Técnicas Algorítmicas – Trabajo Práctico Final

Master in Management + Analytics, Escuela de Negocios, UTDT

Segundo semestre 2020

Observaciones generales:

- El trabajo se debe realizar en grupos de **dos (2) personas**.
- El archivo con el código fuente debe llamarse `tpf.py`, debe correr correctamente en `Python3` y debe estar bien comentado.
- Los reportes pedidos deben presentarse en un informe en formato PDF, llamado `tpf.pdf`.
- Ambos archivos (`tpf.py` y `tpf.pdf`) deben subirse a la *Entrega del TPF* en la página de la materia en el campus virtual.
- La fecha límite de entrega es el **martes 1º de diciembre a las 23:55**. Los TPs entregados fuera de término serán aceptados, pero la demora incidirá negativamente en la nota.

En una materia universitaria, los estudiantes deben realizar en forma individual un trabajo de investigación sobre un tópico. Hay N estudiantes inscriptos y N tópicos disponibles. Por simplicidad, tanto los estudiantes como los tópicos se identifican mediante números de 1 a N.

Cada estudiante ha presentado una lista con los N tópicos, ordenados según sus preferencias. Se quiere asignar a cada estudiante un tópico, de manera de satisfacer lo máximo posible sus preferencias declaradas.

Formalmente, llamamos $\mathcal{A}$ a una asignación de tópicos a estudiantes, tal que a cada estudiante se le asigna exactamente un tópico y que cada tópico se asigna a exactamente un estudiante. Definimos el *costo* de una *Asignación* como:

$$\mathit{costo}(\mathcal{A}) = \sum_{e} \mathit{ranking}_e\big(\mathcal{A}[e]\big)$$

donde $\mathcal{A}[e]$ es el tópico asignado al estudiante e, y $\mathit{ranking}_e(t)$ es la posición del tópico t en las preferencias de e, comenzando en 0. Luego, decimos que una *Asignación* es óptima si tiene costo mínimo.

Ejemplo: Supongamos que $N = 3$ y que las preferencias de los estudiantes 1, 2 y 3 son $[2, 1, 3]$, $[1, 2, 3]$ y $[1, 3, 2]$, respectivamente. Es decir, el estudiante 1 prefiere el tópico 2, seguido del tópico 1 y en último lugar por el tópico 3. Para estas preferencias, la *Asignación* $\mathcal{A}$ definida como $\mathcal{A}(1) = 2$, $\mathcal{A}(2) = 1$, $\mathcal{A}(3) = 3$, tiene costo igual a 1. Además, es óptima, porque es fácil ver que no existe una *Asignación* con costo menor que 1.

Entonces, el problema a resolver consiste en, dadas las preferencias de tópicos declaradas por N estudiantes, encontrar una *Asignación* con el menor costo posible.

Para resolver este problema se cuenta con una implementación del tipo `Preferencias`, que encapsula las preferencias de tópicos declaradas por los estudiantes de la materia, con las siguientes operaciones:

⋆ `Preferencias(nombre)`: Crea una nueva instancia del tipo, según el nombre indicado. Los nombres válidos son: `Ejemplo3`, `Ejemplo5`, `Ejemplo10`, `Ejemplo12`, `Ejemplo15` y `Ejemplo50`, que inicializan distintos ejemplos de preferencias, con $N = 3$, 5, 10, 12, 15 y 50, respectivamente.

⋆ `pr.estudiantes()`: Devuelve el conjunto de estudiantes de la materia (es decir, el conjunto $\{1, 2, ..., N\}$).

- ⋆ `pr.topicos()`: Devuelve el conjunto de tópicos disponibles (es decir, el conjunto $\{1, 2, ..., N\}$).
- ⋆ `pr.preferencias_del_estudiante(e)`: Devuelve una lista con los tópicos ordenados según las preferencias declaradas por el estudiante `e`.
- ⋆ `pr.ranking_de_topico_para_estudiante(t, e)`: Devuelve la posición del tópico `t` en las preferencias del estudiante `e`, tal que para el primer tópico se devuelve 0, y para el último $N - 1$.

También se cuenta con una implementación del tipo `Planilla`, que representa una planilla en donde se puede asignar un tópico a cada estudiante. Una `Planilla` se dice que está *completa* cuando se corresponde con alguna *Asignación* posible, es decir, cuando cada estudiante tiene asignado un tópico y cada tópico fue asignado a un estudiante. El tipo `Planilla` tiene definidas las siguientes operaciones:

- ⋆ `Planilla(preferencias)`: Crea una nueva planilla, indicando las preferencias de los estudiantes con un elemento del tipo `Preferencias`.
- ⋆ `pl.asignar(e, t)`: Asigna el tópico `t` al estudiante `e`. No hace nada si `e` ya tenía tópico asignado, o bien si `t` ya estaba asignado a alguien.
- ⋆ `pl.desasignar(e, t)`: Borra la asignación del tópico `t` con el estudiante `e`. No hace nada si no existía tal asignación.
- ⋆ `pl.estudiantes_con_topico()`: Devuelve el conjunto de estudiantes que tienen tópico asignado.
- ⋆ `pl.estudiantes_sin_topico()`: Devuelve el conjunto de estudiantes sin tópico asignado.
- ⋆ `pl.topicos_con_estudiante()`: Devuelve el conjunto de tópicos que están asignados a alguien.
- ⋆ `pl.topicos_sin_estudiante()`: Devuelve el conjunto de tópicos que no están asignados a nadie.
- ⋆ `pl.topico_asignado_a_estudiante(e)`: Devuelve el tópico asignado al estudiante `e`. Precondición: `e` debe tener un tópico asignado.
- ⋆ `pl.estudiante_asignado_a_topico(t)`: Devuelve el estudiante que tiene asignado el tópico `t`. Precondición: `t` debe estar asignado a alguien.
- ⋆ `pl.estudiante_libre(e)`: Dice si el estudiante `e` está disponible para asignarle un tópico.
- ⋆ `pl.topico_libre(t)`: Dice si el tópico `t` está disponible para asignar a un estudiante.
- ⋆ `pl.calcular_costo()`: Si la `Planilla pl` está completa, devuelve el costo de la *Asignación* representada por `pl`. Si no, es decir, si la `Planilla pl` posee tópicos o estudiantes sin asignar, devuelve infinito.

El archivo `planilla.py` tiene las implementaciones descriptas arriba, que pueden usarse libremente para este trabajo. También define varias preferencias de ejemplo, como se menciona arriba (`Ejemplo3`, `Ejemplo5`, etc.). A continuación, se muestra un ejemplo de uso.

```
1  from planilla import Preferencias, Planilla
2  prefs = Preferencias("Ejemplo3")
3  print(prefs)          # imprime {1: [2, 1, 3], 2: [1, 2, 3], 3: [1, 3, 2]}
4  print(prefs.estudiantes())           # imprime {1, 2, 3}
5  sol = Planilla(prefs)
6  print(sol)                           # imprime {}
7  sol.asignar(1, 2)
8  print(sol.estudiantes_con_topico())  # imprime {1}
9  print(sol.topicos_sin_estudiante())  # imprime {1, 3}
10 print(sol)                           # imprime {1: 2}
11 sol.asignar(1, 1)                    # el estudiante 1 ya tenia topico
```

```
sol.asignar(3, 2)                    # el topico 2 no estaba disponible
print(sol)                           # imprime {1: 2}
print(sol.calcular_costo())          # imprime inf
print(sol.estudiante_libre(3))       # imprime True
sol.asignar(2, 1)
sol.asignar(3, 3)
print(sol)                           # imprime {1: 2, 2: 1, 3: 3}
print(sol.calcular_costo())          # imprime 1
```

Se pide:

1. Escribir una función `asignar_backtracking(prefs)` con la técnica de *backtracking* que, dado un elemento de tipo `Preferencias`, devuelva un elemento de tipo `Planilla`, con una *Asignación* con costo mínimo.
2. Para cada uno de los ejemplos de `Preferencias` provistos:
   (a) Correr la función hasta que termine su ejecución o hasta haber alcanzado una hora de búsqueda (lo que suceda primero).
   (b) Reportar la solución encontrada, su costo y el tiempo que demandó la ejecución del programa (en caso de haber finalizado en una hora o menos).
3. Implementar los siguientes algoritmos basados en técnicas heurísticas:
   (a) Escribir una función `asignar_greedy(prefs)` que encuentre una solución aproximada al problema (es decir, una *Asignación* para la que se busque minimizar el costo, por más que no se obtenga el costo mínimo) usando la heurística *greedy*.
   (b) Escribir una función `asignar_bl(prefs, solucion_inicial)` que encuentre una solución aproximada usando una heurística de *búsqueda local*, tomando como punto de partida una *Asignación* generada de la manera que quieran (por ejemplo, podría ser generada al azar, o con la heurística *greedy*). Explicar en español cuál es la noción de vecindad que utilizan.
   (c) Escribir una función `asignar_bli(prefs, iters)` que repita `iters` veces la heurística de *búsqueda local*, comenzando cada vez con una *Asignación* generada al azar.
4. Para las funciones del punto anterior, medir y reportar los tiempos de ejecución requeridos y el costo de las *Asignaciones* encontradas para cada uno de los ejemplos dados. Comparar con las soluciones encontradas en forma exacta (cuando se haya llegado a una).

Aclaraciones:

- En todos los casos, los algoritmos de *backtracking* y de técnicas heurísticas deben implementarse en el código entregado. No está permitido importar bibliotecas con algoritmos prefabricados.
- Definir todas las funciones auxiliares que se consideren necesarias.
- En el reporte, responder las consignas de manera concisa, preferentemente mediante tablas y/o figuras. No es necesario desarrollar explicaciones ni descripciones extensas.
- El archivo `tpf_template.py` puede servir como punto de partida para resolver el trabajo práctico. Allí se define una función `asignar_random(prefs)`, que genera una *Asignación* en forma aleatoria.