DEPARTAMENTO
INGENIERÍA INFORMÁTICA
Y CIENCIAS DE LA COMPUTACIÓN
FACULTAD DE INGENIERÍA UNIVERSIDAD DE CONCEPCIÓN

Análisis de Algoritmos: Algoritmos Greedy

Alumnos: Fabián Cid Escobar
Rodolfo Fariña Reisenegger

Profesora: Cecilia Hernández Rivas

12 de julio del 2020, Concepción

Introducción

Durante la segunda parte de este semestre se ha aprendido sobre cómo funcionan los algoritmos greedy y la ventaja que estos tienen. Estos toman decisiones greedy, que a la vez son óptimas y que garantizan la respuesta correcta.

En este proyecto hemos tenido que implementar un algoritmo greedy para resolver el problema del grafo de intervalos, el cual es bastante similar al de selección de actividades. Este problema no es resuelto por el mismo algoritmo, y fue necesario implementar uno levemente distinto.

El problema consiste en un conjunto de intervalos, donde los vértices corresponden a los intervalos y se crean aristas cuando hay conflictos entre los intervalos. Luego se colorean los vértices de modo que 2 vértices adyacentes no puedan tener el mismo color.

a) Muestre que usando el algoritmo greedy para resolver el problema de selección de actividades repetidamente no proporciona una solución óptima al problema de asignar todas las actividades al mínimo número de personas.

Para mostrar que el algoritmo de Selección de Actividades Repetidamente no da una solución óptima, lo aplicaremos en el siguiente caso:

actividad	inicio	final
a	1	4
b	2	6
c	6	7
d	5	8

Tabla 1: Caso de contraejemplo

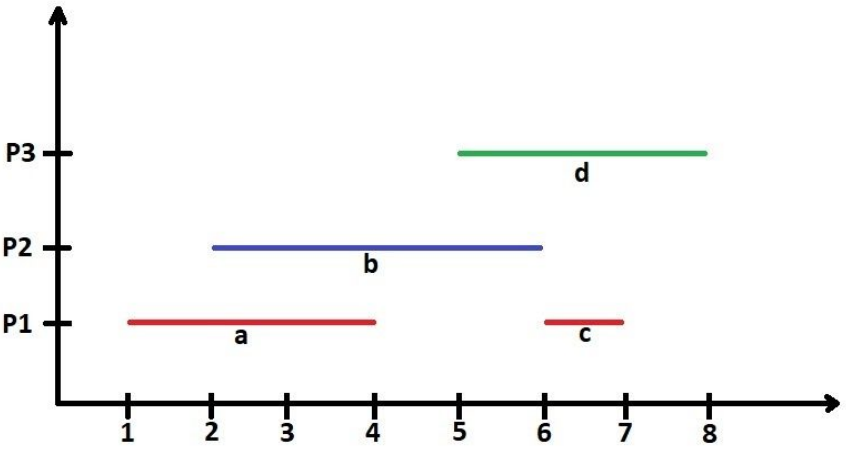
En primer lugar, colocamos la actividad que comienza primero a la cabeza de la tabla y el resto se ordena de menor a mayor según el momento de término de la actividad. Vemos que en este caso la tabla ya está ordenada.

Asignamos la primera actividad a una persona P1, luego buscamos otra actividad que termine y comience después del momento de término de la primera actividad recorriendo la tabla hacia abajo, en este caso, escogemos la actividad c. Se repite el mismo proceso buscando hacia abajo de la tabla, pero como no hay más actividades que cumplan las condiciones. Pasamos a asignar a otra persona.

Después le damos a una persona P2 la actividad que está más arriba en la tabla y que no haya sido asignada, le asignamos b. Después recorremos linealmente la tabla hacia abajo buscando otra actividad que termine y comience luego del momento de termino de b. Se recorre la tabla completa y no hay ninguna que cumpla la condición.

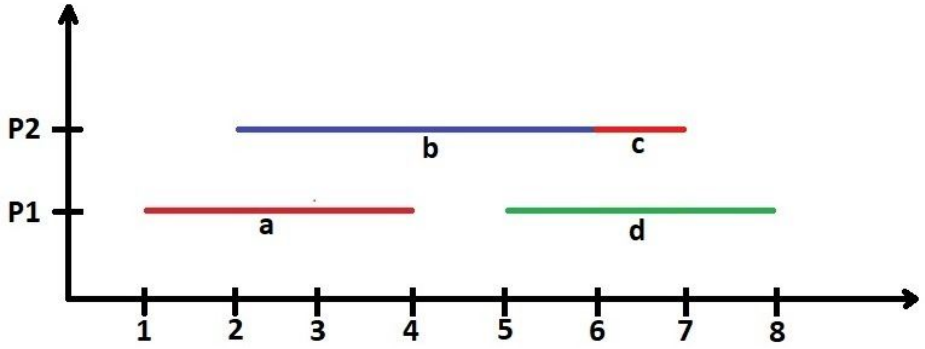
Luego, el algoritmo asigna las 4 tareas a 3 personas.

El esquema de lo anterior es:



Esquema 1: "Situación de contraejemplo con Selección de Actividades Repetidamente"

Pero a simple vista notamos que se puede asignar la actividad c P2 y la actividad d a P1, quedando el esquema:



Esquema 2: "Una alternativa óptima para la situación de contraejemplo"

Donde claramente podemos observar que podemos asignar las mismas 4 tareas a sólo 2 personas. Por ello, el algoritmo de Selección de Actividades Repetidamente no proporciona una solución óptima para todos los casos.

b) Pseudocódigo y explicación breve.

Nuestro algoritmo funciona de manera que generamos un arreglo de eventos para registrar cada inicio o fin de actividad. En este arreglo, guardaremos todos los inicios y fin de actividad, posteriormente, vamos a ordenar este arreglo de menor a mayor, dando prioridad a los eventos de término (si se tiene 4_s y 4_f, entonces 4_f es considerado como menor).

Luego, creamos un stack para tener a todas las personas disponibles en cada momento dado.

Cuando se detecta un evento de start se popea una persona en el stack, y cuando esta termina con dicha actividad, se pushea al stack de nuevo. Esta persona se marcó como “asigned = true”. Al terminar, podemos contar la cantidad de personas que han sido asignadas y esto nos dará la respuesta.

```
contarPersonas (array start, array finish, n, *aux)
    Crear un arreglo llamado eventos tamaño 2n //2n
    //Eventos es un arreglo de tamaño 2n que contiene todos los start y
finish que existen en start y finish
    Ordenar eventos de menor a mayor //nlogn con std::sort
    //se da prioridad a eventos finish por sobre los start
    Crear un stack y llenarlo con n personas //n
    Recorrer stack y setear asigned = false en cada persona //n

    persona* persona
    for i=0 to 2n-1 //2n
        if (event[i]==finish)
            persona = events[i].activity.persona //c
            //determinar persona correspondiente a ese evento
            stack.push(persona) //c
        endif
        if (event[i]==start)
            persona = stack.pop() //c
            events[i].activity.persona = persona //c
            //se le asigna persona a esta actividad
            persona.used = true //c
        endif
    endfor
    //revisar todas las salas para ver cual fue usado
```

```
    for i=0 to n-1 //n
        if (stack.pop().asigned == true)
            count++        //c
            add persona to personasVec //c
        endif
    endfor
    return personasVec
end
```

Sabemos que podemos implementar un stack con push y pop de tiempo O(1), entonces nos fijamos en las demás operaciones. Notar que la parte que toma más tiempo es ordenar el vector de eventos, puesto que esto toma tiempo O(nlogn) y todos los otros ciclos demoran O(n).

Si sumamos todos los costos asintóticos tenemos:

$$2n + nlog\,n + n + n + 2n + n = nlog\,n + 7n = \theta(nlog\,n)$$

c) Para demostrar la correctitud utilizamos un argumento de cota.

Para demostrar que es óptimo siempre podemos probarlo con un argumento de cota mínima. En cualquier momento dado, si el algoritmo decide usar una persona adicional k, eso es porque las personas 1,2,...,k-1 están todas ocupadas. En este momento las personas 1,2,...,k-1 están realizando actividades y hay k-1 actividades activas al mismo tiempo y estamos agregando una actividad adicional k.

En este punto hay K actividades que hacen conflictos entre sí, y no usaremos otra persona adicional a menos que hayan K conflictos en ese momento, y si hay K conflictos en un momento dado, no hay otra manera de hacerlo con menos de K personas. Si hay K personas que hacen conflicto, es imposible organizar las actividades con menos de K personas. Entonces, el número de personas que usamos, siempre está dado por esta cota del número máximo de conflictos de actividades, lo cual es nuestra cota.

Entonces nuestra cota limita la cantidad de personas que realizan actividades, y si esta cota está delimitada por la cantidad máxima de conflictos que podemos tener en cualquier momento dado, esto prueba la correctitud.

Para ilustrar, sea tiempo y número de conflictos

1 c1
2 c2
3 c3
. .
. c_max
. .
2n c_2n

c_max determina la cantidad máxima de conflictos y por ende la cantidad de salas usadas.

El algoritmo usará la persona K en un momento dado del tiempo si y sólo si las personas 1, 2, ..., k-1 están ocupadas. Pero esto implica que en este punto tenemos *K* conflictos. Lo que significa que necesitamos al menos K personas. Cuando usamos una persona es porque NECESITAMOS usarla, no hay otra forma sin usarla, y vamos a necesitar esta sala extra para asignarle la actividad K

d) Ejecución paso a paso del algoritmo propuesto

actividad	inicio	final
a	1	4
b	2	6
c	6	7
d	5	8

Para la misma situación usada en el contraejemplo de a).
Sea ***n*** la cantidad de actividades, se crea un arreglo de eventos de tamaño ***2n*** y se insertan en él los eventos de inicio y final para cada actividad.

Eventos:

ia	fa	ib	fb	ic	fc	id	fd

Donde ix es el evento que da inicio a la actividad x, mientras que fx es el evento que da fin a la actividad x.
El siguiente paso es ordenar de menor a mayor los eventos según el momento en el que ocurren, donde los que finalizan una actividad tienen prioridad para ir antes de un evento de inicio en caso de que ocurran al mismo tiempo.

Eventos:

ia	ib	fa	id	fb	ic	fc	fd

Luego creamos ***n*** personas, ya que en el peor de los casos, todas las actividades tendrán conflicto con todas en el mismo momento.

Cada evento de inicio de actividad, tomará a la persona que esté más a la izquierda en la columna de “Personas Disponibles” en *Tabla 2* y la marcará por haber realizado al menos 1 actividad, cada evento de finalización “devolverá” a la columna a quien esté “ejecutando” la actividad.

Luego de llevar a cabo todos los eventos, se cuentan a las personas que han sido marcadas, teniendo así la cantidad mínima de personas que se necesitan para asignar todas las actividades.

En la siguiente tabla se ilustra paso a paso y en orden de los eventos la ejecución del algoritmo propuesto.

Evento	Personas Disponibles	Actividades hechas por persona	Personas Ocupadas
Default	P1, P2, P3, P4	Ninguna	-
ia	_, P2, P3, P4	P1 = {a}	P1(a)
ib	_, _, P3, P4	P1 = {a} P2 = {b}	P1(a), P2(b)
fa	**P1**, _, P3, P4	P1 = {a} P2 = {b}	P2(b)
id	_, _, P3, P4	P1 = {a, d} P2 = {b}	P1(d), P2(b)
fb	_, **P2**, P3, P4	P1 = {a, d} P2 = {b}	P1(d)
ic	_, _, P3, P4	P1 = {a,d} P2 = {b,c}	P1(d), P2(c)
fc	_, **P2**, P3, P4	P1 = {a,d} P2 = {b,c}	P1(d)
fd	**P1**, **P2**, P3, P4	P1 = {a,d} P2 = {b,c}	-

Tabla 2: Ejecución paso a paso del algoritmo propuesto

Podemos observar que en la última fila de la columna “Personas Desocupadas” hay sólo 2 marcadas y no 3 como se vió en el ítem a) al ejecutar la misma situación, y a la vez la tabla nos muestra que P1 realizó las actividades ***a*** y ***d***, mientras que P2 realizó ***b*** y ***c*** como muestra *Esquema 2*.

e) Implemente su algoritmo para resolver el problema de número de personas mínimo utilizando las estructuras de datos que estime conveniente, y proporcione las razones por las que decide usar tales estructuras.

El código del algoritmo está adjunto (proy2.cpp) a este pdf.

Las estructuras de datos utilizadas para implementar el código son un Stack y algunos Vectores.

Los vectores son principalmente para simplificar la manera de almacenar datos como los eventos de inicio y término de las actividades o para almacenar las actividades que realizan las personas a medida que le son asignadas mediante push back que ya viene implementado y su costo es **o(1)**, así evitamos tener que crear arreglos dinámicos.

Un ejemplo de esto, a una persona P1 se le asignan las actividades ***a***, ***f*** y ***k***:

Tareas asignadas a P1	a

Tareas asignadas a P1	a	f

Tareas asignadas a P1	a	f	k

El Stack es la estructura de datos “principal” en la implementación, podría decirse que es el corazón del código, ya que su metodología LIFO es perfecta para representar a las personas “desocupadas” en cada evento. La estructura almacena en ella a ***n*** personas, donde ***n*** es la cantidad de personas que tendríamos en el peor caso.

A quien esté en el tope del stack, se le asignará la actividad que inicia dado por el vector de eventos ordenado, y de la misma forma, cuando una persona finalice una actividad, estará nuevamente “desocupada” y volverá al stack. Ambas operaciones en **O(1)**.
De esta manera, nos aseguramos de no dar actividades a personas por sobre el mínimo.

El ejemplo de ejecución puede verse en *Tabla 2*, donde el tope del stack está representado por la persona desocupada más a la izquierda, a quien siempre se le asigna la actividad que comienza según el orden de los eventos.

f) Implementar un generador aleatorio que reciba el total de actividades *n*, el tiempo de inicio y termino dentro de un rango [Si, Fi]

El código del generador está adjunto a este PDF bajo el nombre de “generador.cpp”

Crea un archivo con los datos llamado “randInput.txt”

g) Implemente una función que permita leer desde un archivo un conjunto de actividades donde cada línea tiene el formato s_i, f_i de cada actividad.

Para realizar esto, dentro del archivo principal (*proy2.cpp*) se programó la siguiente función:

```
void leerArchivo(int *n, vector<int> *start, vector<int> *finish){

  char name[20];
  int temp1, temp2;

  cout << "Ingrese el nombre del archivo con extension. Ej: input1.txt" << endl;
  scanf("%s", name);
  getchar();

  FILE *fp = fopen(name, "r");

  fscanf(fp, "%d", n);
  //cout << "\nSe registraron " << *n << " actividades"<<endl;

  for(int i = 0; i < *n; i++){
    fscanf(fp, "%d %d", &temp1, &temp2);
    start -> push_back(temp1);
    finish -> push_back(temp2);
  }

  fclose(fp);

}
```

Imagen 1: Código correspondiente a la función "leerArchivo"

Esta función recibe la cantidad de actividades *n*, el vector con los inicios de cada actividad, y el vector con el fin de cada actividad.

Luego el programa solicita el nombre del archivo que uno desea leer, este se abre mediante *fopen.*

Posteriormente se lee una primera línea con la cantidad de actividades en el archivo y en cada línea siguiente se tienen los tiempos de inicio y término de las actividades.

Los tiempos de inicio se guardan en start, mientras que los tiempos de término se guardan en finish.

h) Modificar algoritmo para resolver el problema de coloreado del grafo

En esencia, el problema de colorear el grafo ya estaba resuelto con el algoritmo que habíamos escrito, pero fue necesario realizar algunas modificaciones al código fuente. *El código correspondiente a esta pregunta se encuentra en el archivo grafo.cpp.*

Se ha decidido explicar con personas y actividades puesto que así es más claro. Se agregó un boolean "isActive" a las personas para poder verificar si hay actividades ocurriendo en el actualmente. Al popear una persona del stack se asigna que está activa puesto que está realizando una actividad. Antes de setearlos como activos, se revisa dentro de las otras personas si están realizando una actividad. Si alguna persona está realizando una actividad, es porque se ha generado un conflicto y ahí es cuando se genera una arista. Esta arista se guarda en un vector de aristas, con el ID de ambas actividades que generaron el conflicto.

Al final de la función, se revisan las actividades que han sido realizadas por cada persona. Si la persona 0 realizo las actividades 0 y 1, entonces las actividades 0 y 1 tienen el mismo color en el grafo.

Los nodos del grafo están dados por todas las actividades que fueron realizadas por las personas. Por ejemplo, si las personas 3 y 2 realizaron actividades, y la persona 3 realizó la actividad 0 y 1, mientras que la persona 2 realizar las actividades 2 y 3, se tienen 4 nodos, 0, 1, 2 y 3.

De esta forma, al modificar un poco el código, hemos logrado obtener los nodos, las aristas y los colores correspondientes a cada nodo en el grafo.

Al final el código entrega un archivo de texto llamado "*inputPy.txt*", donde la salida son 3 lineas, la primera con los nodos, la segunda con las aristas, y la tercera con los colores correspondientes a cada nodo.

i) Escribir un script en Python que permita desplegar los grafo de intervalos coloreados por su implementación

Adjunto a este PDF se encuentra el script grafo.py el cual lee el archivo de texto *inputPy.txt* generado por *grafo.cpp* y despliega un grafo donde cada nodo representa una actividad, cada arista un conflicto entre 2 actividades y el nombre de cada vértice.

Conclusiones

Luego de realizar este proyecto ha mejorado considerablemente la comprensión de algoritmos greedy y porque estos son una herramienta útil dentro del repertorio de un programador.

Aún resulta difícil identificar las soluciones greedy y porque estas funcionan, pero la experiencia que hemos adquirido de este proyecto ha sido de gran ayuda.

Fue bastante complicado trabajar con python, puesto que ninguno de los 2 ha programado en ese lenguaje previamente.