3. [Entrega-AD-EC-3] Estimación de búsquedas locales

Grupo1: Naroa Iparraguirre, Xiomara Cáceres y Iraida Abad

21 de enero de 2025

Este documento describe el código para resolver el Problema de Asignación Cuadrática (QAP) utilizando varias técnicas de búsqueda local, como "Best First" y "Greedy", así como métodos para estimar óptimos locales. Se explica cada función del código según su propósito y funcionalidad.

1. Teoria estimación de óptimos locales

1.1. Chao 1984, Chao & Lee 1992, Chao & Bunge 2002

1. Elegir m soluciones al azar
2. Aplicar algoritmo de búsqueda local sobre las soluciones
3. Contar los óptimos locales r
4. Calcular el índice r / m

El índice intenta estimar el número de óptimos locales en el problema. Si todos los óptimos locales son diferentes, entonces el índice r / m será igual a la proporción del espacio de soluciones que está cubierto por los óptimos locales encontrados. Esto implica que al aumentar m, la estimación será más precisa y el valor del índice tenderá a estabilizarse. En cambio, si algunos óptimos locales son iguales, el índice subestima la cantidad total de óptimos distintos.

1.2. Schnabel Census Procedure

1. Elegir m soluciones al azar
2. Comprobar cada solución y verificar si es óptimo local
3. Contar los óptimos locales r
4. Calcular el índice r / m

El porcentaje real de óptimos locales se puede aproximar en ambos casos de manera eficiente cuando el número de muestras m es suficientemente grande. Esto se debe a que a medida que m tiende a infinito ($\lim_{m\to\infty}$), el valor de r/m se estabiliza, proporcionando una representación fiable del porcentaje de óptimos locales en el espacio de soluciones. Para aplicaciones prácticas, es importante equilibrar la cantidad de soluciones probadas (m) con los recursos computacionales disponibles.

2. Funciones Auxiliares

2.1. read_instance_QAP(filepath)

Propósito: Cargar una instancia del problema QAP desde un archivo y devolver los datos correspondientes.

Parámetros:

- filepath: Ruta al archivo que contiene la instancia.

Salida: Una tupla (size, D, H).

- size: Tamaño del problema (número de instalaciones/lugares).
- D: Matriz de distancias entre lugares.
- H: Matriz de flujos entre instalaciones.

Descripción:

1. Abre el archivo y lee el tamaño del problema.
2. Llena las matrices D y H con los datos del archivo.
3. Devuelve los datos procesados.

```python
# QAParen instantzia diskotik irakurtzeko funtzioa
def read_instance_QAP(filepath):
    fp=open(filepath)
    line=fp.readline()
    values=line.split()
    size=int(values[0])
    D=np.zeros((size,size))
    H=np.zeros((size,size))
    for i in range(size):
        line=fp.readline()
        values=line.split()
        for j in range(size):
            D[i][j]=int(values[j])

    for i in range(size,2*size):
        line=fp.readline()
        values=line.split()
        for j in range(size):
            H[i-size][j]=int(values[j])
    fp.close()
    return (size,D,H)
```

2.2. objective_function_QAP(solution, instance)

Propósito: Calcular el valor de la función objetivo para una solución dada.
Parámetros:

- solution: Vector que representa una asignación de instalaciones a lugares.
- instance: Tupla (size, D, H) que representa la instancia del problema.

Salida: El valor de la función objetivo calculado.
Descripción:

1. Recorre todas las combinaciones de instalaciones.
2. Suma los productos de las distancias (matriz D) y los flujos (matriz H) correspondientes a la solución.

```
# Soluzio bat emanik, bere helburu-funtzioaren balioa kalkulatuko
    duen funtzioa.
def objective_function_QAP(solution, instance):
    size=instance[0]
    D=instance[1]
    H=instance[2]
    value=0
    ## kalkulatu soluzio baten helburu funtzioaren balioa goiko
    ekuazioa ikusirik,
    # BETE HEMEN.
    for i in range(size):
        for j in range(size):
            value += D[i][j] * H[solution[i]][solution[j]]
    return value
```

3. Funciones de Generación de Vecinos

3.1. ExchangeVector(V, i, j)

Propósito: Intercambiar dos elementos en un vector.
Parámetros:

- V: Vector de entrada.
- i, j: Índices de los elementos a intercambiar.

Salida: El vector modificado con los elementos intercambiados.

```
def ExchangeVector(V,i,j):
    lag = V[j]
    V[j]=V[i]
    V[i]=lag
    return V
```

3.2. calcularVecinosExchange(N, vectorIni)

Propósito: Generar todos los vecinos posibles de una solución mediante intercambios de elementos.
Parámetros:

- N: Tamaño del vector.
- vectorIni: Solución inicial.

Salida: Una lista con todos los vectores vecinos generados.
Descripción:

1. Recorre todas las combinaciones posibles de pares de índices.
2. Genera un nuevo vector para cada par intercambiado.
3. Devuelve la lista de vecinos.

```
def calcularVecinosExchange(N, vectorIni):
    vecinosExchange=[]
    vectorCopy = vectorIni.copy()
    # print("El vector inicial es as : vector  = ", vectorCopy)
    for i in range(0, N):
        for j in range(i+1, N):
            vector = vectorCopy.copy()
            # print("Esta iteraci n i=", i, ", j=", j)
            vectLag = ExchangeVector(vector,i,j)
            # print("VectLag = ", vectLag)
            vecinosExchange.append(vectLag)
    return vecinosExchange
```

4. Algoritmo Best First

4.1. calcularMejorVecinosExchangeBestFirst(instance, N, vectorIni, valorIni)

Propósito: Encontrar el mejor vecino de una solución inicial utilizando el enfoque "Best First".
Parámetros:

- instance: Instancia del problema.
- N: Tamaño del problema.
- vectorIni: Solución inicial.
- valorIni: Valor de la función objetivo para la solución inicial.

Salida: Una tupla (optimo_global_encontrado, valor_solucion_local, solucion_local).
Descripción:

1. Genera los vecinos de la solución inicial.
2. Calcula el valor de la función objetivo para cada vecino.
3. Devuelve el vecino con el menor valor de la función objetivo.

```
def calcularMejorVecinosExchangeBestFirst(instance, N, vectorIni, valorIni):
    vectorCopy = vectorIni.copy()

    optimo_global_encontrado = False
    valor_solucion_local = valorIni
    solucion_local= vectorIni

    #print("El vector inicial es as : vector  = ", vectorCopy)
    for i in range(0, N):
        for j in range(i+1, N):
            vector = vectorCopy.copy()
            # print("Esta iteraci n i=", i, ", j=", j)
            vectLag = ExchangeVector(vector,i,j)
            valor_solucion_vecina = objective_function_QAP(vectLag, instance)
            # print("COMO VECINO DE ", vectorCopy, "SE HA SACADO EL VECINO: ", vectLag, ", VALOR ", valor_solucion_vecina)
            if valor_solucion_local > valor_solucion_vecina:
                valor_solucion_local = valor_solucion_vecina
                solucion_local = vectLag
                return (optimo_global_encontrado, valor_solucion_local, solucion_local)

    if valorIni == valor_solucion_local:
        optimo_global_encontrado = True

    return (optimo_global_encontrado, valor_solucion_local, solucion_local)
```

4.2. best_first_solution(possible_solution, size, i_max, instance)

Propósito: Iterar el proceso "Best First" hasta alcanzar un óptimo local o superar el número máximo de iteraciones.

Parámetros:

- possible_solution: Solución inicial.
- size: Tamaño del problema.
- i_max: Número máximo de iteraciones.
- instance: Instancia del problema.

Salida: Una tupla con información sobre la última iteración.

Descripción:

1. Actualiza la solución inicial iterativamente usando el mejor vecino.
2. Verifica si se alcanza un óptimo local.
3. Devuelve la solución final y su valor.

```
def best_first_solution(possible_solution, size, i_max, instance):
    i = 0
    optimo_global_encontrado = False

    solucion_global = possible_solution
    valor_solucion_global = objective_function_QAP(solucion_global,
     instance)

    solucion_local = None
    valor_solucion_local = None

    while i<i_max and not optimo_global_encontrado:
        # print("\n BEST FIRST SOLUTION interation number = ", i)
        # print("NODO ACTUAL --> SOLUCI N = ", solucion_global, ",
     VALOR = ", valor_solucion_global)
        (optimo_global_encontrado, valor_solucion_local,
    solucion_local) = calcularMejorVecinosExchangeBestFirst(
    instance, size, solucion_global, valor_solucion_global )
        # print("LO QUE HA DEVUELTO
    calcularMejorVecinosExchangeBestFirst: ")
        # print("      optimo_global_encontrado = ",
    optimo_global_encontrado)
        # print("      valor_solucion_local = ",
    valor_solucion_local)
        # print("      solucion_local = ", solucion_local)
        if valor_solucion_local < valor_solucion_global:
            valor_solucion_global = valor_solucion_local
            solucion_global= solucion_local
        i = i + 1
    return (optimo_global_encontrado, valor_solucion_global,
    solucion_global)
```

5. Algoritmo Greedy

5.1. calcularMejorVecinosExchangeGreedy(instance, N, vectorIni, valorIni)

Propósito: Seleccionar vecinos de forma aleatoria y encontrar una solución mejorada.

Parámetros: Igual a calcularMejorVecinosExchangeBestFirst.

Diferencia Principal: Genera vecinos aleatorios en lugar de generar todos los vecinos.

```
def calcularMejorVecinosExchangeGreedy(instance, N, vectorIni,
    valorIni):
    vectorCopy = vectorIni.copy()

```

```
    optimo_global_encontrado = False
    valor_solucion_local = valorIni
    solucion_local= vectorIni

    #print("El vector inicial es as  : vector  = ", vectorCopy)
    for i in range(0, N):
        for j in range(i+1, N):
            vector = vectorCopy.copy()
            # print("Esta iteraci n i=", i, ", j=", j)
            vectLag = ExchangeVector(vector, math.floor(random.
    uniform(i,j)), math.floor(random.uniform(i,j)))
            valor_solucion_vecina = objective_function_QAP(vectLag,
     instance)
            # print("COMO VECINO DE ", vectorCopy, "SE HA SACADO EL
     VECINO: ", vectLag, ", VALOR ", valor_solucion_vecina)
            if valor_solucion_local > valor_solucion_vecina:
                valor_solucion_local = valor_solucion_vecina
                solucion_local = vectLag

    if valorIni == valor_solucion_local:
        optimo_global_encontrado = True

    return (optimo_global_encontrado, valor_solucion_local,
    solucion_local)
```

5.2. greedy_search_solution(possible_solution, size, i_max, instance)

Propósito: Aplicar el enfoque Greedy iterativamente.
Parámetros: Igual a best_first_solution.
Salida: Igual a best_first_solution.

```
def greedy_search_solution(possible_solution, size, i_max, instance
    ):
    i = 0
    optimo_global_encontrado = False

    solucion_global = possible_solution
    valor_solucion_global = objective_function_QAP(solucion_global,
     instance)

    solucion_local = None
    valor_solucion_local = None

    while i<i_max and not optimo_global_encontrado:
        # print("\n BEST FIRST SOLUTION interation number = ", i)
        # print("NODO ACTUAL --> SOLUCI N = ", solucion_global, ",
     VALOR = ", valor_solucion_global)
        (optimo_global_encontrado, valor_solucion_local,
    solucion_local) = calcularMejorVecinosExchangeGreedy(instance,
    size, solucion_global, valor_solucion_global )
        # print("LO QUE HA DEVUELTO
    calcularMejorVecinosExchangeBestFirst: ")
        # print("      optimo_global_encontrado = ",
    optimo_global_encontrado)
```

```
        # print("      valor_solucion_local = ",
    valor_solucion_local)
        # print("      solucion_local = ", solucion_local)
        if valor_solucion_local < valor_solucion_global:
            valor_solucion_global = valor_solucion_local
            solucion_global= solucion_local
        i = i + 1
    return (optimo_global_encontrado, valor_solucion_global,
    solucion_global)
```

6. Estimación de Óptimos Locales

6.1. ejecutar_Chao_1984_Chao_and_Lee_1992_Chao_and_Bunge_2002(i_max, m, instance, size)

Propósito: Aplicar técnicas basadas en publicaciones académicas para estimar el número de óptimos locales.

Parámetros:

- i_max: Número máximo de iteraciones.
- m: Número de soluciones iniciales aleatorias.
- instance: Instancia del problema.
- size: Tamaño del problema.

Descripción:

1. Genera m soluciones iniciales.
2. Aplica el algoritmo "Best First" a cada una.
3. Cuenta y calcula el índice de óptimos locales encontrados.

```
def ejecutar_Chao_1984_Chao_and_Lee_1992_Chao_and_Bunge_2002(i_max,
     m, instance, size):
    r_soluciones = []
    for i in range(m):
        possible_solution = np.random.permutation(size)
        (optimo_global_encontrado, valor_solucion_global,
    solucion_global) = best_first_solution(possible_solution, size,
     i_max, instance)

        # Verificar si la soluci n ya est  en r_soluciones usando
     np.array_equal
        if not any(np.array_equal(solucion_global,
    existing_solution) for existing_solution in r_soluciones):
            r_soluciones.append(solucion_global)

    r = len(r_soluciones)
    indice = r/m

```

```
    print("1.- Elegir m soluciones al azar, m =",m)
    print("2.- Aplicar Best First sobre las soluciones")
    print("3.- Contar los  ptimos  locales r, r = ", r)
    print("4.- Calcular el  ndice  r / m = ", indice)
```

6.2. ejecutar_schnabel_census_procedure(i_max, m, instance, size)

Propósito: Aplicar la técnica Schnabel para estimar óptimos locales usando el algoritmo Greedy.

Parámetros: Igual a ejecutar_Chao_1984_Chao_and_Lee_1992_Chao_and_Bunge_2002.

Descripción:

1. Genera m soluciones iniciales.
2. Aplica el algoritmo "Greedy" a cada una.
3. Cuenta y calcula el índice de óptimos locales encontrados.

```
    def ejecutar_schnabel_census_procedure(i_max, m, instance, size
    ):
    r = 0
    for i in range(m):
        possible_solution = np.random.permutation(size)
        (optimo_global_encontrado, valor_solucion_global,
    solucion_global) = greedy_search_solution(possible_solution,
    size, i_max, instance)
        if optimo_global_encontrado:
            r+=1
    indice = r/m

    print("1.- Elegir m soluciones al azar, m =",m)
    print("2.- Aplicar greedy sobre las soluciones")
    print("3.- Contar los  ptimos  locales r, r = ", r)
    print("4.- Calcular el  ndice  r / m = ", indice)
```

7. Ejecución Principal

7.1. Bloque Main

1. Carga las instancias del problema desde archivos.
2. Aplica las técnicas de estimación de óptimos locales.
3. Imprime los resultados.

```
#Parametros globales
path1 = "tai5a.dat"
path2 = "tai10a.dat"
path3 = "tai20a.dat"
i_max = 1000
```

```
m = random.randint(100)

print(" \n --- Chao 1984, Chao & Lee 1992, Chao & Bunge 2002 --- ")

#--- tai5a.dat
print(" \n ---- SOLUCIONES tai5a.dat ----")
instance = read_instance_QAP(path1)
size = instance[0]
ejecutar_Chao_1984_Chao_and_Lee_1992_Chao_and_Bunge_2002(i_max, m,
    instance, size)

#--- tai10a.dat
print(" \n ---- SOLUCIONES tai10a.dat ----")
instance = read_instance_QAP(path2)
size = instance[0]
ejecutar_Chao_1984_Chao_and_Lee_1992_Chao_and_Bunge_2002(i_max, m,
    instance, size)

#--- tai20a.dat
print(" \n ---- SOLUCIONES tai20a.dat ----")
instance = read_instance_QAP(path3)
size = instance[0]
ejecutar_Chao_1984_Chao_and_Lee_1992_Chao_and_Bunge_2002(i_max, m,
    instance, size)


print(" \n --- Schnabel Census Procedure --- ")

#--- tai5a.dat
print(" \n ---- SOLUCIONES tai5a.dat ----")
instance = read_instance_QAP(path1)
size = instance[0]
ejecutar_schnabel_census_procedure(i_max, m, instance, size)


#--- tai10a.dat
print(" \n ---- SOLUCIONES tai10a.dat ----")
instance = read_instance_QAP(path2)
size = instance[0]
ejecutar_schnabel_census_procedure(i_max, m, instance, size)

#--- tai20a.dat
print(" \n ---- SOLUCIONES tai20a.dat ----")
instance = read_instance_QAP(path3)
size = instance[0]
ejecutar_schnabel_census_procedure(i_max, m, instance, size)
```

8. Pregunta:

¿Cuál es el valor óptimo de m para que el ratio r/m sea capaz de estimar de manera eficiente el número de óptimos locales?

La elección del valor óptimo de m para que el ratio r/m sea capaz de estimar de manera eficiente el número de óptimos locales depende de la naturaleza del problema y del tamaño del espacio de soluciones. Generalmente, el valor de m

debe ser suficientemente grande para capturar una muestra representativa del espacio de soluciones. Esto asegura que:

- Se encuentren una cantidad significativa de óptimos locales r, proporcionando un cálculo más preciso del ratio r/m.
- La variabilidad en las estimaciones disminuya a medida que m aumenta.

En términos prácticos, el valor óptimo de m puede determinarse experimentalmente mediante iteraciones sucesivas, observando la estabilización del ratio r/m. Idealmente, m debería satisfacer la condición:

$$m \geq k \cdot n$$

donde n es el tamaño del problema (por ejemplo, el número de elementos en el problema de asignación cuadrática) y k es un factor de escala basado en el número esperado de óptimos locales.

En el límite, cuando $m \to \infty$, el ratio r/m convergerá al valor verdadero del porcentaje de óptimos locales en el espacio de soluciones:

$$\lim_{m \to \infty} \frac{r}{m} = porcentaje\_real\_de\_óptimos\_locales$$