Trabajo Práctico N°1
Grupo *undefined*

Integrantes

- Luciano Martin Gamberale. Padrón: 105892.
- Gonzalo Javier Iglesias. Padrón: 107213.
- Alejandro Binker. Padrón: 107056.
- Gabriel La Torre. Padrón: 87796.
- Santiago Nicolás Reynoso Dunjo. Padrón: 107051.

Fecha de entrega:

Lunes 23 de septiembre del 2024

Índice

Parte 1: Maximizando la ganancia del proyecto.

Enunciado

Un proyecto está compuesto por un conjunto de tareas a realizar. Cada tarea puede tener un conjunto de otras que se deben realizar previamente para poder comenzarla. Tenemos personal disponible que solo nos permite hacer una tarea por vez. La ganancia de cada tarea es variable según la cantidad de tareas que ya se realizaron anteriormente. Tenemos conocimiento de la ganancia de cada tarea por cada orden posible de ejecución (independientemente de la duración de cada tarea que es un dato que no afecta al resultado. Si lo considera puede suponer que todas las tareas duran exactamente 1 semana).

Nos solicitan que definamos el orden de todas las tareas para que se obtenga la mayor ganancia posible y a su vez se cumplan las precedencias.

Resolución

1) Explique cómo resolvería el problema utilizando generar y probar. ¿Cuál sería la complejidad?

Para resolver este problema utilizando generar y probar, vamos a crear primero una función generativa con la cual vamos a poder generar todo el espacio posible de soluciones cumpliendo con las condiciones explícitas del problema. En este caso la función generativa nos va a proveer todas las permutaciones posibles de las “n” tareas

Luego debemos verificar la factibilidad de cada solución, dado que las restricciones implícitas nos dicen que cada tarea puede tener un conjunto de tareas previas que se deben realizar. Entonces creamos una función de prueba que asegure que el orden de una solución candidata sea válido.

Finalmente el problema nos pide maximizar la ganancia, ya que según el orden que se realice cada tarea va a tener una ganancia variable. Calculando la ganancia de cada solución factible, vamos a quedarnos con la solución candidata de mayor ganancia hasta que no exista una próxima solución, terminando el algoritmo.

La complejidad de este algoritmo sería que al calcular todo el espacio de soluciones, muchas de ellas pueden no cumplir con las restricciones implícitas del problema lo cual sería ineficiente gastar recursos en explorar órdenes similares a esa solución no factible

2) Proponga y explique una solución del problema mediante Branch and Bound (B&B).

Bajo la condición de que tenemos “n” tareas y por lo tanto también “n” semanas en las cuales las mismas pueden ser resueltas, proponemos:

Se propone un modelo de árbol en donde se parte de una raíz vacía, la cual representa que no se eligió ninguna tarea en la semana cero, Este nodo raíz será ramificado en las “n” tareas que no fueron realizadas. Se filtran aquellas tareas que no cumplan con la función de corte, y luego a las restantes se les calcula la función de costo (estima la máxima ganancia posible que podrías tener al

explorar esa rama). Luego, dado que lo vamos a resolver con best-first search (BeFS), procedemos a explorar el nodo que mayor función de costo nos haya dado. Se genera una nueva ramificación con n-1 elementos y volvemos a aplicar todo lo que comentamos anteriormente. Esto se realiza hasta llegar a la hoja en la n-ésima semana, en la cuál se verifica si supera la solución encontrada hasta el momento. El algoritmo finalizará una vez que no haya nodos posibles por explorar, ya sea por el hecho de haber podado, o por haber recorrido todo el árbol por completo.

Función de Corte

A la tarea actual, se le verifica si las tareas predecesoras a la misma, ya fueron finalizadas. En caso de no ser así, esta rama no debe ser explorada.

Función de Costo

Primero tengo guardadas cuales son las ganancias máximas que se pueden obtener en cada una de las semanas. Para esto obviamente, es necesario haber recorrido una vez el set de datos de las ganancias, para extraer el máximo de cada semana.

Luego, sabiendo que en el i-ésimo nodo, complete “i” tareas y estoy parado en la semana “i”, y además tengo el dato de cuál es la ganancia acumulada hasta el momento, puedo llegar a estimar cuál es la máxima ganancia que podría llegar a tener si recorro esa rama. Esta estimación pondera lógicamente para arriba, suponiendo que en las semanas que no sabes la ganancia, vas a tener la mejor ganancia que existe en la semana, sin considerar si le correspondía a dicha tarea o no.

Esto nos va a permitir comparar la ganancia máxima ponderada, contra la máxima ganancia actual real. En caso de no superarla con esta estimación, podemos desestimar la exploración de esta rama, dado que es imposible lograr un mejor resultado que el actual.

3) Brinde pseudocódigo y estructuras de datos a utilizar para B&B.

```
main:

   tareas las leo del archivo de tareas.txt y las guardo en una lista
   ganancias las leo del archivo ganancias.txt y las guardo en una lista

   estadoTareas = itero tareas y por cada una agrego un elemento con valor falso a una lista vacía
   mejoresGananciasPorSemana = calcularMejoresGananciasPorSemana()

   gananciaMaxima = 0
   ordenTareasGananciaMaxima = []

   calcularMaximaGanancia(0, 0, [])

//============== FUNCIÓN PRINCIPAL ==============//

// funcion backtracking
func calcularMaximaGanancia(nroSemana, gananciaPrevia, ordenTareasRealizadas):

   // estadosDescendientes
   tareasNoRealizadas = obtenerTareasNoRealizadas()

   // descendientes
   tareasNoRealizadasAExplorar = []
   por cada idTarea de tareasNoRealizadas:
      // propiedad de corte
      if todasTareasPreviasRealizadas(idTarea):
         // funcion costo
```

```
        gananciaEstimada = calcularMaximaGananciaPosible(nroSemana, idTarea, gananciaPrevia)
        agrego a (idTarea, gananciaEstimada) a tareasNoRealizadasAExplorar

    cantidadTareasExploradas = 0
    // mientras existan estados descendientes no explorados
    mientras la cantidadTareasExploradas sea menor al tamaño de tareasNoRealizadasAExplorar

        // estadoProximo con mayor fc
        idxDescendiente = buscarIdxTareaNoRealizadaConMayorFc(tareasNoRealizadasAExplorar)
        aumento el contador de cantidadTareasExploradas
        gananciaEstimada = ganancia estimada que saco de tareasNoRealizadasAExplorar con índice idxDescendiente

        // marco al descendiente como visitado
        marco como explorada a la tarea que está en idxDescendiente de tareasNoRealizadasAExplorar

        marco tarea con idTarea como realizada en estadoTareas
        agrego tarea con idTarea en ordenTareasRealizadas

        // si fc es de estadoProximo es mayor a la mejor solucion obtenida
        si la gananciaMaxima es mayor a la gananciaEstimada:
            // si es solucion
            si el ordenTareasRealizadas tiene el mismo tamaño que tareas y la gananciaMaxima es mayor a la
gananciaEstimada:
                gananciaMaxima = gananciaEstimada
                ordenTareasGananciaMaxima = ordenTareasRealizadas

            gananciaTarea = calcularGananciaTarea(nroSemana, idTarea)
            calcularMaximaGanancia(nroSemana + 1, gananciaPrevia + gananciaTarea, ordenTareasRealizadas)

        marco tarea con idTarea como no realizada en estadoTareas
        quito tarea con idTarea en ordenTareasRealizadas

func calcularMejoresGananciasPorSemana:

    // me salteo el id del la tarea
    desde i = 1 hasta tamaño ganancias + 1
        gananciaMaxima = 0
        desde j = 0 hasta el tamaño de ganancias
            si gananciaMaxima < ganancias[j][i]:
                gananciaMaxima = ganancias[j][i]
        agrego a mejoresGananciasPorSemana el valor gananciaMaxima

//============== ESTADOS DESCENDIENTES ==============//

func obtenerTareasNoRealizadas:

    tareasNoRealizadas = []

    por cada tarea de tareas
        idTarea = extraigo el id de tarea
        if not estadoTareas[idTarea - 1]:
            agrego a tareasNoRealizadas el idTarea

    retorno tareasNoRealizadas

//============== PROPIEDAD CORTE ==============//

func todasTareasPreviasRealizadas(idTarea):

    tareasPrevias = extrigo las tareas previas de la tarea con idTarea de tareas

    si no tiene tareasPrevias:
        retorno verdadero

    por cada tarea de las tareasPrevias:
```

```
      si la tarea no fue realizada:
         retorno falso

   retorno verdadero

//============== FUNCION COSTO ==============//

func calcularMaximaGananciaPosible(nroSemana, idTarea, gananciaPrevia):

   gananciaActual = gananciaPrevia
   gananciaActual += calcularGananciaTarea(nroSemana, idTarea)
   gananciaProyectada = calcularProyeccionMaximaGananciaPosible(nroSemana)

   retorno gananciaActual + gananciaProyectada


func calcularGananciaTarea(nroSemana, idTarea):

   retorno ganancias[idTarea - 1][nroSemana + 1]


func calcularProyeccionMaximaGananciaPosible(nroSemana):

   gananciaOptima = 0

   desde i = nroSemana+1 hasta el tamaño de mejoresGananciasPorSemana:
      gananciaOptima += mejoresGananciasPorSemana[i]

   retorno gananciaOptima

//============== ESTADO PROXIMO ==============//

func buscarIdxTareaNoRealizadaConMayorFc(tareasNoRealizadasAExplorar):
   idxMaxFc = -1
   maxFc = 0

   por cada tarea de tareasNoRealizadasAExplorar:
      si la tarea fue explorada:
         salto a primera iteración
      si la tarea no fue visitada y maxFc < fc:
         idxMaxFc = i

   retorno idxMaxFc
```

4) Realice el análisis de complejidad temporal y espacial para B&B.

Por cada i-ésimo nivel del árbol (es decir en cada recursión de la función backtracking), tengo que recorrer todas las tareas para ver cuales no fueron realizadas hasta el momento, lo cual me da una complejidad de O(n) en cada nivel. A cada una de estas tareas no realizadas, les verifico si sus tareas predecesoras ya fueron finalizadas (**función de corte**), lo cual me implica, en el peor de los casos, recorrer una lista con “n” tareas, dando una complejidad temporal de O(n^2). Además, para aquellas que superen la propiedad de corte, se les calcula su potencial ganancia, en donde se recorren en el peor de los casos, “n” semanas. Esto confirma que esta iteración tiene una complejidad de O(n^2)

Luego, por cada tarea no realizada que superó la propiedad de corte, tengo que buscar aquella que me de más ganancia (mayor función de corte), lo cual me implica una complejidad temporal de O(n^2).

Dado que el i-ésimo nodo, se puede ramificar en n-i nuevos nodos, hasta llegar al nivel “n”, que sólo tendrá un nodo posible en el cual ramificarse. Esto me lleva a decir que en el peor de los casos, tendremos que recorrer todas las aristas, llevándome a una complejidad de O(n!)

Para ejemplificar, dejo el siguiente pseudo-codigo simplificado con cada una de las complejidades:

```
Backtrack (estadoActual):
        Sea descendientes los nodos descendientes de estadoActual // O(n)
        Por cada posible estadoDescendiente de estadoActual // O(n^2)
                Si estadoDescendiente supera la propiedad de corte // itero “n” tareas
                        Calcular fc funcion costo de estadoDescendiente // itero “n” semanas
                        Agregar estadoDescendiente a descendientes con fc // insertar elemento al final de lista

        Mientras existan estados descendientes no explorados // O(n^2)
                Sea estadoProximo el estado en descendientes aún no analizado de mayor fc // recorro “n” tareas

                Si el fc de estadoProximo es mayor a la mejor solución obtenida
                        Si estadoProximo es un estado solución y es superior a la mejor
                                mejor = estadoProximo

                        Backtrack(estadoProximo)
```

Por lo tanto, nuestro algoritmo tendrá en el peor de los casos, una complejidad de O(n! n^2).

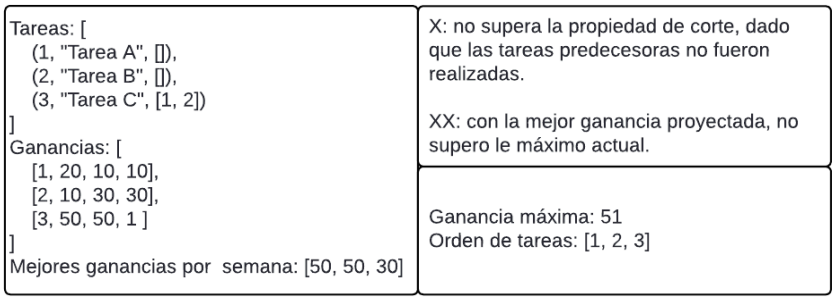
Para la complejidad espacial, tendremos:

- O(n) para la lista de tareas.
- O(n^2) para la lista de ganancias.
- O(n) para la lista de estados de cada tarea.
- tareasNoRealizadas es, espacialmente, una sublista de tareas que se sobreescribe en cada iteración por lo que ocupa O(n).

Podemos decir que, en el peor de los casos tendríamos tres listas de “n” elementos y una de “n^2” elementos, lo cual implica que la complejidad espacial es de O(n^2).

5) Brinde un ejemplo simple paso a paso del funcionamiento de su solución para B&B.

Referencias:

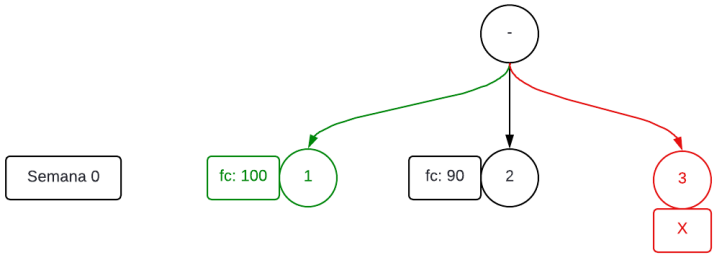


Marco en verde el camino elegido según la función de costo.

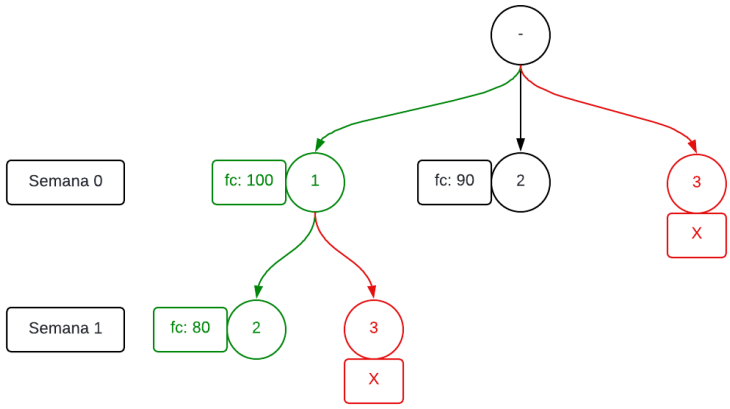
Marco en naranja los caminos ya explorados.

Marco en rojo aquellas podas realizadas.

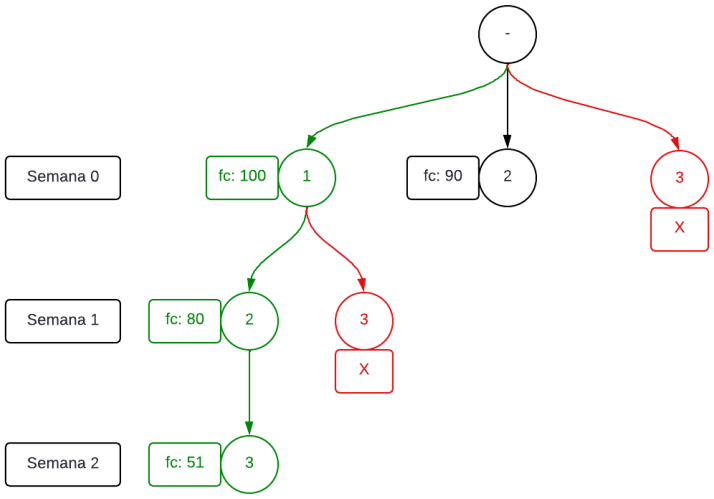
Explicación: *partimos de un estado inicial, en donde estamos en la semana cero, no hay tareas realizadas hasta el momento, la ganancia previa es cero. Le calculo la función de costo a cada una de las tareas no realizadas que superen la condición de corte (la tarea tres necesita que esté la uno y dos terminadas) y exploro la que mayor ganancia me daría (en este caso la tarea uno). Esta función de costo es una estimación de la máxima ganancia que podrías llegar a tener al explorar esa rama.*



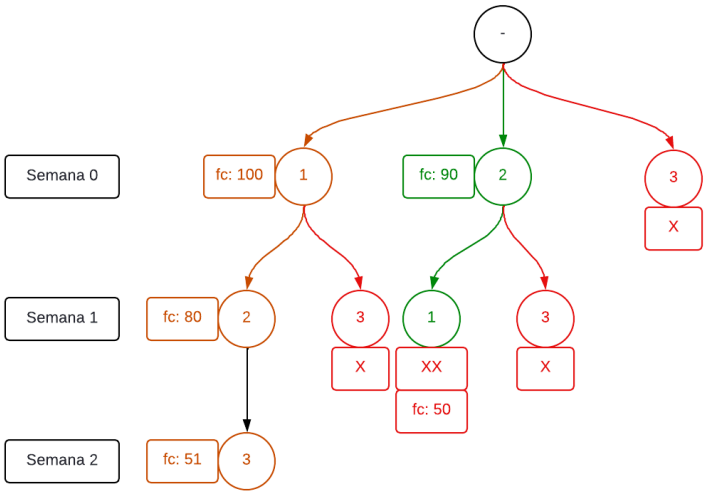
Explicación: *el estado actual es en la semana uno, con la tarea uno realizada y con una ganancia previa de 20. Nuevamente debemos ver cuales son los descendientes y explorar en orden aquellos que den mayor en la función de costo. En este caso solo tengo un descendiente, dado que la tarea tres no supera la función de corte.*



Explicación: *el estado actual es en la semana dos, con la tarea uno y dos realizadas, y con una ganancia previa de 50. El único posible descendiente para este caso es la tarea tres, por lo tanto, verifico qué pasa si agrego la tarea tres, y dado que es un estado solución y supera la máxima actual, guardo el máximo y el orden de las tareas. Cabe destacar que en cada paso anterior, también se verifica si al explorar por esa rama, potencialmente dará una ganancia mayor a la ganancia máxima actual. Dado que la ganancia hasta el momento era cero, esto no tenía efecto.*



Explicación: *ahora debo volver para arriba en el árbol, e iterar los descendientes del nodo padre. Dado que el nodo padre, únicamente tenía un descendiente, vuelvo otro nodo más para arriba, encontrándome así en el estado inicial, en donde estamos en la semana cero, no hay tareas realizadas hasta el momento y la ganancia previa es cero. Sin embargo, ya hay un descendiente explorado, lo cual me lleva a que me quede un único por explorar: la tarea dos. Dado que la tarea dos tiene una función de costo con el valor 90, y el mismo es mayor a la ganancia actual (51), procedo a explorar en sus ramas.*



Explicación: *el estado actual es en la semana uno, con la tarea dos realizada y con una ganancia previa de 10. Dado que como descendiente sólo tenga a la tarea, y como la función de costo de la misma me devuelve 50, no podré superar la máxima ganancia encontrada hasta el momento (51). Por lo tanto, no sigo explorando este descendiente y subo nuevamente.*

Siendo que me quedé sin nodos por explorar, el máximo encontrado es de 51 y el orden de las tareas es [1, 2, 3]

6) Programe su propuesta de B&B.

El programa fué desarrollado en Python. El mismo espera que en el directorio en el cual posicione el archivo “proyecto.py”, se encuentren dos archivos con la información de las tareas y ganancias. El archivo de tareas y ganancias se espera que tengan el formato acordado por el enunciado, en donde las tareas comienzan con el identificador 1, sumado a que no deberían haber dependencias cíclicas entre tareas. Además, también se supone que las ganancias vienen ordenadas de la misma manera que vienen las tareas, comenzando por la especificación de ganancias de la tarea con identificador 1. La manera de ejecutarlo es por línea de comando, con las siguientes especificaciones:

“python3 proyecto.py <nombre_archivo_tareas> <nombre_archivo_ganancias>”

7) Determine si su programa tiene la misma complejidad que su propuesta teórica.

Para comparar la complejidad temporal de la solución programada vamos a omitir la lectura de los archivos, puesto que se omitió en el análisis de la solución teórica también.

inicializarMejoresGananciasPorSemana tiene una complejidad de **O(n^2)** ya que para cada tarea recorremos todas los valores de ganancia por semana.

inicializarEstadoTareas tiene una complejidad de **O(n)** ya que recorremos una lista de tantos elementos como tareas haya.

Analicemos ahora calcularMaximaGanancia:

- obtenerTareasNoRealizadas, en el peor de los casos, recorre toda las listas, así que es O(n).
- Por cada tarea no realizada, llamamos a todasTareasPreviasRealizadas:
    - Con O(1) consigue la tarea.
    - Luego, recorrer las tareas previas, en el peor de los casos, será O(n).
    - calcularMaximaGananciaPosible calcula la ganancia de la tarea para la semana correspondiente y llama a calcularProyeccionMaximaGananciaPosible, que recorre las semanas que faltan por completar. En el peor de los casos será O(n). Por lo que calcularMaximaGananciaPosible queda con O(n).
- Por cada tareasNoRealizadasAExplorar:
    - Llama a buscarIdxTareaNoRealizadaConMayorFc, con O(n).
    - Marca el visitado con O(1).
    - Marca la tarea como realizada con O(1).
    - Si la ganancia estimada actual supera a la ganancia máxima hasta el momento, se copia ordenTareasRealizadas con O(n).
    - calcularGananciaTarea obtiene la ganancia con O(1).
    - se llama a calcularMaximaGanancia aumentando 1 la semana y pasando las tareas realizadas.
    - se marca la tarea como no realizada con O(1).

Esto provocará que calcularMaximaGanancia sea llamado n! veces con las tareas en estado realizada y no realizada alternativamente.

En conclusión proyecto está compuesto de O(n^2) + O(n) + O(n^2) + O(2^n * n^2)=**O(n! * n^2)**.

El análisis espacial será:

- tareas: O(n).
- ganancias: O(n).
- estadoTareas: O(n).
- mejoresGananciasPorSemana: O(n).
- ordenTareasGananciaMaxima: O(n).

Parte 2: El costo de mantenimiento

Enunciado

En una red global existen un conjunto de servidores conectados mediante conexiones punto a punto. No todos los servidores están conectados entre sí, pero la comunicación se puede realizar pasando por servidores intermedios. Cada conexión tiene un patrocinador que paga un costo anual por ella. Operativamente la red tiene un costo elevado de mantenimiento relacionado con el número de conexiones. Los administradores desean conocer al mayor número de conexiones que se pueden eliminar, manteniendo la conectividad entre todos los servidores, minimizando la pérdida por patrocinio.

Resolución

1) Determinar y explicar cómo se resolvería este problema utilizando la metodología greedy.

Partimos de un grafo conexo de conexiones G=(V,E), donde cada vértice es un servidor y cada eje una conexión patrocinada con un costo anual We.

Los servidores pueden comunicarse pasando por otros intermedios:

- Tomando un vértice A y un vértice B pertenecientes a V, podemos afirmar que seguirán conectados dado que existe un camino de A a B

- Lo que significa que si nuestro grafo G contiene algún ciclo, podemos eliminar uno de los ejes pertenecientes al ciclo manteniendo la conectividad entre los servidores.

Debemos también minimizar la pérdida por patrocinio:

- Lo que significa que buscamos un grafo sin ciclos contenido en G, tal que la sumatoria de todos los We sea el máximo posible.

Por las propiedades que tendría nuestra solución, el problema resulta en encontrar un árbol cuya sumatoria de pesos de ejes maximiza el patrocinio. Esto es un árbol recubridor máximo. Bastaría con ver qué ejes de G no se encuentran en el árbol resultante y esas serían las conexiones que se pueden eliminar. Usando el algoritmo de eliminación inversa, podemos acceder de forma más sencilla a esa información ya que el algoritmo se centra en eliminar los ejes (menos pesados en este

caso) manteniendo la conectividad del grafo resultando un árbol recubridor máximo. Podría usarse el algoritmo de Prim o Kruskal también, estos se centran en formar el árbol recubridor agregando ejes por lo que al final del algoritmo hay que comparar con el grafo inicial qué ejes no se incluyeron en el árbol recubridor y esos serian los eliminados. Usaremos el de eliminación inversa.

2) Brinde pseudocódigo y estructuras de datos a utilizar.

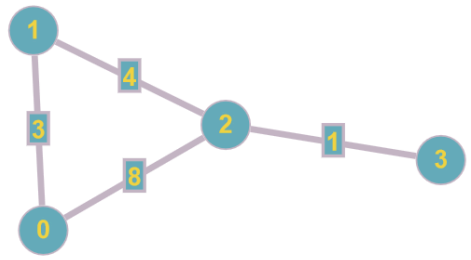
- Ejes y Conexiones eliminadas es un arreglo de ejes con cada eje = (u,v,w)

- AdyacenciaVertices es un arreglo de los vértices del grafo el cual cada uno contiene un arreglo de vértices adyacentes

- Usamos una pila para el recorrido DFS de forma iterativa

```
EliminacionInversa(Ejes, AdyacenciaVertices):
        ConexionesEliminadas = {}
        Ejes = Ordenar ejes de forma ascendente
        Mientras longitud de Ejes > CantidadVertices - 1
            Sea eje = ejes[0]
            Sean u,v los vértices que une eje
            Eliminar v de AdyacenciaVertices[u]
            Eliminar u de AdyacenciaVertices[v]

            Si EsConexo(AdyacenciaVertices)
                Agregar eje a ConexionesEliminadas
            Si no
                Agregar v a AdyacenciaVertices[u]
                Agregar u a AdyacenciaVertices[v]

         Retornar ConexionesEliminadas

EsConexo(AdyacenciaVertices):
            Visitados = {}
            Pila = {}
            Desde i=0 hasta longitud de AdyacenciaVertices
                 Visitados[i] = falso
             Agregar a Pila

             Mientras haya elementos en Pila
                   v = desapilar elemento de Pila

                 Si !visitados[v]
                       visitados[v] = verdadero

                        Por cada vérticeAdyacente en AdyacenciaVertices[v]
                             Si !visitados[vérticeAdyacente]
                                  Apilar vérticeAdyacente en Pila

            Desde i=0 hasta longitud de AdyacenciaVertices
                        Si !Visitados[i]
                                 Retornar falso
            Retornar Verdadero
```

3) De un ejemplo paso a paso. ¿Qué complejidad temporal y espacial tiene la solución?



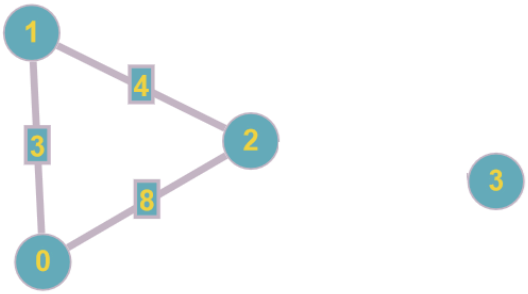
El algoritmo empieza con un arreglo vacío de conexiones eliminadas y se ordenan los ejes del grafo en forma ascendente. Comenzamos a eliminar ejes manteniendo la conectividad hasta que la cantidad de ejes sea cantidad de vértices menos uno (propiedad de arbol). Este paso se realiza en el bucle while.

Los ejes ordenados ascendentemente serian: (3-2, 1), (0-1,3), (2-1,4) y (2-0,8)

La lista de adycencia es: [0: [1,2], 1:[2,0], 2: [0.1.3], 3:[2]]

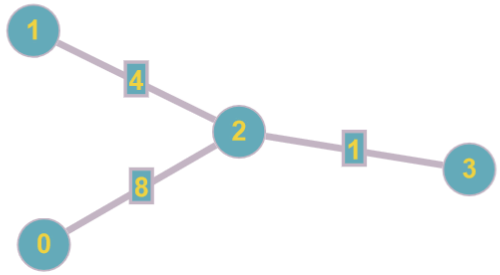
Se toma el primer eje, (3-2, 1) y se elimina 3 de la lista de adyacencias de 2 y viceversa resultando [0: [1,2], 1:[2,0], 2: [0.1], 3:[ ]]. Luego chequeamos si el grafo resultante sigue siendo conexo, haciendo un recorrido DFS en el cual comenzamos agregando 0 a la pila. Se marca 0 como visitado y se agregan los vértices adyacentes 1 y 2 a la pila. El proceso se repite hasta que la pila quede vacía.

Vemos que el vértice 3 nunca se agrega a la pila por ende nunca se lo marca como visitado. El algoritmo recorre la lista de visitados y se encuentra que la posición correspondiente al vértice 3, está marcada como false. Esto indica que el grafo no resultó conexo. Se puede ver gráficamente:



Repitiendo el proceso hasta llegar a ejes = cant vértices - 1, vemos que solo es posible eliminar una conexión. Por orden ascendente el eje que sigue es (0-1,3) que puede ser eliminado manteniendo la

conectividad. El árbol recubridor máximo resultante es el siguiente:



Complejidad temporal: O(V*(V+E) + Elog(E))

E = cantidad de ejes

V = cantidad de vértices

- Ordenar los ejes de forma ascendente tiene un costo de O(Elog(E)) usando mergesort.
- Agregar un vértice a la lista de adyacencia o agregar un eje a la lista de conexiones eliminadas tiene una complejidad de O(1).
- Eliminar un vértice en la lista de adyacencia, en el peor de los casos el vértice tiene como adyacentes el resto. Esto tiene un costo de O(V) pero utilizando una estructura más eficiente como sets puede ser llevado a O(1).
- Verificar si un subgrafo es conexo usando DFS, tiene un costo de O(V + E) ya que hay que visitar todos los vértices, y todos los ejes para explorar los vértices adyacentes. Esta operación se realiza hasta que se forme el árbol recubridor, en el peor de los casos V - 1 veces. Resulta en O(V*(V+E))
- La suma de complejidades resulta en O(V*(V+E) + Elog(E))

Complejidad espacial: O(E + V)

- La lista de adyacencia tiene un costo de O(E + V).
- La lista de ejes tiene un costo de O(E).
- La pila almacenará como mucho todos los vértices a recorrer. O(V).
- La lista de vértices visitados tiene un costo de O(V).
- Sumando las complejidades resulta en O(E + V)

4) Justifique por qué corresponde su propuesta a la metodología greedy.

Este algoritmo corresponde a la metodología greedy, ya que cumple que el algoritmo resuelve el problema final dividiéndolo en subproblemas los cuales busca solucionarlos de la forma más óptima posible. Cada subproblema es un subgrafo del original el cual decidimos eliminar el eje de menor

peso (que no desconecta el grafo). Esto último es la elección greedy. Finalmente al resolver el último subproblema el algoritmo forma un árbol recubridor máximo y devolvemos las conexiones que se eliminaron.

5) Demuestre que su solución es óptima.

Nuestra solución es óptima dado que nuestro algoritmo:

- Va a asegurarse que el grafo siempre permanezca conexo en cada subproblema entonces todos los servidores van a estar conectados.

- Dado un ciclo C, va a eliminar el eje menos pesado en C resultando que esa conexión no pertenezca al árbol recubridor máximo y maximizando el patrocinio. Esto lo hace de acuerdo a la propiedad de ciclo.

- En caso de haber más de una conexión a eliminar con la misma ganancia por patrocinio, se resuelve eliminando la primera encontrada. El árbol recubridor resultante maximiza el patrocinio pero no es el único, lo cual implica que en estos casos hay más de un conjunto posible de conexiones a eliminar.

Parte 3: Los planos de las piezas

Enunciado

Para la elaboración de ciertas piezas se suelen realizar planos con vistas frontales y laterales de ellas. Las piezas están conformadas por diferentes partes. Cada parte tiene forma rectangular representado por rectángulos mediante la tripla (izquierda, altura, derecha). Izquierda y derecha corresponden al eje X y tienen valores positivos. La altura es con respecto al eje Y. También únicamente puede ser un valor cero o positivo. Para generar el plano se cuenta con el listado de las partes sin un orden específico y sus coordenadas en el plano. Por ejemplo:

Pieza 1: (1, 11, 5), (2, 6, 7), (3, 13, 9), (12, 7, 16), (14, 3, 25), (19,18,22)

Una parte del proceso requiere construir el contorno de la pieza representado como una lista de coordenadas “x” y sus alturas. Para el ejemplo anterior:

Contorno: (1,11), (3,13), (9,0), (12,7), (16,3), (19,18), (22,3), (25,0)

Resolución

1) Presentar un algoritmo que lo resuelva utilizando división y conquista.

El algoritmo que proponemos empieza con una lectura de cada parte, pasada por archivo. Luego, a esta lista de partes leída, se la pasaremos a la función encargada de ejecutar un algoritmo que utiliza división y conquista. Esta función tiene como objetivo, el generar “contornos” a partir de “partes”. Para esto, dividirá (**división**) esta lista de partes en dos y a estas nuevas listas, se les volverá a aplicar la misma función de manera recursiva.

En el **caso base**, es decir, cuando tengamos un solo elemento en la lista de partes, armamos un contorno con dos coordenadas a partir de las partes, y lo retornamos. Estos contornos devueltos por esta recursión, serán unidos, formando nuevos contornos. Esto implica que las particiones que fuimos generando, se convertirán en contornos y serán unidos hasta llegar al contorno completo.

Para unir contornos (**combinación**), presentamos una solución que se basa en ir armando intervalos de izquierda a derecha a partir de todas las coordenadas existentes en los contornos a unir. Para esto iremos sacando entre ambos contornos, los mínimos puntos, e iremos armando intervalos. Estos intervalos les corresponde una altura en cada uno de los contornos, y el máximo entre ambas es la que nos interesa, dado que con la misma armaremos una coordenada nueva. En cuanto tenemos todos los intervalos, con su altura máxima, nos quedaremos con el extremo izquierdo del mismo (x1) y la altura (yMax) para armar una coordenada (x1, yMax). Entonces, esto nos generará una lista de coordenadas nuevas, que representan nuestro nuevo contorno. Cabe destacar que se debe hacer un chequeo a la hora de insertar una nueva coordenada a esta lista, que consta en ver si la coordenada anteriormente insertada, tiene la misma altura que la coordenada nueva que voy a insertar, lo cual en caso de que se cumpla, será descartada la última. Por último, con el último intervalo, escribo el último elemento del contorno, a partir del extremo derecho del mismo (x2): coordenada (x2, 0).

Para el caso borde, en donde dos contornos coinciden en un punto, lo cual nos generaría un intervalo de tamaño cero, al mismo se lo descarta y se continúa con el próximo.

2) Mostrar la relación de recurrencia.

Partiendo de la primicia que tengo "n" partes y cada una de ella genera dos coordenadas, tendremos "$2n$" coordenadas en el peor de los casos.

Podemos ver que de cada problema, se desprenden dos nuevos con la mitad de elementos. La combinación de estos elementos, según lo propuesto en el pseudo-código, que se denotará a continuación, tiene una complejidad de $O(n)$, dado que en el peor de los casos recorremos todas coordenadas ($2n$).

Por lo tanto la relación de recurrencia nos quedará de la siguiente manera:

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

3) Presentar pseudocódigo.

```
main:

   listaPartes las leo del archivo de partes.txt

   contorno(listaPartes)

//============== FUNCIÓN PRINCIPAL ==============//

func generarContornoAPartirDePartes(listaPartes):

   si el tamaño de listaPartes = 1
      retorno construirUnContorno(listaPartes[0])

   listaPartes1 = primera mitad de listaPartes
```

```
    listaPartes2 = segunda mitad de listaPartes

    contorno1 = generarContornoAPartirDePartes(listaPartes1)
    contorno2 = generarContornoAPartirDePartes(listaPartes2)

    retorno mergeContornos(contorno1, contorno2)

//============== FUNCIONES AUX ==============//

func construirUnContorno(parte):

    contorno = []

    // parte compuesta por (x1, y, x2)
    extraigo (x1, y, x2) de parte

    contorno[0] = (x1, y)
    contorno[1] = (x2, 0)

    retorno contorno

func mergeContornos(contorno1, contorno2):

    contorno = []

    idxContorno2 = 0
    idxContorno1 = 0

    contornoId = idContornoConMenorCoordenadaX(contorno1, idxContorno1, contorno2, idxContorno2)
    si contornoId = 1
        coordenadaIzq = contorno1[idxContorno1]
        idxContorno1 += 1
    sino
        coordenadaIzq = contorno2[idxContorno2]
        idxContorno2 += 1

    coordenadaActualIzquierda = coordenadaIzq.x

    mientras (idxContorno1 < tamaño de contorno1 ó idxContorno2 < tamaño de contorno2):

        contornoId = idContornoConMenorCoordenadaX(contorno1, idxContorno1, contorno2, idxContorno2)
        si contornoId = 1
            coordenadaIzq = contorno1[idxContorno1]
            idxContorno1 += 1
        sino
            coordenadaIzq = contorno2[idxContorno2]
            idxContorno2 += 1

        x1 = coordenadaActualIzquierda
        x2 = coordenadaIzq.x
        coordenadaActualIzquierda = x2

        intervalo = (x1, x2)

        alturaMaxima = alturaMaxima(contorno1, idxContorno1, contorno2, idxContorno2, intervalo)

        si tamaño de contorno = 0:
            agrego a contorno el valor (x1, alturaMaxima)
        sino si el último elemento del tiene una altura distinta a alturaMaxima:
            agrego a contorno el valor (x1, alturaMaxima)


    agrego a contorno el valor (coordenadaActualIzquierda, 0)

    retorno contorno
```

```
func idContornoConMenorCoordenadaX(contorno1, idxContorno1, contorno2, idxContorno2):

    si tamaño de contorno1 <= idxContorno1 + 1:
        // me quede sin elementos por recorrer
        retorno 2
    si tamaño de contorno2 <= idxContorno2 + 1:
        // me quede sin elementos por recorrer
        retorno 1

    si contorno1[idxContorno1].x <= contorno2[idxContorno2].x:
        retorno 1
    sino:
        retorno 2

func alturaMaxima(contorno1, idxContorno1, contorno2, idxContorno2, intervalo):

    alturaContorno1 = alturaDeContorno(contorno1, idxContorno1, intervalo)
    alturaContorno2 = alturaDeContorno(contorno2, idxContorno2, intervalo)

    retorno max(alturaContorno1, alturaContorno2)


func alturaDeContorno(contorno, idxContorno, intervalo):

    // considero al intervalo cerrado a la izquierda y abierto a la derecha
    // [x1, x2)

    // para saber la altura del contorno, tengo que saber que un
    // intervalo puede haber sido generado por:
    // - una coordenada de cada contorno
    // - dos coordenadas del mismo contorno

    // caso 1: una coordenada de cada contorno
    idxAuxiliar = idxContorno - 1
    si idxAuxiliar < 0:
        retorno 0
    si contorno[idxAuxiliar].x <= intervalo.x1:
        return contorno[idxAuxiliar].y

    // caso 2: dos coordenadas del mismo contorno
    idxAuxiliar = idxContorno - 2
    si idxAuxiliar < 0:
        return 0
    return contorno[idxAuxiliar].y
```

4) Analice la complejidad del algoritmo utilizando el teorema maestro y desenrollando la recurrencia.

Complejidad utilizando Teorema Maestro:

Para aplicar teorema maestro según la la relación de recurrencia presentada anteriormente, ($T(n) = 2T(\frac{n}{2}) + O(n)$) dado que a > 1 y b > 1 constantes (a = 2, b = 2), f(n) una función y $T(n) = 2T(\frac{n}{2}) + O(n)$ una recurrencia con T(0)=cte, entonces se puede aplicar Teorema Maestro.

Dado a = 2, b = 2 y f(n) = O(n):

Caso 2: f(n) = Θ(n^($\log_b$ a)) ⇒ n = Θ(n^($\log_2$ 2)) Es correcta la afirmación

T(n) = Θ(n^($\log_2$ 2) log n) ⇒ *T(n) = Θ(n log n)*

Complejidades de las operaciones de una lista en Python:

- len(): O(1). Es una operación constante porque Python mantiene un contador de la longitud de la lista.
- append(): En promedio es O(1), existe el caso en que se necesite redimensionar la lista cuando se alcanza la capacidad máxima y puede tomar un tiempo O(n)

Complejidad desenrollando la recurrencia:

Siendo Cn el trabajo lineal a realizar en cada subproblema (proceso de dividir y mergear los elementos) entonces el trabajo total del algoritmo será la sumatoria del trabajo de todos los subproblemas. En cada llamada recursiva, nuestro subproblema se divide en otros dos subproblemas hasta llegar al caso base entonces podemos representar nuestro problema como un árbol el cual cada nodo padre tiene 2 nodos hijos con la mitad de elementos.
En un nivel *i* del árbol tendremos 2^i subproblemas cada uno con un tamaño de $\frac{n}{2^i}$ elementos por lo tanto el trabajo total en un nivel será de Cn. Despejando *i* obtenemos:
$i = log_2 n$

Desenrollando la relación de recurrencia con el trabajo lineal Cn:
$T(n) = 2T(\frac{n}{2}) + Cn$

$T(n/2) = 2T(\frac{n}{2}) + \frac{Cn}{2}$

$T(n/4) = 2T(\frac{n}{4}) + \frac{Cn}{4}$

$T(n) = 2^i d + \sum_{j=0}^{i} \frac{2^j Cn}{2^j}$

Reemplazando $i = log_2 n$

$T(n) = 2^{log_2 n} d + \sum_{j=0}^{log_2 n} \frac{2^j Cn}{2^j}$

$T(n) = 2^{log_2 n} d + Cn \sum_{j=0}^{log_2 n} 1^j$

Reemplazando $2^{log_2 n} = n$

$T(n) = n\,d + Cn\, log_2 n$

Despreciando las constantes

$T(n) = O(n \log n)$

5) Brindar un ejemplo de funcionamiento.

// INPUT

Partes: [(1, 10, 3), (3, 15, 6), (2, 20, 4), (8, 30, 9)]
Contorno esperado: [(1, 10), (2, 20), (4, 15), (6, 0), (8, 30), (9, 0)]
Output de "contorno.py": "Contorno: [(1, 10), (2, 20), (4, 15), (6, 0), (8, 30), (9, 0)]"

// DIVISIÓN

// "lista" no es caso base -> split en dos
lista = [(1, 10, 3), (3, 15, 6), (2, 20, 4), (8, 30, 9)]

// "lista_1" no es caso base -> split en dos
lista_1 = [(1, 10, 3), (3, 15, 6)]
// "lista_2" no es caso base -> split en dos
lista_2 = [(2, 20, 4), (8, 30, 9)]

// CASO BASE

// "lista_1_1" es caso base -> genero contorno
lista_1_1 = [(1, 10, 3)] -> contorno_1_1 = [(1, 10), (3, 0)]
// "lista_1_2" es caso base -> genero contorno
lista_1_2 = [(3, 15, 6)] -> contorno_1_2 = [(3, 15), (6, 0)]

// "lista_2_1" es caso base -> genero contorno
lista_2_1 = [(2, 20, 4)] -> contorno_2_1 = [(2, 20), (4, 0)]
// "lista_2_2" es caso base -> genero contorno
lista_2_2 = [(8, 30, 9)] -> contorno_2_2 = [(8, 30), (9, 0)]

// COMBINACIÓN 1

// combino contorno_1_1 con contorno_1_2
contorno_1_1 = [(1, 10), (3, 0)]
contorno_1_2 = [(3, 15), (6, 0)]
contorno_1 = []

// armo un intervalo con coordenadas más a la izquierda
intervalo = (1, 3)
// veo las alturas en cada contorno y me quedo con la máxima
altura_en_contorno_1 = 10 -> altura máxima
altura_en_contorno_2 = 0
// agrego nueva coordenada
contorno_1 = [(1, 10)]

intervalo = (3, 3) -> lo descarto por coincidir ambos puntos y sigo con el próximo

intervalo = (3, 6)
altura_en_contorno_1 = 0
altura_en_contorno_2 = 15 -> altura máxima
contorno_1 = [(1, 10), (3, 15)]

// agrego última coordenada a partir de último intervalo
ultimo_intervalo = (3, 6)
contorno_1 = [(1, 10), (3, 15), (6, 0)] -> contorno resultante

// COMBINACION 2

// combino contorno_2_1 con contorno_2_2
contorno_2_1 = [(2, 20), (4, 0)]
contorno_2_2 = [(8, 30), (9, 0)]
contorno_2 = []

intervalo = (2, 4)
altura_en_contorno_1 = 20 -> altura máxima
altura_en_contorno_2 = 0
contorno_2 = [(2, 20)]

intervalo = (4, 8)
altura_en_contorno_1 = 0 -> altura máxima
altura_en_contorno_2 = 0
contorno_2 = [(2, 20), (4, 0)]

intervalo = (8, 9)
altura_en_contorno_1 = 0
altura_en_contorno_2 = 30 -> altura máxima
contorno_2 = [(2, 20), (4, 0), (8, 30)]

ultimo_intervalo = (8, 9)
contorno_2 = [(2, 20), (4, 0), (8, 30), (9, 0)] -> contorno resultante

// COMBINACION 3

// combino contorno_1 con contorno_2
contorno_1 = [(1, 10), (3, 15), (6, 0)]
contorno_2 = [(2, 20), (4, 0), (8, 30), (9, 0)]
contorno = []

intervalo = (1, 2)
altura_en_contorno_1 = 10 -> altura máxima
altura_en_contorno_2 = 0
contorno = [(1, 10)]

intervalo = (2, 3)
altura_en_contorno_1 = 10
altura_en_contorno_2 = 20 -> altura máxima
contorno = [(1, 10), (2, 20)]

intervalo = (3, 4)
altura_en_contorno_1 = 15
altura_en_contorno_2 = 20 -> altura máxima
// no se agrega coordenada porque tiene misma altura que la anterior
contorno = [(1, 10), (2, 20)]

intervalo = (4, 6)
altura_en_contorno_1 = 15 -> altura máxima
altura_en_contorno_2 = 0
contorno = [(1, 10), (2, 20), (4, 15)]

intervalo = (6, 8)
altura_en_contorno_1 = 0 -> altura máxima
altura_en_contorno_2 = 0
contorno = [(1, 10), (2, 20), (4, 15), (6, 0)]

intervalo = (8, 9)
altura_en_contorno_1 = 0
altura_en_contorno_2 = 30 -> altura máxima
contorno = [(1, 10), (2, 20), (4, 15), (6, 0), (8, 30)]

ultimo_intervalo = (8, 9)
contorno = [(1, 10), (2, 20), (4, 15), (6, 0), (8, 30), (9, 0)]

6) Programe su solución.

El programa fué desarrollado en Python. El mismo espera que en el directorio en el cual posicione el archivo “contorno.py”, se encuentre el archivo con la información de las partes a partir de las cuales se quiere armar un contorno. Se espera que el archivo tenga el formato acordado por el enunciado. La manera de ejecutarlo es por línea de comando, con las siguientes especificaciones:

“python3 contorno.py <nombre_archivo_partes>”

7) Analice si la complejidad de su programa es equivalente a la expuesta en el punto 4.

Parte recursiva principal: `generarContornosAPartirDePartes`:

- Divide el conjunto de partes en dos subproblemas más pequeños (mitades) hasta que se queda con un solo elemento. Cuando se queda con un solo elemento se realiza el cálculo del contorno.
- Luego, con la función `mergeContornos` se combinan los resultados obtenidos de cada mitad.

1 - Este patrón corresponde a una estructura de división y conquista:

- El problema se divide en dos subproblemas de tamaño n / 2
- El costo de combinar las soluciones (función `mergeContornos`) es lineal: $O(n)$

2 - Forma general de la recurrencia: $T(n) = 2T(n/2) + O(n)$

3 - Análisis usando el teorema maestro: La forma general de la recurrencia es

$$T(n) = AT(n/B) + O(n^D)$$

donde:

- $A = 2$ (el problema se divide en 2 subproblemas)
- $B = 2$ (cada subproblema es de tamaño $n/2$)
- $D = 1$ (el costo de combinar las soluciones es lineal, $O(n)$)

Ahora, aplicando el teorema maestro

- $n^{log_B A} = n^{log_2 2} = n$
- Como $n^{log_B A} = O(n^D)$ estamos en el caso 2 del Teorema Maestro.

Según este caso, la complejidad del algoritmo es $O(n \log n)$ por lo tanto la complejidad es equivalente a la expuesta en el punto 4.

Referencias

- Apunte TDA - Backtracking y branch and bound - ING. VÍCTOR DANIEL PODBEREZSKI
- Apunte TDA - Metodología greedy - ING. VÍCTOR DANIEL PODBEREZSKI
- Apunte TDA - División y conquista - ING. VÍCTOR DANIEL PODBEREZSKI
- Documentación sobre la complejidad de las operaciones de una lista en Python
- J. Kleinberg, E. Tardos, *Algorithm Design*, Addison Wesley (2006). [KT]