75.29 Teoría de algoritmos
Primer cuatrimestre de 2023

Trabajo Práctico Nro. 1
Greedy

Alumno	Padrón	email
Puglisi Agustin	104245	apuglisi@fi.uba.ar

Índice

1. Introducción

El trabajo presentado corresponde al primer trabajo práctico de la materia teoría de algoritmos. Los detalles sobre el enunciado a resolver están en: https://algoritmos-rw.github.io/tda/2023-1c/tp1/

2. Resolución

2.1. Proponga una estrategia greedy óptima para resolver el problema

Proponga una estrategia greedy óptima para resolver el problema con la menor complejidad espacial y temporal posible. Justifique su optimalidad. Justifique que sea greedy.

La estrategia propuesta es la siguiente: Partimos de la hipótesis de que los puntos tanto de A como de B son distintos, y una vez que se aprovechó un punto no puede volver a utilizarse. Entonces, ordenamos tantos los puntos de A como de B de forma decreciente de acuerdo a su coordenada x.

El algoritmo greedy propuesto entonces recorre todos los puntos de A y B y siempre eligirá el punto bj de B con la mayor coordenada de x que pueda ser matcheado con el punto ai de A actual. De esta forma el algoritmo encuentra siempre un matching máximo que cumple con la definición de dominancia propuesta por el problema.

El algoritmo planteado es greedy porque expresa una estrategia de búsqueda que sigue una heurística en la que se obtiene la solución óptima para cada paso (en este caso el ai sobre el cual estamos parados) y así llegar a una solución óptima global (todos los puntos de A matchearon con algún punto de B que cumpliera la condición). También podemos decir que las decisiones tomadas en este algoritmo son realizadas de acuerdo al punto de A en el que estemos parados en cada iteración, cada uno de estos puntos se tratarán como un subproblema.

Justificación de optimalidad:

1. Supongo que mi algoritmo no produce el matching máximo M: Existe un matching M' que tiene más pares de puntos que el M producido por mi algoritmo.

2. Llamo m a la cantidad de puntos emparejados en M y a m' la cantidad de puntos emparejados por M'.

3. Dado que M es el matching máximo producido por mi algoritmo, m es la cantidad máxima de pares que pueden emparejarse, entonces para que M' tenga más puntos que M, debe tener al menos (m+1) pares emparejados.

4. Llamo (ai, bk) a un match que está en M' y no está en M. Y llamo (ai, bj) a un match en M. bj != bk

5. Como ai domina a bj de acuerdo a la definición de dominancia propuesta por el enunciado, entonces también domina a bk, por lo tanto la coordenada y de bk es menor o igual que la coordenada y de bj, porque están ordenados en la coordenada x de mayor a menor. Como bj tiene la misma coordenada x (ai) que bk, entonces bk es un candidato válido para ser emparejado con ai en M. Sin embargo, en M, ai ya está emparejado con bj, lo cual contradice la suposición de que M es distinta a M'.

6. Por lo tanto ambas soluciones deben ser iguales, y la solución se demuestra óptima.

2.2. Explique cómo implementar algorítmicamente esa estrategia.

Brinde pseudocódigo y estructuras de datos a utilizar.
La implementación del algoritmo propuesto será:

Algorithm 1: Algoritmo para encontrar el matching máximo entre dos sets de puntos

```
Input: A, B sets de listas de puntos, n tamaño de la lista
Output: M la lista de matches, k la cantidad de matches
Ordenamos A y B de manera decreciente por su coordenada en x;
M ← ∅,
k ← 0;
for cada punto a de A ordenado do
    for cada punto b de B ordenado do
        if a domina a b then
            Agregamos a a M;
            Sacamos b del conjunto B;
            Sumamos 1 a k;
            Rompemos el loop;
        end
    end
end
Imprimimos M y k;
```

2.3. Analice complejidad temporal y espacial de su propuesta.

La complejidad temporal del algoritmo primero al momento de ordenar será 2*O(n log (n)), ya que utilizamos la función sorted que está implementada en python con timsort

Como recorremos por cada punto de A, los puntos de B no utilizados, y dado que el tamaño de A y B es n, la complejidad del ciclo for será de $O(n^2)$.

Remover una lista es O(n), y las demás operaciones utilizadas son O(1). Por lo que la complejidad temporal total de mi algoritmo será de $O(n^2 \log (n))$.

La complejidad espacial será O(n) ya que las únicas estructuras de datos adicionales que se utilizan son la lista de matches M que será siempre menor a n, y las listas temporales A y B después de ordenarlas, cuyo tamaño también es n por lo que no aumenta la complejidad espacial.

2.4. Programe su algoritmo y entregue dos ejemplos para su prueba.

El código del algoritmo junto con los 2 ejemplos propuestos se encuentran en el archivo main.py.

Para ejecutar el programa se puede realizar desde una terminal en donde se encuentre la carpeta donde se guarda el archivo y escribir: python3 main.py

La versión de python utilizada en este trabajo es la 3.10.10

2.5. ¿Su programa mantiene la complejidad espacial y temporal de su algoritmo?

La complejidad temporal se mantiene, puesto que el método de ordenamiento implementado en python es O(n log (n)) como se comentó anteriormente. Y en el resto de la resolución la complejidad no se altera con respecto a lo previsto.

En el caso de la complejidad espacial, al necesitarse más listas para poder parsear el arhivo, esta aumenta a $O(n^2)$

2.6. Analice la solución obtenida por su algoritmo. Es única? ¿Qué podría decir sobre ella? ¿Prevalece cierto emparejamiento frente a otro posible?

El algoritmo propuesto tiene solución óptima siempre y cuando los puntos estén ordenados de forma decreciente por su coordenada x, lo cual sucederá ya que se parte de la hipótesis de que todos los puntos son distintos, y además ambos conjuntos tienen la misma cantidad de puntos.

La solución es única porque al agregar cada punto a de A al conjunto M, se elimina de B el primer punto que cumpla con las condiciones de dominio. Esto significa que como los puntos de B están ordenadas de forma decreciente por su coordenada x, siempre se elige al punto de B con la coordeanda y más grande, lo que nos garantiza qeu el emparejamiento resultante sea máximo y la solución única.

El emparejamiento, debido a lo ya mencionado sobre el orden, que prevalece es el par que tenga la coordenada x más grande de A y de B, y que además cumpla con la condición de dominancia.

Sin embargo pueden existir otras soluciones válidas en este problema, el algoritmo podría ordenar de forma creciente, o hacerlo por la coordenada y, y la lógica no cambiaría, la solución seguiría siendo óptima pero se invertiría el orden.

En un primer intento se tomó como criterio ordenar a A de forma decreciente por x, y a B de forma decreciente por y, pero luego encontramos un contraejemplo (mostrado a continuación) donde esto no funcionaba porque devolvía un matching de tamaño 1, cuando la solución óptima tiene tamaño 2:

[(9.0, 6.0), (7.0, 3.0), (5.6, 4.0), (3.0, 1.0), (1.2, 2.0)]
(6.5, 30.0), (2.0, 12.0), (4.0, 11.0), (7.0, 2.0), (8.0, -3.0)

3. Segunda entrega

3.1. Comentarios sobre las correcciones: Parte 1

Propone ordenar ambos conjuntos de manera decreciente por la coordenada X. Luego comienza a tomar puntos da A y empieza a buscar matchearlos con puntos de B. Esta estrategia no es óptima.

Menciona correctamente por qué el algoritmo es greedy.

La demostración de optimalidad tiene, dentro de todo, un buen desarrollo pero no es correcta. En el paso 5: ¿Como puedo asumir que la coordenada Y de bk es menor o igual a la coordenada Y de bj? Es cierto que ai domina a ambos, pero ¿Cómo puedo afirmar algo sobre la coordenada Y entre ambos?

¿Por qué la complejidad temporal es $O(n^2 \log n)$? El análisis por partes es correcto, pero tener $2*O(n \log (n)) + O(n^2)$ termina siendo $O(n^2)$.

Correctamente menciona que la complejidad espacial es O(n).

No se deben mencionar detalles del lenguaje utilizado en el análisis de complejidad teórico. Ordenar elementos es teóricamente O(n log n) porque hay algoritmos que lo resuelven con dicha complejidad. Que luego Python utilice Timsort es cuestión de implementación y es acorde al análisis de la complejidad real.

Menciona brevemente la complejidad real, ¿Por qué la complejidad espacial aumenta a $O(n^2)$? No veo el crecimiento cuadrático en memoria.

En el punto final menciona la existencia de otras alternativas y explica un caso en el que utilizar otro criterio da una solución no óptima. Me gustaría poder ver un análisis un poquito más detallado de otras soluciones válidas. Por ejemplo, como se menciona, utilizando la coordenada Y. ¿Que matchings se priorizarían en cada caso? ¿Que impacto puede tener en nuestras soluciones?

3.1.1. Comentarios sobre las correcciones: Programa

Incluye código, instrucciones de ejecución y 2 ejemplos de prueba. El programa debería poder recibir por parámetro los nombres de archivos y el valor N de elementos, no debería tener que estar cambiando el código para ello.

Falla en el siguiente caso: A = [[5,0 5,0], [3,0 3,0]] B = [[2,0 4,0], [2,8 1,5]]
Devuelve:

Tamaño del matching: 1
(5.0, 5.0) ->(2.8, 1.5)

3.2. Correcciones

3.2.1. Proponga una estrategia greedy óptima para resolver el problema

Proponga una estrategia greedy óptima para resolver el problema con la menor complejidad espacial y temporal posible. Justifique su optimalidad. Justifique que sea greedy.

La estrategia propuesta en la entrega anterior resultó no ser óptima porque al no tener tan en cuenta la coordenada y, emparejaba a cada punto ai de A con el primer punto bj de B que cumpliera la condición, aunque no sea la mejor opción disponible.

La nueva estrategia que se propone será ordenar ambos conjuntos de forma esta vez creciente, y por cada punto de A y cada punto de B, me fijo si la componente de x de B es mayor que la de A, si se da el caso al estar los puntos ordenados por la coordenada x los puntos bj no pueden ser emparejados con el actual punto ai, entonces corto el ciclo interno.

Si la componente y de B es mayor que la de A, continúo al siguiente punto de B.

Para cada posible match calculo la distancia en y del punto ai con respecto al punto bj en módulo

Si la distancia calculada es menor a la mínima distancia calculada hasta el momento, actualizo la mínima distancia calculada, y selecciono al punto ai y bj como "mejor match".

Cuando termino de recorrer los puntos de B, si hay alguno seleccionado entonces será el punto más cercano al ai, por lo que los agrego a la lista de matches, y elimino al punto bj del conjunto.

Si no se seleccionó ningún punto de B entonces no hago nada y continúo al siguiente punto de A.

El algoritmo planteado es greedy porque expresa una estrategia de búsqueda que sigue una heurística en la que se obtiene la solución óptima para cada paso (en este caso el ai sobre el cual estamos parados) y así llegar a una solución óptima global (todos los puntos de A matchearon con algún punto de B que cumpliera la condición). También podemos decir que las decisiones tomadas en este algoritmo son realizadas de acuerdo al punto de A en el que estemos parados en cada iteración, por lo que cada uno de estos puntos serán tratados como un subproblema y una vez que una pareja (ai, bj) es agregada al conjunto M, esta pareja no se rompe.

Justificación de optimalidad:

1. Supongo que mi algoritmo no produce el matching máximo M: Existe un matching M' que tiene más pares de puntos que el M producido por mi algoritmo.
2. Llamo m a la cantidad de puntos emparejados en M y a m' la cantidad de puntos emparejados por M'.
3. Dado que M es el matching máximo producido por mi algoritmo, m es la cantidad máxima de pares que pueden emparejarse, entonces para que M' tenga más puntos que M, debe tener al menos (m+1) pares emparejados.
4. Llamo (ai, bk) a un match que está en M' y no está en M. Y llamo (ai, bj) a un match en M. bj != bk
5. Como ai domina a bj de acuerdo a la definición de dominancia propuesta por el enunciado, entonces también domina a bk, por lo tanto la coordenada y de bk debe ser necesariamente igual o menor a la coordenada y de bj, porque de acuerdo a la heurística del algoritmo Greedy que propuse, se toma al punto más lejano en las x y más cercano en la coordenada vertical para realizar el match. Pero si bk <bj entonces debería haber sido seleccionado por mi algoritmo Greedy por lo que sería absurdo que este sea el caso, y en la posibilidad de ser iguales como bj tiene la misma coordenada x (ai) que bk, entonces bk es un candidato válido para ser emparejado con ai en M. Sin embargo, en M, ai ya está emparejado con bj, lo cual contradice la suposición de que M es distinta a M'.
6. Por lo tanto ambas soluciones deben ser iguales, y la solución se demuestra óptima.

3.2.2. Explique cómo implementar algorítmicamente esa estrategia.

Brinde pseudocódigo y estructuras de datos a utilizar.

La implementación del algoritmo propuesto será:

Algorithm 2: Algoritmo para encontrar el matching máximo entre dos sets de puntos

```
Input: A, B sets de listas de puntos, n tamaño de la lista
Output: M la lista de matches, k la cantidad de matches
Ordenamos A de manera creciente por su coordenada en x;
Ordenamos B de manera creciente por su coordenada en x;
M ← ∅,
k ← 0;
i ← 0;
for cada punto ai de A ordenado do
    for cada punto bj de B ordenado do
        if La coordenada x de bj es mayor que la coordenada x de ai then
            Corto el ciclo;
        end
        if La coordenada y de bj es mayor que la coordenada x de ai then
            Continúo al siguiente punto de B;
        end
        if El punto bj es el más cercano hasta el momento con respecto al punto ai en la
          coordeanda y then
            Guardo el match;
        end
    end
    if Hay algún punto bj matcheado con el punto ai then
        Agrego el match a M;
        Borro bj;
    end
    Continúo al siguiente punto ai
end
Imprimimos M y k;
```

3.2.3. Analice complejidad temporal y espacial de su propuesta.

La complejidad temporal del algoritmo primero al momento de ordenar será 2*O(n log (n)).

Como por cada punto de A se recorren todos los puntos de B al menos una vez, la complejidad será de $O(n^2)$

Remover una lista es O(n), y las demás operaciones utilizadas son O(1). Por lo que la complejidad temporal total de mi algoritmo será de $O(n^2)$

La complejidad espacial será O(n) ya que las únicas estructuras de datos adicionales que se utilizan son la lista de matches M que será siempre menor a n, y las listas temporales A y B después de ordenarlas, cuyo tamaño también es n por lo que no aumenta la complejidad espacial.

3.2.4. Programe su algoritmo y entregue dos ejemplos para su prueba.

El código del algoritmo junto con los 2 ejemplos propuestos se encuentran en el archivo main.py.

Para ejecutar el programa se puede realizar desde una terminal en donde se encuentre la carpeta donde se guarda el archivo y escribir: **python3 main.py rutaArchivo1 rutaArchivo2 n**

La versión de python utilizada en este trabajo es la 3.10.10

3.2.5. ¿Su programa mantiene la complejidad espacial y temporal de su algoritmo?

La complejidad temporal del algoritmo primero al momento de ordenar será O(n log (n)), ya que utilizamos la función sorted que está implementada en python con timsort

La complejidad temporal al momento de eliminar un elemento seguirá siendo O(n) de acuerdo a la documentación ya que en el peor de los casos el punto a eliminar estará en el final del conjunto B y habrá que recorrerlo todo.

Al momento de recorrer los puntos, por cada punto de A se deberá recorrer todos los puntos de B al menos una vez, pero a medida que se van formando los matches el conjunto B disminuye. Por lo que podemos decir que seguirá tendiendo a $O(n^2)$ pese a que cuantos más matches se formen más rápido se realizará el segundo bucle.

La complejidad espacial seguirá siendo O(n)

3.2.6. Analice la solución obtenida por su algoritmo. Es única? ¿Qué podría decir sobre ella? ¿Prevalece cierto emparejamiento frente a otro posible?

La solución que ofrece mi algoritmo es única, es decir no cambia en relación a las ejecuciones. El emparejamiento propuesto recorre los puntos del plano de izquierda a derecha, y prioriza los matcheos en donde la coordenada x sea más chica y la coordenada vertical sean lo más cercanas posibles. Esto se puede observar en el ejemplo 2 propuesto en este trabajo, en el que al ejecutarse se imprimirá por pantalla la siguiente solución:

Tamaño del matching: 6
(A ->B)
(-1.0, 50.0) ->(-2.0, 30.0)
(0.0, 0.0) ->(-1.0, -1.0)
(3.0, 3.0) ->(2.8, 1.5)
(4.0, 5.1) ->(2.0, 4.0)
(4.2, 3.1) ->(2.2, 0.0)
(13.5, 20.0) ->(-15.0, 7.0)

Vemos que en la solución los puntos de A aparecen de forma ordenada y creciente. Esto se debe a la heurística elegida por mi algoritmo Greedy y la forma en la que busca el óptimo para cada punto de A a la hora de formar cada match.

Otro emparejamiento posible sería utilizando la misma lógica general, ordenar los puntos de forma creciente pero esta vez por su coordenada y, y una vez hecho esto matchear en donde la coordenada x sea la más cercana. Realizando esta pequeña modificación, mi algoritmo devuelve lo siguiente:

Tamaño del matching: 6
(A ->B)
(0.0, 0.0) ->(-1.0, -1.0)
(3.0, 3.0) ->(2.8, 1.5)
(4.2, 3.1) ->(2.2, 0.0)
(5.0, 5.0) ->(2.0, 4.0)
(13.5, 20.0) ->(9.1, 6.0)
(-1.0, 50.0) ->(-2.0, 30.0)

Y observamos que a diferencia de lo anterior, esta vez los puntos están ordenados de forma creciente pero de acuerdo a la coordeanda y de los puntos de A, el tamaño de M sigue siendo el máximo posible, pero los conjuntos de puntos que devuelve como resultado en algunos casos cambian. Esto se debe a que si bien el algoritmo no cambió tanto, se prioriza esta vez a la otra coordenada por lo que el resultado sigue siendo óptimo, pero no es el mismo que antes.