udp Escuela de Informática y Telecomunicaciones

Facultad de Ingeniería y Ciencias
Escuela de Informática y Telecomunicaciones

Algoritmos Exactos y Metaheurísticas
Tarea 02

Jazmín Almendra Jassive Cuitiño Mendoza
Felipe Nicolás Gutierrez Lazo
jazmin.cuitino@mail.udp.cl
felipe.gutierrez@mail.com

23 de mayo de 2024

Índice general

1. Introducción

En esta actividad, se pone a prueba el conocimiento de los alumnos en relación a la creación de técnicas incompletas, como es el caso de algoritmos *greedy*, *hill climbing* y una variante de este último mencionado. La actividad consta de lo siguiente:

- Una región dividida en sectores.
- Clínicas que se pueden instalar.
- Lugares dónde instalar las clínicas para cumplir la demanda del sector.
- Costo de instalación de una clínica en un lugar.

El objetivo principal es determinar los lugares donde se deben instalar las clínicas para lograr satisfacer toda la demanda de una región, con el mínimo costo posible.

Para esto, en la parte inicial de la actividad, se utilizan dos variantes de una técnica incompleta, *greedy*. Se realiza un *greedy* determinista y uno estocástico. Luego, se realiza el algoritmo *Hill Climbing*. Finalmente, se genera una variante al algoritmo que consiste en eliminar valores de la solución para ver su comportamiento.

Luego de la generación del código, se procede a realizar un análisis con respecto a los resultados obtenidos, gráficos de convergencia, comportamiento de los algoritmos y diversos factores que influencian en la solución final.

Finalmente, se genera una conclusión sobre el trabajo, comparando técnicas, valores, implementación y dificultades presentadas en el trayecto de la actividad.

2. Descripción Actividades

Dado que se presenta un problema de optimización con restricciones, existen los siguientes componentes: **función objetivo, variables de decisión** y **restricciones**. Se proponen 3 enfoques distintos para dar solución al problema planteado:

1. **Greedy Determinista y Estocástico:** El algoritmo *Greedy Determinista* sigue un enfoque voraz, tomando decisiones localmente óptimas en cada paso. Por otro lado, el algoritmo *Greedy Estocástico* introduce un elemento aleatorio para explorar diferentes soluciones y así evitar estancarse en óptimos locales. La gran diferencia que tienen ambas características es la solución final; el greedy determinista siempre entrega la misma solución, eligiendo los mismos elementos cada vez que se ejecuta, mientras que el greedy estocástico en cada ejecución entrega soluciones distintas.

2. **Variante de Hill-Climbing:** El segundo paso de la actividad es utilizar el algoritmo Greedy Estocástico para generar soluciones iniciales, las que son tomadas por un Hill-Climbing para realizar una búsqueda local en su vecindario.

3. **Variante del Greedy Estocástico:** Finalmente, se propone una variante del algoritmo Greedy Estocástico diseñado al comienzo de la actividad. En este punto, a la solución generada por este greedy se le quitan n elementos (con n configurable) para posteriormente reanudar la búsqueda desde la solución parcial resultante. La idea de esta técnica es explorar diferentes regiones del espacio de búsqueda y así mejorar la calidad de las soluciones encontradas.

Una vez resuelto el problema con las 4 técnicas propuestas se presenta un análisis comparativo entre los resultados de las mismas.

3. Desarrollo actividades

A diferencia de la actividad anterior, no es necesario modelar las componentes del problema tales como la función objetivo y las restricciones, por lo que el enfoque solamente está en generar los códigos.

3.1. Parte 01 - Greedy Determinista y Estocástico

Para resolver el problema planteado con esta estrategia, se diseñan 2 funciones: *greedy_determinista* y *greedy_estocastico*. Sin embargo, previo a estas funciones es muy importante la lectura de los *benchmarks* y el posterior almacenamiento en un formato que facilite el trabajo. La siguiente función implementa lo anterior:

Listing 3.1: Función para extraer la informacion del archivo

```
def leer_datos_archivo(ruta_archivo):
    with open(ruta_archivo, 'r') as archivo:
        m = int(archivo.readline().strip())  # cantidad de sectores
        n = int(archivo.readline().strip())  # cantidad de lugares donde
            instalar cl nicas
        # Leer el vector de costos de instalacion
        costos = []
        while len(costos) < n:
            line = archivo.readline().strip()
            if line:  # Asegurarse de que la linea no este vacia
                costos.extend(map(int, line.split()))
        # Leer las coberturas
        coberturas = []
        contenido = archivo.readlines()  # Lee el resto del archivo de una
            vez
        i = 0
        while i < len(contenido):
            # Lee la cantidad de datos para el siguiente arreglo
            if contenido[i].strip():  # Asegura que no este vacia la linea
                cantidad = int(contenido[i].strip())
                i += 1
                # Inicializa una lista vacia para almacenar los datos
                datos = []
                # Continua leyendo las siguientes lineas hasta completar
                    la cantidad necesaria
                while len(datos) < cantidad and i < len(contenido):
                    linea = contenido[i].strip()
                    if linea:  # Asegura que la linea no este vacia
                        datos.extend(map(int, linea.split()))
                    i += 1
                # Agrega la lista de datos a las coberturas
                coberturas.append(datos)
    return m, n, costos, coberturas
```

De esta forma, la función *leer_datos_archivo* retorna la información estructurada en un formato adecuado para que los algoritmos greedy puedan utilizarla sin problema alguno.

3.1.1. Greedy Determinista

La función que se diseña para implementar el algoritmo asociada al greedy determinista es la siguiente:

Listing 3.2: Función Greedy Determinista

```
def greedy_determinista(sectores, lugares, costos, coberturas):
    sectores_cubiertos = set()
    lugares_elegidos = []
    # Mientras no se cubran todos los sectores
    while len(sectores_cubiertos) < sectores:
        mejor_costo_beneficio = float('inf')
        mejor_lugar = -1
        mejor_cobertura = []
        # Encontrar el lugar con el mejor costo-beneficio
        for lugar in range(lugares):
            # Calcular los sectores que cubre este lugar
            sectores_que_cubre = set()
            for sector in range(sectores):
                if lugar in coberturas[sector]:
                    sectores_que_cubre.add(sector)
            # Calcular la cantidad de sectores no cubiertos que cubre este
                lugar
            sectores_no_cubiertos = [s for s in sectores_que_cubre if s
               not in sectores_cubiertos]
            if sectores_no_cubiertos:
                costo_beneficio = costos[lugar] / len(
                   sectores_no_cubiertos)
                if costo_beneficio < mejor_costo_beneficio:
                    mejor_costo_beneficio = costo_beneficio
                    mejor_lugar = lugar
                    mejor_cobertura = sectores_no_cubiertos
        # Si encontramos un lugar valido, lo agregamos a los lugares
           elegidos y actualizamos los sectores cubiertos
        if mejor_lugar != -1:
            lugares_elegidos.append(mejor_lugar)
            sectores_cubiertos.update(mejor_cobertura)
        else:
            # No hay mas lugares que puedan cubrir sectores no cubiertos
            break
    return lugares_elegidos
```

A continuación, se explican las partes clave del código:

1. **Parámetros de entrada:** La función recibe parámetros de entrada, los cuales son:
   - **sectores**: Referencia al número total de sectores que deben ser cubiertos.
   - **lugares**: Refiere al número total de lugares disponibles que pueden ser seleccionados para cubrir sectores.
   - **costos**: Lista que contiene el costo de instalar una clínica en cada lugar.
   - **coberturas**: Vector de vectores. Cada posición *coberturas[i]* indica los lugares que cubren la demanda del sector i.

2. **Inicialización de variables:** Es importante definir las variables necesarias para iniciar el algoritmo:
   - **sectores_cubiertos = set():** Conjunto que almacena los sectores ya cubiertos.
   - **lugares_elegidos = []:** Lista para almacenar los lugares que han sido seleccionados para cubrir la demanda de los sectores.

Luego, se crea un bucle principal, que se ejecuta mientras el número de sectores cubiertos sea menor que el total de sectores a cubrir. Dentro de este se encuentra:

1. **mejor_costo_beneficio**: Variable que permite almacenar el menor valor en la relación costo/beneficio. Se inicializa en infinito para tener seguridad que cualquier valor que encuentre sea menor.
2. **mejor_lugar**: Almacena el índice del lugar con la mejor relación costo/beneficio.
3. **mejor_cobertura**: Lista de sectores no cubiertos que el mejor lugar puede cubrir.

Posteriormente, se procede a iterar sobre cada lugar. Inicialmente se calculan los sectores que cada lugar puede cubrir. La variable **sectores_que_cubre** se encarga de recolectar todos los sectores que un lugar en específico puede cubrir. Luego, se realiza un proceso de verificación, donde para cada sector, se verifica si el lugar está en su lista de coberturas. Si está presente, se agrega al conjunto **sectores_que_cubre**.

Hecho esto, se calculan los sectores que no están cubiertos y que el lugar actual puede cubrir. Para esto, se utiliza la variable **sectores_no_cubiertos**, que almacena una lista de sectores que el lugar actual puede cubrir pero que aún no han sido cubiertos. Si se da este caso (sectores que aún no han sido cubiertos, pero que el lugar si puede cubrir), se calcula la relación costo/beneficio dividiendo el costo de instalar una clínica en el lugar por el número de sectores no cubiertos que se pueden cubrir. Este valor se almacena en la variable **costo_beneficio**. Si el valor que se almacena en *costo_beneficio* es menor a *mejor_costo_beneficio*, se actualiza en conjunto con *mejor_lugar* y *mejor_cobertura* con los valores del lugar actual.

El siguiente paso dentro de la función es seleccionar el mejor lugar. Si un lugar válido se encuentra (mejor_lugar =! 1), se agrega a la variable *lugares_elegidos* y se actualiza la variable *sectores_cubiertos* con *mejor_cobertura*. Si no hay más lugares que puedan contribuir a cumplir con la demanda de los sectores no cubiertos, se rompe el bucle con el comando **break**.

Finalmente, la función retorna la lista de **lugares_elegidos**, que contiene los lugares seleccionados que permiten cubrir la demanda de la región considerando el mínimo costo asociado.

Ahora, cada componente del greedy es elegido en base a la mejor adaptación que el grupo considera para la resolución del problema. En este caso, la representación corresponde a una lista (lugares_elegidos) que almacena los lugares que satisfacen con la demanda de la región, cubriendo cada uno de los sectores necesarios. La función miope corresponde a la evaluación de la relación costo/beneficio que cada lugar tiene asociado para satisfacer la demanda de cierto sector.

3.1.2. Greedy Estocástico

Para este algoritmo, se utiliza el siguiente código:

Listing 3.3: Función Greedy Estocástico

```
def greedy_estocastico(sectores, lugares, costos, coberturas, alpha=0.1):
    sectores_cubiertos = set()
    lugares_elegidos = []
    #Mientras no se cubran todos los sectores
    while len(sectores_cubiertos) < sectores:
        candidatos = [] #lista que almacena candidatos con costo/beneficio
        #Iterar sobre cada lugar para evaluar sectores a cubrir
        for lugar in range(lugares):
            sectores_que_cubre = set() #sectores que el lugar cubre
            for sector in range(sectores):
                if lugar in coberturas[sector]: #verificar si lugar esta
                    en lista de cobertura del sector
```

```
                sectores_que_cubre.add(sector)
        #Determina que sectores aun no estan cubiertos
        sectores_no_cubiertos = [s for s in sectores_que_cubre if s
            not in sectores_cubiertos]
        if sectores_no_cubiertos: #si hay sectores no cubiertos,
            calcular relacion costo/beneficio
            costo_beneficio = costos[lugar] / len(
                sectores_no_cubiertos)
            #almacenar relacion, lugar y sectores que cubre
            candidatos.append((costo_beneficio, lugar,
                sectores_no_cubiertos))
    # Ordenar los candidatos por costo-beneficio
    candidatos.sort(key=lambda x: x[0])
    # Seleccionar los mejores candidatos con un factor de aleatoriedad
    k = max(1, int(alpha * len(candidatos)))
    mejor_candidato = random.choice(candidatos[:k])
    # Agregar el mejor candidato encontrado a los lugares elegidos y
        actualizar sectores cubiertos
    _, mejor_lugar, mejor_cobertura = mejor_candidato
    lugares_elegidos.append(mejor_lugar)
    sectores_cubiertos.update(mejor_cobertura)

return lugares_elegidos
```

A continuación, se describe la función en profundidad:

1. **Parámetros de entrada**:
   - **sectores**: Referencia al número total de sectores que deben ser cubiertos.
   - **lugares**: Refiere al número total de lugares disponibles que pueden ser seleccionados para cubrir sectores.
   - **costos**: Lista que contiene el costo de instalar una clínica en cada lugar.
   - **coberturas**: Vector de vectores. Cada posición *coberturas[i]* indica los lugares que cubren la demanda del sector i.
   - **alpha**: Factor de aleatoriedad para hacer una selección de los mejores candidatos.
2. **Inicialización de variables**:
   - **sectores_cubiertos:** Conjunto que almacena los sectores que son cubiertos por los lugares seleccionados.
   - **lugares_elegidos:** Lista para almacenar los lugares que han sido seleccionados para cubrir los sectores.

Luego de esto, se genera el bucle inicial, que se ejecuta mientras el número de sectores cubiertos sea menor que el total de sectores a cubrir. Dentro del bucle, se declara la variable **candidatos**, que permite almacenar los posibles lugares, junto con su relación costo/beneficio y los sectores que pueden cubrir.

Luego, se itera sobre cada lugar, y se realiza el cálculo de los sectores que cada lugar puede cubrir. La variable **sectores_que_cubre** es un conjunto que recoge todos los sectores que un lugar específico puede cubrir. Además, para cada sector se verifica si el lugar está en su lista de coberturas. De ser así, el sector se agrega a la variable **sectores_que_cubre**.

Posterior a esto, se evalúa la relación costo/beneficio que tiene cada lugar. **sectores_no_cubiertos** es una lista de sectores que el lugar actual tiene la capacidad de cubrir y que aún no son cubiertos. Si hay sectores que no han sido cubiertos, se calcula el costo/beneficio, dividiendo el costo del lugar por el número de sectores no cubiertos que se pueden cubrir. Seguido a esto, agrega una tupla (*costo_beneficio, lugar, sectores_no_cubiertos*) a la lista de candidatos si hay sectores que el lugar actual puede cubrir y aún no han sido cubiertos.

Luego, se realiza la selección de los candidatos. Lo primero que se hace es ordenar por el valor que tiene cada candidato en base a su relación costo/beneficio, donde un menor valor resulta mejor, indicando un menor costo por sector no cubierto. A esto, y generando la diferencia con el greedy determinista, se le agrega una selección estocástica de los **k** mejores candidatos. La variable **k** representa el número de candidatos entre los cuales se realiza una elección al azar, como un máximo entre 1 y el producto de *alpha* por la longitud de la lista de candidatos. Recordar que alpha corresponde a un indicador del porcentaje sobre el cuál se desea realizar la elección de candidatos. La variable **mejor_candidato** realiza la selección aleatoria de uno de los primeros **k** mejores candidatos para seguir construyendo la solución.

Seguido a esto, se actualizan el conjunto de lugares elegidos y sectores cubiertos. El lugar del **mejor_candidato** se agrega a **lugares_elegidos**, y se actualiza el conjunto de **sectores_cubiertos** con los sectores que cubre el lugar actual.

Finalmente, la solución devuelve la lista **lugares_elegidos**, que contiene los lugares seleccionados que logran minimizar el costo maximizando la cobertura de los sectores.

Ahora, justificando los componentes del greedy estocástico, se tiene:

- **Representación:** Al igual que en el greedy determinista, la representación es una lista (*lugares_elegidos*) que contiene todos los lugares que satisfacen la demanda de los sectores.
- **Función miope:** Corresponde a la evaluación de la relación costo/beneficio que cada lugar tiene asociado para poder satisfacer la demanda de cierto sector.
- **Variable estocástica:** Una variable estocástica corresponde a una variable que toma un valor en base a una probabilidad determinada. En este caso, corresponde al parámetro *alpha*, el cual permite evitar que siempre se escoja el mismo lugar en ejecuciones repetidas, y además, permite potencialmente escapar de óptimos locales en términos de costo y cobertura. Con relación a la exploración y explotación, esta variable resulta ser el componente necesario para tener un balance entre la exploración y la explotación que genera el greedy determinista, ya que permite generar soluciones distintas, movilizándose dentro de distintos espacios del espacio de búsqueda. Además, esta variable permite utilizar la distribución de probabilidad necesaria para realizar el algoritmo estocástico, ya que define la cantidad de posibles elementos a elegir para seguir construyendo la solución.

3.2. Parte 02 - Hill-Climbing

En esta sección, es necesario construir un algoritmo de búsqueda local como el **Hill-Climbing**. Este algoritmo se caracteriza por pertenecer a la categoría de solución única o trayectoria, es decir, toma una solución que la va mejorando en el tiempo. Como es un algoritmo de búsqueda local, sólo se centra en la explotación. Además, se considera perturbativo, al contrario del greedy, que son técnicas constructivas. Una técnica perturbativa corresponde a una técnica que necesita de una solución inicial para poder llegar a una solución final; en cambio, una técnica constructiva genera una solución desde cero. En este caso, se utiliza el greedy estocástico de la sección anterior para generar la solución inicial requerida por el *Hill-Climbing*.

A continuación, se adjunta el código de la función que realiza el algoritmo.

Listing 3.4: Función Hill Climbing

```
def hill_climbing(sectores, lugares, costos, coberturas, alpha=0.05,
    max_iter=100, reinicios=10):
    # Inicializar con una soluci n del Greedy Estoc stico
    mejor_solucion = greedy_estocastico(sectores, lugares, costos,
        coberturas, alpha)
    # Calcular el valor de la funci n objetivo para la soluci n inicial
    mejor_valor = sum(costos[lugar] for lugar in mejor_solucion)
    iter_sin_mejora = 0
    valores_funcion_objetivo = [mejor_valor]  # Registro de valores de la
        funci n objetivo
    #iteraciones = [0]  # Registro de iteraciones
```

```
    tiempos_mejora = [0]  # Registro de tiempos de mejora en segundos
        desde el inicio

    # Informar el valor inicial
    print("Inicializaci n - Mejor valor:", mejor_valor)

    # Bucle principal hasta el m ximo de iteraciones
    for i in range(1, max_iter + 1):
        # Crear un vecino por intercambio
        vecino = list(mejor_solucion)
        indices = random.sample(range(len(vecino)), 2)
        vecino[indices[0]], vecino[indices[1]] = vecino[indices[1]],
            vecino[indices[0]]

        # Calcular el valor de la funci n objetivo para el vecino
        valor_vecino = sum(costos[lugar] for lugar in vecino)

        # Si el vecino es mejor, actualizar la mejor soluci n
        if valor_vecino < mejor_valor:
            mejor_solucion = vecino
            mejor_valor = valor_vecino
            iter_sin_mejora = 0
            valores_funcion_objetivo.append(mejor_valor)
            tiempos_mejora.append(time.time() - start_time)  # Actualizar
                el tiempo de mejora
            print(f"Iteraci n {i} Nuevo mejor valor = {mejor_valor}")
        else:
            iter_sin_mejora += 1

        # Si se alcanza el l mite de iteraciones sin mejora, realizar un
            reinicio
        if iter_sin_mejora >= reinicios:
            nuevo_intento_solucion = greedy_estocastico(sectores, lugares,
                costos, coberturas, alpha)
            valor_actual = sum(costos[lugar] for lugar in
                nuevo_intento_solucion)
            # Si el reinicio proporciona una mejor soluci n, actualizar
            if valor_actual < mejor_valor:
                mejor_solucion = nuevo_intento_solucion
                mejor_valor = valor_actual
                valores_funcion_objetivo.append(mejor_valor)
                tiempos_mejora.append(time.time() - start_time)  #
                    Actualizar el tiempo de mejora
                print(f"Iteraci n {i} - r - nuevo mejor valor = {
                    mejor_valor}")
            iter_sin_mejora = 0  # Reiniciar contador de estancamiento

    # Devolver la mejor soluci n encontrada, registro de valores y
        iteraciones
    return mejor_solucion, valores_funcion_objetivo, tiempos_mejora
```

La descripción en profundidad de la función se realiza a continuación:

1. **Parámetros de entrada**
   - **sectores:** Representa el número total de sectores que deben ser cubiertos.

- **lugares:** Número total de lugares disponibles que pueden ser seleccionados para cubrir la demanda de los sectores.
- **costos:** Lista donde cada elemento corresponde al costo de seleccionar un lugar específico para instalar una clínica.
- **coberturas:** Lista de listas, donde cada posición *coberturas[i]* contiene los lugares que satisfacen la demanda del sector *i*.
- **alpha:** Determina el nivel de aleatoriedad en el método *greedy_estocastico*.
- **max_iter:** Define el número máximo de iteraciones que el **Hill-Climbing** ejecuta antes de detenerse.
- **reinicios:** Especifíca el número de iteraciones consecutivas sin mejora en el valor de la función objetivo antes de realizar un reinicio a partir de una solución generada por *greedy_estocastico*.

2. **Inicialización de variables:**
   - **mejor_solucion:** Almacena la solución inicial generada por el *greedy_estocastico*.
   - **mejor_valor:** Calcula el mejor valor de la solución inicial entregada por el greedy.

Luego de la inicialización de las variables, se genera el bucle principal el cual itera hasta la cantidad de **max_iter** entregada por parámetro a la función. Dentro del bucle, se genera una solución vecina intercambiando dos lugares aleatorios, y evaluando esa solución. Si el valor de la función objetivo mejora, **mejor_solucion** se actualiza y se reinicia el contador **iter_sin_mejora**. Si esta variable alcanza el valor de **reinicios**, se genera una nueva solución a partir del greedy estocástico.

Dentro de los componentes del **Hill-Climbing**, se tiene:

- **Función objetivo:** Se enfoca en minimizar el costo total de seleccionar un conjunto de lugares para cubrir todos los sectores necesarios. Corresponde a la línea de código *mejor_valor = sum(costos[lugar] for lugar in mejor_solucion)*.
- **Operador de movimiento:** El operador de movimiento seleccionado corresponde a intercambiar dos elementos dentro de la solución actual. Permite explorar soluciones vecinas dentro del espacio de búsqueda. Se elige por sobre otro operador de movimiento por la simplicidad que tiene de implementación y la eficacia que tiene para realizar exploraciones locales en el espacio de soluciones.
- **Criterio de término:** Definido por dos componentes principales: **max_iter** y **reinicios**. **max_iter** permite establecer un límite superior en el número de iteraciones que ejecuta el hill climbing, mientras que el número de reinicios detiene el proceso iterativo si alcanza un número especificado de iteraciones consecutivas sin lograr mejora alguna en el valor de la función objetivo. Se utilizan estos criterios de término para asegurar que el algoritmo sea robusto tanto en la prevención de ejecuciones excesivas como en proporcionar mecanismos efectivos para poder escapar de los mínimos locales, explorando el espacio de soluciones adecuadamente.

Finalmente, la variante utilizada es **alguna-mejora** con **restarts.** Esta variante permite que el algoritmo se ejecute de manera "lazy", aceptando cualquier solución que cumpla con las restricciones, sin necesidad de recorrer el vecindario completo. Además, los restarts permiten escapar de los óptimos locales en los que se estanca la ejecución, permitiendo mejorar la solución final.

3.3. Parte 03 - Variante Greedy Estocástico

Finalmente, se implementa una variante del Greedy Estocástico. La idea es generar soluciones con la función **greedy_estocástico()**, que es una técnica constructiva. Una vez generada una solución, se le elimina una cantidad n de lugares, con el propósito de llamar a una nueva función llamada **completar_solucion**, que tiene la misma lógica que el greedy estocástico, pero que recibe una solución incompleta y va a iterar hasta retornar nuevamente una solución factible distinta de la original (va a iterar m veces). De este modo, la función para eliminar n valores de los lugares seleccionados por el greedy es:

Listing 3.5: Función para destruir la solución inicial

```
def eliminar_lugares(solucion, n):
    return random.sample(solucion, len(solucion) - n)
```

Se aprecia, en la función anterior, que se eliminan n valores al azar de la solución inicial, para posteriormente completarla nuevamente. Esto permite explotar el vecindario en el que se encuentra la solución generada inicialmente, y encontrar óptimos locales; de esto se desprende un problema, el estancamiento en óptimos locales, tema que sera abordado en el análisis. De este modo, la función que realiza el trabajo de completar el conjunto de lugares seleccionados funciona de la siguiente manera:

Listing 3.6: Función Completar Solución

```
def completar_solucion(sectores, lugares, costos, coberturas,
   solucion_parcial):
    sectores_cubiertos = set()
    for lugar in solucion_parcial:
        for sector in range(sectores):
            if lugar in coberturas[sector]:
                sectores_cubiertos.add(sector)

    while len(sectores_cubiertos) < sectores:
        candidatos = []

        for lugar in range(lugares):
            if lugar in solucion_parcial:
                continue

            sectores_que_cubre = set()
            for sector in range(sectores):
                if lugar in coberturas[sector]:
                    sectores_que_cubre.add(sector)

            sectores_no_cubiertos = [s for s in sectores_que_cubre if s
               not in sectores_cubiertos]
            if sectores_no_cubiertos:
                costo_beneficio = costos[lugar] / len(
                   sectores_no_cubiertos)
                candidatos.append((costo_beneficio, lugar,
                   sectores_no_cubiertos))

        candidatos.sort(key=lambda x: x[0])
        if candidatos:
            mejor_candidato = candidatos[0]
            _, mejor_lugar, mejor_cobertura = mejor_candidato
            solucion_parcial.append(mejor_lugar)
            sectores_cubiertos.update(mejor_cobertura)
        else:
            break

    return solucion_parcial
```

No se va a entrar en detalles sobre esta función, pues es solamente una variante del greedy estocástico que recibe una solución parcial y la completa. Sin embargo, esta función es utilizada la función `ejecutar_variante`, que es la columna vertebral del programa.

Listing 3.7: Función Completar Solución

```
def ejecutar_variante(sectores, lugares, costos, coberturas, n, m):
    start_time = time.time()

    solucion = greedy_estocastico(sectores, lugares, costos, coberturas,
       alpha=0.1)
    valor_mejor_solucion = sum(costos[lugar] for lugar in solucion)
    tiempos = [0]
    valores = [valor_mejor_solucion]

    for _ in range(m):
        solucion_parcial = eliminar_lugares(solucion, n)
        solucion = completar_solucion(sectores, lugares, costos,
           coberturas, solucion_parcial)
        valor_nueva_solucion = sum(costos[lugar] for lugar in solucion)

        if valor_nueva_solucion < valor_mejor_solucion:
            valor_mejor_solucion = valor_nueva_solucion

        tiempos.append(time.time() - start_time)
        valores.append(valor_mejor_solucion)

    return tiempos, valores
```

Notar entonces que en este fragmento de código se aplica la variante al greedy estocástico diseñado inicialmente. Para explicar el paso a paso de esta variante se presenta los siguientes pasos en orden cronológico:

1. **Creación de la solución inicial:** El primer paso es generar una solución con la función *greedy_estocástico*. Esta solución sera la base de la iteración actual.

2. **Ciclo de destrucción de la solución:** La solución generada inicialmente se va a destruir y completar nuevamente m veces, almacenando las mejoras a la solución inicial, y retornando finalmente el tiempo que demora y los valores óptimos obtenidos. Es importante señalar que la destrucción de la solución puede eliminar n valores, siendo n un parámetro configurable. Posteriormente se destruirá esta solución m!=n veces.

4. Análisis y Resultados

4.1. Parte 01 - Greedy Determinista vs Estocástico

Para poder realizar un análisis sobre el comportamiento de cada algoritmo en cada benchmark, se adjunta una tabla comparativa de los resultados y tiempos de ejecución.

	Determinista	Estocástico
Valor F.O promedio C1	467	764.9
Tiempo promedio C1 (s)	3.83	3.94
Valor F.O promedio C2	270	735.8
Tiempo promedio C2 (s)	47.75	47.22

Tabla 4.1: Resultados promedios luego de 10 ejecuciones en cada benchmark.

Como se puede observar en la tabla anterior, los tiempos de ejecución de cada greedy son bastante similares, no así sus resultados. El greedy determinista implementado entrega valores mucho más cercanos al óptimo local de cada benchmark, y esto se debe a que va eligiendo en la construcción de la solución el valor óptimo localmente, y también se debe a la evaluación existente en la función miope, realizando una buena elección. En cambio, dentro del greedy estocástico, existe una aleatoriedad en la selección de las opciones para la construcción de la solución, lo que puede llevar a explorar una variedad más amplia dentro del espacio de soluciones, no garantizando la optimalidad local, resultando en una solución menos óptima que la entregada por el greedy determinista.

Cabe destacar que en la implementación del greedy estocástico, se entrega como parámetro la variable que se encarga de otorgar la aleatoriedad de la solución, por lo que se realizan 5 ejecuciones con distinto valor de **alpha** para poder comprender el impacto que genera en la construcción de la solución. A continuación, se adjunta una tabla resumen del promedio de cada valor.

Alpha	Valor promedio G.E C1	Tiempo promedio C1 (s)	Valor promedio G.E C2	Tiempo promedio C2 (s)
0.05	770.0	3.89	751.0	49.72
0.1	731.4	3.76	759.8	50.80
0.2	743.8	4.16	779.2	50.82

Tabla 4.2: Resultados promedios luego de 5 ejecuciones con cada alpha.

Como se puede observar, para el benchmark **C1**, el alpha que permite mejores resultados ejecutando el greedy estocástico es **0.1**, mientras que para el benchmark **C2** el alpha que permite los mejores resultados del greedy es equivalente a **0.05**. Es por esto que en cada ejecución de las variantes y distintos tipos de algoritmos se utiliza el alpha correspondiente a cada benchmark. Además, con estos resultados, se puede generar una idea de la influencia que tiene el elitismo dentro de la selección de candidatos a incluir en la solución.

4.2. Parte 02 - Hill-Climbing & Greedy Estocástico

En esta comparación, mediante múltiples ejecuciones, es que el grupo encuentra una mejora de la solución obtenida por el greedy estocástico. A continuación, se adjuntan los valores obtenidos por el Hill-Climbing para cada benchmark.

	Valor promedio H.C	Tiempo promedio (s)
C1	539.2	347.88
C2	532.0	419.64

Tabla 4.3: Resultados promedios luego de 5 ejecuciones con cada benchmark.

Como se observa en la tabla anterior, el aplicar el algoritmo de **Hill-Climbing**, específicamente la variante *alguna-mejora* con *restarts*, genera una mejora de la solución bastante contundente. Esto se debe a la técnica de explotación que aporta el algoritmo, pudiendo así ahondar más en el espacio de búsqueda local, mientras que los restarts permiten la escapatoria de posibles óptimos locales, pudiendo explorar así otras áreas prometedoras dentro del espacio de búsqueda. Sin embargo, el tiempo de ejecución para ambos archivos refleja también un aumento, y esto se puede explicar por varias razones:

- **Lenguaje utilizado**: Para realizar la actividad se utiliza Python, lenguaje interpretado. Esto suma tiempo de ejecución en comparación con otros lenguajes, por ejemplo **C++**, que es un lenguaje compilado.
- **Generación del vecindario:** La generación del vecindario también influye en la solución, ya que recorre posibles soluciones múltiples veces, agregando así tiempo para encontrar una solución óptima.
- **Operador de movimiento:** El operador de movimiento también juega un rol en el tiempo de ejecución, ya que esta característica define cómo el algoritmo va desplazándose sobre su vecindario.
- **Variante:** A pesar de que la variante utiliza es *alguna-mejora*, los tiempos resultaron elevados al esperado. Sin embargo, mediante el análisis del grupo, se concluye que los tiempos de ejecución con la variante *mejor-mejora* son más elevados, debido a la búsqueda exhaustiva que genera dentro del vecindario, evaluando una por una las posibles soluciones.
- **Variable de aleatoriedad:** Esta variable también influye en el tiempo de ejecución del algoritmo. Aunque no está relacionada directamente con el Hill-Climbing, si lo está con el greedy estocástico, ya que esta es la que permite la selección de un candidato entre el x % mejor.
- **Criterio de término:** En este caso, la variable que tiene relevancia con el tiempo de ejecución es la cantidad máxima de iteraciones. Se realizan pruebas con distintas iteraciones (100, 250 y 500) y resulta en distintos tiempos de ejecución.

A continuación, se adjunta el gráfico de convergencia para una iteración de este algoritmo con los distintos valores de iteraciones máximas.

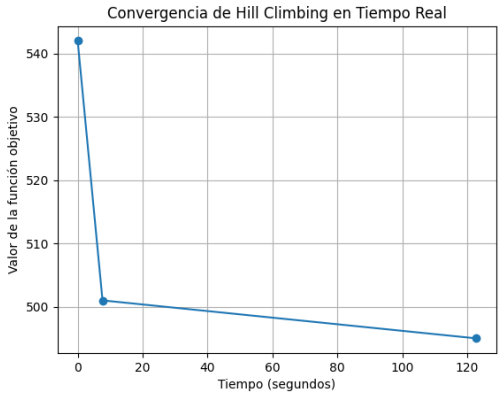

Figura 4.1: Gráfico convergencia Hill Climbing C1 con 100 iteraciones como iteraciones máximas.

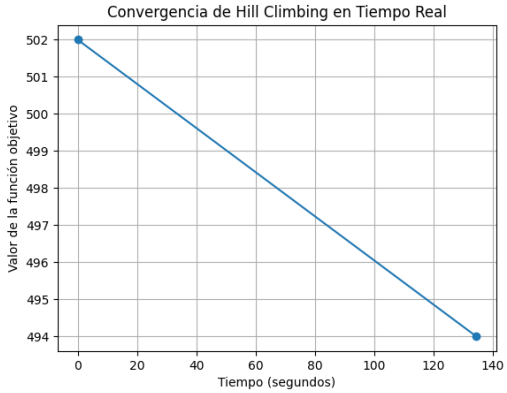

Figura 4.2: Gráfico convergencia Hill Climbing C1 con 250 iteraciones como iteraciones máximas.

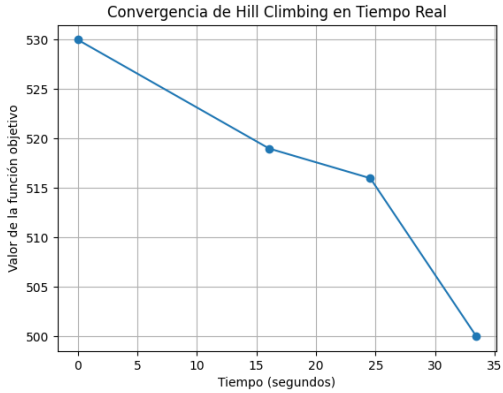

Figura 4.3: Gráfico convergencia Hill Climbing C1 con 500 iteraciones como iteraciones máximas.

Cómo se observa en los gráficos, el tiempo en cada una de las ejecuciones con distinto criterio de término va cambiando, esto debido a la cantidad de veces que se debe ejecutar el algoritmo con todo el proceso involucrado.

4.3. Parte 03 - Variante Greedy Estocástico

Para lograr un mejor análisis de esta variante del *Greedy Estocástico* es que se ejecutara el programa descrito anteriormente 6 veces, cada una con distintos valores para las siguientes variables:

- **n:** Este parámetro indica cuantos lugares se van a eliminar de la solución inicial generada por el greedy estocástico. Estos valores eliminados deben ser luego completados por este algoritmo variante del estocástico inicial.
- **m:** Este parámetro indica cuantas veces se va a destruir y completar nuevamente la solución inicial generada por el greedy estocástico.

De este modo, para las pruebas se hará una combinatoria con los siguientes valores:: n = 10, 20, 30 y m = 15, 30, 45. A continuación se exponen para C1 los gráficos asociados a n=10;m=15 vs n=10;m=45 para luego comparar n=30;m=15 vs n=30;m=45.

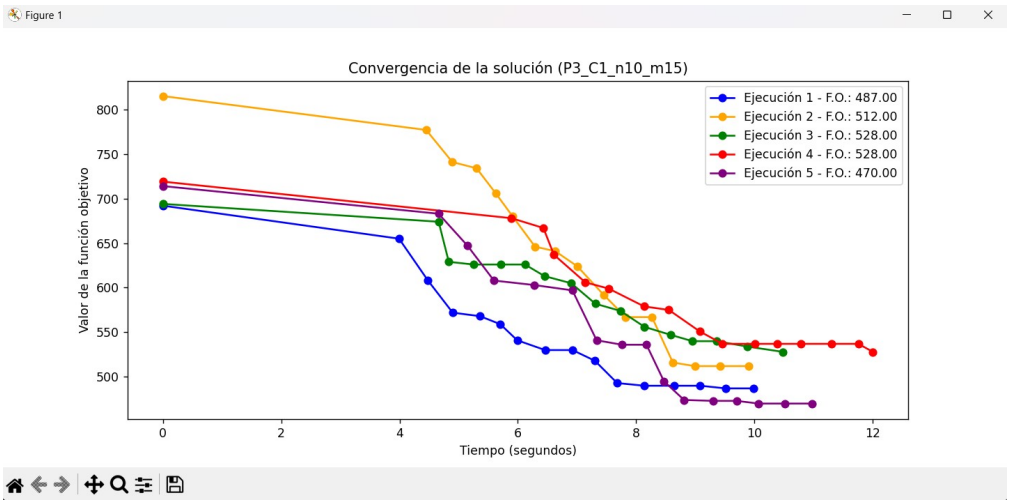

Figura 4.4: Gráfico c1_n10_m_15

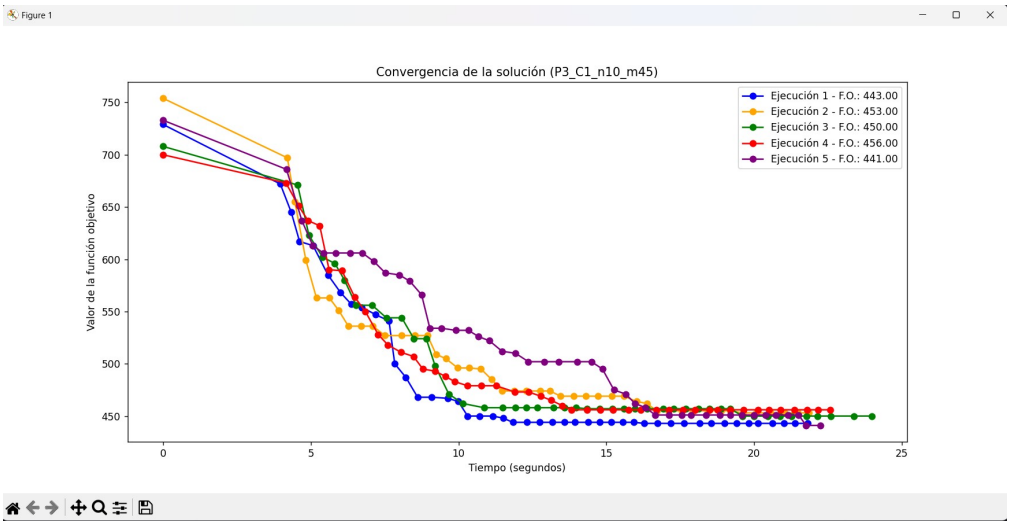

Figura 4.5: Gráfico c1_n10_m_45

En este caso, se eliminan n=10 lugares de la solución inicial y se reconstruye la solución primero 15 y luego 45 veces. Se ve como el mejor valor para la F.O. con m=15 es 470, versus F.O. 441 con m=45. Esto evidencia que al eliminar 10 valores de la solución inicial, a mayor cantidad de reconstrucción de la solución, mejor va a ser el valor obtenido para F.O., considerando que luego de la 18 segundos aproximadamente se estanca y se evidencian pocas mejoras.

Ahora se analizara el caso para n=30 con m=15,45:

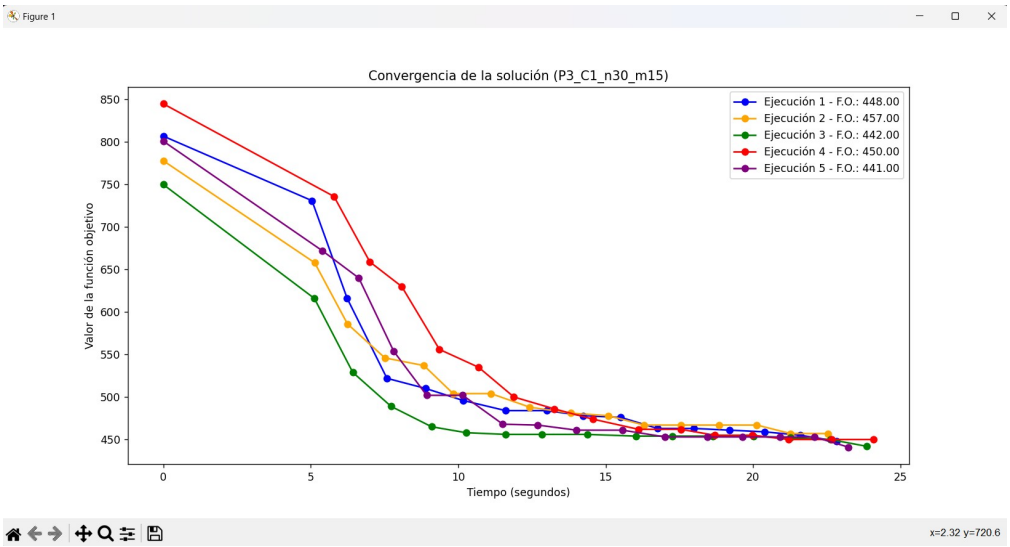

Figura 4.6: Gráfico c1_n30_m_15

En este caso, se ve como con n=30 y m=15 se consigue un valor de F.O. de 441, muy cerca de los 440 alcanzados para n=30 y m45, aunque en este caso el promedio de las iteraciones es menor.

De esta forma, para C1 queda claro que el parámetro principal que se debe variar para obtener el valor mas óptimo de la función objetivo es **m = cantidad de veces que se destruye y reconstruye la solución**, pues si bien es cierto que con n=10 no se escapa mucho del óptimo local encontrado, si varia el espacio de búsqueda a medida que aumenta m, pues la solución se destruye y reconstruye tantas veces que se fuerza un cambio de vecindario, pudiendo encontrar cada vez mejores soluciones. Para este caso, es muy

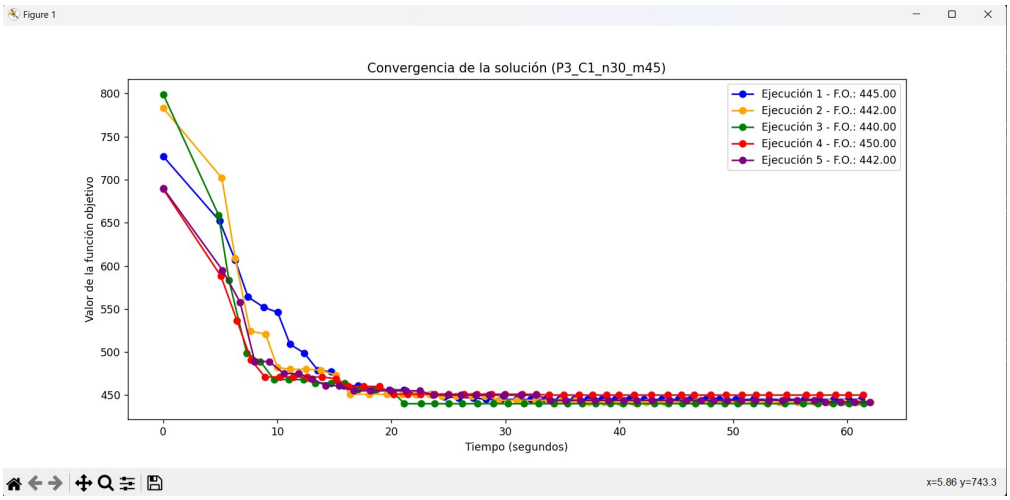

Figura 4.7: Gráfico c1_n30_m_45

importante que los lugares eliminados de la función objetivo sean al azar, de esta forma se garantiza escapar del vecindario en el que se esta inicialmente, y a medida que aumenta m, mas lejos se puede llegar.

Ahora bien, ya se ve claramente la importancia de variar m con respecto a un n fijo, pero: ¿Que pasa con un n mayor? ¿Impactaría positiva o negativamente destruir mas aun la solución inicial, y que implicaría esto? Si contrastamos los resultados expuestos en la figura4.4 versus los obtenidos en la figura4.6, queda en evidencia como el variar el parámetro n y fijar m también afecta significativamente la optimizador de la F.O. obtenida, pues al destruir de una forma mas agresiva la solución inicial se obliga al algoritmo a saltar del vecindario en el que esta trabajando, analizando el espacio de búsqueda de una forma mas variada, y por ende mas completa. Esto en cierto punto es bueno, pues se explora mas y se explota menos, sin embargo, de igual manera se obtienen gratos resultados.

Para el archivo C2, dado que la naturaleza del problema es la misma, lo que implica que el análisis es similar, se ofrece al lector solo la siguiente comparación:

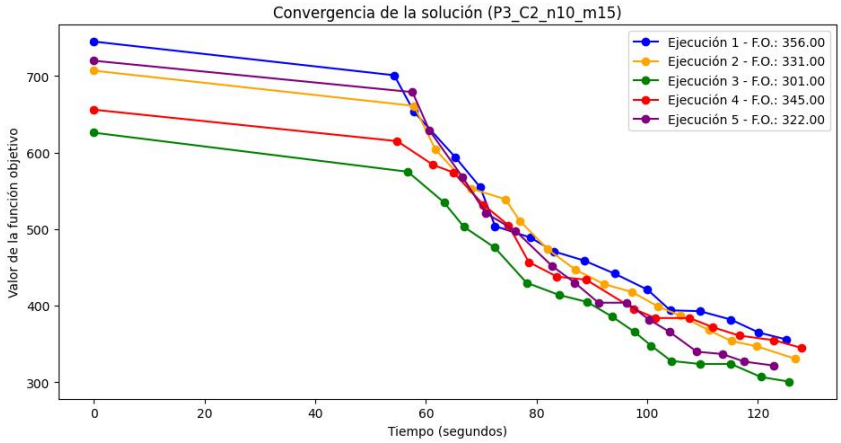

Figura 4.8: Gráfico c1_n10_m_15

Finalmente, y para complementar el análisis de esta parte 3, se presentan ante el lector las siguientes tablas resúmenes de las iteraciones para C1:

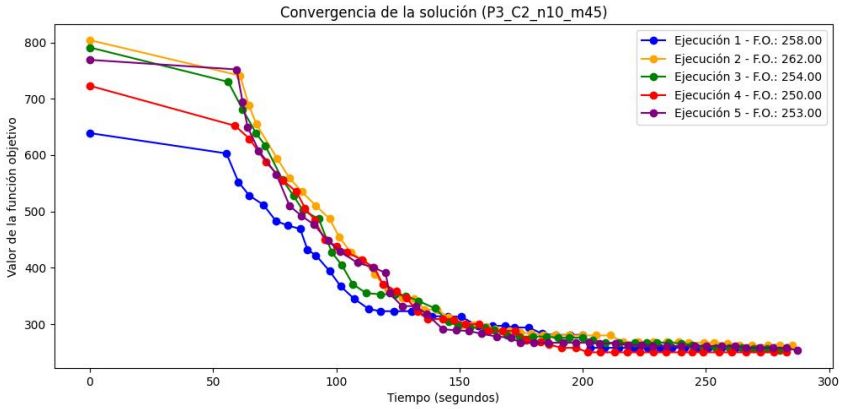

Figura 4.9: Gráfico c1_n10_m_45

n/m	15	30	45
10	10.6	16.6	23.1
20	16.1	29.1	41.1
30	23.31	42.23	61.6

Tabla 4.4: Tiempos promedio para C1 según variación de parámetros

n/m	15	30	45
10	505.0	459.0	451.2
20	460.7	448.6	443.8
30	447.6	445.4	443.8

Tabla 4.5: Promedio valor de la función objetivo para C1 según variación de parámetros

Importante: **Dado que los valores obtenidos para C2 siguen la misma tendencia, no son adjuntadas las tablas para ese archivo, sin embargo todas las pruebas y archivos asociados a C2 se pueden encontrar en el GitHub adjunto a este informe (anexos).**

5. Conclusiones

Dentro de las actividades desarrolladas, el equipo implementa distintas soluciones para la misma problemática. De esta manera, y mediante un análisis y experimentos con el código, es que se concluye y se entiende de mejor manera el funcionamiento y las ventajas que tienen las técnicas incompletas cuando se implementan adecuadamente.

Dentro de la actividad 1, se logra comprender y palpar la diferencia entre la implementación y la importancia de la definición de la variable que genere la aleatoriedad necesaria para que el algoritmo sea estocástico. De igual forma, se genera un mayor acercamiento a la generación de las técnicas constructivas deterministas, que siempre encuentran la mejor solución, ya que van seleccionando siempre el óptimo local en cada paso, a diferencia de la variante estocástica.

Siguiendo con la actividad 2, se logra aterrizar el conocimiento adquirido en clases sobre el Hill-Climbing, algoritmo de búsqueda local, que explota un área dentro del espacio de búsqueda entregado por la solución inicial. Dentro de este enfoque, es que se visualiza además la importancia de entregarle una buena solución inicial a este tipo de técnicas perturbativas, ya que son directamente dependientes de la calidad de la solución inicial para entregar un resultado robusto.

Ya en la parte 3 y final de la actividad, se realizan distintas pruebas con una variante del greedy estocástico utilizado en la parte 1. Para revisar de que forma se obtienen mejores resultados con esta variante es que se hicieron distintas pruebas para distintos valores de n y m, observando que a medida que aumentan estos valores se alcanzan mejores resultados, acercándose cada vez mas al óptimo del problema. Al contrastar los distintos gráficos expuestos en la sección anterior, se puede apreciar como aumentar el valor de n implica que el algoritmo se mueve mas bruscamente dentro del espacio de búsqueda, explorando de mejor manera, y, complementariamente, aumentar el valor de m permite que se explote aún mas el vecindario en el que se esta trabajando. En primera instancia, parece razonable proponer que para acercarse lo mas posible a los óptimos de C1 y C2 basta con aumentar los valores de n y m, explorando y explotando aun mas el espacio de búsqueda casi como lo haría un algoritmo de búsqueda completa, sin embargo, el tener este beneficio implica una desventaja inherente a este tipo de algoritmos: aumentaría el costo computacional, y con ello también aumenta el tiempo de convergencia. De esta forma, se tiene un *trade-off* entre optimizador y tiempo/recursos disponibles, en donde si se quiere llegar a un valor mas cercano al óptimo local se debe invertir necesariamente mas tiempo y recursos.

Finalmente, y con una comparación entre cada técnica, es que se logra visualizar las ventajas y falencias que tiene cada una, por ejemplo, las técnicas deterministas siempre buscan la mejor opción posible, repitiendo el mismo camino cada vez que se ejecutan, mientras que las estocásticas dependen de una distribución de probabilidad para poder seleccionar candidatos para agregar a la solución en construcción; las técnicas perturbativas, en particular Hill-Climbing, tiene variantes (Mejor-Mejora y Alguna-Mejora), las cuáles explotan de distinta manera el vecindario, influyendo así en la calidad de la solución y el tiempo de ejecución del mismo. Ventajas de utilizar la variante alguna mejora sobre mejor mejora corresponden a la eficiencia en tiempo, y a la utilidad que presenta en escenarios de optimización cuando el espacio de búsqueda resulta ser extenso y complejo.

6. Anexos

Repositorio de la actividad: `https://github.com/iJass21/Algoritmos_exactos_y_metaheuristicas/`