Algoritmo Minimum Spanning Tree (MST)

Rafael G. Nagel

2018-06-22

Outline

Aplicações[1] e problema

- Geralmente aplicações associadas a otimização em redes
  - telefone
  - elétrica
  - hidráulica
  - TV a cabo
  - caminhos
- Outras aplicações indiretas
  - aprendizagem de características para verificação facial em tempo real (?)
  - caminhos de máximo gargalos

[1] https://www.geeksforgeeks.org/applications-of-minimum-spanning-tree/

Condições para aplicação do algoritmo

- grafo com arestas de pesos positivos (> 0)
- arestas não-direcionadas
- encontrar um conjunto de *arestas* que somadas tenha um peso mínimo e que conecte todos os vértices
- número de *arestas* que se deseja:

```
arestas = número de vértices - 1
```

Algoritmos que implementam MST[2]

1. algoritmo de Kruskal
2. algoritmo de Prim
3. algoritmo de Boruvka

- por que Kruskal?
  - mais fácil de aplicar e compreender

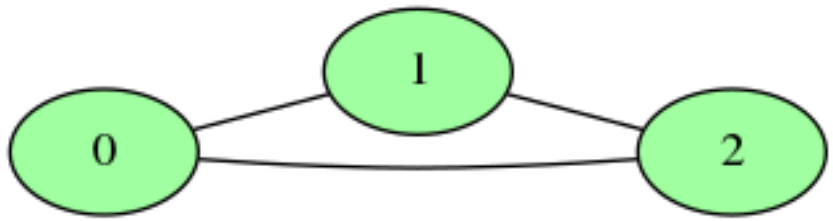
[2] https://www.ics.uci.edu/~eppstein/161/960206.html

Visão geral

1. ordena-se as arestas por peso *crescemente*
2. pega-se uma aresta e verifica-se se ao adicioná-la no grafo final (MST) um ciclo será formado
   - se ciclo formado, então descarta-se a aresta (do MST)
   - se não forma ciclo, adicione-a ao grafo (MST)
3. repita o passo 2. até: `arestas = vértices - 1`

Como checar se um grafo formou um ciclo?

- algoritmo *Union-find*[3]
  - baseia-se no uso temporário de um vetor para salvar os *pais* dos vértices.



- Cada vértice é um sub-conjunto
- "find" $\rightarrow$ encontrar conjunto do vértice
- "union" $\rightarrow$ juntar dois sub-conjuntos

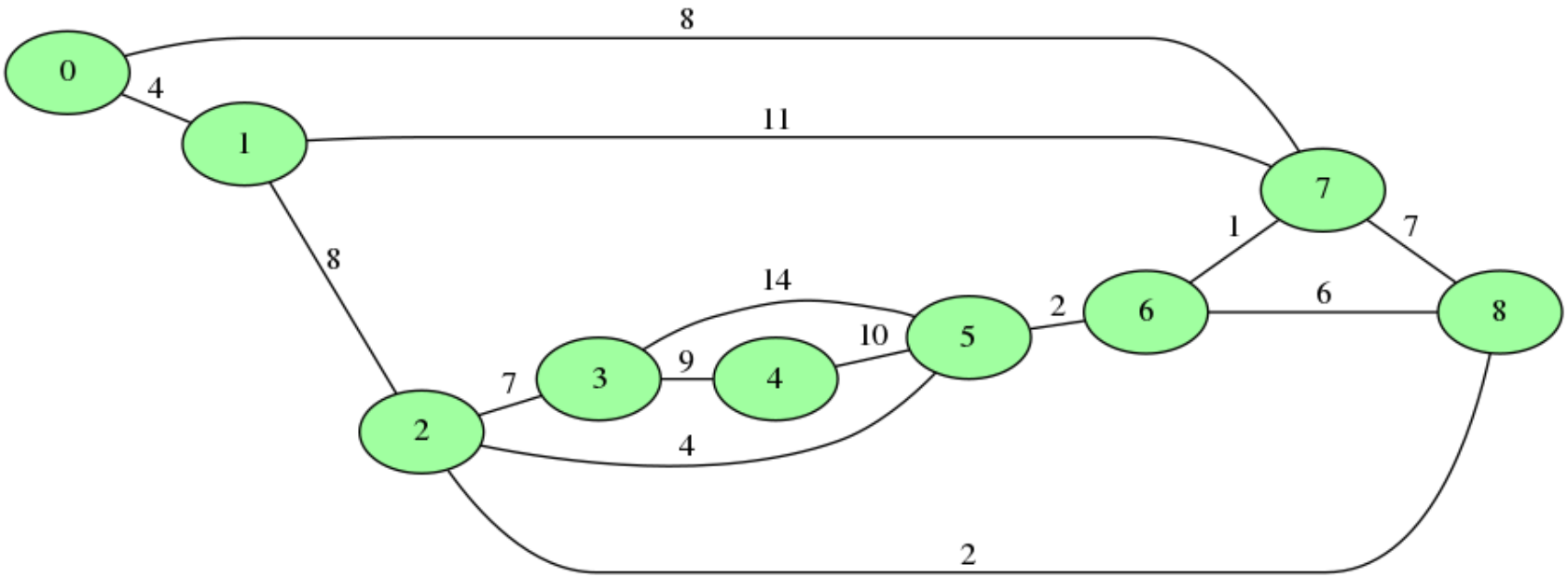
0	1	2
-1	-1	-1

0	1	2
1	-1	-1

0	1	2
1	2	-1

[3] https://www.geeksforgeeks.org/union-find/

Passo a passo com um exemplo[4]

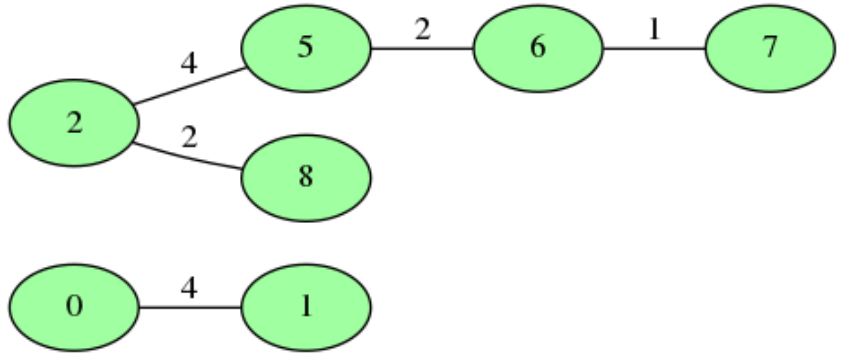


[4] https://www.geeksforgeeks.org/greedy-algorithms-set-2-kruskals-minimum-spanning-tree-mst/

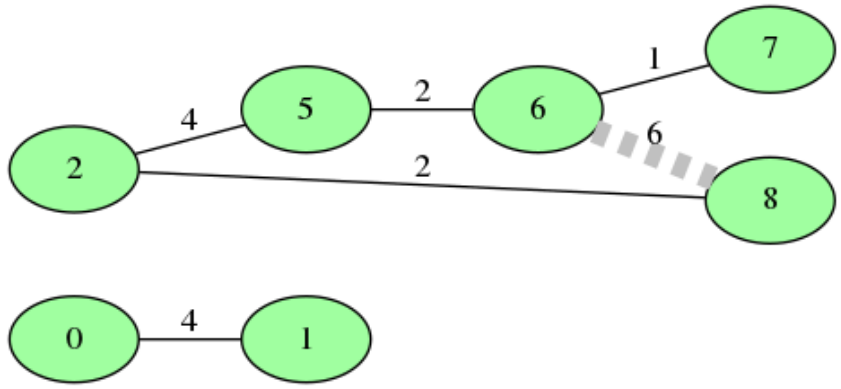
Grafo ordenado por arestas (crescemente)

weight	src	dst
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

pega-se os vértices da lista ordenada e tenta-se inserir cada um

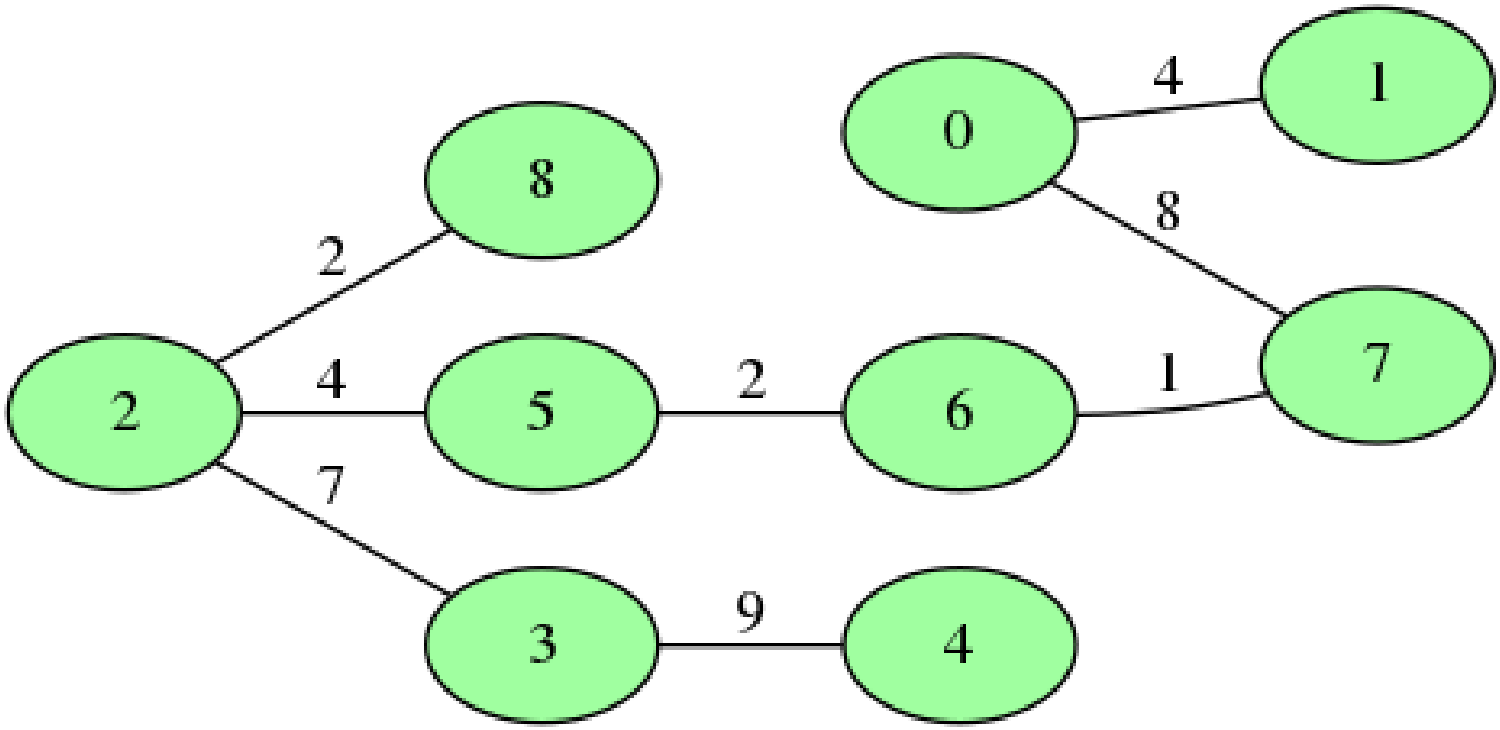


. . . sem ciclos até agora.



. . . ao incluir 8-6 forma-se um ciclo. Portanto não inclua essa aresta.

repita até: arestas = vértices - 1



Complexidade

relembrando os passos:

1. ordenar grafo crescentemente (qsort $\rightarrow$ O(nlogn))
2. aplicar algoritmo MST (Kruskal)
   - 2.1 para *cada aresta* da lista ordenada:
     - 2.1.1 inclua-a no grafo MST
     - 2.1.2 checa-se se *ciclo formado*; se sim remove essa aresta

     (algoritmo find-union)

complexidade do algoritmo Kruskal

- O *find-union* é O(n) no *presente trabalho*
  - poderíamos melhorar isso para O(logn) usando *union by Rank or Height*[5]

[5] https://www.geeksforgeeks.org/union-find/

Complexidade

Considerar o pior caso

- Toda vez que um vértices é adicionado no grafo, podemos ter:

```
arestas += número de vértices - 1
```

- Assim, número de arestas no pior caso é:

$$a \approx v^2$$
$$\text{quando } a \to \infty$$

Porém, essa implementação:

$$O(e \times \log e) + O(e \times O(v))$$

$$\mathrm{O}(\mathrm{e} \times \log e + v^2 \times v)$$
$$\mathrm{O}(\mathrm{e} \times \log v + v^3)$$

Complexidade

Supondo a implementação com: `union-find = O(logn)`

$$O(e \times \log e) + O(e \times \log v)$$

$$e = v^2 \rightarrow \log e = \log v^2 = 2 \times \log v \approx \log v$$

$$\therefore$$

$$O(e \times (\log v + \log v)) = O(e \times 2 \times \log v)$$

$$\therefore$$

$$O(e \times \log v)$$

Código

Estruturas

- grafo
- vertice
- lista
- nó

funções (extras)

- hasCycle()
- union()
- find()
- `grafo_remove_ultima_aresta()`
- `compara_arestas()`
  - usada no *qsort* da *lib c*
- outras menos importante
  - e.g. `grafo_get_arestas_arr()`

Conclusões

- Ao invés de criar novas *estruturas* ou *módulos*, adicionou-se *membros* às estruturas e novas *funções* aos módulos.

Como checar adição duplicada de vértices sem ser O(n)?

- Essa questão aparece quando adiciona-se arestas ao gráfo (MST) para checar se há *ciclos*
- Assim, essa complexidade tem mais um fator v:

$$O(e \times \log v + v^4)$$

Por fim: algoritmo Kruskal é *muito mais fácil* de implementar:

- Tradeoffs: Complexidade vs. Tempo de código