Método Guloso

José Elias Claudio Arroyo

Departamento de Informática
Universidade Federal de Viçosa

INF 332 - 2022/2

Outline

Método guloso

O **paradigma guloso** sugere a construção de uma solução através de uma **sequência de passos**.
Em cada passo **escolhe** o item/**elemento mais atractivo** que vê pela frente para fazer parte da solução atual.

Método guloso

O **paradigma guloso** sugere a construção de uma solução através de uma **sequência de passos**.
Em cada passo **escolhe** o item/**elemento mais atractivo** que vê pela frente para fazer parte da solução atual.

Uma escolha deve ser:

- **Factível**
- **Localmente ótima**
- **Irreversível**

Método guloso

O **paradigma guloso** sugere a construção de uma solução através de uma **sequência de passos.**
Em cada passo **escolhe** o item/**elemento mais atractivo** que vê pela frente para fazer parte da solução atual.

Uma escolha deve ser:

- **Factível**: satisfaz as restrições do problema;
- **Localmente ótima**
- **Irreversível**

Método guloso

O **paradigma guloso** sugere a construção de uma solução através de uma **sequência de passos.**
Em cada passo **escolhe** o item/**elemento mais atractivo** que vê pela frente para fazer parte da solução atual.

Uma escolha deve ser:

- **Factível**: satisfaz as restrições do problema;
- **Localmente ótima**: deve ser a melhor escolha local entre todas as escolhas válidas disponíveis nesse passo;
- **Irreversível**

Método guloso

O **paradigma guloso** sugere a construção de uma solução através de uma **sequência de passos.**
Em cada passo **escolhe** o item/**elemento mais atractivo** que vê pela frente para fazer parte da solução atual.

Uma escolha deve ser:

- **Factível**: satisfaz as restrições do problema;
- **Localmente ótima**: deve ser a melhor escolha local entre todas as escolhas válidas disponíveis nesse passo;
- **Irreversível**: uma vez feita, não pode ser desfeita em passos seguintes.

Método guloso

- Algoritmos gulosos são **intuitivos**, **simples** e muito **rápidos**.
- Um algoritmo guloso, geralmente, é útil para resolver **problemas de otimização** que consistem em determinar uma solução S que minimiza ou maximiza um determinado valor (ou seja, deseja-se encontrar a "melhor"solução dentre todas as soluções possíveis).
- Para alguns problemas, a solução construída resulta numa **solução ótima**, para outros resulta numa **aproximação**.
- Normalmente a **complexidade de tempo** é **linear** (ou **polinomial**).
- Um passo de pré-processamento muito comum é **ordenar os elementos**.
- O difícil é provar a otimalidade da solução.

Método guloso

Soluções ótimas obtidas por algoritmos gulosos:

- Troco para sistemas “normais” de moedas;
- Seleção de atividades;
- Código de Huffman.
- Árvore geradora mínima;
- Caminhos mínimos em grafos;

Aproximações:

- Problema do caixeiro viajante;
- Problema da mochila 0/1;
- Outros problemas de otimização combinatorial.

Problema do troco

- Dadas n moedas $d_1, \ldots d_n$, com quantidade ilimitadas. Queremos dar um troco de valor T usando o menor número possível de moedas.

Problema do troco

Problema do troco

- Dadas n moedas $d_1, \ldots d_n$, com quantidade ilimitadas. Queremos dar um troco de valor T usando o menor número possível de moedas.

Exemplo

- Dar um troco $T = 48c$ utilizando as seguinte moedas: $d_1 = 25c, d_2 = 10c, d_3 = 5c, d_4 = 1c$.

Problema do troco

- Dadas n moedas $d_1, \ldots d_n$, com quantidade ilimitadas. Queremos dar um troco de valor T usando o menor número possível de moedas.

Exemplo

- Dar um troco $T = 48c$ utilizando as seguinte moedas: $d_1 = 25c, d_2 = 10c, d_3 = 5c, d_4 = 1c$.

Solução (estratégia) gulosa:
Ordenar as moedas em ordem decrescente de valor. Ou seja, escolher as moedas em ordem decrescente de valor.
Usar a maior quantidade possível da moeda de maior valor, de forma a não passar o valor do troco.

- 1 de 25c, 2 de 10c e 3 de 1c, $\Rightarrow$ total 6 moedas.

Problema do troco

Esta estratégia gulosa (que ordena as moedas em ordem decrescente) é:

- **Ótima** para qualquer sistema "normal" de moedas;
- **Não é ótima** para algum conjunto de moedas;
  - *Por exemplo*: $d_1 = 7c, d_2 = 5c, d_3 = 1c$ e $T = 11c$.
  - Estratégia gulosa: $T = d_1 + 4d_3$, $\Rightarrow$ 5 moedas
  - Solução ótima: $T = 2d_2 + d_3$, $\Rightarrow$ 3 moedas

Soma de Custo Mínimo

Suponha que somar a e b custa $a + b$.
Por exemplo, somar 4 com 10 custaria 14.

Soma de Custo Mínimo

Suponha que somar a e b custa $a + b$.
Por exemplo, somar 4 com 10 custaria 14.

Somar um conjunto de n números, por exemplo $\{12; 4; 8\}$, existem várias maneiras (ordens) de fazer, resultando custos totais diferentes:

- Maneira 1:
  (12+4) = 16 (custo 16)
  16 + 8 = 24 (custo **24**)
  Custo total = 40
- Maneira 2:
  (12+8) = 20 (custo 20)
  20 + 4 = 24 (custo **24**)
  Custo total = 44
- Maneira 3:
  (4+8) = 12 (custo 12)
  12 + 12 = 24 (custo **24**)
  Custo total = 36

Soma de Custo Mínimo

Problema da Soma de Custo Mínimo:
Somar um conjunto de n números, com o menor custo possível.

Soma de Custo Mínimo

Estratégia Gulosa:

Escolher em cada passo os dois menores números!

Soma de Custo Mínimo

Estratégia Gulosa:

Escolher em cada passo os dois menores números!

- quanto menores os números, menor o custo
- a **última soma** sempre terá o mesmo custo (seja qual for a ordem dos números)
- consideremos os números a, b e c.

  Se a solução gulosa optar por $a + b$ é porque $a \leq c$ e $b \leq c$.

  Sendo assim, o custo de $a + b$ é $\leq$ que $a + c$ e $b + c$.

  A **última soma** terá custo $a + b + c$.
- a estratégia gulosa sempre determina a **solução ótima**.

Soma de Custo Mínimo

Algoritmo Guloso:

Entrada: conjunto $A = \{a_1, a_2, ..., a_n\}$

- $custo\_total = 0$
- Enquanto $|A| > 1$:
  - (a, b) = Remover do conjunto os dois menores números
  - $custo\_total = custo\_total + (a + b)$
  - $A = A \cup \{a + b\}$ *//Inserir (a + b) no conjuto A*
- return *custo_total*

Soma de Custo Mínimo

Algoritmo Guloso:
Entrada: conjunto $A = \{a_1, a_2, ..., a_n\}$

- $custo\_total = 0$
- Enquanto $|A| > 1$:
  - (a, b) = Remover do conjunto os dois menores números
  - $custo\_total = custo\_total + (a + b)$
  - $A = A \cup \{a + b\}$ *//Inserir (a + b) no conjuto A*
- return *custo_total*

Complexidade: $O(n * T(n))$, onde $T(n)$ é o tempo para procurar e remover os dois menores números

Algoritmo Guloso2:

- Ordenar de forma crescente o conjunto de n números: $\{a_1, a_2, ..., a_n\}$
- $custo\_total = soma = a_1 + a_2$
- Para $i = 3$ até n faça:
  - $soma = soma + a_i$
  - $custo\_total = custo\_total + soma$
- return $custo\_total$

Soma de Custo Mínimo

Algoritmo Guloso2:

- Ordenar de forma crescente o conjunto de n números: $\{a_1, a_2, ..., a_n\}$
- $custo\_total = soma = a_1 + a_2$
- Para $i = 3$ até n faça:
    - $soma = soma + a_i$
    - $custo\_total = custo\_total + soma$
- return $custo\_total$

Complexidade: $O(n \log n) + O(n) = O(n \log n)$

Seleção de Atividades

- Uma atividade a é definida por um intervalo de tempo $[s, f]$, onde s e f são, respectivamente, o início (*start*) e o término (*finish*) da atividade.
- Duas atividades $x = [s(x), f(x)]$ e $y = [s(y), f(y)]$ são **disjuntas** se elas **não se sobrepoēm**, ou seja,
  $f(x) \leq s(y)$ (x é anterior a y) ou $f(y) \leq s(x)$ (y é anterior a x)

Exemplos:

$x = [3, 6]$ e $y = [6, 12]$ são disjuntas.

$w = [3, 8]$ e $z = [5, 11]$ não são disjuntas.

Seleção de Atividades

- Dado um conjunto de n atividades $A = \{[s_1, f_1], [s_2, f_2], ..., [s_n, f_n]\}$, o problema da Seleção de Atividades consiste em determinar o **maior subconjunto de atividades disjuntas.**

- Dado um conjunto de n atividades $A = \{[s_1, f_1], [s_2, f_2], ..., [s_n, f_n]\}$, o problema da Seleção de Atividades consiste em determinar o **maior subconjunto de atividades disjuntas.**

Exemplo: Conjunto com $n = 11$ atividades
$A = \{[3, 8], [5, 7], [12, 14], [3, 5], [1, 4], [5, 9], [6, 10], [8, 11], [0, 6], [2, 13], [8, 12]\}$

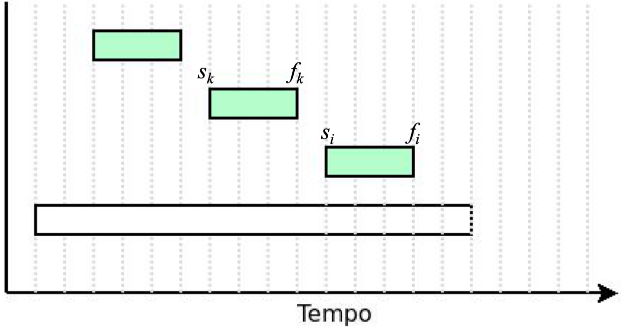
$S = \{[1, 4], [5, 7], [8, 11], [12, 14]\}$ é um subconjunto de 5 atividades disjuntas ($|S| = 5$).

S é máximo?

Seleção de Atividades

Estratégia Gulosa:

- Ordenar as atividades em **ordem crescente** do **tempo de término** f_i.
- Em seguida, selecionar atividades **disjuntas** a partir da primeira atividade (alocar por ordem ascendente de f_i).
- A **primeira atividade** da lista ordenada **sempre será escolhida**.
- A próxima atividade i será escolhida **se** seu tempo de inicio s_i for $\geq$ do tempo de término f_k da última atividade k já escolhida.



Seleção de Atividades

Exemplo:

- $A = \{[3,8],[5,7],[12,14],[3,5],[1,4],[5,9],[6,10],[8,11],$ $[0,6],[2,13],[8,12]\}$
- Ordem ascendente de f_i:
  $A = \{$**[1,4]**$,[3,5],[0,6],$**[5,7]**$,[3,8],[5,9],[6,10],$ **[8,11]**, $[8,12],[2,13],$**[12,14]**$\}$
- $S = \{[1,4],[5,7],[8,11],[12,14]\}$ é o maior subconjunto de atividades disjuntas.

Exemplo:

- $A = \{[3,8],[5,7],[12,14],[3,5],[1,4],[5,9],[6,10],[8,11],$ $[0,6],[2,13],[8,12]\}$
- Ordem ascendente de f_i:
  $A = \{$**[1,4]**$, [3,5],[0,6],$ **[5,7]**$, [3,8],[5,9],[6,10],$ **[8,11]**$,$ $[8,12],[2,13],$ **[12,14]**$\}$
- $S = \{[1,4],[5,7],[8,11],[12,14]\}$ é o maior subconjunto de atividades disjuntas.

A estratégia gulosa sempre determina a **solução ótima**.

Exercício

Escreva o algoritmo guloso para determinar o maior subconjunto de atividades disjuntas.

Código de Huffman

A codificação de Huffman é aplicada na compressão de dados.

- Dado um *texto* formado por um conjunto de n caracteres (ou símbolos) de um alfabeto $A = \{c_1, ..., c_n\}$.
- Para cada caractere c_i tem-se sua frequência (número de ocorrências no texto): $freq(c_i)$.
- Determinar uma codificação ou representação binária para cada caractere c_i do texto tal que o número total de bits usados seja mínimo.

Utilizar uma codificação binária de prefixo: o código de um caractere não pode ser prefixo de outro código. Assim, não haverá ambiguidade para reconhecer um caractere.

- Suponha que temos um arquivo de texto com as seguintes frequências dos caracteres.
- Na tabela são apresentadas duas codificações binárias de prefixo: **codificação fixa** e codificação **variável**.

	a	*b*	*c*	*d*	*e*	*f*
Frequência	45.000	13.000	12.000	16.000	9.000	5.000
Codificação **Fixa** (3 bits)	000	001	010	011	100	101
Codificação **Variável** (1- 4 bits)	0	101	100	111	1101	1100

	a	b	c	d	e	f
Frequência	45.000	13.000	12.000	16.000	9.000	5.000
Codificação **Fixa** (3 bits)	000	001	010	011	100	101
Codificação **Variável** (1- 4 bits)	0	101	100	111	1101	1100

- Note que temos um arquivo de texto com 100.000 caracteres.
- Usando a **codificação fixa**, gasta-se 300.000 bits para armazenar o arquivo.
- Se usarmos a **codificação variável** gastaríamos: $(45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) \times 1.000 = 224.000$ bits (25% de economia)
- Como determinar a codificação ótima (com número total de bits mínimo)?

Código de Huffman

David A. Huffman (1952), propôs um algoritmo baseado em uma estratégia gulosa.

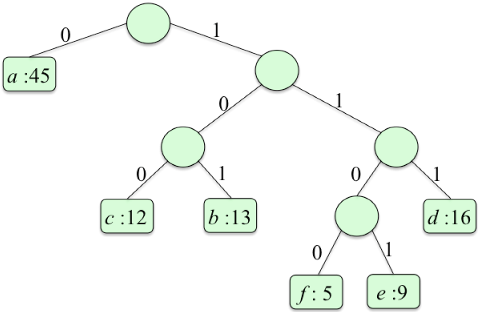
Codificação de Huffman (estratégia gulosa): Para os caracteres com **maior frequência** usar **códigos menores** (i.e. com menor número de bits), enquanto os caracteres com **menor frequência** usar **códigos maiores**.

Código de Huffman

Para representar a codificação de Huffman é usada uma **árvore binária** cujas **folhas** indicam os **caracteres** do alfabeto e cada **aresta** é rotulada com **0** (esquerda) ou **1** (direta).

Para determinar o código de um caractere, percorrer do nó folha até a raiz.

caractere	código
a	0
b	101
c	100
d	111
e	1101
f	1100



Código de Huffman

Entrada: $A = \{a, b, c, d, e, f\}$ com n caracteres e frequência de ocorrência dos caracteres no texto.

	a	b	c	d	e	f
Frequência	45.000	13.000	12.000	16.000	9.000	5.000

Algoritmo guloso para construir a árvore de Huffman

- Iniciar com *n* sub-árvores, cada um contendo um único nó. Cada nó contém o caractere e sua frequência.

$a:45$ $b:13$ $c:12$ $d:16$ $e:9$ $f:5$

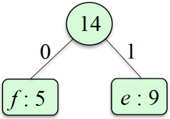
Algoritmo guloso para construir a árvore de Huffman

- Iniciar com *n* sub-árvores, cada um contendo um único nó. Cada nó contém o caractere e sua frequência.

$a:45$ $b:13$ $c:12$ $d:16$ $e:9$ $f:5$

- **Unir** as duas sub-árvores com as **menores frequências**, criando uma nova árvore com raiz contendo como frequência a **soma das frequências** das duas sub-árvores.

$a:45$ $b:13$ $c:12$ $d:16$

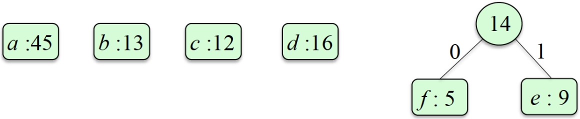


Código de Huffman

Algoritmo guloso para construir a árvore de Huffman

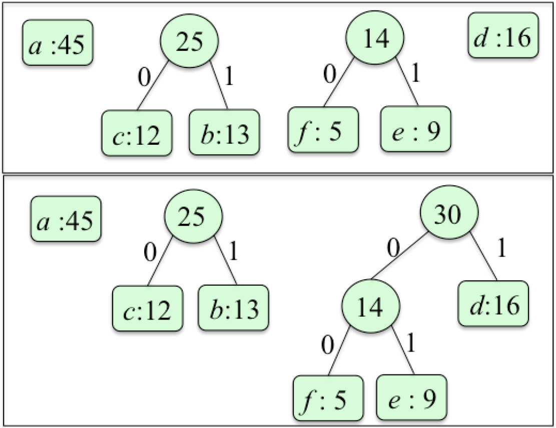
- Iniciar com *n* sub-árvores, cada um contendo um único nó. Cada nó contém o caractere e sua frequência.

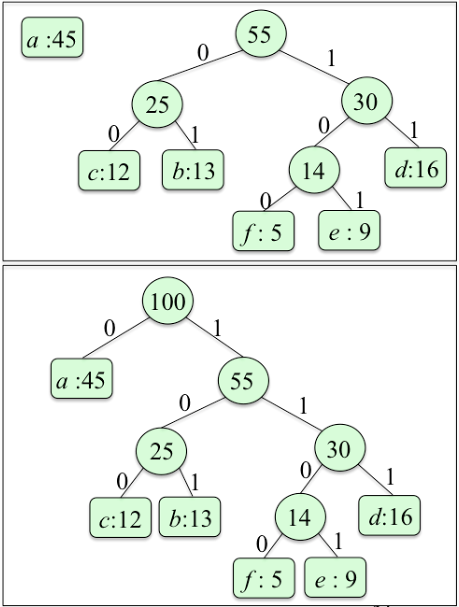
- **Unir** as duas sub-árvores com as **menores frequências**, criando uma nova árvore com raiz contendo como frequência a **soma das frequências** das duas sub-árvores.



- Repita a união até obter uma única árvore

Código de Huffman




a :45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f: 5
e : 9
100
0
1
a :45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f: 5
e : 9

AlgoritmoHuffman (Arvore A[], *n*){

Construir uma fila de prioridade *Q* com os *n* elementos de *A*;
Para $i = 1$ até $n - 1${

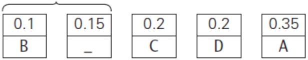
```
    x = ExtraiMinHeap(Q);
    y = ExtraiMinHeap(Q);
    z = CriaNovoNó();
    z.esq = x; x.pai = z;
    z.dir = y; y.pai = z;
    z.freq = x.freq + y.freq;
    InsereHeap(z, Q);
}
H = extraiMinHeap(Q);
H.pai = NULL;
}
```

Árvore de Huffman

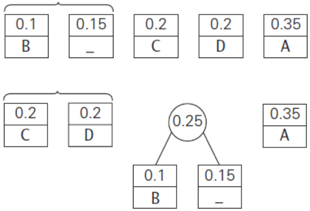
Exemplo 2: Determinar a codificação ótima para os caracteres:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

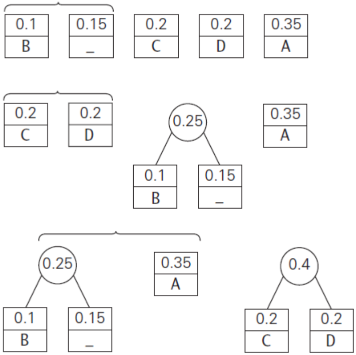
Árvore de Huffman



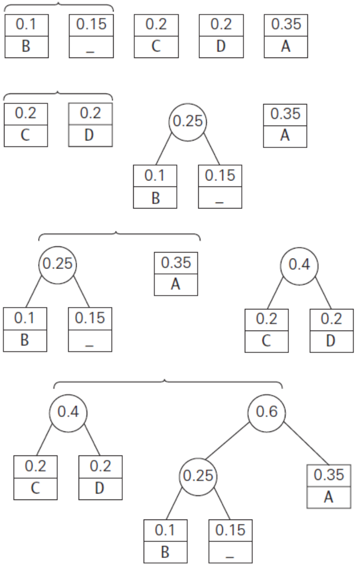
Árvore de Huffman



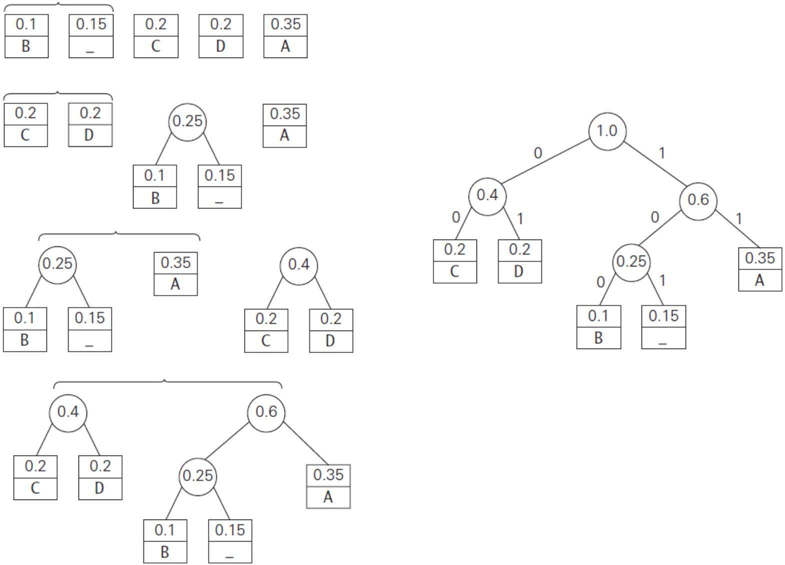
Árvore de Huffman



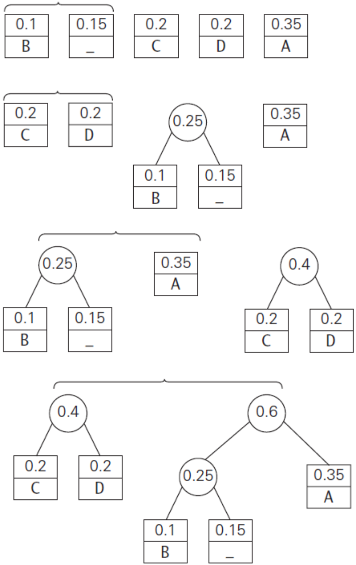
Árvore de Huffman

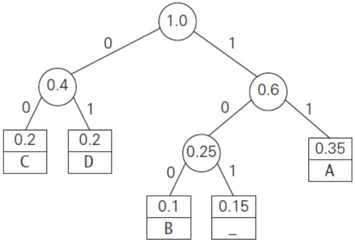


Árvore de Huffman



Árvore de Huffman





The resulting codewords are as follows:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

Problema da Árvore Geradora Mínima

Seja $G = (V, E)$ um grafo conectado, não direcionado e ponderado, onde $w(u, v)$ é uma função especificando o custo de conectar u e v ("custo da aresta"). Encontrar o subconjuto de arestas $T \subseteq E$ que conecte todos os vértices do grafo com o menor custo total, ou seja, que minimize a seguinte função:

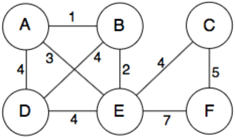
$$w(T) = \sum_{(u,v) \in T} w(u, v) \, .$$

Árvore Geradora Mínima

Problema da Árvore Geradora Mínima

Seja $G = (V, E)$ um grafo conectado, não direcionado e ponderado, onde $w(u, v)$ é uma função especificando o custo de conectar u e v ("custo da aresta"). Encontrar o subconjuto de arestas $T \subseteq E$ que conecte todos os vértices do grafo com o menor custo total, ou seja, que minimize a seguinte função:

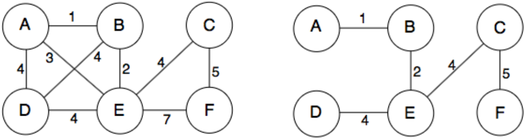
$$w(T) = \sum_{(u,v)\in T} w(u, v)\,.$$



Problema da Árvore Geradora Mínima

Seja $G = (V, E)$ um grafo conectado, não direcionado e ponderado, onde $w(u, v)$ é uma função especificando o custo de conectar u e v ("custo da aresta"). Encontrar o subconjuto de arestas $T \subseteq E$ que conecte todos os vértices do grafo com o menor custo total, ou seja, que minimize a seguinte função:

$$w(T) = \sum_{(u,v) \in T} w(u, v)\,.$$

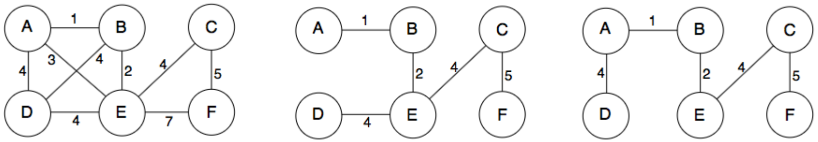


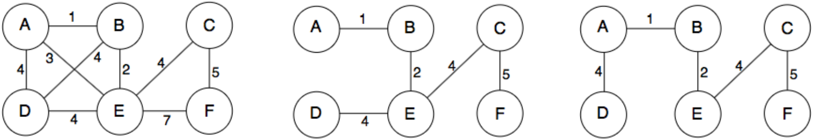
Árvore Geradora Mínima

Problema da Árvore Geradora Mínima

Seja $G = (V, E)$ um grafo conectado, não direcionado e ponderado, onde $w(u, v)$ é uma função especificando o custo de conectar u e v ("custo da aresta"). Encontrar o subconjuto de arestas $T \subseteq E$ que conecte todos os vértices do grafo com o menor custo total, ou seja, que minimize a seguinte função:

$$w(T) = \sum_{(u,v) \in T} w(u, v) .$$





O resultado é:

- um sub-grafo gerador;
- uma árvore (conectado e acíclico);
- possui $n - 1$ arestas para um grafo com n vértices.

Algoritmo de **Prim**

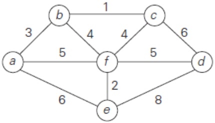
- Comece com uma árvore T_1 consistindo de um vértice (qualquer um);
  Expandir a árvore acrescentando um vértice por vez até produzir uma AGM T_n (árvore com n vértices).

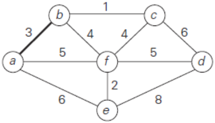
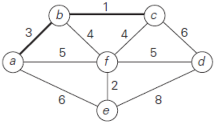
Algoritmo de **Prim**

- Comece com uma árvore T_1 consistindo de um vértice (qualquer um);
  Expandir a árvore acrescentando um vértice por vez até produzir uma AGM T_n (árvore com n vértices).
- Em cada iteração é construido uma sub-árvore T_{i+1} a partir de T_i através da adição de um vértice.
  O vértice escolhido é un vértice que não esteja em T_i e que esteja o mais próximo dos vértices já em T_i (passo guloso).

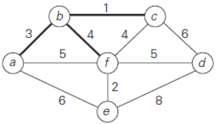
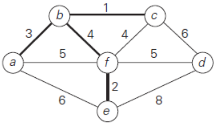
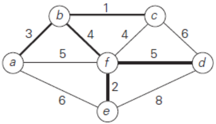
Algoritmo de **Prim**

- Comece com uma árvore T_1 consistindo de um vértice (qualquer um);
  Expandir a árvore acrescentando um vértice por vez até produzir uma AGM T_n (árvore com n vértices).
- Em cada iteração é construido uma sub-árvore T_{i+1} a partir de T_i através da adição de um vértice.
  O vértice escolhido é un vértice que não esteja em T_i e que esteja o mais próximo dos vértices já em T_i (passo guloso).
- Pare quando todos os vértices já tiverem sido incluídos na árvore, obtendo a AGM T_n.



Tree vertices	Remaining vertices	Illustration
a(−, −)	**b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c(b, 1)** d(−, ∞) e(a, 6) f(b, 4)	

Algoritmo de Prim

Tree vertices	Remaining vertices	Illustration
c(b, 1)	d(c, 6) e(a, 6) **f(b, 4)**	
f(b, 4)	d(f, 5) **e(f, 2)**	
e(f, 2)	**d(f, 5)**	
d(f, 5)		

Minimal spanning tree - Prim's algorithm

ALGORITHM *Prim(G)*

//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$
for $i \leftarrow 1$ **to** $|V| - 1$ **do**
    find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u)
    such that v is in V_T and u is in $V - V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
return E_T

Pseudocódigo retirado de: *Introduction to the design & analysis of algorithms / Anany Levitin*

Kruskal's algorithm

- Sort the edges in nondecreasing order of lengths.

Kruskal's algorithm

- Sort the edges in nondecreasing order of lengths.
- "Grow" tree one edge at a time to produce MST through a series of expanding **forests** $F_1, F_2, \ldots, F_{n-1}$.

Kruskal's algorithm

- Sort the edges in nondecreasing order of lengths.
- "Grow" tree one edge at a time to produce MST through a series of expanding **forests** $F_1, F_2, \ldots, F_{n-1}$.
- On each iteration, add the next edge on the sorted list unless this would create a cycle (If it would, skip the edge).

Kruskal's algorithm

- Sort the edges in nondecreasing order of lengths.
- "Grow" tree one edge at a time to produce MST through a series of expanding **forests** $F_1, F_2, \ldots, F_{n-1}$.
- On each iteration, add the next edge on the sorted list unless this would create a cycle (If it would, skip the edge).
- **Greedy** because always takes the **smallest-length** edge.

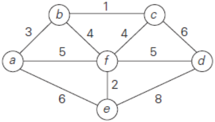
Minimal spanning tree - Kruskal's algorithm

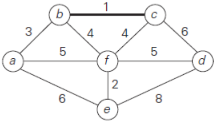
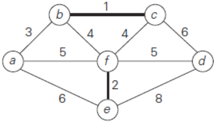
ALGORITHM *Kruskal(G)*

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights $w(e_{i_1}) \leq \cdots \leq w(e_{i_{|E|}})$
$E_T \leftarrow \varnothing$; $\quad ecounter \leftarrow 0$ //initialize the set of tree edges and its size
$k \leftarrow 0$ //initialize the number of processed edges
while $ecounter < |V| - 1$ **do**
 $k \leftarrow k + 1$
 if $E_T \cup \{e_{i_k}\}$ is acyclic
  $E_T \leftarrow E_T \cup \{e_{i_k}\}$; $\quad ecounter \leftarrow ecounter + 1$
return E_T

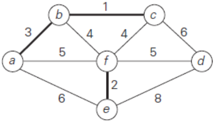
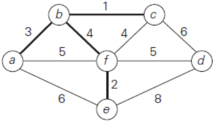
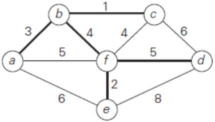
Pseudocódigo retirado de: *Introduction to the design & analysis of algorithms / Anany Levitin*

Minimal spanning tree - Kruskal's algorithm



Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	

Minimal spanning tree - Kruskal's algorithm

Tree edges	Sorted list of edges	Illustration
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ab 3	bc ef ab **bf** cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bf 4	bc ef ab bf cf af **df** ae cd de 1 2 3 4 4 5 5 6 6 8	
df 5		

Minimal spanning tree - Kruskal's algorithm

- Uma implementação eficiente para a detecção de ciclos no algoritmo de Kruskal usa a estrutura de dados Conjuntos Disjuntos (**Union-Find**).
- Como o algoritmo de Kruskal determina várias subárvores (floresta), então cada conjunto contém os vértices de uma subárvore.
- Inicialmente são construídos n **subárvores**, ou seja, n **conjuntos disjuntos**, cada um contendo apenas um vértice.
- Cada conjunto é identificado por um vértice, ou seja, cada conjunto possui um vértice representante chamado **raiz do conjunto** (**apontador** do conjunto).

- Uma implementação eficiente para a detecção de ciclos no algoritmo de Kruskal usa a estrutura de dados Conjuntos Disjuntos (**Union-Find**).
- Como o algoritmo de Kruskal determina várias subárvores (floresta), então cada conjunto contém os vértices de uma subárvore.
- Inicialmente são construídos *n* **subárvores**, ou seja, *n* **conjuntos disjuntos**, cada um contendo apenas um vértice.
- Cada conjunto é identificado por um vértice, ou seja, cada conjunto possui um vértice representante chamado **raiz do conjunto** (**apontador** do conjunto).
- A cada passo, dois conjuntos são unidos (o número de conjuntos diminui).
- O algoritmo finaliza quando se obtém apenas um conjunto (uma única árvore).

Minimal spanning tree - Kruskal's algorithm

O algoritmo de Kruskal utiliza as seguintes operações:

- Make_Sets( ):
  - Inicializa os n conjuntos.
  - Cada conjunto conterá um único vértice, ou seja, define-se um conjunto para cada vértice.
  - A raiz de cada conjunto é o único vértice do conjunto.
- Find_Set(u): .
  - Procura o conjunto que contem o vértice u e retorna a raiz desse conjunto
  - Note que, dois vértices u e v estarão no mesmo conjunto (ou na mesma árvore) se Find_Set(u) = Find_Set(v).
  - Se os vértices u e v estão na mesma árvore, a adição da aresta (u, v) formará um ciclo.
- Union(u, v):
  - Esta operação faz a união dos conjuntos que contêm, respectivamente, os vértices u e v.
  - A união das árvores que contem, respectivamente, os vértices u e v é feita adicionando a aresta (u, v).

Minimal spanning tree - Kruskal's algorithm

Kruskal (A: conjunto de arestas, n: número vértices):

$E_T = \varnothing$; //

Ordenar(A) //arestas ordem crescente de custo;

Make_Sets(n)

$k = 0$

Equanto $k < |A|$ e $|E_T| < n\text{-}1$:

(u, v) = escolhe a k-ésima aresta de A;

$k = k + 1$

Se **Find_Set**(u) != **Find_Set**(v):

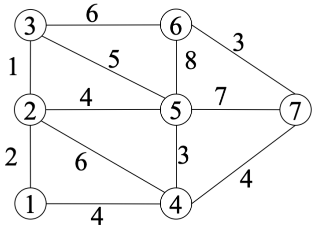
$E_T = E_T \cup (u,v)$;

Union(u,v);

return E_T;

Minimal spanning tree - Kruskal's algorithm

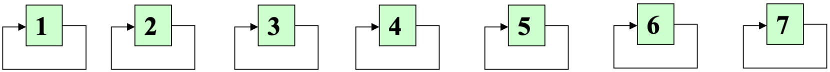
Exemplo:



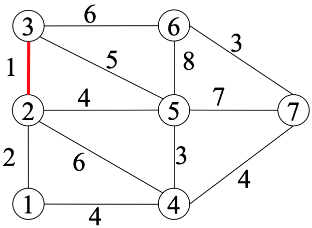
Arestas ordenadas:

A = {(2,3), (1,2), (4,5), (6,7), (1,4), (2,5), (4,7), (3,5), (2,4), ...,(5,6) }

Make_Sets: forma 7 conjuntos disjuntos



Exemplo:

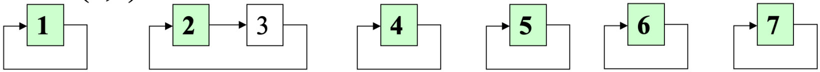


A = {(2,3), (1,2), (4,5), (6,7), (1,4), (2,5), (4,7), (3,5), (2,4), ...,(5,6) }

Aresta (2,3)

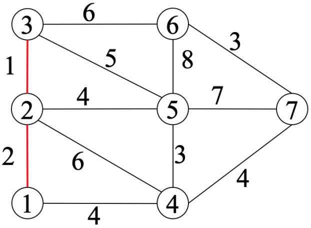
Os vértices 2 e 3 estão em conjuntos diferentes.

Union(2,3):



Union consiste em mudar as raízes dos elementos do menor conjunto.

Exemplo:

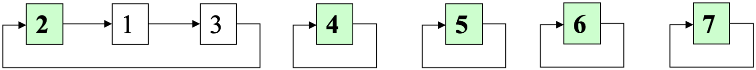


A = {(2,3), (1,2), (4,5), (6,7), (1,4), (2,5), (4,7), (3,5), (2,4), ...,(5,6) }

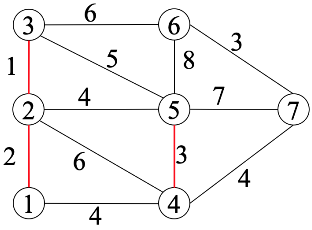
Aresta (1,2)

Os vértices 1 e 2 estão em conjuntos diferentes.

Union(1,2):



Exemplo:

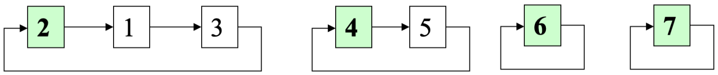


A = {(2,3), (1,2), (4,5), (6,7), (1,4), (2,5), (4,7), (3,5), (2,4), ...,(5,6) }

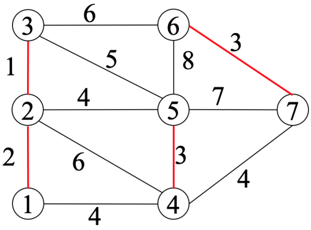
Aresta (4,5)

Os vértices 4 e 5 estão em conjuntos diferentes.

Union(4,5):



Exemplo:

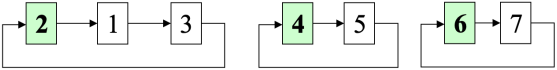


A = {(2,3), (1,2), (4,5), (6,7), (1,4), (2,5), (4,7), (3,5), (2,4), ...,(5,6) }

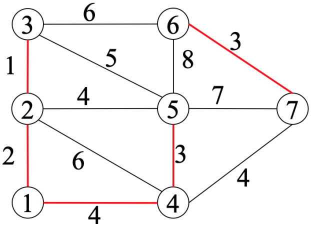
Aresta (6,7)

Os vértices 6 e 7 estão em conjuntos diferentes.

Union(6,7):



Exemplo:

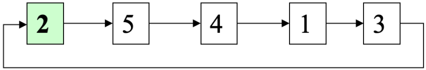


A = {(2,3), (1,2), (4,5), (6,7), (1,4), (2,5), (4,7), (3,5), (2,4), ...,(5,6) }

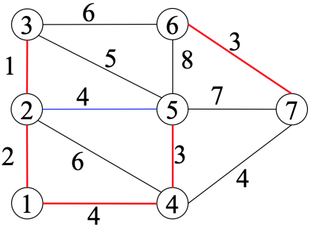
Aresta (1,4)

Os vértices 1 e 4 estão em conjuntos diferentes.

Union(1,4):



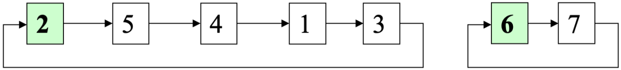
Exemplo:



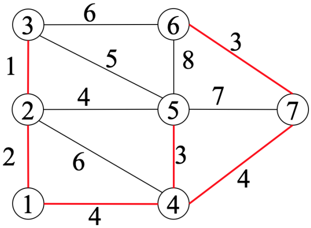
A = {(2,3), (1,2), (4,5), (6,7), (1,4), (2,5), (4,7), (3,5), (2,4), ...,(5,6) }

Aresta (2,5)

Os vértices 2 e 5 estão no mesmo conjunto.



Exemplo:

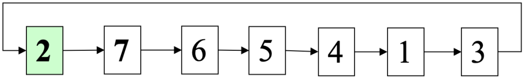


A = {(2,3), (1,2), (4,5), (6,7), (1,4), (2,5), (4,7), (3,5), (2,4), ...,(5,6) }

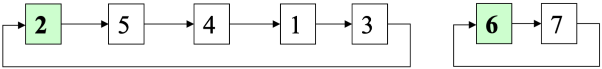
Aresta (4,7)

Os vértices 4 e 7 estão em conjuntos diferentes.

Union(4,7):

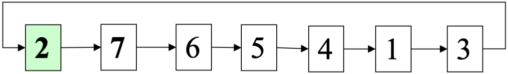


Minimal spanning tree - Kruskal's algorithm



	1	2	3	4	5	6	7
prox	**3**	**5**	**2**	**1**	**4**	**7**	**6**
raiz	**2**	**2**	**2**	**2**	**2**	**6**	**6**

Union:



	1	2	3	4	5	6	7
prox	**3**	**7**	**2**	**1**	**4**	**5**	**6**
raiz	**2**	**2**	**2**	**2**	**2**	**2**	**2**

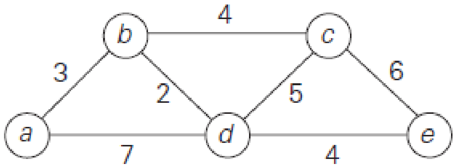
Shortest paths

- For a given vertex called the **source** in a weighted connected graph, find **shortest paths** to all its other vertices.
- We are not interested in a single path that starts at the source and visits all the other visits (this is the TSP!)
- We ask for a **family of paths**, each leading from the source to a different vertex

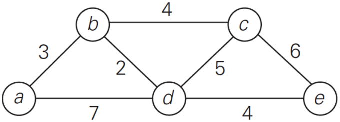
ALGORITHM *Dijkstra*(G, s)

//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph $G = \langle V, E \rangle$ with nonnegative weights
// and its vertex s
//Output: The length d_v of a shortest path from s to v
// and its penultimate vertex p_v for every vertex v in V
Initialize(Q) //initialize priority queue to empty
for every vertex v in V
  $d_v \leftarrow \infty$; $p_v \leftarrow$ **null**
  Insert(Q, v, d_v) //initialize vertex priority in the priority queue
$d_s \leftarrow 0$; *Decrease*(Q, s, d_s) //update priority of s with d_s
$V_T \leftarrow \varnothing$
for $i \leftarrow 0$ **to** $|V| - 1$ **do**
  $u^* \leftarrow$ *DeleteMin*(Q) //delete the minimum priority element
  $V_T \leftarrow V_T \cup \{u^*\}$
  for every vertex u in $V - V_T$ that is adjacent to u^* **do**
    if $d_{u^*} + w(u^*, u) < d_u$
      $d_u \leftarrow d_{u^*} + w(u^*, u)$; $p_u \leftarrow u^*$
      Decrease(Q, u, d_u)

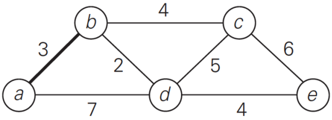
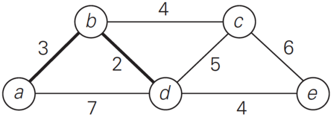
Shortest paths - Dijkstra's algorithm



Shortest paths - Dijkstra's algorithm

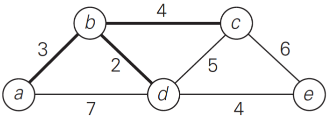


$v(p_v, d_v)$

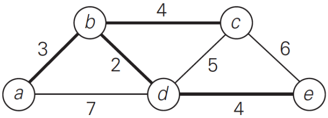
Tree vertices	Remaining vertices	Illustration
$a(-, 0)$	$\mathbf{b(a, 3)}$ $c(-, \infty)$ $d(a, 7)$ $e(-, \infty)$	
$b(a, 3)$	$c(b, 3+4)$ $\mathbf{d(b, 3+2)}$ $e(-, \infty)$	

Shortest paths - Dijkstra's algorithm

d(b, 5) **c**(**b**, **7**) e(d, 5 + 4)



c(b, 7) **e**(**d**, **9**)



e(d, 9)

The shortest paths (identified by following nonnumeric labels backward from a destination vertex in the left column to the source) and their lengths (given by numeric labels of the tree vertices) are as follows:

from a to b :	$a - b$	of length 3
from a to d :	$a - b - d$	of length 5
from a to c :	$a - b - c$	of length 7
from a to e :	$a - b - d - e$	of length 9