Prácticas de Algorítmica

Práctica 3: Ramificación y poda
(parte 1, ensamblaje)

Curso 2023-2024

Objetivos de la práctica

La práctica 3 consta de 4 sesiones:

- Primera parte: 1 sola sesión. Veremos el uso de algoritmos voraces en la resolución del **problema del ensamblaje** utilizando la técnica de ramificación y poda (RyP).
- Seguna parte: 3 sesiones. Trata del problema del **viajante de comercio** (*Traveling Salesman Problem* o TSP) usando también RyP.

En esta primera parte:

- Veremos el problema del ensamblaje que aparece en la sección 9.4 de los apuntes de teoría.
- Se implementarán 3 algoritmos voraces para crear una solución aproximada (ninguno garantiza encontrar la solución óptima).
- Los algoritmos implementados se utilizarán para estudiar el efecto de empezar RyP con una solución inicial.

El problema del ensamblaje

Problema descrito en los apuntes de teoría en la sección 9.4:

- Hay que ensamblar M piezas con **el menor coste posible**.
- El coste de ensamblar la pieza i depende del nº piezas ensambladas.
- Los datos de entrada se resumen en una matriz `costMatrix` de tamaño $M \times M$ con valores positivos (no hace falta que sean enteros). El valor `costMatrix[i,j]` representa el coste de situar la pieza `i` (identificador entre `0` y `M-1`) cuando ya se han ensamblado `j` piezas.
- Las soluciones son tuplas de la forma $(x_0, x_1, \ldots, x_{M-1})$ donde x_i es el nº piezas ya montadas en el momento en que se decide montar la pieza que identificamos con el índice i.
- Función objetivo: $f((x_0, x_1, \ldots, x_{M-1})) = \sum_{0 \leq i < M} \text{costMatrix}[i, x_i]$
- Todas las permutaciones serían factibles, se trata de encontrar una que corresponda a un **coste mínimo** (podría haber empates).
- Se trata de un problema conocido en teoría de grafos, el *"Problema de la asignación"* o *"Assignment problem"* para el que existen algoritmos como Kuhn Munkres.

Generación de instancias

Para generar instancias concretas para una talla dada, vamos a recurrir a la generación de números aleatorios utilizando la siguiente función:

```
import numpy as np
def genera_instancia(M, low=1, high=4, seed=42):
    np.random.seed(seed)
    return np.random.randint(low=low,high=high,
                             size=(M,M),dtype=int)
```

La siguiente línea:

```
costMatrix = genera_instancia(4,high=10)
```

genera una matriz con valores entre 1 y 9 como ésta (cada vez que lo ejecutes dará normalmente otra distinta):

```
array([[7, 3, 7, 2],
       [9, 9, 4, 1],
       [9, 4, 8, 1],
       [3, 4, 8, 4]])
```

Ejemplo: el coste de ensamblar la pieza 0 en la cuarta posición (después de haber ensamblado 3 piezas) es `costMatrix[0,3]` que vale `2`.

Solución naíf

Es trivial obtener una solución porque cualquier permutación de índices entre `0` y `M-1` es una solución válida. Es decir, podríamos hacer algo así:

```
def naive_solution(costMatrix):
    solution = list(range(costMatrix.shape[0]))
    # solution es de la forma [0,1,...M-1]
    return compute_score(costMatrix,solution), solution
```

Donde la función que calcula el *score* de una solución sería:

```
def compute_score(costMatrix, solution):
    return sum(costMatrix[pieza,instante]
               for pieza,instante in enumerate(solution))
```

Es fácil encontrar contraejemplos que muestran que no siempre da la solución óptima: Sin ir más lejos, con el ejemplo de de la *slide* anterior tendríamos la solución `[0,1,2,3]` con *score* $7 + 9 + 8 + 4 = 28$ que puede ser superado fácilmente por otras soluciones. De hecho, la óptima para esta instancia es `[1,2,3,0]` con *score* igual a $3 + 4 + 1 + 3 = 11$.

Actividad 1: Solución voraz

Vamos a plantear otras soluciones de tipo voraz mejores a la trivial:

1. La función `voraz_x_pieza` va pieza por pieza (fila i de la matriz) y elige el valor x_i (el momento de colocación de esa pieza) que resulte más barato de los que siguen disponibles. Es decir, ir fila por fila de la matriz y elegir (para esa fila) la columna menor de las columnas previamente no elegidas.
2. La función `voraz_x_instante` recorre instante por instante (columna de la matriz) y elige para cada uno la pieza (fila de la matriz) que sea más barata de colocar en ese instante (de entre las piezas disponibles).
3. La función `voraz_x_coste` ordena todos valores de la matriz de costMatrix de menor a mayor recordando sus coordenadas. Después las recorre utilizando los valores que correspondan a parejas de pieza e instante disponibles hasta situarlas todas.
4. `voraz_combina` calcula todas las soluciones anteriores y devuelve la mejor.

Actividad 1: Solución voraz

Traza para `voraz_x_pieza` con la matriz del ejemplo:

```
array([[7, 3, 7, 2],
       [9, 9, 4, 1],
       [9, 4, 8, 1],
       [3, 4, 8, 4]])
```

- Recordemos que esta versión va fila por fila.
- Primero eligiríamos la columna 3 para la fila 0, con coste 2.
- A continuación, para la fila 1, la columna 2 (la 3 ya no está disponible), coste 4.
- Para la 3ª pieza/fila, seleccionamos la columna 1 con coste 4.
- Finalmente, para la última pieza/fila, la columna 0 con coste 3.

Solución `[3,2,1,0]` con coste $2 + 4 + 4 + 3 = 13$.

Actividad 1: Solución voraz

Traza de `voraz_x_instante` (columna por columna) con misma matriz:

```
array([[7, 3, 7, 2],
       [9, 9, 4, 1],
       [9, 4, 8, 1],
       [3, 4, 8, 4]])
```

- En el 1er instante (col. 0) conviene situar la pieza (fila) 3 con coste 3.
- En el 2º instante (col. 1) la pieza 0 con coste 3.
- En el 3er instante (col. 2) pieza 1, coste 4.
- En el último instante (col. 3) sólo podemos elegir la pieza 2, coste 1.

Es decir:

Pieza	Columna
0	1
1	2
2	3
3	0

La solución es `[1,2,3,0]` con coste $3 + 4 + 1 + 3 = 11$ (valor óptimo).

Actividad 1: Solución voraz

Traza para `voraz_x_coste` con la matriz del ejemplo:

```
array([[7, 3, 7, 2],
       [9, 9, 4, 1],
       [9, 4, 8, 1],
       [3, 4, 8, 4]])
```

- El menor coste es 1 en (pieza 1, instante 3) (empata con otra pero elegimos esta para desempatar), nos sirve.
- El siguiente coste 1 comparte instante 3 ya usado, no nos sirve.
- El siguiente coste es 2 para (pieza 0, instante 3), tampoco sirve.
- Luego viene coste 3 (pieza 0, instante 1 por desempatar) sí sirve. De momento tenemos piezas `{0,1}`, instantes `{1,3}`.
- El siguiente coste menor es 3 con (pieza 3, instante 0). También nos sirve. Tenemos piezas `{0,1,3}` e instantes `{0,1,3}`.
- Podemos ir mirando costes ordenadamente hasta seleccionar el único válido que es pieza 2 instante 2 con coste 8.

Solución `[1,3,2,0]` con coste $3 + 1 + 8 + 3 = 15$.

Actividad 1: Solución voraz

Una vez completadas las funciones voraces, puedes ejecutar

```
python3 ensamblaje.py -A
```

que ejecuta la función `comparar_algoritmos` y da este resultado:

talla	naif	x_pieza	x_instante	x_coste	combina	RyP
5	10.20	7.30	7.10	6.90	6.90	6.10
6	12.20	9.50	9.00	8.60	8.60	7.40
7	14.00	9.20	9.10	8.70	8.50	8.10
8	16.40	9.80	10.10	9.80	9.70	8.60
9	18.80	11.80	12.10	11.20	11.10	9.50
10	19.70	12.10	12.00	12.20	11.90	10.30
11	22.60	14.20	14.60	14.00	13.80	11.60
12	24.30	14.40	14.30	14.20	14.10	12.10
13	27.20	15.30	15.30	15.20	15.00	13.20
14	28.70	15.70	16.00	15.60	15.50	14.10
15	31.20	17.40	17.10	17.00	16.60	15.10

Se han probado tallas de 5 a 15 y, para cada una, se han generado 10 instancias. La columna `RyP` dar medias inferiores porque es la única que calcula la solución óptima.

Actividad 2: Ramificación y poda

Por desgracia, ninguna de las soluciones voraces anteriores da siempre **una** solución óptima. Por ese te motivo vamos a utilizar RyP para resolver el problema del ensamblaje.

Decimos *una* solución óptima...

...porque el *score* de la solución óptima sí es único, pero pueden existir varias asignaciones que empaten y den el valor óptimo, por lo que la solución óptima no tiene por qué ser única.

¿Significa que las soluciones voraces no sirven para nada?

Afortunadamente sí tienen utilidad, ya que nos permiten inicializar la mejor solución encontrada hasta el momento para que la poda por cota optimista (que comentaremos a continuación) pueda ser utilizada desde el inicio.

Actividad 2: Ramificación y poda

Representación de los estados y del conjunto de estados activos

- Un estado intermedio $(x_0, x_1, \ldots, x_{k-1})$ corresponde a haber montado un nº piezas k que puede ser inferior al total.
- Se representa mediante una lista Python con esos mismos valores.
- El estado inicial será la lista vacía `[]`.
- Un estado solución será una lista de talla M.

Conjunto de estados activos

- El esquema de RyP mantiene un conjunto de estados activos A que inicialmente contiene únicamente el estado `[]`.
- Cada estado de dicho conjunto tiene asociada una cota optimista.
- Ramificación y poda es un algoritmo iterativo y en cada iteración extrae el estado activo más prometedor (menor cota optimista en nuestro caso, al ser un problema de minimización) y lo usa para ramificar.

Actividad 2: Ramificación y poda

El conjunto de estados activos funciona como cola de prioridad y se implementa mediante un *minheap* (estudiado en la asignatura EDA). Vamos a utilizar una biblioteca estándar de python llamada `heapq`:

```
import heapq
A = [] # es una lista Python, representa cjt A vacío
for score,s in [(10,[1]),(3,[2]),(100,[0,1])]:
    heapq.heappush(A,(score,s)) # insertar
while len(A)>0:
    print(A[0], end=' -> ') # A[0] contiene el mínimo
    score,s = heapq.heappop(A) # extraer el mínimo
    print(score,s)
```

Da el siguiente resultado:

```
(3, [2]) -> 3 [2]
(10, [1]) -> 10 [1]
(100, [0, 1]) -> 100 [0, 1]
```

Observa que guardamos tuplas donde el primer campo es el *score* para que así se ordenen por *score* de menor a mayor (nos sirve porque estamos en un problema de **minimización**).

Actividad 2: Ramificación y poda

Código de ramificación y poda (donde `self.x` y `self.fx` son, respectivamente, la mejor solución hasta el momento y su *score*):

```
def solve(self):
    A = [ self.initial_solution() ] # cola de prioridad
    # bucle principal ramificacion y poda (PODA IMPLICITA)
    while len(A)>0 and A[0][0] < self.fx:
        lenA = len(A)
        s_score, s = heapq.heappop(A)
        for child_score, child in self.branch(s_score, s):
            if self.is_complete(child): # si es completo
                # es factible (pq branch solo genera factibles)
                # falta ver si mejora la mejor solucion en curso
                if child_score < self.fx:
                    self.fx, self.x = child_score, child
            else: # no es completo
                # lo metemos en el cjt de estados activos si
                # supera la poda por cota optimista:
                if child_score < self.fx:
                    heapq.heappush(A, (child_score, child) )
    return self.fx, self.x
```

Actividad 2: Ramificación y poda

`solve` es un método de la clase `Ensamblaje`. El constructor recibe la matriz de costes y calcula un vector con la forma más barata de ensamblar cada pieza (para el score de la solución inicial `[]` y para la cota optimista):

```
    def __init__(self, costMatrix, initial_sol = None):
        """
        costMatrix es una matriz numpy MxM con valores positivos
        costMatrix[i,j] es el coste de ensamblar la pieza i cuando ya
        se han ensamblado j piezas.
        """
        self.costMatrix = costMatrix
        self.M = costMatrix.shape[0]
        # la forma más barata de ensamblar la pieza i si podemos
        # elegir el momento de ensamblaje que más nos convenga:
        self.minPieza = [costMatrix[i,:].min() for i in range(self.M)]
        self.x = initial_sol
        if initial_sol is None:
            self.fx = np.inf
        else:
            self.fx = compute_score(costMatrix,initial_sol)
```

Actividad 2: Ramificación y poda

El método `is_complete` es muy fácil, basta con ver si la longitud de la solución parcial alcanza el valor M:

```
def is_complete(self, s):
    """
    s es una solución parcial
    """
    return len(s) == self.M
```

El método `initial_solution` devuelve el *score* asociado a la solución inicial `[]`. Este score tiene 2 componentes:

- La parte conocida es 0
- La parte desconocida es asumir que cada objeto se pone en el instante más barato para él.

Basta con sumar la matriz `self.minCoste`:

```
def initial_solution(self):
    return (sum(self.minPieza),[])
```

Actividad 2: Ramificación y poda

El método `branch` resulta menos trivial. Incluye el cálculo de la cota optimista de los estados generados. Esta cota se calcula de manera **incremental**:

```
def branch(self, s_score, s):
    """
    s_score es el score de s
    s es una solución parcial
    """
    i = len(s) # i es la siguiente pieza a montar, i<M

    # costMatrix[i,j] coste ensamblar objeto i en instante j
    for j in range(self.M): # todos los instantes
        # si j no ha sido utilizado en s
        if j not in s: # NO es la forma más eficiente
                       # al ser lineal con len(s)
            new_score = s_score - self.minPieza[i] + costMatrix[i,j]
            yield (new_score, s + [j])
```

Actividad 2: Ramificación y poda

- En esta actividad debes analizar el efecto de inicializar RyP con una solución inicial y la calidad de dicha solución inicial.
- Una solución inicial permite realizar poda por cota optimista antes de encontrar la primera solución. La calidad de esta solución inicial permite tener más probabilidades de podar.
- Debes crear una función `comparar_sol_inicial` que haga un barrido similar al de `comparar_algoritmos`:

```
for label,function in cjtAlgoritmos.items():
```

y que genere una solución inicial para luego llamar a RyP con ella. Cuando el bucle llegue a `'RyP'` evitar ejeuctarla y utiliza `None,np.inf` para que se lance ramificación y poda sin solución inicial. Debes recopilar todas las estadísticas (nº iteraciones, etc.) para mostrar las medias de cada una para las instancias generadas en cada talla.

Actividad 2: Ramificación y poda

- Ejemplo de la salida tras ejecutar `python3 ensamblaje.py -I` que hace uso de la función `comparar_sol_inicial` a completar:

	iterations					
talla	naif+RyP	x_pieza+RyP	x_instante+RyP	x_coste+RyP	combina+RyP	RyP
5	8.60	7.00	7.40	6.20	6.20	8.60
6	27.60	27.60	25.20	24.10	24.10	27.60
7	33.50	29.70	30.80	27.70	27.00	33.50
8	40.40	36.60	36.60	36.60	36.60	40.40
9	111.90	111.90	111.90	111.90	111.90	111.90
10	33.60	32.60	32.60	32.60	32.60	33.60
11	1320.90	1320.90	1320.90	1320.90	1320.90	1320.90
12	656.70	656.70	656.70	656.70	656.70	656.70
13	1783.90	1448.40	1448.40	1448.40	1448.40	1783.90
14	942.90	938.70	938.70	938.70	938.70	942.90
15	1533.50	1532.00	1532.00	1532.00	1532.00	1533.50

- Igual que aquí se muestran `iterations` debes mostrar el resto de estadísticas (`gen_states`, `podas_opt`,...).
- Tienes esta salida de ejemplo en el fichero `ejemplo_salida.txt`.
- ¿Que medidas de coste se ven más afectadas por el tipo de inicialización?
- Como trabajo opcional puedes incluir la medición del tiempo de ejecución.