

FGV EMap

Escrita: Thalís Ambrosim Falqueto

Projeto e Análise de Algoritmos

Revisão para A2

Rio de Janeiro

2025

Conteúdo

1	Técnicas de Projeto	3
1.1	Método guloso (Greedy)	4
1.1.1	O problema do agendamento de tarefas	4
1.1.2	O problema da mochila fracionária	7
1.2	Dividir e Conquistar	9
1.2.1	O problema de contagem de inversões	10
1.2.2	O problema de pares mais próximos	13
1.2.3	como faz isso cara como é 11 7, 5 sla	14
1.2.4	Implementação em Python	14
1.3	Programação Dinâmica	14
1.3.1	O problema de Fibonacci	15
1.3.2	O problema da mochila (não fracionária)	16
2	Grafos	21
2.1	Relembrando conceitos	22
2.2	Estruturas de dados para representar grafos	24
2.2.1	Matriz de adjacência	24
2.2.2	Lista de adjacência	24
3	Busca em Grafos	26
3.1	DFS	28
3.2	Grafo topológico	29
3.2.1	fazer exercício de numeração topológica	30
3.3	DFS modificado	30
3.4	Propriedades úteis advindas do DFS	34
3.4.1	explicar visualmente	36
3.4.2	explicação e implementação em Python	36
3.5	BFS	36

Técnicas de Projeto

1.1 Método guloso (Greedy)

O método guloso é um famoso paradigma utilizado para projetos de algoritmo, onde a estratégia consiste em escolher a cada iteração a opção com maior valor, e avaliar se deve ser adicionada ao resultado final.

Seguindo essa abordagem, as opções precisam ser ordenadas pro algum critério. Costuma ser simples e eficiente, porém nem todo projeto pode ser resolvido através dessa abordagem.

1.1.1 O problema do agendamento de tarefas

Dado o conjunto de tarefas $T = \{t_1, t_2, \dots, t_n\}$ com n elementos, cada uma com tempo de início $[t_k]$, e um tempo de término $[t_k]$, encontre o maior subconjunto de tarefas que pode ser alocado sem sobreposição temporal.

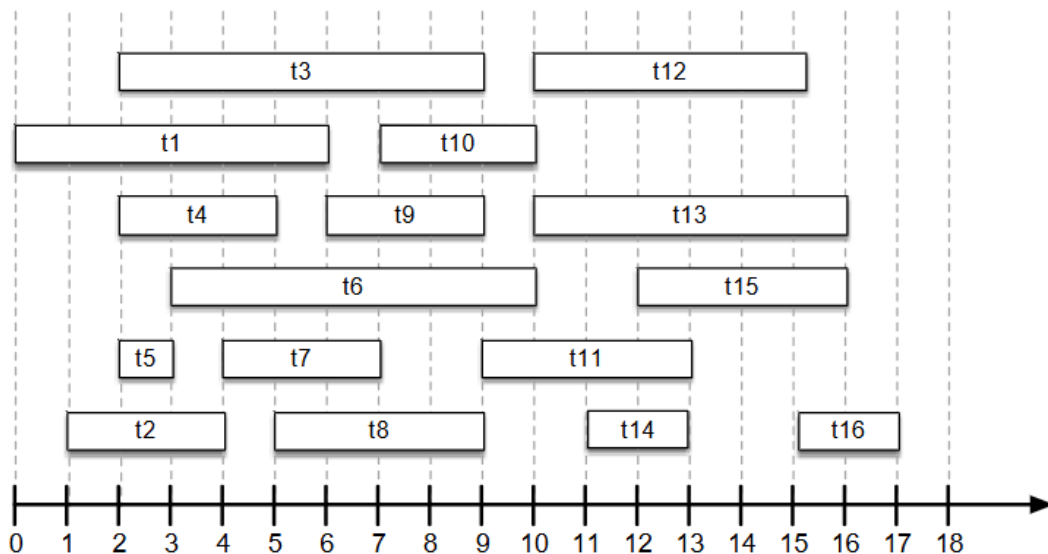


Figura 1: Exemplo do problema de agendamento

Vamos projetar a solução!

Perguntas:

1. Quais serão as opções a serem avaliadas a cada iteração?
 - Conjunto de tarefas que ainda não foi alocada ou descartada.
2. Qual critério iremos utilizar para ordenar as opções?
 - Tempo de início?
 - Menor duração?
 - Menor número de projetos?
 - Tempo de término?

Vamos analisar cada critério tentando construir ao menos um cenário que demonstre que o critério gera um resultado não-ótimo.

Que tal se colocarmos o critério de seleção como o tempo de início do agendamento?

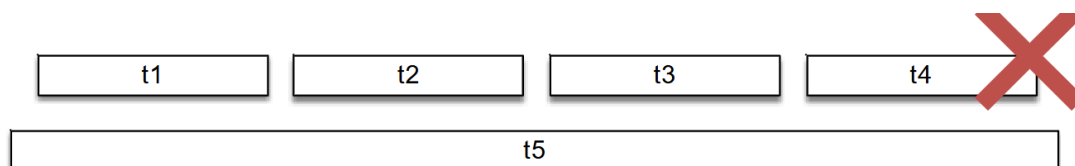


Figura 2: Contra-exemplo para uma possível solução do problema de agendamento

Isso não daria certo, pois nesse caso, por exemplo, o t_5 seria escolhido, enquanto a melhor escolha seria pegar as 4 primeiras tarefas.

E se escolhessemos pela menor duração?

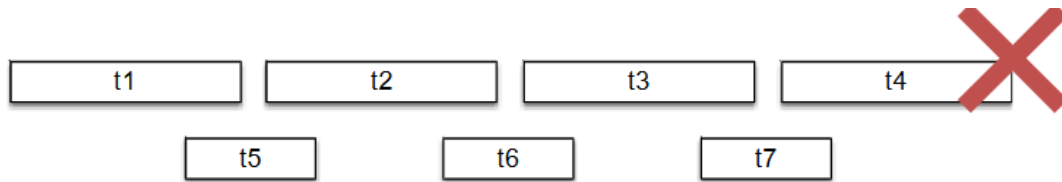


Figura 3: Contra-exemplo para uma possível solução do problema de agendamento

Isso também daria errado, já que escolheríamos t_5, t_6 e t_7 enquanto, novamente, a melhor escolha seriam os 4 primeiros.

Nada funcionou... Mas e se nosso critério fosse o tempo do término?

Ideia geral:

- 1 **ordene** T (pelo tempo de término)
- 2 $T_a[1, \dots, n] = 0$
- 3 **Insira a primeira tarefa da lista** t_0 **em** T_a
- 4 $t_{\text{prev}} = t_0$
- 5 **para** t_k **em** T :
- 6 **se** $\text{start}[t_k] \geq \text{end}[t_{\text{prev}}]$:
- 7 **adicione** t_k **em** T_a
- 8 $t_{\text{prev}} = t_k$
- 9 **retorne** T_a .

Essa ideia não é muito difícil. Como a lista é ordenada pelo horário de saída, então o primeiro elemento a ser adicionado é simplesmente o primeiro elemento. Note que, como o objetivo é apenas a quantidade máxima de tarefas, e não o tempo máximo que podemos otimizar para todas as tarefas que temos, então pegar o menor término **desde o início** é o que realmente faz o algoritmo funcionar (por exemplo, se tivéssemos os horários $[(5, 10), (5, 12)]$, pegar o menor tempo de saída nos ajudaria no caso de termos outra tarefa, como $(11, 14)$).

Após selecionarmos a primeira tarefa da lista, basta compararmos os tempos de entrada das próximas tarefas, já que, pelo mesmo raciocínio do porque escolher a menor saída, se a próxima tarefa não colidir com a saída passada, então podemos pegar nossa nova tarefa e atualizar com o tempo de saída da nova tarefa atual (como a lista está ordenada pelo tempo de fim, a nova tarefa a ser pega garantiria que seria a melhor tarefa, já que seria a primeira que se encaixa com o tempo de finalização da última tarefa selecionada e a mais curta já que estamos olhando por ordenação).

Nosso pseudocódigo usa apenas um for sem nada demais dentro dele, mas precisamos ordenar a lista antes. Isso nos traz uma complexidade de $\Theta(n \log(n))$.

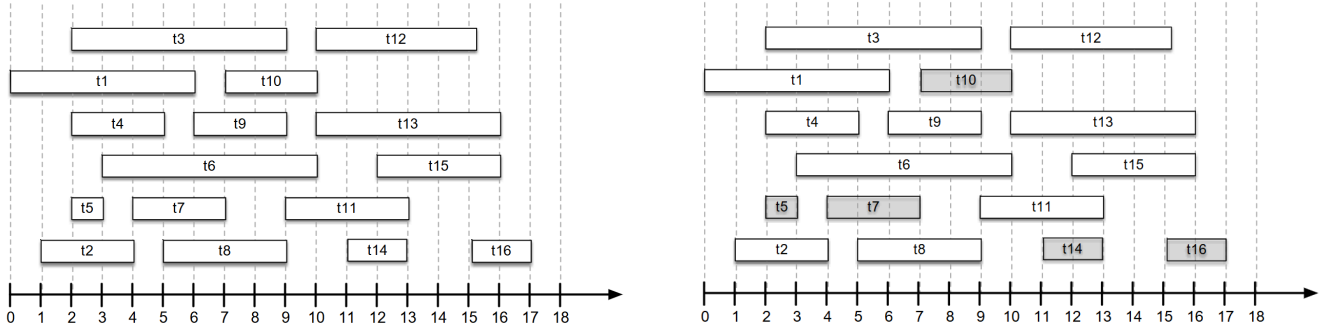


Figura 4: Solução para o problema de tarefas usando o algoritmo proposto

Por que essa solução é ótima?

Definição 1.1.1.1: Escolha gulosa

Uma solução ótima global pode ser atingida realizando uma sequência de escolhas locais ótimas (gulosas).

- A escolha local não considera o resultado das escolhas posteriores, e produz um sub-problema contendo um número menor de elementos.
- A definição do critério de escolha nos auxilia à organizar os elementos de forma que o algoritmo seja eficiente.

Definição 1.1.1.2: Sub-estrutura ótima

Ocorre quando uma solução ótima de um problema apresenta dentro dela soluções ótimas para sub-problemas.

Definição 1.1.1.3: Swap argument (Argumento de troca)

Considere que temos uma solução ótima S , e a solução gulosa G . Então é possível substituir iterativamente os elementos de S por elementos de G sem que a solução deixe de ser viável e ótima, provando assim que G é, no mínimo, tão boa quanto S .

Vamos usar o que aprendemos então:

Seja $T_a = \{g_1, g_2, \dots, g_k\}$ o conjunto de k tarefas selecionadas pelo nosso algoritmo guloso, já ordenadas pelo tempo de término (como no pseudocódigo). Seja $S = \{s_1, s_2, \dots, s_m\}$ uma *solução ótima* qualquer, com m tarefas, também ordenadas por tempo de término.

Nosso objetivo é provar que T_a é ótima, ou seja, que $k = m$.

Queremos primeiro provar que a primeira escolha gulosa, g_1 , pode fazer parte de *alguma* solução ótima.

1. g_1 é a tarefa escolhida por nosso algoritmo, então ela é a tarefa em *todo* o conjunto T com o *menor tempo de término*.
2. s_1 é a primeira tarefa da solução ótima S . Ela tem o menor tempo de término *dentro* de S .

Por definição, como g_1 tem o menor tempo de término de *todas* as tarefas, seu tempo de término deve ser menor ou igual ao de s_1 :

$$\text{end}[g_1] \leq \text{end}[s_1] \quad (1)$$

Agora, vamos comparar g_1 e s_1 .

1. Caso 1: $g_1 = s_1$. Se a primeira tarefa da solução ótima S é a mesma da solução gulosa T_a , então S já começa com a escolha gulosa.
2. Caso 2: $g_1 \neq s_1$. Vamos “trocar” s_1 por g_1 na solução ótima S . Considere uma nova solução S' : $S' = \{g_1, s_2, s_3, \dots, s_m\}$. Precisamos verificar se S' ainda é uma solução viável (sem sobreposições).
 - Como S era uma solução viável, todas as suas tarefas eram compatíveis. Sabemos que s_2 devia começar após s_1 terminar: $\text{start}[s_2] \geq \text{end}[s_1]$.
 - Mas, como vimos na Etapa 1, $\text{end}[g_1] \leq \text{end}[s_1]$.
 - Combinando os fatos, temos que $\text{start}[s_2] \geq \text{end}[g_1]$.
 - Isso significa que g_1 não se sobrepõe a s_2 , e o resto das tarefas (s_3, \dots) também não, pois já eram compatíveis com s_2 .

A nova solução S' é, portanto, viável. O mais importante é que S' tem m tarefas, o mesmo tamanho da solução ótima S . Isso significa que S' também é uma solução ótima.

Concluimos que *sempre* existe uma solução ótima (seja S ou S') que começa com a primeira escolha gulosa g_1 . Podemos repetir esse processo indutivamente. Em cada passo i , trocamos s_i por g_i , transformando a solução ótima S na solução gulosa T_a , sem nunca diminuir o número de tarefas, usando sub-estruturas ótimas. Isso só é possível se as duas soluções tiverem o mesmo tamanho desde o início. Portanto, $k = m$.

Logo, a solução gulosa T_a é, de fato, uma solução ótima.

Implementação em Python:

```
1  def scheduling_problem(tasks):  
2      if len(tasks) == 0: #caso de contorno  
3          return 0  
4  
5      sorted_by_end = sorted(tasks, key= lambda x:x[1]) #ordena pelo término  
6      choosed_tasks = []  
7      choosed_tasks.append(sorted_by_end[0])  
8      t_prev = sorted_by_end[0]  
9  
10     for task in sorted_by_end[1:]: #começa depois da primeira  
11         if task[0] >= t_prev[1]: #tempo maior que o de saída  
12             choosed_tasks.append(task)  
13             t_prev = task  
14     return choosed_tasks, len(choosed_tasks) #retorna lista, quantidade
```

1.1.2 O problema da mochila fracionária

Dado um conjunto de itens $\mathbb{I} = \{1, 2, 3, \dots, n\}$ em que cada item $i \in \mathbb{I}$ tem um peso w_i e um valor v_i , e uma mochila com capacidade de peso W , encontre o subconjunto $S \subseteq \mathbb{I}$ tal que $\sum_{i \in S} \alpha_i w_i \leq W$ e $\sum_{i \in S} \alpha_i v_i$ seja máximo, considerando que $0 < \alpha_k \leq 1$.

Item	Peso	Valor
1	1	2
2	3	4
3	4	6
4	5	10
5	7	8

Figura 5: Tabela de exemplo para o exemplo da mochila

Exemplo:

- $W = 9$
 - A escolha $\{1, 2, 3\}$ tem peso 8, valor 12 e cabe na mochila;
 - A escolha $\{3, 5\}$ tem peso 11, valor 14 e **não** cabe na mochila
 - A escolha $\{3_{50\%}, 5_{100\%}\}$ tem peso 9, valor 11 e cabe na mochila
 - A escolha $\{1_{100\%}, 3_{75\%}, 4_{100\%}\}$ tem peso 9, valor 16.5 e cabe na mochila

Seria possível criar um algoritmo capaz de encontrar uma solução ótima para esse problema?

- Pergunta 1: quais são as opções a serem avaliadas à cada iteração?
 - Itens (ou fragmentos de itens) que ainda não foram adicionados ou descartados.
- Pergunta 2: Qual critério iremos utilizar para ordenar as opções?
 - Menor peso?
 - Menor valor?
 - Maior razão peso/valor?

Essa ideia de razão parece fazer sentido, já que podemos separar e pegar a proporção que quisermos. Daí vem a ideia do algoritmo:

```

1 Mochila ( $I, v, w, n, W$ ) :
2   ordene  $I$  (pela razão valor/peso)
3    $C, i = W, 1$ 
4    $M[1, \dots, n] = 0$ 
5   enquanto  $i \leq n$  e  $C \geq w_i$ :
6      $M[i] = 1$ 
7      $C = C - w_i$ 
8      $i += 1$ 
9   se  $i \leq n$  :
10     $M[i] = \frac{C}{w_i}$ 
    retorne  $M$ 

```

Onde I é o conjunto de itens, v o vetor de valores de cada item, w o vetor de pesos de cada item, n a quantidade de itens e W é a capacidade máxima da mochila.

Analizando brevemente, ordenamos **descrescentemente** o vetor de itens \mathbb{I} , e declaramos a variável C , de capacidade, e i , de índice. Criamos o vetor de zeros M (de tamanho n), que é o vetor de porcentagem, referente a cada item. Note que como estamos ordenando pela proporção

de valor por peso decrescentemente, pegar o primeiro item significa pegar o que item que mais vale a pena. Logo, o while serve para, enquanto couber a capacidade, pegara maior quantidade possível de valores. Quando o while quebra (no índice i), o algoritmo verifica se não chegou ao final, e, caso não tenha chegado, pega a proporção máxima da capacidade máxima restante sobre o peso, e retorna a lista de pesos ao final.

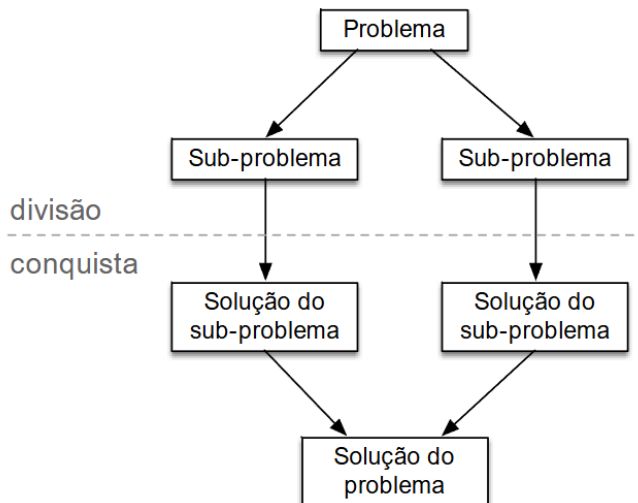
O mais complexo é a ordenação, que pode ser garantido com $\Theta(n \log(n))$.

Implementação em Python:

```
1 def fractional_bag_problem(I, v, w, max_w):
2     n = len(I) #as três listas têm o mesmo tamanho
3     idx_w_ratio = []
4     for i in range(n):
5         ratio = v[i]/w[i]
6         idx_w_ratio.append((i, w[i], ratio))#lista que armazena o índice, peso e
           razão
7         #ordena por razão logo abaixo
8     idx_w_ratio = sorted(idx_w_ratio, key = lambda x: x[2], reverse=True)
9     capacity, i = max_w, 0
10    M = [0] * n
11
12    while i < n and capacity >= idx_w_ratio[i][1]:
13        M[i] = 1 #faz o while normal
14        capacity -= idx_w_ratio[i][1]
15        i += 1
16    if i < n:
17        M[i] = capacity/idx_w_ratio[i][1]
18
19    itens_chosed = [0] * n #lista que referencia a cada item a sua
20    for j in range(n): #porcentagem escolhida
21        if M[j] != 0:
22            itens_chosed[idx_w_ratio[j][0]] = M[j]
23
24    return itens_chosed #retorna a lista de índices com a %
```

1.2 Dividir e Conquistar

O nome já diz muito, e esse paradigma é dividido em três etapas:



- Dividir o problema em um conjunto de sub-problemas menores.
- Resolver cada sub-problema recursivamente.
- Combinar os resultados de cada sub-problema gerando a solução.

Figura 6: Exemplificação do paradigma Dividir e Conquistar.

1.2.1 O problema de contagem de inversões

Dado um problema com n números, calcule o número de inversões necessário para torná-la ordenada.

Exemplo:

Considere a sequência $A = [3, 7, 2, 9, 5]$

O número de inversões é 4: $(7, 2), (3, 2), (9, 5), (7, 5)$

A solução por força bruta seria verificar todos os pares, exigindo $\Theta(n^2)$.

A solução baseada em dividir e conquistar deverá definir estratégias para resolver cada sub-problema do número de inversões, e depois juntar, claro. Podemos dividir a sequência em dois grupos com aproximadamente metade (O primeiro array até $\frac{n}{2}$, o segundo de $\frac{n}{2} + 1$ até n). Essa operação é constante, portanto $O(1)$.

A estratégia de resolução deve contar o número de inversões de cada grupo:

Exemplo:

2	5	4	1	9	7	6	11	3	12	8	10
---	---	---	---	---	---	---	----	---	----	---	----

2	5	4	1	9	7	6	11	3	12	8	10
---	---	---	---	---	---	---	----	---	----	---	----

5 inversões						6 inversões					
(2,1), (5,4), (5,1), (4,1), (9,7)						(6,3), (11,3), (11,8), (11,10), (12,8), (12,10)					

Figura 7: Exemplo do problema da contagem de inversões

Esse resultado pode ser obtido executando o algoritmo recursivamente ($\sim T(\frac{n}{2})$). Claro que, por fim, teremos que contar as inversões da junção das duas listas:

+ 7 inversões ao combinar
 $(5,3), (4,3), (9,6), (9,3), (9,8), (7,6), (7,3)$

Totalizando, assim, 18 inversões.

Ok, a ideia está concisa, mas como fazer essa junção? Se ordenarmos cada segmento, e “juntarmos” direto, conseguiríamos fazer isso de forma fácil. Voltemos ao exemplo após ordenar:

1	2	4	5	7	9	3	6	8	10	11	12
---	---	---	---	---	---	---	---	---	----	----	----

Para a contagem de inversões para a junção, considere a sequências à esquerda de L e à direita de R e defina $i = 0$.

```

1 para  $a_j$  de  $R$ :
2   | incremente  $i$  até  $L[i] > a_j$ 
3   |  $\text{inv}_{a_j} = |L| - i$ .
```

Com inv_{a_j} sendo a quantidade de inversões do elemento a_j de R .

Para explicar, considere o exemplo da imagem anterior e lembre um pouco do algoritmo de ordenação MergeSort. Dado que as listas menores já estão ordenadas, se colocarmos um ponteiro no início de cada lista, digamos l e r , então basta verificar se $L[l] \leq R[r]$, e incrementarmos l (ou r , dependendo da comparação).

Vamos continuar o exemplo (à princípio, pense que apenas juntamos as duas listas):

- Para $l = 0$ e $r = 0$, temos $1 \leq 3$, que é verdade. Incrementamos o l .
- Para $l = 1$ e $r = 0$, temos $2 \leq 3$, que ainda é verdade. Incrementamos o l .
- Para $l = 2$ e $r = 0$, temos $4 \leq 3$, que é mentira. Então, podemos aplicar o raciocínio: sabendo que todos os elementos após o 4, como estão ordenados, são maiores ou iguais a 4 e, portanto, também teriam que ser considerados como maiores do que 3, então do índice l atual até $|L|$, uma inversão precisaria ser feita, se concatenássemos as duas listas. Logo, temos $|L| - l$ trocas a serem feitas a partir de $L[l]$. Isso explica o algoritmo para a contagem de inversões para a junção.

Agora que resolvemos esse problema, podemos desenhar o algoritmo final:

```

1 CountInversions ( $A$ ) :
2   | se  $\text{len}(A) = 1$  :
3   |   | retorne 0
4   | divida a lista em  $L$  e  $R$ 
5   |  $i_l = \text{CountInversions}(L)$ 
6   |  $i_r = \text{CountInversions}(R)$ 
7   |  $L = \text{Sort}(L)$ 
8   |  $R = \text{Sort}(R)$ 
9   |  $i = \text{Combine}(L, R)$ 
10  | retorne  $i_l + i_r + i$ 
```

Avaliando a complexidade desse algoritmo, temos:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n \log(n)) = O(n(\log(n))^2) \quad (2)$$

O $O(n \log(n))$ vem da ordenação das duas listas, e o $2T(\frac{n}{2})$ das duas listas que separamos. Os métodos de complexidade aprendidos na A1 nos levam a uma solução simples de $O(n(\log(n))^2)$, já que sabemos que essa divisão pela metade traz um peso de $\log(n)$.

Seria possível trazer uma otimização ainda maior? Sim, se eliminássemos a etapa de ordenação explícita. Podemos fazer isso se, no algoritmo de contagem de inversão, além de contar, invertessemos e realizássemos o merge, trazendo o array ordenado direto.

Novo algoritmo:

```
1 CountInversions ( $A$ ) :  
2   se  $\text{len}(A) = 1$  :  
3     | retorne 0,  $A$   
4   divida a lista em  $L$  e  $R$   
5    $i_l, L = \text{CountInversions}(L)$   
6    $i_r, R = \text{CountInversions}(R)$   
7    $i = \text{Combine}(L, R)$   
8    $A = \text{Merge}(L, R)$   
9   retorne  $(i_l + i_r + i), A$ 
```

Aqui, nós nos aproveitamos da recursão para fazer a ordenação, em vez de fazer isso por outro algoritmo. Dado que a recursão vai até quando só tivermos um elemento, e que sempre fazemos um merge, a lista está garantidamente sempre ordenada. Isso permite a execução do Combine sem problemas.

A função Combine é $O(n)$, já que apenas conta a quantidade de inversões que precisaríamos. A função Merge também é $O(n)$, já que apenas junta as duas listas. Portanto, temos:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) = n \log(n) \quad (3)$$

Implementação em python

Para essa implementação, vamos relembrar basicamente a função MergeSort, só que contando a quantidade de inversões. Ainda, vamos juntar as funções CountInversions e a Merge numa mesma.

```
1 def combine_merge(v, start_a, start_b, end_b):  
2     r = [0] * (end_b - start_a)  
3     a_idx = start_a  
4     b_idx = start_b  
5     r_idx = 0  
6     num_inv = 0 #adicionado  
7  
8     while a_idx < start_b and b_idx < end_b:  
9         if v[a_idx] <= v[b_idx]:  
10             r[r_idx] = v[a_idx]  
11             a_idx += 1  
12         else:  
13             num_inv += start_b - a_idx #adicionado (explicado anteriormente)  
14             r[r_idx] = v[b_idx]
```

```

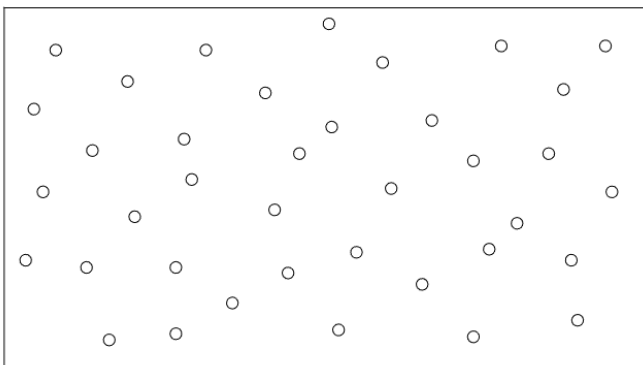
15         b_idx += 1
16         r_idx += 1
17
18     while a_idx < start_b:
19         r[r_idx] = v[a_idx]
20         a_idx += 1
21         r_idx += 1
22
23     while b_idx < end_b:
24         r[r_idx] = v[b_idx]
25         b_idx += 1
26         r_idx += 1
27
28     for i in range(len(r)):
29         v[start_a + i] = r[i]
30
31     return num_inv #adicionado
32
33 def count_inversions(v, start_idx, end_idx):
34     if (end_idx - start_idx) > 1:
35         mid_idx = (start_idx + end_idx) // 2
36         il = count_inversions(v, start_idx, mid_idx) #essas linhas mudaram
37         ir = count_inversions(v, mid_idx, end_idx) #apenas a igualdade
38
39         i = combine_merge(v, start_idx, mid_idx, end_idx) #antes não tinha
40         return il + ir + i #adicionado
41     else:
42         return 0

```

Todo o código (a menos de linhas comentadas) foi tirado da versão original do MergeSort. Como dito, fizemos o MergeSort contando a quantidade de inversões. :)

1.2.2 O problema de pares mais próximos

Dado uma sequência com n pontos em um plano, encontre o par com a menor distância euclidiana.



A primeira solução que vem a cabeça é simplesmente testar cada par com cada outro par, trazendo uma complexidade de $O(n^2)$

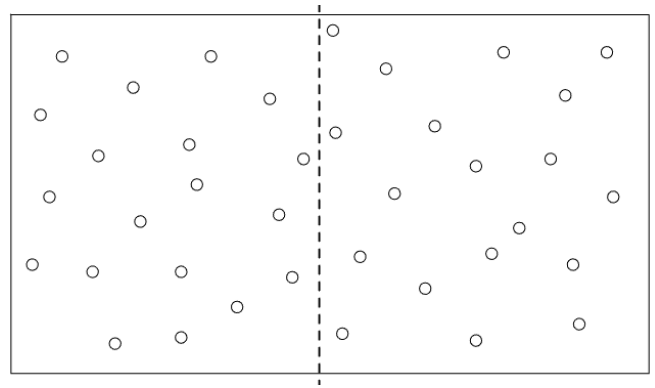
Como desenvolver uma solução melhor com o método que aprendemos?

Figura 10: Exemplificação do problema de pares mais próximos

Podemos dividir o plano de forma que cada lado tenha aproximadamente o mesmo número de pontos (ordenando pelo eixo x).

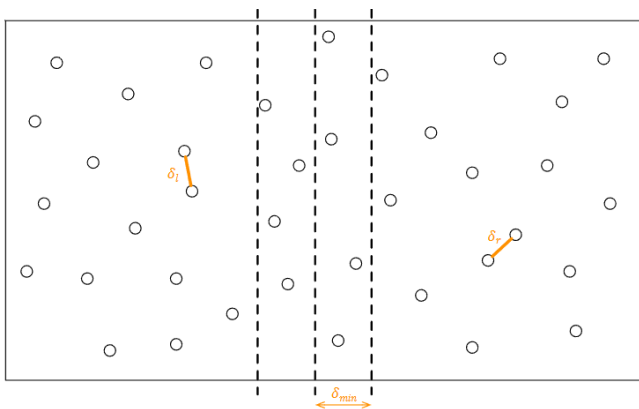
Em seguida, resolva cada lado encontrando o par mais próximo recursivamente.

Figura 11: Exemplificação da solução do problema de pares mais próximos



Com o plano dividido, combine os resultados comparando o par mais próximo no lado direito, o par mais próximo do lado esquerdo, e o par mais próximo em cada lado. A última comparação parece exigir $\Theta(n^2)$, não parece muito bom.

Se pensarmos apenas na comparação da divisão dos planos, sejam δ_l e δ_r os pares com menor distância nos lados esquerdo e direito, respectivamente.



Como estamos procurando o par mais próximo, seja $\delta_{\min} \leq \min(\delta_l, \delta_r)$ (sabemos que δ_{\min} está restrito a, no máximo, essa distância).

Ideia: procurar somente os pontos que estejam no máximo à δ_{\min} da divisória, ordenando os pontos na faixa $2\delta_{\min}$ pela posição o eixo y.

Qual seria a complexidade desse algoritmo?

Figura 11: Exemplo da distância de comparação.

Bom, não seria $O(n^2)$, pois a distância em cada lado é no mínimo δ_{\min} .

1.2.3 como faz isso cara como é 11 7, 5 sla

1.2.4 Implementação em Python

1.3 Programação Dinâmica

O paradigma de programação dinâmica consiste em quebrar em sub-problemas menores e resolvê-los de forma independente. Semelhante ao dividir e conquistar, porém com foco em sub-problemas que usam repetição. Nessa técnica, um sub-problema só é resolvido caso não tenha sido resolvido antes (caso contrário é usado o resultado anterior guardado previamente).

1.3.1 O problema de Fibonacci

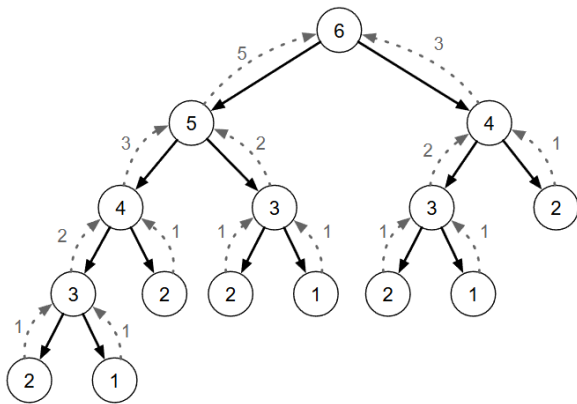


Figura 12: Exemplo de como seria fib(6)

Dado um inteiro $n \geq 1$, encontre F_n . Solução recursiva (e ineficiente):

```
1 def fib(n):  
2     if n <= 2:  
3         return 1  
4     return fib(n-1) + fib(n-2)
```

Note como, para n , o tempo de execução é exponencial, e que, grande parte dos problemas são re-computados. Podemos utilizar um cachê para reaproveitar resultados. Vamos fazer isso!

Solução Top-Down (recursiva):

```
1 def Fib(n):  
2     if n == 0:  
3         return 0  
4     if n == 1:  
5         return 1  
6  
7     F = [-1] * (n + 1)  
8     F[0], F[1], F[2] = 0, 1, 1  
9  
10    def FibAux(k):  
11        if F[k] == -1:  
12            F[k] = FibAux(k - 1) + FibAux(k-2)  
13        return F[k]  
14  
15    return FibAux(n)
```

1	2	3	4	5	6
1	1	-1	-1	-1	-1
1	1	2	-1	-1	-1
1	1	2	3	-1	-1
1	1	2	3	5	-1
1	1	2	3	5	8

Figura 13: Exemplo de como fica o cachê para Fib(6)

A complexidade desse algoritmo é $\Theta(n)$, sabendo que usamos apenas a lista para guardar os valores da lista de Fibonacci.

Solução Bottom-Up (iterativa):

```
1 def Fib(n):  
2     if n <= 2:  
3         return 1  
4     F = [0] * n  
5     F[1], F[2] = 1, 1  
6     for i in range(3, n + 1, 1):  
7         F[i] = F[i - 1] + F[i - 2]  
8     return F[n]
```

A complexidade é a mesma da recursiva, desde que continuamos usando apenas um for e usando-a para guardar apenas uma lista.

A abordagem Top-Down é recursiva, somente executa a recursão caso o sub-problema não tenha sido resolvido e inicia no problema maior, enquanto em problemas Bottom-Up a solução é iterativa e resolve os sub-problemas do menor para o maior. Além disso, em geral apresentam a mesma complexidade.

1.3.2 O problema da mochila (não fracionária)

Dado um conjunto de itens $\mathbb{I} = \{1, 2, 3, \dots, n\}$ em que cada item $i \in \mathbb{I}$ tem um peso w_i e um valor v_i , e uma mochila com capacidade de peso W , encontre o subconjunto $S \subseteq \mathbb{I}$ tal que $\sum_{i \in S} w_i \leq W$ e $\sum_{i \in S} v_i$ seja máximo.

A diferença desse problema para o que vimos no paradigma Guloso é que agora, não podemos pegar uma fração do item, apenas ou o pegamos ou não. Isso faz com que, agora, por mais que o item seja o mais valoroso possível na proporção valor/peso, ainda assim possa existir alguma outra combinação que tenha um valor maior.

Exemplo:

$W = 11$

A escolha $\{1, 2, 4\}$ tem peso 9, valor 29 e cabe na mochila.

A escolha $\{3, 5\}$ tem peso 12, valor 46 e não cabe na mochila.

Item	Peso	Valor
1	1	1
2	2	6
3	5	18
4	6	22
5	7	28

Figura 14: Tabela auxiliar para exemplo

Solução(ineficiente): criar um algoritmo de força bruta que testa todas as possibilidades e escolhe a que cabe na mochila com maior valor.

Tentando usar o que estamos aprendendo aqui (programação dinâmica), temos:

- para cada item i , considere a possibilidade de adicioná-lo ou não a mochila;
 - se adicionado, o valor é incrementado de v_i e a capacidade é reduzida de w_i
 - avalie qual o melhor valor obtido em cada caso.

Após considerar esse item, restam $n - 1$ itens disponíveis para serem avaliados (encontramos a sub-estrutura ótima).

Ideia geral (sem cachê):

```

1 Mochila ( $I, v, w, W$ ) :
2   se  $|I| = 0$  ou  $W = 0$ 
3     | retorne 0
4   escolha um item  $i \in I$ 
5   se  $w_i > W$  :
6     | retorne Mochila ( $I - i, v, w, W$ ) :
7   value_using =  $v_i + \text{Mochila}$  ( $I - i, v, w, W - w_i$ )
8   value_not_using = Mochila ( $I - i, v, w, W$ )
  
```


9 | **retorna** $\max\{\text{value_using}, \text{value_not_using}\}$

Vamos para as soluções definitivas, usando o paradigma que estamos aprendendo. A ideia, como temos que fazer uma comparação a cada item que podemos pegar com e sem ele, é usar uma matriz $I \times W$, onde o valor de cada célula $M[i][w]$ responde a seguinte pergunta: Qual é o valor máximo que consigo obter usando apenas os itens de 1 até i , com uma mochila de capacidade máxima w (não W).

Solução Top-Down:

```

1 Mochila ( $n, v, w, W$ ) :
2   crie uma matriz  $n \times W$ 
3   para  $i = 0$  até  $W$ :
4      $M[0][i] = 0$ 
5     para  $j = 1$  até  $n$ :
6        $M[j][0] = 0$ 
7        $M[j][i] = -1$ 
8   retorna MocilhaAux( $n, v, w, W$ )

```

$W = 11$

i	w_i	v_i	0	1	2	3	...
0			0	0	0	0	...
1	1	1	0	-1	-1	-1	...
2	2	6	0	-1	-1	-1	...
3	5	18	0	-1	-1	-1	...
4	6	22	0	-1	-1	-1	...
5	7	28	0	-1	-1	-1	...

onde n é o total de itens.

Figura 15: Exemplo da matriz para o algoritmo Top-Down e valores anteriores.

Continuação da solução:

```

1 MochilaAux ( $i, v, w, W$ ) :
2   se  $M[i][W] = -1$ :
3     se  $w_i > W$ :
4        $M[i][W] = \mathbf{MochilaAux}(i - 1, v, w, W)$ 
5     caso contrário:
6        $\text{using} = v_i + \mathbf{MochilaAux}(i - 1, v, w, W - w_i)$ 
7        $\text{not\_using} = \mathbf{MochilaAux}(i - 1, v, w, W)$ 
8        $M[i][W] = \max\{\text{using}, \text{not\_using}\}$ 
9   retorna  $M[i][W]$ 

```

Onde i é o item que estamos considerando no momento. Essa solução usa a ideia explicada anteriormente, de fazer a verificação entre o melhor caso, adicionando e não adicionando. Vamos agora para a solução Bottom-Up:

```

1 Mochila ( $n, v, w, W$ ) :
2   crie uma matriz  $n \times W$ 
3   para  $i = 0$  até  $W$ :
4      $M[0][i] = 0$ 
5   para  $j = 1$  até  $n$ :
6      $M[j][0] = 0$ 
7   para  $j = 1$  até  $n$ :
8     para  $i = 1$  até  $W$ :
9       se  $w_j > i$  :

```

```

10 | | | |  $M[j][i] = M[j - 1][i]$ 
11 | | | | caso contrário:
12 | | | |  $\text{using} = v_j + M[j - 1][i - w_j]$ 
13 | | | |  $\text{not\_using} = M[j - 1][i]$ 
14 | | | |  $M[j][i] = \max\{\text{using}, \text{not\_using}\}$ 
15 | retorna  $M[n][W]$ 

```

Sabendo que toda a análise e o algoritmo é baseado na criação da matriz, onde dentro da criação de cada item acontecem apenas verificações, então a complexidade $\Theta(nW)$ (o que **não** é polinomial, já que W é um tamanho, não um valor). Vamos ver agora como essa matriz ficaria no final:

$W = 11$			$W + 1$											
i	w_i	v_i	0	1	2	3	4	5	6	7	8	9	10	11
0			0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
2	2	6	0	1	6	7	7	7	7	7	7	7	7	7
3	5	18	0	1	6	7	7	18	19	24	25	25	25	25
4	6	22	0	1	6	7	7	18	22	24	28	29	29	40
5	7	28	0	1	6	7	7	18	22	28	29	34	34	40

Figura 16: Resultado final da matriz finalizando o primeiro exemplo da mochila fracionária.

Lembre qual a função da matriz: o índice i (na linha) representa que podemos pegar qualquer dos itens 1 até i , e o peso W (na coluna) é o peso w que foi escolhido, e o número no índice $n \times w$ é o valor que conseguimos nessa combinação. Portanto, podemos interpretar que, na segunda linha, na coluna de $w = 0$, temos 0 itens para ser colocados e podemos colocar até um peso 0, logo, o valor máximo é 0. Ao continuar dessa linha, conseguimos ver que, a partir de quando o peso fica ≥ 1 , conseguimos colocar o único item liberado (1), com peso 1 e valor 1. Por isso, toda a segunda linha é igual a 1 a partir do momento que $w \geq 1$.

Nota: Seguindo esse raciocínio, você, caro leitor, pode verificar cada valor da tabela. Existe **um** erro na tabela. Convido a você interpretá-la e entendê-la e encontrar o erro. Se quiser validar que encontrou o erro, mande uma mensagem (Thalis).

Para finalizar, precisamos definir quais itens devem ser adicionados à mochila:

i	w_i	v_i	0	1	2	3	4	5	6	7	8	9	10	11
0			0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
2	2	6	0	1	6	7	7	7	7	7	7	7	7	7
3	5	18	0	1	6	7	7	18	19	24	25	25	25	25
4	6	22	0	1	6	7	7	18	22	24	28	29	29	40
5	7	28	0	1	6	7	7	18	22	28	29	34	34	40

```

1  $S, i, j = \{\}, W, n$ 
2 enquanto  $j \geq 1$ :
3     se  $M[j][i] = M[j-1][i - w_j] + v_j$ 
4          $S = S \cup \{j\}$ 
5          $i = i - w_j$ 
6      $j = j - 1$ 
7 retorna  $S$ 

```

Figura 17: Exemplo da busca dos itens adicionados (as células pintadas de laranja são as células visitadas pelo algoritmo)

Vamos entender o código: j itera nas linhas, W nas colunas. Em teoria, a última célula da matriz ($n \times W$) carrega com certeza o maior valor que satisfaz a condição do problema, e por isso começamos por ela. O que estamos fazendo é verificar se $M[j][i] = M[j-1][i - w_j] + v_j$, ou seja, se o valor da célula voltando o peso do item atual (supondo que ele foi adicionado) e voltando um item ($j-1$) somado ao valor de v_j é igual ao valor da célula atual, pois, se isso for verdade, significa que adicionamos esse valor ao descobrir o item j .

Vamos olhar para o exemplo da tabela:

- Ponto de partida: $M[5][11]$. Valor = 40. O item 5 (peso 7, valor 28) foi usado para obter esse valor de 40?
 - ▶ Comparamos o valor atual ($M[5][11] = 40$) com o valor da célula de cima ($M[4][11] = 7 + v_j = 7 + 28 \neq 40$).
 - ▶ Como os valores não são iguais, significa que o item 5 **não** foi incluído. A solução ótima para capacidade 11 já existia sem ele.
 - ▶ Então o algoritmo “sobe” para a célula $M[4][11]$.
- Posição Atual: Célula $M[4][11]$. Valor = 40. O item 4 (peso 6, valor 22) foi usado?
 - ▶ Comparamos o valor atual ($M[4][11] = 40$) com o valor da célula de cima ($M[3][11] = 18 + v_j = 18 + 22 = 40$).
 - ▶ Os valores são iguais. Isso significa que o item 4 **foi** incluído!
 - ▶ Adicionamos o item 4 ao nosso conjunto de solução S . O algoritmo “sobe” para a linha anterior ($i = 3$) e “anda para a esquerda” subtraindo o peso do item 4 da capacidade: $11 - 6 = 5$. O novo ponto de análise é $M[3][5]$.

E assim sucessivamente!

Implementação em Python

```

1 def bag_problem_bottom_up(n, v, w, W):
2     #primeira parte (criar a matriz e inserir os valores)
3     M = [[0] * (W + 1) for _ in range(n + 1)]
4     for j in range(1, n + 1):
5         for i in range(1, W + 1):
6             peso_item_j = w[j-1]
7             valor_item_j = v[j-1]
8             if peso_item_j > i:
9                 M[j][i] = M[j-1][i]

```

py

```

10         else:
11             not_using = M[j - 1][i]
12             using = valor_item_j + M[j - 1][i - peso_item_j]
13             M[j][i] = max(using, not_using)
14
15     #segunda parte (identificar os itens selecionados)
16     itens_selecionados = []
17     valor_maximo = M[n][W]
18     j, i = n, W
19     while j > 0 and i > 0:
20         peso_item_j = w[j-1]
21         valor_item_j = v[j-1]
22         if peso_item_j <= i and M[j][i] == (M[j- 1][i - peso_item_j] + valor_item_j):
23             itens_selecionados.append(j)
24             i -= peso_item_j
25             j -= 1
26     itens_selecionados.reverse()
27     return valor_maximo, M, itens_selecionados

```

Convido o caro leitor a implementar a solução Top-Down.

Grafos

Sinceramente, fazer uma revisão de conceitos muito básicos de grafos que já vimos em Matemática Discreta é um pouco chato, para não dizer desnecessário. Irei relembrar alguns termos e definir outros que não me lembro de termos vistos. Vamos lá:

2.1 Relembrando conceitos

- $V \rightarrow$ vértices;
- $E \rightarrow$ arestas;
- Uma aresta é definida pelo par (v_i, v_j) ;
- O **tamanho** de um grafo é definido por $|V| + |E|$, onde $|\cdot|$ é a cardinalidade (quantidade de elementos);
- Dado $e = (v_i, v_j)$, v_i e v_j são **extremos** da aresta se e é incidente em v_i e v_j , e v_i e v_j é incidente em e ;
- Vértices relacionados por uma aresta são **adjacentes**;
- Duas arestas são **paralelas** se incidem ao mesmo vértice;
- **Laço** $= (v_i, v_i)$;
- $\sum_{i=1}^{|V|} g(v_i) = 2 |E|$, onde $g(v_i)$ é o **grau** do vértice i ;
- Um grafo é **completo** se cada vértice possuir todos os demais adjacentes à ele;
- O número de arestas em um grafo completo é definido por: $\frac{|V|(|V|-1)}{2}$ (observe que isso é ligeiramente menor que $\frac{(|V|)^2}{2}$);
- Um grafo é **regular** se todos os vértices possuírem o mesmo grau (ou k -regular, para grau k);
- O número de arestas em um grafo k -regular é $|V| \frac{k}{2}$
- Um grafo é **denso** se o seu tamanho for proporcional ao quadrado do número de vértices ($|V| + |E| \propto |V|^2$), e é **esparso** se $(|V| + |E|) \propto |V|$.
- O grafo $H = G(V', E')$ é um **subgrafo** de $G = (V, E)$ se $V' \subseteq V$ e $E' \subseteq E$.
- O grafo $H = G(V', E')$ é um **subgrafo gerador** de $G = (V, E)$ se H for um subgrafo de G e $V' = V$.

Exemplo:

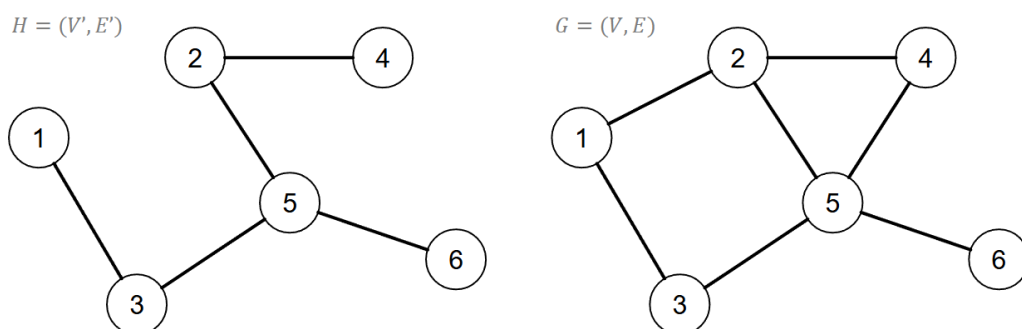


Figura 18: Exemplo de subgrafo gerador.

- O grafo $H = G(V', E')$ é um **grafo induzido** de $G = (V, E)$ se E' for definido por todas as arestas de E adjacentes a um par de vértices V' .

Exemplo:

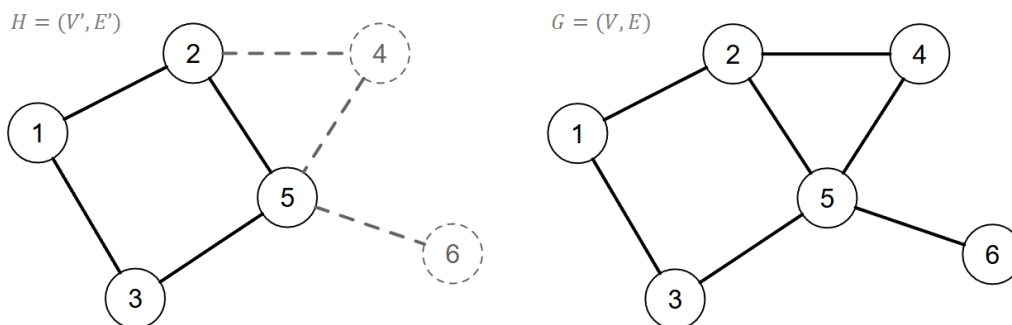


Figura 19: Exemplo de grafo induzido (os vértices escolhidos foram $\{1, 2, 3, 5\}$ e as arestas(e vértices) que não são desses vértices não aparecem no subgrafo induzido).

- O grafo $H = G(V', E')$ é um **grafo próprio** de $G = (V, E)$ se $H \subset G$.

Exemplo:

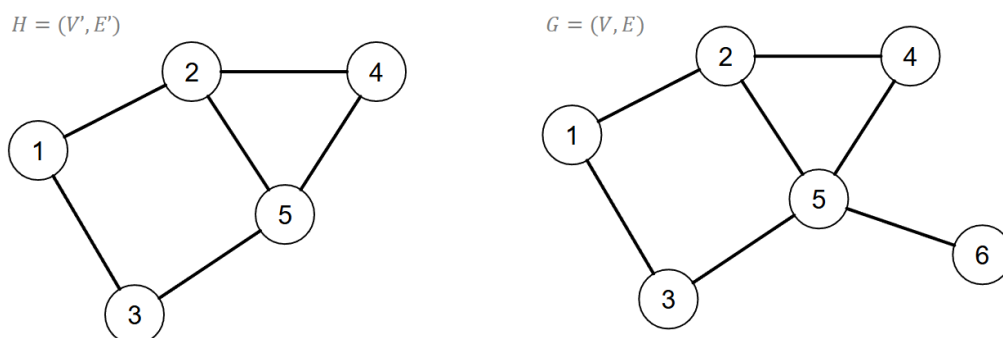


Figura 20: Exemplo de grafo próprio (note que é \subset , não \subseteq . Então, um subgrafo próprio é um subgrafo menor, e não igual ao grafo original).

- Um **caminho** P em $G(V, E)$ consiste em uma sequência de n vértices, finita e não vazia tal que v_{i+1} é adjacente a v_i .
- Um caminho é **simples** se não possuir vértices repetidos.
- Um caminho é **fechado** se $v_1 = v_n$.
- O **comprimento** de um caminho é definido pelo número de arestas do caminho.
- Um grafo $G = (V, E)$ é **conexo** se para qualquer par de vértices existe um caminho em G .
- Quando um grafo não é conexo podemos segmentá-lo em **componentes conexos** (um par está no mesmo componente se existe um caminho).
- Um grafo $G(V, E)$ é uma **árvore** se G for conexo e acíclico (possui $|V| - 1$ arestas, a remoção de qualquer aresta torna o grafo não-conexo e para todo par de vértices existe um único caminho);
- Um grafo $G(V, E)$ é uma **floresta** se for um grafo acíclico;
- Um grafo é **planar** se puder ser representado graficamente em um plano de tal forma que não haja cruzamento de arestas;
- Um grafo $G(V, E)$ é **bipartido** se os vértices puderem ser divididos em dois conjuntos V_1 e V_2 de forma que toda aresta e_k é incidente em (v_i, v_j) tal que $v_i \in V_1$ e $v_j \in V_2$;
- Um grafo $G(V, E)$ é **orientado** se as arestas possuírem um sentido. Nesse caso, a nomenclatura que definimos (v_i, v_j) significa que ela começa em v_i e termina em v_j .
- O grau de **saída** $g_s(v_i)$ é definido pelo número de arestas que saem de v_i . Raciocínio análogo para grau de **entrada** $g_e(v_i)$;
- $\sum_{i=1}^{|V|} g_e(v_i) = \sum_{i=1}^{|V|} g_s(v_i) = |E|$;
- O vértice v_i é uma **fonte** se $g_e(v_i) = 0$;
- O vértice v_i é um **sorvedouro** se $g_s(v_i) = 0$;
- O vértice v_i é **isolado** se for sorvedouro e fonte;

- Um grafo (orientado ou não) é **ponderado** se cada aresta estiver associado a um peso;

2.2 Estruturas de dados para representar grafos

Dependendo do problema, a escolha da estrutura pode variar, e, em geral, usamos duas formas de implementar essa representação:

2.2.1 Matriz de adjacência

Consiste em um matriz quadrada A de ordem $|V|$ cujas linhas e colunas são indexadas pelos vértices de V . Exemplo para grafos orientados:

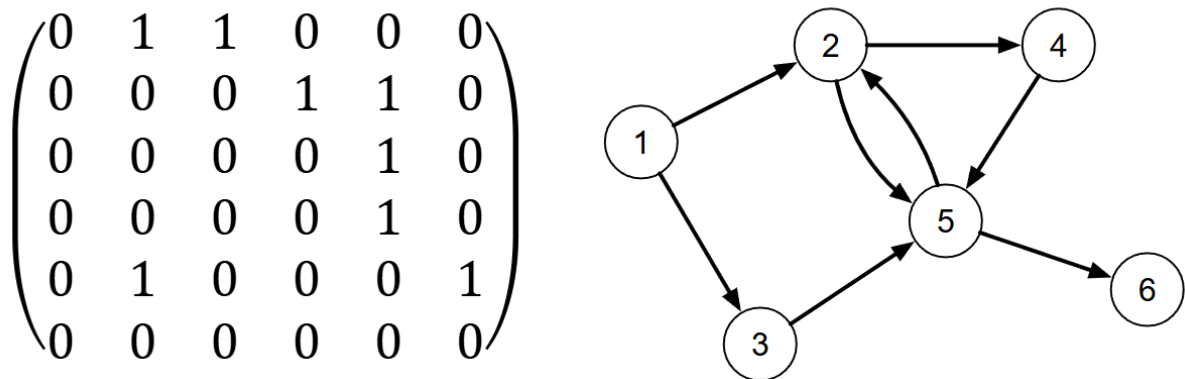


Figura 21: Exemplo de matriz de adjacência para o grafo à direita.

Analogamente, para não orientados:

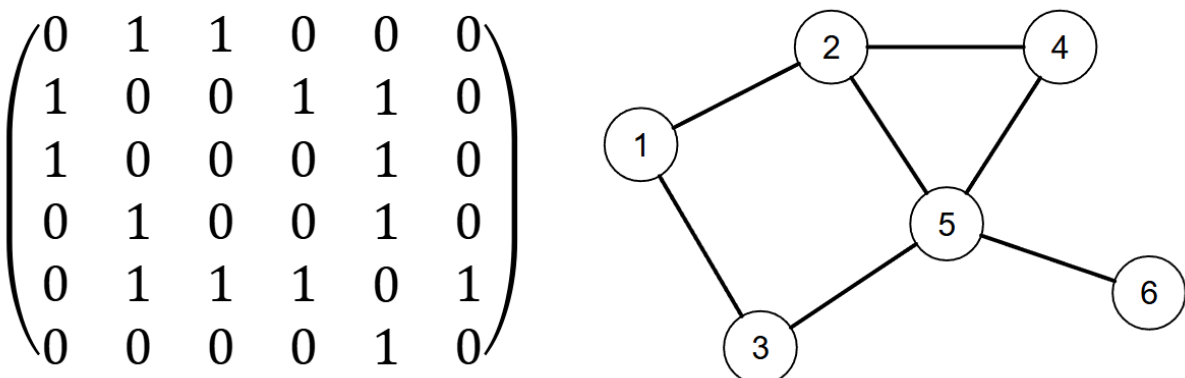


Figura 22: Exemplo de matriz de adjacência para o grafo à direita. Nota: a matriz é simétrica!

A complexidade de acessar(ou verificar) uma aresta é $\Theta(1)$, e claramente conta com uma complexidade de espaço de $\Theta(|V|^2)$. Além disso, o fato da matriz ser simétrica para grafos não-orientados faz com que o tamanho se reduza para a metade, podendo se armazenar apenas a diagonal superior ou inferior da matriz.

2.2.2 Lista de adjacência

Consiste em uma sequência de vértices contendo na estrutura de cada ponteiro para uma lista encadeada com elemento representando as arestas adjacentes ao vértices. Exemplo para grafo dirigido:

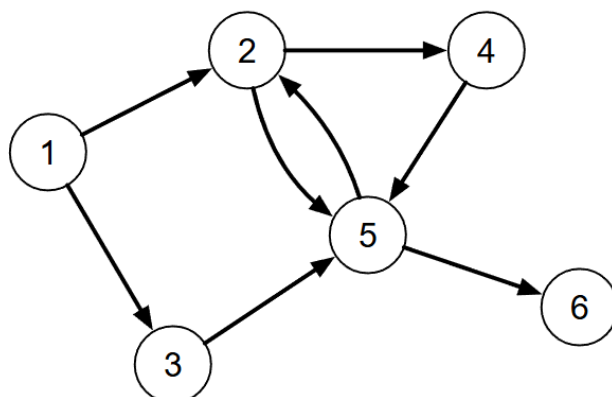
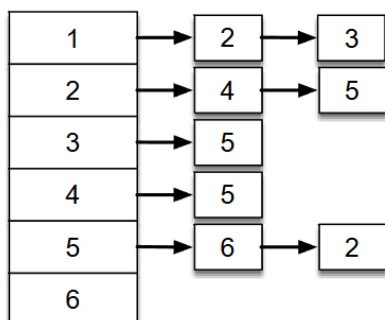


Figura 23: Exemplo da lista de adjacência para o grafo à direita.

Exemplo para grafo não-dirigido:

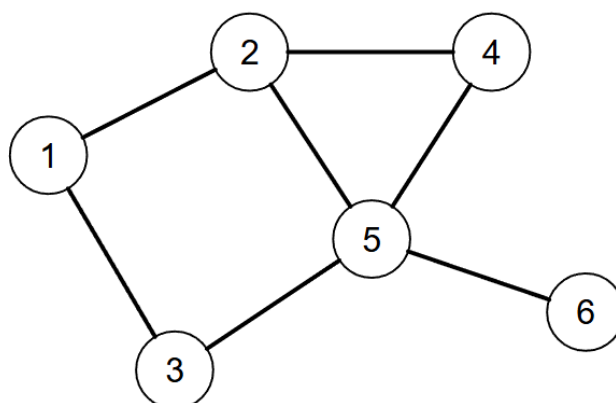
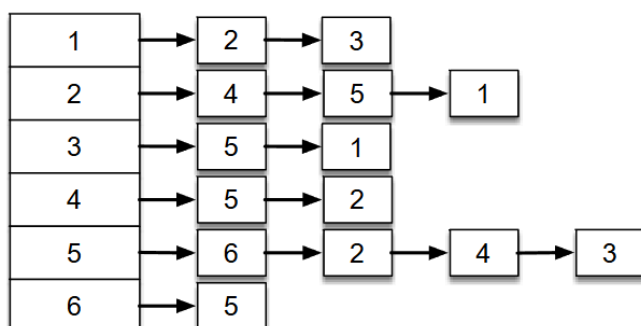


Figura 24: Exemplo da lista de adjacência para o grafo à direita.

A complexidade de acessar o conjunto de arestas de um vértice é $\Theta(1)$ (mas encontrar uma aresta específica é $\Theta(|V|)$ no pior caso). Ainda, uma lista de adjacência exige um espaço $\Theta(|V| + |E|)$.

As estruturas de dados do vértice e da aresta podem ser estendidas para armazenar informações específicas do problema.

Nota: Os exercícios passados no slide não serão feitos aqui (pois isso é um “resumo” teórico), e sim na pasta Exercises.

Busca em Grafos

Após relembrar conceitos e aprender algumas estruturas, temos um novo problema:

Dado um par de vértices v_i e v_j verificar se v_j pode ser alcançado iniciando um caminho em v_i .

Como resolver esse problema?

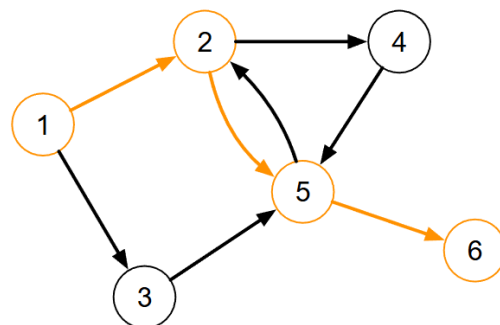


Figura 25: Exemplo do caminho $P = \{1, 2, 5, 6\}$

A ideia principal é passar pelo grafo e marcar cada nó e aresta visitada, voltando ao nó anterior se chegar a uma junção já visitada ou sorvedouro. Podemos escrever um algoritmo que percorre o grafo a partir do vértice v_i armazenando os vértices visitados, e, ao final, verificar se v_j foi visitado. Exemplo dessa ideia em Python para a estrutura de matriz de adjacência:

```
1 def _reach_recursive_matrix(v_atual, visited, matrix, num_vertices):
2     visited[v_atual] = True
3     for v_vizinho in range(num_vertices):
4         if matrix[v_atual][v_vizinho] == 1 and not visited[v_vizinho]:
5             _reach_recursive_matrix(v_vizinho, visited, matrix, num_vertices)
6
7 def can_reach_matrix(matrix, v1, v2):
8     num_vertices = len(matrix)
9     visited = [0] * num_vertices
10    _reach_recursive_matrix(v1, visited, matrix, num_vertices)
11    return visited[v2]
```

Como seria a execução desse algoritmo no que grafo que acabamos de ver?

(1, 2)	100000
(2, 4)	110000
(4, 5)	110100
(5, 2)	110110
(5, 6)	110110
(2, 5)	110111
(1, 3)	110111
(3, 5)	111111

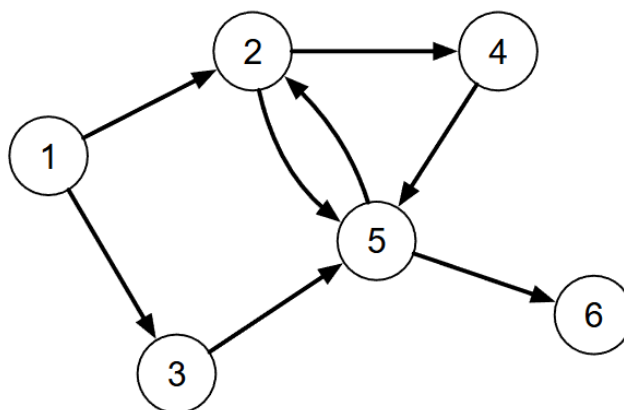


Figura 26: Exemplo da execução do algoritmo `can_reach` para o mesmo grafo

À esquerda temos a aresta escolhida e ao lado a iteração anterior (começando do vértice 1). Observe que o resultado também indica todos os demais vértices que podem ser alcançados a partir de v_i .

Um **algoritmo de busca** em grafo é qualquer algoritmo que visita todos os vértices percorrendo as arestas definidas (a ordem de pesquisa depende do algoritmo).

3.1 DFS

O algoritmo de **busca em profundidade (Depth First Search)** consiste em visitar todos os vértices ao menos uma vez e levantar propriedades sobre a estrutura do grafo

Vamos começar identificando a ordem de descoberta dos vértices (ou pré-ordem). Em **C++**:

Esse código assume a existência de tipos como 'vertex', 'EdgeNode' e variáveis de membro como 'm_numVertices' e 'm_edges', que seriam parte de uma classe de Grafo. Ele usa algumas funções que não definiu antes, e eu não vou ficar tentando entender isso agora :p

```
1 void dfs(int * preOrder) {
2     int counter = 0;
3     for (vertex v=0; v < m_numVertices; v++) {
4         preOrder[v] = -1;
5     }
6
7     for (vertex v=0; v < m_numVertices; v++) {
8         if (preOrder[v] == -1) {
9             dfsRecursive(v, preOrder, counter);
10        }
11    }
12 }
13
14 void dfsRecursive(vertex v1, int * preOrder, int & counter) {
15     preOrder[v1] = counter++;
16     EdgeNode * edge = m_edges[v1];
17     while (edge) {
18         vertex v2 = edge->otherVertex();
19         if (preOrder[v2] == -1) {
20             dfsRecursive(v2, preOrder, counter);
21         }
22         edge = edge->next();
23     }
24 }
```

Implementação em Python:

Intuitiva com a ideia, inicia a lista de descoberto como -1 e se o vértice é -1 (ainda não descoberto), realiza a busca profunda partindo desse vértice, e atualizando a lista preorder da ordem que foi a partir do primeiro v .

```
1 def dfs_preorder(adj_list):
2     num_vertices = len(adj_list)
3     preorder = [-1] * num_vertices
4     counter = 0
5     for v in range(num_vertices):
6         if preorder[v] == -1:
7             counter = dfs_recursive(v, preorder, counter, adj_list)
8     return preorder
```

```

9
10 def dfs_recursive(v_atual, preorder, counter, adj_list): #Auxiliar
11     preorder[v_atual] = counter
12     counter += 1
13     for v_vizinho in adj_list[v_atual]:
14         if preorder[v_vizinho] == -1:
15             counter = dfs_recursive(v_vizinho, preorder, counter, adj_list)
16     return counter

```

Voltando ao exemplo inicial, veja como seria o algoritmo:

```

(1,3)      0 - - - - -
.(3,5)     0 - 1 - - -
..(5,2)    0 - 1 - 2 -
...(2,5)   0 3 1 - 2 -
...(2,4)   0 3 1 - 2 -
....(4,5)  0 3 1 4 2 -
..(5,6)     0 3 1 4 2 -
(1,2)      0 3 1 4 2 5

```

```

v      = 1 2 3 4 5 6
preOrder[v] = 0 3 1 4 2 5

```

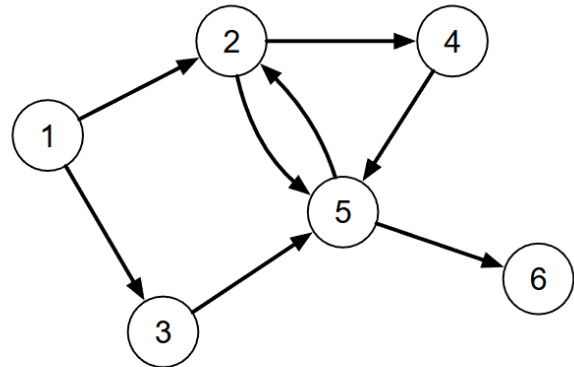


Figura 27: Exemplo da execução do algoritmo DFS para o mesmo grafo

À esquerda, os parênteses indicam as arestas visitadas e a lista da ordem de visita. A interpretação de preorder é: o vértice 1 foi o primeiro a ser visitado, o vértice 3 foi o segundo, o 5 o terceiro e assim sucessivamente.

Dizemos então que um vértice v é **visitado** quando $preOrder[v]$ é consultado e que um vértice v é **descoberto** quando $preOrder[v]$ é definido (a busca pode iniciar em qualquer vértice).

A abordagem de tentar percorrer a partir de cada vértice garante que todos os vértices serão inspecionados, mesmo que o grafo não seja conexo. Além disso, um grafo pode apresentar múltiplas sequências de pré-ordem (depende da ordem em que as arestas são inspecionadas).

Falando de complexidade, sabemos pelo algoritmo que cada vértice será processado uma única vez, e em cada vértice são verificadas cada $g_s(v_i)$ arestas. Sabendo que isso soma $|E|$, fica claro que temos uma complexidade de $\Theta(|V| + |E|)$ usando lista de adjacências, e $\Theta(|V|^2)$ para matriz de adjacência.

3.2 Grafo topológico

Um **grafo topológico** é um grafo que admite uma ordenação dos vértice de forma que para toda aresta (v_i, v_j) temos que $i < j$.

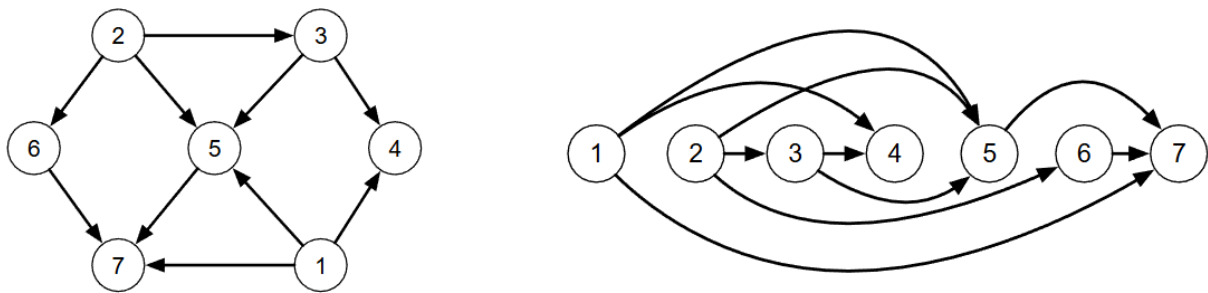


Figura 28: Exemplo da grafo topológico (Note que se os vértices forem dispostos em ordem crescente toda aresta irá apontar para o sentido de crescimento dos números).

Algumas propriedades de grafos topológicos:

- Não apresentam ciclos;
- Todo vértice é:
 - o término de um caminho que começa numa fonte;
 - a origem de um caminho que termina num sorvedouro;
- Se um grafo é topológico, podem existir várias numerações topológicas diferentes;

Como verificar se um grafo $G = (V, E)$ possui numeração topológica e determiná-la?

Podemos eliminar uma fonte $g_e(v_k) = 0$ de G produzindo um subgrafo G' , e repetindo o procedimento sobre ele. Se redumovermos a fonte inicial, isso provavelmente vai criar (caso não tenhamos outra) outra fonte. Se não criar, isso significa que o restante dos vértices estão presos em um ciclo. Numere os vértices removidos, e, se todos eles forem removidos, a numeração é topológica.

Nota: O exercício de como fazer o algoritmo que verifica a topologia do grafo está na pasta Exercises.

Uma **floresta radicada** é um grafo topológico sem vértices com grau de entrada maior que 1

As fontes de uma floresta radicada são as raízes das árvores, e os sorvedouros são folhas.

A floresta gerada pela execução do algoritmo de busca em profundidade também é chamada de floresta DFS (essa floresta é também um grafo gerador).

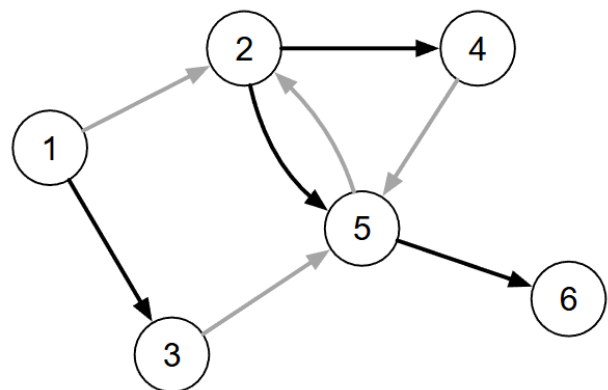


Figura 29: Exemplo de floresta radicada (a raiz no 2 foi proposital)

3.2.1 fazer exercício de numeração topológica

3.3 DFS modificado

Dado que o grau de entrada de cada vértice é no máximo 1, podemos representar a floresta DFS como um vetor de pais (parents). Portanto, o algoritmo pode ser modificado para gerar a árvore DFS da seguinte forma:

Esse código assume a existência de tipos como 'vertex', 'EdgeNode' e variáveis de membro como 'm_numVertices' e 'm_edges', que seriam parte de uma classe de Grafo.

Esse código é bem parecido com o DFS anterior, a menos da marcação para parents.

```
1 void dfs(int * preOrder, int * parents) {
```

C++

```

2     int counter = 0;
3     for (vertex v=0; v < m_numVertices; v++) {
4         preOrder[v] = -1;
5         parents[v] = -1;
6     }
7
8     for (vertex v=0; v < m_numVertices; v++) {
9         if (preOrder[v] == -1) {
10            parents[v] = v;
11            dfsRecursive(v, preOrder, counter, parents, 0);
12        }
13    }
14 }
15
16
17 void dfsRecursive(vertex v1, int * preOrder, int & counter, int * parents, int
level=0) {
18     preOrder[v1] = counter++;
19     EdgeNode * edge = m_edges[v1];
20     while (edge) {
21         vertex v2 = edge->otherVertex();
22         if (preOrder[v2] == -1) {
23             parents[v2] = v1; // Set parent first
24             dfsRecursive(v2, preOrder, counter, parents, level + 1);
25         }
26         edge = edge->next();
27     }
28 }

```

Focando na função `dfs_parents`, e, usando lista de adjacência, criamos a lista de ordem e de pais, e o `counter` (para marcação de pré-ordem) como 0. Para cada item da ordem do vértice, se a pré-ordem for `-1`, ou seja, se não tivermos descoberto o vértice ainda (procurando vértices de partida), então ele é marcado como item de partida (se referenciando `parents[i] = i`). Após isso para cada vértice de partida, iniciamos a marcação.

No `dfs_recursive_parents`, incrementamos o `counter` a cada uso da função (para atualizar o `preorder`), e a cada filho da lista de adjacências, marca o vértice atual como pai (apenas se esse filho não tiver sido visitado, ignorando filhos já visitados por “outros pais”).

```

1  def dfs_recursive_parents(v_atual, preorder, parents, counter, adj_list):
2      preorder[v_atual] = counter
3      counter += 1
4      for v_vizinho in adj_list[v_atual]:
5          if preorder[v_vizinho] == -1:
6              parents[v_vizinho] = v_atual
7              counter = dfs_recursive_parents(v_vizinho, preorder, parents, counter,
adj_list)
8

```

```

9     return counter
10
11 def dfs_parents(adj_list):
12     num_vertices = len(adj_list)
13     preorder = [-1] * num_vertices
14     parents = [-1] * num_vertices
15     counter = 0
16
17     for v in range(num_vertices):
18         if preorder[v] == -1:
19             parents[v] = v
20             counter = dfs_recursive_parents(v, preorder, parents, counter, adj_list)
21     return preorder, parents

```

Como isso funcionaria no exemplo que já vimos até agora?

```

(1, 3)      0 - - - - -
.(3, 5)     0 - 0 - - -
..(5, 2)    0 - 0 - 2 -
...(2, 5)   0 4 0 - 2 -
...(2, 4)   0 4 0 - 2 -
....(4, 5)  0 4 0 1 2 -
..(5, 6)    0 4 0 1 2 -
(1, 2)      0 4 0 1 2 4

      v   = 1 2 3 4 5 6
parent[v] = 1 5 1 2 3 5

```

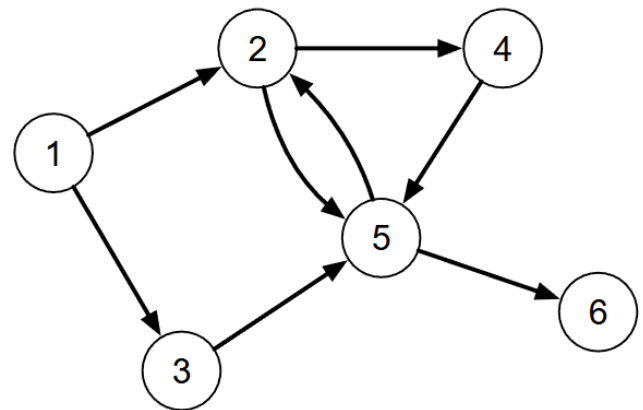


Figura 30: Exemplo do algoritmo dfs_parents para o grafo de exemplo.

um vértice é **exaurido** (essa definição não é minha e não está nos slides do Thiago) no momento em que a busca já explorou todos os caminhos possíveis que saem daquele vértice.

Uma outra informação que podemos gerar a partir da execução de um DFS é a ordem em que os vértices são exauridos (essa sequência é conhecida como pós-ordem).

O algoritmo de DFS pode ser modificado de forma que registre o momento em que o algoritmo termina a avaliação do vértice, da seguinte forma:

```

1 void dfs(int * preOrder, int * postOrder,
2         int * parents) {
3     int preCounter = 0;
4     int postCounter = 0;
5     for (vertex v=0; v < m_numVertices; v++) {
6         preOrder[v] = -1;
7         parents[v] = -1;
8         postOrder[v] = -1;
9     }
10

```



```

11     for (vertex v=0; v < m_numVertices; v++) {
12         if (preOrder[v] == -1) {
13             parents[v] = v;
14             dfsRecursive(
15                 v, preOrder, preCounter,
16                 postOrder, postCounter, parents);
17         }
18     }
19 }
20
21 void dfsRecursive(vertex v1, int * preOrder, int & preCounter, int * postOrder,
22                 int & postCounter, int * parents) {
23     preOrder[v1] = preCounter++;
24     EdgeNode * edge = m_edges[v1];
25     while (edge) {
26         vertex v2 = edge->otherVertex();
27         if (preOrder[v2] == -1) {
28             parents[v2] = v1;
29             dfsRecursive(v2, preOrder, preCounter,
30                         postOrder, postCounter, parents);
31         }
32         edge = edge->next();
33     }
34     postOrder[v1] = postCounter++;
35 }

```

Note que ele é o mesmo algoritmo que o do DFS modificado, a menos de uma declaração da lista de pós-ordem e preenchimento no fim do while, após exaurir o vértice. Note que

```

1  def dfs_recursive_full(v_atual, preorder, postorder, parents, pre_counter,
    post_counter, adj_list):
2      preorder[v_atual] = pre_counter
3      pre_counter += 1
4      for v_vizinho in adj_list[v_atual]:
5          if preorder[v_vizinho] == -1:
6              parents[v_vizinho] = v_atual
7              pre_counter, post_counter = dfs_recursive_full(v_vizinho, preorder,
                postorder, parents, pre_counter, post_counter, adj_list)
8      postorder[v_atual] = post_counter
9      post_counter += 1
10     return pre_counter, post_counter
11
12 def dfs_full(adj_list):
13     num_vertices = len(adj_list)
14     preorder = [-1] * num_vertices
15     postorder = [-1] * num_vertices
16     parents = [-1] * num_vertices

```

```

17     pre_counter = 0
18     post_counter = 0
19     for v in range(num_vertices):
20         if preorder[v] == -1:
21             parents[v] = v
22             pre_counter, post_counter = dfs_recursive_full(v, preorder, postorder,
23                                                         parents, pre_counter, post_counter, adj_list)
23     return preorder, postorder, parents

```

Focando na pós-ordem, como seria a execução desse algoritmo nos grafos que vimos até agora?

```

(1, 3)      - - - - -
. (3, 5)    - - - - -
.. (5, 2)   - - - - -
... (2, 5)  - - - - -
... (2, 4)  - - - - -
.... (4, 5) - - - - -
.. (5, 6)   - 1 - 0 - -
(1, 2)      - 1 4 0 3 2

```

```

          v   = 1 2 3 4 5 6
postOrder[v] = 5 1 4 0 3 2

```

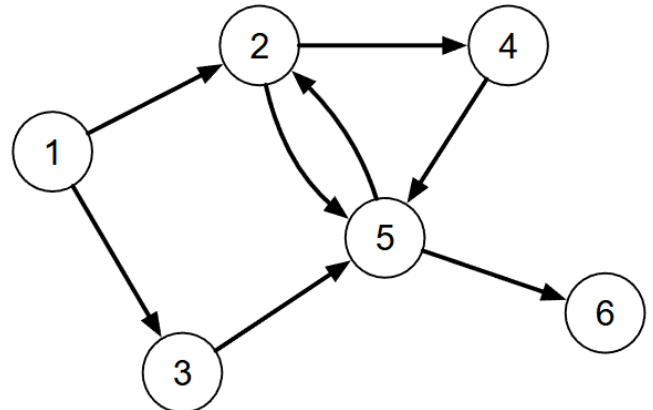


Figura 31: Exemplo do algoritmo `dfs_parents_full` para o grafo de exemplo.

3.4 Propriedades úteis advindas do DFS

Algumas delas já vimos: ordenação topológica, floresta DFS, etc. Vamos ver outras

O **intervalo de vida (lifespan)** de um vértice no contexto da busca ocorre entre o momento que ele é descoberto e o momento em que ele é exaurido. Ele não pode ser definido como $(\text{preOrder}[v], \text{postOrder}[v])$, pois são numerações independentes (isso APENAS no algoritmo passado, normalmente o lifespan é definido dessa forma).

Considere dois vértices v_1 e v_2 .

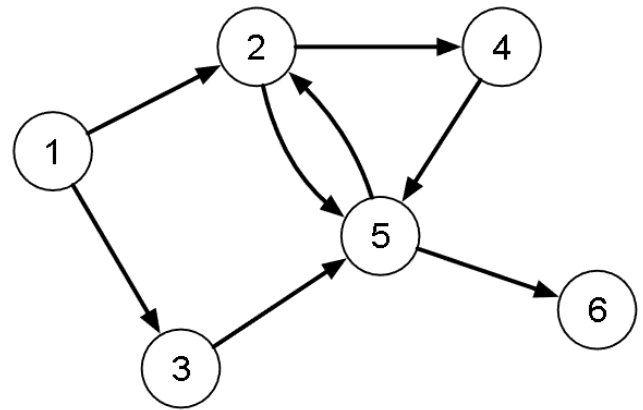
- Se v_1 é descoberto antes de v_2 , então v_1 é exaurido:
 - Antes de v_2 ser descoberto (v_2 não tem nenhum parentesco próximo de v_1 , por isso v_1 e seus filhos são vistos, v_1 é exaurido e só depois v_2 é descoberto);
 - Depois de v_2 ser exaurido (para o caso em que v_2 é filho de v_1 , que acontece porque na chamada recursiva o filho tem que ser limpo primeiro).

Como podemos representar o intervalo de vida da execução do DFS no grafo a seguir?

```

(1, 3)      1
.(3, 5)     1 3
..(5, 2)    1 3 5
...(2, 5)   1 3 5 2
...(2, 4)   1 3 5 2
....(4, 5)   1 3 5 2 4
..(5, 6)     1 3 5
...6         1 3 5 6
(1, 2)      1

```



```

(1 (3 (5 (2 (4) 2) (6) 5) 3) 1)

```

Figura 32: Exemplo do mapeamento do lifespan no grafo de exemplo.

À esquerda temos a aresta escolhida e ao lado a iteração anterior (começando do vértice 1). a listagem à direita da escolha à esquerda é o histórico da chamada de funções para o vértice i . A lista em baixo representa visualmente o lifespan de cada vértice.

Dado dois vértices v_1 e v_2 de uma floresta radicada produzida por uma execução DFS, o relacionamento desses dois vértices pode ser:

- Ancestral: v_1 é ancestral de v_2 se, para chegar em v_2 , o algoritmo DFS “passou por” v_1 primeiro. (lifespan de v_2 contido no lifespan de v_1);
- Descendente: É o oposto de ancestral. v_2 é descendente de v_1 se v_1 for seu ancestral;
- Primo descreve qualquer par de vértices que não tem relação de ancestralidade (lifespans disjuntos).

Primos ainda podem ser comparados:

- v_1 é primo mais velho de v_2 se:
 - $\text{preOrder}[v_1] < \text{preOrder}[v_2]$
- v_1 é primo mais novo de v_2 se:
 - $\text{preOrder}[v_1] > \text{preOrder}[v_2]$

Arestas que não pertencem à floresta DFS podem ser classificados de acordo com o grau do parentesco:

- Uma aresta é de retorno caso v_j seja ancestral de v_i ;
- Uma aresta é de avanço caso v_j seja descendente de v_i ;
- Uma aresta é cruzada caso v_j seja primo de v_i .

Algumas outras características:

- Vértices de arestas cruzadas podem estar em diferentes árvores da floresta;
- Arestas cruzadas são sempre de um primo mais novo para um primo mais velho;
- Grafos não-orientados não possuem arestas cruzadas.

Problema: Dada uma aresta não pertencente à floresta DFS, como determinar algoritmicamente se:

- É uma aresta de avanço:
 - se o intervalo de v_i está contido no intervalo de v_j , ou seja:
 - $\text{preOrder}[v_i] < \text{preOrder}[v_j]$ AND $\text{postOrder}[v_i] > \text{postOrder}[v_j]$
- É uma aresta de retorno:
 - se o intervalo de v_j contém o intervalo de v_i , ou seja:
 - $\text{preOrder}[v_i] > \text{preOrder}[v_j]$ AND $\text{postOrder}[v_i] < \text{postOrder}[v_j]$

- É uma aresta cruzada:
 - se o intervalo de v_j ocorre antes do intervalo de v_i , ou seja:
 - $\text{preOrder}[v_i] > \text{preOrder}[v_j]$ AND $\text{postOrder}[v_i] > \text{postOrder}[v_j]$

3.4.1 explicar visualmente

Algumas outras propriedades:

- Um grafo é acíclico se e somente se possuir uma numeração topológica.
- Grafos acíclicos também são chamados de **DAGs** (Direct acyclic graphs).
- Uma floresta radcada é um DAG sem vértices com grau de entrada maior que 1.
- Uma árvore radcada é um DAG em que exatamente um vértice tem grau de entrada zero, e os demais grau de entrada 1.

Problema: Como determinar se um grafo $G = (V, E)$ possui ao menos um ciclo?

Basta executar a busca DFS e procurar por uma aresta de retorno comparando os intervalos de vida encontrados para cada vértice. Veja o código em C++:

```
1  bool hasCycle(int * preOrder, int * postOrder) {
2  dfs(preOrder, postOrder);
3  for (vertex v1=0; v1 < m_numVertices; v1++) {
4      EdgeNode * edge = m_edges[v1];
5      while(edge) {
6          vertex v2 = edge->otherVertex();
7          if (preOrder[v1] > preOrder[v2]
8              && postOrder[v1] < postOrder[v2]) {
9              return true;
10         }
11         edge = edge->next();
12     }
13 }
14 return false;
15 }
```

3.4.2 explicação e implementação em Python

3.5 BFS

O algoritmo de busca em largura (BFS - Breadth First Search) é uma outra estratégia de varredura em um grafo. A ideia principal é: