Informe TPE EDA

Grupo 7

25 de mayo de 2010

Profesores:

- Garberoglio, Marcelo
- Gregoire, Andrés

Integrantes del grupo:

- Magnorsky, Alejandro
- Merchante, Mariano

Fecha de entrega: 28/05/2010

Índice

1. Problemas reales asociados al coloreo de un grafo

- Se tiene una tabla de invitados donde para cada uno, aparece con quienes no se lleva bien. El objetivo es averiguar cuál es la disposición en mesas de dichos invitados de tal forma que todos se sientan a gusto y se necesiten la menor cantidad de mesas posible.
- En un acuario se busca invertir la menor cantidad de dinero posible para tener en distintas peceras, animales acuáticos cuyo depredador no se encuentra en su misma pecera.

2. Algoritmos creados

2.1. Algoritmo perfecto

2.1.1. Fuerza Bruta

Este algoritmo se basa en comenzar coloreando un nodo de los n disponibles. Luego se elije colorear otro nodo de los $n-1$ restantes y así sucesivamente se prueba para todas las posibilidades. Es decir, habiendo elegido un nodo al principio, se prueba uno de los $n-1$ restantes, cuando termina de recorrer ese camino, prueba con otro nodo de los n-1 y así con todos. El coloreo se realiza de tal forma que se cumpla siempre que dado un nodo, ninguno de sus vecinos tiene su mismo color o está descoloreado.

A su vez, cuando llegó a una solución, guarda la cantidad de colores que se usaron para la misma, y cuando recorre otros caminos posibles, si requieren igual o mayor cantidad de colores, prueba con otro. Como usa la estrategia de backtracking, cada vez que termina de analizar qué sucedía si coloreaba un nodo, lo descolorea para permitir que luego sea coloreado nuevamente - tal vez, por un color distinto - habiendo llegado a él por medio de otro orden de elección de nodos.

Para disminuir la complejidad temporal se usaron algunas colecciones tales como quantColor, una lista de enteros que contiene la cantidad de nodos que usan el color i-esimo, y available que contiene todos los colores que se están usando hasta el momento. También se guardó en la variable usedColores, el número de colores usados en la mejor solución.

El orden de complejidad de este algoritmo es $n * n!$ donde n es la cantidad de vértices que tiene el grafo ya que al principio se tienen n posibilidades para colorear, elegida alguna, se recorren todos los demás nodos para determinar qué color se usa. Luego se hace lo mismo para $n-1$ y así hasta que queda un nodo, por lo que $n * (n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1) = n * n! => O(n * n!)$

El orden de complejidad espacial de este algoritmo es $n!$ ya que por cada llamada al método recursivo perfectColoring(), se crea un nodo que contiene el estado actual del vértice que se está coloreando y se agrega al árbol de estados. Cabe mencionar que se considera que no se realiza ninguna poda y el espacio que ocupan los nodos no se libera ya que forman parte del árbol que se devolverá en caso que el usuario así lo solicite.

2.1.2. Matriz de adyadencia invertida

El concepto del algoritmo es utilizar una matriz de adyacencia, con la diferencia de que cada bit esta invertido. Como los multigrafos y lazos se ignoran, esto permite representar la dependencia de vértices entre sí mediante ceros y unos. La matriz se la pensó como un conjunto de filas, donde cada fila es un string de bits de longitud n, donde n es la cantidad de vértices del grafo. El algoritmo utiliza los índices de un recorrido DFS para identificar cada vértice. Cuando se habla de un vértice, se está hablando realmente de su posición en el recorrido, y los conjuntos de vértices contienen solo índices. Inicialmente, cada fila representa al vértice i-ésimo segun el recorrido DFS (como la matriz de adyacencia original), pero puede contener un conjunto de vértices, que luego será útil. El algoritmo funciona resolviendo dos problemas: primero construye un conjunto con todas las posibles clases de equivalencia en el grafo representado por la matriz.

Básicamente, la primera parte del algoritmo consiste en encontrar todos los conjuntos de vértices que pueden tener el mismo color. Cada uno de estos conjuntos es una posible clase de equivalencia. Esto se logra mediante la sucesiva aplicacion del operador 'y' lógico entre cada string de bits. Si luego de aplicar dicha operación el resultado es inconsistente, entonces la ramificación se ignora. Definimos que un string de bits es consistente si para cada elemento del conjunto de vértices $\{v1, v2...vn\}$ existe un 1 en la posicion i-ésima del bit, donde i es la posición del vértice en una lista generada por el algoritmo DFS. Sin embargo, si el resultado es consistente, se agrega el conjunto de vértices que representa dicho resultado (recordemos que cada string de bits contiene un conjunto de veritces), y se intenta lo descrito previamente de forma recursiva.

Por una cuestión de optimización y organización, las posibles clases de equivalencias están ordenadas por grupos, donde cada grupo representa un vértice. De esta forma, el grupo i-ésimo contiene al vértice v_i en todas sus posibles clases de equivalencia. Esto no implica que ese mismo vértice no aparezca en otros grupos, pero simplifica la complejidad.

La segunda parte consiste en combinar todas las posibles clases de equivalencias y seleccionar al conjunto de clases más óptimo. Esto se logra juntando todas las posibles clases de equivalencia calculados en el paso anterior, con la condición de que la intersección entre dichas clases sea nula. Esto es análogo a generar todos los posibles coloreos del grafo y seleccionar el más óptimo. La forma para seleccionar al más óptimo es buscando aquel conjunto de clases de equivalencias que contenga la mínima cantidad de clases (colores), pero que la unión de todas las clases contenga a todos los vértices (asegurándose así que el grafo quede completamente coloreado).

Finalmente, luego de obtener al conjunto de clases de equivalencia más óptimo, se debe colorear al grafo. Debido a que cada clase contiene los índices de los vértices en un recorrido DFS, simplemente se utiliza el recorrido para identificar cada vértice y asignarle el color correspondiente según la clase de equivalencia a la cual pertenece.

El orden de complejidad temporal y espacial de este algoritmo parece ser $O((n!)!)$ en el peor caso (un grafo ciclo), donde n es la cantidad de vértices que tiene el grafo. Esto se debe a que primero debe calcular todas las posibles clases de equivalencia. Esta operación tiene una complejidad temporal de orden $n!$, y,

como cada clase se guarda en memoria, tiene el mismo orden de complejidad espacial. En el segundo paso del algoritmo, se utilizan los conjuntos del primer paso y se los combinan nuevamente, lo cual tiene una naturaleza factorial. Como además se guardan las combinaciones, tiene el mismo orden de complejidad temporal como espacial, y es $O((n!)!)$.

Debido a la complejidad espacial, ciertos grafos simplemente son imposibles de colorear con este algoritmo debido al tamaño del heap de Java. También es interesante notar que muchas veces la construcción de la matriz toma cierto tiempo, ya que la complejidad espacial es $O(n^2)$. Estas son grandes desventajas de la matriz de adyacencia invertida, y por su naturaleza no se pueden evitar.

2.1.3. Resultado final perfecto

Lo interesante de ambos algoritmos es que andan muy bien para ciertos casos. En particular, el algoritmo de fuerza bruta es bueno para grafos donde la poda es efectiva, como un C_n con n par, mientras que el algoritmo de matriz de adyacencia invertida es bueno para grafos con muchas aristas, ya que cuantas más aristas, menos ramificaciones hay (porque hay más dependencia entre vértices).

Debido a esto, se decidió utilizar ambos algoritmos para determinados casos: si el grafo es denso se utilizará la matriz de adyacencia invertida, y sino se utilizará fuerza bruta. Para esto, se debe definir dicha densidad:

La densidad de un grafo se define como $D = \frac{2|E|}{|V|\,(|V|-1)}$.

Ahora bien, si la densidad es menor a 0.5, entonces se utilizará fuerza bruta, y sino, se utilizará la matriz de adyacencia invertida. Ésto proporciona una considerable ventaja a la complejidad, ya que se eliminan los peores casos de ambos algoritmos.

2.2. Algoritmos de aproximación

2.2.1. Algoritmo greedy

Éste algoritmo consiste en seleccionar iterativamente un vértice y asignarle un color aleatorio de entre los posibles colores que puede tener. Si bien no siempre colorea óptimamente al grafo, resulta ser muy rápido y eficiente como solución inicial para ciertas heurísticas. Originalmente la selección del color se logra eligiendo al mínimo color posible; sin embargo, se decidió elegir un color aleatorio, ya que esto incrementa las posibilidades de dar con un orden de selección óptimo de colores.

El orden de complejidad espacial y temporal de este algoritmo es $O(n^2)$, ya que para cada vértice, el algoritmo busca todos sus adyacentes y genera conjuntos con dichos vértices.

2.2.2. Algoritmo de búsqueda tabú

Comienza por una solución inicial obtenido de un algoritmo greedy que toma el mínimo color disponible para colorear un nodo cumpliendo la condición de coloreo de un grafo y obviamente, amplia la cantidad de colores disponibles si es necesario. Luego, calcula las soluciones vecinas que consisten en modificar el color de un nodo; si esa modificación hizo que dos nodos adyacentes tengan el

mismo color, modifica el color de ese nodo también y así hasta que quede un coloreo correcto.

La evaluación se basa simplemente en la cantidad de colores que se usaron, si la de la solución vecina es menor, la reemplaza por la misma y chequea sus vecinos. Esto lo realiza una cantidad de veces determinada por la variable 'max tries'. Cabe destacar que si se elegió una solución donde se cambio el nodo i, ese nodo no volverá a elegirse para el método que cambia el color, hasta que pasen j iteraciones.

Para calcular el orden de complejidad de este algoritmo es necesario considerar el peor caso que es un K_n. Usar el método color() de Coloring para cada nodo de un K_n es de orden n^2, ya que para cada nodo recorre todos los demas para chequear su color y determinar cuál poner. El método changeColor tiene como peor caso hacer eso comenzando desde todos los nodos, es decir, $n \cdot n^2 = n^3$. Como es el que mayor orden tiene en todo el algoritmo, $O(n^3)$.

Por otro lado, su orden de complejidad espacial es n^2 porque dada una localSolution, cada vez que crea una solución vecina, la misma contiene una colección de estados. Como se crean n soluciones vecinas, el orden es $O(n^2)$.

3. Tablas de comparación

4. Conclusiones