el algoritmo Greedy sigue siendo el más efectivo, tanto en tiempo como en calidad de solución. Con una sola evaluación consigue un fitness promedio de 14.625, lo cual tiene mucho sentido, ya que selecciona directamente los nodos más influyentes de la red. En

este tipo de problemas, donde los nodos con alta conectividad suelen ser los más relevantes, una estrategia tan directa resulta muy potente.

En cambio, los algoritmos genéticos y meméticos consiguen buenos resultados, cerca de los 12 puntos de fitness. Aunque no alcanzan al Greedy, sí superan con claridad a métodos más básicos como Random Search, LSall o BLsmall. Dentro de este grupo, AM1 es el que mejor rendimiento ofrece, con un fitness promedio de 12.10, equilibrando bien la exploración global con la explotación local mediante búsqueda. También se aprecia que usar el cruce con orden (como en AGG\_conorden o AGE\_conorden) puede mejorar ligeramente los tiempos sin empeorar los resultados.

Por otro lado, Random Search cumple con lo esperado: obtiene resultados aceptables pero sin destacar, ya que se basa solo en la suerte. LSall, que aplica una búsqueda local intensiva, no logra resultados especialmente buenos, lo que sugiere que explorar únicamente soluciones vecinas no es suficiente en este problema. BLsmall, aunque muy rápido, tiene el peor fitness promedio. Esto se debe a su criterio de parada temprano y a que explora poco el espacio de soluciones.