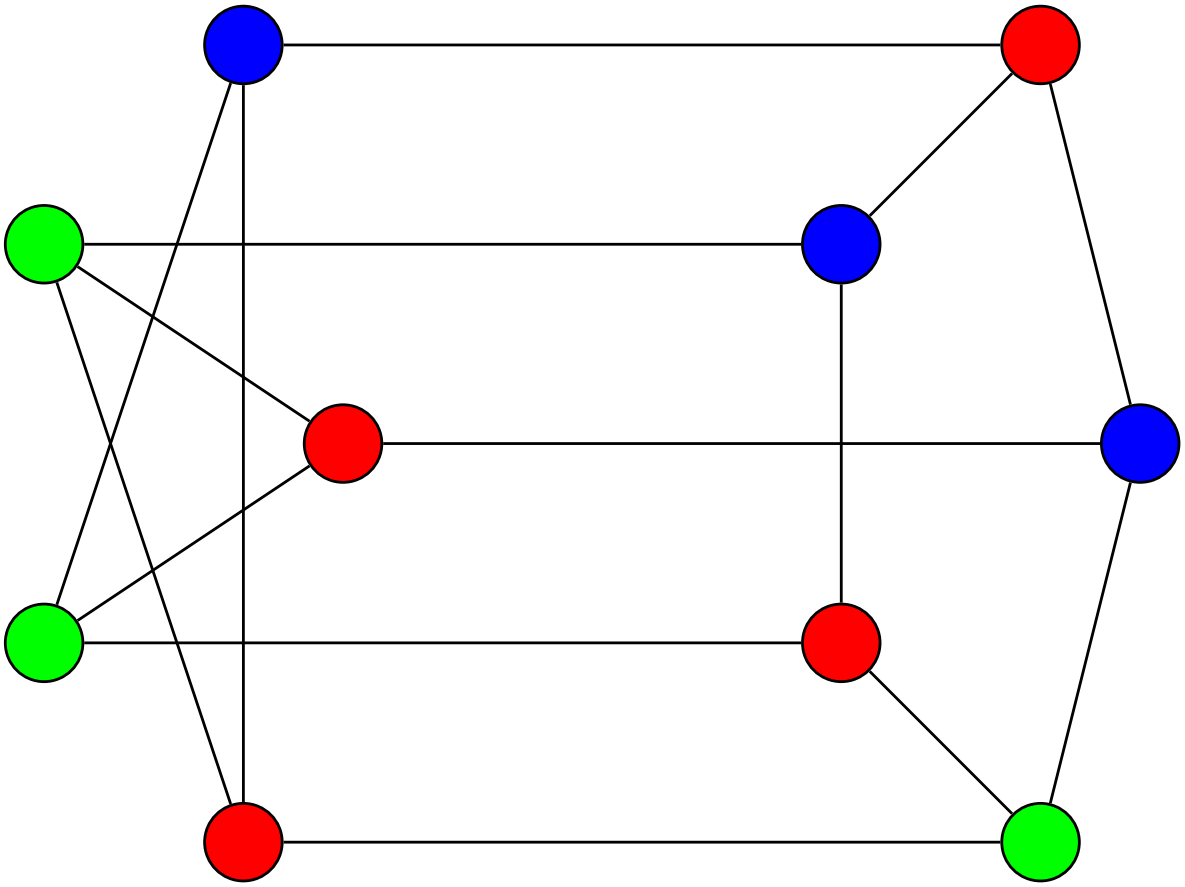
Centro de Matemática, Computação e Cognição
Universidade Federal do ABC
Profa. Carla Negri Lintzmayer

Notas de aula da disciplina Teoria dos Grafos



Esse material é um compilado dos seguintes:

- Sedgewick, R.. *Algorithms in C, part 5: graph algorithms*. 3rd ed. Addison-Wesley. 2002.
- Bondy, J. A.; Murty, U. S. R.. *Graph Theory*. Graduate Texts in Mathematics. Springer. New York. 2008.
- Lintzmayer, C. N.; Mota, G. O.. *Notas de aulas - Análise de algoritmos e estruturas de dados*. Em construção.

9 Aula 11: detalhes de implementação dos algoritmos de Prim e Kruskal

Para o bom acompanhamento desta aula, você precisa conhecer as estruturas de dados Heap e Union-Find.

9.1 Algoritmo de Prim

- O detalhe importante para a implementação do algoritmo de Prim é como encontrar eficientemente uma aresta de menor peso no corte entre os vértices visitados e os não visitados.
- Podemos fazer uso do TAD Fila de Prioridades (de mínimo):
  - É uma coleção dinâmica de elementos que possuem prioridades associadas e cuja operação de remoção deve sempre remover o elemento que possui maior prioridade.
  - Além dessa remoção, temos as operações: consulta ao elemento de maior prioridade, inserção de um novo elemento, alteração da prioridade de um elemento já armazenado e construção a partir de um conjunto pré-existente de elementos.
  - O termo prioridade é usado de maneira genérica, no sentido de que ter maior prioridade não significa necessariamente ter o maior *valor indicativo* de prioridade.
  - Em uma fila de prioridades *de mínimo*, os valores indicativos de prioridade estão multiplicados por -1.
- Para utilizar essa estrutura no Prim, consideraremos as seguintes propriedades:
  - Todos os vértices não visitados (fora da árvore) devem estar na fila.
  - A prioridade de um vértice v é o custo da aresta uv de menor custo tal que u foi visitado, ou ∞ se não houver uma aresta desse tipo.
- Assim, RemoveDaHeap devolve o próximo vértice que deve ser adicionado à árvore que está sendo construída.

```
1: Função PRIM(G, ω)
2:     Para cada v ∈ V(G) faça
3:         prioridade[v] ← −∞
4:         visitado[v] ← 0
5:         pred[v] ← −1
6:     seja s ∈ V(G) qualquer
7:     prioridade[s] ← 0
8:     pred[s] = s
9:     seja Q uma Heap
10:    Para cada v ∈ V(G) faça
11:        INSERENAHEAP(Q, v, prioridade[v])
12:    Enquanto Q ≠ ∅ faça
13:        u ← REMOVEDAHEAP(Q)
14:        visitado[u] ← 1
15:        Para cada v ∈ N(u) faça
16:            Se visitado[v] = 0 e prioridade[v] < −w(uv) então
17:                pred[v] ← u
18:                ALTERAHEAP(Q, v, −w(uv))
```

Proposição 9.1. *Seja G um grafo e $\omega\colon E(G) \to \mathbb{R}$ uma função de pesos nas arestas. O laço **enquanto** de PRIM(G, ω) mantém a seguinte invariante:*

"No início de qualquer iteração, vale que (1) o subgrafo T tal que $V(T) = \{v \in V(G)\colon visitado[v] = 1\}$ e $E(T) = \{\{v, pred[v]\}\colon v \in V(T) \text{ e } pred[v] \neq v\}$ é uma árvore que está contida em uma árvore geradora mínima de G e (2) para qualquer vértice $v \in Q$ com $pred[v] \neq -1$ e $pred[v] \neq v$, o valor $-prioridade[v]$ é o peso de uma aresta de menor peso que conecta v a um vértice visitado."

Lema 9.2. *Seja G um grafo e $\omega\colon E(G) \to \mathbb{R}$ uma função de pesos nas arestas. Ao final de PRIM(G, ω), o subgrafo T tal que $V(T) = \{v \in V(G)\colon visitado[v] = 1\}$ e $E(T) = \{\{v, pred[v]\}\colon v \in V(T) \text{ e } pred[v] \neq v\}$ é uma árvore geradora mínima para G.*

- Tempo de execução de PRIM(G, s), considerando a implementação da fila de prioridades como um Heap Binário:
  - Claramente as linhas 2 a 9 levam tempo total $\Theta(V)$ para executar.
  - Cada inserção na Heap leva tempo $O(\log V)$, de modo que o laço da linha 10 leva tempo total $O(V \log V)$.
  - Cada iteração do laço **enquanto** remove um vértice da Heap e visita-o, e um vértice não é mais inserido na Heap. Assim, as linhas 12, 13 e 14 executam $\Theta(V)$ vezes, sendo apenas a linha 13 não leva tempo constante para executar, mas sim tempo $O(\log V)$.
  - As linhas 15 e 16 dependem da estrutura de dados que está implementando o grafo:
    - Em matriz de adjacências, elas levam tempo $\sum_{u \in V(G)} \Theta(V) = \Theta(V^2)$.
    - Em listas de adjacências, levam tempo $\sum_{u \in V(G)} \Theta(d(u)) = \Theta(E)$.
  - Finalmente, note que a linha 18 será executada no máximo uma vez para cada aresta do grafo (quando visitamos um de seus extremos), sendo que uma execução leva tempo $O(\log V)$, tendo tempo total $O(E \log V)$, portanto.
  - Assim, o tempo total em matriz de adjacências é $\Theta(V)+O(V \log V)+O(V \log V)+\Theta(V^2) + O(E \log V) = O(V^2 + E \log V)$.
  - E o tempo total em listas de adjacências é $\Theta(V) + O(V \log V) + O(V \log V) + \Theta(E) + O(E \log V) = O(E \log V)$.

9.2 Algoritmo de Kruskal

- O detalhe importante para a implementação do algoritmo de Prim é como verificar eficientemente se uma aresta está entre componentes conexas distintas de $G[F]$.
- Podemos fazer uso do TAD Conjuntos Disjuntos:
  - É uma coleção dinâmica de elementos que estão *particionados* em grupos (cada elemento está em algum grupo e um elemento não está em dois grupos diferentes).
  - Todo grupo possui um representante, que é algum elemento contido nele que o identifica.
  - Ele fornece operações que detectam em qual grupo um elemento está (*find*) e que unem dois grupos diferentes (*union*).
- Para utilizar essa estrutura no Kruskal, consideraremos as seguintes propriedades:
  - Os vértices serão os elementos.
  - Um grupo deve conter todos os vértices de uma componente conexa de $G[F]$.
- Assim, se a aresta uv sendo testada for tal que $\textsc{Find}(u) \neq \textsc{Find}(v)$, então podemos adicioná-la a F.

```
1:  Função KRUSKAL(G, ω)
2:      sejam C[1..E] e F[1..V − 1] dois vetores
3:      copie as arestas de G para C com seus respectivos pesos
4:      ordene C de modo não-decrescente de acordo com o peso das arestas
5:      Para cada v ∈ V(G) faça
6:          MAKESET(v)
7:      k ← 1
8:      Para i ← 1 até E faça
9:          seja uv a aresta em C[i]
10:         Se FIND(u) ≠ FIND(v) então
11:             UNION(u, v)
12:             F[k] ← {uv}
13:             k ← k + 1
14:     Devolve F
```

Proposição 9.3. *Seja G um grafo e $\omega\colon E(G) \to \mathbb{R}$ uma função de pesos nas arestas. O segundo laço* ***para*** *de* KRUSKAL*(G, ω) mantém a seguinte invariante: "No início de qualquer iteração, vale que o subgrafo $G[F]$ está contido em uma árvore geradora mínima de G".*

- Tempo de execução de KRUSKAL(G, s):
  - A linha 3 depende da estrutura de dados que está implementando o grafo:
    - em matriz de adjacências, leva tempo $\Theta(V^2)$;
    - em listas de adjacências, leva tempo $\Theta(V + E)$.
  - A linha 4 leva tempo $O(E \log E)$ usando algum algoritmo de ordenação como MergeSort ou HeapSort.
  - O laço da linha 5 leva tempo $\Theta(V)$.
  - As linhas 8 e 9 claramente levam tempo $\Theta(E)$.
  - As linhas 12 e 13 levam tempo $\Theta(V)$.
  - Por fim, note que são feitas V operações de *make*, $V - 1$ operações de *union* e $2E$ operações de *find* ao todo: a linha 10 executa uma vez para cada aresta do grafo inicial e a linha 11 executa uma vez para cada aresta da árvore final. Assim, são $\Theta(V + E)$ operações.
  - Assim, se usarmos a implementação de conjuntos disjuntos com listas ligadas e *weighted-union*, teremos o tempo total:

* $\Theta(V^2) + O(E \log E) + \Theta(V) + \Theta(E) + \Theta(V) + O(V + E + V \log V) = O(V^2 + E \log V)$ em matriz de adjacências;
* $\Theta(V + E) + O(E \log E) + \Theta(V) + \Theta(E) + \Theta(V) + O(V + E + V \log V) = O(E \log V)$ em listas de adjacências.

– E se usarmos a implementação de conjuntos disjuntos com florestas e *union by rank*, teremos o tempo total:

* $\Theta(V^2)+O(E \log E)+\Theta(V)+\Theta(E)+\Theta(V)+O(E \log V) = O(V^2+E \log V)$ em matriz de adjacências;
* $\Theta(V+E)+O(E \log E)+\Theta(V)+\Theta(E)+\Theta(V)+O(E \log V) = O(E \log V)$ em listas de adjacências.