Dynamic Programming I

Franklin Oliveira; Lucas Ribeiro

O que é Dynamic Programming?

É um método para solução (otimização) de problemas com estrutura recursiva.

“Programming”, nesse contexto, não se refere exatamente a “escrever código”.

O que é Dynamic Programming?

Podemos entender como um paradigma de otimização para problemas que possuem **subestrutura ótima**.

O que é Dynamic Programming?

Podemos entender como um paradigma de otimização para problemas que possuem **subestrutura ótima**.

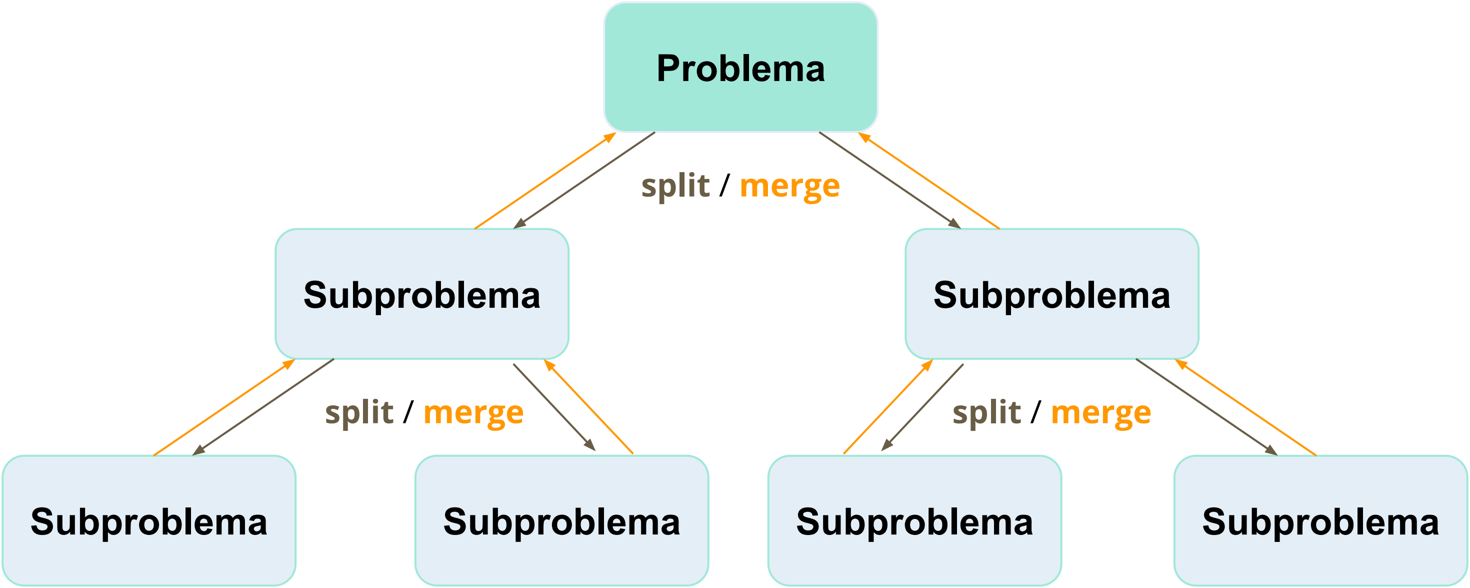
O que é Dynamic Programming?

Podemos entender como um paradigma de otimização para problemas que possuem **subestrutura ótima**.

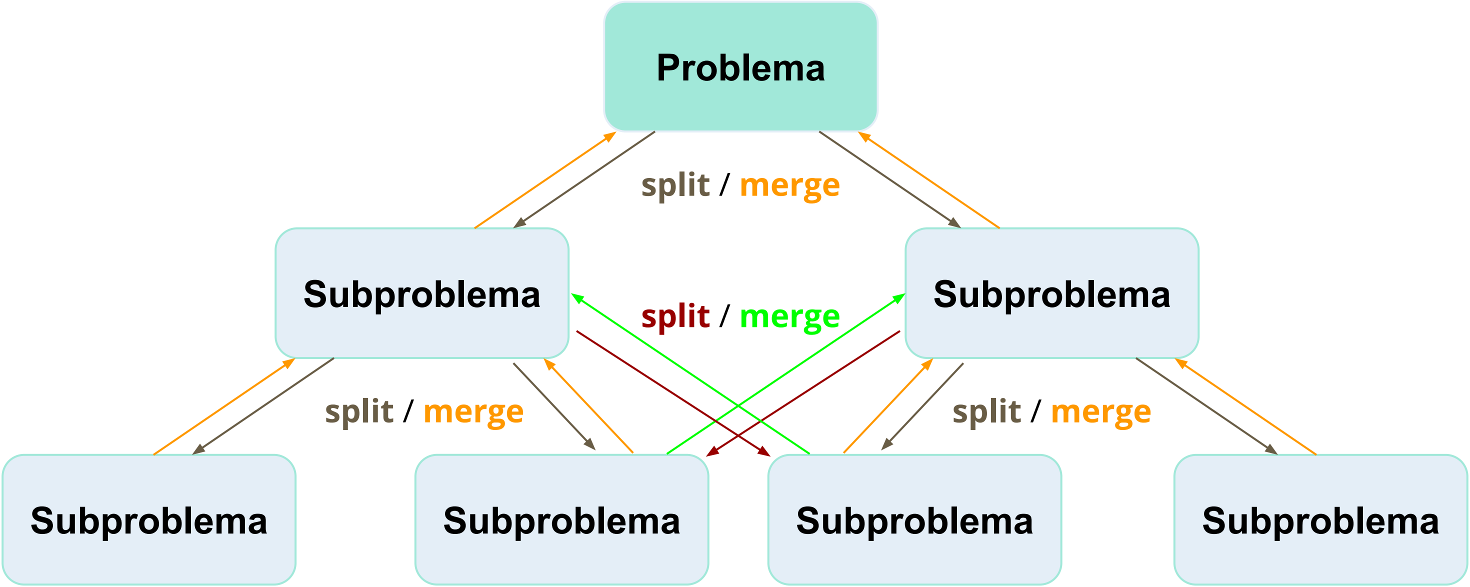
Problemas cuja solução ótima pode ser obtida recursivamente a partir da solução ótima de problemas menores.

Já vimos isso em algum lugar...

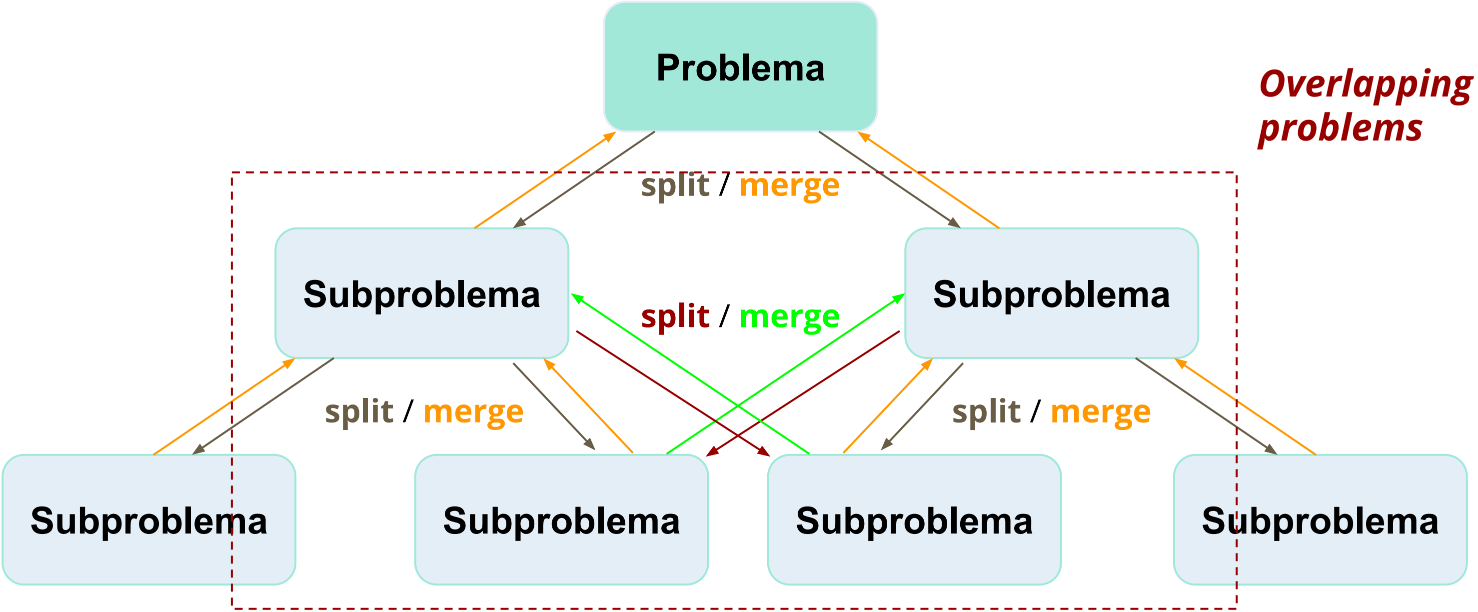
Divide and Conquer



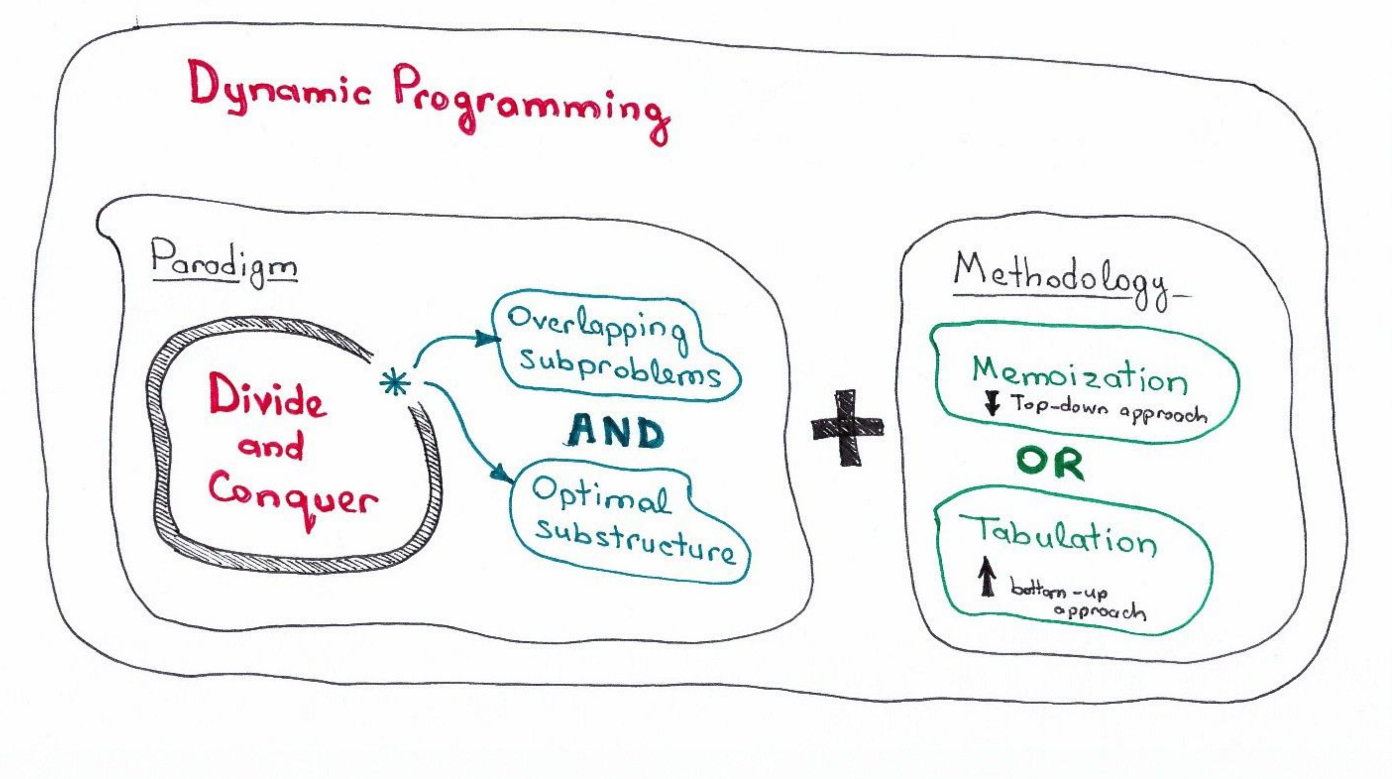
Dynamic Programming



Dynamic Programming



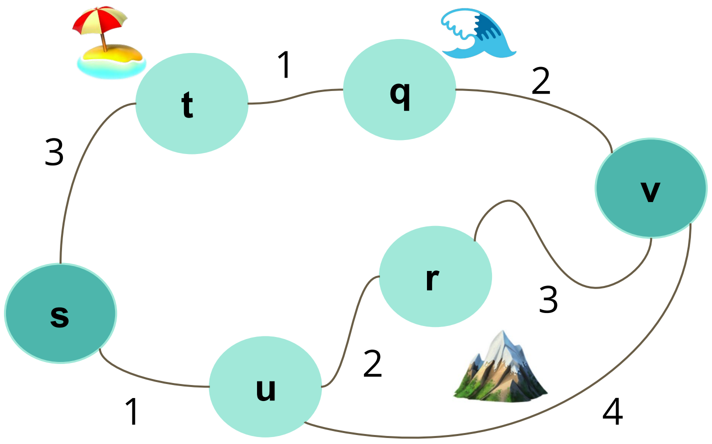
Dynamic Programming vs Divide and Conquer



OK... Mas o que eu faço com isso?

Dynamic programming é uma boa alternativa para resolver problemas com as seguintes características:

1. Subestrutura ótima, i.e., permite formulação recursiva.

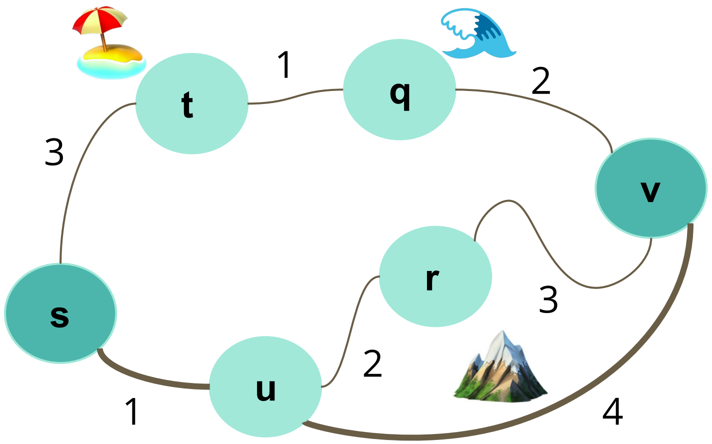


O menor caminho de ***s*** a ***v*** consiste em um menor caminho de ***s*** a ***u*** e um menor caminho de ***u*** a ***v***.

OK... Mas o que eu faço com isso?

Dynamic programming é uma boa alternativa para resolver problemas com as seguintes características:

1. Subestrutura ótima, i.e., permite formulação recursiva.



O menor caminho de ***s*** a ***v*** consiste em um menor caminho de ***s*** a ***u*** e um menor caminho de ***u*** a ***v***.

<u>A solução ótima é composta pela solução ótima de problemas menores e independentes.</u>

OK... Mas o que eu faço com isso?

Dynamic programming é uma boa alternativa para resolver problemas com as seguintes características:

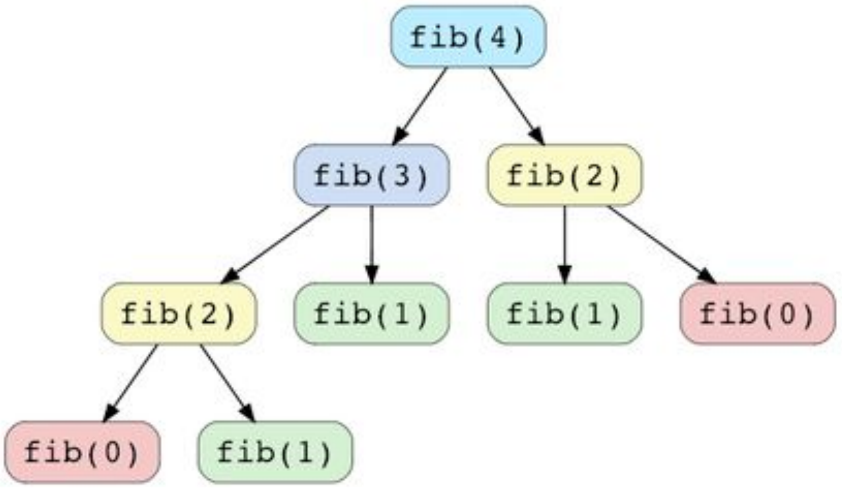
1. Subestrutura ótima, i.e., permite formulação recursiva.

2. *Overlapping subproblems*, i.e., permite armazenar respostas pré-computadas (economiza tempo na recursão)

EXEMPLOS, POR FAVOR!!!

Exemplo de Problemas Sobrepostos

Sequência de Fibonacci



Exemplo de Aplicação de Dynamic Programming

Suponha que, por algum motivo, queremos encontrar o n-ésimo termo da sequência de Fibonacci.

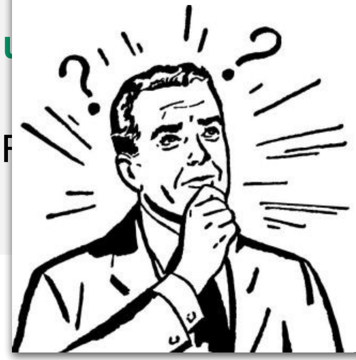
```
int Fibonacci(int n) {
    if (n <= 1) {
        return n;
    }
    return Fibonacci(n-1) + Fibonacci(n-2);
}
```

Referência

Exemplo de Aplicação de Dynamic Programming

Suponha que, por algum motivo, queremos encontrar o n-ésimo termo da sequência de Fibonacci.

```
int Fibonacc
    if (n <=
        retu
    }
    return F
}
```



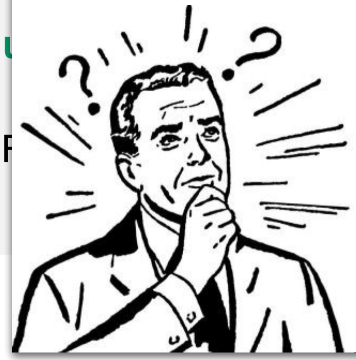
Qual a complexidade desse algoritmo?

Referência

Exemplo de Aplicação de Dynamic Programming

Suponha que, por algum motivo, queremos encontrar o n-ésimo termo da sequência de Fibonacci.

```
int Fibonacc
    if (n <=
        retu
    }
    return F
}
```



Qual a complexidade desse algoritmo?

R: $O(2^n)$

Referência

Exemplo de Aplicação de Dynamic Programming

Podemos reduzir esse tempo computacional com programação dinâmica.

```
int Dynamic_Fibonacci_Bottom_Up(int n) {
    f[0] = 0;
    f[1] = 1;
    for (int i = 2; i <= n; i++) {
        f[i] = f[i-1] + f[i-2];
    }
    return f[n];
```

Referência

Exemplo de Aplicação de Dynamic Programming

Podemos reduzir esse tempo computacional com programação dinâmica.

```
int Dynamic_Fibonacci_Bottom_Up(int n) {
    f[0] = 0;
    f[1] = 1;
    for (int i = 2; i <= n; i++) {
        f[i] = f[i-1] + f[i-2];
    }
    return f[n];
```

Complexidade: O(n)

Referência

Maneiras de implementar

Basicamente, vamos criar uma tabela ***T*** com as soluções ótimas dos sub-problemas. Mas como vamos preenchê-la?

1. **Bottom-Up:** começa armazenando as soluções dos menores sub-problemas. Então, armazena-se as soluções dos problemas maiores usando a estrutura ótima recursiva.

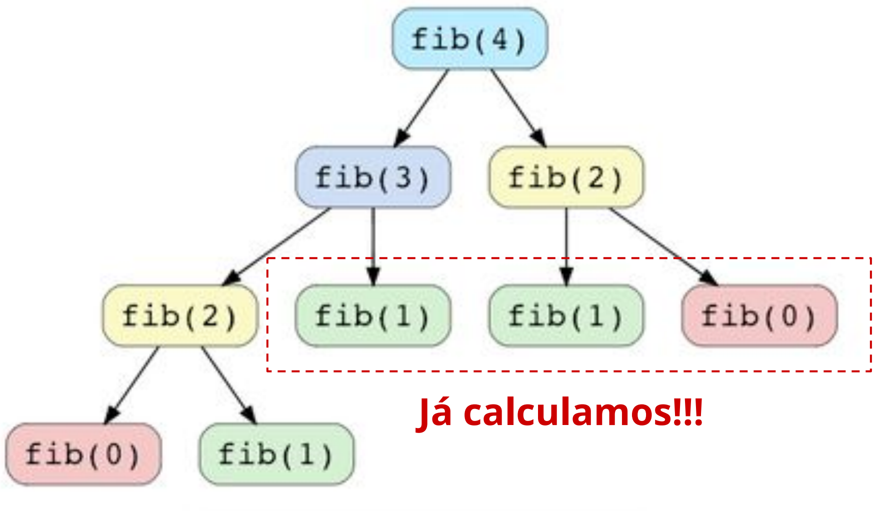
2. **Top-Down:** a solução do problema inteiro é computada recursivamente. A cada chamada recursiva, vamos buscar ou preencher a resposta na tabela.

Boa notícia!

As duas formas são perfeitamente equivalentes.

Vamos ver então como ficaria a implementação do exemplo...

Sequência de Fibonacci



Step	Value
fib(0)	1
fib(1)	1
fib(2)	2
fib(3)	3
fib(4)	5

Sequência de Fibonacci: Bottom-Up

Foi o exemplo que vimos anteriormente. Relembrando...

```
int Dynamic_Fibonacci_Bottom_Up(int n) {
    f[0] = 0;
    f[1] = 1;
    for (int i = 2; i <= n; i++) {
        f[i] = f[i-1] + f[i-2];
    }
    return f[n];
```

Complexidade: $O(n)$

Sequência de Fibonacci: Top-Down

Aplicando *memoization*...

```
int Dynamic_Fibonacci_TD(int n) {
    if (f[n] != 0) {
        return f[n];
    }
    if (n == 0 or n == 1) {
        return 1;
    }
    f[n] = Dynamic_Fibonacci_TD(n-1) + Dynamic_Fibonacci_TD(n-2);
    return f[n];
```

Complexidade: $O(n)$

Passo a passo:

1. Estruturar o problema (identificar características).

2. Caracterizar a estrutura da solução ótima (em função das soluções dos sub-problemas).

3. Calcular, recursivamente, a solução ótima dos sub-problemas (*Top-Down* ou *Bottom-Up*).

4. Construir a solução ótima a partir das informações computadas.

Bellman-Ford

Bellman-Ford

- Assim como o algoritmo de Dijkstra, o algoritmo de Bellman-Ford resolve o problema de *single-source shortest path* para grafos com pesos.

Bellman-Ford

- Assim como o algoritmo de Dijkstra, o algoritmo de Bellman-Ford resolve o problema de *single-source shortest path* para grafos com pesos.

- Dijkstra não tem garantia de sucesso com peso de arestas negativo.

Bellman-Ford

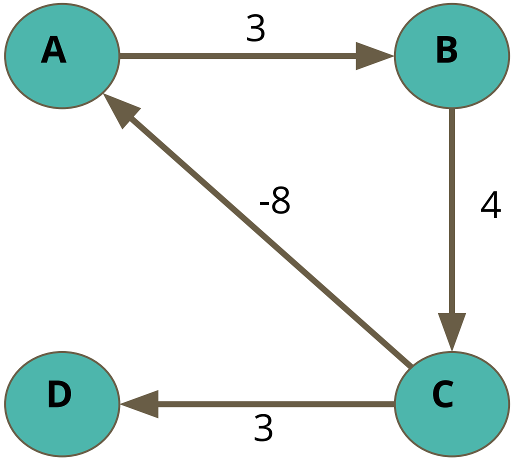
- Assim como o algoritmo de Dijkstra, o algoritmo de Bellman-Ford resolve o problema de *single-source shortest path* para grafos com pesos.

- Dijkstra não tem garantia de sucesso com peso de arestas negativo.

- Bellman-Ford sempre resolve o problema, desde que exista o menor caminho.

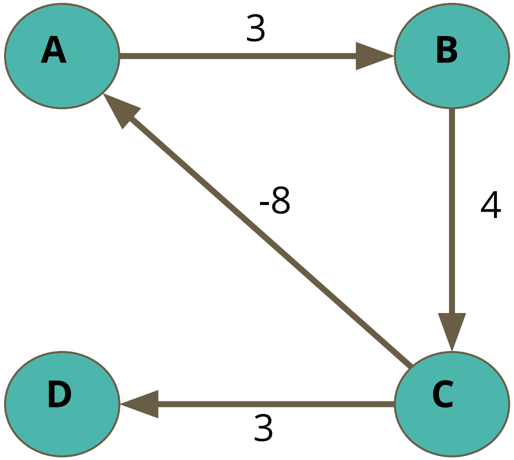
Quando não existe um menor caminho?

Considere o grafo abaixo.



Quando não existe um menor caminho?

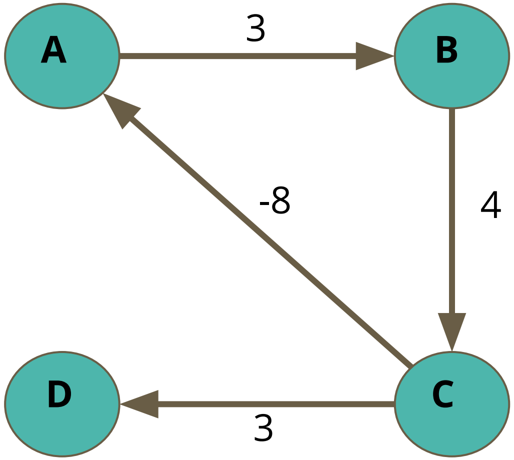
Considere o grafo abaixo.



- Existe um ciclo negativo (A-B-C-A), portanto não existe um menor caminho entre A e C, por exemplo.

Quando não existe um menor caminho?

Considere o grafo abaixo.

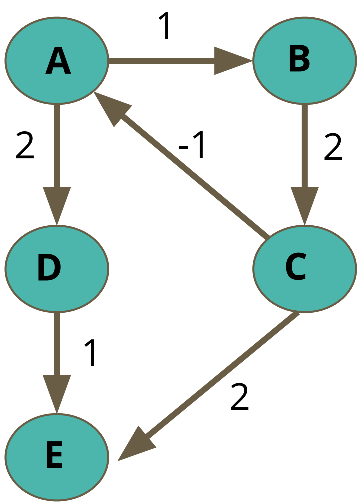


- Existe um ciclo negativo (A-B-C-A), portanto não existe um menor caminho entre A e C, por exemplo.

- Quando não existem ciclos negativos, sempre há um menor caminho.

Quando não existe um menor caminho?

Considere o grafo abaixo.



- Existe um ciclo negativo (A-B-C-A), portanto não existe um menor caminho entre A e C, por exemplo.

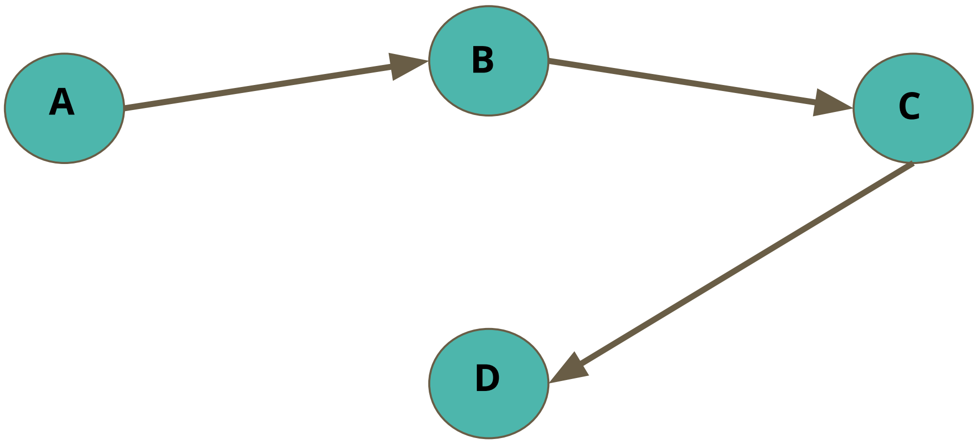
- Quando não existem ciclos negativos, sempre há um menor caminho (desde que exista um caminho entre os vértices).

Quando não existe um menor caminho?

- Um **caminho simples** em um grafo com **v** vértices contém no máximo **v-1** arestas, pois caso contivesse mais, haveria pelo menos um ciclo.

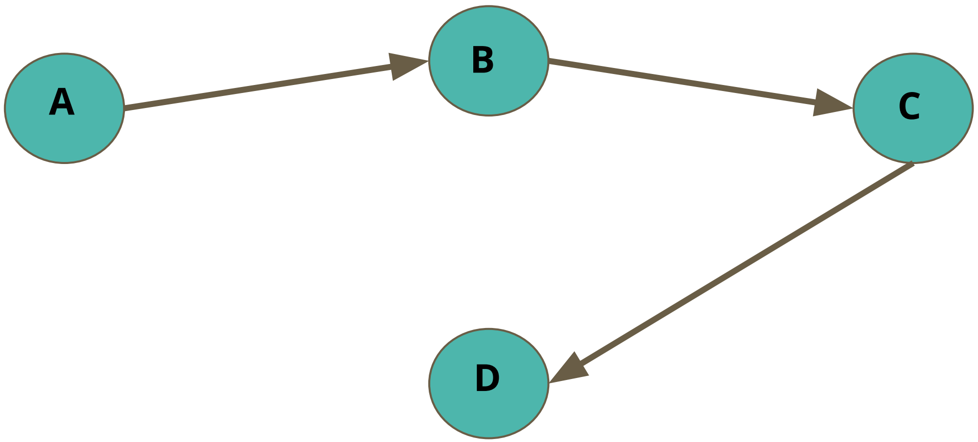
Quando não existe um menor caminho?

- Um **caminho simples** em um grafo com **v** vértices contém no máximo **v-1** arestas, pois caso contivesse mais, haveria pelo menos um ciclo.



Quando não existe um menor caminho?

- Um **caminho simples** em um grafo com **v** vértices contém no máximo **v-1** arestas, pois caso contivesse mais, haveria pelo menos um ciclo.



- Se não existem ciclos negativos, então qualquer ciclo aumentaria a distância, ao invés de diminuir.

Relaxamento

O algoritmo de Bellman-Ford faz uso de **relaxamento.**

Relaxamento

O algoritmo de Bellman-Ford faz uso de **relaxamento.**

Não esse tipo de relaxamento...

Relaxamento

O algoritmo de Bellman-Ford faz uso de **relaxamento.**

Seja **G = (V,E)** um grafo. Fixe um vértice **s ∈ V**.

Relaxamento

O algoritmo de Bellman-Ford faz uso de **relaxamento.**

Seja **G = (V,E)** um grafo. Fixe um vértice **s ∈ V**.

1. Defina **d[v]** como uma estimativa da distância entre **s** e **v.**

Relaxamento

O algoritmo de Bellman-Ford faz uso de **relaxamento.**

Seja **G = (V,E)** um grafo. Fixe um vértice **s ∈ V**.

1. Defina **d[v]** como uma estimativa da distância entre **s** e **v.**
2. **Relaxar** a aresta **(u,v)** significa (potencialmente) diminuir **d[v]** passando por **u**.

Relaxamento

O algoritmo de Bellman-Ford faz uso de **relaxamento.**

Seja **G = (V,E)** um grafo. Fixe um vértice **s ∈ V**.

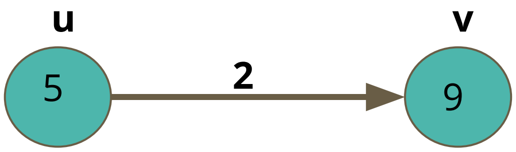
1. Defina **d[v]** como uma estimativa da distância entre **s** e **v.**
2. **Relaxar** a aresta **(u,v)** significa (potencialmente) diminuir **d[v]** passando por **u**.
3. **d[v] = min(d[v], d[u] + w(u,v))**

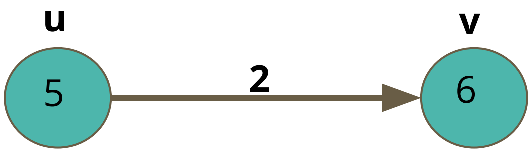
Relaxamento

O algoritmo de Bellman-Ford faz uso de **relaxamento.**

Seja **G = (V,E)** um grafo. Fixe um vértice **s ∈ V**.

1. Defina **d[v]** como uma estimativa da distância entre **s** e **v.**
2. **Relaxar** a aresta **(u,v)** significa (potencialmente) diminuir **d[v]** passando por **u**.
3. **d[v] = min(d[v], d[u] + w(u,v))**



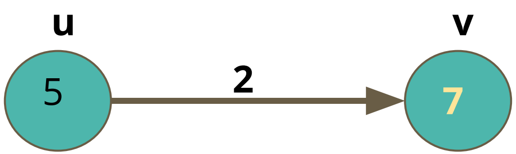


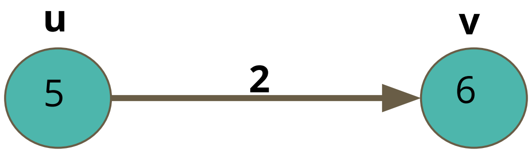
Relaxamento

O algoritmo de Bellman-Ford faz uso de **relaxamento.**

Seja **G = (V,E)** um grafo. Fixe um vértice **s ∊ V**.

1. Defina **d[v]** como uma estimativa da distância entre **s** e **v.**
2. **Relaxar** a aresta **(u,v)** significa (potencialmente) diminuir **d[v]** passando por **u**.
3. **d[v] = min(d[v], d[u] + w(u,v))**





O algoritmo de Bellman-Ford

- Utiliza sucessivamente relaxamento nas arestas do grafo. O algoritmo de Dijkstra relaxa cada aresta apenas uma vez, optando por seguir sempre o menor caminho.

O algoritmo de Bellman-Ford

- Utiliza sucessivamente relaxamento nas arestas do grafo. O algoritmo de Dijkstra relaxa cada aresta apenas uma vez, optando por seguir sempre o menor caminho.

1. Defina $\mathbf{d^k[v]}$ como uma estimativa da menor distância entre **s** e **v** passando por **k** arestas.

O algoritmo de Bellman-Ford

- Utiliza sucessivamente relaxamento nas arestas do grafo. O algoritmo de Dijkstra relaxa cada aresta apenas uma vez, optando por seguir sempre o menor caminho.

1. Defina $\mathbf{d^k[v]}$ como uma estimativa da menor distância entre **s** e **v** passando por **k** arestas.

2. Para cada $\mathbf{k = 0, \ldots, |V| - 1}$, o algoritmo relaxa todas as arestas **(u,v)** passando por **k** arestas.

O algoritmo de Bellman-Ford

```
algorithm bellman_ford(G,s):
    dᵏ = [] for k = 1 to |V|
    d⁰[v] = ∞ for v ≠ s
    d⁰[s] = 0
    for k = 1 to |V| - 1:
        for v in V:
            dᵏ[v] = min(dᵏ⁻¹[v], min(dᵏ⁻¹[a] + w(a,b)))
                                 (a,b) in E
    return(d|V|-1)
```

O algoritmo de Bellman-Ford

$\textbf{algorithm}\ \texttt{bellman\_ford}(G,s)$:

$\quad d^k = [\,]\ \textbf{for}\ k = 1\ \textbf{to}\ |V|$

$\quad d^0[v] = \infty\ \textbf{for}\ v \neq s$

$\quad d^0[s] = 0$

$\quad \textbf{for}\ k = 1\ \textbf{to}\ |V| - 1$:

$\qquad \textbf{for}\ v\ \textbf{in}\ V$:

$$d^k[v] = \min\left(d^{k-1}[v],\ \min_{(a,b)\ \text{in}\ E}\left(d^{k-1}[a] + w(a,b)\right)\right)$$

$\quad \textbf{return}(d^{|V|-1})$

Cria-se uma lista para armazenar as estimativas do menor caminho entre s e todos os outros vértices do grafo.

O algoritmo de Bellman-Ford

```
algorithm bellman_ford(G,s):
    d^k = [] for k = 1 to |V|
    d^0[v] = ∞ for v ≠ s
    d^0[s] = 0
    for k = 1 to |V| - 1:
        for v in V:
            d^k[v] = min(d^(k-1)[v], min          (d^(k-1)[a] + w(a,b)))
                                    (a,b) in E
    return(d^(|V|-1))
```

Cria-se uma lista para armazenar as estimativas do menor caminho entre s e todos os outros vértices do grafo.

Retorna uma lista com o menor caminho entre s e v, para todo v em V.

O algoritmo de Bellman-Ford

```
algorithm bellman_ford(G,s):
    dᵏ = [] for k = 1 to |V|
    d⁰[v] = ∞ for v ≠ s
    d⁰[s] = 0
    for k = 1 to |V| - 1:
        for v in V:
            dᵏ[v] = min(dᵏ⁻¹[v],  min       (dᵏ⁻¹[a] + w(a,b)))
                                 (a,b) in E
    return(d|V|-1)
```

Cria-se uma lista para armazenar as estimativas do menor caminho entre s e todos os outros vértices do grafo.

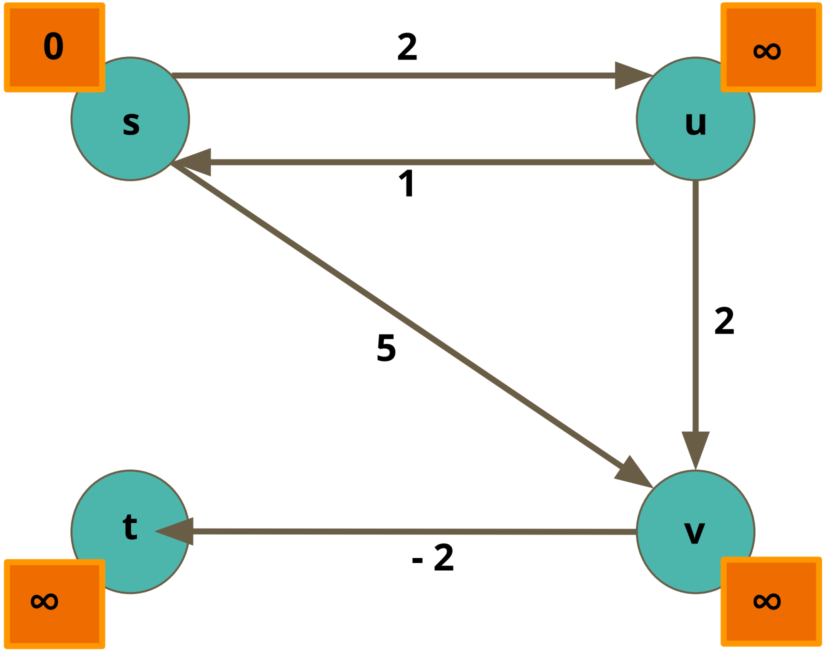
Retorna uma lista com o menor caminho entre s e v, para todo v em V.

Complexidade: $O(|V||E|)$

O algoritmo de Bellman-Ford

Um exemplo...

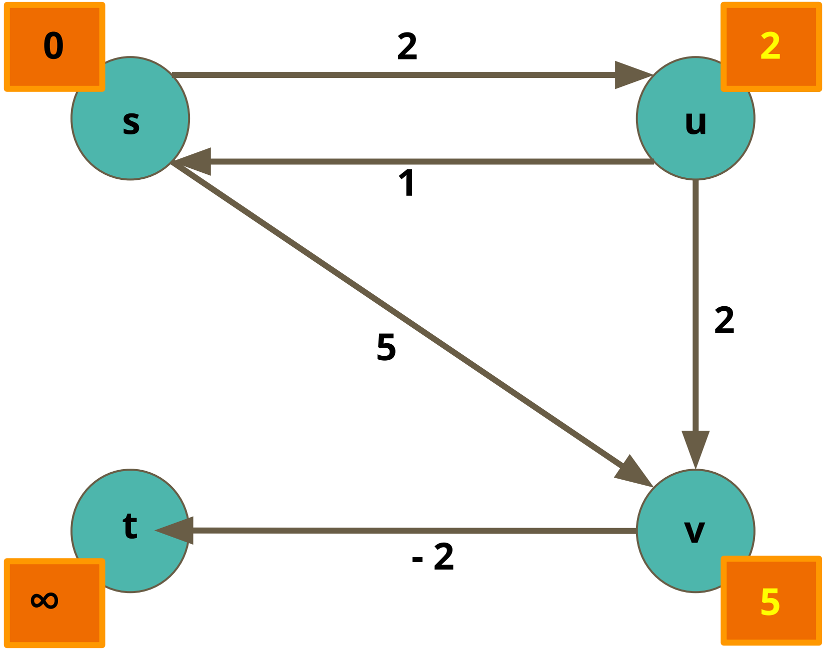
	s	u	v	t
d^0	0	∞	∞	∞
d^1				
d^2				
d^3				



O algoritmo de Bellman-Ford

Um exemplo...

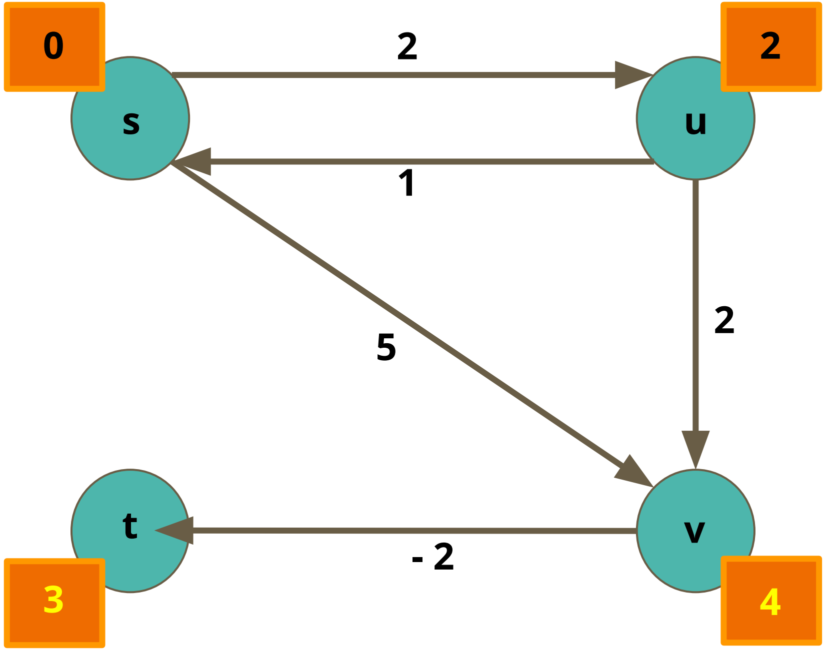
	s	u	v	t
d^0	0	∞	∞	∞
d^1	0	2	5	∞
d^2				
d^3				



O algoritmo de Bellman-Ford

Um exemplo...

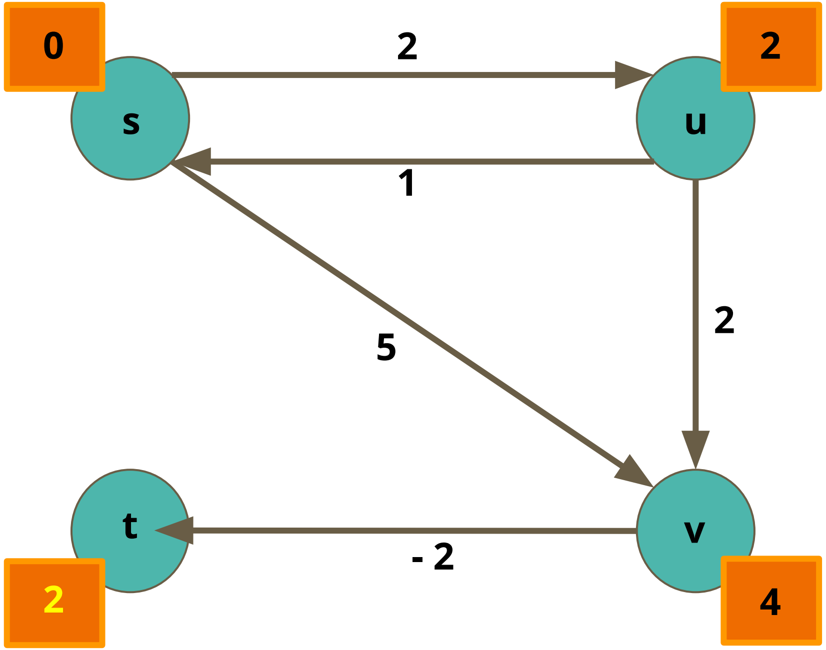
	s	u	v	t
d^0	0	∞	∞	∞
d^1	0	2	5	∞
d^2	0	2	4	3
d^3				



O algoritmo de Bellman-Ford

Um exemplo...

	s	u	v	t
d^0	0	∞	∞	∞
d^1	0	2	5	∞
d^2	0	2	4	3
d^3	0	2	4	2



Link Interessante...

https://www-m9.ma.tum.de/graph-algorithms/spp-bellman-ford/index_en.html

Bellman-Ford - Corretude

Lema: Sejam G = (V,E) um grafo com peso nas arestas e $\mathbf{s \in V}$. Então $\mathbf{d^{|V|-1}[u]}$ é o tamanho do menor caminho entre **s** e **u** com no máximo **|V|-1** arestas.

Bellman-Ford - Corretude

Lema: Sejam G = (V,E) um grafo com peso nas arestas e $\mathbf{s} \in \mathbf{V}$. Então $\mathbf{d^{|V|-1}[u]}$ é o tamanho do menor caminho entre **s** e **u** com no máximo **|V|-1** arestas.

Prova: Indução em **k**, o número de iterações no algoritmo. Para **k = 1,** o menor caminho entre **s** e **v** possui no mínimo **1** aresta, caso $\mathbf{s \neq v}$, portanto não existe caminho entre **s** e **v** com **0** arestas, logo $\mathbf{d^0[u] = \infty}$. Se $\mathbf{s = v}$, $\mathbf{d^0[s] = 0}$. Sendo assim, a condição é satisfeita.

Bellman-Ford - Corretude

Lema: Sejam G = (V,E) um grafo com peso nas arestas e **$s \in V$.** Então **$d^{|V|-1}[u]$** é o tamanho do menor caminho entre **s** e **u** com no máximo **$|V|-1$** arestas.

Prova: Indução em **k**, o número de iterações no algoritmo. Para **$k = 1$,** o menor caminho entre **s** e **v** possui no mínimo **1** aresta, caso **$s \neq v$,** portanto não existe caminho entre **s** e **v** com **0** arestas, logo **$d^0[u] = \infty$.** Se **$s = v$, $d^0[s] = 0$.** Sendo assim, a condição é satisfeita.

Suponha que, na iteração **k**, **$d^{k-1}[v]$** represente o custo do menor caminho entre **s** e **v** com no máximo **k-1** arestas.

Bellman-Ford - Corretude

Lema: Sejam G = (V,E) um grafo com peso nas arestas e $\mathbf{s \in V}$. Então $\mathbf{d^{|V|-1}[u]}$ é o tamanho do menor caminho entre **s** e **u** com no máximo **|V|-1** arestas.

Prova: Indução em **k**, o número de iterações no algoritmo. Para **k = 1,** o menor caminho entre **s** e **v** possui no mínimo **1** aresta, caso $\mathbf{s \neq v}$, portanto não existe caminho entre **s** e **v** com **0** arestas, logo $\mathbf{d^0[u] = \infty}$. Se $\mathbf{s = v}$, $\mathbf{d^0[s] = 0}$. Sendo assim, a condição é satisfeita.

Suponha que, na iteração **k**, $\mathbf{d^{k-1}[v]}$ represente o custo do menor caminho entre **s** e **v** com no máximo **k-1** arestas.

<u>**Caso 1:**</u> $\mathbf{d^{(k-1)}[v] < \min_a\{d^{(k-1)}[a] + w(a, v)\}}$. Ou seja, o menor caminho entre **s** e **v** tem menos do que **k** arestas. Então, como o relaxamento é definido como $\mathbf{d^k[v] = \min\{d^{k-1}[v], \min_a\{d^{k-1}[a] + w(a,v)\}\}}$, o algoritmo corretamente define $\mathbf{d^k[v] = d^{k-1}[v]}$.

Bellman-Ford - Corretude

Lema: Sejam G = (V,E) um grafo com peso nas arestas e $s \in V$. Então $d^{|V|-1}[u]$ é o tamanho do menor caminho entre **s** e **u** com no máximo **|V|-1** arestas.

Prova: Indução em **k**, o número de iterações no algoritmo. Para **k = 1,** o menor caminho entre **s** e **v** possui no mínimo **1** aresta, caso $s \neq v$, portanto não existe caminho entre **s** e **v** com **0** arestas, logo $d^0[u] = \infty$. Se $s = v$, $d^0[s] = 0$. Sendo assim, a condição é satisfeita.

Suponha que, na iteração **k**, $d^{k-1}[v]$ represente o custo do menor caminho entre **s** e **v** com no máximo **k-1** arestas.

Caso 1: $d^{(k-1)}[v] < \min_a\{d^{(k-1)}[a] + w(a, v)\}$. Ou seja, o menor caminho entre **s** e **v** tem menos do que **k** arestas. Então, como o relaxamento é definido como $d^k[v] = \min\{d^{k-1}[v], \min_a(d^{k-1}[a] + w(a,v)\}\}$, o algoritmo corretamente define $d^k[v] = d^{k-1}[v]$.

Caso 2: $d^{(k-1)}[v] \geq \min_a\{d^{(k-1)}[a] + w(a, v)\}$. Aqui, o menor caminho possui exatamente **k** arestas e o algoritmo corretamente define $d^k[v] = \min_a\{d^{(k-1)}[a] + w(a, v)\}$, minimizando o custo do caminho com **k** arestas.

Bellman-Ford - Corretude

Lema: Sejam G = (V,E) um grafo com peso nas arestas e **s** $\in$ **V.** Então **$d^{|V|-1}$[u]** é o tamanho do menor caminho entre **s** e **u** com no máximo **|V|-1** arestas.

Prova: Indução em **k**, o número de iterações no algoritmo. Para **k = 1,** o menor caminho entre **s** e **v** possui no mínimo **1** aresta, caso **s ≠v,** portanto não existe caminho entre **s** e **v** com **0** arestas, logo **d^0[u] = ∞.** Se **s = v, d^0[s] = 0.** Sendo assim, a condição é satisfeita.

Suponha que, na iteração **k**, **d^{k-1}[v]** represente o custo do menor caminho entre **s** e **v** com no máximo **k-1** arestas.

<u>**Caso 1:**</u> **$d^{(k-1)}[v] < \min_a\{d^{(k-1)}[a] + w(a, v)\}$.** Ou seja, o menor caminho entre **s** e **v** tem menos do que **k** arestas. Então, como o relaxamento é definido como **$d^k[v] = \min\{d^{k-1}[v], \min_a(d^{k-1}[a] + w(a,v))\}$,** o algoritmo corretamente define **$d^k[v] = d^{k-1}[v]$.**

<u>**Caso 2:**</u> **$d^{(k-1)}[v] \geq \min_a\{d^{(k-1)}[a] + w(a, v)\}$.** Aqui, o menor caminho possui exatamente **k** arestas e o algoritmo corretamente define **$d^k[v] = \min_a\{d^{(k-1)}[a] + w(a, v)\}$,** minimizando o custo do caminho com **k** arestas.

Portanto, ao final da iteração **k = |V|, d^{k-1}[v]** de fato é o menor custo do caminho entre **s** e **v** com no máximo **|V|- 1** arestas.

Bellman-Ford - Corretude

Teorema: O algoritmo de Bellman-Ford é correto, desde que não haja ciclos negativos.

Bellman-Ford - Corretude

Teorema: O algoritmo de Bellman-Ford é correto, desde que não haja ciclos negativos.

Prova: Pelo lema anterior, $\mathbf{d^{|V|-1}[v]}$ é de fato o custo do menor caminho entre **s** e **v** com no máximo **|V| - 1** arestas. Se não existem ciclos negativos, então o menor caminho necessariamente é simples e, portanto, tem no máximo **|V| - 1** arestas. Sendo assim, $\mathbf{d^{|V|-1}[v]}$ é o custo do menor caminho entre **s** e **v**.

Bellman-Ford e Programação Dinâmica

O algoritmo de Bellman-Ford é um exemplo de **Programação Dinâmica!**

Bellman-Ford e Programação Dinâmica

O algoritmo de Bellman-Ford é um exemplo de **Programação Dinâmica!**

Subestruturas ótimas:

$d^k[v] = \min\{d^{k-1}[v], \min_a\{d^{k-1}[a] + w(a,v)\}\}$, para todo **k** entre **0** e **|V| - 1.**

Bellman-Ford e Programação Dinâmica

O algoritmo de Bellman-Ford é um exemplo de **Programação Dinâmica!**

Subestruturas ótimas:

$$d^k[v] = \min\{d^{k-1}[v], \min_a\{d^{k-1}[a] + w(a,v)\}\}$$, para todo **k** entre **0** e **|V| - 1.**

Overlapping problems: Usamos a já calculada lista d^{k-1} para calcular d^k.

Métodos da Programação Dinâmica

Relembrando as duas abordagens ao se pensar em programação dinâmica:

1. **Bottom-Up:** Começa a iteração resolvendo os problemas menores. Por exemplo, o algoritmo de Bellman-Ford exposto resolve primeiro $\mathbf{d^0}$, depois $\mathbf{d^1}$ e continua até $\mathbf{d^{|V|-1}}$.

Métodos da Programação Dinâmica

Relembrando as duas abordagens ao se pensar em programação dinâmica:

1. **Bottom-Up:** Começa a iteração resolvendo os problemas menores. Por exemplo, o algoritmo de Bellman-Ford exposto resolve primeiro $\mathbf{d}^0$, depois $\mathbf{d}^1$ e continua até $\mathbf{d}^{|V|-1}$.

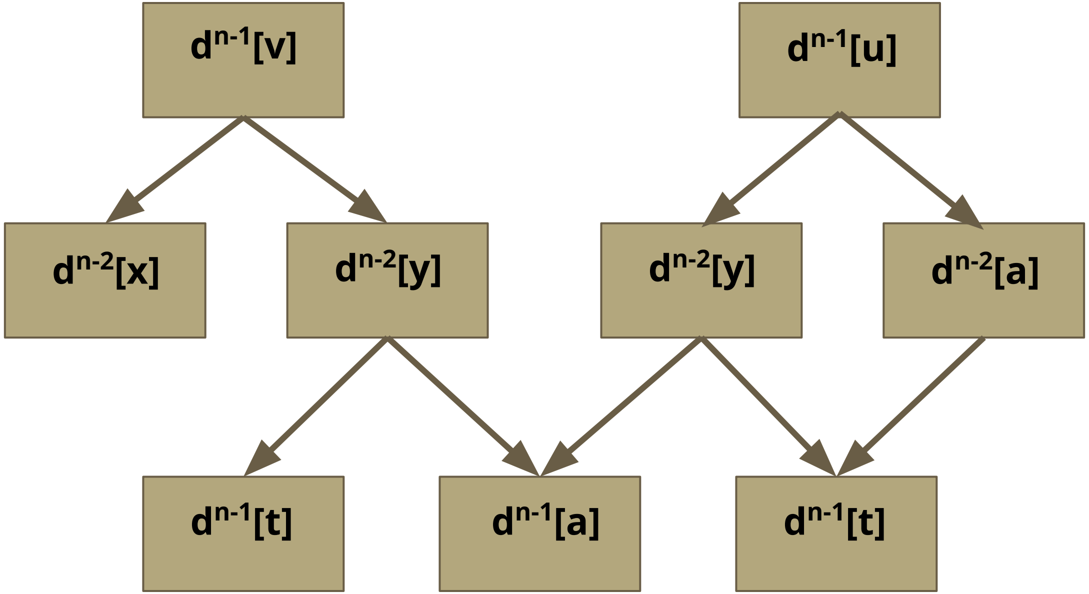
2. **Top-Down:** Resolve recursivamente os problemas, onde cada recursão resolve um problema ainda mais simples.

Top-Down Bellman-Ford - Pseudocódigo

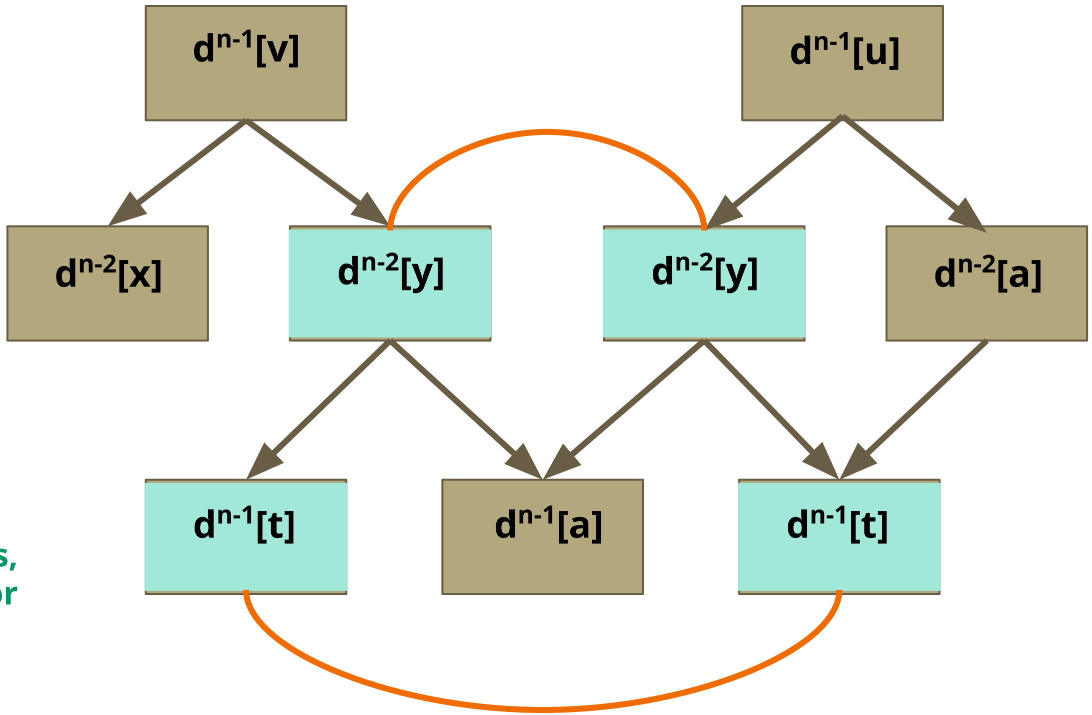
```
algorithm Bellman_Ford(G,s):
    dᵏ = [None]*k for k = 0,...,|V| - 1.
    d⁰[s] = 0
    d⁰[v] = ∞ for v ≠ s
    for b in V:
        d^{|V| - 1}[b] = BF_helper(G,b,|V|-1)

BF_helper(G,b,n):
    A = {a | (a,b) in E}U{b}
    for a em A:
        if d^{n-1}[a] == None:
            d^{n-1}[a] = BF_helper(G,b,n-1)
    return min{d^{n-1}[b], min_a{d^{n-1}[a],d^{n-1}[a] +
w(a,b)}}
```

Visualização do Top-Down Bellman-Ford

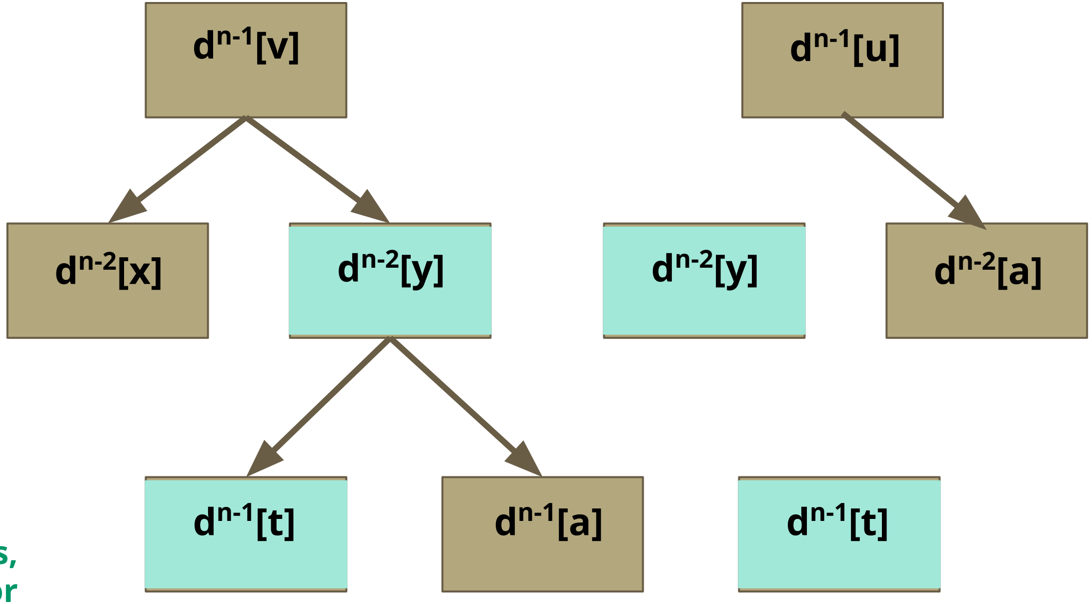


Visualização do Top-Down Bellman-Ford



Não precisamos calcular os valores conectados duas vezes, pois guardamos o valor da primeira vez que o calculamos!

Visualização do Top-Down Bellman-Ford



Não precisamos calcular os valores conectados duas vezes, pois guardamos o valor da primeira vez que o calculamos!

Pergunta surpresa!!!

Stanford - CS161 (Winter 2011)

True or False?

Let G be a directed graph in which edges can have a positive or negative edge lengths, but that has no negative cycles. The Bellman-Ford algorithm correctly computes shortest-path lengths from a given origin ***s*** to every other vertex ***v***.

Pergunta surpresa!!!

Stanford - CS161 (Winter 2011)

True or False?

Let G be a directed graph in which edges can have a positive or negative edge lengths, but that has no negative cycles. The Bellman-Ford algorithm correctly computes shortest-path lengths from a given origin ***s*** to every other vertex ***v***.

Resposta: **`True! Apenas observe que o menor caminho pode ser infinito, caso não haja um caminho entre os vértices u e v.`**

Floyd-Warshall

O algoritmo de Floyd-Warshall

Resolve o problema do menor caminho para todos os pares (*all-pairs shortest paths APSP)*. Ou seja, encontra a menor distância entre **u** e **v** para todo **u** e **v** em **V.**

O algoritmo de Floyd-Warshall

Resolve o problema do menor caminho para todos os pares (*all-pairs shortest paths APSP)*. Ou seja, encontra a menor distância entre **u** e **v** para todo **u** e **v** em **V.**

Uma solução simples seria executar o seguinte código:

```
for v in V:
     BellmanFord(G,v)
```

O algoritmo de Floyd-Warshall

Resolve o problema do menor caminho para todos os pares (*all-pairs shortest paths APSP)*. Ou seja, encontra a menor distância entre **u** e **v** para todo **u** e **v** em **V.**

Uma solução simples seria executar o seguinte código:

```
for v in V:
    BellmanFord(G,v)
```

Complexidade: $O(|V|^2|E|)$

O algoritmo de Floyd-Warshall

Resolve o problema do menor caminho para todos os pares (*all-pairs shortest paths APSP)*. Ou seja, encontra a menor distância entre **u** e **v** para todo **u** e **v** em **V.**

Uma solução simples seria executar o seguinte código:

```
for v in V:
     BellmanFord(G,v)
```

É possível fazer melhor!!

O algoritmo de Floyd-Warshall

Seja **G(V,E)** um grafo com pesos e sejam **u**, **v ∈ V**.

Para cada vértice, atribua um único valor entre **{1, … , |V|}.**

O algoritmo de Floyd-Warshall

Seja **G(V,E)** um grafo com pesos e sejam **u**, **v ∈ V**.

Para cada vértice, atribua um único valor entre **{1, ... , |V|}.**

Para cada **k ∈ {1, ... , |V|}**, defina **D^k[u,v]** como o valor do menor caminho entre **u** e **v** que passa pelos vértices **{1,...,k}.**

O algoritmo de Floyd-Warshall

Seja **G(V,E)** um grafo com pesos e sejam **u**, **v ∈ V**.

Para cada vértice, atribua um único valor entre **{1, ... ,|V|}.**

Para cada **k ∈ {1, ... ,|V|}**, defina **D^k[u,v]** como o valor do menor caminho entre **u** e **v** que passa pelos vértices **{1,...,k}.** Defina **D^0[u,v]** como o valor do caminho entre **u** e **v** composto por apenas uma aresta.

O algoritmo de Floyd-Warshall

Seja **G(V,E)** um grafo com pesos e sejam **u**, **v** $\in$ **V**.

Para cada vértice, atribua um único valor entre **{1, ... , |V|}.**

Para cada **k** $\in$ **{1, ... , |V|}**, defina $D^k[u,v]$ como o valor do menor caminho entre **u** e **v** que passa pelos vértices **{1,...,k}.** Defina $D^0[u,v]$ como o valor do caminho entre **u** e **v** composto por apenas uma aresta.

No Bellman-Ford, armazenamos uma **lista** d^k contendo os valores $d^k[v]$, para todo **v** $\in$ **V.**

O algoritmo de Floyd-Warshall

Seja **G(V,E)** um grafo com pesos e sejam **u**, **v ∈ V**.

Para cada vértice, atribua um único valor entre **{1, ... ,|V|}.**

Para cada **k ∈ {1, ... ,|V|}**, defina **D^k[u,v]** como o valor do menor caminho entre **u** e **v** que passa pelos vértices **{1,...,k}.** Defina **D^0[u,v]** como o valor do caminho entre **u** e **v** composto por apenas uma aresta.

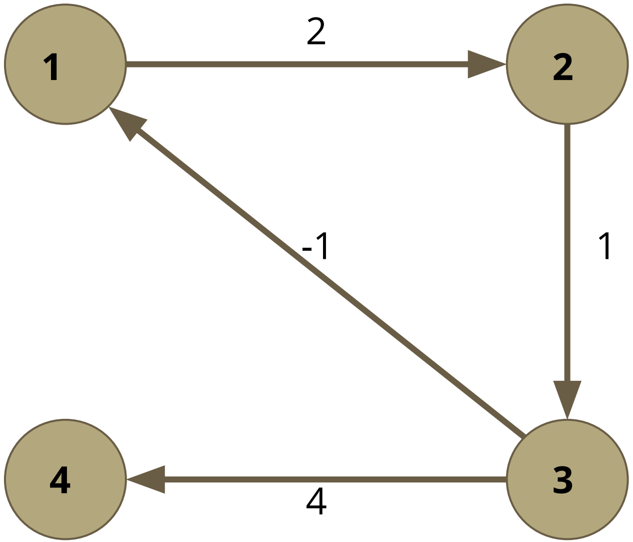
No Bellman-Ford, armazenamos uma **lista d^k** contendo os valores **d^k[v]**, para todo **v ∈ V.**

No algoritmo de Floyd-Warshall, armazenamos uma **matriz D^k**, com os valores de **D^k[u,v]**, para todos os pares **(u,v) ∈ V×V.**

O algoritmo de Floyd-Warshall

	1	2	3	4
1	0	2	∞	∞
2	∞	0	1	∞
3	-1	∞	0	4
4	∞	∞	∞	0

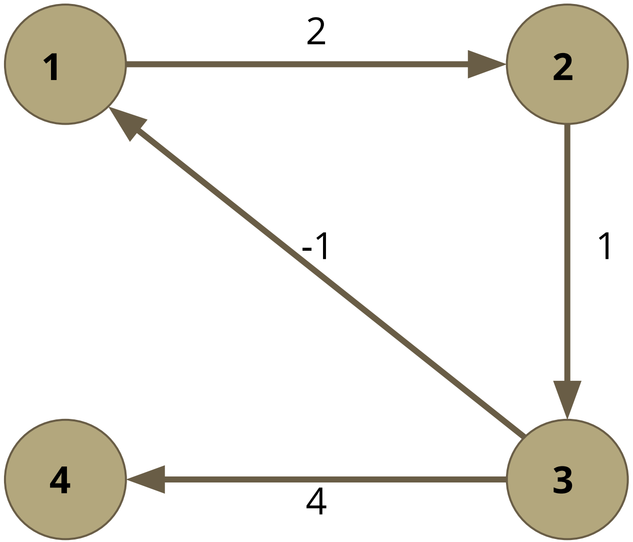
$D^0[u,v]$



O algoritmo de Floyd-Warshall

	1	2	3	4
1	0	2	∞	∞
2	∞	0	1	∞
3	-1	1	0	4
4	∞	∞	∞	0

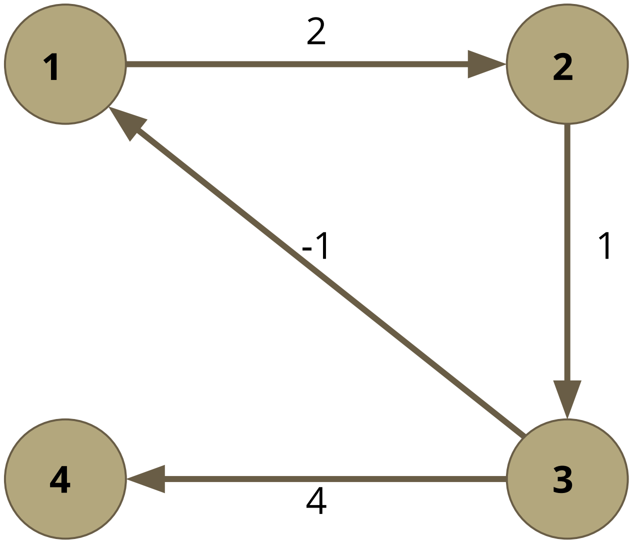
$D^1[u,v]$



O algoritmo de Floyd-Warshall

	1	2	3	4
1	0	2	3	7
2	∞	0	∞	∞
3	-1	1	0	4
4	∞	∞	∞	0

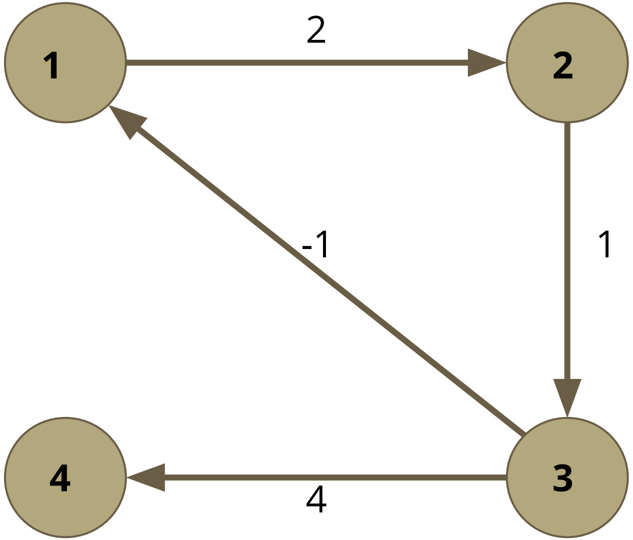
$D^2[u,v]$



O algoritmo de Floyd-Warshall

	1	2	3	4
1	0	2	3	7
2	0	0	1	5
3	-1	1	0	4
4	∞	∞	∞	0

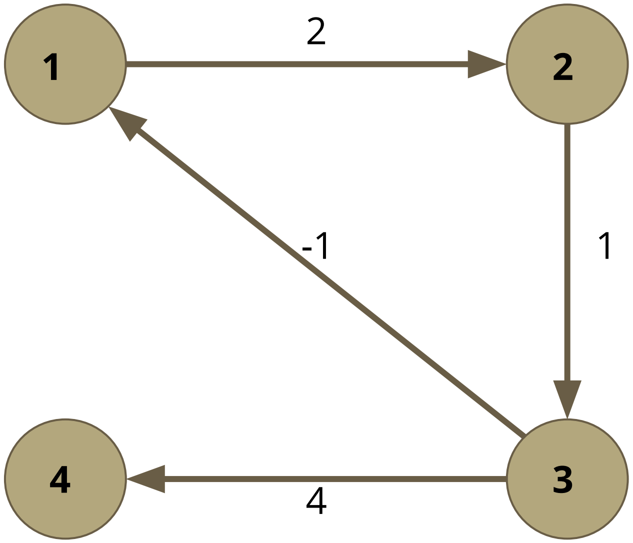
$D^3[u,v]$



O algoritmo de Floyd-Warshall

	1	2	3	4
1	0	2	3	7
2	0	0	1	5
3	-1	1	0	4
4	∞	∞	∞	0

$D^4[u,v]$



O algoritmo de Floyd-Warshall

Em geral, como os valores de $\mathbf{D}^k$ são atualizados?

O algoritmo de Floyd-Warshall

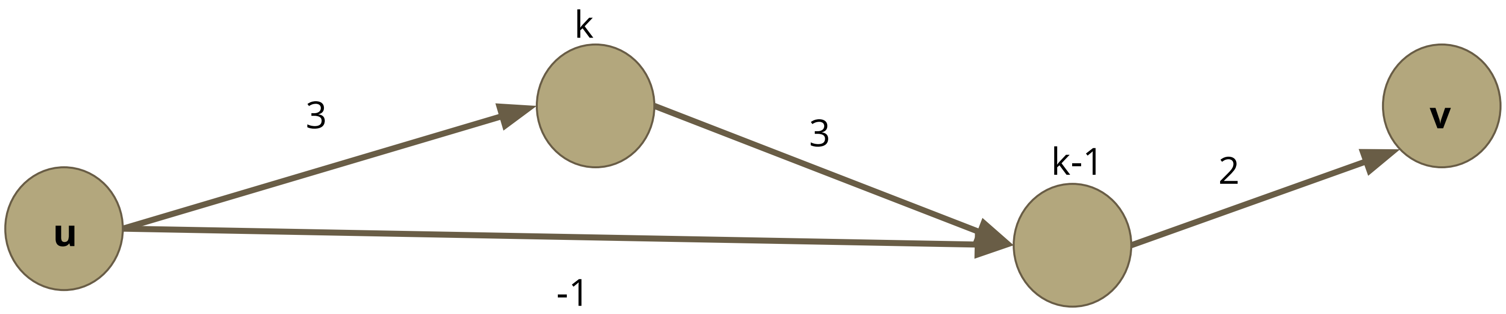
Em geral, como os valores de $\mathbf{D^k}$ são atualizados?

$$\mathbf{D^k[u,v] = \min\{D^{k-1}[u,v],\ D^{k-1}[u,k] + D^{k-1}[k,v]\}}$$

O algoritmo de Floyd-Warshall

Em geral, como os valores de $\mathbf{D}^k$ são atualizados?

$$\mathbf{D^k[u,v] = \min\{D^{k-1}[u,v],\ D^{k-1}[u,k] + D^{k-1}[k,v]\}}$$

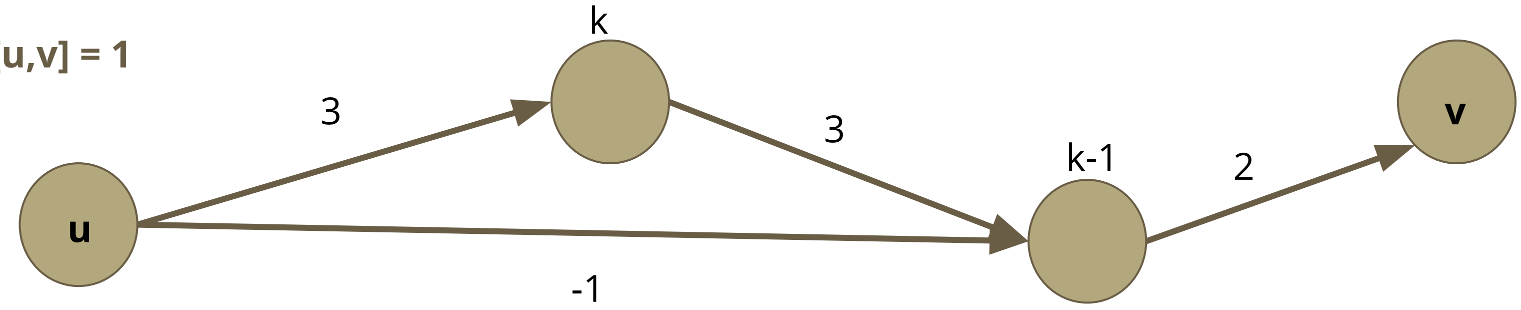


O algoritmo de Floyd-Warshall

Em geral, como os valores de $\mathbf{D^k}$ são atualizados?

$$\mathbf{D^k[u,v] = min\{D^{k-1}[u,v],\ D^{k-1}[u,k] + D^{k-1}[k,v]\}}$$

$\mathbf{D^{k-1}[u,v] = 1}$

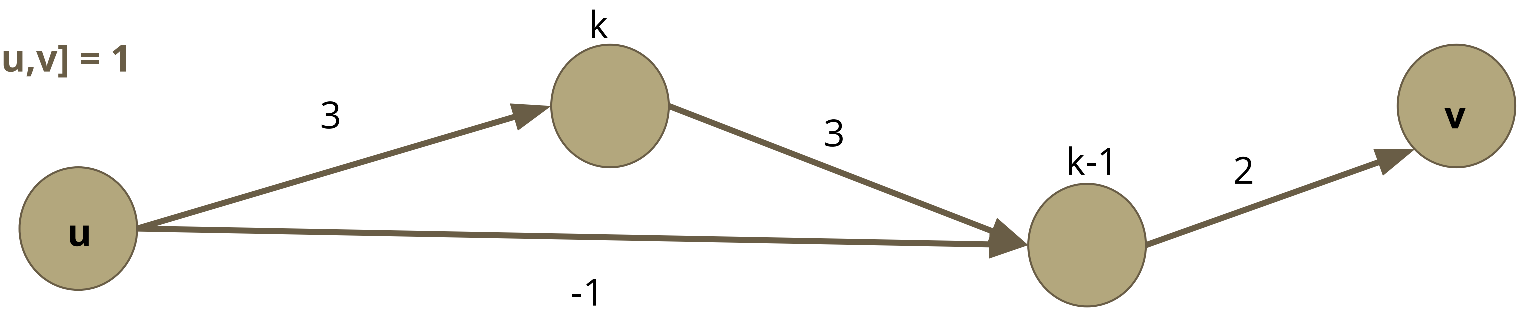


O algoritmo de Floyd-Warshall

Em geral, como os valores de $\mathbf{D^k}$ são atualizados?

$$D^k[u,v] = \min\{D^{k-1}[u,v],\ D^{k-1}[u,k] + D^{k-1}[k,v]\}$$

$\mathbf{D^{k-1}[u,v] = 1}$

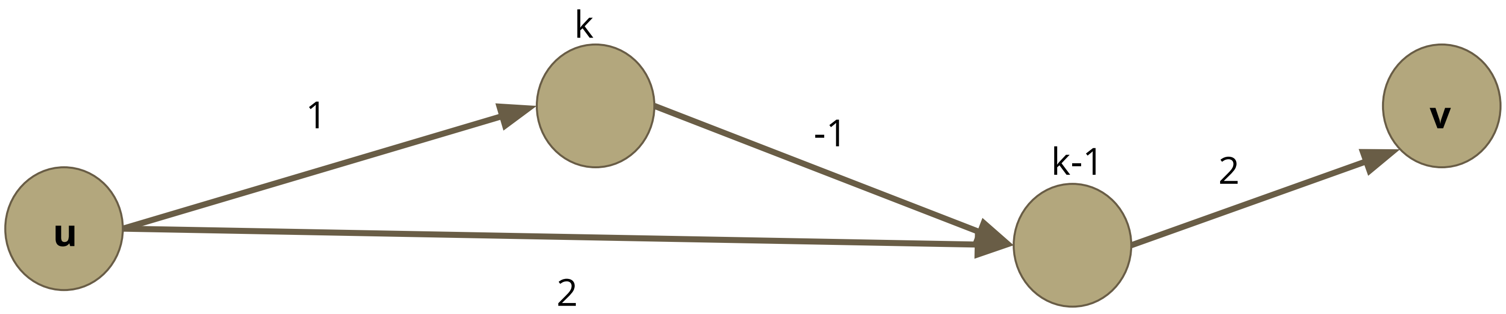


Neste caso, não precisamos do vértice **k!!**

O algoritmo de Floyd-Warshall

Em geral, como os valores de $\mathbf{D}^k$ são atualizados?

$$D^k[u,v] = \min\{D^{k-1}[u,v],\ D^{k-1}[u,k] + D^{k-1}[k,v]\}$$



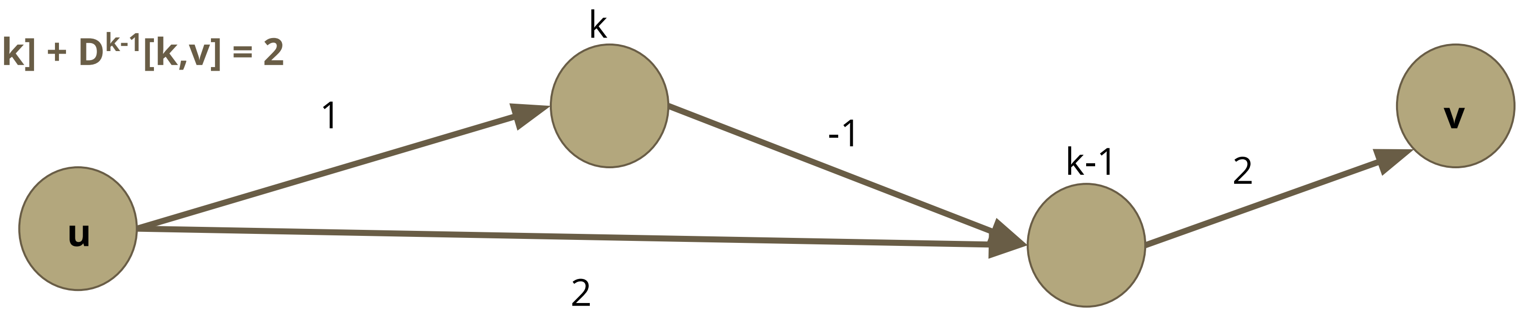
O algoritmo de Floyd-Warshall

Em geral, como os valores de $\mathbf{D^k}$ são atualizados?

$$D^k[u,v] = \min\{D^{k-1}[u,v],\ D^{k-1}[u,k] + D^{k-1}[k,v]\}$$

$D^{k-1}[u,v] = 4$

$D^{k-1}[u,k] + D^{k-1}[k,v] = 2$



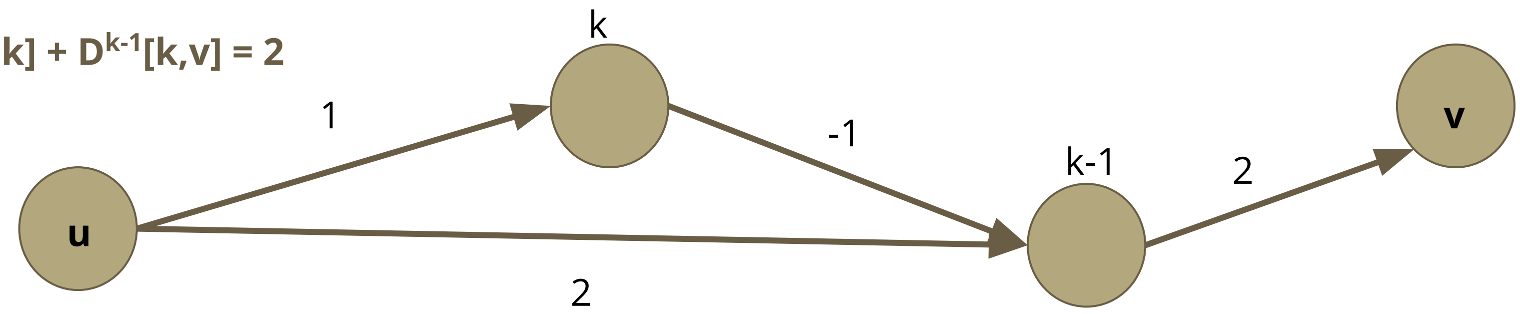
O algoritmo de Floyd-Warshall

Em geral, como os valores de $\mathbf{D^k}$ são atualizados?

$$D^k[u,v] = \min\{D^{k-1}[u,v],\ D^{k-1}[u,k] + D^{k-1}[k,v]\}$$

$D^{k-1}[u,v] = 4$

$D^{k-1}[u,k] + D^{k-1}[k,v] = 2$



Neste caso, **precisamos** do vértice **k!!**

O algoritmo de Floyd-Warshall

```
algorithm Floyd_Warshal(G):
    Dᵏ[u,u] = 0 for k = 0,...,|V|
    Dᵏ[u,v] = ∞ for u ≠ v and k = 0,...,|V|
    D⁰[u,v] = w(u,v), for (u,v) in E
    for k = 1,...,|V|:
        for (u,v) in V²:
            Dᵏ[u,v] = min{Dᵏ⁻¹[u,v],Dᵏ⁻¹[u,k] + Dᵏ⁻¹[k,v]}
    return D^|V|
```

O algoritmo de Floyd-Warshall

```
algorithm Floyd_Warshal(G):
    Dᵏ[u,u] = 0 for k = 0,...,|V|
    Dᵏ[u,v] = ∞ for u ≠ v and k = 0,...,|V|
    D⁰[u,v] = w(u,v), for (u,v) in E
    for k = 1,...,|V|:
        for (u,v) in V²:
            Dᵏ[u,v] = min{Dᵏ⁻¹[u,v],Dᵏ⁻¹[u,k] + Dᵏ⁻¹[k,v]}
    return D|V|
```

Complexidade: $O(|V|^3)$

Floyd-Warshall - Corretude

Lema: Seja **G = (V,E)** um grafo orientado e com pesos. Suponha que **G** não contenha ciclos negativos. Então, para todo **(u,v) ∈ V** e para todo **k = 0,...,|V|**, $\mathbf{D^k[u,v]}$ armazena a menor distância entre **u** e **v** passando pelos vértices **{1,...,k}.**

Floyd-Warshall - Corretude

Lema: Seja **G = (V,E)** um grafo orientado e com pesos. Suponha que **G** não contenha ciclos negativos. Então, para todo **(u,v) ∈ V** e para todo **k = 0,…,|V|**, $\mathbf{D^k[u,v]}$ armazena a menor distância entre **u** e **v** passando pelos vértices **{1,…,k}.**

Prova: Indução em **k.** Para **k = 0,** o resultado é claro, pois por definição

$$D^0[u,v] = w(u,v) \quad \text{ou } \infty$$

Que é o valor da distância entre **u** e **v** sem passar por nenhum vértice.

Floyd-Warshall - Corretude

Suponha que $\mathbf{D^{k-1}[u,v]}$ seja o valor da menor distância entre **u** e **v** passando pelo vértice **k-1.**
Considere $\mathbf{D^{k}[u,v] = \min\{D^{k-1}[u,v], D^{k-1}[u,k]+D^{k-1}[k,v]\}}$.

Como visto anteriormente, temos dois casos:

Floyd-Warshall - Corretude

Suponha que $D^{k-1}[u,v]$ seja o valor da menor distância entre **u** e **v** passando pelo vértice **k-1.**
Considere $D^{k}[u,v] = \min\{D^{k-1}[u,v], D^{k-1}[u,k]+D^{k-1}[k,v]\}$.

Como visto anteriormente, temos dois casos:

1. O vértice **k** aumenta a distância entre **u** e **v.** Neste caso, a fórmula corretamente atribui o valor $D^{k-1}[u,v]$, como a menor distância e, pela hipótese de indução, esta é a menor distância entre **u** e **v** passando por **k-1,** portanto, também será a menor passando por **k.**

Floyd-Warshall - Corretude

Suponha que **D^{k-1}[u,v]** seja o valor da menor distância entre **u** e **v** passando pelo vértice **k-1.** Considere **$D^k[u,v] = \min\{D^{k-1}[u,v], D^{k-1}[u,k]+D^{k-1}[k,v]\}$.**

Como visto anteriormente, temos dois casos:

1. O vértice **k** aumenta a distância entre **u** e **v.** Neste caso, a fórmula corretamente atribui o valor **D^{k-1}[u,v],** como a menor distância e, pela hipótese de indução, esta é a menor distância entre **u** e **v** passando por **k-1,** portanto, também será a menor passando por **k.**
2. O vértice **k** diminui a distância entre **u** e **v.** Neste caso, como **G** não possui ciclos negativos, podemos supor sem perda de generalidade que o caminho entre **u** e **v** passando por **k** é simples (por quê?). Como os caminhos entre **(u** e **k)** e **(k** e **v),** são simples e mínimos, pela hipótese de indução, segue que **$D^k[u,v] = D^{k-1}[u,k]+D^{k-1}[k,v]$,** pois caso **$D^k[u,v] < D^{k-1}[u,k]+D^{k-1}[k,v]$,** pelo menos um entre **D^{k-1}[u,k]** e **D^{k-1}[k,v]** não seriam mínimos.

Floyd-Warshall - Corretude

Teorema: Seja **G = (V,E)** um grafo direcionado com pesos. Suponha que não exista ciclo negativo em **G.** Então o algoritmo de Floyd-Warshall retorna uma matriz $\mathbf{D}^{\mathbf{n}}$ tal que

$$\mathbf{D}^{\mathbf{n}}[\mathbf{u},\mathbf{v}] = \text{menor distância entre } \mathbf{u} \text{ e } \mathbf{v}$$

Floyd-Warshall - Corretude

Teorema: Seja **G = (V,E)** um grafo direcionado com pesos. Suponha que não exista ciclo negativo em **G.** Então o algoritmo de Floyd-Warshall retorna uma matriz $\mathbf{D^n}$ tal que

$$\mathbf{D^n[u,v]} = \text{menor distância entre } \mathbf{u} \text{ e } \mathbf{v}$$

Prova: Corolário do Lema anterior.

Pergunta (não tão) **surpresa!!!**

Stanford - CS161 (Winter 2011)

True or False?

Let *G* be a directed graph in which edges can have positive or negative edge lengths, but that has no negative cycles. The Floyd-Warshall algorithm correctly computes the shortest-path lengths between every pair of vertices.

Pergunta (não tão) **surpresa!!!**

Stanford - CS161 (Winter 2011)

True or False?

Let *G* be a directed graph in which edges can have positive or negative edge lengths, but that has no negative cycles. The Floyd-Warshall algorithm correctly computes the shortest-path lengths between every pair of vertices.

Resposta: `True! Lembre que o algoritmo Floyd-Warshall calcula a distância entre TODOS os pares de nós.`

Mais uma pergunta (juro que é a última...)

Stanford - CS161 (Practice Final 2017)

Breadth-first search, Dijkstra's algorithm, the Bellman-Ford algorithm and the Floyd-Warshall algorithm can all be used to find shortest paths in a graph. **Draw a line from each question to the best answer to that question.**

When might you prefer Floyd-Warshall to Bellman-Ford?

When might you prefer breadth-first search to Dijkstra's algorithm?

When might you prefer Bellman-Ford to Dijkstra's algorithm?

When the graph has negative edge weights.

When the graph is unweighted.

When you want to find the shortest paths between all pairs of vertices.

When you want to find the shortest paths from a specific vertex ***s*** to any other vertex ***t***.

Mais uma pergunta (juro que é a última...)

Stanford - CS161 (Practice Final 2017)

Breadth-first search, Dijkstra's algorithm, the Bellman-Ford algorithm and the Floyd-Warshall algorithm can all be used to find shortest paths in a graph. **Draw a line from each question to the best answer to that question.**

When might you prefer Floyd-Warshall to Bellman-Ford?

When might you prefer breadth-first search to Dijkstra's algorithm?

When might you prefer Bellman-Ford to Dijkstra's algorithm?

When the graph has negative edge weights.

When the graph is unweighted.

When you want to find the shortest paths between all pairs of vertices.

When you want to find the shortest paths from a specific vertex ***s*** to any other vertex ***t***.

Resumo: Algoritmos *Shortest Path*

	BFS	Dijkstra	Bellman-Ford	Floyd-Warshall
Complexidade	O(V+E)	O((V+E) log V)	O(V . E)	$O(V^3)$
Tamanho Recomendado do Grafo	Grande	Médio / Grande	Pequeno / Médio	Pequeno
Bom para *All Pair Shortest Path*?	Só funciona em *unweighted graphs*	OK	Não	Sim
Pode detectar ciclos negativos?	Não	Não	Sim	Sim
Shortest Path on Weighted Graphs	Resposta Incorreta	Funciona bem se os pesos são positivos	Ruim	Ruim

Referência: Competitive Programming 3, pag. 161.

Referências

[1]Cormen, T. Introduction to Algorithms, Third Ed. 2009.

[2]http://web.stanford.edu/class/archive/cs/cs161/cs161.1178/notes/bf-fw.pdf

[3]https://www.eecs.yorku.ca/course_archive/2003-04/S/3101A/notes/dp.pdf

[4]https://medium.com/@maheshkariya/difference-between-divide-and-conquer-algo-and-dynamic-programming-4a657bcb6187

[5]http://web.stanford.edu/class/archive/cs/cs161/cs161.1176/Slides/Lecture12.pdf

Now it's Lab time!!!