Rainbow Version of Dirac's Theorem: An Algorithmic Approach

Nathan Luiz Bezerra Martins
Willian Miura Mori

Orientadora: Yoshiko Wakabayashi

Departamento de Ciência da Computação,
Instituto de Matemática e Estatística, Universidade de São Paulo

Resumo

Dada uma coleção $G = G_1, G_2, \ldots, G_n$ de grafos de ordem n, definidos sobre o mesmo conjunto de vértices e que satisfazem a condição de Dirac para cada G_i, existe um G-transversal que forma um circuito hamiltoniano, também conhecido como Circuito Hamiltoniano Rainbow. Neste trabalho, desenvolvemos um algoritmo eficiente que encontra um Circuito Hamiltoniano Rainbow. Fizemos implementações tanto em `C++` quanto em `Python` e realizamos testes de desempenho para comparar as duas versões. Utilizamos a biblioteca `manim` para fazer uma animação gráfica do algoritmo.

Conceitos básicos

Definições principais:

Um grafo simples é um grafo não direcionado sem laços e sem arestas múltiplas.

Um **ciclo hamiltoniano** de G é um ciclo que visita cada vértice de G exatamente uma vez.

$\delta(G)$ é o grau mínimo de um vértice em G.

Teorema de Dirac (1952): Se um grafo simples G com n vértices tem grau mínimo $\delta(G) \geq \frac{n}{2}$, então G contém um ciclo hamiltoniano.

Versões Rainbow de problemas clássicos

Definição:
A versão rainbow de um problema na teoria dos grafos é uma variação que adiciona a restrição de cores à solução desejada. Nesse contexto, o termo rainbow (arco-íris) refere-se a estruturas em um grafo que utilizam elementos provenientes de diferentes subconjuntos, ou arestas com diferentes rótulos ou cores, garantindo que não haja repetições.

Teorema de Dirac (*Versão Rainbow*):
Dada uma coleção $G = G_1, G_2, \ldots, G_n$ de grafos de ordem n, definidos sobre o mesmo conjunto de vértices e que satisfazem a condição de Dirac para cada G_i, existe um G-transversal que forma um circuito hamiltoniano, também conhecido como Circuito Hamiltoniano Rainbow. Cada grafo G_i pode ser enxergado como se suas arestas fossem coloridas com a cor i.

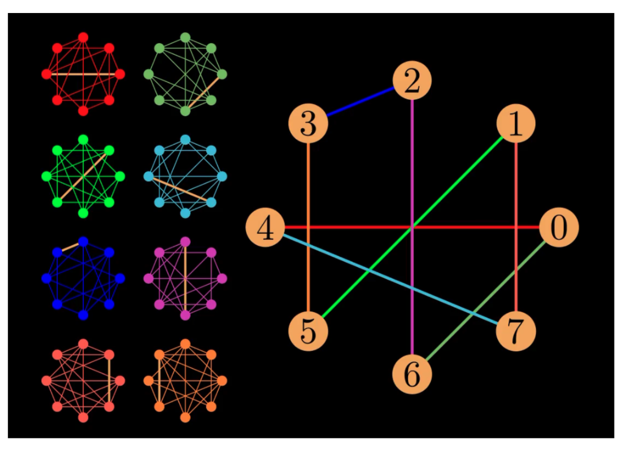

Figura 1: Exemplo de um Circuito Hamiltoniano Rainbow para uma coleção de grafos com 8 vértices.

Mais exemplos clássicos:

Floresta Geradora Mínima Rainbow: Dado um grafo G e uma coleção de cores, encontre uma floresta geradora mínima que utilize exatamente uma aresta de cada cor.

Emparelhamento Perfeito Rainbow: Dado um grafo bipartido G e uma coleção de cores, encontre um emparelhamento perfeito que utilize exatamente uma aresta de cada cor.

As link-cut trees fornecem a seguinte interface:

- `make_root(u)`: enraíza no vértice u a árvore que o contém
- `link(u, v, w)`: dado que os vértices u e v estão em árvores separadas, transforma v em raiz de sua árvore e o liga como filho de u, colocando peso w na nova aresta criada
- `cut(u, v)`: retira da floresta a aresta com pontas em u e v, quebrando a árvore que continha estes vértices em duas novas árvores
- `is_connected(u, v)`: retorna `verdadeiro` caso u e v pertençam à mesma árvore, `falso` caso contrário
- `maximum_edge(u, v)`: retorna o peso máximo de uma aresta no caminho entre os vértices u e v

Todas essas operações consomem tempo $O(\log n)$ amortizado, onde n é o número de vértices na floresta.

Union-Find retroativo

O union-find é uma estrutura de dados utilizada para manter uma **coleção de conjuntos disjuntos**, isto é, conjuntos que não se intersectam.

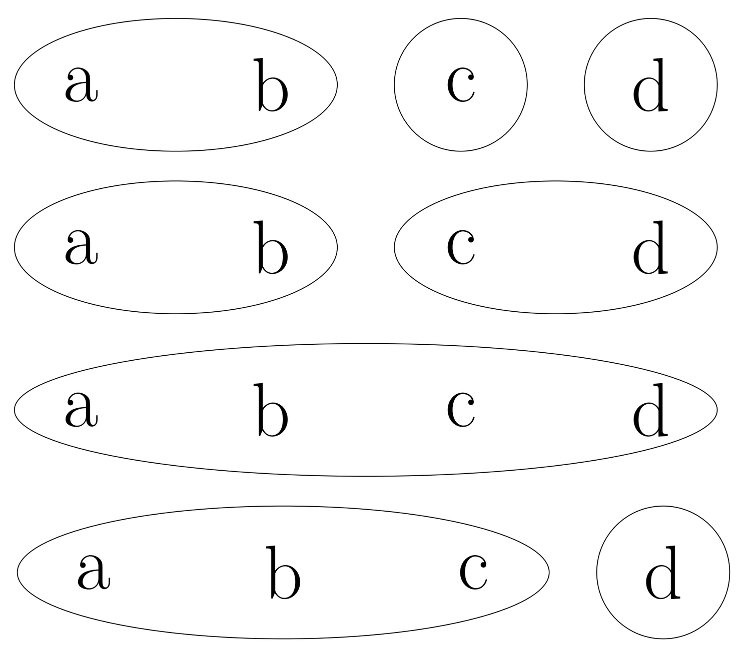

Figura 2: Representação dos conjuntos com os elementos $\{a, b, c, d\}$ após a seguinte sequência de operações: `create_union(a, b, 2)`, `create_union(c, d, 3)`, `create_union(b, c, 4)` e `delete_union(3)`. Cada linha mostra o estado atual da coleção imediatamente após uma operação.

Na sua versão retroativa, implementamos as seguintes operações:

- `create_union(a, b, t)`: adiciona a união dos conjuntos que contém a e b no instante de tempo t
- `same_set(a, b, t)`: consulta se dois elementos pertenciam ao mesmo conjunto no instante t
- `delete_union(t)`: desfaz a união realizada em t

Por exemplo, a Figura 2 mostra o estado de uma coleção de conjuntos disjuntos após quatro operações serem aplicadas. Antes da operação `delete_union(3)`, as consultas `same_set(a, b, 3)` e `same_set(c, d, 3)` retornam `verdadeiro`. Por outro lado `same_set(a, d, 3)` e `same_set(c, d, 3)` retornam `falso` após a chamada da função `delete_union(3)`.

Ideia: Fazer com que os elementos dos conjuntos sejam vértices na floresta mantida por uma link-cut tree, onde cada aresta representa uma operação de `union`. Assim, uma chamada `create_union(a, b, 3)` cria uma aresta de valor 3 entre os vértices a e b. Da mesma forma, uma chamada `delete_union(t)` simplesmente exclui a aresta criada no instante t. Para conferir se dois elementos a e b, no instante de tempo t, estão em um mesmo conjunto, basta conferir se eles estão em uma mesma árvore e se o valor da maior aresta no caminho entre eles é menor ou igual a t, o que significa que todas as uniões já foram realizadas no instante consultado.

Animação

Informações e contato

Para mais informações, acesse a página do trabalho: `https://linux.ime.usp.br/~felipen/mac0499`

Endereço para contato: `felipe.castro.noronha@usp.br`

Referências