Coloreo de grafos

Estructuras de Datos y Algoritmos

Informe de desarrollo

Información sobre el desarrollo
del Trabajo Práctico Especial

Jorge Ezequiel Scaruli
Matías Ezequiel Colotto

ITBA - 2010

Índice

1. Introducción

Se propuso como Trabajo Práctico Especial el desarrollo de una aplicación de coloreo mínimo de grafos. El objetivo del presente informe es exponer la manera en que se obtuvieron soluciones exactas y aproximadas para este problema, incluyendo los algoritmos creados y su orden de complejidad.

Asimismo, en un principio se mencionan problemas reales que pueden modelarse como grafos y ser resueltos mediante un algoritmo de coloreo de grafos, para dar una posible utilidad a los algoritmos luego presentados.

Finalmente, se realiza una comparación entre los tiempos que tarda cada algoritmo en colorear grafos.

2. Problemas que se pueden solucionar mediante coloreo de grafos

A continuación se detallan algunos problemas que podrían ser solucionados mediante coloreo de grafos.

2.1. Coloreo de mapas

La manera en que se colorean los mapas físicos de dos dimensiones (aquellos que muestran la división entre países/estados) es un ejemplo en el cual pueden aplicarse algoritmos de coloreo de grafos. En este problema, se exige que dos países/estados limítrofes se coloreen con un color distinto.

Para simplificar, se considera el caso de un mapamundi, en el cual se observa la división entre países. Aquí, el problema se puede modelar mediante un grafo de la siguiente manera: cada nodo del grafo representa un país, y dos nodos son adyacentes si representan países limítrofes.

Como observación, cabe destacar que los algoritmos aquí presentados no solucionarían un caso como el de un mapamundi, pues su representación mediante un grafo constituiría uno no conexo (pues hay países que constituyen islas), pero bastaría con agregarles el soporte para grafos no conexos.

2.2. Resolución de un sudoku

La resolución de un sudoku (juego de tablero tradicional) también puede modelarse mediante el coloreo de grafos.

Para ilustrarlo, se supone un sudoku simple, de 9×9 celdas, a fines prácticos. Sin embargo, puede generalizarse para otras dimensiones $N \times N$ del tablero, donde N es una potencia perfecta.

El grafo que modelaría a un sudoku se constituiría de la siguiente manera: cada nodo representa una celda, y dos nodos son adyacentes si representan nodos que:

- Están en la misma fila.
- Están en la misma columna.
- Están en la misma región de 3×3 celdas.

A cada valor posible para una celda $(1, 2, ..., 9)$ se le asigna un color.

Los nodos que representan celdas que ya tienen un valor inicial asignado se colorean con el color que representa a dicho valor, y se busca un 9-coloreo del grafo teniendo en cuenta estos nodos ya coloreados.

Una vez que se encontró un coloreo, se llena cada celda con el valor representado por el color que tiene el nodo correspondiente en el grafo, con lo que se obtiene una solución al juego.

2.3. Distribución de horarios en una universidad

Otro problema que puede representarse mediante grafos y cuya solución se modela mediante algoritmos de coloreo es la distribución de horarios para el dictado de materias en una universidad.

Supóngase que no es deseable que un profesor o a un alumno esté en la situación de tener que dictar o presenciar dos o más materias en el mismo horario. Esta situación puede modelarse con un grafo, cuyos nodos representan las materias dictadas en un cuatrimestre de la universidad. Dos nodos son adyacentes si un alumno cursa esas dos materias o un profesor las dicta.

Para este grafo, puede encontrarse un coloreo propio mínimo. Una vez encontrado, pueden armarse grupos de nodos del mismo color. Estos nodos representan las materias que pueden ser dictadas en el mismo horario, pues, al no ser adyacentes sus nodos, no hay ningún profesor o alumno que las dicte o presencie durante el mismo cuatrimestre.

3. Estructuras

Para representar un grafo, se utilizaron las estructuras `GraphAdjList` y `Graph` proporcionadas por la cátedra, con algunos cambios[1].

La clase abstracta `GraphAdjList<T,S>` se utiliza para representar un grafo, incluyendo sus nodos y aristas. Un nodo se representa con la clase `Node`, que posee una llave (`key`) de tipo `T` para acceder al grafo eficientemente mediante un `HashMap` y un `info` de tipo `S`, que es la información relacionada con ese nodo (en este caso, el color). Además, posee una lista de aristas que indica quiénes son sus nodos adyacentes. Los nodos se almacenan en una lista interna de la clase, y además existe un `HashMap` que mapea elementos del tipo `T` con nodos. Esto se utiliza para que el usuario pueda acceder a un nodo mediante una clave. Este `HashMap`, que es utilizado por todos los algoritmos al acceder al valor de un nodo, posee complejidad $O(n)$ para la consulta (donde n es la cantidad de vértices del grafo), ya que en el peor caso, el *hash* asigna a todos los nodos al mismo *bucket*: sin embargo, el acceso en realidad es muy rápido, como si fuera $O(1)$. Por lo tanto, se consideró esta última opción, la de acceso con $O(1)$, para el cálculo del orden de complejidad de los algoritmos.

La clase concreta `Graph<T,S>`, que extiende a `GraphAdjList<T,S>`, representa un grafo no dirigido, y posee operaciones correspondientes a esta estructura de datos.

Además de éstas, se diseñó la clase `ColoringGraph`[2], que usa objetos de la clase `ColoringKey` como claves, para un acceso más eficiente. En principio, se pensó en utilizar el nombre de un nodo como clave, que es un objeto de la clase `String`, dado que dos nodos nunca tienen el mismo nombre. Sin embargo, finalmente se optó por descartar esta alternativa debido a la complejidad, para un `String`, de los métodos `hashCode()` y `equals()` utilizados para el acceso mediante un `HashMap`. `ColoringGraph` es la clase que utilizan los algoritmos de coloreo.

[1] Si se dispone del código fuente, se recomienda ver el paquete **model.graph**
[2] Si se dispone del código fuente, se recomienda ver el paquete **model.algorithms.graph**

4. Algoritmos

En esta sección, se muestran los algoritmos que se implementaron para realizar coloreos propios mínimos de grafos.

No se encuentra en esta sección una descripción exacta de los pasos que sigue cada uno de los algoritmos, ya que se consideró que incluir descripciones de ese tipo sería demasiado extenso y haría que el informe sea difícil de seguir. Se sugiere ver el código fuente si se desea profundizar en ellos.

4.1. Algoritmo exacto

El algoritmo exacto se basa en el coloreo secuencial de vértices de un grafo, el cual tiene la siguiente forma:

```
f: {v_1, v_2, ... v_n} -> {color_1, color_2, ... , color_n}

FOR (i = 1 TO n) {
        f(v_i) = color más pequeño no utilizado en los vecinos de v_i;
}

RETURN f;
```

Para lograr un coloreo mínimo, este algoritmo debe aplicarse a cada posible permutación de los vértices. La demostración de que haciendo esto se llega a un coloreo mínimo se encuentra en el **Anexo**.

El algoritmo tendría entonces la siguiente forma, a grandes rasgos:

```
f: {v_1, v_2, ... v_n} -> {color_1, color_2, ... , color_n}

permutación = (v_1, v_2, ..., v_n);
coloreoMínimo;

WHILE(queden permutaciones no utilizadas) {
        FOR (i = 1 TO n) {
                f(v_i) = color más pequeño no utilizado en los vecinos de v_i;
        }

        if(f < coloreoMínimo) {
                coloreoMínimo = f;
        }

        permutación = siguientePermutación();
}

RETURN coloreoMínimo;
```

Este algoritmo fue implementado utilizando una estrategia de *backtracking*, en donde las permutaciones y el coloreo se hacen al mismo tiempo. En la implementación, además, se hace una poda si al pintar un nodo ya se sabe que, sin importar en que orden se pinten los restantes, no se llegará a un coloreo que use menos colores que el mejor encontrado hasta el momento. El orden de complejidad

temporal del mismo es $O(n!)$ donde n es la cantidad de vértices de la entrada.

Se propuso mejorar este algoritmo. Para ello, se pensó en que si se encontraba un clique (subgrafo K_m del grafo de entrada) y se pintaban todos sus vértices inicialmente con un color distinto, procediendo luego con los restantes como con el algoritmo anterior, aunque no reduciría el orden de complejidad del algoritmo, sí bajaría el tiempo que tardaría. El clique no necesariamente debe ser el máximo: sólo es una herramienta para lograr resultados más rápidamente.

El algoritmo usado para encontrar un clique es básicamente como sigue:

```
cliqueMaximo = {};
cliqueActual = {};

agregarAlClique(v) {
        IF(v es adyacente a todos los nodos de cliqueActual) {
                Agregar v a cliqueActual;
                IF(tamaño de cliqueActual > tamaño de cliqueMaximo) {
                        cliqueMaximo = cliqueActual;
                }
                FOREACH(w: nodo adyacente a v que no pertenezca a cliqueActual) {
                        agregarAlClique(w);
                        IF(cliqueMaximo es satisfactorio) {
                                RETURN;
                        }
                }
                Quitar v de cliqueActual;
        }
}

FOR(i = 1 TO n) {
        agregarAlClique(v_i);
        IF(cliqueMaximo es satisfactorio) {
                RETURN cliqueMaximo;
        }
}

RETURN cliqueMaximo;
```

Este algoritmo es exacto y tiene una complejidad temporal de $O(n!)$, pero gracias a la poda que se realiza en el mismo tarda mucho menos tiempo en correr que el algoritmo de coloreo exacto presentado anteriormente.

Con lo cual, la idea del algoritmo verdaderamente usado termina siendo:

```
cliqueMaximo = hallarCliqueMaximo();
m = tamaño(cliqueMaximo);

V = vértices del grafo que no están en cliqueMaximo;

Pintar los vértices del clique todos de distinto color;
```

```
f: V -> {color_1, color_2, ... , color_n};

coloreoMínimo;

permutación = una permutación de V, (v_1, ..., v_(n-m));

WHILE(queden permutaciones no utilizadas) {
        FOR (i = 1 TO n - m) {
                f(v_i) = color más pequeño no utilizado en los vecinos de v_i;
        }

        if(f < coloreoMínimo) {
                coloreoMínimo = f;
        }

        permutación = siguientePermutación();
}

RETURN coloreoMínimo;
```

En el **Anexo** se puede ver la demostración de que el coloreo propio producido por este algoritmo es mínimo.

El algoritmo, en la forma presentada, no es óptimo, con lo cual se implementó, al igual que el primero, usando *backtracking* y empleando poda. Se decidió también que un clique sea satisfactorio si contiene al menos la mitad de los vértices del grafo, por las siguientes razones:

1. Si el grafo tiene pocas aristas, el algoritmo que busca el clique no demorará mucho tiempo, con lo cual no hay problema en dejarlo correr.
2. Si el grafo tiene muchas aristas, es probable que haya un clique que contenga más de la mitad de los vértices del grafo, y es probable también que ese clique sea hallado rápidamente.

Los resultados de este algoritmo fueron satisfactorios y muy superiores al algoritmo anterior. Los grafos K_n se resolvieron muy rápidamente puesto que se los detectó como clique. Como mínimo, además, el algoritmo encuentra un K_2 al buscar un clique (puesto que el grafo es conexo) con lo que grafos como el C_{11} tardaron 100 veces menos tiempo en colorearse que con el algoritmo original.

La complejidad temporal de este algoritmo es $O(n!)$ donde n es la cantidad de vértices del grafo, pues en el peor caso (y suponiendo que un clique nunca se considerara satisfactorio) el algoritmo que busca el clique máximo recorrería todas las permutaciones de los vértices.

La complejidad espacial de este algoritmo es $O(n^2)$ donde n es la cantidad de vértices del grafo, puesto que por cada llamado recursivo a la función que colorea se guarda el conjunto de nodos adyacentes al mismo. Se observó que $O(n^2)$ es más que aceptable, pues debido a la gran complejidad temporal del algoritmo, se consideró que éste no sería usado para grafos muy grandes. Notar que si se pide el árbol de búsqueda del algoritmo, la complejidad espacial es mucho mayor: $O(n!)$ puesto que por cada paso se crea un vértice en dicho árbol.

El árbol de búsqueda generado por este algoritmo, si se es solicitado, no contiene el árbol de búsqueda del clique máximo, pues se consideró que resultaba difícil de seguir. En vez de eso, se muestran directamente los nodos del clique máximo pintados.

4.2. Algoritmo *greedy*

Se probaron muchos algoritmos diferentes que resuelven el problema de coloreo de grafos de manera aproximada, utilizando una estrategia *greedy*.

Las variantes que se utilizaron fueron:

- **Utilizando una estrategia DFS:** Se realiza un recorrido DFS del grafo, y conforme a este recorrido se va coloreando cada nodo con el color mínimo posible, dependiendo este valor de los colores de los nodos adyacentes. El orden de complejidad temporal de este algoritmo es $O(N^2)$, debido a que, para cada nodo del recorrido, debe recorrer todos los adyacentes para asegurar que el color mínimo elegido no sea el mismo de alguno de sus adyacentes. El orden de complejidad espacial es $O(N^2)$, ya que en cada llamado recursivo a la función que colorea un nodo, se almacena una lista con los nodos adyacentes al mismo.

- **Utilizando una estrategia BFS:** La estrategia es similar al ítem anterior, pero realizando un recorrido BFS. El orden de complejidad temporal es $O(N^2)$. El orden de complejidad espacial es $O(N)$, debido a que en cada iteración se construye una lista de nodos adyacentes al que se está por pintar, pero luego de la misma el *garbage collector* se encarga de liberar la memoria utilizada para la misma.

- **Utilizando el grado máximo de un vértice:** La estrategia utilizada en este caso se basó en colorear vértices con grado máximo. El funcionamiento del algoritmo consiste en lo siguiente:

  - En primera instancia, tomar un vértice aleatorio del grafo y colocarlo en un conjunto de nodos coloreados. Luego, en cada una de $N - 1$ iteraciones, donde N es la cantidad de vértices del grafo:
    - Tomar un vértice q al azar de los que ya fueron coloreados, que tenga vértices adyacentes sin colorear.
    - De todos los colores con los que se pueden colorear los adyacentes a q que no están coloreados, tomar el mínimo.
    - De todos los nodos adyacentes a q que se pueden colorear con dicho color mínimo, tomar el de grado máximo.
    - Colorear el vértice tomado en el paso anterior con el color mínimo.

  Esta estrategia se utilizó pensando que, al colorear un vértice de grado máximo con el mínimo color posible, los adyacentes a éste van a ser coloreados con un color no muy grande. De esta manera, es más probable que el algoritmo que utiliza este método utilice una cantidad de colores bastante aceptable.
  Este algoritmo no es determinista pues elige nodos al azar, pero suponiendo que esa elección es de orden $O(1)$, el orden de complejidad temporal de este algoritmo es $O(N^3)$, pues en cada una de las $N - 1$ iteraciones toma un nodo, recorre todos sus adyacentes y, para cada uno de ellos, determina el color mínimo con el que se puede colorear (estas dos últimas operaciones tienen orden temporal $O(N)$). El orden de complejidad espacial es $O(N)$, que corresponde al espacio necesario para guardar los nodos adyacentes al que se está recorriendo en un determinado momento del algoritmo.

- **Iterando por colores:** Esta estrategia corresponde a una iteración por los posibles colores con que se pueden pintar los nodos, en lugar de un recorrido por los vértices. Es decir: se comienza

por el primer color y se colorean todos los nodos posibles con ese; luego, se realiza lo mismo para el segundo; así sucesivamente, hasta que el grafo esté coloreado en su totalidad. El orden de complejidad temporal de este algoritmo es $O(N^3)$, y el orden de complejidad espacial es $O(N)$, ya que se construye una lista de nodos adyacentes al que se está por colorear, además de un stack para almacenar los elementos que deben ser eliminados luego de cada iteración.

Luego de varios testeos, se conluyó que el algoritmo a ser utilizado como *greedy* para la implementación final sea el que utiliza la estrategia DFS. Esta elección fue en base a que, considerando que los algoritmos *greedy* no tienden a encontrar un coloreo propio con una cantidad de colores cercana al mínimo, se priorizó como criterio el tiempo que tardan en realizar un coloreo aproximado. Conceptualmente, por tener $O(N^2)$, los candidatos a ser elegidos como algoritmos *greedy* principales fueron el DFS y el BFS. Sin embargo, pruebas empíricas hicieron que la elección se inclinara por el DFS[3].

A continación se muestra una tabla que muestra algunos testeos hechos, que determinaron la elección del método DFS como algoritmo *greedy*. Notar que $R_{v,e}$ significa un grafo generado aleatoriamente con v vértices y e aristas.

	DFS		**BFS**		**Grado máximo**		**Por color**	
Prueba	Tiempo	Colores	Tiempo	Colores	Tiempo	Colores	Tiempo	Colores
100 grafos $R_{200,500}$	47	513	35	528	137	513	55	544
100 grafos $R_{200,12000}$	665	4349	665	4339	19966	4151	4020	4369
100 grafos $H_{35,160}$	163	2400	159	2600	1445	2538	590	2600
10 grafos K_{100}	31	1000	32	1000	662	1000	455	1000
10 grafos K_{200}	113	2000	113	2000	5095	2000	3370	2000

Basándose en esta tabla, se puede observar que el algoritmo de grado máximo produce resultados mucho mejores para algunos grafos, pero que para otros no son tan buenos, y en ambos casos, esto ocurre a expensas de un mayor tiempo de ejecución. Se ve también que el algoritmo que itera por colores no produjo resultados buenos en ningún caso, ni en cantidad de colores ni en tiempo.

El BFS tiene una mejor performance que el DFS para grafos con muchas aristas, pero para grafos con una cantidad baja o media de aristas, el DFS es superior, como puede verse en la tabla. Es por ello que se eligió a éste como algoritmo *greedy* a utilizar.

4.3. Algoritmo que utiliza la heurística *Tabu Search*

El algoritmo que implementa *Tabu Search* obtiene una solución inicial basada en el algoritmo *greedy* utilizado. Se podría haber comenzado con una solución inicial peor pero más rápida de calcular (donde cada nodo tiene un color distinto) pero se descartó pues se necesitaban más iteraciones para llegar a un resultado aceptable. El algoritmo *greedy*, además, tiene complejidad temporal despreciable frente a la del *Tabu Search*, con lo que no influye en la misma.

Para implementar la heurística *Tabu Search*, se utilizaron las siguientes variables:

- Una memoria (arreglo) de N enteros, donde N es la cantidad de vértices del grafo de entrada. En principio está inicializada en 0.
- Una variable con el color máximo del grafo, y un contador con la cantidad de vértices actualmente coloreados con ese color. Cuando se cambia el color de un vértice, este contador se actualiza; de llegar a 0, se recalcula el color máximo y la cantidad de vértices pintados con dicho color, ya que

[3]Si se dispone del código fuente, se recomienda ver el paquete **model.algorithms.deprecated**, donde se encuentran las implementaciones de los algoritmos no utilizados.

significa que el algoritmo ha logrado reducir en al menos 1 la cantidad de colores utilizados en el grafo.

- Una variable *maxColorTabu* que indica si el color máximo puede ser utilizado para colorear un vértice o no. Ésta permite que, si un vértice v fue coloreado con el color máximo M utilizado hasta ahora, éste color, además de v, sea declarado como "tabú" por una cantidad dada de iteraciones. Por lo tanto, en las próximas iteraciones (hasta que M deje de ser "tabú") los vértices seleccionados por el algoritmo serán coloreados con colores menores a M. De esta manera, será probable que, al dejar de ser "tabú" el vértice v, éste pueda ser coloreado con un color menor a M. El objetivo de este manejo es minimizar la cantidad de colores utilizada por el algoritmo.

El algoritmo realiza una cantidad de iteraciones igual al mínimo entre 5000 y N^2. Esto es así pues para grafos pequeños, N^2 es un numero muy chico y se consideró que se pueden realizar más iteraciones para conseguir un coloreo aún mejor.

En cada iteración, se selecciona una posición al azar de la memoria. Si la memoria en dicha posición tiene un valor superior a la cantidad de iteraciones completadas hasta el momento, entonces se vuelve a calcular una posición al azar de la misma, y así sucesivamente hasta encontrar alguna que no cumpla dicha condición. Debido a esto, la penalización por "tabú" queda restringida a un entero que debe ser menor o igual a N.

Luego de seleccionar una posición al azar válida, se obtiene el nodo correspondiente a dicha posición. Se crea un conjunto de colores válidos, que en principio son desde el 0 hasta el $M-1$, donde M es el color máximo utilizado hasta el momento, y se quitan de dicho conjunto tanto el color actual de ese vértice como el color de cada uno de sus vecinos. Aquí entra en juego la variable *maxColorTabu*. Si ésta tiene un valor mayor a la cantidad actual de iteraciones, significa que el color máximo es "tabu" en esta iteración. De no ser así, M también es agregado en el conjunto de colores válidos, previamente a remover los colores de los vecinos y el propio.

Se selecciona luego del conjunto restante, aleatoriamente, un color nuevo. Esto asegura dos cosas:

1. El nuevo coloreo obtenido será propio, y, por tanto, una solución válida.
2. El nuevo coloreo obtenido nunca tendrá una cantidad de colores cuyo color máximo sea mayor al anterior. Esto hace que el algoritmo pueda decrementar la cantidad de colores utilizados más rápidamente.

Si en el conjunto no había quedado ningún color disponible, el vértice no se colorea. En el caso en que sí, se colorea con el color elegido. En ambos casos, el vértice es declarado como "tabú"; es decir, la posición correspondiente a este vértice en la memoria tabú se setea en cantIteracionesActual + $N/2$.

Para que el manejo del color máximo siga siendo válido, se chequea si el vértice coloreado tenía, antes de ser coloreado, el color máximo. Si se da este caso, entonces se debe verificar la variable de la cantidad de vértices coloreados con el mismo, y hacer el recálculo mencionado al principio si es necesario. Si el vértice coloreado tenía un color distinto al máximo y se lo pinta como máximo, entonces se declara al color máximo como "tabú". Esto consiste en un proceso similar a declarar un vértice como tabú, excepto que el color máximo se deja "tabú" por más tiempo que un vértice (en total, $2N$ iteraciones).

El algoritmo no es determinístico pues los nodos y colores se eligen al azar entre los posibles. Sin embargo, si se considera que todos los procedimientos dejados al azar son de $O(1)$, la complejidad temporal del algoritmo es $O(N^3 \log N)$ pues en cada una de las iteraciones, para cada vecino, quita

su color del conjunto de colores. En el peor caso, el nodo tiene N vecinos, con lo cual se hacen N borrados: como el conjunto está implementado como un `TreeSet`, se hace por cada iteración algo que es $O(N \log N)$, con lo cual, al hacerse N^2 iteraciones, hace que el algoritmo posea la complejidad temporal mencionada anteriormente.

La complejidad espacial de este algoritmo es $O(N)$, debido a la memoria de "tabú" utilizada.

4.4. Comparaciones entre algoritmos

Para evaluar los algoritmos, se implementó toda una estructura de clases[4] dedicada especialmente a hacer testeos de rendimiento sobre los mismos. La misma permite trabajar con una cantidad arbitraria de algoritmos y con testeos de todo tipo. En general, se utilizaron grafos generados por computadora para probar los algoritmos, en vez de tomarlos de archivos aparte.

Los testeos más utilizados fueron:

- `KnTest(iter, n)`: prueba los algoritmos utilizados `iter` veces con un grafo K_n.
- `CnTest(iter, n)`: prueba los algoritmos utilizados `iter` veces con un grafo C_n.
- `RandomTest(iter, v, e)`: prueba los algoritmos utilizados con `iter` grafos distintos, generados aleatoriamente, con v vértices y e aristas.
- `HararyTest(iter, k, n)`: prueba los algoritmos utilizados `iter` veces con un grafo $H_{k,n}$ (grafo de Harary, k-conexo con n vértices).

A continuación se realiza una comparación de los tiempos (en ms) y cantidad de colores utilizados por los algoritmos. Notar que $R_{v,e}$ es un grafo aleatorio de v vértices y e aristas, y que $Q_{m,n}$ es un grafo *Queen's tour* de parámetros m y n.

	Exacto		**Greedy**		**Tabu Search**	
Prueba	Tiempo	Colores	Tiempo	Colores	Tiempo	Colores
100 grafos C_{11}	18669	300	1	300	452	300
100 grafos K_{10}	36	1000	6	1000	1065	1000
100 grafos $H_{9,16}$	959899	600	3	600	944	600
100 grafos $Q_{7,7}$	-	-	45	1000	1825	885
100 grafos $R_{13,20}$	2369	307	2	328	581	307
100 grafos $R_{15,70}$	15881	623	5	696	914	623
1 grafo $R_{3000,10000}$	-	-	68	7	10856	5
1 grafo $R_{3000,450000}$	-	-	108	71	332031	53

Los campos completados con **-** indican que no hubo prueba para ese algoritmo, debido a que el tiempo que lleva podría ser demasiado grande.

Puede verse, a partir de la comparación, que los mejores resultados los brinda siempre el exacto, pero que para grafos grandes, tarda demasiado tiempo. Puede verse también que el algoritmo *greedy* provee resultados muy rápidamente, incluso para grafos muy grandes, pero que el algoritmo *Tabu Search* siempre los mejora (salvo que sean un coloreo propio mínimo), aunque tardando más tiempo (que sin embargo, es una cantidad aceptable si se desean resultados buenos).

Se observa también que el algoritmo exacto es muy rápido si los grafos tienen muchas aristas, y más lento en caso contrario.

A su vez, se observa que la cantidad de aristas es un parámetro muy importante en la cantidad de tiempo que tardan los otros dos algoritmos (aunque en particular, se ve mucho más para el *Tabu Search*). Esto es así puesto que para cada iteración, se debe controlar el color de cada uno de los adyacentes al nodo actual: si hay más aristas, hay más adyacentes en promedio, con lo cual los algoritmos

[4] Si se dispone del código fuente, se recomienda ver el paquete **view.test**.

son más lentos.

Si bien no está presente en la tabla, es claro que el algoritmo *Tabu Search* produce mejores resultados con una cantidad mayor de iteraciones, pero que esos mejores resultados están acompañados por tiempos más grandes. Se ve, en base a los resultados de la tabla, que con la cantidad de iteraciones elegida los resultados son excelentes para grafos pequeños, y aceptables para grafos más grandes. De desear que dicho algoritmo sea más preciso (aunque a cuesta de mayor complejidad temporal) puede simplemente incrementarse la cantidad de iteraciones.

5. Otras clases útiles

La siguiente sección brinda una descripción de clases que fueron utilizadas en el desarrollo del Trabajo Práctico, y que pueden ser de interés (para conocer con más detalle la implementación de los algoritmos y de los testeos, o para su uso en otros desarrollos). Las mismas se encuentran en el paquete `view`.

5.1. Generación de grafos

En el paquete `view.input.graphfactory` pueden encontrarse clases para generar grafos automáticamente. Éstas son las siguientes:

- `AutoGraph`: clase abstracta que modela a las clases concretas que generan grafos.
- `KnGraph`: genera grafos completos. Para ello, se especifica un número de vértices en el constructor.
- `CnGraph`: genera ciclos. Se especifica también en su constructor el número de vértices.
- `RandomGraph`: genera grafos aleatorios. En su constructor se especifican el número de vértices y de aristas.
- `HararyGraph`: genera grafos de *Harary*[5] de parámetros k y n. En su constructor se indican dichos parámetros.
- `QueensTourGraph`: genera grafos *Queen's tour*[6] de parámetros m y n. En su constructor se indican dichos parámetros.

5.2. Creación de archivos de salida

En el paquete `view.output` se encuentran clases para generar archivos de salida correspondientes a grafos de entrada. Las mismas son las siguientes, e implementan la interface *GraphOutput*:

- `ArcsGraphOutput`: produce archivos con el formato especificado por la cátedra para representar grafos. Esto es, archivos en los que, para cada arista $\{v, w\}$, se escribe una línea
  `nombre_v, nombre_w`
  o
  `nombre_w, nombre_v`
  donde `nombre_v` es el resultado de aplicar el método `toString` al objeto que se usa como clave en el vértice v (`nombre_w` tiene el mismo significado, para el vértice w).
- `ColorsGraphOutput`: genera archivos de salida para grafos coloreados. Por cada vértice v del grafo cuya salida se desea obtener, se escribe una línea como la siguiente:
  `nombre_v = COLOR`
  `nombre_v` tiene el mismo significado que en el ítem anterior, y `COLOR` es el número que identifica al color con el que está coloreado v.
- `GraphvizGraphOutput`: genera un archivo que representa un grafo, con el formato utilizado por el programa *Graphviz*[7].

[5] Un grafo de *Harary* de parámetros k y n es un tal que posee n vértices y es k-conexo.

[6] Un grafo *Queen's tour* de parámetros m y n es aquél en el que cada vértice representa una celda en un tablero de ajedrez de $m \times n$, y dos vértices son adyacentes si representan dos celdas tal que una reina puede moverse de la primera a la segunda.

[7] Para más información sobre el formato de archivos utilizados por *Graphviz*, visitar http://www.graphviz.org/doc/info/lang.html.

- `GraphvizColorGraphOutput`: genera un archivo representando un grafo coloreado, con el formato utilizado por el programa *Graphviz*. Los nombres de los nodos están divididos en dos líneas: la primera es el número de color con el cual fue pintado, y la segunda, el identificador del nodo.

6. Conclusiones

Luego de la implementación de los algoritmos de coloreo de grafos, se pueden sacar ciertas conclusiones.

Enfocándose en el tema de la construcción de los algoritmos, se pueden mencionar los siguientes aspectos:

- En el algoritmo exacto, hacer un preprocesamiento inicial encontrando un K_n en principio resultó complicado, ya que no fue sencillo demostrar que el algoritmo secuencial de coloreo de vértices (utilizando las distintas permutaciones) puede funcionar si el grafo tiene vértices ya coloreados antes de aplicárselo. Sin embargo, finalmente esta poda inicial fue satisfactoria en términos de tiempo.
- Los algoritmos *greedy*, aunque su orden de complejidad pudo ser calculado previamente a su implementación, no dieron indicios de cuál podría ser mejor antes de verificarlo empíricamente. Fue por eso que se tuvieron que implementar todas las variantes que surgieron, para poder determinar el más apropiado.

Con respecto de los problemas que aparecieron al desarrollar el trabajo, se puede decir que el algoritmo que utiliza la heurística *Tabu Search* fue el que trajo más complicaciones. Por tratarse de una heurística y, además, ser un algoritmo no determinista, fue complicado encontrar ciertos criterios que impliquen una mayor eficiencia o un resultado más acertado. Fue por ello que ciertos valores, como la cantidad de iteraciones, se hicieron en base a testeos empíricos.

Finalmente, conceptualmente se concluye que es útil la implementación de estos algoritmos, ya que ciertos problemas del mundo real pueden llegar a modelarse como un problema de coloreo de un grafo, y dichos problemas pueden resolverse aplicando los algoritmos ideados.

7. Referencias

- Material didáctico provisto por la cátedra.
- Material didáctico de Matemática Discreta.
- *Wolfram MathWorld*: http://mathworld.wolfram.com/.

8. Anexo

8.1. Demostraciones de algoritmos

8.1.1. Demostración del primer algoritmo de coloreo exacto

Sea un grafo G conexo tal que $V(G) = \{v_1, \ldots, v_n\}$ ($\#V(G) = n$). Este grafo tiene un coloreo propio: en particular, tiene al menos un coloreo propio mínimo. Sea f, entonces, un coloreo propio mínimo de G, y supóngase que los colores de ese coloreo propio mínimo son $\{1, \ldots, m\}$ con $m \leq n$.

Sea la permutación de los vértices de G: $(w_1, \ldots, w_n)$ tal que si $i < j$ entonces $f(w_i) \leq f(w_j)$. Sea $p(v)$ el color con el que el algoritmo pinta al vértice v para la permutación dada. Se demostrará que $p(w_i) \leq f(w_i)\ \forall i$.

Si $i = 1$, entonces no hay vértices pintados en el grafo; entonces $p(w_1) = 1$ y $1 \leq f(w_1)$.

Hipótesis inductiva: si $i < k$ entonces $p(w_i) \leq f(w_i)$.
Tesis: si $i = k$ entonces $p(w_i) \leq f(w_i)$.

Demostración: sea w_s un vértice cualquiera adyacente a w_k. Entonces se cumple una de las siguientes:

1. $s < k$. Entonces por hipótesis inductiva, $p(w_s) \leq f(w_s)$. Como $s < k$, $f(w_s) \leq f(w_k)$; como además w_s es adyacente a w_k, entonces $f(w_s) < f(w_k)$. Por lo tanto $p(w_s) < f(w_k)$; en particular, $p(w_s) \neq f(w_k)$.
2. $s > k$. Entonces el algoritmo nunca ha pintado este vértice, por lo que no incide en $p(w_k)$.

Por lo tanto, el algoritmo siempre podrá pintar el vértice con el color $f(w_k)$: entonces, $p(w_k) \leq f(w_k)$.

Queda demostrado entonces que $p(w_i) \leq f(w_i) \forall i$. Por lo tanto, el algoritmo produce un coloreo propio mínimo con esta permutación.

8.1.2. Demostración del segundo algoritmo de coloreo exacto

Sea un grafo G conexo tal que $V(G) = \{v_1, \ldots, v_n\}$ ($\#V(G) = n$). Sea, además, un clique de G, con vértices $\{c_1, \ldots, c_r\}$ (existe pues K_1 siempre es subgrafo de G). Este grafo tiene un coloreo propio: en particular, tiene al menos un coloreo propio mínimo. Sea f, entonces, un coloreo propio mínimo de G, y supóngase que los colores de ese coloreo propio mínimo son $\{1, \ldots, m\}$ con $m \leq n$ y tal que $f(c_i) = i\ \forall i \mid 1 \leq i \leq r$.

Sea la permutación de los vértices de G que no están en el clique: $(w_{r+1}, \ldots, w_n)$ tal que si $i < j$ entonces $f(w_i) \leq f(w_j)$. Sea $p(v)$ el color con el que el algoritmo pinta al vértice v para la permutación dada: para los vértices del clique, $p(c_i) = i$ por la definición del algoritmo. Se demostrará que $p(w_i) \leq f(w_i)\ \forall i$.

Si $i = r + 1$, entonces sea v un vértice adjacente a w_{r+1}. Ocurre una de las siguientes:

1. v no pertenece al clique, entonces v no está pintado y no influye en $p(w_{r+1})$.
2. v pertenece al clique, entonces por la definición del algoritmo, $p(v) = f(v)$ y como son adyacentes, $f(v) \neq f(w_{r+1})$.

El algoritmo entonces puede pintar w_{r+1} con el color $f(w_{r+1})$, con lo que $p(w_{r+1}) \leq f(w_{r+1})$.

Hipótesis inductiva: si $i < k$ entonces $p(w_i) \leq f(w_i)$.
Tesis: si $i = k$ entonces $p(w_i) \leq f(w_i)$.

Demostración: sea w_s un vértice cualquiera adyacente a w_k. Si w_s pertenece al clique, entonces $p(w_s) = f(w_s)$; como w_s es adyacente a w_k, entonces $f(w_s) \neq f(w_k)$, y por lo tanto $p(w_s) \neq f(w_k)$. De lo contrario, se cumple una de las siguientes:

1. $s < k$ y w_s no pertenece al clique. Entonces por hipótesis inductiva, $p(w_s) \leq f(w_s)$. Como w_s es adyacente a w_k, entonces $f(w_s) \neq f(w_k)$, por lo tanto $p(w_s) < f(w_k)$ y, en particular, $p(w_s) \neq f(w_k)$.

2. $s > k$ y w_s no pertenece al clique. Entonces el algoritmo nunca ha pintado este vértice, por lo que no incide en $p(w_k)$.

Por lo tanto, el algoritmo siempre podrá pintar el vértice con el color $f(w_k)$: entonces, $p(w_k) \leq f(w_k)$.

Queda demostrado entonces que $p(w_i) \leq f(w_i)\ \forall i$. Por lo tanto, el algoritmo produce un coloreo propio mínimo con esta permutación.