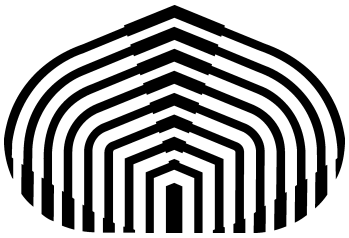

Universidad Simón Bolívar

Diseño de Algoritmos I

Exámen I

Lorenzo Fundaró - 0639559

29 de marzo de 2010

Índice

1. Introducción

1.1. Motivación del Proyecto

Encontrar la coloración mínima es un problema de comoplejidad NP. Sus soluciones son ampliamente aplicadas en casos de la vida real, a nombrar:

- Problema de planificación de horarios
- Asignación de frecuencia a radios móviles.
- Ubicación de registros en la computadora.
- Análisis de datos arqueológicos y biológicos

Por éstas razones el problema de coloración mínima en un grafo se hace interesante hasta el punto de tratar de resolver dicho problema con algoritmos optimizados.

1.2. Breve descripción del problema

El problema se aborda con ayuda de la combinación los algoritmos Brelaz+Interchange y Enumeración Implícita. El algoritmo de Enumeración implícita es una combinación del algoritmo propuesto por Kubale y Jackowski y el procedimiento de etiquetado propuesto por Peemoller. El Brelaz+Interchante es usado para encontrar un cota superior y una cota inferior, luego el algoritmo de Enumeración Implícita usa estas cotas para podar el árbol de explorado, reducir su profundidad (estableciendo una clique que siempre esta coloreada de la misma manera) y detectar cuando se ha llegado a una coloración mínima durante la exploración del árbol.

1.3. Descripción del contenido del informe

En este informe se explica el concepto de diseño que se utilizó para lograr el objetivo, así también como los detalles de implementación, instrucciones de operación, estado actual, conclusiones y referencias bibliográficas.

2. Diseño

2.1. Descripción y justificación del modelo utilizado para representar el problema

Para representar el grafo en el computador se utilizó una matriz de adyacencias que consta de un arreglo de igual tamaño al número de vértices del grafo, cada posición del arreglo contiene la información del vertice numerado con esa posición, así, la información del vértice cero esta al principio del arreglo, la del vertice cuatro esta en la posición cuatro del arreglo... Cada casilla de dicho arreglo es una estructura

Graph que contiene un apuntador a un arreglo de adyacencias, un color, un arreglo que llamamos color-around y una estructura Label que se define más adelante en la sección de estructuras de datos. Se decidió utilizar un arreglo ya que proporciona acceso constante a sus elementos.

Para las adyacencias se podría haber utilizado una lista enlazada ya que cada vez que se requiere saber los adyancentes a un vértice dado siempre es necesario recorrerlos todos, y generar una lista de enlazada con los adyacentes es más fácil y rápido que hacer un arreglo (debido a que no se sabe cuanto va a medir ese arreglo durante la primera lectura del archivo con la especificación del grafo), sin embargo, la lista enlazada podría inutilizar el cache del procesador, mientras que el arreglo no.

No se decidió hacer una matriz de NxN porque en ese caso, para recorrer los adyacentes a un nodo, se deben recorrer invariablemente N casillas, lo cual representa tiempo gastado inútilmente, sobre todo para grafos poco densos, ya que es necesario recorrer las casillas de los nodos que no son adyacente y las de los que son adyacentes. Debido a esto vemos que la estructura de matriz de adyacencias se comporta mejor que la matriz de NxN para grafos poco densos e igual de bien para grafos muy densos.

2.2. Estructuras de datos y algoritmos involucrados en la aplicación

2.2.1. Estructuras de datos

- **Graph**: Estructura que se utiliza para representar un vértice en la matriz de adyacencias. El apuntador a arreglo de ints permite consultar el arreglo de adyacentes. El apuntador a entero llamado color-around es un arreglo de ints que permite saber cuáles colores son adyacentes al vértice. La estructura Label permite saber si ese nodo está etiquetado y a que profundidad de la rama explorada se encuentra.
- **Label**: Tupla que consta de dos enteros, uno es un flag que indica si el vértice esta etiquetado o no y el otro es su profundidad de la rama que se estaba explorando cuando fue etiquetado.
- **Tuple-List**: Lista enlazada de tuplas que contienen dos enteros, estas se utilizan para generar las listas de los nodos etiquetados. Uno de los int es el número del nodo y el otro es su profundidad en la rama explorada actualemente.
- **Tuple**: Tupla que sirve para representar los vertices con su grado en el arreglo ordenado por grados que pide DSTATUR.
- **Pair**: Pair es la estructura que retorna Dsatur. Clique representa la cota inferior de coloración. Coloring representa la cota superior.

Members es un arreglo que contiene los miembros que conforman la clique máxima encontrada por el algoritmo.

- **Linked-Array-list**: contiene un apuntador a arreglo, un color, y un apuntador a un arreglo. Es utilizada en interchange para guardar componentes de un grafo representadas como conjuntos vectoriales.

2.2.2. Algoritmos involucrados

- **Dsatur+interchange**: para el ordenamiento decreciente de los vértices se utilizó el Algoritmo Quicksort. Los grados de saturación sólo se actualizan para los vértices que no han sido coloreados, de esta manera se ahorra costo de operaciones. El algoritmo recibe un argumento llamado start-point el cuál es -1 para indicar que se quiere hayar la cota superior y recibe algún número mayor o igual que cero hasta el número de vértices del grafo cuando se quiere encontrar la clique máxima (cota inferior). Éste algortimo usa los siguientes algorimos:
  - **get-max-degree**: función que en caso de ocurrir una igualdad en los números de saturación devuelve el próximo vértice no coloreado de grado mayor.
  - **update-satur**: modifica estructuras de datos en vértices adyacentes a v(sub)i en el momento que se colorea un vértice v(sub)i todos los adyacentes a éste sufren un aumento de saturación siempre que el color utilizado no sea el de uno adyacente a un adyacente de v(sub)i. Si el grado de saturación de un elemento es -1 significa que dicho elemento ya fue coloreado. Por otro lado, se aprovecha de colocar en 1 la casilla de corresponde al color utilizado por v(sub)i indicando que el existe un vértice en las adyacencias del vértice con un color dado.
  - **leasp-color**: retorna el menor color posible. Dado un vértice v(sub)i que se quiere colorear se utiliza la estructura color-around. Sobre ésta se itera desde el principio hasta conseguir alguna casilla en 0 (indicando la ausencia del color i en la casilla).
  - **repeated**: algoritmo que en caso de encontrar una repetición de grados de saturación devuelve el próximo vértice de grado mayor no coloreado.
- **Degree**: prepara los vértices en una estructura vértice-grado llamada tupla, luego todos estos elementos se almacenan en un arreglo de tuplas llamado deg-vert.
- **Compare**: función de comparación utilzada por Quicksort.
- **Enumeración-Implícita**: este algoritmo prosigue tal como se enuncia en el proyecto. No tiene optimizaciones que pueda hacer más pequeño el umbral de soluciones. Sin embargo en un principio

se ideó colocar la clique máxima hayada por Brelaz+Interchange al principio del arreglo de vértices que se permutaría, de manera que los vértices pertenecientes a dicha clique quedarían fijos y se permutaría el resto. En un momento dado cuando se colorea la parte permutada del arreglo si se detecta una secuencia de coloraciones distintas, se chequea si esa secuencia de vértices forman un clique entre sí. En caso de ser así, se comprobaría que el número de tales vértices es mayor que la cota inferior hasta ahora registrada, en cuyo caso se actualizaría dicha cota. Por otro lado si los vértices que conforman la secuencia de coloraciones distintas están todos conectados entre sí por un arco directo y éstos a su vez se conectan con otro arco directo a los vértices fijos del arreglo entonces podemos agregarlos a la parte fija del arreglo.

- **twoOnN**: algoritmo que cálcula la combinatoria de 2 elementos en número colores.
- **interchange**: se calculan las componentes por medio de un DFS en los nodos. Se verifica la condición de intercambio y se hace el intercambio en caso de proceder.

3. Detalles de Implementación

3.1. Reseña de los elementos implementados, problemas encontrados y solución

3.1.1. Representación y Actualización Del Grafo

Para la representación del grafo el principal desafío fue poder actualizar coloración y saturación rápidamente, incluso después de un intercambio.

El grafo esta representado como un arreglo de listas de adyacencias. Cada índice representa un nodo, los cuales enumeramos a partir de cero. Cada índice del arreglo guarda el color del nodo, un apuntador a su lista de adyacentes y un apuntador a un arreglo que guarda la cantidad de vecinos que tienen un determinado color al que llamaremos **color-around**, hablaremos de **color-around** más adelante.

Las listas de adyacentes solo contienen el índice del vecino en el arreglo de listas de adyacencias. Estas listas ameritan ser liberadas de memoria en tiempo O(Grado del grafo).

El color se guarda como un entero, ya que también numeramos los colores desde el cero. Para nodos no pintados, el color es -1. color-around contiene, para el indice i, el numero de vecinos que están coloreados con el color i. Este arreglo es tan largo como el número de vértices de grafo.

Además de esta estructura, también mantenemos un arreglo, llamado degree-vert, que mantiene un registro de la saturación de los nodos del grafo. degree-vert contiene para el índice i, la cantidad de colores distintos adyacentes al vértice i. Este arreglo también es tan largo como nodos tenga el grafo.

El hecho de que las listas solo guarden el índice del vecino y el hecho de que el color se guarde directamente en el arreglo con las listas de adyacencias nos permite actualizar el color de un nodo en O(1). color-around combinado con degree-vert nos permite actualizar la saturación en O(grado(i)), siendo grado(i) la función que devuelve el grado del vértice i. Cuando se colorea un nodo, basta con chequear para cada vecino si ya tenían referencias al nuevo color del nodo, si no tenían tales referencias se aumenta en uno(1) su grado de saturación en degree-vert, luego se aumenta en uno(1) el numero de referencias en color-around. Cuando se cambia el color de un nodo basta con (para cada vecino) restar en color-around uno(1) a las referencias del color viejo, y si llegase a cero, se disminuye la saturación en degree-vert en uno(1), la actualización por el color nuevo es igual a cuando se colorea un nodo por primera vez.

3.1.2. Intercambio

Hubo dos temas cruciales en el diseño de intercambio, uno fue la búsqueda y representación de las componentes inducidas por los dos colores elegidos, el otro fue el cálculo de todas las combinaciones de dos colores del conjunto de colores adyacentes el nodo del intercambio.

Encontrar componentes en esta estructura puede lograrse, gracias a las listas enlazadas, en O(Nodos alcanzables desde el punto de partida), usando DFS o BFS. Nosotros elegimos DFS por su simplicidad.

Para guardar las componentes encontradas en intercambio se usa una lista enlazada simple que contiene apuntadores a arreglos. Estos arreglos se crean de tamaño igual a la cantidad de vértices del grafo y se utilizan como conjuntos vectoriales (1 si el elemento esta, 0 en caso contrario). Estos arreglos vectoriales nos permiten revisar rápidamente si un nodo pertenece o no a una componente y llenarlos también es fácil y rápido. Su desventaja es que ameritan que sean liberados de memoria cuidadosamente y en tiempo O(Número de componentes).

Nunca vimos la necesidad de inducir un grafo. La exploración en un grafo inducido la pudimos lograr discriminando que nodos examinar viendo sus propiedades.

Para el cálculo de las combinaciones de colores ideamos un algoritmo que solo funciona para calcular combinatorias de dos elementos, pero que solo amerita un(1) intercambio para obtener siguiente combinación. La desventaja de este algoritmo es que requiere de una pequeña estructura de control para que pueda funcionar a través de varias llamadas la función que lo contiene.

El algoritmo se basa en el hecho de que, una vez que se combina un elemento con todos los demás elementos de su conjunto, se puede dejar de considerar ese elemento en las combinaciones posteriores, y se puede operar recursivamente sobre lo que sobra del conjunto.

3.1.3. Enumeración Implícita

El problema más grande fue calcular las permutaciones, queríamos un algoritmo iterativo y no recursivo para el resolver el problema, así que usamos el algoritmo de Dijkstra, el cual para una permutación genera la próxima permutación en orden lexicográfico.

4. Instrucciones de Operación

4.1. Código fuente de aplicación

El código fuente se puede conseguir en el archivo tar.gz que viene con éste informe.

4.2. Descripción detallada de como compilar y correr la aplicación

4.3. ¿Cómo descomprimir el archivo tar.gz?

En una cónsola de linux se debe ejecutar el siguiente comando:

```
tar -vzxf proyecto1-Fundaro-Jaber.tar.gz
```

A continuación se crea una carpeta llamada proyecto1 dónde se encuentran los archivos fuentes del proyecto.

4.4. ¿Cómo compilar?

En una cónsola de linux debe dirigirse a la carpeta donde estan los archivos fuentes, desde allí se deberá ejecutar el siguiente comando:

```
make all
```

Con esto se compilan todas las fuentes. En caso de querer borrar los archivos generados por la compilación se procede hacer:

```
make clean
```

4.5. ¿Cómo ejecutar?

En la carpeta de archivos fuentes se provee de un script llamado script.sh. Si se quieren ejecutar todas las instancias en el comando debe ser el siguiente:

```
./script.sh resultado
```

donde resultado es el nombre del archivo donde arrojarán los resultados de la corrida. Si se quiere, éste archivo puede recibir cualquier nombre diferente a resultado.

Si se quiere ejecutar una instancia particular se procede con el siguiente comando:

```
./main < grafos/instancia-particular
```

4.6. ¿Cómo interpretar resultados?

Cada instancia arroja un resultado con la siguiente estructura:

```
1 Grafo 14-0,9-1.col
2 Resultados de Brelaz+Interchange
3 Cota superior = 9
4 Cota inferior = 7
5 Resultado de enumeración implícita
6 Número cromático = 9
7 Tiempo en segundos de ejecución del programa: 96.2635
```

Explicación de cada línea:

1. Nombre de la instancia, el primer número indica el la cantidad de vértices, el segundo la densidad y el tercero el número de archivo.
2. Indicador de algoritmo
3. Coloración arrojada por Brelaz-Interchange
4. Cota inferior. Clique máxima encontrada al aplicar Brelaz-Interchange N veces.
5. Indicador de algoritmo
6. Número cromático encontrado por enumeración implícita.
7. Tiempo en segundos según función provista en el proyecto

Si Brelaz-Interchange arroja una cota superior igual a la inferior entonces el algoritmo de enumeración implícita no se ejecuta.

5. Estado Actual

5.1. Indicación del estado final de la aplicación

El proyecto se encuentra funcional para la mayoría de las instancias excepto para una que se menciona más abajo en la sección de errores. Con la mayoría de las instancias el algoritmo funciona y cuando es posible (a menos que exceda 5 min.) devuelve el número cromático de una instancia dada.

5.2. Errores

La aplicación falla en la instancia 10-0,7-10.col. No se pudo averiguar la razón de dicha falla.

6. Otros

6.1. Demostración formal de Enumeración Implícita

6.2. Ejemplo de Brelaz+Interchange sin solución óptima

Si se corre el proyecto para la instancia 12-0,5-3.col la coloración que resulta de aplicar Brelaz+Interchange es 5. Sin embargo aplicando el algoritmo de enumeración implícita se puede notar que la coloración mínima del grafo es de 4 colores.

6.3. Tabla de resultados

7. Conclusiones y recomendaciones

El algoritmo de Dsatur-Interchange es útil para dar unas cotas de referencia al algoritmo de enumeración implícita. Sin embargo el algoritmo de enumeración implícita tarda una cantidad considerable de tiempo si no se aplica ningún tipo de optimizaciones que logren reducir el umbral de las soluciones. Se recomienda seguir las recomendaciones planteadas en la sección de Algoritmos involucrados donde se describe en el punto de enumeración implícita una manera de podar el árbol de soluciones.

8. Referencias Bibliográficas

- http://nicolas-lara.blogspot.com/2009/01/permutations.html (consultada el 17-02-10)
- http://en.wikipedia.org/wiki/Petersen-graph (consultada el 17-02-10)
- Meza Oscar, Ortega Maruja. "Grafos y Algoritmos". Editorial Equinoccio, Universidad Simón Bolívar. 2007. ISBN 980-237232-3
- Brélaz 1979 Daniel Brélaz. «New Methods to Color the Vertices of a Graph», Communications of the ACM 22-4, 1979, 251-256