UNIVERSIDAD DE GRANADA

Práctica 1: 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 1: 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:

- **Búsqueda Voraz (Greedy).** Construye soluciones paso a paso eligiendo siempre la solución óptima localmente.
- **Búsqueda Aleatoria (Random).** Construye soluciones aleatorias y selecciona la mejor.
- **Búsqueda Local.** Mejora una solución inicial explorando soluciones cercanas.

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
    devolver 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
```

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. Búsqueda aleatoria (Random Search)

La clase `RandomSearch` implementa un algoritmo de búsqueda aleatoria. Su método `optimize` explora soluciones generadas al azar dentro del espacio de búsqueda de SNIMP, evaluando su calidad y conservando la mejor solución encontrada hasta agotar `maxevals`. En esta práctica, hacemos que el máximo de evalauciones es 1000.

```
Algoritmo RandomSearch(problem, maxevals):
    mejor_solucion <- []
    mejor_fitness <- -1

    Para i desde 0 hasta maxevals-1:
        solucion_actual <- createSolution()
        fitness_actual <- fitness(solucion_actual)

        Si fitness_actual > mejor_fitness o mejor_fitness < 0:
            mejor_solucion <- solucion_actual
            mejor_fitness <- fitness_actual

    return Resultado(mejor_solucion, mejor_fitness, maxevals)
```

Lo que hacemos es almacenar la mejor solución y su fitness en dos variables diferentes y se inicializa a -1 para asegurar que la primera solución generada siempre sea aceptada. A continuación, genera una nueva solución aleatoria en cada iteración usando y compara su fitness con el mejor fitness registrado: si es mejor (o es la primera solución), actualiza las variables. Finalmente, devuelve un objeto `ResutMH` que encapsula la mejor solución encontrada, su fitness y el número total de evaluaciones realizadas.

5. Búsqueda Voraz (Greedy)

El algoritmo greedy construye la solución seleccionando en cada paso el nodo con mayor valor heurístico, hasta alcanzar el tamaño deseado de solución. La función heurística utilizada mide la "influencia" potencial de cada nodo, definida comola suma del número de vecinos del nodo más el total de vecinos de éstos.

La función heurística ha sido explicada en el apartado anterior, incluyendo su pseudocódigo. El pseudocódigo de la función `optimize` del algoritmo Greedy es:

```
Algoritmo Greedy(problem, maxevals):
    snimp ← castear problem a Snimp
    numNodos ← snimp.getSize()

    heuristics ← mapa vacío
    Para i desde 0 hasta numNodos - 1:
        heuristics[i] ← Heurística(i)

    solución ← []

    Para i desde 0 hasta snimp.getSolutionSize() - 1:
        mejorNodo ← -1
        mejorHeurística ← -1

        para cada (nodo, valorHeurística) en heuristics:
            si valorHeurística > mejorHeurística:
                mejorNodo ← nodo
                mejorHeurística ← valorHeurística

        añadir mejorNodo a solución
        eliminar mejorNodo de heuristics

    fitness ← snimp.fitness(solución)
    devolver Resultado(solución, fitness, maxevals)
```

Al principio, creamos un mapa para almacenar el valor heurístico de cada nodo llamando a la función heurística de `Snimp`. A continuación hace la selección voraz, es decir, en cada iteración busca el nodo con mayor heurística del mapa y lo añade a la solución. Además, lo elimina del mapa para evitar repeticiones y poder buscar el siguiente mejor nodo. Cuando la solución alcanza el tamaño deseado de `solSize`, el algoritmo para y devuelve la solución, su fitness y el número de evaluaciones.

6. Búsqueda local

La búsqueda local es una técnica que trata de mejorar una solución aleatoria inicial iterativamente, explorando el espacio de soluciones vecinas. En esta práctica hemos implementado dos variantes de búsqueda local: LSall y BLsmall.

Tenemos una función común para generar vecinos aleatorios que todavía no se hayan explorado. Esta función es común para ambas variantes de la búsqueda local. Lo que hace es generar todos los posibles vecinos de una solución actual mediante intercambios de nodos, de forma aleatoria para evitar sesgos en la exploración.

Recibe como entrada una lista de nodos seleccionados (la solución) y una lista de nodos disponibles para el intercambio (los que no están en la solución). A partir de ahí, crea todas las combinaciones posibles donde se reemplaza un nodo de la solución actual por un nodo no seleccionado. Finalmente, baraja el orden de los vecinos para evitar que el algoritmo siempre explore en el mismo orden.

```
Función generarVecinosAleatorios(sol, noSeleccionados):
    vecinos ← lista vacía
    para cada i en 0 hasta tamaño de sol:
        para cada j en noSeleccionados:
            agregar par (i, j) a vecinos
    barajar(vecinos)
    devolver vecinos
```

Un ejemplo para entender lo que recibe y lo que devuelve la función es:

```
Recibe sol = [10,20,30] y noSeleccionado = [40,50]
Devuelve vecinos = [
    (0,40),
    (0,50),
    (1,40),
    (1,50),
    (2,40),
    (2,50)
]

Tras barajar: vecinos = [
    (1,40),
    (0,50),
    (1,50),
    (2,40),
```

```
        (0,40),
        (2,50)
    ]
```

6.1. LSall

El algoritmo Local Search all (LSall) busca mejorar una solución mediante el método de búsqueda de primero-mejor, que sigue un enfoque de búsqueda local hasta que no se pueda mejorar más o se haya alcanzado el límite de evaluaciones máximo (`maxevals`). En cada iteración, el algoritmo genera vecinos aleatorios e intenta mejorar la solución actual.

```
Algoritmo LSall(problem, maxevals):
    assert(maxevals > 0)
    snimp ← castear problem a Snimp
    assert(snimp no es nulo)

    sel ← crear solución inicial utilizando snimp
    bestFitness ← fitness de la solución sel
    evals ← 1
    improvement ← verdadero

    mientras improvement sea verdadero y evals < maxevals:
        improvement ← falso
        noSel ← calcular nodos no seleccionados (nodos_totales - sel)

        vecinos ← generarVecinosAleatorios(sel, noSel, rng)

        para cada vecino (i, j) en vecinos:
            nueva ← solución sel con i reemplazado por j
            fit ← fitness de la solución nueva
            evals ← evals + 1

            si fit > bestFitness:
                sel ← nueva
                bestFitness ← fit
                improvement ← verdadero
                romper el bucle (primer-mejor)

            si evals >= maxevals:
                romper el bucle
```

```
    devolver ResultadoMH(sel, bestFitness, evals)
```

Al principio generamos una solución inicial aleatoriamente y almacenamos su fitness. Además, inicializamos el contador de evaluaciones y un booleano que nos dirá si ha habido o no mejora en todo el vecindario. En el bucle principal, el programa guarda los nodos disponibles para intercambiar (todos los que no están en la solución) llamando a la función explicada anteriormente. A continuación, crea soluciones vecinas reemplazando un nodo de la solución por otro de los no seleccionados y acepta la primera mejora encontrada. Si se mejora el fitness, actualiza la solución y reinicia la exploración. El bucle continúa mientras haya mejoras y no se exceda `maxevals`. Finalemnte, devuelve la mejor solución, su fitness y las evaluaciones usadas.

6.2. BLsmall

El algoritmo BLsmall es una variante de LSall que incluye una condición de parada: si no se encuentra mejora en 20 evaluaciones consecutivas, el algoritmo se detiene.

```
Algoritmo BLsmall(problem, maxevals):
    assert(maxevals > 0)
    snimp ← castear problem a Snimp
    assert(snimp no es nulo)

    sel ← crear solución inicial utilizando snimp
    bestFitness ← fitness de la solución sel
    evals ← 1
    evalsSinMejora ← 0
    improvement ← verdadero

    mientras improvement sea verdadero y evals < maxevals y evalsSinMejora < 20:
        improvement ← falso
        noSel ← calcular nodos no seleccionados (nodos_totales - sel)

        vecinos ← generarVecinosAleatorios(sel, noSel, rng)

        para cada vecino (i, j) en vecinos:
            nueva ← solución sel con i reemplazado por j
            fit ← fitness de la solución nueva
            evals ← evals + 1

            si fit > bestFitness:
                sel ← nueva
```

```
                bestFitness ← fit
                improvement ← verdadero
                evalsSinMejora ← 0  // Resetear contador de sin mejora
                romper el bucle (primer-mejor)
            sino:
                evalsSinMejora ← evalsSinMejora + 1

            si evals >= maxevals o evalsSinMejora >= 20:
                romper el bucle

    devolver ResultadoMH(sel, bestFitness, evals)
```

Al igual que en el anterior algoritmo de búsqueda local, generamos aleatoriamente una solución inicial y calculamos su función objetivo. También inicializamos el contador de evaluaciones y un nuevo contador de evaluacioes consecutivas sin mejora. El bucle continúa mientras haya mejora, no se exceda el máximo de evaluaciones y no se alcancen 20 evaluaciones sin mejora. Es el único cambio con respecto al algoritmo anterior, procede de forma similar pero reiniciando el contador de evaluaciones sin mejora cada vez que se actualiza la mejor solución y el mejor fitness. Si no se actualiza dicho contador, entonces se aumenta en 1.

7. 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 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 (RandomSearch, Greedy, LSall o BLsmall), guarda sus resultados:

```
42 RandomSearch "2741 786 4605 1796" 11.5 1000 209
42 Greedy "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 (42 a 47) para ser comparables. Además, el fitness se calcula igual para todos (semilla 10), permitiendo una comparación justa. La búsqueda aleatoria (random search) tiene los siguientes parámetros:

- Evaluaciones: Prueba 1000 soluciones completamente aleatorias

El algoritmo Greedy tiene los siguiente parámetros:

- Evaluaciones: Solamente hace 1 evaluación (construye la solución en un solo paso)

El algoritmo de búsqueda local completa (LSall):

- Exploración: En cada paso, examina todos los vecinos posibles de la solución actual
- Evaluaciones: 1000 evaluaciones o menos, en caso de no mejorar la solución tras explorar todo el vecindario

Finalmente, los parámetros de la búsqueda local reducida (BLSmall) son:

- Exploración: En cada paso, examina todos los vecinos posibles de la solución actual
- Evaluaciones: 1000 evaluaciones o menos, en caso de no mejorar la solución tras explorar 20 vecinos consecutivos

8.2. Análisis de algoritmos

Tabla 1: Resultados para el Algoritmo Random Search

Conjunto	**Fitness**	**Tiempo (segs)**	**Evaluaciones**
ca-GrQC	12.5	5.716×10^{-1}	1000
p2p-Gnutella05	11.38	6.798×10^{-1}	1000
p2p-Gnutella08	11.36	5.086×10^{-1}	1000
p2p-Gnutella25	11.14	1.3786×10^{0}	1000

Tabla 2: Resultados para el Algoritmo Greedy

Conjunto	**Fitness**	**Tiempo (segs)**	**Evaluaciones**
ca-GrQC	18.8	1.10×10^{-2}	1
p2p-Gnutella05	13.7	1.48×10^{-2}	1
p2p-Gnutella08	12.6	1.02×10^{-2}	1
p2p-Gnutella25	13.4	4.62×10^{-2}	1

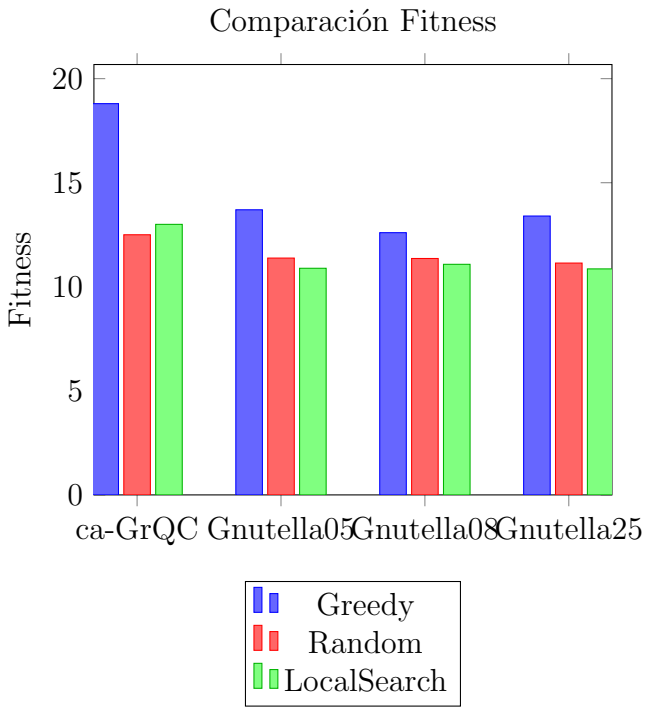
Tabla 3: Resultados para el Algoritmo Local Search

Conjunto	**Fitness**	**Tiempo (segs)**	**Evaluaciones**
ca-GrQC	13.2	3.67×10^{-1}	514.3
p2p-Gnutella05	10.89	1.26×10^{-1}	512.7
p2p-Gnutella08	11.08	2.20×10^{-1}	511.1
p2p-Gnutella25	10.86	1.98×10^{-1}	518.1

Podemos observar en el gráfico de arriba que el algoritmo Greedy es el que mejores resultados da sin duda: logra el mayor fitness en todos los conjuntos y en el conjunto `ca-GrQC.txt` da un resultado muy alto de 18.8. Es lo que se espera, ya que selecciona los nodos con más vecinos y más vecinos de estos, por lo que tiene sentido que sean los nodos que más pueden influenciar. Además, el tiempo es el más corto, ya que solo hace una sola evaluación. En todas las métricas: fitness, tiempo y evaluaciones, da los mejores resultados.

Por otro lado, la búsqueda aleatoria queda en un término medio. Al explorar aleatoriamente, tiene probabilidad de encontrar soluciones de todo tipo. Esto es ventajoso porque evita quedar estancado en soluciones malas, pero tampoco asegura buenos resultados. Puede acertar por casualidad, pero no es fiable.

Finalmente, la búsqueda local da los peores resultados. La razón es que solo explora cambios mínimos (de 1 nodo) en cada solución y en grafos muy grandes, esto apenas consigue mejorar el fitness. Además,



se queda atrapado en óptimos locales, a diferencia de la búsqueda aleatoria, por lo que da peores resultados.

En el caso del conjunto `ca-GrQC.txt` la búsqueda local supera a la búsqueda aleatoria, lo que sugiere que el óptimo local encontrado es casualmente bueno. Pero esto parece más bien suerte, ya que depende de la solución inicial aleatoria.

En este problema Greedy triunfa porque es aditivo, es decir, lo que suma un nodo no depende del resto de nodos. Así, al seleccionar los 10 mejores nodos, asegura buenos resultados. La búsqueda local falla porque se queda en soluciones mediocres y, al solo permitir cambios pequeños, no puede dar el salto a soluciones mucho mejores que podrían estar más lejos. La búsqueda aleatoria está en el medio: no es sistemática como Greedy, pero al menos no se atasca como la búsqueda local.

Como conclusión, para problemas como SNIMP, los métodos simples como Greedy puedes ser mejores que algoritmos más complejos.

8.3. 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 4: Resultados promedio de ca-GrQc.txt

Algoritmo	**Posición**	**Fitness**	**Tiempo (segs)**	**Evaluaciones**
RandomSearch	3	12.5	5.72×10^{-1}	1000
Greedy	1	18.8	1.10×10^{-2}	1
LSall	2	15.1	7.08×10^{-1}	1000
BLsmall	4	11.3	2.58×10^{-2}	28.6

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 el Greedy obtiene el mejor fitness con una única evaluación, lo que sugiere que la estrategia de elegir los 10 mejores nodos funciona bien en este caso. LSall logra un fitness de 15.1, que indica que una búsqueda local extensiva no proporciona mejoras con respecto a Greedy. El algoritmo Random search alcanza 12.5 de fitness, reflejando la aleatoriedad del método y el BLsmall es el peor en este conjunto, debido a que se detiene prematuramente tras 29 evaluaciones de media.

Tabla 5: Resultados promedio de p2p-Gnutella05.txt

Algoritmo	**Posición**	**Fitness**	**Tiempo (segs)**	**Evaluaciones**
RandomSearch	2	11.38	6.80×10^{-1}	1000
Greedy	1	13.7	1.48×10^{-2}	1
LSall	3	10.98	2.18×10^{-1}	1000
BLsmall	4	10.8	3.48×10^{-2}	25.4

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. Greedy vuelve a obtener el mejor fitness, lo que sugiere que esta estrategia sigue siendo efectiva. El Random search consigue un fitness de 11.38, superando a los otros dos, lo que indica que la aleatoriedad funciona mejor que una búsqueda local intensa.

Tabla 6: Resultados promedio de p2p-Gnutella08.txt

Algoritmo	**Posición**	**Fitness**	**Tiempo (segs)**	**Evaluaciones**
RandomSearch	2	11.36	5.09×10^{-1}	1000
Greedy	1	12.6	1.02×10^{-2}	1
LSall	3	11.36	2.76×10^{-1}	1000
BLsmall	4	10.8	1.64×10^{-2}	22.2

Este conjunto corresponde a otra instancia de la red Gnutella, con 6301 nodos y 20777 enlaces. Es ligeramente más pequeña que la anterior. Greedy sigue obteniendo el mejor fitness y el RandomSearch

sigue manteniendo un rendimeinto similar al caso anterior. Se comprueba que la exploración profunda no es beneficiosa en este problema.

Tabla 7: Resultados promedio de p2p-Gnutella25.txt

Algoritmo	**Posición**	**Fitness**	**Tiempo (segs)**	**Evaluaciones**
RandomSearch	2	11.14	1.38×10^{0}	1000
Greedy	1	13.4	4.62×10^{-2}	1
LSall	3	10.92	2.84×10^{-1}	1000
BLsmall	4	10.8	1.11×10^{-1}	36.2

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. La búsqueda local obteien los valores más bajos, indicando que la búsqueda más intensiva de vecinos cercanos a una solución vecina no es efectiva en esta red de gran tamaño.

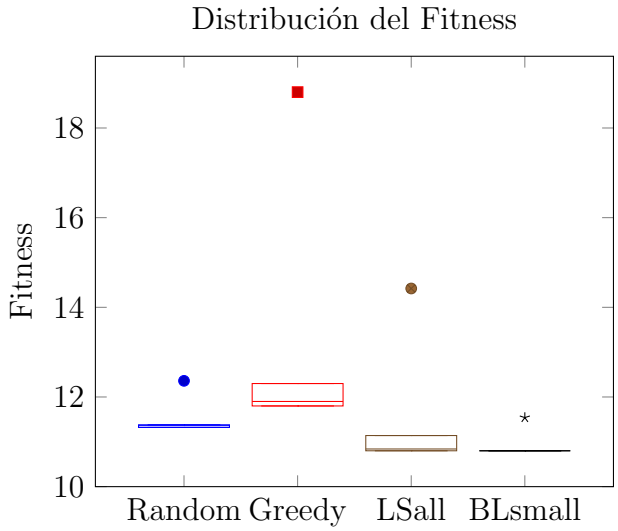


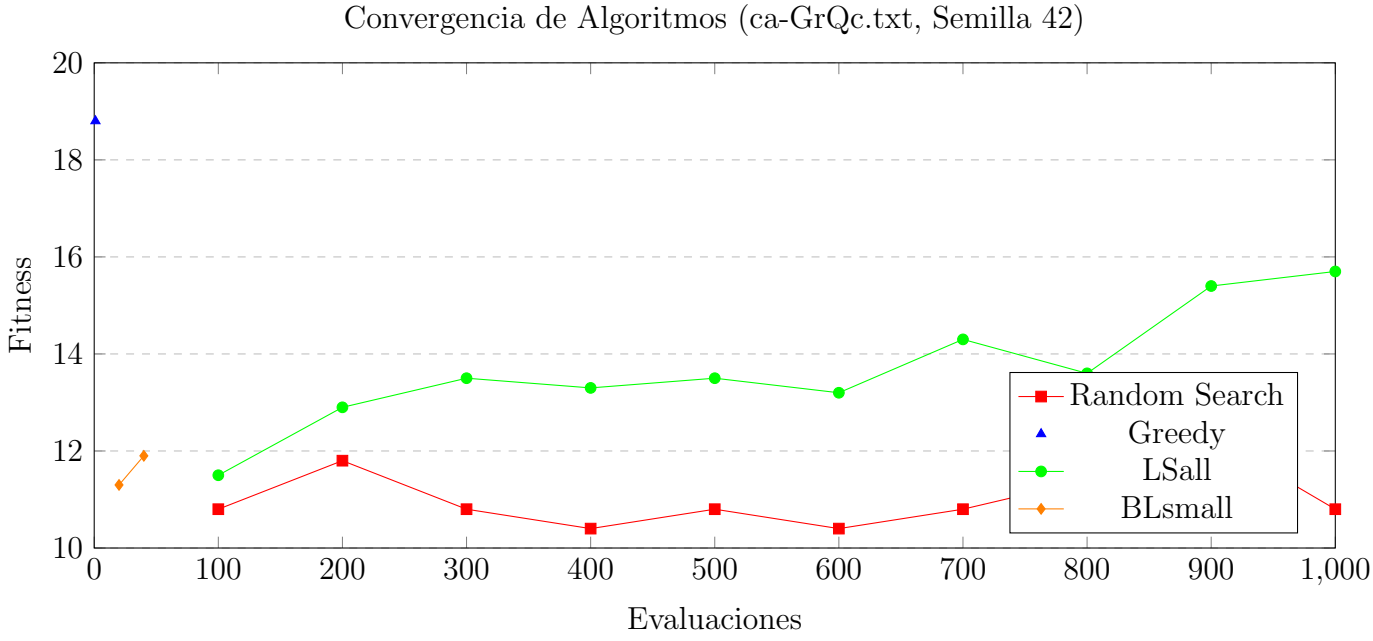
Figura 1: Boxplot de los valores de fitness para cada algoritmo

El diagrama de caja muestra la distribución de los valores del fitness obtenidos por los cuatro algoritmos. La búsqueda aleatoria (Random) presenta un fitness promedio relativamente bajo, además los valores del fitness no se dispersan demasiado, es decir, que independientemente del caso, siempre presenta un fitness parecido.

El Greedy es el algoritmo que mayor dispersión de fitness muestra, es decir, que depende de la conexión que tengan los nodos entre sí, puede dar un fitness muy diferente. Esto lo podemos comprobar con el outlier tan alto que tiene, que indica que en un caso el algoritmo logra un fitness muy superior al resto.

El LSall presenta menos dispersión que el Greedy, es decir, tiene un rango de valores más compacto. Su mediana es más baja que la de Random Search, lo que sugiere que obtiene peores resultados en general. Aunque tiene un valor atípico alto, lo que indica que puede llegar a soluciones de fitness significativamente mejores en algunos casos.

Finalmente, el BLsmall tiene la menor dispersión en sus valores de fitness, con un rango muy estrecho. Su mediana es la más baja, lo que sugiere que su rendimiento es más consistente, pero peor.



En la gráfica muestra el comportamiento de los cuatro algoritmos en el conjunto `ca-GrQC.txt` usando la semilla 42, donde el eje horizontal representa las evaluaciones realizadas y el eje vertical el valor del fitness.

La curva del Greedy vemos que es un punto azul en la evaluación 1, que destaca con un fitness de 18.8. Esto confirma su eficacia para problemas de maximización, donde seleccionar los nodos clave (los altamente conectados) proporciona soluciones óptimas rápidamente.

La línea verde de LSall muestra una mejora progresiva desde 11.5 hasta 15.7 en 1000 evaluaciones. La curva es generalmente creciente, ya que cada vez explora los vecinos de las mejores soluciones de cada vecindario. Aunque hay veces que el fitness decrece, esto es porque se está cogiendo el fitness de la solución evaluada en cada iteración, no necesariamente el mejor registrado. Sin embargo, su convergencia lenta indica que, aunque mejora la solución inicial, requiere muchas evaluaciones para refinarla.

La línea roja del Random search oscila aleatoriamente entre 10.4 y 12.3 sin converger, como era espe-

rable. La falta de tendencia ascendente muestra que, aunque explora diversas regiones del espacio de soluciones, no aprovecha la información para mejorar. Su mejor fitness sigue siendo significativamente peor al del Greedy.

Los puntos naranjas pertenecen a la línea de BLsmall, que se detiene tras 40 evaluaciones por su criterio de parada (20 iteraciones sin mejora). Aunque eficiente, su fitness final es bajo comparado con el resto de algoritmos, sugiriendo que el criterio de parada es demasiado restrictivo para un vecindario tan grande de cada solución y un grafo tan complejo.

8.4. Análisis Final

Tabla 8: Tabla final de resultados

Algoritmo	**Fitness Promedio**	**Tiempo Promedio (segs)**	**Evaluaciones Promedio**
RandomSearch	11.595	7.85×10^{-1}	1000
Greedy	14.625	2.06×10^{-2}	1
LSall	11.5075	2.28×10^{-1}	514.05
BLsmall	10.99	2.94×10^{-2}	28.2

Los resultados muestran que el algoritmo Greedy es claramente el más eficiente en términos de fitness y tiempo de ejecución, ya que selecciona directamente los nodos con mayor conectividad, lo que maximiza la influencia. Random Search, aunque menos efectivo, evita quedar atrapado en óptimos locales, lo que le da una ventaja sobre Local Search en ciertos casos. Local Search, por su parte, presenta limitaciones importantes al restringir sus cambios a modificaciones pequeñas, lo que lo hace ineficaz para encontrar soluciones óptimas en redes grandes y complejas.

Este análisis sugiere que, para problemas de selección de nodos influyentes en redes similares a las analizadas, estrategias simples y directas como Greedy pueden ser más efectivas que métodos más elaborados. Sin embargo, en escenarios donde las restricciones del problema impidan una selección tan directa, podría ser necesario combinar técnicas como Greedy y Local Search para mejorar la exploración del espacio de soluciones.