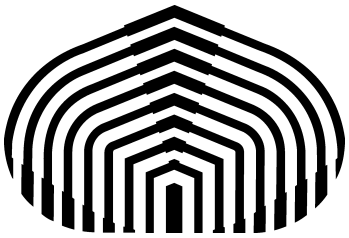

Universidad Simón Bolívar

Diseño de Algoritmos I

Proyecto II

Lorenzo Fundaró - 0639559
Germán Jaber - 0639749

30 de marzo de 2010

Índice

1. Introducción **4**
- 1.1. Motivación del Proyecto 4
- 1.2. Breve descripción del problema 4
- 1.3. Descripción del contenido del informe 4

2. Diseño **4**
- 2.1. Descripción y justificación del modelo utilizado para representar el problema 4
  - 2.1.1. Representación de Grafo 4
  - 2.1.2. Vértices adyacentes 5
  - 2.1.3. Historial de backtracks: Trace 5
  - 2.1.4. Estructura Popularity 5
  - 2.1.5. Máximo color utilizado 6
  - 2.1.6. Vértice de partida para Forward y de entrada para Backwards 6
  - 2.1.7. Mejor coloración obtenida hasta el momento 6
  - 2.1.8. Procedimiento de Etiquetado 6
- 2.2. Estructuras de datos y algoritmos involucrados en la aplicación 7
  - 2.2.1. Estructuras de datos 7
  - 2.2.2. Algoritmos involucrados 7
  - 2.2.3. Enumeración implícita 8

3. Detalles de Implementación **11**
- 3.1. Reseña de los elementos implementados, problemas encontrados y solución 11
  - 3.1.1. Representación y Actualización Del Grafo para Brelaz+Interchange 11
  - 3.1.2. Intercambio 11
  - 3.1.3. Enumeración Implícita 12
  - 3.1.4. Bugs encontrados 13

4. Instrucciones de Operación **14**
- 4.1. Código fuente de aplicación 14
- 4.2. Descripción detallada de como compilar y correr la aplicación 14
- 4.3. ¿Cómo descomprimir el archivo tar.gz? 14
- 4.4. ¿Cómo compilar? 14
- 4.5. ¿Cómo ejecutar? 14
- 4.6. ¿Cómo interpretar resultados? 15

5. Estado Actual **16**
- 5.1. Indicación del estado final de la aplicación 16

6. Otros **16**
- 6.1. Demostración formal de Enumeración Implícita 16
- 6.2. Tabla de resultados 18

7. Conclusiones y recomendaciones **19**

8. Referencias Bibliográficas **19**

1. Introducción

1.1. Motivación del Proyecto

Encontrar la coloración mínima es un problema de complejidad 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 estas 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 algoritmo "Brown's modified algorithm with look-ahead rule" propuesto por Peemoller, también se tomaron ideas del algoritmo de Brelaz. 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 en su exploració, reducir su profundidad y detectar cuando se ha llegado a una coloración mínima durante la busqueda de soluciones.

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

2.1.1. Representación de Grafo

Para representar el grafo en el computador se utilizarón arreglos de adyacencias, estos estan formados por 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.

2.1.2. Vértices adyacentes

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 inevitablemente 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 arreglos de adyacencias se comporta mejor que la matriz de NxN para grafos poco densos e igual de bien para grafos muy densos. Además añadiendo un ordenamiento a los arreglos de adyacencias, cada vez que se necesita saber si un vértice es vecino de otro, simplemente se busca ese vértice en log(n) utilizando búsqueda binaria.

2.1.3. Historial de backtracks: Trace

Debido a que el algoritmo de backtracking es iterativo no se cuenta con la pila para mantener un rastro del backtrack que se ha ido haciendo. Por lo tanto se decidió utilizar un arreglo de tamaño n (número de vértices) llamado **trace** en donde se van poniendo los vértices coloreados a medida que el procedimiento Forward avanza en la coloración. La traza siempre tendrá un cota hasta donde es posible leer datos confiables, y esta cota es la profundidad en el arbol de backtrack, es decir cuantos niveles se ha descendido en el árbol. La profundida se representa con la variable **depth**. Cualquier índice en la traza que esté después de la posición depth contendrá basura o datos que no son útiles para el algoritmo.

2.1.4. Estructura Popularity

Esta estructura es un arreglo de tamaño “Cota superior de Brelaz+Interchange” y cada índice se corresponde a un color. En cada ca-

silla se encuentra la popularidad de un color, es decir, si tenemos dos vértices coloreados con el color 4, la casilla 4 de popularity contendrá el número 2.

2.1.5. Máximo color utilizado

Se determina iterando desde la última casilla del arreglo Popularity hasta llegar al primer elemento que contenga una popularidad mayor estricta a cero. Se determina de esta manera ya que como popularity es de tamaño Cota Superior de Brelaz+Interchange entonces si la última casilla tiene un número mayor estricto que cero entonces dicho número representa el mayor color utilizado. Sino se busca en la penúltima casilla y así sucesivamente hasta llegar a la casilla 0.

2.1.6. Vértice de partida para Forward y de entrada para Backwards

La variable current-vertex se comparte entre los procedimientos Forward y Backwards. Ésta es la variable **r** que se utiliza en el algoritmo de Kubale y Jackowski.

2.1.7. Mejor coloración obtenida hasta el momento

Se denota por el arreglo **coloring** una asignación de colores a cada índice el cual representa un vértice del grafo.

2.1.8. Procedimiento de Etiquetado

Para que un vértice x_k sea candidato a etiquetarse, debe cumplir con tres propiedades:

1. Ser adyacente a x_k
2. Ser de menor rango que x_k
3. Ser el de menor rango entre todos los candidatos que cumplen las dos propiedades anteriores y que tengan el mismo color que el vértice en cuestión.

Una manera rápida de encontrar éste vértice candidato. Se hace de la siguiente manera:

- Se prepara un arreglo llamado colors inicializado en ceros.
- Se comienza a iterar desde el principio de la traza hasta una posición anterior al vértice desde donde se llama el procedimiento label. Se verifica que cumpla la propiedad 1. Luego se obtiene su color, y se verifica si ese color ya se ha encontrado antes durante este proceso con otro vértice. Si para tal color el arrecho colors está en cero, entonces estamos habremos elegido un candidato a ser etiquetado porque es aquel que cumple con las propiedades 1,2 y 3.

- Seguimos el procedimiento hasta llegar a una posición anterior al vértice desde donde se llama label.

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 los arreglos 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. La estructura FC es un arreglo que se agrega en este proyecto, el cual representa un conjunto de Colores Factibles (Feasible Colors). El entero adj-size es útil para saber la cantidad de vecinos de un vértice.

- **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**: Tupla que sirve para representar los vertices con su grado en el arreglo ordenado por grados que pide DSATUR.

- **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 las siguientes rutinas:

  - **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 que corresponde al color utilizado por v(sub)i indicando que 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.

- **compare-vertices**: función de comparación utilizada por Quicksort y Búsqueda binaria.
- **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. Éste arreglo es utilizado en Enumeración implícita y en Dsatur.
- **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.

2.2.3. Enumeración implícita

Función que encuentra una coloración mínima por medio de una estrategia de backtracking.

Se implementó el algoritmo “Brown’s modified algorithm with look-ahead rule” de Peemoller con varias modificaciones y correciones. Se utilizó como plantilla el algoritmo generalizado de enumeración implícita propuesto por Kubale y Jackowski. Las modificaciones al algoritmo de Peemoller fueron:

- El próximo vértice a colorear se decide en base al grado de saturación, no en base al número de prevenciones y bloqueos.
- El próximo color a usar se decide en base a la popularidad de ese color, no se elige el menor color posible.

- Cuando se consigue una coloración completa los FC se vuelven a generar a partir de una cota que está por debajo de todos los colores que se usaron en la coloración pasada.
- Cuando existe un color que bloquea a algún vértice no coloreado, se llama el procedimiento de label para etiquetar los vértices que cumplen las propiedades de labeling, pero no se procede hacer backtracking a menos que el vértice con el color bloqueante tenga un FC vacío.

El algoritmo explora el grafo de todas las posibles particiones de los nodos del grafo, sin embargo, por la manera en que se generan los hijos de los nodos, podemos decir que exploran todas las posibles coloraciones parciales y completas del grafo. La manera concreta en que se generan los nodos del árbol es la que sigue:

- Un nodo es válido si todos los nodos pertenecientes a un mismo conjunto no son adyacentes entre sí. Interpretamos el hecho de que una cantidad X de nodos pertenezcan a un conjunto como que esos X nodos están pintados con el mismo color.
- Un sucesor válido de un nodo es aquel donde se ha anãdido el vértice con mayor grado de saturación a uno de los conjuntos de la partición donde no sea adyacente a ningún vertice de ese conjunto, o donde se ha creado un nuevo conjunto y anãdido el nodo a ese conjunto.
  Tenemos que, por el teorema Brown de asignación de atributos indistingibles evitando generar soluciones redundates, se exploran todas las posibles coloraciones parciales y completas del grafo evitando por completo la redundancia de soluciones.

Este grafo implicícito se recorre de manera iterativa como sigue:

- Se encuentra una clique y cota inferior usando Brelaz. Se colorean los vértices de esa clique.
- Se encuentra una cota superior aplicando Brelaz+interchange.
- El próximo vértice a colorear será siempre el que tenga mayor grado de saturación.
- Para colorear un vértice se elige del conjunto de colores que hallan sido usados anteriormente y que no generen una coloración inválida, o se introduce un nuevo color. Para determinar exactamente en que orden se eligirán los colores se implementaron dos heurísticas.
  - Se elige el color más usado o más popular en la coloración explorada que no halla sido usado anteriormente para colorear el vértice que se está procesando.
  - Se elige el color menos usado o menos popular en la coloración explorada que no halla sido ya usado anteriormente para colorear el vértice que se está procesando. Sin embargo se asegura que introducir un nuevo color siempre será considerado como la última opción.

No se consideran aquellos colores que sean mayores a la cota superior previamente calculada. Tampoco se consideran los colores que hagan el conjunto de colores de alguno de los vecinos adyacentes al nodo procesado vacío (look-ahead), si esto llega a pasar, se aplica el algoritmo de etiquetado para look-ahead propuesto por Peemoller.

- Se repiten los dos pasos anteriores hasta encontrar una coloración completa o hasta llegar a un nodo con conjunto de colores posibles vacío.
- Tenemos dos posibilidades:
  - Se encontró una coloración completa. En ese caso se guarda esta nueva coloración y se revisa si esa usa tantos colores como la cota inferior, de ser ese el caso, el algoritmo termina, si no, se reduce la cota superior en uno y se hace backtrack hasta el primer vértice coloreado con el color más alto usado y se aplica el proceso de labeling propuesto por Peemoller.
  - Si se llegó a un vértice con un conjunto de colores posibles vacío, se aplica el procedimento de labeling de Peemoller directamente.
- Una vez que se a aplicado labeling y se ha llegado al vértice desde donde se reanudará la exploración, se repite todo el proceso desde el paso número tres. En caso de que el proceso de labeling no pueda conseguir un nodo desde donde resumir la exploración, el algoritmo retorna y devuelve la mejor coloración hasta el momento encontrada, que será la óptima como veremos más adelante.

3. Detalles de Implementación

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

3.1.1. Representación y Actualización Del Grafo para Brelaz+Interchange

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 algoritmo se implemento siguiendo el esquema propuesto por Kubale y Jackowski, por lo que el algoritmo dividido en dos funciones principales, Forward y backwards.

Debido a que se elige el próximo vértice a colorear en base al grado de saturación, tomamos muchas ideas del algoritmo de Brelaz para la implementación de este algoritmo. De hecho, lo que hace la función Forward es, básicamente, ir coloreando el grafo en base al grado de saturación, siempre asegurandose de no repetir soluciones o explorar varias veces una misma rama.

Durante la implementación del algoritmo de Brelaz se decidió que en las deciciones de diseño se sopesaría más la eficiencia en tiempo que en espacio, ya que el algoritmo no consume mucha memoria per se y se buscaba una implementación rápida.

Esta desición generó muchos arreglos y estructuras de datos de control para guardar la información necesaria para calcular y actualizar los grados de saturación rápidamente.

Como ya se mencionó, debido a la heurística de elección del próximo vértice se tomaron muchas ideas de la implementación de Brelaz permitieron hacer una función de forward muy rápida y concisa. Sin

embargo, la cantidad de estructuras de datos que se usaron para llevar un control de la saturación de los vértices se volvieron una carga bastante grande a la hora de usarlas en una implementación por backtracking en vez de greedy, ya que así como se actualizan durante el Forward, se deben actualizar durante el backwards, lo cual implica que se gasta tiempo actualizando las estructuras durante cada uno de los backtracks.

Para representar el conjunto FC se añadieron estructuras a los vértices en los arreglos de adyacencias. La estructura Graph ahora tiene un arreglo de enteros que representa en forma de conjunto vectorial la pertenencia de un color al conjunto de colores válidos para un vértice, se utiliza un entero para indicar cual es el color más grande que se puede usar. Durante el Forward estos se sobreescriben, ya que la información que tuvieran previamente no es representativa de nada ya que se está explorando una nueva rama. Hay que agregar que durante la generación del FC se aplica look-ahead para tratar de reducir su cardinalidad, tambien se utiliza el procedimiento de etiquetado propuesto por Peemoller para el look-ahead.

Para representar el conjunto CP se usó el procedimiento de labeling de Peemoller. Se agrego a la estructura Graph un atributo Label que indica si el nodo está etiquetado o no y cual es su profundidad en la rama explorada actualmente. Con esto se puede buscar fácilmente el etiquetado de menor rango.

3.1.4. Bugs encontrados

La fase de debuggin fue bastante larga ya que se cometió el error de diseño de hacer dos funciones Backwards y Forwads que compartiesen datos muy delicados para la obtención de resultados. Ambas funciones propiciaban efectos de borde a estos datos compartidos, y cualquier error cometido en la modificación de estos datos hacía que el algoritmo simplemente nunca llegara a una solución o fallara en el intento. Luego de solucionar todos estos problemas de dependencia de datos, se arreglaron algunos “leaks” de memoria (memoria no liberada).

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 proyecto2-Fundaro-Jaber.tar.gz
```

A continuación se crea una carpeta llamada proyecto2 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.// Si desea compilar el algoritmo DSATUR+ haga el siguiente comando:

```
make mas
```

Si desea compilar el algoritmo DSATUR* haga el siguiente comando:

```
make estrella
```

En caso de querer borrar los archivos generados por la compilación proceda a 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 el comando que debe ejecutar es:

```
./script.sh <resultado>
```

Si desea ejecutar solo aquellas instancias donde las cotas generadas por DSATUR no son iguales, ejecute el comando:

```
./script.sh -c <resultado>
```

¡resultado¿en este caso es el nombre del archivo donde se guardaran 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 < instancias/instancia-particular
```

4.6. ¿Cómo interpretar resultados?

Cada instancia arroja un resultado con la siguiente estructura:

```
1 ---- 20-0,7-1.col -----
2 Resultados de Brelaz+interchange
3 Cota superior = 9
4 Cota inferior = 8
5 ------------------
6 Enumeración implícita
 Mejor Coloración Actual --> Cota_superior - 1
. ....
. ....
. Mejor Coloración Actual --> Cota_inferior
7 Numero cromatico: 9
8 Backtracks: 3
9 Tiempo en segundos de ejecución del programa: 0.0262
10 Vertice --> Color
   1 --> 1
   2 --> 6
   3 --> 2
   4 --> 3
   5 --> 3
   6 --> 7
   7 --> 4
   8 --> 8
   9 --> 2
   10 --> 5
   11 --> 8
   12 --> 9
   13 --> 6
   14 --> 2
   15 --> 5
   16 --> 7
   17 --> 1
   18 --> 4
   19 --> 4
   20 --> 8
```

Explicación de cada línea:

1. Nombre de la instancia, el primer número indica 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. (Enumeración Implícita)

6. A continuación líneas que van informando del progreso del algoritmo de backtracking.
7. Número cromático encontrado por enumeración implícita.
8. Cantidad de backtracks realizados por el algoritmo.
9. Tiempo en segundos según función provista en el proyecto.
10. A partir de la línea 10 tenemos la coloración encontrada.

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

La aplicación en estos momentos está totalmente funcional. Las tablas reflejan que el algoritmo DSATUR+ es mucho más efectivo que DSATUR* debido a la heurística utilizada para elegir el próximo color a utilizar.

6. Otros

6.1. Demostración formal de Enumeración Implícita

Dado que es un algoritmo de backtracking, se empezará por definir el grafo implícito.

El grafo implícito es el todas las posibles particiones de nodos del grafo, definimos el grafo implícito como sigue:

- Un nodo válido es aquel que contiene una partición de los vértices del grafo.
- Un hijo válido es aquel donde se ha agregado un vértice del grafo a uno de los conjuntos de la partición, siempre y cuando ese vértice no halla sido agregado antes a ninguna de las particiones de esa rama del árbol.
- Un hijo válido también puede ser una partición a la cual se le ha agregado un conjunto un vértice del grafo, siempre y cuando ese vértice no halla sido agregado antes a ninguna de las particiones de esa rama del árbol.

Una partición puede ser interpretada como una coloración (no necesariamente válida) donde cada conjunto de la partición representa un color único y donde el hecho de que un nodo pertenezca a un conjunto se interpreta como que ese nodo está coloreado del color que representa la partición. Por lo tanto:

- Dado que el árbol contiene todas las particiones posibles, toda coloracion válida existe en alguna hoja del árbol.

- Dado que cualquier solución óptima es una coloración válida, ésta debe estar en el árbol.

Se establece una cota superior buscando una coloración válida.
Se establece una cota inferior buscando una clique en el grafo.
Cuando se genera el FC, se excluyen los colores que generaían una coloración inválida, también se excluyen colores que generarían una coloración con máyor o igual número de colores la cota superior.
Existen dos formas de elegir el próximo color:

- Eligiendo el color que más se ha usado en la coloración parcial hasta ahora generada.
- Eligiendo el color que menos se ha usado en la coloración parcial hasta ahora generada. Siempre asegurando que se elija introducir un color nuevo de último.

Cada vez que se encuentra una coloración completa, se actualiza la cota superior al número de colores usados por esa nueva coloración.
El CP contiene sólo nodos coloreados adyacentes al nodo desde donde se hace el backtrack. Más aún, sólo tiene aquellos adyacentes al nodo actual que tienen mínimo rango entre los adyacentes de su color. Por lo tanto se tiene que:

- Si se usa la primera estrategia para elegir el próximo color, introducir un color nuevo siempre sera la última opción, ya que siempre será el que tiene menos popularidad. Si se utiliza la segunda estrategia, siempre se eligirá un nuevo color de último ya que es parte de la definición del criterio.

- Si se hace backtrack porque el FC es igual a vacío cambiar el color de un nodo no adyacente al nodo actual no cambiará su FC.

- Si se hace backtrack porque se llegó a una coloración completa, se desea una coloración con menos colores, por lo que se debe cambiar el color del primer nodo (rango mínimo) con el máximo color usado. El nodo con el máximo color usado debe tener un FC=¡maximo_color

  (debido a la forma en la que se eligen los colores), por el punto anterior, uno de sus vecinos coloreados debe cambiar.

- Dado que se incluyen todos los nodos que podrían inducir un cambio en el color del nodo del backtrack, recolorear cualquier otro nodo llevaría a un FC=vacío, por lo que no se excluyen soluciónes válidas en la construcción del CP.

- Dado que cuando se genera el FC, solo se incluyen colores que generan coloraciones válidas menores a la cota superior, no se excluyen soluciónes óptimas en la construcción de FC.

- No se excluyen soluciones óptimas durante la exploración del grafo, y ya que se exploran todas las soluciones no podadas, se debe explorar una solución óptima necesariamente.

El algoritmo se detiene cuando encuentra una coloración con tantos colores como la cota inferior. Esto indica que la coloración debe ser optimal, ya que no pueden existir coloraciones con menor número de colores que la cota inferior.

Si el algoritmo nunca encuentra una coloración con tantos colores como la cota inferior, devolverá la mejor coloración encontrada. Dado que el algoritmo devuelve la mejor solución encontrada, y dado que el algoritmo explora una solución óptima, la mejor solución encontrada debe ser la óptima encontrada.

6.2. Tabla de resultados

7. Conclusiones y recomendaciones

El algoritmo propuesto de backtracking iterativo resulta ser útil para resolver la mayoría de las instancias donde Brelaz+Interchange tardan más del tiempo esperado. Sin embargo, cuando las densidades son muy altas el algoritmo tarda mucho tiempo en converger, y cuando decimos mucho tiempo nos referimos a que excede el tiempo límite de 5 minutos. Es necesario entonces agregar nuevas optimizaciones al algoritmo para así conseguir tiempos más bajos de solución.

Como muestran los resultados la técnica del labeling no logró solucionar todas las instancias propuestas. En muchas, el número de backtracks era tan alto que el algoritmo nunca logró converger. Sin embargo las instancias que se corrieron en el proyecto pasado se lograron resolver con la coloración óptima.

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
- Marek Kubale Boguslaw Jackowski. “A Generalized Implicit Enumeration Algorithm for Graph Coloring”. Communications of the ACM April 1985 Volume 28, Number 4.
- Jurgen Peemoller. “A correction to Brelaz’s modification of Brown’s coloring algorithm”. Communications of the ACM, April 1983 Volume 26 Number 8.