Universidad de Buenos Aires

Facultad de Ingeniería

75.29 Teoría de Algoritmos

Trabajo Práctico 2

Integrantes

- Andújar, Martín
- Moriello, Khalil Alejandro
- Gavrilov, Seva

Primer cuatrimestre de 2018

Índice

1. Parte 1: Spy Vs Spy

1.1. Consigna

Dos Agentes secretos intentan hacerse con unos informes clasificados. El espía 1 - que es el que los tiene - se encuentra en un punto de la ciudad escondido y tiene que ir al aeropuerto. El espía 2 se encuentra en otro punto de la ciudad y desea interceptarlo. Para eso tiene que llegar al aeropuerto antes que su rival y emboscarlo.

Dado un mapa de una ciudad, las ubicaciones de los espías y el aeropuerto determine quién se quedará con los informes. Repita el procedimiento pero introduciendo costos en los caminos. Analice la complejidad añadida si se solicita retornar el camino realizado por cada espía en los pasos 1 y 2. Realiza los pasos 1 y 2 nuevamente retornando como salida de ejecución los caminos realizados por cada espía. El mapa de la ciudad:

El mapa de la ciudad y sus caminos esta representados por un grafo que se almacena en un archivo con formato texto. Cada vértice del grafo está representado por 2 coordenadas: x e y. Estas coordenadas solo toman valores numéricos enteros y positivos. Las conexiones entre los puntos del grafo se almacenan una por línea del archivo "mapa.coords" de la forma:

p1.x p1.y - p2.x p2.y

Ejemplo:

```
10 3 - 6 5
4 134 - 9 100
```

Para el punto 1 considere simplemente la cantidad de puntos del mapa recorridos como la distancia Para el punto 2, el costo de cada camino está dado por la distancia euclídea de cada tramo. Adicionalmente el programa recibirá por línea de comandos las posiciones del espía 1, 2 y el aeropuerto que deberán ser alguno de los puntos del mapa (se ingresaran cada uno como un numero entero positivo equivalente a la posición en el archivo del mapa de un punto del grafo).

Los programas deben funcionar para cualquier mapa de ciudad. Los algoritmos deben ser eficientes y adecuados al tipo de problema.

1.2. Algoritmo de resolución

1.2.1. Uso de Algoritmo

Se aplica un algoritmo de Dijkstra con cola de prioridad. Este algoritmo consiste en aplicar un Dijkstra para encontrar el camino mínimo entre varios vértices pero con la variante de agregar una cola de prioridad.

1.2.2. Explicacion

En la cola de prioridad se hace un push como primer elemento de una tupla de (nodo, distancia). Se extrae de esta estructura esta tupla . A patir de eso, se busca el nodo más cercano y se hace un push en la cola de prioridad de la tupla (nodo, distancia). Se recorre así el grafo conexo, guardando en la cola las distancias mínimas que fue encontrando. Se realiza este procedimiento hasta que la cola esté vacía.

1.2.3. Justificación del algoritmo

En un Dijkstra común para este problema se encontraría la solución esperada pero es ineficiente porque en el peor de los casos que los espías estén bastante alejados entre sí y entre el aeropuerto, es de

$$O(|E| + |V|^2)$$

Al agregarse una estructura nueva que en nuestro caso es la cola de prioridad, se permite guardar el grafo con las mínimas distancias y se llega a una complejidad de

$$O(Elogv)$$

en el peor caso

1.2.4. Resolución

El código del algoritmo se puede encontrar en los archivos `Grafo.py` y `Nodo.py`, y se puede ejecutar el problema, modificando el archivo `mapa.coords` y llamando al script `TP2-1.py` de la siguiente forma:

```
python TP2-1.py
```

Si el contenido de `mapa.coords` es el siguiente:

```
10 3 - 6 5
6 5 - 5 4
5 4 - 4 134
10 3 - 4 134
4 134 - 5 34
5 34 - 9 100
```

La salida de esa ejecución es la siguiente:

```
Espia 1 ubicado en (6,5)
Espia 2 ubicado en (10,3)
Aeropuerto ubicado en (9,100)
----------------------------
distancia de (6,5) a (4,134):131.41805965932517
distancia de (6,5) a (9,100):297.5441605535976
distancia de (6,5) a (5,4):1.4142135623730951
distancia de (6,5) a (10,3): infinito
distancia de (6,5) a (5,34):231.4230595343314
distancia de (6,5) a (6,5):0
distancia de (10,3) a (4,134):131.13733259449805
distancia de (10,3) a (9,100):297.2634334887705
distancia de (10,3) a (5,4):5.8863495173726745
distancia de (10,3) a (10,3):0
distancia de (10,3) a (5,34):231.1423324695043
distancia de (10,3) a (6,5):4.47213595499958
El ganador es el espia 2 y la ruta que recorrio fue (9,100) (5,34) (4,134) (10,3)
```

2. Parte 2

2.1. Ejercicio 1

2.1.1. Consigna

Dadas dos cadenas de texto S1 y S2 de longitud n, se desea determinar si la segunda es una rotación cíclica de la primera.

Ejemplo: DABRAABRACA es una rotación cíclica de ABRACADABRA

1. Resolverlo por fuerza bruta: Partiendo de S2 ir realizando rotación de 1 caracter y evaluando si son iguales. Determinar la complejidad del algoritmo.
2. Modificar la solución anterior del problema para que utilice el algoritmo de KMP (Knuth, Morris, Pratt) o Boyer-Moore. Explique por qué es una reducción y analice la complejidad resultante.
3. Evaluar si es posible mejorar la segunda solución llevándola a tener que realizar una única ejecución del algoritmo KMP (o Boyer-Moore). En caso afirmativo, vuelva a calcular la complejidad.

2.1.2. Resolución

2.1.2.1. Algoritmo de fuerza bruta

El algoritmo de fuerza bruta consiste en ir rotando el string (removiendo el primer caracter y poniendolo al final), y comparando con el segundo string. Si después de realizar `n` (donde `n` es es el tamaño de la primera string) comparaciones, ninguna comparación dió verdadero - las cadenas no son rotaciones cíclicas. La comparación se realiza recorriendo la cadena en toda su longitud y comparando cada elemento de una cadena con la otra.

Cabe aclarar que para que dos cadenas sean rotaciones cíclicas, necesariamente tienen que tener la misma cantidad de caracteres, entonces tanto esta implementación como la siguiente verifica esta condición y no hace procesamiento extra si da falso.

El algoritmo resuelto por fuerza bruta al entrar al primer loop, tiene una complejidad de $O(n)$ siendo n el tamaño de la cadena. Chequear luego la igualdad de una cadena con otra, en el peor de los casos es $O(n)$. Así que en total se llega como peor caso a $O(n^2)$

El código de esta solución se puede ver en el archivo `fuerzabruta.py`.

2.1.2.2. Algoritmo ineficiente con KMP

El algoritmo Knuth–Morris–Pratt (KMP) se utiliza para encontrar subocurrencias de una cadena de texto en otra. La implementación de ese algoritmo utilizada por nosotros, dadas dos cadenas de caracteres devuelve el índice de ocurrencia de la segunda string en la primera, o 0 en el caso de que no exista. Internamente utiliza una tabla generada a partir de la primer cadena, conocida también como función de fallo, con el objeto de almacenar el prefijo de la cadena más largo que sea a su vez un sufijo. El cálculo de la tabla se realiza en $O(n)$ y la posterior búsqueda de subocurrencias toma $O(n)$. El orden total de esta implementación es de $O(2n) = O(n)$.

El uso más directo del algoritmo KMP para solucionar nuestro problema es reemplazar la comparación de string en implementación de fuerza bruta por la llamada al algoritmo KMP. Si dos strings son iguales, necesariamente cualquiera de las dos es una subocurrencia de la otra. Entonces en este caso estamos haciendo

una reducción de comparación de strings al algoritmo de KMP. Para poder utilizar esta reducción, primero debemos verificar que las cadenas tengas el mismo tamaño.

La conversión de la entrada no es necesaria, ya que la comparación de dos cadenas tiene las mismas dos cadenas que el algoritmo de KMP en nuestra implementación. La salida del algoritmo de KMP nos dice a partir de qué índice una cadena es subocurrencia de la otra, o un índice menor al primero si no hubo subocurrencia. Podemos convertir esta salida en una decisión sobre si las cadenas son iguales en $O(1)$. De esta manera la reducción no altera el orden del algoritmo que reducimos.

El orden de este algoritmo sigue siendo de $O(n^2)$, ya que reemplazamos la comparación de strings por una llamada a KMP, que tiene orden $O(n)$.

El código tanto de esta implementación, como el de la siguiente se puede ver en el archivo `kmp.py`.

Si el código de `kmp.py` se modifica de la siguiente forma:

```
W = "HOLA"
S = "AHOL"
print(es_rotacion_ineficiente(S,W))
```

La salida de la ejecución de `python kmp.py` es la siguiente:

```
True
```

2.1.2.3. Mejora al algoritmo con KMP

Una forma de resolver el problema de deducir si dos strings (X e Y) son rotación cíclica entre si, es buscando si Y es una subocurrencia de una concatenación de X con X. El motivo de esto radica en que X + X contiene todas las rotaciones ciclicas posibles de X, por lo que si Y es una rotación ciclica de X, el algoritmo KMP nos devolvera que el patrón Y está contenido en la cadena X+X.

El orden de este algoritmo es de $O(2n) = O(n)$.

Como ejemplo:

Datos

```
X = HOLA
Rotaciones de HOLA:
HOLA
OLAH
LAHO
AHOL

X + X = HOLAHOLA
Y = OLAH
```

Si el código de `kmp.py` se modifica de la siguiente forma:

```
W = "HOLA"
S = "AHOL"
print(es_rotacion_eficiente(S,W))
```

La salida de la ejecución de `python kmp.py` es la siguiente:

```
True
```

2.2. Ejercicio 2

2.2.1. Consigna

Ustedes son los principales desarrolladores de un producto de software. El dueño multimillonario de la empresa para la que trabajan les pidió que visiten a clientes en n ciudades para discutir detalles técnicos. Él va a cubrirles todos los gastos, y deja que organicen ustedes mismos el itinerario. Como no pagan de sus bolsillos el costo de los aviones y no se llevan tan bien con el dueño, quieren conseguir la mayor cantidad posible de millas como viajeros frecuentes de la aerolínea.

Algunas reglas:

- Todos los vuelos son directos.
- Viajan siempre en grupo, todos juntos.
- Los precios de los vuelos son fijos (no cambian según el día que vuelen).
- Las millas que les da la aerolínea equivalen a la distancia (euclídea) entre las ciudades.
- Van solo una vez a cada ciudad, y visitan ahí a todos los clientes de la misma.
- Empiezan y terminan en la ciudad donde viven y trabajan.

Dar un algoritmo eficiente (pseudocódigo de alto nivel) para definir en qué orden van a visitar las n ciudades, o bien demostrar que el problema es difícil (NP-Completo o NP-Hard).

2.2.2. Resolución

2.2.2.1. Suposiciones

Bajo la consigna del enunciado deducimos que conseguir la mayor cantidad de millas equivale a maximizar la distancia recorrida entre ciudades. Para completar la consigna, asumimos acertado agregar las siguientes suposiciones (después de consensuarlas con los ayudantes de la materia):

- Todas las ciudades están conectadas.
- De cualquier ciudad se puede ir a cualquier otra ciudad.

2.2.2.2. Análisis del problema

Podemos representar el problema como un grafo de la siguiente manera: las ciudades son nodos de ese grafo, y las aristas entre los nodos son las conexiones entre las ciudades correspondientes. Cada arista tiene como peso la distancia euclídea entre las ciudades correspondientes. Como todas las ciudades están conectadas, dicho grafo es completo, y como de cualquier ciudad se puede ir a cualquier otra, el grafo es no direccionado.

Dada dicha representación, el problema consiste en problema planteado es equivalente a encontrar un circuito que visita cada nodo exactamente una vez (circuito simple) y que maximiza el peso total del recorrido. Lo que buscamos es un circuito, y no un camino, por la necesidad de volver a la ciudad en la que arrancamos el recorrido.

Un problema ya conocido con una consigna muy parecida es el *problema del viajante* (TSP de sus siglas en inglés - Travelling Salesman Problem). El enunciado del problema de TSP es: dada una lista de ciudades, y las distancias entre ellas, encontrar el circuito (camino, que arranca y termina en la misma ciudad) de distancia mínima que visita todas las ciudades exactamente una vez. Es conocido que este problema es NP-Hard (su versión de decisión es NP-Complete, pero a nosotros nos interesa su versión de optimización). A continuación vamos a demostrar que nuestro problema es NP-Hard usando TSP.

2.2.2.3. Demostración de NP-Hardness

Un problema es NP-Hard si cualquier problema NP se puede reducir a ese problema en tiempo polinomial. Para demostrar que nuestro problema es NP-Hard, vamos a reducir el problema del viajante a él, y ver que esa reducción es lineal.

Supongamos que existe un algoritmo que resuelve nuestro problema. Ese algoritmo dado un grafo completo $G = (V, E)$ no direccionado con pesos, devuelve un circuito simple sobre ese grafo, en el que el peso total es máximo. Para obtener la solución del TSP sobre el mismo grafo G (es decir, obtener el un circuito simple que minimiza el peso total) usando nuestro algoritmo, basta con asignarle peso $K - a_i$ a las aristas de E, donde a_i es el peso inicial de la arista e_i, y K es un número suficientemente grande, como para ser mayor que cualquiera de los pesos existentes (para que no queden pesos negativos).

Vamos a obtener el conjunto de aristas E', donde cada arista tiene el peso "invertido" respecto a E (por ejemplo, la arista de mayor peso de E' es la arista de menor peso de E). Aplicando nuestro algoritmo a $G' = (V, E')$ vamos a obtener el circuito que maximiza el peso total sobre G'. Resulta ser que ese mismo circuito es el que minimiza el peso total sobre G, por la forma de la que construimos G'.

Como la conversión de aristas que realizamos para la reducción se puede hacer en tiempo lineal, ese paso no altera el orden total del algoritmo, por ende podemos afirmar que **nuestro problema es NP-Hard**.

2.2.2.4. Investigación adicional

Ya vimos que el problema es NP-Hard. Podríamos decir que nuestro problema es *NP-Completo*? Para eso, lo único que nos quedaría demostrar es que es NP, es decir, que dada una solución a nuestro problema existe un algoritmo que puede comprobar si es la solución óptima en tiempo polinomial.

Por la similitud al problema del viajante en su versión de optimización, para nuestro problema podemos afirmar que decir si un circuito es el que maximiza la distancia recorrida no es más *fácil* que encontrar el circuito óptimo. Por lo cual no podemos decir que es NP-Completo.

Durante la investigación sobre el problema nos encontramos con que existen algunos algoritmos polinomiales para encontrar el circuito con peso máximo, para la que es muy difícil encontrar un contraejemplo. En el campo de los problemas NP-Completos o NP-Hard esos algoritmos se conocen como heurísticas, y si bien no son precisos, suelen dar una solución óptima en bastantes casos. Para algunas heurísticas puede llegar a ser bastante complicado encontrar contraejemplos. En particular, en nuestro caso una dificultad adicional es que los pesos no son arbitrarios, sino son distancias euclídeas. Entonces cualquier conjunto de ciudades se podría representar en un plano euclídeo, y por ende a cualquier herística se le debería poder encontrar un contraejemplo gráficamente.

Un algoritmo muy simple para el que no pudimos encontrar un contraejemplo es el siguiente: del conjunto de las aristas ir seleccionando las de peso más grande, y agregándolas a la solución, siempre y cuando esa arista se pueda conectar (que la suma de las aristas incidentes en los nodos que conecta la arista en cuestión sea menor a 2).

2.3. Ejercicio 3

2.3.1. Consigna

Una universidad quiere dictar un conjunto de cursos C1, C2 ... Cn donde cada curso se puede dar solo en el intervalo de tiempo Ti, ya que los docentes tienen poca flexibilidad horaria. Puede que varios cursos se den a la vez, por ejemplo el curso 1 puede dictarse de 3 a 6 y el curso 2 de 4 a 8. Conocemos el horario de inicio y finalización de cada uno de los cursos. El objetivo es ver cuál es la menor cantidad de aulas necesarias para acomodar todos los cursos (suponer que todas las aulas son iguales).

1) Dar un algoritmo eficiente (pseudocódigo de alto nivel) que resuelva el problema.
2) Reducir el problema a una instancia de coloreo de grafos, en su versión de optimización (minimizar la cantidad de colores).
3) A partir de los dos puntos anteriores, ¿podemos asegurar que P = NP? ¿Por qué?

2.3.2. Algoritmo eficiente

Se está buscando un algoritmo que, dado como entrada un set de intervalos correspondientes al tiempo en el que se pueden dar los cursos, definir cual es la cantidad mínima de aulas, necesarias para dar todos los cursos. Cada intervalo *i* es un par *(s(i), f(i))*, donde *s(i)* es el tiempo de comienzo del intervalo, y *f(i)* es el tiempo de finalización de ese intervalo. Un intervalo *j* se superpone con un intervalo *i* si existe algún punto *t(j)* del intervalo *j* tal que *s(i) <= t(j) <= f(i)*.

Los cursos superpuestos se tienen que dar en aulas diferentes. La cantidad de aulas buscada va a ser igual a la cantidad máxima de cursos superpuestos en algún momento en la línea de tiempo, por ende un algoritmo que detecte la máxima cantidad de superposiciones nos dará como resultado la cantidad mínima de aulas que se van a necesitar.

Una solución polinomial sería una variante simplificada del algoritmo para problema de coloreo de intervalos. La idea es recorrer los intervalos, ordenandos por su tiempo de comienzo, y ver con cuantos intervalos previos se superpone el intervalo actual. El pseudocódigo de esa solución es el siguiente:

```
def minima_cantidad_recursos(intervalos)
    intervalos_ordenados := ordenar intervalos por tiempo de comienzo

    resultado := 0
    para cada intervalo j en intervalos_ordenados:
        cantidad_de_recursos := 0
        para cada intervalo i en intervalos_ordenados que le precede a j:
            si (i se superpone con j):
                cantidad_de_recursos := cantidad_de_recursos + 1

        resultado := max(resultado, cantidad_de_recursos)

    return resultado
```

El ordenamiento de los intervalos se resuelve en $O(nlog(n)$, pero se puede ver por los dos loops internos que el orden total del algoritmo termina siendo $O(n^2)$. De hecho, el orden termina siendo el mismo que el de

un algoritmo por fuerza bruta, en el cada intervalo se compara con todos los demás. Podemos plantear un algoritmo más eficiente, si también consideramos los tiempos de finalización.

La idea detrás del algoritmo mejorado es la siguiente: si los intervalos se ordenan por tiempos de comienzo por un lado, y por tiempos de finalización por el otro, la profundidad de los intervalos anteriores al tiempo de finalización f_i es igual a la cantidad de intervalos que empezaron antes de f_i menos la cantidad de intervalos que terminaron antes de f_i, que es igual al $i-1$ (por el orden de los tiempos de finalización). El pseudocódigo de ese algoritmo es el siguiente:

```
def minima_cantidad_recursos_mejorado(intervalos)
    n := tamaño (intervalos)
    intervalos_comienzo := ordenar intervalos por tiempo de comienzo
    intervalos_fin := ordenar intervalos por tiempo de finalización

    resultado := 0
    profundidad_actual := 0
    i := 0; j := 0

    mientras (i < n y j < n):
        si (intervalos_comienzo[i] < intervalos_fin[j]):
            profundidad_actual := profundidad_actual + 1
            resultado := max(resultado, profundidad_actual)
            i := i + 1
        sino:
            profundidad_actual := profundidad_actual - 1
            j := j + 1

    return resultado
```

La ejecución de esta mejora del algoritmo incluye dos ordenamientos, que se hacen en $O(2 * nlog(n))$, y el procesamiento que toma como máximo $O(2n)$. El orden final es $O(2 * nlog(n) + 2n) = O(nlog(n))$.

2.3.3. Reducción a coloreo de grafos

Un coloreo $X(G)$ es una función que dado un vértice (coloreo de vértices) o una arista (coloreo de aristas) devuelve una etiqueta, tradicionalmente llamada "color". El **Problema de Coloreo de Grafos (GCP)** en su versión de optimización tiene el siguiente enunciado: dado un grafo G, retornar un coloreo $X(G)$ tal que no haya dos elementos adyacentes del grafo G con el mismo color, que utilice la mínima cantidad de colores. GCP en su versión de optimización es conocido *NP-Hard*, por ende no se conoce un algoritmo eficiente que lo solucione.

Suponiendo que existe un algoritmo que soluciona GCP, para reducir nuestro problema (que vamos a llamar mínima cantidad de recursos) al GCP, tendremos que convertir la entrada de nuestro problema a la entrada del GCP, y la salida del GCP a la salida del nuestro problema. El orden de ambas conversiones no tiene que ser mayor al orden del algoritmo que soluciona GCP para que la reducción tenga sentido. Sin embargo, como no se conoce ningún algoritmo eficiente para solucionar GCP, nos alcanza con que las conversiones sean polinomiales.

```
Sea Y la entrada y K la salida de nuestro problema,
```

```
Buscamos una reducción tal que:

C1(Y) -> X
GCP(X) -> R
C2(R) -> K

Donde las funciones de conversión C1 y C2 son polinomiales.
```

Una forma fácil de hacer la conversión de la entrada $C1$ es creando un grafo $G = (V, E)$, donde los vértices son los intervalos. Un par de vértices es conectado por una arista si y solo si los intervalos correspondientes a esos vértices se superponen. Si usamos esa representación como entrada al algoritmo que soluciona GCP (en su versión de coloreo de vértices), vamos a obtener como resultado un coloreo $X(G)$ tal que si dos vértices están conectados por una arista (es decir dos intervalos se superponen), tienen color diferente. Como el algoritmo GCP optimiza la cantidad de colores usados, para realizar la conversión de la salida $C2$ nos basta con contar la cantidad de colores diferentes en el coloreo $X(G)$.

La primera conversión $C1$ se puede hacer por fuerza en $O(n^2)$ (buscando para cada intervalo todos los intervalos con los que se superpone). Posiblemente no sea la conversión más eficiente, pero no nos va a alterar el orden de la reducción final. La segunda conversión se hace en tiempo lineal, recorriendo los vértices de G y consultando su color en $X(G)$.

2.3.4. ¿P = NP?

Redujimos un algoritmo, que sabemos del primer punto que tiene una solución en $O(nlog(n))$ a un problema NP-Hard del cual no se conoce una solución eficiente. La solución final quedó del orden del problema al que redujimos, pero la reducción es válida, porque de esta manera demostramos que nuestro algoritmo no es más difícil que el algoritmo que soluciona GCP.

Sin embargo no podemos hacer la reducción inversa, es decir conseguir una conversión de entrada de GCP a la entrada a nuestro algoritmo, y otra conversión de la salida de nuestro algoritmo, de modo que las conversiones no tengan mayor orden que $O(nlog(n))$. Por ende, no podemos afirmar que $P = NP$.

3. Instalación y ejecución

El desarrollo se utilizó con python 3.5.

3.1. Parte A

Instrucciones para correr el programa. Desde la carpeta parteA se ejecuta de la siguiente manera:

```
python TP2-1.py
```

Para correr se necesita el archivo con el mapa de la ciudad (`mapa.coords`).

3.2. Parte B, Ejercicio 1

Para verificar que dos cadenas son rotaciones entre sí utilizando fuerza bruta, hay que correr el archivo `fuerzabruta.py`:

```
python fuerzabruta.py
```

Para verificar que dos cadenas son rotaciones entre sí utilizando las comparación con KMP (eficiente e ineficiente), hay que correr el archivo `kmp.py`:

```
python kmp.py
```

En cada archivo se pueden modificar las cadenas que se desea saber si son rotación cíclica.