MAR, práctica 2: Coloreado de grafos

Manuel Sánchez Pérez

Introducción

La siguiente práctica elabora una implementación de un algoritmo tipo *branch and bound* para el problema del coloreado de un grafo.
Este documento describe dicha implementación, sus fundamentos, decisiones de diseño, y finalmente conclusiones.

Problema del coloreado

El problema de optimización sobre el coloreado de grafos consiste en encontrar una forma de asignar un *"color"* a cada nodo de un grafo de forma que dos nodos adyacentes (*"vecinos"*), es decir unidos por una arista, no compartan el mismo color.

Existen múltiples algoritmos para abordar este problema, la práctica se centra en un algoritmo con esquema *branch and bound* con cota optimista y pesimista.

Algoritmo

El algoritmo consiste en explorar el espacio de soluciones del problema dado un grafo de entrada `G` sin colorear, de tamaño `n` (número de nodos). Para simplificar la notación, utilizaremos enteros `c >= 0` para denotar colores. `-1` representa *"no color"*.
Inicialmente se considera que el primero nodo, `G(0)`, tiene el primer color, `0`. Partiendo de ese estado, el árbol de exploración se expande de forma que en el nivel `k`-èsimo se intenta aplicar un color a `G(k)`. Nótese que el número de nodos que se expanden a partir de una solución parcial es variable, ya que depende del grafo y colores ya utilizados.

Definimos la estructura `TreeNode` que guardará todos los metadatos necesarios para representar una solución parcial parte del árbol de exploración:

```
struct TreeNode
{
    std::vector<int> solution; // n colors, one per node
    int optimistic = 0;
    int pessimistic = 0;
    int last_color = 0;
    std::size_t k = 0;
};
```

`TreeNode::solution` es un vector de `n` elementos que contiene el color asignado

a cada uno de los nodos en ese momento. Así la solución al problema es dicho vector para un nodo de nivel `k = n - 1`. Además se guardan los valores de las cotas optimistas y pesimistas para ese nodo así como el último color utilizado por dicha solución parcial. *Último no en tiempo, sino en colores utilizados en total: Si `solution` es `{0,-1,-1,2,-1,1,-1}`, `last_color` debe ser `2`.*

A la hora de expandir un nodo de nivel `k`, hay que tener en cuenta qué posibles colores puede recibir el nodo `G(k)` del grafo. En principio puede utilizar alguno de los ya utilizados, es decir un `c en [0,last_color]`, **siempre y cuando no tenga algún vecino con dicho color**. En ese caso, se aplica el color `last_color + 1`.

Cotas

La forma mas fácil de acotar este problema consiste en ponerte en el peor y el mejor caso. Como cota optimista para un nodo `y` de nivel `k` `optimista(y) -> y.last_color`. Es decir, supone que va a tener suerte y no va a utilizar mas colores. Y como cota pesimista `pesimista(y) -> n`.

Por supuesto dichas cotas no son muy realistas, resultando en un algoritmo que casi se podría considerar backtracking sin poda alguna, con una media de 500 millones de nodos abiertos para grafos de tamaño `n = 30`. *Ver apartado "conclusiones" a continuación para mas información*.

La solución presentada utiliza unas cotas algo mas elaboradas:

- **Cota Pesimista:** Para un nodo `y` de nivel `k` que ya ha utilizado `y.last_color` colores, suponemos que en el peor caso todos los nodos restantes estarán interconectados, y por tanto tendrá que utilizar un color para cada nodo. Es decir: `pesimista(y) -> y.last_color + n - y.k - 1`. No se puede podar con la cota pesimista ya que por la forma del problema hay que explorar la rama completa hasta el nivel `k = n - 1` para encontrar una solución.
- **Cota optimista:** En lugar de ser totalmente optimistas, examinamos el grupo de vecinos de `G(y.k)` para ver como estan interconecectados: Para el nodo `G(y.k)` extraemos sus vecinos directos (De *"primer nivel"*). Por cada uno de estos, se extrae a su vez el conjunto de sus vecinos (Vecinos *"de segundo nivel"* de `G(y.k)`) y se cuenta cuantos de estos últimos pertenecen a su vez al primer grupo. En otras palabras, **cuantos vecinos de `G(y.k)` estan conectados a su vez a otros vecinos de `G(y.k)`**, formando una pseudo componente conexa. Así suponemos que **por cada grupo de vecinos conectados entre sí habrá que utilizar un nuevo color**.

Implementación

Nota: *El código de la práctica se encuentra en la subcarpeta `blocks/manu343726/practica2mar/`. El resto son dependencias externas y scripts de compilación*.

Estructuras de datos

En primer lugar es necesaria una estructura de datos para guardar la topología de un grafo. Dicha estructura es la plantilla `graph<Node>` definida en el archivo `graph.hpp`. `graph` implementa las operaciones de añadir nodos al grafo, obtener el conjunto de vecinos de un nodo dado, etc. `graph` utiliza un vector de nodos y una mariz de adyacencia para las aristas, representada por el tipo `adyacency_matrix`.

Matriz de adyacencia

La matriz de adyacencia consiste en un vector de booleanos de tamaño `n*n` representando una matriz bidimensional. Nótese que `std::vector<bool>` no es un array de booleanos sino un bitarray. Esto puede llegar a ser problemático ya que técnicamente dicha especialización de `vector` no cumple las propiedades de un contenedor, pero tiene el beneficio de reducir en gran medida el consumo de memoria. ***Nota: `std::vector<bool>` está optimizado para reducir espacio, pero no mejorar el rendimiento***.

Operaciones como obtener la lista de aristas del grafo y el conjunto de vecinos de un nodo `G(i)` son implementadas por dicha estructura. La implementación utiliza la biblioteca `range-v3`, una propuesta para C++17 de añadir rangos y algoritmos lazy a la STL. Para entender la implementación, sirva la función `adyacency_matrix::neighbors(i)` que devuelve los índices de los nodos vecinos de `G(i)`. En pseudocódigo:

```
func neighbors(i)
    return iota(0,n-1)
           //generar rango de enteros [0,n-1]
           filter(lambda x -> !matriz_bools[i][x])
           //filtrar aquellos x de [0,n-1] donde matriz_booleanos[i][x]
           //es falso
```

La mecánica es muy parecida a la de lenguajes declarativos como haskell. Se eligió dicha biblioteca por la sencillez de uso y el rendimiento que provee: Dado que la aplicación de operaciones sobre un rango es perezosa, dichas operaciones se realizan bajo demanda no requiriendo ninguna petición/consumo de memoria para recibir y manipular sus resultados. Esto es importante ya que hoy en día el cuello de botella del rendimiento es la memoria, no el uso de CPU. Así, se han probado matrices de adyacencia de 100.000 nodos (i.e. 10.000.000.000 entradas en la matriz de adyacencia) con un tiempo de ejecución de 10ms para el algoritmo de cómputo de aristas.

Grafo

La implementación del grafo es bastante simple, ya que las operaciones de manipulación de este son responsabilidad de la matriz de adyacencia. En la mayoría de los casos, los métodos de `graph` no son mas que *views* de los resultados de `adyacency_matrix`, dándoles formato de nodo en lugar de índices.

Además `graph.hpp` contiene una función para generar grafos aleatorios.

Algoritmo

El algoritmo se implementa en la función `colorize()` de `main.cpp`, que toma como argumentos el grafo a colorear y las cotas, devolviendo la solución encontrada junto con algunas estadísticas. Se ha intentado seguir el esquema dado en clase lo máximo posible.

Ejecución

Para compilar la práctica, ejecute los siguientes comandos en la carpeta raiz de la entrega:

```
$ cmake . -DCMAKE_BUILD_TYPE=Release
$ make
```

Es necesario un compilador que soporte C++14. GCC a partir de la versión 4.9.2, Clang 3.5 en adelante. El ejecutable se encuentra en la subcarpeta `bin/`. `Este toma como argumento el tamaño del grafo a procesar:

```
$ ./bin/manu343726_practica2mar_main 100
```

Conclusiones

Todas las pruebas se han realizado en un equipo con un procesador Intel i7 4790k @ 4.5GHz, 16GB RAM 1300Mhz . Dichas pruebas confirman la dificultad de encontrar buenas cotas para este problema: Para grafos de tamaño `n >= 100` el problema se hacía inmanejable, con tiempos de ejecución indefinidos (teniendo que abortar la ejecución) y expansión del orden de decenas de millones de nodos.

Para tamaños de grafos triviales (n entre 10 y 50) el algoritmo se comporta de una manera razonable, con unos tiempos de ejecución del orden de decenas de segundos. Cabe destacar que dichos tiempos dependen en gran medida del orden de evaluación del problema: Se han observado casos poco frecuentes en los que, para un tamaño pequeño del grafo que en la mayoría de los casos se resuelve en poco tiempo, se dan ejecuciones con tiempos del orden de minutos en lugar de segundos, debido a la topología del grafo, orden de expansión (la solución aparece muy tarde), etc.

> Written with StackEdit.