UNIVERSIDAD DE GRANADA

Práctica 3: SNIMP

Metaheurísticas

Estudiante: Carmen Azorín Martí

DNI: 48768328W

Curso: 5º DGIIM

Correo: carmenazorin@correo.ugr.es

Prácticas: Lunes 15.30 - 17.30

Índice

Metaheurísticas
Práctica 3: 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 influencian.

El número total de soluciones posibles es combinacional, 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 exhaustica es impracticable. En nuestro caso, usaremos las siguientes metaheurísticas para el problema SNIMP, que es NP-completo:

- **Enfriamiento Simulado (ES).** Algoritmos probabilístico que simula el proceso de enfriamiento térmico, permitiendo aceptar soluciones peores en fases iniciales con el objetivo de escapar de óptimos locales.
- **Búsqueda Multiarranque Básica (BMB).** Algoritmo basado en generar múltiples soluciones aleatorias y aplicar sobre cada una de ellas una búsqueda local para intensificar la exploración.
- **Búsqueda Local Reiterada (ILS).** mejora la exploración aplicando sucesivas búsquedas locales sobre soluciones mutadas de la mejor encontrada hasta el momento.
- **Hibridación ILS-ES.** variante de ILS donde la intensificación se realiza mediante Enfriamiento Simulado en lugar de búsqueda local convencional.
- **GRASP.** Algoritmo greedy aleatorizado que guía la generación de soluciones mediante heurísticas, aplicada tanto sin búsqueda local (GRASP-NOBL) como combinada con búsqueda local (GRASP-SIBL).

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. Mutación de soluciones con porcentaje

La función de mutación con porcentaje implementa un operador que reemplada el x % de los nodos de la solución actual por otros nodos que no estén incluidos en ella.

```
Función mutate(solución, porcentaje):
    mutated ← copia de solution
    used ← conjunto de nodos en solution

    unusedNodes <- nodos que no están en la solución

    If unusedNodes está vacío:
        return mutated

    Calcular número de posiciones a mutar:
        numMutaciones = máximo(1, int(tamaño_solución * porcentaje))
        numMutaciones = mínimo(numMutaciones, tamaño_solución, tamaño_unusedNodes)

    Mezclar aleatoriamente los unusedNodes

    Para i desde 0 hasta numMutaciones - 1:
        pos <- índice aleatorio de la solución
        newNode <- nodo aleatorio no usado
        mutated[pos] <- newNode

    return mutated
```

2.7. 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.

2.8. 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. Enfriamiento Simulado (ES)

La clase ES implementa el algoritmo de Enfriamiento Simulado basado en el proceso físico de enfriamiento de metales, en el cual un material caliente se enfría lentamente para alcanzar un estado de mínima energía. Este algoritmo permite, en las primeras fases de la búsqueda, aceptar soluciones peores con cierta probabilidad, ara explorar el espacio de soluciones y evitar estancarse en óptimos locales.

El algoritmo empieza con una solución aleatoria inicial, y se genera un número controlado de vecinos por cada valor de la temperatura. La temperatura va disminuyendo a medida que avanza el algoritmo, reduciendo la probabiliad de aceptar soluciones peores. Y se detiene cuando no se aceptan vecinos en un ciclo de enfriamiento o se alcanza el número máximo de evaluaciones.

```
Función EnfriamientoSimulado(problem, maxevals)
    s ← Generar solución inicial aleatoria
    fs ← fitness(s)
    bestSol ← s
    bestFitness ← fs
    evals ← 1

    T0 ← (mu * fs) / (-ln(phi))
```

```
Si T0 <= Tf, entonces T0 ← 10 * Tf
beta ← (T0 - Tf) / (M * T0 * Tf)
T ← T0

Para k = 1 hasta M y mientras evals < maxevals hacer
nVecinos ← 0
nExitos ← 0

Mientras nVecinos < máx_vecinos y nExitos < máx_éxitos y evals < maxevals hacer
  s' ← mutar(s)
  f' ← fitness(s')
  evals ← evals + 1
  nVecinos ← nVecinos + 1
  Delta ← fs - f'

  Si (Delta < 0) o (rand(0,1) <= exp(-Delta / T)) entonces
    s ← s'
    fs ← f'
    nExitos ← nExitos + 1

    Si fs > bestFitness entonces
      bestSol ← s
      bestFitness ← fs
    Fin Si
  Fin Si
Fin Mientras

T ← T / (1 + beta · T)
Si nExitos == 0 entonces terminar
Fin Para

devolver (bestSol, bestFitness)
```

Los pasos del algoritmo son:

1. Se genra una solución inicial aleatoria s con fitness $f(s)$. Y se calcula la temperatura inicial

$$T_0 = \frac{\mu \cdot f(s)}{-\ln(\phi)}$$

donde μ es el porcentaje de empeoramiento tolerado y ϕ la probabilidad de aceptarlo.

2. Fijamos una temperatura final T_f muy pequeña y establecemos un número de niveles de enfriamiento $M = \frac{total\_evaluaciones}{max\_vecinos}$. Y calculamos el parámetro de enfriamiento Cauchy modificado
$$\beta = \frac{T_0 - T_f}{M \cdot T_0 \cdot T_f}$$

3. Entramos en un bucle principal que se repetirá hasta que la temperatura llegue hasta T_f o se agote el número de evaluaciones:

   a) Inicializamos contadores de $nVecinos$ y $nExitos$

   b) Mientras no superemos el máximo de vecinos ni el máximo de éxitos: generamos un vecino aleatorio s' con fitness $f(s')$ y calculamos $\Delta = f(s) - f(s')$. Si $f(s') > f(s)$ aceptamos s' como nueva solución y, en caso contrario, aceptamos con probabilidad $exp(-\Delta/T)$. Y si se acepta, incrementamos el número de éxitos.

   c) Siempre tenemos que actualizar la temperatura $T = \frac{T}{1 + \beta T}$

   d) Y si no se ha aceptado ningún vecino, es decir, no hay éxitos, entonces termina.

4. Devolvemos la mejor solución encontrada.

Este algoritmo permite explorar el espacio globalmente al principio (temperatura alta) y explota sus soluciones buenas hasta el final (temperatura baja). La aceptación de soluciones peores al principio puede ayudar a escapar de óptimos locales.

5. Búsqueda Multiarranque Básica (BMB)

La clase BMB implementa el algoritmo de Búsqueda Multiarranque Básica. Este algoritmo en lugar de depender de una única búsqueda local que podría estancarse en un óptimo local, repite la búsqueda local desde diferentes soluciones iniciales, aumentando la probabilidad de alcanzar mejores soluciones globales.

```
Función BMB(problem, maxevals)
    bestSol ← vacío
    bestFitness ← -infinito
    evals_total ← 0

    Repetir 10 veces o hasta alcanzar maxevals:
    s0 ← generar solución aleatoria
    f0 ← fitness(s0)
```

```
    evals_total ← evals_total + 1

    s_opt ← aplicar Búsqueda Local sobre s0 (máximo 100 evaluaciones)
    f_opt ← fitness(s_opt)
    evals_total ← evals_total + evaluaciones usadas en BL

    Si f_opt > bestFitness entonces
      bestSol ← s_opt
      bestFitness ← f_opt

    devolver (bestSol, bestFitness)
```

El procedimiento consiste en realizar un número fijo de reinicios (10 en nuestro caso). Y en cada reinicio:

1. Se genera una solución inicial aleatoria.
2. Se aplica una búsqueda local con un máximo de 100 evaluaciones.
3. Se guarda la mejor solución obtenida entre todas las ejecuciones.

Finalmente, se devuelve la mejor solución global.

Este algoritmo funciona porque cada solución aleatoria parte de un punto distinto del espacio. La búsqueda local explota la zona cercana. Al repetir este proceso, se evitan los malos máximos locales y se explora más que con solo una búsqueda.

6. Búsqueda Local Reiterada (ILS)

La clase ILS implementa el algoritmo de Búsqueda Local Reiterada. Parte de una solución inicial aleatoria, la mejora con una búsqueda local, y luego genera nuevas soluciones mutando la mejor solución encontrada hasta el momento, aplicando de nuevo búsqueda local sobre cada mutación.

```
Función ILS(problem, maxevals)
    s0 ← generar solución inicial
    f0 ← fitness(s0)
    evals_total ← 1

    s_best ← búsquedaLocal(s0, f0, 100)
    f_best ← fitness(s_best)
    evals_total ← evals_total + evaluaciones de BL
```

```
Repetir 9 veces y mientras evals_total < maxevals:
s_mut ← mutar(s_best,0.2)
f_mut ← fitness(s_mut)
evals_total ← evals_total + 1

s_opt ← búsquedaLocal(s_mut, f_mut, 100)
f_opt ← fitness(s_opt)
evals_total ← evals_total + evaluaciones de BL

Si f_opt > f_best entonces
  s_best ← s_opt
  f_best ← f_opt

devolver (s_best, f_best)
```

El proceso consiste en:

1. Crear una solución inicial y evaluarla.
2. Aplicamos búsqueda local a dicha solución inicial (BLsmall) y se guarda esta solución como la mejor encontrada hasta el momento.
3. Se realizan 9 iteraciones, donde:
   a) Mutamos la solución un 20 %, para ello usamos el nuevo método de la clase `Snimp`.
   b) Aplicamos la búsqueda local a la solución mutada.
   c) Actualizamos la mejor solución si la mejora.
4. Finalmente, devolvemos la mejor solución encontrada,

La diferencia con BMB es que ILS siempre parte de la mejor solución hallada y la perturba, aprovechando la información anterior. BMB, sin embargo, reinicia totalmente en cada intento. Por lo que se puede ocnsiderar que ILS es más progresivo, aunque BMB es más diverso.

7. Hibridación de ILS y ES (ILS-ES)

La clase ILS-ES implementa la hibridación de ILS con Enfriamiento Simulado. Lo que hace es matiene las iteraciones del ILS (10 ciclos de intensificación), pero sustituye la búsqueda local por Enfriamiento Simulado para mejorar.

```
Función ILS-ES(problem, maxevals)
    s0 ← generar solución inicial
    f0 ← fitness(s0)
    evals_total ← 1

    s_best ← aplicar ES sobre s0 con máx. 100 evaluaciones
    f_best ← fitness(s_best)
    evals_total ← evals_total + evaluaciones de ES

    Repetir 9 veces y mientras evals_total < maxevals:
    s_mut ← mutar(s_best,0.2)
    f_mut ← fitness(s_mut)
    evals_total ← evals_total + 1

    s_opt ← aplicar ES sobre s_mut con máx. 100 evaluaciones
    f_opt ← fitness(s_opt)
    evals_total ← evals_total + evaluaciones de ES

    Si f_opt > f_best entonces
      s_best ← s_opt
      f_best ← f_opt

    devolver (s_best, f_best)
```

Lo que hace el algoritmo es generar una solución aleatoria inicial y evaluarla. A continuación, aplica Enfriamiento Simulado a esta solución para obtener una versión optimizada y la guarda como la mejor solución actual. Durante 9 iteraciones:

1. Aplica una mutación sobre la mejor solución (cambiando el 20 % de los elementos).

2. Aplicar ES a la solución mutada.

3. Si mejora la mejor solución conocida, se actualiza.

El proceso termina tras 10 aplicaciones de ES o cuando se alcanza el número máximo de evaluabaciones.

El algoritmo alterna entre exploración global (por las mutaciones) y explotación intensiva (por el ES). El ILS se encarga de explorar nuevos entornos, asegurando diversidad. Por otro lado, el ES profundiza en cada entorno, permitiendo aceptar incluso soluciones peores de forma controlada gracias a la temperatura.

8. GRASP sin BL

La clase GRASP_NOBL implementa el algoritmo GRASP sin búsqueda local, que genera múltiples soluciones usando la función heurística.

```
Función GRASP-NOBL(problem, maxevals)
  bestSol ← vacío
  bestFitness ← -infinito
  evals_total ← 0

  Repetir 10 veces o hasta maxevals:
    sel ← vacío
    candidatos ← nodos()
    sel ← {nodo aleatorio}
    candidatos ← candidatos - sel

    Mientras |sel| < m:
      Calcular heurísticas h_i de todos los nodos en candidatos
      heurMax ← max(h_i), heurMin ← min(h_i)
      umbral ← heurMax - rand() * (heurMax - heurMin)

      LRC ← {nodos con h_i >= umbral}
      nodo ← nodo aleatorio en LRC
      sel ← sel union {nodo}
      candidatos ← candidatos - {nodo}

    fitness ← fitness(sel)
    evals_total ← evals_total + 1

    Si fitness > bestFitness:
      bestFitness ← fitness
      bestSol ← sel

  devolver (bestSol, bestFitness)
```

El algoritmo repite el proceso 10 veces:

1. Se construye una solución de manera greedy aleatorizada.
2. Se evalúa la solución generada.
3. Se actualiza la mejor solución si hay mejora.

A continuación, se devuelve la mejor de las 10 soluciones.

La construcción greedy aleatorizada se hace en un método auxiliar, donde:

1. Se empieza con un nodo inicial aleatorio.
2. En cada paso, se calculan heurísticas de los nodos no seleccionados y se identifican el valor máximo y mínimo.
3. Se genera una Lista Restringida de Candidatos (LRC) con los nodos cuya heurística supera un cierto umbral.
$$umbral = heurMax - rand() \cdot (heurMax - heurMin)$$
4. Se selecciona aleatoriamente un nodo desde la LRC y se añade a la solución.
5. Se repite hasta completar la solución.

Este algoritmo explora el espacio de soluciones gracias a la aleatoriedad de la construcción. Greedy con aleatoriedad le permite escapar de soluciones malas sin necesidad de búsqueda local. Además, al no hacer refinamiento, es más rápido, aunque puede quedarse lejos del óptimo.

9. GRASP con BL

La clase GRASP_SIBL implementa el GRASP con búsqueda local, igual que en anterior pero intensificando mediante búsqueda local.

```
Función GRASP-SIBL(problem, maxevals)
  bestSol ← vacío
  bestFitness ← -infinito
  evals_total ← 0

  Repetir 10 veces o hasta maxevals:
    sel ← vacío
    candidatos ← nodos()
    sel ← {nodo aleatorio}
    candidatos ← candidatos - sel

    Mientras |sel| < m:
      Calcular heurísticas h_i de todos los candidatos
      heurMax ← max(h_i), heurMin ← min(h_i)
      umbral ← heurMax - rand() * (heurMax - heurMin)
      LRC ← {nodos con h_i >= umbral}
```

```
      nodo ← nodo aleatorio en LRC
      sel ← sel union {nodo}
      candidatos ← candidatos - {nodo}

    fitness ← fitness(sel)
    evals_total ← evals_total + 1

    sel_opt ← búsquedaLocal(sel, fitness, 100)
    evals_total ← evals_total + evaluaciones de BL

    Si fitness(sel_opt) > bestFitness:
      bestSol ← sel_opt
      bestFitness ← fitness(sel_opt)

  devolver (bestSol, bestFitness)
```

El algoritmo repite 10 veces:

1. Se construye una solución greedy aleatorizada y se evalúa la solución.
2. Se aplica búsqueda local a la solución.
3. Se guarda la mejor solución encontrada.

Y se devuelve la mejor solución obtenida tras las 10 construcciones.

Esta variante refina las soluciones y mejora la calidad.

10. Estructura del código

El código está estructurado como la plantilla proporcionada por el profesor en `https://github.com/dmolina/template_mh`.A continuación se decribe 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. Ademñas, 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 careta `inc/` se han añadido los siguientes archivos:
    - `ES.h` que contiene la implementación del Enfriamiento Simulado.
    - `BMB.h` que contiene la implementación de la Búsqueda Multiarranque Básica.
    - `ILS.h` que contiene la implementación de la Búsqueda Local Reiterada.
    - `ILS_ES.h` que contiene la implementación de la hibridación de ILS y ES.
    - `GRASP_SIBL.h` que contiene la implementación de GRASP con Búsqueda Local.
    - `GRASP_NOBL.h` que contiene la implementación de GRASP sin Búsqueda Local.
- 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.

10.1. 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 (ES, BMB, ILS, ILS_ES, GRASP_SIBL, GRASP_NOBL), guarda sus resultados:

```
seed algoritmo ejecucion best_solution best_fitness evaluations time_ms
43 ES 1 "1071 1301 2398 3668 284 101 4019 269 301 277" 15.2 687 1313
43 BMB 1 "2046 1273 4809 3745 2391 3670 1188 291 280 3813" 12.8 348 342
```

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`.

11. Experimentos y análisis de resultados

11.1. 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. El Enfriamiento Simulado tiene los siguientes parámetros:

- Temperatura inicial
$$T_0 = \frac{\mu Coste(S_0)}{-Lin(\phi)}$$
con $\mu = 0.2$ y $\phi = 0.3$. Se ajusta para que $T_0 > T_f = 10^{-3}$.
- Enfriamiento (Cauchy modificado):
$$T_{k+1} = \frac{T_k}{1 + \beta T_k} \text{ donde } \beta = \frac{T_0 - T_f}{M T_0 T_f}$$
con M =número de ciclos = $\frac{maxevals}{max_vecinos}$.
- Condición de parada interna $L(T)$:
  - Se generan un máximo de $max\_vecinos = 5m$ vecinos por temperatura.
  - Se aceptan un máximo de $max\_exitos = 0.1 max\_vecinos$.
- Condición de parada global: sin éxitos en un enfriamiento o límite de evaluaciones.

La Búsqueda Multiarranque Básica utiliza los siguientes parámetros:

- Número de reinicios: 10.
- Evaluaciones por búsqueda local: hasta 1000 evaluaciones, o parada anticipada si no hay mejora en el entorno o se alcanzan 20 intentos sin mejora (BLsmall).
- Evaluaciones total máximas: maxevals, que son 1000.

El ILS utiliza los siguiente parámetros:

- Número de iteraciones: 10 (1 inicial + 9 mutaciones).
- Evaluaciones por búsqueda local: hasta 100 por iteración y parada anticipada si no hay mejora en 20 intentos.

- Mutación: se reemplaza el 20 % (mínimo 2) de los elementos de la solución por elementos no seleccionados.

La hibridación ILS-ES tiene los siguientes parámetros:

- Número de iteraciones: 10 (1 inicial + 9 mutaciones).
- Mutación igual que ILS.
- Durante la intensificación se sustituye la búsqueda local por ES.

El GRASP tiene los siguientes parámetros:

- Número de construcciones: 10.
- Evaluaciones: 10 sin BL y maxevals (1000) con BL.

11.2. Análisis de cada caso

Se han utilizado cuatro conjuntos de datos reales extraídos de la colección Standford 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	5.72×10^{-1}	1000
Greedy	1	18.8	1.10×10^{-2}	1
LSall	4	15.1	7.08×10^{-1}	1000
BLsmall	11	11.3	2.58×10^{-2}	28.6
AM1	6	13.56	8.47×10^{-1}	1000
ES	5	13.76	7.30×10^{-1}	444.0
BMB	8	12.60	2.60×10^{-1}	303.4
ILS	7	12.78	3.10×10^{-1}	320.2
ILS-ES (COMB)	9	12.52	2.50×10^{-1}	168.4
GRASP_NOBL	3	17.22	4.10×10^{-1}	10.0
GRASP_SIBL	2	17.80	8.10×10^{-1}	308.0

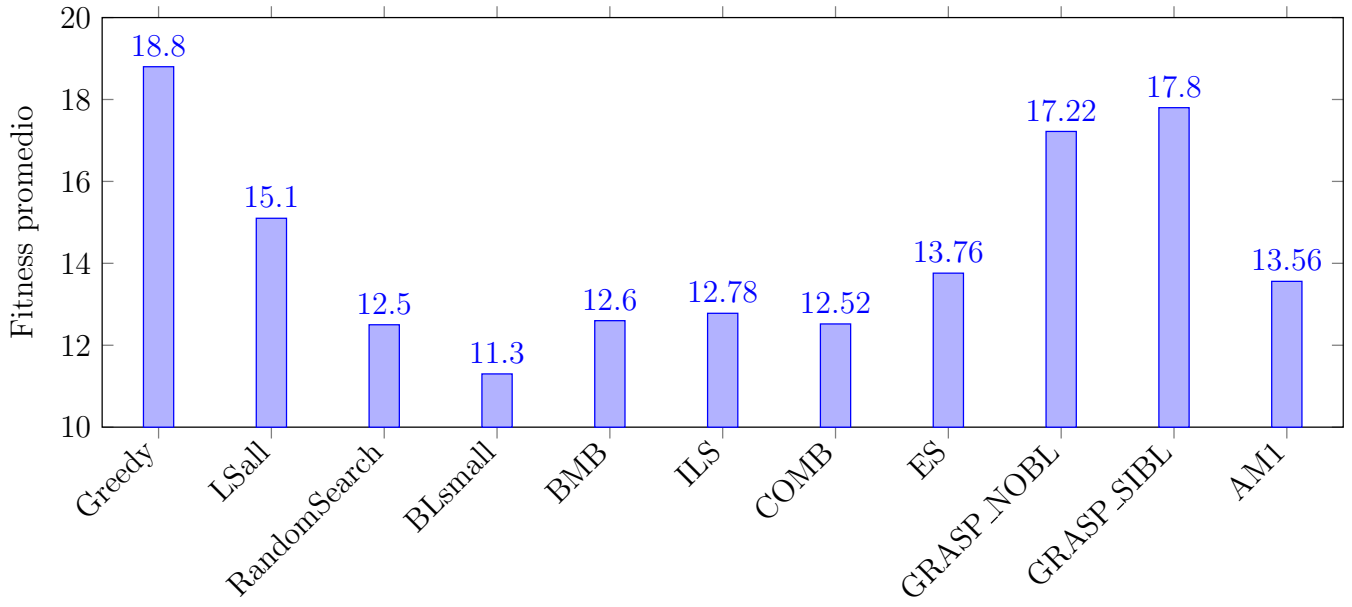


Figura 1: Fitness promedio por algoritmo en el conjunto SNIMP

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.

Vemos que Greedy alcanza el mejor fitness con una única evaluación y en el tiempo más bajo. Esto indica que la heurística de es muy buena para el problema SNIMP, pero no quiere decir que Greedy vaya a ser el mejor en todos los problemas. Por otro lado, tenemos los algoritmos GRASP, tanto con como sin búsqueda local. Esto es lo esperado, ya que GRASP combina una constraucción aleatorizada guiada con una heurística.

También tenemos las búsquedas locales que, de media, son mejores que la búsqueda aleatoria, pero no alcanzan resultados destacables. Esto, como ya explicamos en la primera práctica, puede deberse a una falta de exploración del espacio de soluciones, debido a una condición de parada demasiado restrictiva.

Los métodos basados en trayectoria, como ES, BMB y ILS, presentan un buen rendimiento. ES realiza un número de evaluaciones inferior al máximo y alcanzando un gran fitness, lo que muestra que ajusta la exploración en función del progreso. En cambio, ILS e ILS-ES no superan por demasiado la búsqueda aleatoria ni BMB, probablemente debido a una limitada capacidad de mejora por la búsqueda local ineficaz.

Tabla 2: Resultados promedio de p2p-Gnutella05.txt

Algoritmo	**Posición**	**Fitness**	**Tiempo (segs)**	**Evaluaciones**
RandomSearch	5	11.38	6.80×10^{-1}	1000
Greedy	1	13.7	1.48×10^{-2}	1
LSall	9	10.98	2.18×10^{-1}	1000
BLsmall	11	10.8	3.48×10^{-2}	25.4
AM1	4	11.72	6.99×10^{-1}	1000
ES	6	11.14	2.80×10^{-1}	142.4
BMB	10	10.92	2.40×10^{-1}	228.0
ILS	7	11.02	2.30×10^{-1}	227.8
ILS-ES (COMB)	8	11.00	2.30×10^{-1}	119.4
GRASP_NOBL	3	12.94	6.30×10^{-1}	10.0
GRASP_SIBL	2	12.98	8.60×10^{-1}	231.8

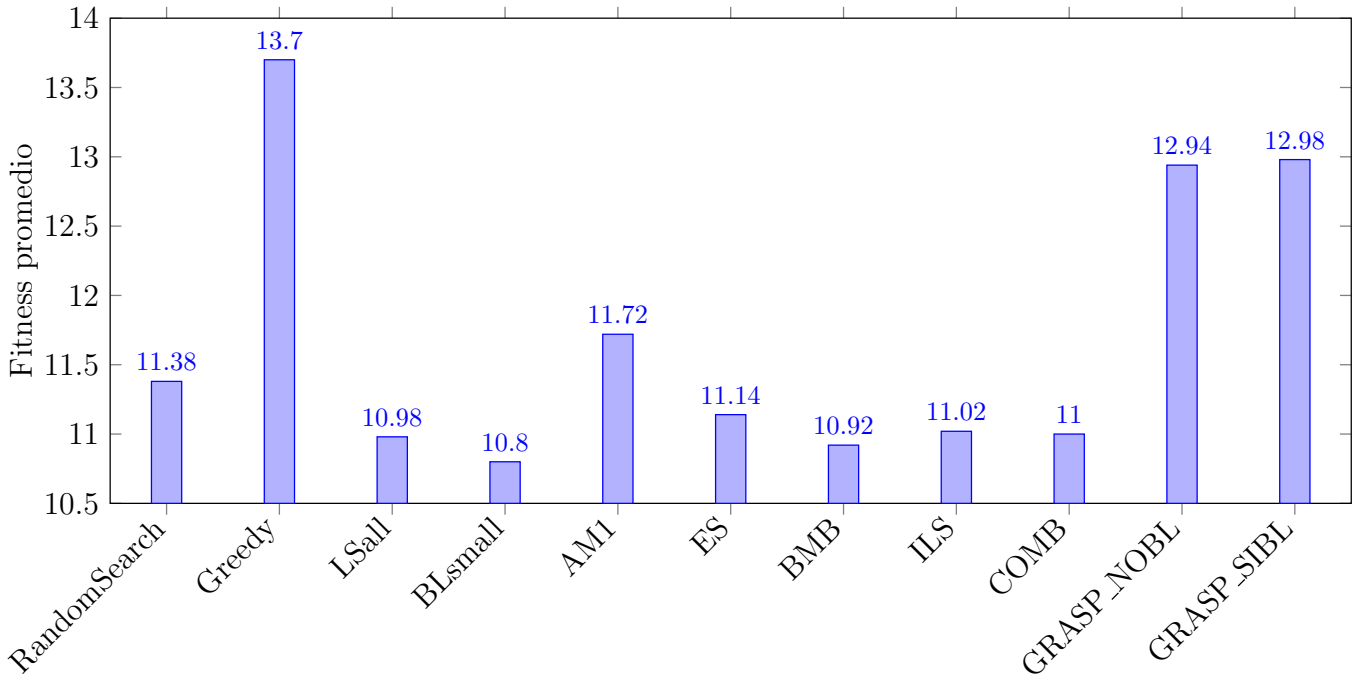


Figura 2: Comparación de fitness promedio por algoritmo en `p2p-Gnutella05.txt`

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.

El algoritmo Greedy vuelve a quedarse con la mejor solución usando solo una evaluación. Lo cual tiene

sentido porque el problema tiene una estructura que favorece las decisisones rápidas bien elegidas. Aunque, como hemos dicho anteriormente, no tiene que ser así para todos los problemas.

Los dos algoritmos GRASP también destacan. GRASP_SIBL alcanza casi el mismo fitness que Greedy y lo hace en 230 evalucioanes, que es considerablemente bueno en términos de eficiencia. GRASP_NOBL se queda un poco por debajo, pero con solo 10 evaluaciones, lo cual demuestra que la construcción aleatoriazada es muy eficaz por sí sola.

Más abajo aparecen los métodos meméticos y basados en trayectorias como ES, BMB o ILS, que aunque consumen bastantes evaluaciones, se quedan en torno a un fitness de 11. Esto indica que no están explotando bien las soluciones o que su búsqueda local no es lo suficientemente eficaz. En particular, AM1, a pesar de gastar 1000 evaluaciones, no mejora mucho frente a métodos más simples.

Las búsquedas locales tienen el peor rendimiento, lo cual es lógico porque dependen de la calidad de la solución inicial y, en el caso de BLsmall, el bajo número de evaluaciones limita mucho.

Tabla 3: Resultados promedio de p2p-Gnutella08.txt

Algoritmo	**Posición**	**Fitness**	**Tiempo (segs)**	**Evaluaciones**
RandomSearch	5	11.36	5.09×10^{-1}	1000
Greedy	1	12.60	1.02×10^{-2}	1
LSall	5	11.36	2.76×10^{-1}	1000
BLsmall	11	10.80	1.64×10^{-2}	22.2
AM1	4	11.56	4.54×10^{-1}	1000
ES	7	11.06	2.10×10^{-1}	147.2
BMB	8	11.02	1.80×10^{-1}	235.0
ILS	10	10.92	1.60×10^{-1}	219.8
ILS-ES (COMB)	9	10.94	1.80×10^{-1}	119.8
GRASP_NOBL	2	12.40	4.50×10^{-1}	10.0
GRASP_SIBL	2	12.40	6.40×10^{-1}	241.4

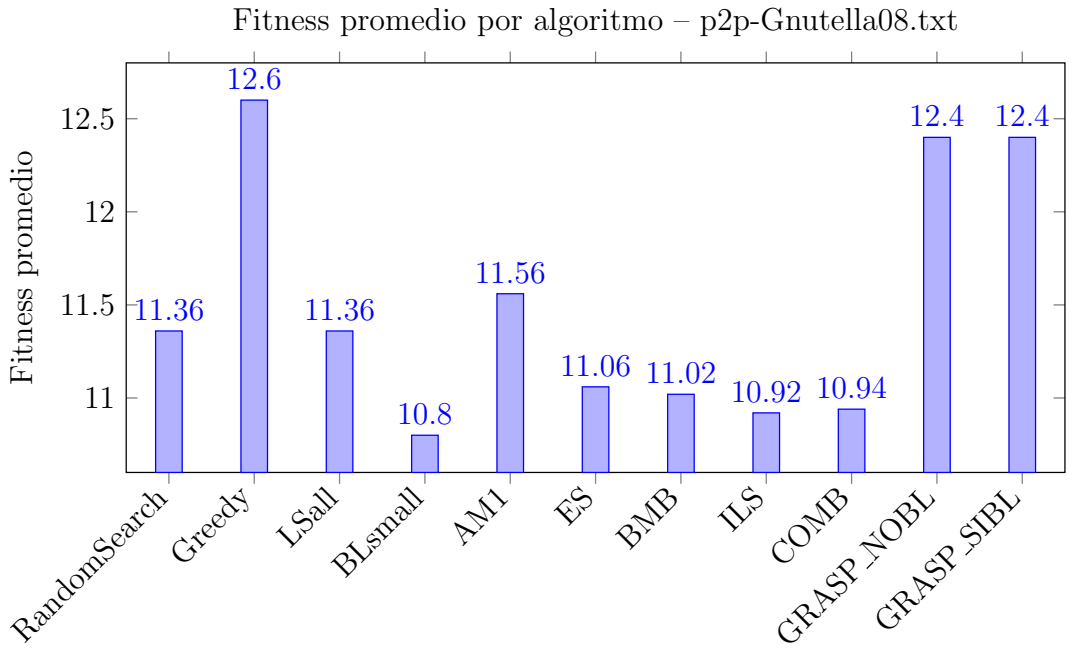


Figura 3: Comparación del fitness promedio de los algoritmos en el dataset `p2p-Gnutella08.txt`

Este conjunto corresponde a otra instancia de la red Gnutella, con 6301 nodos y 20777 enlaces. Es ligeramente más pequeña que la anterior.

Volvemos a obtener el patrón anteior: el algoritmo Greedy consigue el mejor resultado con una sola evaluación y apenas unas milésimas de segundo. Lo que refuerza la idea de que la heurística encaja

muy bien con la instancia del problema.

Los métodos GRASP, ambos tiene un fitness de 12.40. Lo interesante es que GRASP_NOBL lo consigue solo con 10 evaluaciones, a diferencia de GRASP_SIBL que necesita 240. En cualquier caso, confirman que la construcción guiada por heurística es muy efectiva.

En el resto de algoritmos, el rendimiento es bastante bajo. AM1 es el mejor entre ellos, aunque no puede llegar a competir con los tres primeros.

Tabla 4: Resultados promedio de p2p-Gnutella25.txt

Algoritmo	**Posición**	**Fitness**	**Tiempo (segs)**	**Evaluaciones**
RandomSearch	5	11.14	1.38×10^{0}	1000
Greedy	1	13.4	4.62×10^{-2}	1
LSall	8	10.92	2.84×10^{-1}	1000
BLsmall	11	10.8	1.11×10^{-1}	36.2
AM1	4	11.54	8.71×10^{-1}	1000
ES	6	11.12	6.20×10^{-1}	139.6
BMB	8	10.92	6.80×10^{-1}	280.0
ILS	7	10.94	4.70×10^{-1}	237.0
ILS-ES (COMB)	8	10.92	5.30×10^{-1}	120.0
GRASP_NOBL	3	13.06	1.57×10^{0}	10.0
GRASP_SIBL	2	13.15	2.49×10^{0}	230.5

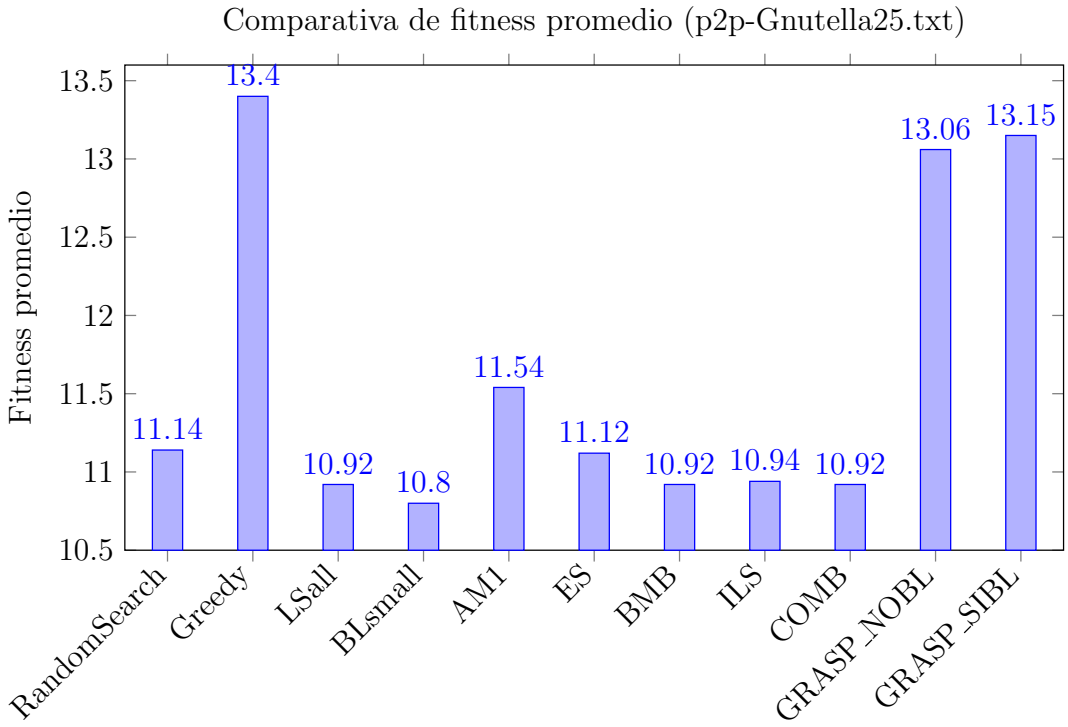


Figura 4: Comparación de fitness promedio por algoritmo en `p2p-Gnutella25.txt`

Esta instancia es la más grande analizada, con 22687 nodos y 54705 enlaces.

Volvemos a repetir el patrón anterir, el algoritmo Greedy obtiene el mejor resultado con una so-

la evaluación. Justo después estan GRASP en sus dos versiones, con fitness también muy altos, confirmando que construir soluciones de forma heurística es muy efectivo.

Los algoritmos complejos como AM1 o ES no logran acercarse a los anteriores resultados, a pesar de usar muchas más evaluaciones. Lo mismo pasa con ILS, BMB e ILS-ES, que se mueven alrededor de fitness 10.9-11.1.

Finalmente, las búsquedas locales no obtienen buenos resultados, llegando a ser peores que la búsqueda aleatoria. Esto se debe a las soluciones iniciales aleatorias y al estancamiento en ellas.

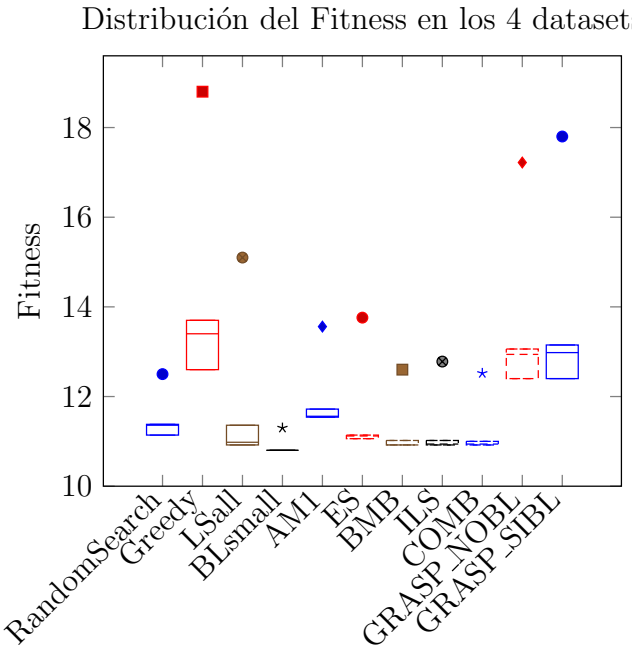


Figura 5: Boxplot del Fitness de todos los algoritmos en los 4 conjuntos de datos

El diagrama de caja muestra la distribución de los valores del fitness obtenidos por los 11 algoritmos. El eje X contiene los nombres de los algoritmos evaluados, mientras que el eje Y muestra el valor de fitness alcanzado. Cada algoritmo tiene varios puntos asociados, representando el fitness obtenido en cada uno de los cuatro conjuntos de datos. Finalmente, las cajas indican la dispersión del fitness en cada algoritmo.

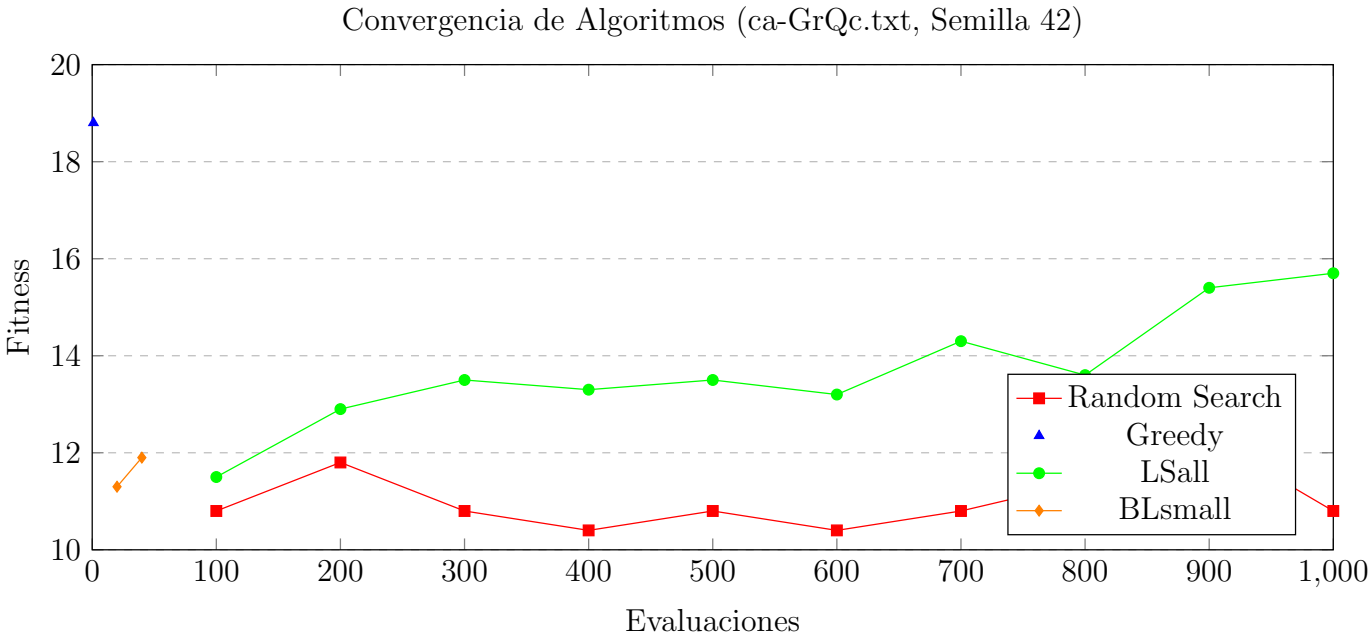
Greedy y GRASP dominan claramente, obteniendo los mejores valores de fitness, rondando 18 en algunos casos. Esto se debe a que la heurística funciona especialmente bien en este problema SNIMP, y GRASP logra buenos resultados manteniendo diversidad.

La búsqueda aleatoria no tiene mucha varianza, aunque se esperaría que la tuviese debido a su

naturaleza aleatoria. A veces podría encontrar soluciones decentes, pero no de forma fiable.

Los algoritmos ES, BMB, ILS y ILS-ES muestran un rendimiento modesto. Todos se sitúan en una franja de fitness baja (entre 10.9 y 11.5 aproximadamente), bastante alejada de GRASP. Además, su dispersión es muy baja, lo que indica que tienden a quedarse atrapados en zonas del espacio de soluciones de menor calidad.

Entre ellos, ES es el que obtiene los resultados ligeramente más altos y variados, lo que sugiere que su esquema de enfriamiento le permite escapar de óptimos locales en algunos casos. Sin embargo, ILS, ILS-ES y BMB tienden a converger hacia soluciones similares y más limitadas, lo que podría deberse a una falta de diversidad en las soluciones iniciales o mutaciones poco disruptivas.



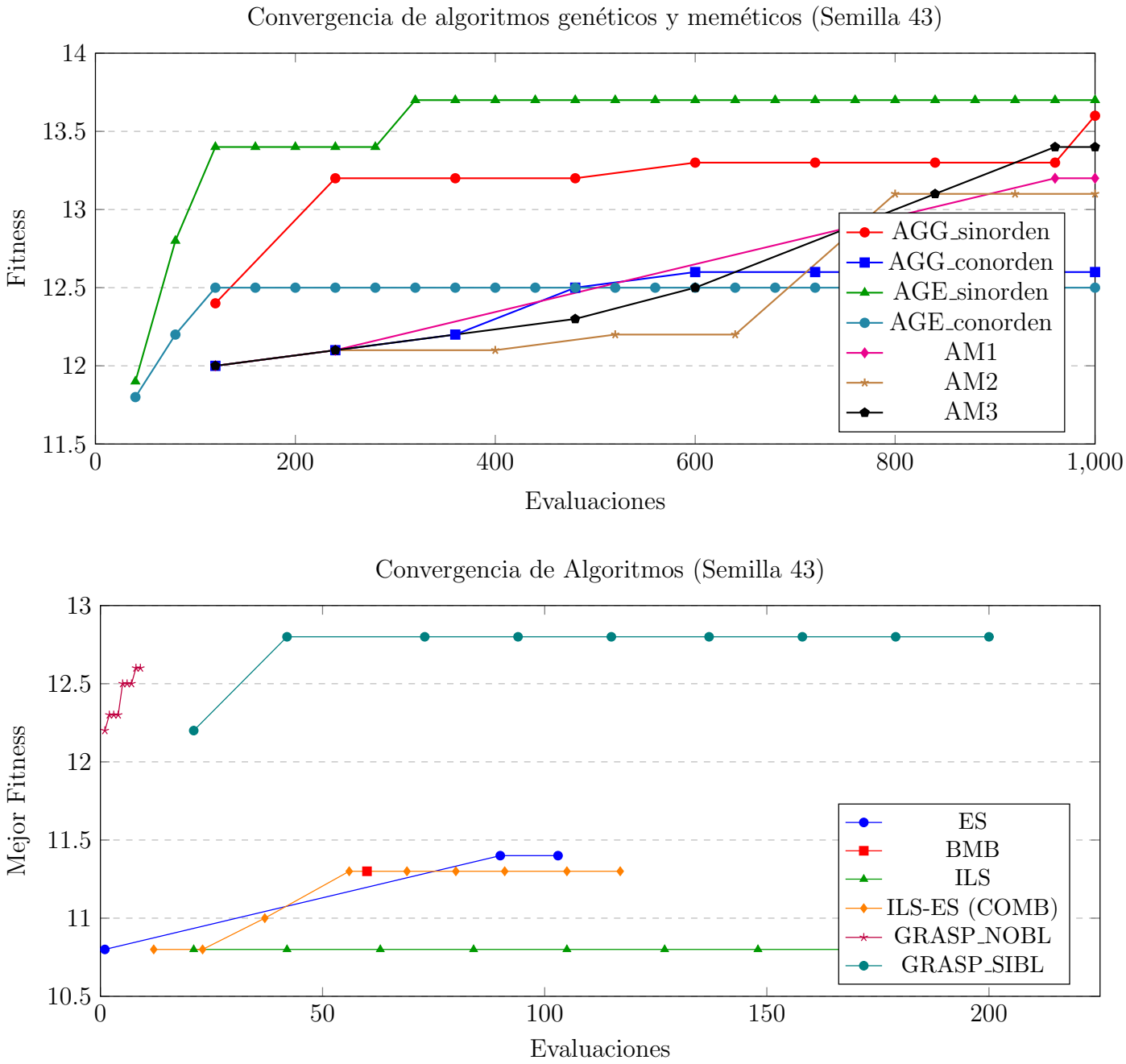


Figura 6: Convergencia de los algoritmos sobre SNIMP (semilla 43)

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.

Vemos que GRASP_SIBL es el más eficaz y estable, alcanza un fitness superior a 12.5 en solo unas 30 evalauciones. A partir de ahí, se mantiene estable, lo que indica que encuentra buenas soluciones rápidamente y no mejora más porque ya ha convergido a una zona de alta calidad. Tiene una alta eficacia inicial y convergencia rápida, claramente superior al resto.

GRASP_NOBL con 10 evalauciones supera el 12.5 de fitness, ligeramente por debajo de GRASP_SIBL. Esta velocidad tan alta se debe a que no aplica bñusqueda local, por lo que evalúa muchas soluciones construidas de forma diversa y eficiente desde el principio.

Enfriamiento Simulado parte con un fitness inicial algo bajo hasta estabilizarse cerca del 11.5. Tiene una convergencia lenta pero constante, lo que es esperable de un algoritmo de enfriamiento, pero se queda lejos del rendimiento de GRASP.

ILS-ES tiene una curva ascendente parecida a ES, pero más limitada. Las mejores son graduales a medida que exploran nuevas soluciones, pero el ritmo es lento, indicando que su rendimeinto depende mucho de las soluciones iniciales y de mutaciones que no generen suficiente diversidad.

ILS es plano, clavado en un fitness cercano a 10.8, lo que indica que no consigue mejorar nada con las evaluaciones. Esto puede deberse a que explora soluciones similares o que la bñusqueda local no aporta mejoras relevantes en su vecindario. Es el algoritmo con peor convergencia de todos los mostrados.

11.3. Análisis Final

Tabla 5: Tabla final de resultados

Algoritmo	**Posición Promedio**	**Tiempo Promedio (segs)**	**Evaluaciones Promedio**
Greedy	1.0	0.02	1.0
GRASP_SIBL	2.0	1.2	252.92
GRASP_NOBL	2.75	0.76	10.0
AM1	4.5	0.72	1000.0
ES	6.0	0.46	218.3
RandomSearch	6.25	0.79	1000.0
LSall	6.5	0.37	1000.0
ILS	7.75	0.29	251.2
BMB	8.5	0.34	261.6
ILS-ES (COMB)	8.5	0.3	131.9
BLsmall	11.0	0.05	28.1

Con esta tabla podemos ver los resultados finales, analizando las diferencias en el rendimiento de los algoritmos estudiados. Greedy consigue la mejor posición en todos los conjuntos de datos, en

un tiempo prácticamente instantáneo, por lo que se consolida como la estrategia más eficaz en esta práctica. Este rendimeinto confirma que la heurística utilizada está muy bien alineada con la estructura del SNIMP.

Ambas variantes de GRASP obtiene posiciones excelentes (2.0 y 2.75). GRASP_SIBL, aunque más costoso en tiempo y evaluaciones, es muy competitivo y mejora a la variante sin búsqueda local. GRASP_NOBL logra muy buenos resultados con tan solo 10 evaluaciones, lo que demuestra que la calidad de su componente constructiva es suficiente para alcanzar soluciones cercanas a las óptimas, incluso sin búsqueda local.

AM1, el algoritmo memético, consigue una buena posición, pero con un coste computacional muy elevado debido a las 1000 evaluaciones. Por otro lado, ES se comporta bastante bine, usando un número moderado de evaluaciones, lo que demuestra su capacidad para explorar con eficiencia sin necesidad de agotar el tiempo.

RandomSearch y LSall alcanzan posiciones medias, pero a costa de consumir todas las evaluaciones. RandomSearch tiene una capacidad limitada y muy variable, por lo que no se puede considerar muy buena opción. LSall, aunque mejor que BLsmall, no logra muy buenos resultados, probablemente por empezar por soluciones aleatorias muy malas.

ILS, BMB y su combinación tiene posiciones promedio muy malas, estos métodos iterativos se quedan muy por detrás de ES y GRASP, a pesar de usar un número considerable de evaluaciones. Esto sugiere que su combinación de mutación y búsqueda local no es tan eficaz en este problema, posiblemente debido a falta de diversidad en las soluciones generadas o una búsqueda local poco potente.

Finalmente, BLsmall ocupa el último lugar, con un tiempo y número de evaluaciones muy bajo. Esto es esperable, ya que su condición de parada a las 20 soluciones sin mejora no permite explorar correctamente el espacio de soluciones.