Fiuba
Facultad de Ingeniería
Universidad de Buenos Aires

U.B.A. - FACULTAD DE INGENIERÍA

Departamento de Computación

75.29 Teoría de Algoritmos I

Trabajo Práctico n° 1 - 1er entrega

Segundo cuatrimestre de 2022

Cátedra Podberezski

Padrón, Alumna:
90596, Valeria Magdalena Rocha Bartaburu

Fecha de entrega: 07/09/22

Introducción

El presente trabajo plantea la resolución del problema de seleccionar el mayor número posible de invitados de una lista de personas, cumpliendo la condición de que cada persona invitada sólo puede ser seleccionada si conoce a al menos otras cuatro personas que hayan sido elegidas.
Para la resolución se utilizará un algoritmo Greedy y se analizará la complejidad temporal y espacial de la solución planteada.
Luego se introducirá una nueva condición al problema para estudiar qué ocurre con la complejidad obtenida. Dicha condición consistirá en que para ser elegida, además de cumplir con la consigna antes mencionada, la persona no debe conocer a otros cuatro invitados.

1 - Estrategia Greedy

Para resolver el problema planteado, se toma como precondición que n > 4. En caso contrario no se podría armar la lista de invitados con al menos 4 conocidos.

Se parte de la siguiente idea, la cual luego se reformulará para que cumpla con la menor complejidad temporal posible. Se considera que previamente se dispondrá de la lista de socios con sus datos asociados en una estructura a determinar en el punto 2.

1. Ordenar los socios por cantidad de conocidos que posee de menor a mayor. O(nlogn)
2. Recorrer la lista de socios y por cada uno: O(n)
    1. Si el socio no conoce a más de 3 personas:
        a. Descartarlo. O(1)
        b. Eliminar este socio de la lista de conocidos de los otros socios ya que no asistirá a la fiesta. O(n)
        c. Volver a ordenar como en el paso inicial. O(n logn)
    2. Si el socio conoce a más de 3 personas:
        a. Agregar desde ese socio en adelante toda la lista de invitados

De esta forma la complejidad temporal en el peor caso queda dada por:
O(n logn) + O(n) * (O(n) + O(n logn)) = 0(n^2 * log n)

A continuación se plantea una mejor estrategia, apuntando a disminuir dicha complejidad:

1. Hacer una lista cuyos elementos tengan los siguientes datos: nombre de socio, cantidad de conocidos. O(n)
2. Ordenar lista de menor a mayor utilizando una estructura que me permita ordenar en O(n)
3. Mientras en el diccionario haya una persona que no conoce a más de tres personas, saco al primero de la lista: O(n)
    a. Si el socio no conoce a más de 3 personas:
        Eliminarlo del diccionario. O(1)
        Borrarlo de la lista de todos los otros socios de forma que permita ser O(1)
        Volver a hacer la lista con datos del socio O(n) y ordenar en O(n). 2n entonces me queda O(n)
    b. Si el socio conoce a más de 3 personas:
        b. Agregar desde ese socio en adelante toda la lista de invitados.

De esta forma la complejidad temporal en el peor caso queda dada por:
O(n) + O(n) + O(n) * 2O(n) = 2*O(N) + O(n^2) = O(n^2)

La estrategia realizada es óptima porque en cada paso busca eliminar el socio que menos personas tenga en su lista de conocidos. De esta forma se asegura que al llegar a una persona que conozca a 4 socios, como mínimo, todos ellos estarán incluídos en la lista de invitados final.

La estrategia es Greedy, resuelve un problema de optimización en el que intenta maximizar la cantidad de socios invitados. Se realiza una división en subproblemas y en cada paso se aplica la solución antes mencionada. Al finalizar obtenemos una solución óptima.

2 - Pseudocódigo y estructuras utilizadas

Las estructuras a utilizar son:

- Estructura base: Diccionario cuya clave es el número de socio y su valor una tripleta compuesta por: nombre, cantidad de conocidos, lista de conocidos.
- Lista de tuplas.
- Heap: cola de prioridades *[2]*. Permite tomar el elemento con menor cantidad de conocidos en O(1)

Pseudocódigo:

1. for socio in diccionario de socios: agregar a la lista la tupla (numero de socio, cantidad de conocidos).
2. Hacer un heap para disponer en primer lugar del socio con menor cantidad de socios conocidos.
3. Mientras en el diccionario haya una persona que no conoce a más de tres personas:
    c. Tomo el primer elemento del heap. O(1)
    d. Si el socio no conoce a más de 3 personas:
        Eliminarlo del diccionario.
        Borrarlo de la lista de todos los otros socios de forma que permita ser O(1)
        Volver a hacer la lista de tuplas O(n) y armar un heap O(n)
    e. Si el socio conoce a más de 3 personas:
        c. Agregar desde ese socio en adelante toda la lista de invitados.

3 - Complejidad de la solución

La complejidad temporal fue analizada anteriormente, siendo la menor obtenida: O(n^2)
Hay que tener en cuenta una alternativa que sería usar una lista en lugar de un heap y ordenarla con el algoritmo bucket sort *[6].* En este caso la complejidad del ordenamiento también sería de O(n).

Para la complejidad espacial se debe tener en cuenta que se utilizan las siguientes estructuras:
Un heap: O(n) para crearlo y O(1) para obtener el primer elemento.
Diccionario con una lista dentro: O(n^2)
Lista temporal: O(n)

Lo que se traduce en una complejidad espacial de O(n^2), si no se tiene en cuenta la estructura base de los datos, la complejidad sería O(n).

4 - Algoritmo propuesto

repositorio: https://github.com/valeriarochab/tda-tp1

```
import os
import sys
import heapq

partners = {}

def main():
    arguments = sys.argv[1:]
    n = int(arguments[0])
    file_name = arguments[1]

    if n <= 4:
        print("n debe ser mayor a 4")
        return

    get_partners(file_name)
    partners_list = create_heap()

    while len(partners.items()) >= 5:
        candidate = partners_list[0]
        if candidate[0] <= 3:
            del partners[candidate[1]]
            remove_partner(candidate[1])
            partners_list = create_heap()
        else:
            break

    show_result()


def get_partners(file_name):
    current_path = os.path.dirname(os.path.abspath(__file__))
    with open("{}/{}".format(current_path, file_name)) as fp:
        lines = fp.readlines()
        for line in lines:
            line = line.replace("\n", "")
            elements = line.split(",")
            partner_number = elements[0]
            partner_name = elements[1]
            known_partners = elements[2:]
            known_partners_length = len(known_partners)
```

```
            completed_partners = [0] * int(known_partners[-1])

            for x in known_partners:
                completed_partners[int(x) - 1] = 1

            partners[partner_number] = (partner_name,
known_partners_length, completed_partners)


def create_heap():
    partners_list = [(v[1], k) for k, v in partners.items()]
    heapq.heapify(partners_list)
    return partners_list


def remove_partner(partner_number):
    for key, value in partners.items():
        partner_name = value[0]
        known_partners_length = value[1]
        completed_partners = value[2]
        if int(partner_number) <= len(completed_partners) and
completed_partners[int(partner_number) - 1] == 1:
            known_partners_length = known_partners_length - 1
            completed_partners[int(partner_number) - 1] = 0
            partners[key] = (partner_name, known_partners_length,
completed_partners)


def show_result():
    for partner in partners.items():
        print(partner[0] + ", " + partner[1][0])

main()
```

Ejecución

Para probar el programa se debe contar con Python instalado (recomendado versión 3.X.X). Extraer los archivos del .zip y ejecutar el siguiente comando en una consola/terminal posicionado en el mismo directorio donde se extraen los archivos:
$ python main.py n file_name

El primer parámetro corresponde al nombre del archivo que contiene el código fuente.
El segundo parámetro corresponde a la cantidad n de socios en la lista.
El tercer parámetro corresponde al nombre del archivo que contiene la lista de socios.

Por ejemplo para ejecutar con python3, 9 socios y el archivo de prueba test.txt provisto por la cátedra:
$ python3 main.py 9 test.txt

Ejemplos de prueba

El archivo test.txt corresponde al ejemplo provisto por la cátedra.

Archivo test1.txt
1,Socio A,4,5,8,9,10
2,Socio B,3,5,6,7,11,12,13
3,Socio C,2,5,10,11,12
4,Socio D,1,6,7,8
5,Socio E,1,2,3,11,12
6,Socio F,2,4,7
7,Socio G,2,4,6,11,12,13
8,Socio H,1,4
9,Socio J,1
10,Socio K,1,3
11,Socio L,2,3,5,7,13
12,Socio M,2,3,5,7,13
13,Socio N,2,7,11,12

Resultado esperado:
11, Socio L
13, Socio N
12, Socio M
3, Socio C
2, Socio B
5, Socio E
7, Socio G

Archivo test2.txt
1,Socio A,3,4,5,8,9,10
2,Socio B,4,5,6,7,9
3,Socio C,1,10
4,Socio D,1,2,5,6,7
5,Socio E,1,2,4,6,7,8
6,Socio F,2,4,5,7
7,Socio G,2,4,5,6
8,Socio H,1,5
9,Socio J,1,2
10,Socio K,1,3

Resultado esperado: el socio 1 si bien conoce a más de 4 personas, no conoce a más de 4 que estén invitadas, por eso queda fuera de la lista.

2, Socio B
5, Socio E
4, Socio D
7, Socio G
6, Socio F

5 - Complejidad del programa vs. complejidad del algoritmo

Para la realización del programa se utilizó el lenguaje Python.
Un aspecto importante para que el programa mantenga la complejidad del algoritmo es utilizar estructuras de datos acorde. Para crear el heap a partir de una lista de utilizó la función heapify *[3]* de la biblioteca heapq provista por el lenguaje, que mantiene el acceso en O(1)
Heapify(lista) transforma la lista, in place, en un tiempo lineal *[4]*

Se debe tener en cuenta que el programa agrega un recorrido O(n) para generar el diccionario base y otro para mostrar la solución. Pero para el armado de la estructura base se considera que no afecta la complejidad espacial.

6 - Incorporar una nueva condición al problema

Condición: Sólo puede asistir si NO conoce al menos otras 4 personas invitadas.
Para poder incorporar esta nueva condición se debe agregar un paso antes de generar la lista final de invitados.
Esto impacta en la complejidad planteada anteriormente ya que por cada socio que se va a agregar a la lista final, se debe verificar que no hay ya agregadas otras 4 personas que conoce.

A partir de la lista obtenida con la condición inicial, por cada socio:

1. Ver si la cantidad total de socios a invitar menos la cantidad de socios que conoce el socio analizado es menor que 4:
    a. Si es menor que 4 agregarlo a la lista final de invitados.
    b. Si no, descartarlo y eliminar de la lista base a dicho socio.

Referencias

[1] Algoritmos y Complejidad - Algoritmos greedy
Pablo R. Fillottrani
Depto. Ciencias e Ingeniería de la Computación - Universidad Nacional del Sur
http://www.cs.uns.edu.ar/~prf/teaching/AyC17/downloads/Teoria/Greedy-1x1.pdf

[2] Algorítmica: Heaps y heapsort - Conrado Martínez Universidad Politécnica de Catalunya:
http://algorithmics.lsi.upc.edu/docs/pqueues.pdf

[3] heapq — Algoritmo de colas montículos (heap) - Python.org
https://docs.python.org/es/3/library/heapq.html#

[4] heapq Library - Python
https://github.com/python/cpython/blob/3.8/Lib/heapq.py

[5] Algoritmos y Complejidad - Análisis Amortizado de Estructuras de Datos
Pablo R. Fillottrani
Depto. Ciencias e Ingeniería de la Computación - Universidad Nacional del Sur
http://cs.uns.edu.ar/~prf/teaching/AyC17/downloads/Teoria/AnalisisAmortizado-4x1.pdf

[6] Implementación del Heap Mínimo y del Heap Máximo - Datascience.eu
https://datascience.eu/es/programacion/implementacion-del-heap-minimo-y-del-heap-maximo-como-funciona-todo/

[7] Bucket Sort Algorithm: Overview, Time Complexity - Simplilearn
https://www.simplilearn.com/tutorials/data-structure-tutorial/bucket-sort-algorithm

[8] Teoría de Algoritmos 1 - Metodología greedy
Ing. Víctor Daniel Podberezski.