Practice 2

Abraham Álvarez Cruz

Department of Computer Science
School of Engineering
University of Cádiz

abraham.alvarezcruz@alum.uca.es

11th November 2023

Summary

En este informe trataremos el problema de la *Planificación de exámenes* dándole una solución codificada en C++ e intentando presentar todo lo referente al problema y sus características así como el análisis de los resultados obtenidos por la solución planteada.

Contents

Tables

Figures

1 Introduction

Los problemas de asignación de recursos son muy útiles desde el punto de vista de la computación porque son problemas que se pueden extrapolar a otros ámbitos y tareas íntimamente relacionadas con ella. En nuestro caso concreto estamos tratando el problema de la planificación de exámenes. Este problema se caracteriza porque no tiene una solución analítica que nos permita obtener la solución óptima (o al menos una de ella si es que hubiese varias soluciones óptimas) de manera directa.
Esta falta de existencia de una solución analítica nos lleva a tener que buscar otros enfoques y formas de poder resolver el problema. Existen distintas formas y enfoques para encontrar soluciones al mismo pero todas tienen el mismo problema y es que son muy costosas computacionalmente (todas son de órdenes exponenciales).
Nuestro enfoque ha sido reducir el problema a uno para el que sí contamos con soluciones conocidas (igualmente son de orden exponencial) y es el problema del coloreado de grafos. Para la resolución del problema del coloreado de grados utilizamos un esquema de búsqueda de retroceso (también conocido cómo backtracking), en el cuál, a cada paso vamos procesando un determinado nodo del grafo sobre el que se está trabajando y se va construyendo una solución parcial hasta que se satisfacen una serie de condiciones, es entonces cuando se devuelve la solución parcial generada y podemos afirmar que tenemos una solución final o total[1]
El principal problema de este tipo de algoritmos de búsqueda con retroceso es que son procesos de búsqueda exhaustiva, ello conlleva que sean muy lentos y les demore mucho poder encontrar una solución o incluso no encontrar ninguna. Esto es debido a que esta familia de algoritmos van comprobando todas las posibles combinaciones que se pueden generar a cada paso, generando así un árbol de búsqueda que explora todas las posibles situaciones que se pueden dar en el espacio de búsqueda del problema.
Existen algunas formas de optimizar estos algoritmos buscando maneras de *podar* el árbol de búsqueda del problema para acotar la búsqueda y realizar menos operaciones.

[1]Según la implementación del algoritmo puede ser que la *solución final* no sea realmente una solución válida al problema y sólo sea una solución parcial que estaba en proceso de construcción.

2 Methods

Para la resolución del problema partimos de un pseudocódigo que presenta una solución empleando la mencionada técnica de backtracking. En dicho algoritmo se ha incluido además, una pequeña mejora con la que se pretende podar el árbol de búsqueda cuando se cumpla una condición. En esencia, dicha condición consiste en no continuar por ramas en las cuáles "se empleen más colores de los que ha necesitado la mejor solución encontrada hasta el momento".
El pseudocódigo del algoritmo codificado se muestra en la Figura 1. Para su funcionamiento el algoritmo necesitaría:

- **G**: Grafo a procesar por el algoritmo que tiene una serie de vértices (**V(G)**) y aristas (**E(G)**).
- **k**: Mayor color usado hasta el momento en una solución parcial.
- **B**: Menor color usado en una solución completa[2]
- **C**: Vector con los conjuntos de colores asignables a cada vértice del grafo, de esta manera, C[v] sería "el conjunto de colores asignables al vértice v".

El comportamiento del algoritmo sería aproximadamente el siguiente:

1. El algoritmo comprueba que queden vértices del grafo por procesar. En caso negativo, devuelve el mayor color usado hasta el momento por la rama procesada (k).
2. Cuando quedan vértices por procesar se comprueba que a todos se les puedan asignar colores (al menos uno) menores a una cota superior (B).
3. Como quedan vértices por procesar con colores menores a la cota, esto quiere decir que podemos encontrar una asignación más óptima para los vértices del grafo. Entonces se escoge un vértice del grafo (v) y para cada uno de los posibles colores que se le puedan asignar al vértice que sean menores que la cota superior (B) se hace:
   (a) Eliminamos el color escogido (c) del conjunto de colores asignables a los adyacentes del vértice que se está procesando (v).
   (b) Realizamos una llamada recursiva a la función pero como valor del parámetro k le pasaremos el *mayor valor entre el k actual y c* (color escogido).
   (c) El algoritmo nos retornará entonces un valor, llamémosle B'. Actualizaremos el valor de B con B'. Esto se hace de manera unilateral porque, en el peor de los casos, B' valdrá lo mismo que B y en cualquier otro caso será menor lo que implicaría haber encontrado una solución mejor a la actual.

[2]Una solución completa es aquella que pertenece al conjunto de soluciones del problema, dicho de otra manera, es una solución válida al problema.

```
backGCP(G, k, B, C) → B
if V(G) = ∅
  return k
for all v ∈ V(G)
  if no-colour-under-bound(C[v], B)
    return B
v ← select(G)
for all c ∈ C[v]
  if c < B
    for all u ∈ adjacents(v)
      remove(c, C[u])
    B ← backGCP(G - {v}, max(k, c), B, C)
    for all u ∈ adjacents(v)
      insert(c, C[u])
return B
```

Figure 1: Pseudocódigo del algoritmo de coloreado de grafos.

(d) Después de haber hecho la llamada recursiva, devolvemos el color escogido al conjunto de colores asignables a los adyacentes del vertice que se está procesando.

4. Por último, tras haber procesado todos los posibles colores asignables al vértice v, se devuelve B que estará actualizada al mejor valor encontrado al procesar la rama actual.

Como podemos ver, el algoritmo no llega a ser del todo complejo aunque es cierto que no es muy intuitivo ya que la realización de llamadas recursivas engorrona el proceso de entendimiento del algoritmo.

En nuestro caso, la implementación de este algoritmo necesita de una serie de pequeños cambios adicionales que nos permita almacenar la asignación de colores realizada. Esto lo hago añadiendo un parámetro adicional bS que es un vector en el que almaceno, para cada uno de los vértices del grafo, el color que finalmente se le ha asignado.

Partiendo de la descripción anterior, los cambios que he incorporado al algoritmo han sido:

1. Se mantiene igual. Se devuelve k junto con el vector bS.
2. Se mantiene igual. Se devuelve B junto con el vector bS.
3. Se mantiene igual. Adicionalmente se crea una copia de bS (llamemosle $bSCopy$). Esta copia contendrá la mejor asignación encontrada realizar las llamadas recursivas.
   (a) Se mantiene igual.
   (b) Se mantiene igual.
   (c) La llamada recursiva retornaría ahora dos valores: B' y bS'. La actualización de B no es unilateral ya que, si B' es menor que B, implica que hemos hallado una mejor solución y por ello tendríamos que actualizar $bSCopy$.
   (d) Se mantiene igual.
4. Se mantiene igual. Se devuelve el valor de B y $bSCopy$.

Con los pequeños cambios realizados al algoritmo original ahora podemos saber aparte de la cantidad mínima de colores necesaria para colorear el grafo también sabemos qué color hay que asignar a cada uno de los vértices.
En la práctica la implementación del algoritmo en $C++$ es un poco más compleja ya que hay que tener en cuenta una serie de cosas que a más alto nivel no. Algunas de las mismas son:

- Empleo o no de punteros.
- Estructuras usadas para representar las cosas (grafo, colores...)

- Etc.

En mi caso, para la representación del grafo emplee una clase vista en la asignatura EDNL donde se emplea una matriz de adyacencia. Eliminé algunos métodos y funciones supérfluos e introduje otros que me ayudasen a resolver el problema o a reducir la complejidad de ciertas operaciones.
Generalmente he elegido paso por valor en vez de referencias (aunque ello implique un coste adicional) ya que son algoritmos complejos de depurar y prefería presentar una solución (aunque sea más costosa) en un menor tiempo antes que presentar una solución más óptima a costa de emplear demasiado tiempo en la asignatura dejándo de lado al resto. No obstante, la implementación que presento podría modificarse sin demasiada (aparente) complejidad ayudando a reducir el coste de las copias de ciertas estructuras costosas (vectores y conjuntos).
Finalmente, se proponía en el guión de la práctica aplicar una pequeña optimización que consistía en "seleccionar el vértice a procesar según el grado del mismo en el grafo". Esta mejora consiste en seleccionar primero "aquellos vértices que son más conflictivos" al ser adyacentes a una mayor cantidad de vértices. De esta manera, seleccionando primero a esta serie de vértices podemos reducir el tamaño del árbol de búsqueda ya que conseguimos "detectar con mayor rapidez (etapas más tempranas) aquellas ramas que no van a conducir a una solución del problema, evitando así posponer dicha tarea de descarte a niveles más profundos del árbol de búsqueda".
Implementé dicha modificación en paralelo a la versión sin optimizar. En el siguiente apartado presento los resultados obtenidos para ambas versiones.

3 Results and discussion

En general ya conocemos el comportamiento en términos asintóticos del algoritmo que estamos trabajando. Al realizar una búsqueda exhaustiva es casi imposible que un algoritmo tenga un orden de eficiencia computacional en términos de tiempo inferior a una exponencial, esto es: k^N siendo k un valor constante.
Al pertenecer a esta categoría podemos afirmar que el algoritmo es *computacionalmente costoso* o que es *computacionalmente no eficiente*.

3.1 No-Optimized Version

La anterior hipótesis concuerda muy bien con las figuras 2 y 3. En ellas se puede apreciar perfectamente como el orden asintótico de nuestro algoritmo se encuentra en algún punto intermedio entre 2^N y 10^N. Esto va en línea con la idea inicial que teníamos de la **no** pertenencia de este algoritmo al conjunto de algoritmos *computacionalmente eficientes*.

Cabe recalcar que para la obtención de los resultados aquí presentados tuve que emplear un tamaño de entrada bastante reducido (concretamente $N = 12$) ya que para la propuesta del guión de la práctica, cuando $N = 15$, el ordenador llevaba alrededor de 20-25 minutos calculando la solución y no terminaba de arrojar ninguna solución.

3.2 Optimized Version

La versión optimizada, aunque arroja pequeñas mejoras en los tiempos tomados por el algoritmo para devolver una solución, no terminan de ser suficientes para reducir el orden del algoritmo, perteneciendo al mismo orden que en la anterior versión.
En las figuras 4 y 5 presento los resultados obtenidos por la versión optimizada del problema. De nuevo se puede observar como el orden asintótico no varía, dejando esta optimización como una pequeña mejora con efectos residuales casi imperceptibles en el impacto asintótico del mismo.

Finalmente, para poder visualizar las mejoras en términos absolutos de la versión optimizada frente a la no optimizada, he recogido los tiempos obtenidos para cada tamaño de entrada en la tabla 1.

Tamaño de entrada	Tiempo (versión optimizada)	Tiempo (versión clásica)
2	8	6
3	11	18
4	24	48
5	89	216
6	446	1281
7	3327	8206
8	28175	31748
9	260039	269140
10	2866567	2901361
11	33604228	3410909
12	435000891	435730094

Table 1: Tiempos para la ejecución del algoritmo. La medida está en microsegundos.

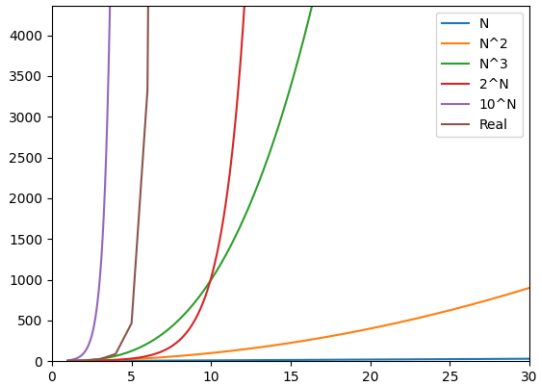

Figure 2: Tiempo vs Tamaño de entrada para la versión sin optimizar del algoritmo de coloreado de grafos.

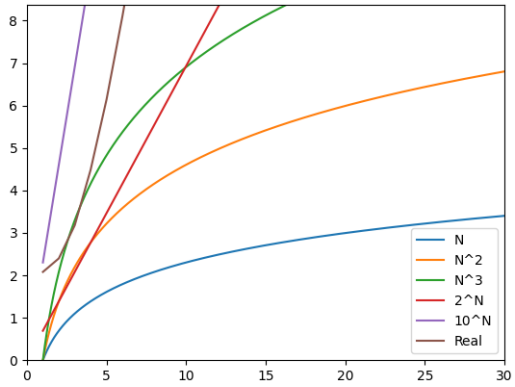

Figure 3: Logaritmo del tiempo vs Tamaño de entrada para la versión sin optimizar del algoritmo de coloreado de grafos.

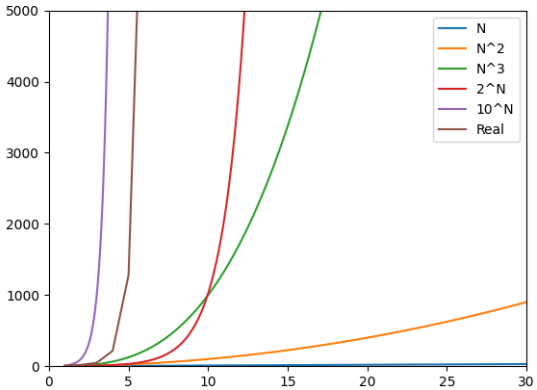

Figure 4: Tiempo vs Tamaño de entrada para la versión optimizada del algoritmo de coloreado de grafos.

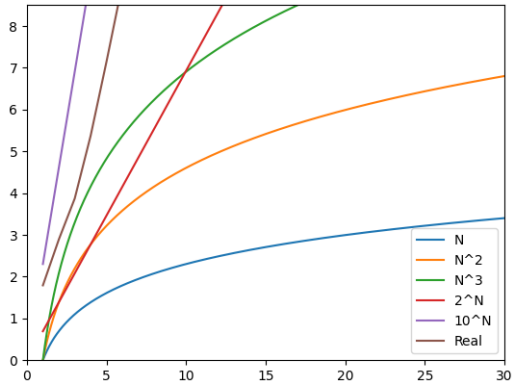

Figure 5: Logaritmo del tiempo vs Tamaño de entrada para la versión optimizada del algoritmo de coloreado de grafos.

4 Conclusions

El algoritmo en pseudocódigo es fácil de entender una vez que lo miras con detenimiento aunque al principio puede costar un poco entenderlo. La implementación del mismo en *C++* si que me ha ocasionado algunos problemas (especialmente al tratar con puntero), aunque he podido arreglar los fallos surgidos sin demasiada dificultad.
En cuanto al generador de instancias del problema, con los números presentados en clase, es imposible obtener un grafo que no sea completo. Todas y cada una de las instancias generadas han creado conflictos con el resto de vértices del grado lo cuál ocasiona que el algoritmo esté siempre en el peor caso posible (tiene que recorrer un mayor número de nodos del árbol de búsqueda).
También a la hora de calcular los tiempos de cada versión del algoritmo tuve problemas por la cantidad de tiempo que tomaba para valores de $N > 12$. Es por eso que tuve que acotar los tamaños de entrada del algoritmo a un rango inferior para poder obtener tiempos suficientes con los que poder ver la forma aproximada de la función en una gráfica.
Aunque todos los casos vistos han conducido a una búsqueda casi exhaustiva del árbol de búsqueda del problema, me ha ayudado a comprender mejor los problemas de este tipo de algoritmos que tienen que realizar búsquedas completas en su árbol de búsqueda y cómo hemos podido dar solución a un problema de la vida real (muy relacionado con otros tipos de problemas de otras áreas), reduciendo el problema a uno ya conocido y para el cuál tenemos un algoritmo que nos pueda proporcionar una solución.

Bibliography

[1] Oded Goldreich. *P, NP, and NP-completeness. The basics of computational complexity*. Cambridge University Press, 2010.