Aula 14 - Árvore Geradora Mínima

Luís Felipe

UFF

19 de Maio de 2026

Luís Felipe

19/05/26

A estratégia gulosa

- Em um jogo de xadrez, jogar pensando apenas no que é bom no momento da jogada, pode ser desastroso.
- Mas em vários outros jogos, como o jogo de palavras cruzadas, é possível se sair bem fazendo o que parece melhor no momento.
- A estratégia gulosa é aquela na qual a solução é construída passo a passo e o próximo passo é sempre o mais vantajoso no momento.
- Um algoritmo guloso nunca se arrepende de um passo anterior.
- Obs.: Algoritmo de Dijkstra é guloso (Aula 11).
- Para vários problemas, esta pode não ser a estratégia ótima, mas para vários outros um algoritmo guloso pode ser uma abordagem atraente e ótima.

Luís Felipe

19/05/26

Árvore geradora Mínima

- Imaginem que temos alguns computadores, conexões entre eles e precisamos fazer uma rede "sem redundância" com esses computadores.
- A conexão entre computadores possui um custo.
- Nosso objetivo é interligá-los com custo mínimo.
- Este problema pode ser modelado utilizando Grafos:
  - Cada computador é um vértice,
  - Cada possível ligação entre os computadores é uma aresta.
- Que estrutura devemos construir para minimizar o custo?
  - Manter ciclos seria uma boa estratégia?

Luís Felipe

19/05/26

Teoria dos Grafos

- Relembrando: Dado um grafo conexo $G = (V, E)$, uma árvore geradora de G é um subgrafo gerador $H = (V, E')$ de G, i.e., um subgrafo que possui o mesmo conjunto de vértices de G, que é uma árvore, i.e., um grafo conexo e acíclico.
- No problema da árvore geradora mínima (MST), o grafo G é ponderado, ou seja, a cada aresta é atribuído um custo.
- O objetivo é construir uma árvore geradora para G de menor custo total.
- A principal justificativa para a construção de uma árvore geradora é que a remoção de arestas de ciclos não desconecta o grafo.

Luís Felipe

19/05/26

Abordagem Gulosa 1 - Kruskal

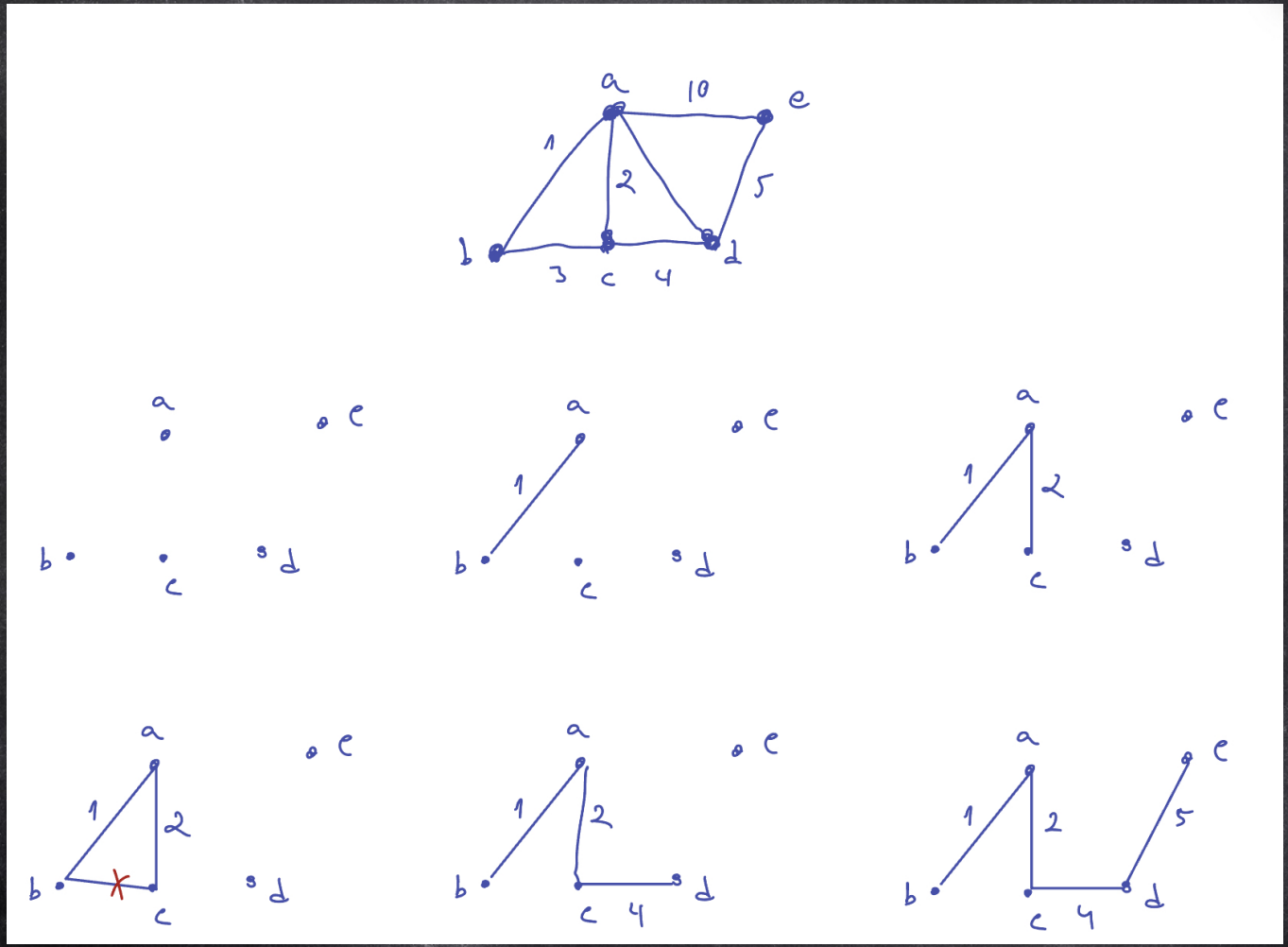
- Começa com o grafo nulo, i.e., sem arestas.
- Adiciona sempre a aresta de menor custo e que não forma ciclo no momento.
- É guloso, pois adicionar a aresta de menor custo é a estratégia mais vantajosa no momento.

```
1  Entrada: grafo G = (V, E) com pesos nas arestas
2
3  A <- conjunto vazio          # conjunto de arestas da árvore
4
5  ordenar arestas de E em ordem crescente de peso
6
7  para cada aresta (u, v) em E (nessa ordem) faca
8      se adicionar (u, v) não forma ciclo entao
9          A <- A 'união' {(u, v)}
10
11 retornar A
```

Luís Felipe

19/05/26

Exemplo



Luís Felipe

19/05/26

Corretude do Kruskal

- Baseia-se em propriedade da teoria da conectividade
- Um corte de arestas, denotado por $[S, S']$ é um conjunto de arestas que têm um extremo em $S \subset V$ e outro em $S' = V \setminus S$.

  Propriedade do corte: Suponha que um conjunto de arestas X faça parte de uma MST de $G = (V, E)$. Para todo $S \subset V$ tal que X não possua aresta em $[S, S']$, $X \cup \{e\}$ faz parte de alguma MST, onde e é a aresta de $[S, S']$ de menor custo.
- Tradução: Você sempre pode adicionar a aresta de menor custo de qualquer corte, dado que X não possui arestas no corte.
- Mas por que funciona??

Luís Felipe

19/05/26

Corretude do Kruskal

Ideia da prova:

Seja X o conjunto de arestas de uma MST, T. Se e é uma aresta de X, não há nada a mostrar.

Suponha que $e \notin X$. Vamos construir uma árvore T' diferente de T de custo mínimo e que possua a aresta e.

Adicione e, aresta de custo mínimo em $[S, S']$ a T. Cria-se um ciclo. Existe outra aresta, digamos e', no corte $[S, S']$. Remova e'.

O grafo é conexo e continua com o mesmo número de arestas de T. Logo, o grafo é uma árvore.

O custo total de T' é o custo de T - o custo de e' + custo de e.

Como e tem custo mínimo em $[S, S']$, o custo de e' é no mínimo o custo de e, mas como T é MST, então sabemos que o custo de e é igual ao custo de e'.

Logo, T' é uma MST e e faz parte de T'.

Luís Felipe

19/05/26

Complexidade do Kruskal

- Primeiro, ordena as arestas do grafo por peso:
  $O(m \log m)$, onde m é o número de arestas
- Depois, utilizando uma estrutura de dados adequada, verifica a não formação de ciclos e faz-se a fusão:
  $O(m \log n)$, onde n é o número de vértices
- Como $\log m = O(\log n^2) = O(2 \log n) = O(\log n)$, temos complexidade $O(m \log n)$

Como é essa estrutura de dados?

A cada etapa, o algoritmo escolhe uma aresta para ser adicionada na solução parcial.

Assim, precisamos testar se uma aresta candidata uv é de forma que u e v estejam em componentes conexas distintas.

Em caso positivo, as componentes devem ser unidas.

Estrutura utilizada: Conjuntos Disjuntos

Operações:

- *makeset*(x): cria um conjunto unitário contendo x
- *find*(x): retorna qual conjunto contém x
- *union*(x, y): junta conjuntos contendo x e y.

Com isso, podemos escrever com mais detalhes o algoritmo de Kruskal.

Luís Felipe

19/05/26

Algoritmo de Kruskal

Entrada: Um grafo ponderado conexo $G(V, E)$
Saída: Uma MST definida pelas arestas X.

1. Para cada vértice $u \in V$:
2. $\quad$ *makeset*(u)

3. $X = \{\}$
4. $E =$ conjunto E ordenado pelos pesos
5. Para cada aresta $\{u, v\} \in E$:
6. $\quad$ Se *find*$(u) \neq$ *find*(v):
7. $\quad\quad$ adicione aresta $\{u, v\}$ a X
8. $\quad\quad$ *union*(u, v)

Obs.: Total de operações:
makeset: $|V|$, linha 1;
find: $2|E|$, linhas 5 e 6, uma para cada extremo de aresta;
union: $|V| - 1$, linha 8, uma árvore há $|V| - 1$ arestas.

Luís Felipe

19/05/26

Como efetuar as operações?

Armazenamos um conjunto em uma árvore direcionada:

- A raiz representa o nome do conjunto;
- Cada elemento x possui um marcador $rank(x)$, representando qual altura da árvore com x sendo a raiz.
- Cada nó x possui um ponteiro para seu nó pai na árvore, $\pi(x)$. Se x for raiz, então $\pi(x) = x$.

makeset(x):

$\pi(x) = x$

$rank(x) = 0$

Complexidade: $O(1)$

find(x):

Enquanto $x \neq \pi(x)$:

$x = \pi(x)$

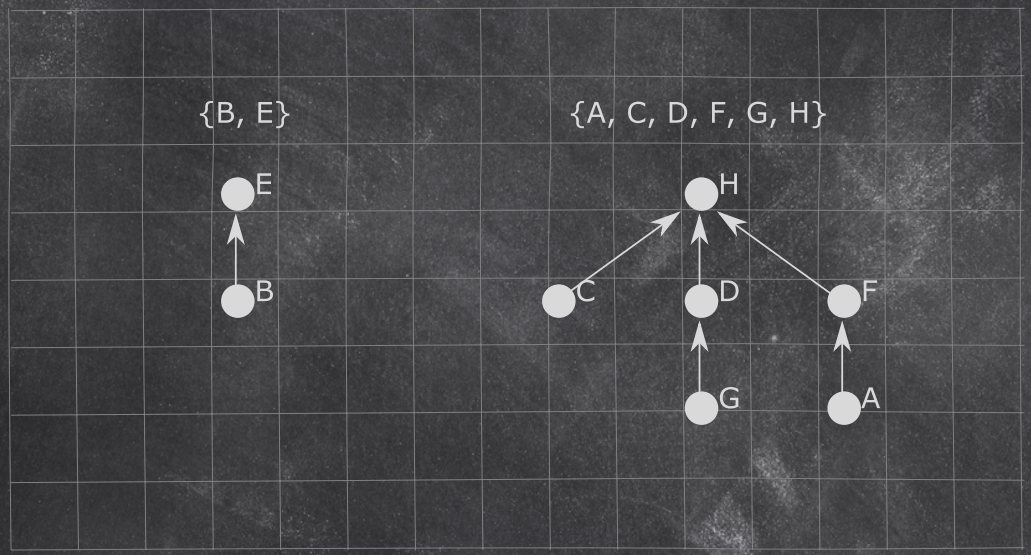
Retorne x

Complexidade: $O(h)$, onde h é a altura da árvore associada a x

Luís Felipe

19/05/26

Árvore



Note que a altura da árvore associada a um conjunto depende da operação de união de conjuntos.

Desejamos construir uma árvore de menor altura possível.

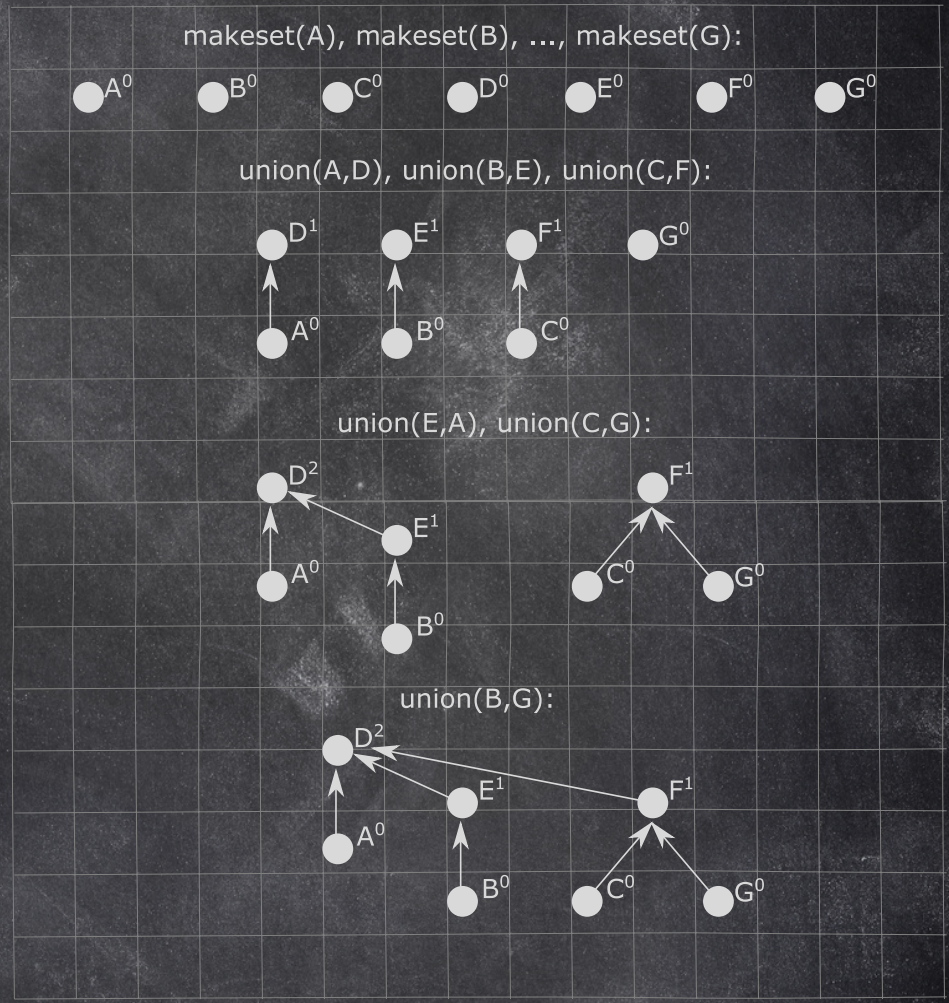
Unimos duas árvores tornando a raiz de uma delas a raiz da nova árvore.

Luís Felipe

19/05/26

União de duas árvores

union(x, y):
  $r_x = find(x)$
  $r_y = find(y)$
  Se $r_x = r_y$:
    Retorne r_x
  Se $rank(r_x) > rank(r_y)$:
    $\pi(r_y) = r_x$
  Senão:
    $\pi(r_x) = r_y$
    Se $rank(r_x) = rank(r_y)$:
      $rank(r_y) = rank(r_y) + 1$



$rank(x)$ está representado pelo índice sobrescrito do nó x.

Luís Felipe

19/05/26

Propriedade central

Note que para saber se dois elementos estão em um mesmo conjunto (que é o mesmo que dizer que dois vértices estão numa mesma componente conexa), devemos verificar se *find* dos dois vértices é igual ou não.

Isso implica percorrermos no pior caso a altura das árvores associadas para ter essa resposta.

- A altura da árvore é exatamente o rank(x) tal que x é a raiz da árvore.

  Por que? Consequência do algoritmo $union(x, y)$

Lema: O *rank* de qualquer árvore é no máximo $O(\log n)$.

Prova: Para que uma árvore passe a ter $rank = k$ pela primeira vez como resultado de uma união, temos pelo menos duas árvores de altura $k - 1$ cada.

Isso faz com que após a união uma subárvore da raiz tenha altura $k - 2$ e a outra com altura $k - 1$. Seja $|T_h|$ número de nós numa árvore de altura h. (continua...)

Luís Felipe

19/05/26

Assim, $|T_k| \geq 1 + |T_{k-1}| + |T_{k-2}|$, onde $|T_0| = 1$, $|T_1| \geq 2$.

Considerando $|F_k|$ o k-ésimo elemento da sequência de Fibonacci, temos: $|T_k| \geq |F_k|$.

Como $\frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^k < 1$, para $k > 0$, então:
$|F_k| = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^k - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^k > \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^k - 1$:

$|T_k| > \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^k - 1$, como a base da potenciação é uma constante, então:

$|T_k| > O(c^k)$.

Aplicando $\log_c$ em ambos os lados e fazendo mudança de base:
$\log_c |T_k| > k \ \ \therefore k < \frac{1}{\log_2 c} \log_2 |T_k| = O(\log |T_k|)$

Ou seja: $k < O(\log n)$

Luís Felipe

19/05/26

Concluindo análise do Kruskal

- Ordenação das arestas: $O(m \log m) = O(m \log n)$
- Para cada aresta verificamos se seus extremos estão numa mesma árvore parcial: $O(\log n)$.
  - Como fazemos isso para todas arestas no pior caso, então: $O(m \log n)$
- Ao todos, temos $O(m \log n) + O(m \log n) = O(m \log n)$

Luís Felipe

19/05/26

Algoritmo de Kruskal em python

```python
def makeset(x, pai, rank):
    pai[x] = x
    rank[x] = 0

def find(x, pai):
    while x != pai[x]:
        x = pai[x]
    return x

def union(x, y, pai, rank):
    rx = find(x, pai)
    ry = find(y, pai)

    if rx == ry:
        return rx

    if rank[rx] > rank[ry]:
        pai[ry] = rx
        return rx
    else:
        pai[rx] = ry

        if rank[rx] == rank[ry]:
            rank[ry] += 1

        return ry
```

```python
def kruskal(V, E):
    """
    V: lista de vertices
    E: lista de arestas no formato (peso, u, v)

    Retorna:
    X: conjunto de arestas da MST
    """

    pai = {}
    rank = {}

    for u in V:
        makeset(u, pai, rank)

    X = []

    E = sorted(E)

    for peso, u, v in E:
        if find(u, pai) != find(v, pai):
            X.append((u, v, peso))
            union(u, v, pai, rank)

    return X
```

Luís Felipe

19/05/26

Algoritmo de Prim

- Tem o mesmo objetivo do algoritmo de Kruskal.
- Começa considerando apenas um vértice.
- Escolhe sempre a aresta de menor peso que liga um vértice já incluído na árvore a um vértice ainda não incluído.
- Diferentemente do algoritmo de Kruskal, que constrói florestas e depois as funde, em cada passo do algoritmo de Prim tem-se uma árvore.
- A corretude deste algoritmo também é justificada pela propriedade do corte.
- A complexidade do Algoritmo de Prim é $O(m \log n)$, se uma estrutura de dados adequada for utilizada.

Luís Felipe

19/05/26

Algoritmo de Prim em python

```python
import heapq

def prim(G, W, r):      # G: lista de adjacencia; W: dicionario de pesos; r: raiz inicial

    visitado = {v: False for v in G}
    pai = {v: None for v in G}
    chave = {v: float('inf') for v in G}

    chave[r] = 0
    H = [(0, r)]

    X = []

    while H:
        peso, u = heapq.heappop(H)

        if visitado[u]:
            continue

        visitado[u] = True

        if pai[u] is not None:
            X.append((pai[u], u, peso))

        for v in G[u]:
            if not visitado[v] and W[(u,v)] < chave[v]:
                chave[v] = W[(u,v)]
                pai[v] = u
                heapq.heappush(H, (chave[v], v))

    return X
```

Luís Felipe

19/05/26

Algoritmo de Boruvka

- Começa com o grafo nulo.
- Para cada vértice, o algoritmo adiciona a aresta de menor custo, sempre entre vértices de componentes distintos.
- Isto pode resultar em um grafo desconexo.
- Cada componente conexo é visto como um vértice e o procedimento se repete.
- A corretude deste algoritmo também é justificada pela propriedade do corte.
- A complexidade do Algoritmo de Boruvka é $O(m \log n)$, se uma estrutura de dados adequada for utilizada.

Luís Felipe

19/05/26

Algoritmo de Boruvka em python

```python
def makeset(x, pai, rank):
    pai[x] = x
    rank[x] = 0

def find(x, pai):
    while x != pai[x]:
        x = pai[x]
    return x

def union(x, y, pai, rank):
    rx = find(x, pai)
    ry = find(y, pai)

    if rx == ry:
        return rx

    if rank[rx] > rank[ry]:
        pai[ry] = rx
        return rx
    else:
        pai[rx] = ry

        if rank[rx] == rank[ry]:
            rank[ry] += 1

        return ry
```

```python
def boruvka(V, E):
    """
    V: lista de vertices
    E: lista de arestas (peso, u, v)

    Retorna:
    X: arestas da MST
    """

    pai = {}
    rank = {}

    for v in V:
        makeset(v, pai, rank)

    X = []

    num_componentes = len(V)
```

Luís Felipe

19/05/26

Algoritmo de Boruvka em python (continuação)

```python
    while num_componentes > 1:

        # menor aresta para cada componente
        menor = {}

        for peso, u, v in E:
            ru = find(u, pai)
            rv = find(v, pai)

            if ru == rv:
                continue

            if ru not in menor or menor[ru][0] > peso:
                menor[ru] = (peso, u, v)

            if rv not in menor or menor[rv][0] > peso:
                menor[rv] = (peso, u, v)

        # adiciona as arestas escolhidas
        for comp in menor:
            peso, u, v = menor[comp]

            if find(u, pai) != find(v, pai):
                X.append((u, v, peso))
                union(u, v, pai, rank)
                num_componentes -= 1

    return X
```

Luís Felipe

19/05/26

Comparação dos Algoritmos de MST

- Todos os algoritmos resolvem o mesmo problema da Árvore Geradora Mínima (MST)
- Diferem na forma como constroem a solução:

Algoritmo	Estratégia	Estrutura	Ciclos
Kruskal	Escolhe menor aresta global	Union-Find	Evita explicitamente
Prim	Expande uma árvore	Heap (fila de prioridade)	Evita naturalmente
Borůvka	Múltiplas árvores em paralelo	Union-Find	Evita explicitamente

- Kruskal: bom para grafos esparsos.
- Prim: eficiente com boas estruturas (heap).
- Borůvka: permite paralelismo de forma natural.

Todos são corretos pela propriedade do corte.