FGV EMAp

Escrita: Thalis Ambrosim Falqueto

Projeto e Análise de Algoritmos

Revisão para A2

Rio de Janeiro

2025

Conteúdo

Técnicas de Projeto

1.1 Método guloso (Greedy)

O método guloso é uma 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 $T = \{t_1, t_2, ..., t_n\}$ com n elementos, cada uma com tempo de ínicio `[tk]`, e um tempo de término `[tk]`, encontre o maior subconjunto de tarefas que pode ser alocado sem sobreposição temporal.

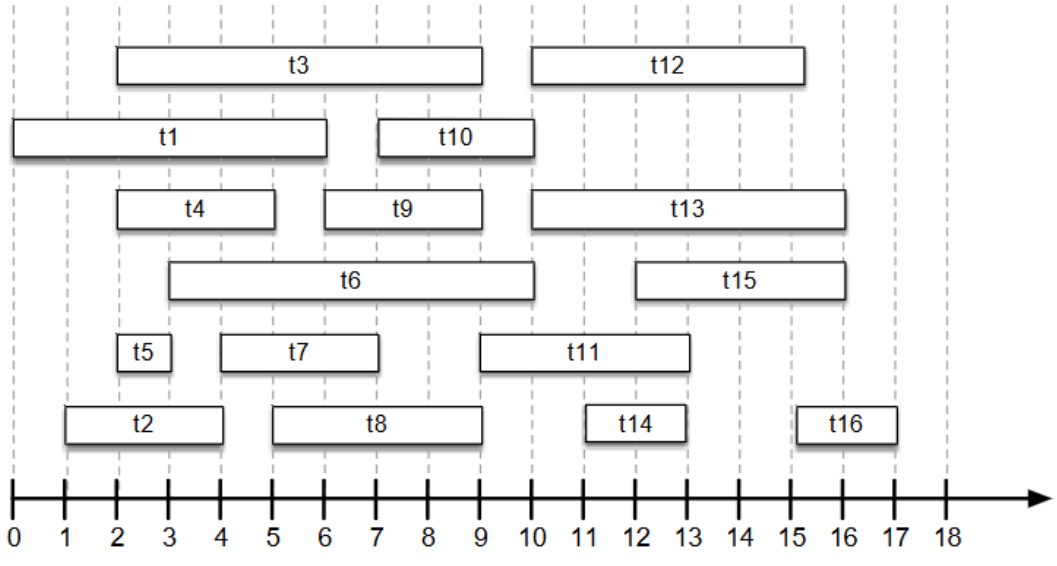


Figura 1: Exemplo do problema de agendamento

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 ínicio?
   - 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?

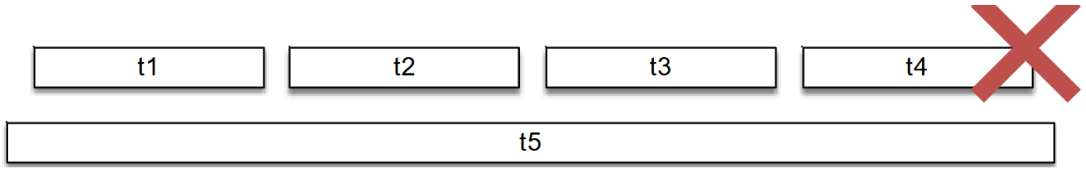


Figura 2: Contra-exemplo para uma possível solução do problema de agendamento

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?

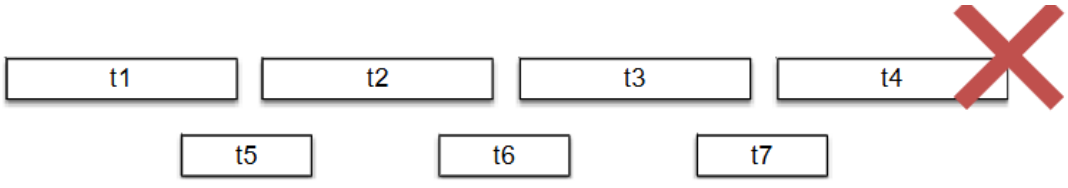


Figura 3: Contra-exemplo para uma possível solução do problema de agendamento

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:

ordene T (pelo tempo de término)
$T_a[1, ..., n] = 0$
Insira a primeira tarefa da lista t_0 **em** T_a
$t_{\text{prev}} = t_0$
para t_k **em** T:
    se $\text{start}[t_k] \geq \text{end}[t_{\text{prev}}]$:
        adicione t_k **em** T_a
        $t_{\text{prev}} = t_k$
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 atulizar 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 $\Theta(n \log(n))$.

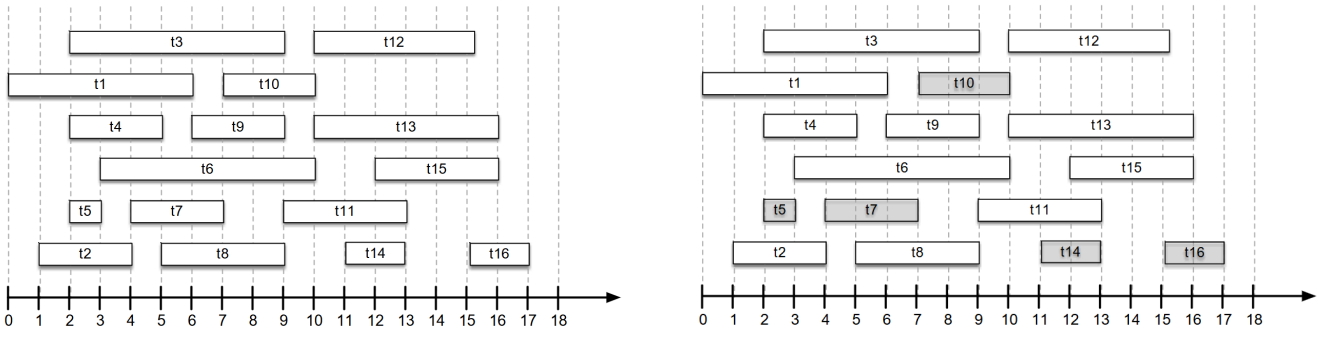


Figura 4: Solução para o problema de tarefas usando o algoritmo proposto

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 algortimo 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 $T_a = \{g_1, g_2, ..., g_k\}$ o conjunto de k tarefas selecionadas pelo nosso algoritmo guloso, já ordenadas pelo tempo de término (como no pseudocódigo). Seja $S = \{s_1, s_2, ..., s_m\}$ uma s*olução 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:

$$\text{end}[g_1] \leq \text{end}[s_1] \tag{1}$$

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': $S' = \{g_1, s_2, s_3, ..., s_m\}$ 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 ($s_3, ...$) 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*.

Concluímos 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:

```py
def scheduling_problem(tasks):
    if len(tasks) == 0:                                  #caso de contorno
        return 0

    sorted_by_end = sorted(tasks, key= lambda x:x[1]) #ordena pelo término
    choosed_tasks = []
    choosed_tasks.append(sorted_by_end[0])
    t_prev = sorted_by_end[0]

    for task in sorted_by_end[1:]:                       #começa depois da primeira
        if task[0] >= t_prev[1]:                         #tempo maior que o de saída
            choosed_tasks.append(task)
            t_prev = task
    return choosed_tasks, len(choosed_tasks)             #retorna lista, quantidade
```

1.1.2 O problema da mochila fracionária

Dado um conjunto de itens $\mathbb{I} = \{1, 2, 3, ..., n\}$ 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 $\sum_{i \in S}^{|S|} \alpha_i w_i \leq W$ e $\sum_{i \in S}^{|S|} \alpha_i v_i$ 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:

Mochila (I, v, w, n, W) :
    ordene I (pela razão valor/peso)
    $C, i = W, 1$
    $M[1, ..., n] = 0$
    enquanto $i \leq n$ e $C \geq w_i$:
        $M[i] = 1$
        $C = C - w_i$
        $i += 1$
    se $i \leq n$:
        $M[i] = \frac{C}{w_i}$
    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.

Analisando 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.

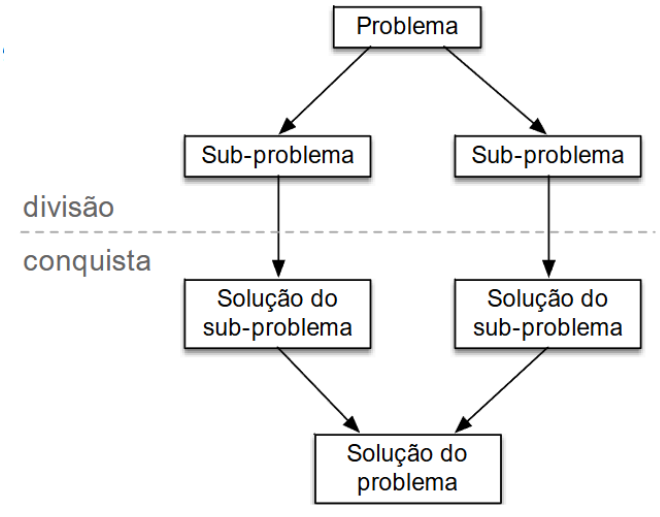
O mais complexo é a ordenação, que pode ser garantido com $\Theta(n \log(n))$.

Implementação em Python:

```py
def fractional_bag_problem(I, v, w, max_w):
    n = len(I)                              #as três listas têm o mesmo tamanho
    idx_w_ratio = []
    for i in range(n):
        ratio = v[i]/w[i]
        idx_w_ratio.append((i, w[i], ratio))#lista que armazena o índice, peso e razão
                                            #ordena por razão logo abaixo
    idx_w_ratio = sorted(idx_w_ratio, key = lambda x: x[2], reverse=True)
    capacity, i = max_w, 0
    M = [0] * n

    while i < n and capacity >= idx_w_ratio[i][1]:
        M[i] = 1                            #faz o while normal
        capacity -= idx_w_ratio[i][1]
        i += 1
    if i < n:
        M[i] = capacity/idx_w_ratio[i][1]

    itens_choosed = [0] * n                 #lista que referencia a cada item a sua
    for j in range(n):                      #porcentagem escolhida
        if M[j] != 0:
            itens_choosed[idx_w_ratio[j][0]] = M[j]

    return itens_choosed                    #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:

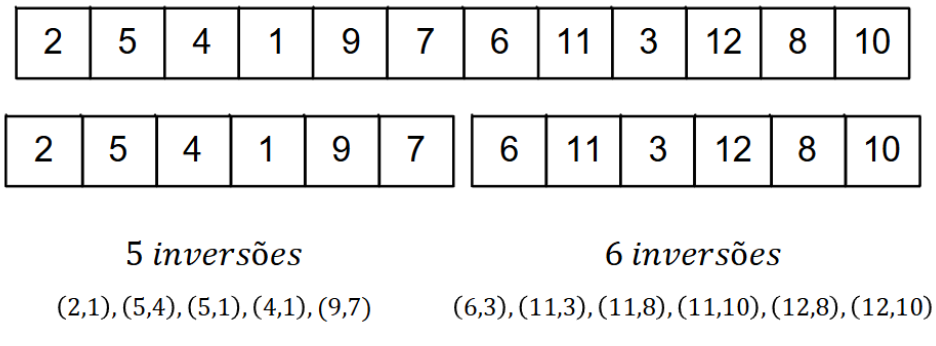


Figura 7: Exemplo do problema da contagem de inversões

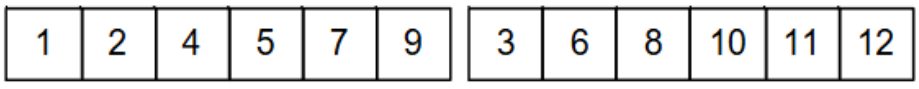
Esse resultado pode ser obtido executando o algoritmo recursivamente ($\sim T(\frac{n}{2})$). Claro que, por fim, teremos que contar as inversões da junção das duas listas:

$$+ 7 \text{ inversões ao combinar}$$

$$(5,3), (4,3), (9,6), (9,3), (9,8), (7,6), (7,3)$$

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:



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

para a_j **de** R:
  incremente i **até** $L[i] > a_j$
  $\text{inv}_{\text{aj}} = |L| - i$.

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

Para explicar, considere o exemplo da imagem anterior e relembre 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 ínidice 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:

CountInversions (A) :
  se $\text{len(A)} = 1$:
    retorne 0
  divida a lista em L e R
  $i_l = \text{CountInversions}(L)$
  $i_r = \text{CountInversions}(R)$
  $L = \text{Sort}(L)$
  $R = \text{Sort(R)}$
  $i = \text{Combine (L, R)}$
  retorne $i_l + i_r + i$

Avaliando a complexidade desse algoritmo, temos:

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

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 eliminassemos a etapa de ordenação explícita. Podemos fazer isso se, no algoritmo de contagem de inversão, além de contar, invertessemos e realizassemos o merge, trazendo o array ordenado direto.

Novo algoritmo:

CountInversions (A) :
  se $\text{len(A)} = 1$:
    retorne $0, A$
  divida a lista em L e R
  $i_l, L = \text{CountInversions}(L)$
  $i_r, R = \text{CountInversions}(R)$
  $i = \text{Combine (L, R)}$
  $A = \text{Merge}(L, R)$
  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:

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

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.

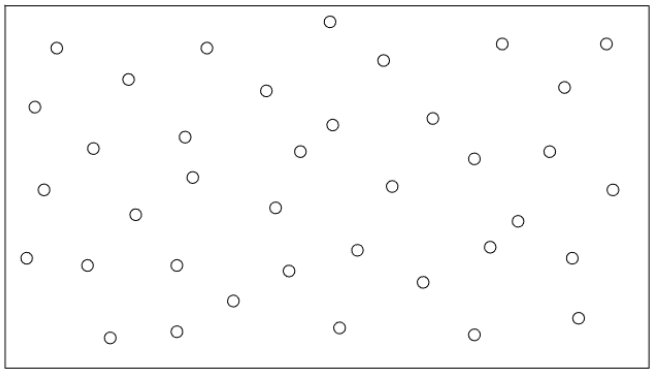
```py
def combine_merge(v, start_a, start_b, end_b):
    r = [0] * (end_b - start_a)
    a_idx = start_a
    b_idx = start_b
    r_idx = 0
    num_inv = 0                         #adicionado

    while a_idx < start_b and b_idx < end_b:
        if v[a_idx] <= v[b_idx]:
            r[r_idx] = v[a_idx]
            a_idx += 1
        else:
            num_inv += start_b - a_idx  #adicionado (explicado anteriormente)
            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.

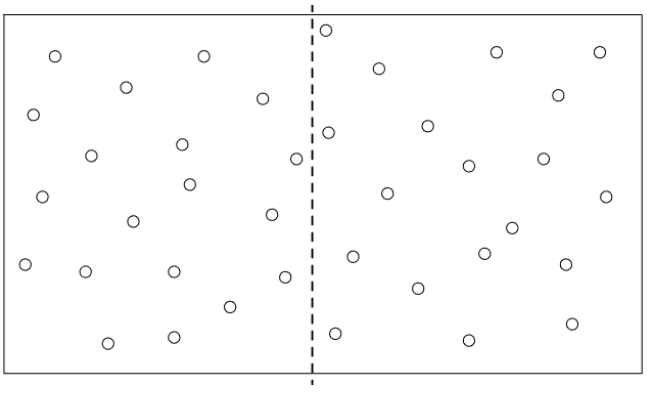
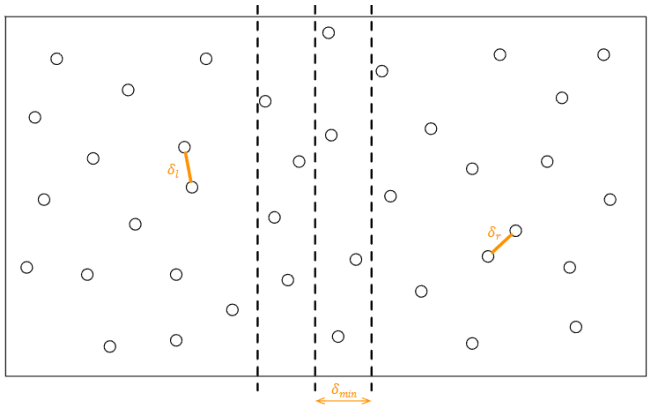


Figura 11: Exemplificação da solução do problema de pares mais próximos

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 $\delta_{\min}$ está restrito a, no máximo, essa distância).

Ideia: procurar somente os pontos que estejam no máximo à $\delta_{\min}$ 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 $\delta_{\min}$.

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

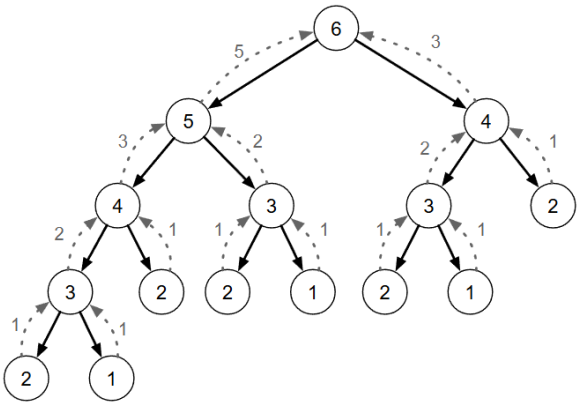


Figura 12: Exemplo de como seria $\mathrm{fib}(6)$

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

```py
def fib(n):
  if n <= 2:
    return 1
  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):

```py
def Fib(n):
  if n == 0:
      return 0
  if n == 1:
      return 1

  F = [-1] * (n + 1)
  F[0], F[1], F[2] = 0,1,1

  def FibAux(k):
    if F[k] == -1:
      F[k] = FibAux(k - 1) + FibAux(k-2)
    return F[k]

  return FibAux(n)
```

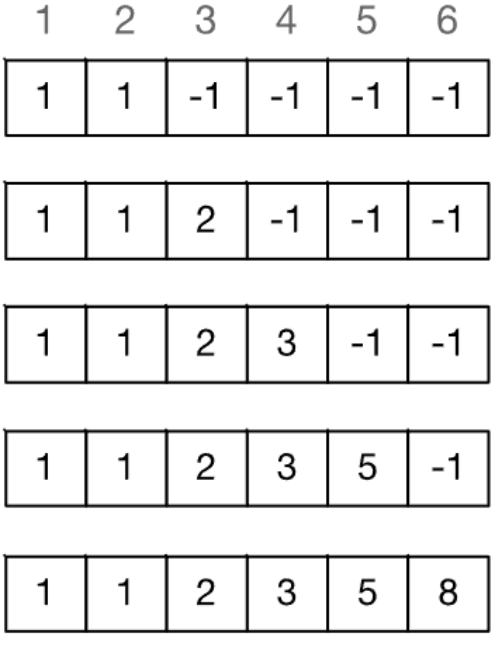


Figura 13: Exemplo de como fica o cachê para $\mathrm{Fib}(6)$

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

Solução Bottom-Up (interativa):

```py
def Fib(n):
  if n <= 2:
      return 1
  F = [0] * n
  F[1], F[2] = 1, 1
  for i in range (3,n + 1,1):
      F[i] = F[i - 1] + F[i - 2]
  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 $\mathbb{I} = \{1, 2, 3, ..., n\}$ 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 $\sum_{i \in S}^{|S|} w_i \leq W$ e $\sum_{i \in S}^{|S|} v_i$ 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 adiconado, 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 ∈ I
5 |  se w_i > W :
6 |  |  retorne Mochila (I − i, v, w, W) :
7 |  value_using = v_i + Mochila (I − i, v, w, W − w_i)
8 |  value_not_using = Mochila (I − i, v, w, W)
```

```
    retorna max{value_using, 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 IxW, 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:

```
Mochila (n, v, w, W) :
    crie uma matriz nxW
    para i = 0 até W:
        M[0][i] = 0
        para j = 1 até n:
            M[j][0] = 0
            M[j][i] = −1
    retorna MochilhaAux(n, v, w, W)
```

onde n é o total de itens.

$W = 11$

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	...

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

Continuação da solução:

```
MochilaAux (i, v, w, W) :
    se M[i][W] = −1:
        se w_i > W:
            M[i][W] = MochilaAux (i − 1, v, w, W)
        caso contrário:
            using = v_i + MochilaAux (i − 1, v, w, W − w_i)
            not_using = MochilaAux (i − 1, v, w, W)
            M[i][W] = max{using,not_using}
    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:

```
Mochila (n, v, w, W) :
    crie uma matriz nxW
    para i = 0 até W:
        M[0][i] = 0
    para j = 1 até n:
        M[j][0] = 0
    para j = 1 até n:
        para i = 1 até W:
            se w_j > i :
```

```
            M[j][i] = M[j − 1][i]
        caso contrário:
            using = v_j + M[j − 1][i − w_j]
            not_using = M[j − 1][i]
            M[j][i] = max{using,not_using}
    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:

$W + 1$

$W = 11$

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

$n + 1$

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 nxw 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-lá 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

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

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

Vamos entender o código: j itera nas linhas, W nas colunas. Em teoria, a última célula da matriz (nxW) 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 $M[j][i] = M[j-1]\left[i - w_j\right] + v_j$, 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 $(M[4][4] = 7 + v_j = 7 + 28 \neq 40)$.
  - 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 $(M[3][5] = 18 + v_j = 18 + 22 = 40)$.
  - 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

```py
def bag_problem_bottom_up(n, v, w, W):
    #primeira parte (criar a matriz e inserir os valores)
    M = [[0] * (W + 1) for _ in range(n + 1)]
    for j in range(1, n + 1):
        for i in range(1, W + 1):
            peso_item_j = w[j-1]
            valor_item_j = v[j-1]
            if peso_item_j > i:
                M[j][i] = M[j - 1][i]
```

```
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 $|.|$ 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:

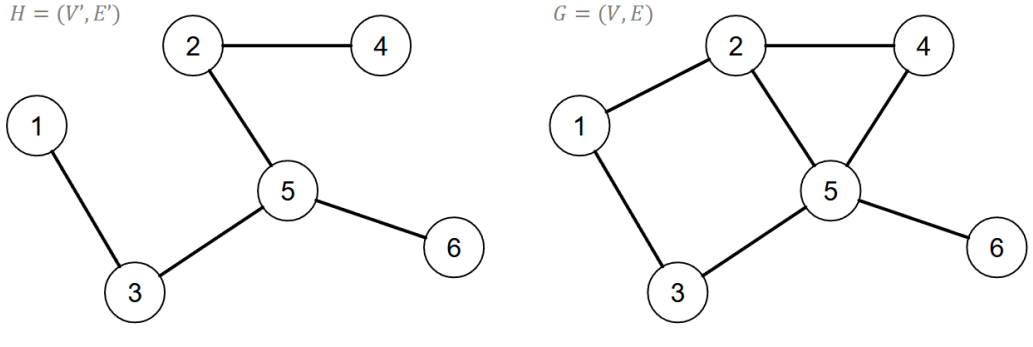


Figura 18: Exemplo de subgrafo gerador.

- 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:

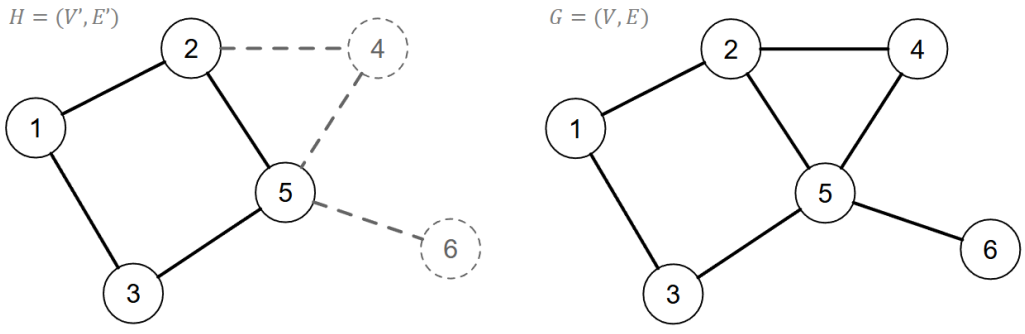


Figura 19: Exemplo de grafo induzido (os vértices escolhidos foram $\{1, 2, 3, 5\}$ e as arestas(e vértices) que não são desses vértices não aparecem no subgrafo induzido).

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

  Exemplo:

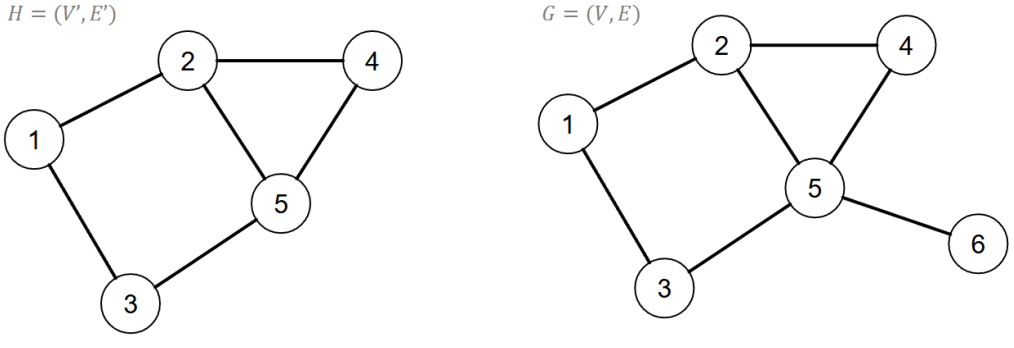


Figura 20: Exemplo de grafo próprio (note que é $\subset$, não $\subseteq$. Então, um subgrafo próprio é um subgrafo menor, e não igual ao grafo original).

- 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 possuirem 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 **sáida** $g_s(v_i)$ é definido pelo número de arestas que saem de v_i. Raciocíno 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 orienteados:

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

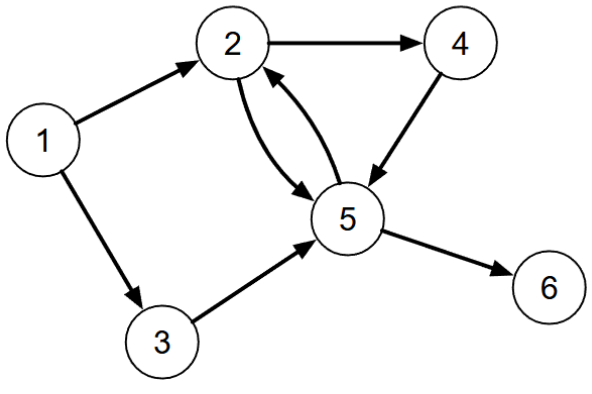


Figura 21: Exemplo de matriz de adjacência para o grafo à direita.

Analogamente, para não orientados:

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

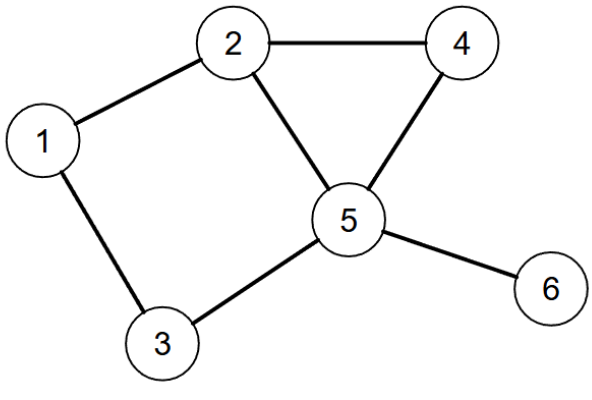


Figura 22: Exemplo de matriz de adjacência para o grafo à direita. Nota: a matriz é simétrica!

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:

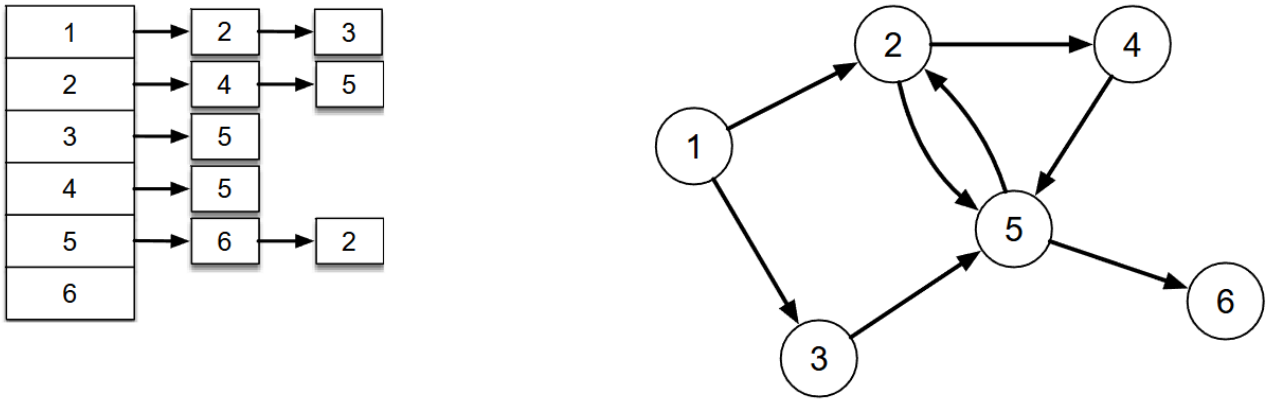


Figura 23: Exemplo da lista de adjacência para o grafo à direita.

Exemplo para grafo não-dirigido:

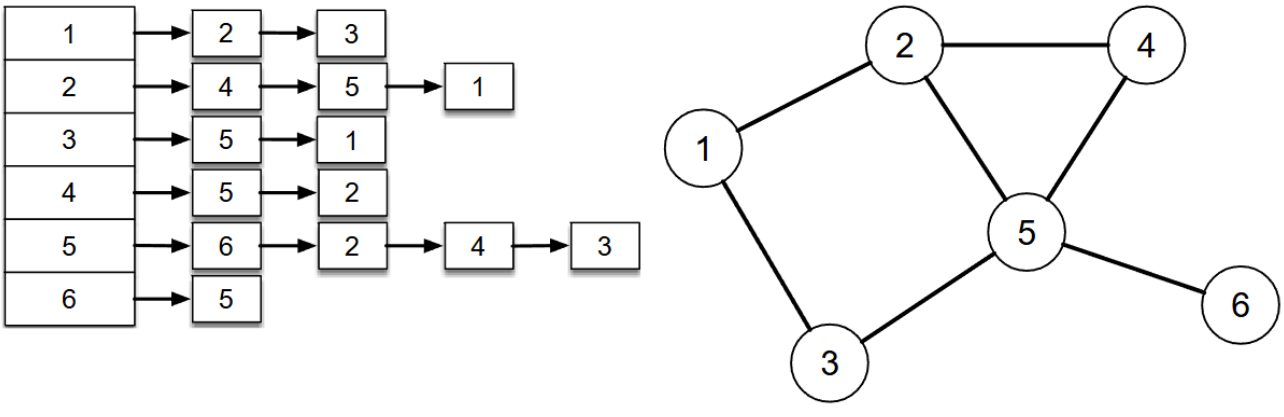


Figura 24: Exemplo da lista de adjacência para o grafo à direita.

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?

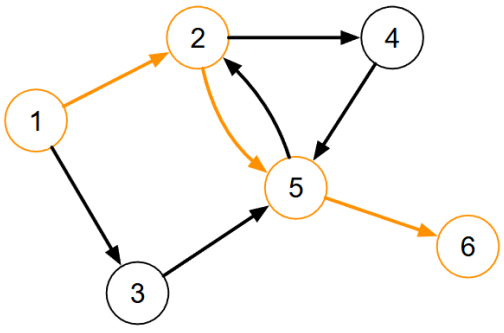


Figura 25: Exemplo do caminho $P = \{1, 2, 5, 6\}$

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:

```py
def reach_recursive_matrix(v_atual, visited, matrix, num_vertices):
    visited[v_atual] = True
    for v_vizinho in range(num_vertices):
        if matrix[v_atual][v_vizinho] == 1 and not visited[v_vizinho]:
            reach_recursive_matrix(v_vizinho, visited, matrix, num_vertices)

def can_reach_matrix(matrix, v1, v2):
    num_vertices = len(matrix)
    visited = [0] * num_vertices
    reach_recursive_matrix(v1, visited, matrix, num_vertices)
    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
```

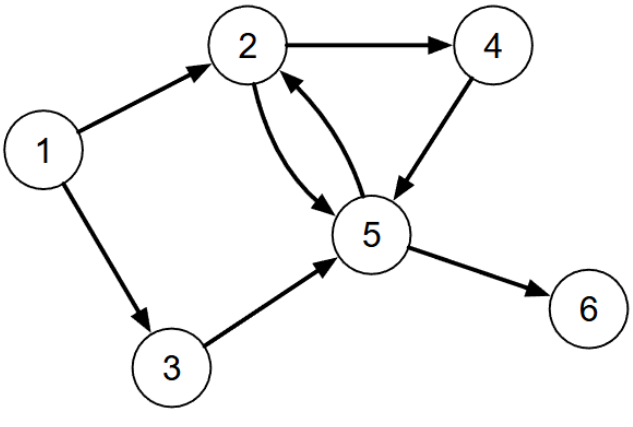


Figura 26: Exemplo da execução do algoritmo can_reach para o mesmo grafo

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

```cpp
void dfs(int * preOrder) {
    int counter = 0;
    for (vertex v=0; v < m_numVertices; v++) {
        preOrder[v] = -1;
    }
    for (vertex v=0; v < m_numVertices; v++) {
        if (preOrder[v] == -1) {
            dfsRecursive(v, preOrder, counter);
        }
    }
}

void dfsRecursive(vertex v1, int * preOrder, int & counter) {
    preOrder[v1] = counter++;
    EdgeNode * edge = m_edges[v1];
    while (edge) {
        vertex v2 = edge->otherVertex();
        if (preOrder[v2] == -1) {
            dfsRecursive(v2, preOrder, counter);
        }
        edge = edge->next();
    }
}
```

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.

```py
def dfs_preorder(adj_list):
    num_vertices = len(adj_list)
    preorder = [-1] * num_vertices
    counter = 0
    for v in range(num_vertices):
        if preorder[v] == -1:
            counter = dfs_recursive(v, preorder, counter, adj_list)
    return preorder

```

```python
def dfs_recursive(v_atual, preorder, counter, adj_list): #Auxiliar
    preorder[v_atual] = counter
    counter += 1
    for v_vizinho in adj_list[v_atual]:
        if preorder[v_vizinho] == -1:
            counter = dfs_recursive(v_vizinho, preorder, counter, adj_list)
    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
```

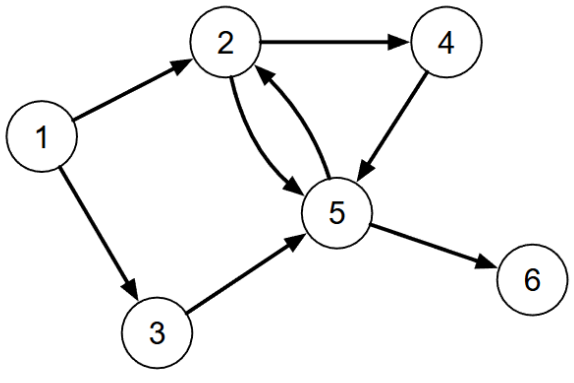


Figura 27: Exemplo da execução do algoritmo DFS para o mesmo grafo

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 $i < j$.

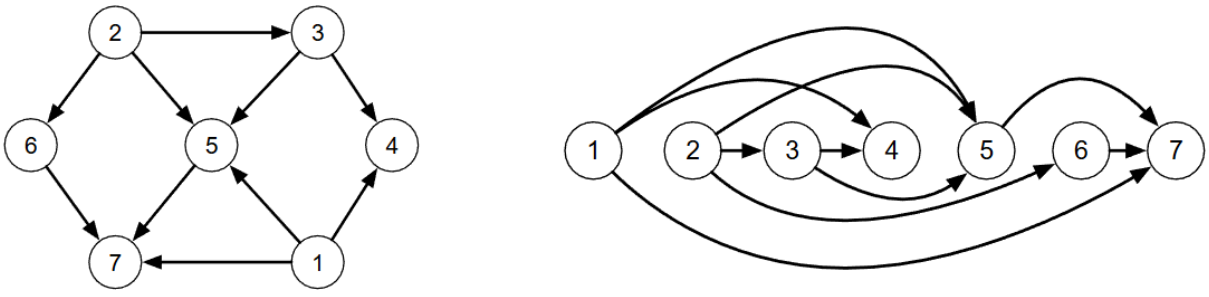


Figura 28: Exemplo da grafo topológico (Note que se os vértices forem dispostos em ordem crescente toda aresta irá apontar para o sentido de crescimento dos números).

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).

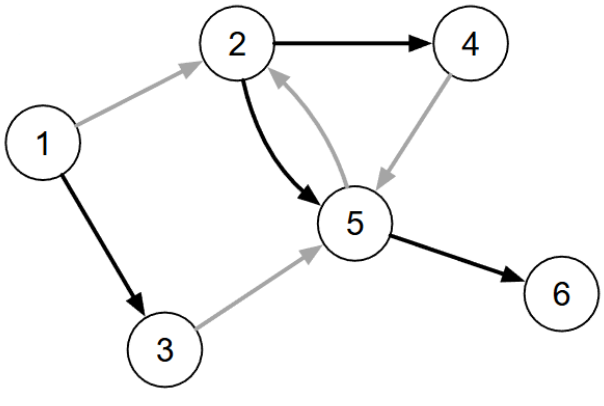


Figura 29: Exemplo de floresta radicada (a raiz no 2 foi proposital)

3.3 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;
```

```
    for (vertex v=0; v < m_numVertices; v++) {
        preOrder[v] = -1;
        parents[v] = -1;
    }

    for (vertex v=0; v < m_numVertices; v++) {
        if (preOrder[v] == -1) {
            parents[v] = v;
            dfsRecursive(v, preOrder, counter, parents, 0);
        }
    }
}


void dfsRecursive(vertex v1, int * preOrder, int & counter, int * parents, int level=0) {
    preOrder[v1] = counter++;
    EdgeNode * edge = m_edges[v1];
    while (edge) {
        vertex v2 = edge->otherVertex();
        if (preOrder[v2] == -1) {
            parents[v2] = v1; // Set parent first
            dfsRecursive(v2, preOrder, counter, parents, level + 1);
        }
        edge = edge->next();
    }
}
```

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”).

```py
def dfs_recursive_parents(v_atual, preorder, parents, counter, adj_list):
    preorder[v_atual] = counter
    counter += 1
    for v_vizinho in adj_list[v_atual]:
        if preorder[v_vizinho] == -1:
            parents[v_vizinho] = v_atual
            counter = dfs_recursive_parents(v_vizinho, preorder, parents, counter, adj_list)

    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
```

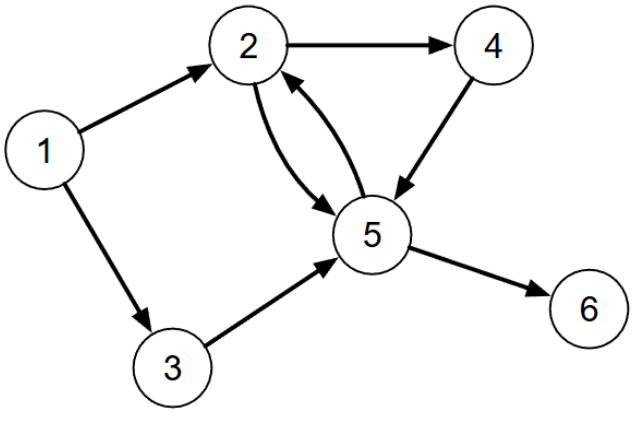


Figura 30: Exemplo do algoritmo dfs_parents para o grafo de exemplo.

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:

C++

```
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++) {
```

```
        if (preOrder[v] == -1) {
            parents[v] = v;
            dfsRecursive(
                v, preOrder, preCounter,
                postOrder, postCounter, parents);
        }
    }
}

void dfsRecursive(vertex v1, int * preOrder, int & preCounter, int * postOrder,
                  int & postCounter, int * parents) {
    preOrder[v1] = preCounter++;
    EdgeNode * edge = m_edges[v1];
    while (edge) {
        vertex v2 = edge->otherVertex();
        if (preOrder[v2] == -1) {
            parents[v2] = v1;
            dfsRecursive(v2, preOrder, preCounter,
                         postOrder, postCounter, parents);
        }
        edge = edge->next();
    }
    postOrder[v1] = postCounter++;
}
```

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

```py
def dfs_recursive_full(v_atual, preorder, postorder, parents, pre_counter, post_counter, adj_list):
    preorder[v_atual] = pre_counter
    pre_counter += 1
    for v_vizinho in adj_list[v_atual]:
        if preorder[v_vizinho] == -1:
            parents[v_vizinho] = v_atual
            pre_counter, post_counter = dfs_recursive_full(v_vizinho, preorder, postorder, parents, pre_counter, post_counter, adj_list)
    postorder[v_atual] = post_counter
    post_counter += 1
    return pre_counter, post_counter

def dfs_full(adj_list):
    num_vertices = len(adj_list)
    preorder = [-1] * num_vertices
    postorder = [-1] * num_vertices
    parents = [-1] * num_vertices
    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,
                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
```

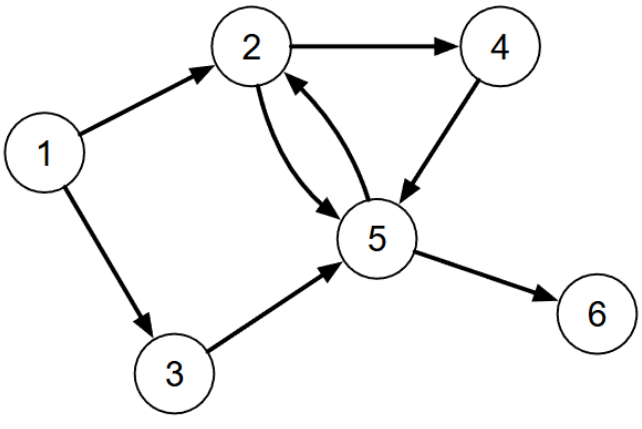


Figura 31: Exemplo do algoritmo dfs_parents_full para o grafo de exemplo.

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 (preOrder[v], postOrder), 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)
```

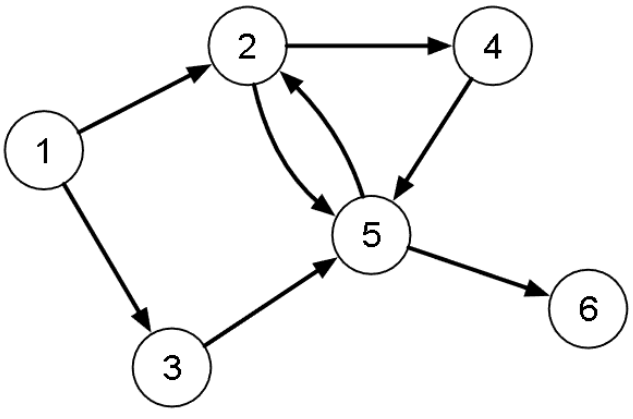


Figura 32: Exemplo do mapeamento do lifespan no grafo de exemplo.

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 pro 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:
  - preOrder[v1] < preOrder[v2]
- v_1 é primo mais novo de v_2 se:
  - preOrder[v1] > preOrder[v2]

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_j está contido no intervalo de v_i, ou seja:
  - `preOrder[vi] < preOrder[vj] AND postOrder[vi] > postOrder[vj]`
- É uma aresta de retorno:
  - se o intervalo de v_j contém o intervalo de v_i, ou seja:
  - `preOrder[vi] > preOrder[vj] AND postOrder[vi] < postOrder[vj]`

- É uma aresta cruzada:
  - se o intervalo de v_j ocorre antes do intervalo de v_i, ou seja:
  - `preOrder[vi] > preOrder[vj] AND postOrder[vi] > postOrder[vj]`

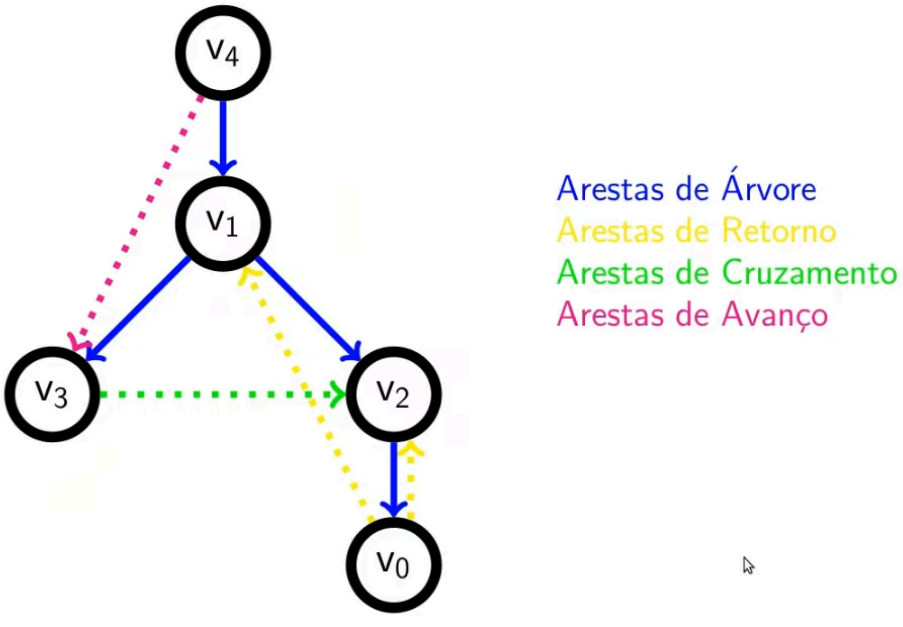


Figura 33: Exemplo de arestas de avanço, retorno e cruzada.

Nota: essas propriedades para as arestas que não são da árvore são apenas quando usamos o `preOrder` e o `postOrder` com a contagem junta, ou seja, dependentes, da forma:

```py
# Índices:     0  1  2  3  4
pre_order  = [ 4, 2, 3, 7, 1]
post_order = [ 5, 9, 6, 8, 10]
```

Nesse caso, as definições valem do jeito que foram passadas.

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 radicada é um DAG sem vértices com grau de entrada maior que 1.
- Uma árvore radicada 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++:

```C++
bool hasCycle(int * preOrder, int * postOrder) {
dfs(preOrder, postOrder);
for (vertex v1=0; v1 < m_numVertices; v1++) {
    EdgeNode * edge = m_edges[v1];
    while(edge) {
        vertex v2 = edge->otherVertex();
        if (preOrder[v1] > preOrder[v2]
            && postOrder[v1] < postOrder[v2]) {
            return true;
```

```
10          }
11          edge = edge->next();
12      }
13  }
14  return false;
15  }
```

Implementação em Python:

Indo para Python, vamos considerar que passamos as listas de preorder e postorder:

```py
def has_cycle(adj_list, preorder, postorder):
    num_vertices = len(adj_list)
    for v1 in range(num_vertices):
        for v2 in adj_list[v1]:
            if preorder[v1] > preorder[v2] and postorder[v1] < postorder[v2]:
                return True
    return False
```

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:

Percorrer o grafo por camadas, ou seja:

- Inicia visitando um grafo v_0;
- Visita seus vértices adjacentes;
- Visita os adjacentes dos adjacentes (que ainda não foram visitados);
- Continua até todos os vértices terem sido visitados.

Assim como o DFS, define a ordem de descoberta dos vértices.

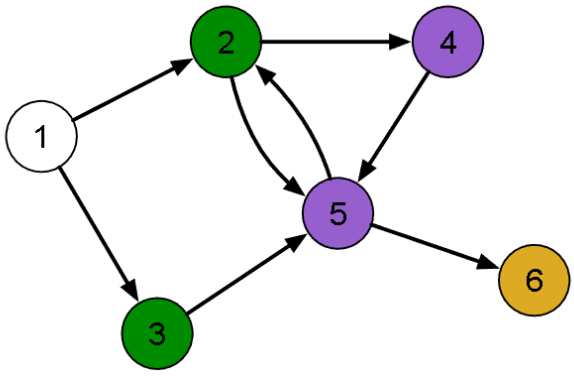


Figura 34: Exemplo do algoritmo BFS (note que cada nível está de uma cor).

Uma implementação comum desse algoritmo utiliza uma fila para armazenar os vértices descobertos que ainda não foram explorados. Vamos ver o código:

```C++
void bfs(vertex v0, int * order) {
    queue<int> queue;
    int counter = 0;
    for (int i=0; i < m_numVertices; i++) {
        order[i] = -1;
    }
    order[v0] = counter++;
    queue.push(v0);
    while (!queue.empty()) {
        int v1 = queue.front();
        queue.pop();
```

```
12          EdgeNode * edge = m_edges[v1];
13          while (edge) {
14              vertex v2 = edge->otherVertex();
15              if (order[v2] == -1) {
16                  order[v2] = counter++;
17                  queue.push(v2);
18              }
19              edge = edge->next();
20          }
21      }
22  }
```

Essa função recebe um vértice inicial `v[0]` e um ponteiro para a lista de ordem que será dada a ele. Inicia-se também uma fila (estrutura de dados que vimos em ED) e um counter que vai determinar a posição de cada vértice (a ordem). Após preencher a lista de ordem como –1, ele marca a posição do elemento `v[0]` na lista de ordem e faz o push de v_0 na fila.

Continuando, enquanto a fila não for vazia, chamamos de v_1 o primeiro item da fila, o retiramos da fila e pegamos sua lista de adjacência. Enquanto tiverem vértices nessa lista, pegamos o vértice do outro lado da aresta (v_2) e verificamos se ele não está na lista de ordem (já visitado). Caso já não tenha sido visitado, ele é adicionado na fila, e passamos para o próximo vértice.

O que podemos ver aqui é que a utilização da fila como estrutura de dados para esse algoritmo faz total diferença, já que isso faz com que, começando do vértice v_0, passamos por todos os seus filhos, e o uso da fila faz com que apenas os próximos i a serem visitados sejam exatamente os i filhos de v_0, e assim sucessivamente, trazendo uma busca em nível. Observe que essa implementação número apenas os vértices a partir de v_0 (funciona bem quando você sabe que é um grafo com apenas uma componente conexa e com v_0 como raiz).

Como faríamos para garantir um algoritmo que numera todos os vértices?

```
1   void bfsForest(int * order) {                                    C++
2       int counter = 0;
3       for (int i=0; i < m_numVertices; i++) { order[i] = -1; }
4       for (int i=0; i < m_numVertices; i++) {
5           if (order[i] != -1) {
6               continue;
7               }
8           order[i] = counter++;
9           queue<int> queue;
10          queue.push(i);
11          while (!queue.empty()) {
12              int v1 = queue.front();
13              queue.pop();
14              EdgeNode * edge = m_edges[v1];
15              while(edge) {
16                  vertex v2 = edge->otherVertex();
17                  if (order[v2] == -1) {
18                      order[v2] = counter++;
```

```
                    queue.push(v2);
                }
                edge = edge->next();
            }
        }
    }
}
```

O que muda desse algoritmo para o anterior é simplesmente a inicialização, pois agora nos baseamos no número de vértices para preencher a ordem como -1 e além disso, fazemos um for para passar por todos os vértices. Mas a ideia é a mesma, pois dentro desse for continuamos se ele já foi visitado, e se não foi, marcamos sua posição, e fazemos a mesma verificação para a lista de adjacências dele.

Legal, temos um array (`order`) que mostra a ordem de visitação, mas isso não me mostra exatamente como chegar de um vértice a outro diretamente. E se marcassemos o pai de cada vértice?

C++

```
void bfs(vertex v0, int * order, int * parent) {
    queue<int> queue;
    int counter = 0;
    for (int i=0; i < m_numVertices; i++) {
        order[i] = -1;
        parent[i] = -1;
    }
    order[v0] = counter++;
    parent[v0] = v0;
    queue.push(v0);
    while (!queue.empty()) {
        int v1 = queue.front();
        queue.pop();
        EdgeNode * edge = m_edges[v1];
        while (edge) {
            vertex v2 = edge->otherVertex();
            if (order[v2] == -1) {
                order[v2] = counter++;
                parent[v2] = v1;
                queue.push(v2);
            }
            edge = edge->next();
        }
    }
}
```

Note que precisamos voltar com o v_0, já que marcar o vértice no caminho mais curto vindo da origem sem uma origem não faz muito sentido.

Ele é exatamente igual o algoritmo anterior, a menos do vetor `parents`, que inicialment é declarado como -1 para todo vértice e, quando entra no if do não visitado, é marcado que v_1 é seu pai. Simples assim!

Analisando a complexidade (do último algoritmo), passamos por um for no número de vértices ($O(V)$), depois fazemos um while na queue. Como a queue terá no máximo tamanho $|V|$, pois o if verifica se já foi adicionado, e no while de dentro passamos por cada aresta de v_i (que sabemos que $\sum_{i=1}^{|V|} g_s(v_i) = |E|$), temos uma complexidade de no máximo $\Theta(V + E)$ ao utilizar lista de adjacências.

Ao utilizar matriz de adjacências, teríamos que buscar cada ligação de cada vértice sem receber uma lista pronta com isso, o que traria uma complexidade de $\Theta(V^2)$. Ainda, para grafos densos, ambas as estruturas de dados traria uma complexidade de $\Theta(V^2)$.

Implementação em Python

Vamos implementar os dois últimos algoritmos, pois são os mais completos. Forest:

```py
from collections import deque

def bfs_forest (list_adj):
    num_vertices = len(list_adj)
    counter = 0
    order = [-1] * num_vertices
    for i in range(num_vertices):
        if order[i] != -1:
            continue
        fila = deque()
        order[i] = counter
        counter += 1
        fila.append(i)
        while fila:
            v1 = fila.popleft()
            for vizinho in list_adj[v1]:
                if order[vizinho] == -1:
                    order[vizinho] = counter
                    counter += 1
                    fila.append(vizinho)

    return order
```

Note que ambos precisam de usar deque(fila com ponteiros para início e fim) para funcionarem com as mesmas complexidades. BFS:

```py
from collections import deque

def bfs (v0, list_adj):
    num_vertices = len(list_adj)
    fila = deque()
    counter = 0
    order = [-1] * num_vertices
    parent = [-1] * num_vertices

```

```
    order[v0] = counter
    counter += 1
    parent[v0] = v0
    fila.append(v0)
    while fila:
        v1 = fila.popleft()
        for vizinho in list_adj[v1]:
            if order[vizinho] == -1:
                order[vizinho] = counter
                counter += 1
                parent[vizinho] = v1
                fila.append(vizinho)
    return parent, order
```

Fim! Mas agora, como achar o menor caminho em um grafo??

Menor caminho em Grafos

Na seção anterior, tentamos verificar que o caminho existe. Agora, temos um novo problema:

Dados dois vértices v_i e v_j em um grafo $G = (V, E)$, encontre o caminho mínimo P que começa em v_i e termina em v_j.

Se o grafo for tiver mais de uma componente conexa, pode não existir um caminho entre v_i e v_j (podemos dizer que a distância é infinita). Além disso, se o grafo for orientado, a distância entre v_i e v_j pode ser diferente de v_j para v_i.

Para encontrar o caminho mais curto entre v_i e v_j é inevitável encontrar todos os caminhos que iniciam em v_i. A árvore produzida pela busca do menor caminho é chamada de Árvore de caminhos mais curtos (SPT - Shortest Path Tree).

A árvore SPT é uma sub-árvore radicada de G:
- Todos os vértices de G estão presentes na SPT.
- Todo caminho na SPT a partir da raiz é mínimo no grafo G.

Um grafo possui uma SPT caso todos os vértices sejam acessíveis a partir de v_i. Caso não exista uma SPT para um grafo G a partir de um grafo v_i, existe uma SPT para um sub-grafo induzido de G com os vértices acessíveis a partir de v_i.

Portanto, para encontrar o menor caminho entre v_i e v_j precisamos encontrar a árvore radicada desse sub-grafo com raiz em v_i.

4.1 Caminho mais curto em um DAG

Problema: Como criar um algoritmo capaz de gerar a SPT de um DAG iniciando na sua única fonte? (Considere que você possui uma possível ordem topológica para o DAG)

Dica: use as propriedades do DAG!

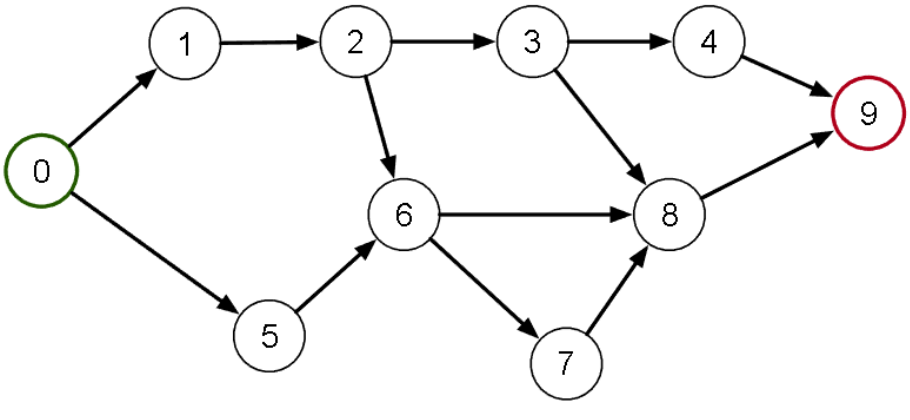


Figura 35: Exemplo de DAG com uma fonte (verde) e um sorvedouro (vermelho)

Solução:

- Inicialize cada vértice com a distância infinita para a raiz ($d[v_i] = \infty$) e pai indefinido ($\text{parent}[v_i] = -1$)
- Defina a raiz com distância zero ($d[v_1] = 0$)
- Percorra os vértices seguindo a ordem topológica
  - Avalie para cada vértice adjacente se $d[v_i] + 1 \leq d[v_j]$
    - se for, atualiza $d[v_j]$ no vértice adjacente com a menor distância e define o novo pai do vértice adjacente.

Como seria a execução desse algoritmo para o grafo de exemplo?

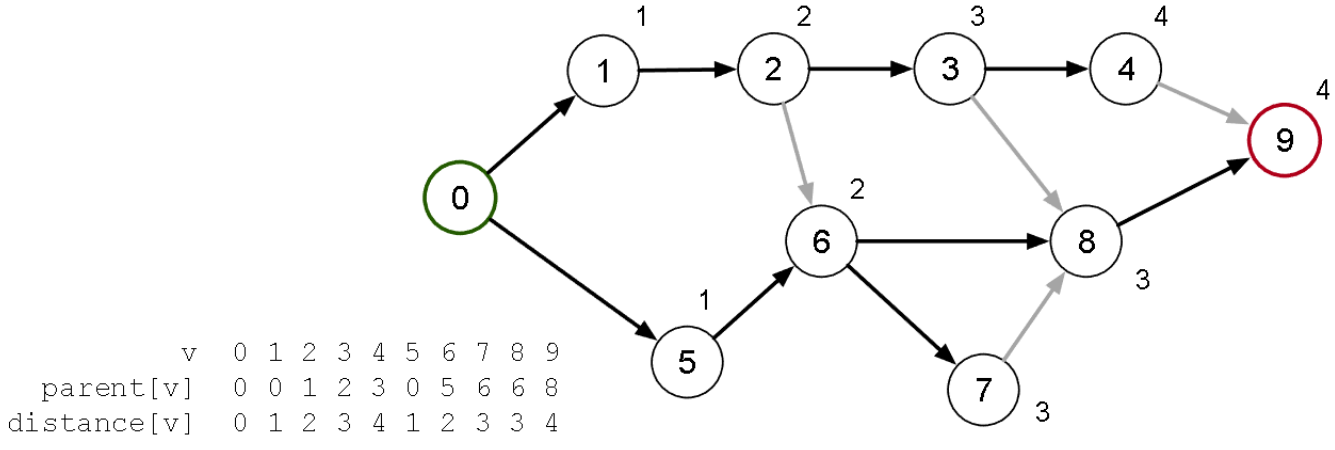


Figura 36: Exemplo do algoritmo para o grafo dado anteriormente.

Esse algoritmo funciona pois o vetor `parent` define uma árvore radicada T com raiz em v_0 e, para toda aresta $e = (v_i, v_j)$, se v_i foi processado então e já foi avaliada. Por fim, ao término de execução T é uma árvore radicada de um grafo induzido H, induzido de G, contendo os vértices acessíveis a partir de v_0, toda aresta de H foi avaliada e T é uma árvore geradora de H.

Nota: a implementação disso está nos Exercises

4.2 Caminho mais curto em grafos não-dirigidos/ciclo

A comparação $d[v_i] + 1 \leq d[v_j]$ e a eventual atualização no vetor de distância é conhecida como **operação de relaxamento.** Uma aresta está **relaxada** se $d[v_j] - d[v_i] \leq 1$ e **tensa** se $d[v_j] - d[v_i] > 1$.

> *Exemplo*: estou no vértice v_i, e quero ir para o meu vizinho v_j, sabendo que $d[v]$ é a distância mínima conhecida até agora da origem até aquele vértice. Se $d[v_j] - d[v_i] \leq 1$ (considerando que os pesos são inteiros e unitários), significa que a aresta (e_i, e_j), mesmo com peso 1, não conseguiria fazer com que o valor de $d[v_j]$ mude, pois ele já é o menor possível, e então essa aresta é relaxada. Pelo contrário, se $d[v_j] - d[v_i] > 1$, significa que se o peso da aresta entre os dois vértices é 1, então $d[v_j]$ consegue ser atualizado, por isso é uma aresta tensa.

Chamamos de potencial relaxado uma numeração para os vértices que torne todas as aresta do grafo relaxadas. O vetor de distâncias resultante do algoritmo anterior é um potencial relaxado.

Voltando ao problema inicial: desejamos encontrar o caminho mais curto entre dois vértices em qualquer grafo. Como produzir uma solução para grafos não-dirigidos e/ou que possuem ciclos?

Podemos adaptar o algoritmo de busca em largura (BFS) de forma que a numeração dos vértices represente a distância para a raiz.

Ideia geral: modificar o ciclo do BFS para remover o vértice v_i da fila, visitar seus vértices adjacentes e:

- se $d[v_j]$ não estiver definida:
  - $d[v_j] = d[v_i] + 1$
  - $\text{parent}[v_j] = v_i$
  - inserir v_j na fila

Note que o valor $d[v]$ é alterado somente uma vez.

Nota: a implementação disso está nos Exercises

4.3 Caminho mais barato em grafos

Um grafo (orientado ou não) é ponderado se cada aresta estiver associada à um valor (pode ser custo, peso, capacidade, etc).

Problema: dado dois vértices v_i e v_j em um grafo, encontre o caminho p com o custo mínimo que comea-ça em v_i e termina em v_j. (O custo é a soma das arestas e o custo mínimo é o menor valor possível de custo de um caminho de v_i a v_j).

A distância entre v_i e v_j é definida pelo comprimento do caminho mais barato. Essa distância pode ser negativa se existirem arstas negativas, se não existir caminho entre dois vértices podemos dizer que a distância é infinita. Ainda, a distância entre os mesmos dois vértices podem ser diferentes caso o grafo seja orientado.

Um **ciclo negativo** é um ciclo cujo custo restante da soma de suas arestas é negativo. Se um grafo possuir ciclos negativos o caminho mais barato entre dois vértices pode não ser simples.

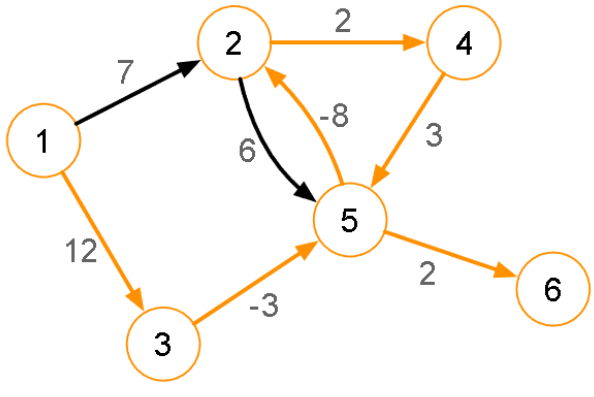


Figura 37: Exemplo de grafo com ciclo negativo.

O problema de busca pelo caminho mais barato se torna muito mais simples quando não existem ciclos negativos, já que se v_0 é um vértice que não possui negativos ao seu alcance, todo caminho p é simples e todo trecho inicial de p entre v_0 e v_k é o caminho mais barato de v_0 e v_k.

A árvore encontrada na busca pelo caminho mais barato é chamada de Árvore de caminhos mais barato - CPT (Cheapest Path Tree)

Essa árvore é sub-radicada em G:

- Todos os vértices de G estão presentes na CPT;
- Todo caminho na CPT a partir da raiz é mínimo no grafo G.

4.3.1 Djikstra

Esse famoso algoritmo, que provavelmente você, caro leitor, já viu em MD, é capaz de encontrar caminhos mais baratos em um grafo $G = (V, E)$ que possua arestas com custos positivos.

A solução consiste em crescer uma árvore radicada a partir do vértice inicial v_0 até que ela seja uma árvore geradora do subgrafo induzido a partir de v_0.

Antes de explicar melhor, a **franja** de uma árvore radicada T com raiz em v_0 é o conjunto das arestas (v_i, v_j) que possuem v_i em T e v_j fora de T. A franja pode ser vista como o grau de saída do conjunto de vértices de T.

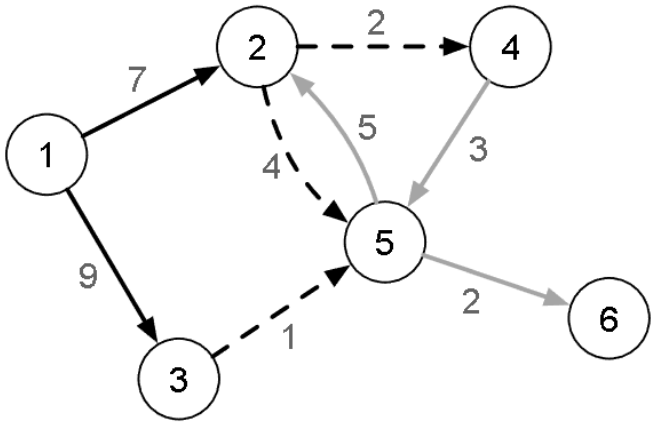


Figura 38: Exemplo de franja. Os vértices $(1, 2, 3)$ já estão árvore radicada, e a franja é o conjuntos de arestas pontilhadas de peso $(2, 4, 1)$.

Depois de entender esse conceito, essa é a ideia geral para o Djikstra:

1 **insira v_0 em T**
2 **defina $d[v_0] = 0$**
3 **enquanto a franja não estiver vazia:**
4 | **escolha v_k de menor distância**
5 | **aplique a operação de relaxamento em todas as arestas de v_k da franja**
6 | **insira v_k em T**

Veja a implementação em C++:

```cpp
void cptDijkstraSlow(vertex v0, vertex *parent, int *distance) {
  std::vector<bool> checked(m_numVertices);
  for (vertex v = 0; v < m_numVertices; v++) {
      parent[v] = -1;
      distance[v] = INT_MAX;
      checked[v] = false;
  }
  parent[v0] = v0;
  distance[v0] = 0;

  while (true) {
      int minDistance = INT_MAX;
      vertex v1 = -1;
      for (vertex i = 0; i < m_numVertices; i++) {
          if (!checked[i] && distance[i] < minDistance) {
              minDistance = distance[i];
              v1 = i;
          }
      }
      if (minDistance == INT_MAX || v1 == -1) break;
      checked[v1] = true;
      EdgeNode *edge = m_edges[v1];
```

```
23        while (edge) {
24            vertex v2 = edge->otherVertex();
25            if (!checked[v2]) {
26                int cost = edge->cost();
27                if (distance[v1] != INT_MAX && distance[v1] + cost < distance[v2]) {
28                    parent[v2] = v1;
29                    distance[v2] = distance[v1] + cost;
30                }
31            }
32            edge = edge->next();
33        }
34    }
35 }
```

O algoritmo inicializa os vetores do começo `parent`, `distance` e `checked`, e declara as informações iniciais de v_0. Depois, inicializa um while onde, a cada iteração declara a maior distância como um inteiro máximo, e, após isso, o vértice escolhido de -1(pois não escolhemos ainda). Nosso objetivo no for de baixo é escolher, dentre os que tem distância definida e ainda não foram checados (ou seja, a franja), quem tem a menor distância. Após selecionar esse vértice e verificarmos a condição de parada(se a menor distância não mudou, então não temos mais vértices para checar), marcamos ele como checado e fazemos a análise para cada vizinho.

Para cada vizinho, chamamos de v_2 o vizinho a ser analisado. Se ele não tiver sido checado, pegamos o seu custo e se a distância do $v_1 +$ custo da aresta for menor, então atualizamos a distância e o pai de v_2, e passamos para o próximo vértice. (a outra verificação desse if é pra evitar overflow).

Implementação em Python

Aqui, consideramos que a lista de adjacências é da forma:

```py
1 [
2 [(vértice, custo), (vértice, custo)], #arestas do vértice 0
3 [(vértice, custo), (vértice, custo)], #arestas do vértice 1
4 ]
```

Isso é apenas a transformação do código para Python, usando essa estrutura mais simples, que fica dessa forma:

```py
1  def cpt_djikstra_slow(v0, list_adj):
2      num_vertices = len(list_adj)
3      checked = [0] * num_vertices
4      parent = [-1] * num_vertices
5      distance = [float('inf')] * num_vertices
6      parent[v0] = v0
7      distance[v0] = 0
8
9      while True:
10         mindistance = float('inf')
11         v1 = -1
```

```
        for i in range(num_vertices):
            if checked[i] == False and distance[i] < mindistance:
                mindistance = distance[i]
                v1 = i
        if mindistance == float('inf') or v1 == -1:
            break
        checked[v1] = True
        for vizinho, custo in list_adj[v1]:
            if checked[vizinho] == False:
                if distance[v1] != float('inf') and distance[v1] + custo <
                distance[vizinho]:
                    parent[vizinho] = v1
                    distance[vizinho] = distance[v1] + custo

    return parent,  distance
```

Esse código é bem parecido com os que já vimos, ele faz várias declarações de variáveis $O(V)$, e o while True roda no máximo V vezes, já que depende de um for que roda também V vezes (pois a verificação em algum intervalo $[0, |V]]$ é interrompida, porque cai no caso de condição de parada). Esse for só faz verificações constantes, e por isso é $O(V)$. continuando, temos um if e outro for que passa por $g_s(v_k)$ arestas, que ao final somam E. Portanto, dentro do while temos $O(V(V + g_s(v_k))) = O(V^2 + E)$, já que $\sum_{i=1}^{|V|} g_s(v_i) = |E|$.

Existe uma característica importante nesse algoritmo: a cada iteração onde verificamos o elemento da franja a ser escolhido (passando por mais elementos que o necessário para escolher o menor, pois passamos por todos), uma iteração qualquer é sempre semelhante à iteração anterior. Com essa informação, como podemos melhorar o desempenho do algoritmo?

4.3.2 Djikstra "Rápido"

Ideia: manter os vértices da franja em uma fila de prioridades, implementada como um heap mínimo e que contém todos os vértices que ainda não foram verificados.

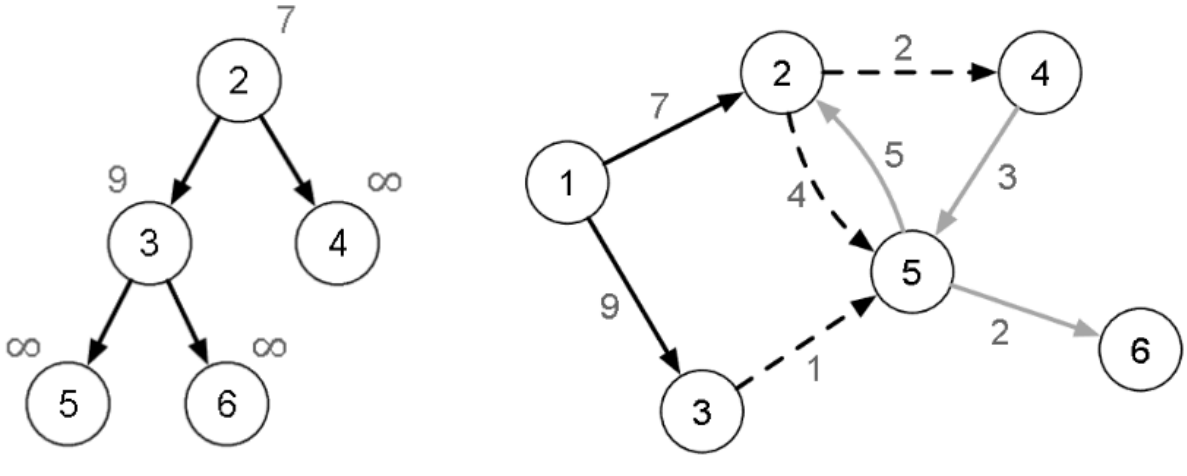


Figura 39: Exemplo do estado do algoritmo após processar o vértice 1. A imagem à esquerda mostra o heap nessa iteração.

Esse é o código em C++:

```
1   void cptDijkstraFast(vertex v0, vertex * parent, int * distance) {
```
C++

```
    bool checked[m_numVertices];
    Heap heap; // Create the heap
    for (vertex v=0; v < m_numVertices; v++) {
        parent[v] = -1;
        distance[v] = INT_MAX;
        checked[v] = false;
    }
    parent[v0] = v0;
    distance[v0] = 0;

    heap.insert_or_update(distance[v0], v0);
    while (!heap.empty()) {
        vertex v1 = heap.top().second; // Min vertex
        heap.pop(); // Remove from heap
        if (distance[v1] == INT_MAX) { break; }
        EdgeNode * edge = m_edges[v1];
        while (edge) {
            vertex v2 = edge->otherVertex();
            if (!checked[v2]) {
                int cost = edge->cost();
                if (distance[v1] + cost < distance[v2]) {
                    parent[v2] = v1;
                    distance[v2] = distance[v1] + cost;
                    heap.insert_or_update(distance[v2], v2);
                }
            }
            edge = edge->next();
        }
        checked[v1] = true;
    }
}
```

Sabendo que o heap ordena pelo menor valor e no caso de adicionarmos (distância, vértice) ele ordena pelo primeiro elemento, o código atualiza o heap inicial com v_0, e, enquanto o heap não estiver vazio, chamamos de v_2 o próximo vizinho e, se ele não tiver sido visitado, acessa seu custo e verifica se pode relaxar a aresta. Se puder, adiciona também ao heap. Por fim, marca como checado.

Implementação em Python

```py
import heapq

def cpt_dijkstra_fast(v0, list_adj):
    num_vertices = len(list_adj)
    checked = [False] * num_vertices
    parent = [-1] * num_vertices
    distance = [float('inf')] * num_vertices
```

```
    parent[v0] = v0
    distance[v0] = 0
    heap = []
    heapq.heappush(heap, (0, v0))

    while heap:
        dist_v1, v1 = heapq.heappop(heap)
        if dist_v1 > distance[v1]:
            continue
        if distance[v1] == float('inf'):
            break
        for v2, cost in list_adj[v1]:
            if not checked[v2]:
                if distance[v1] + cost < distance[v2]:
                    parent[v2] = v1
                    distance[v2] = distance[v1] + cost
                    heapq.heappush(heap, (distance[v2], v2))
        checked[v1] = True
    return parent, distance
```

A explicação é análoga. Para analisar a complexidade, note que o heappop é feito dentro do while que acontece para cada vértice(pois o while heap roda no máximo V vezes), e que o heappush acontece dentro do for das arestas (que já discutirmos ter complexidade E). Portanto, é fácil ver que a complexidade é $O(\log(V)(V+E))$. Note:

- Se todos os vértices forem acessíveis a partir de v_0, temos $O(E\log(V))$ (pois isso significa que o grafo é conexo e, sabemos que $E \geq V-1$, e, por isso, E domina).
- Se o grafo for esparso (onde $E \approx V$), a complexidade será $O(V\log(V))$.
- Se o grafo for denso (onde $E \approx V^2$), temos que a complexidade é $O(V^2\log(V))$.Ou seja, apresenta um desempenho inferior à abordagem anterior, que mantém $O(V^2)$ fixo independentemente da densidade das arestas.

4.3.3 eventualmente adicionar corretude

Por fim, como seria a implementação de um algoritmo que funcionasse em grafos com ciclos negativos?

4.3.4 Bellman-Ford

Esse algoritmo é capaz de encontrar caminhos mais baratos em um grafo $G=(V,E)$ mesmo que as arestas possuam custos positivos e negativos. Ele retorna falso se detectar um ciclo negativo.

O algoritmo consiste em relaxar as arestas do grafo sistematicamente,reduzindo progressivamente uma estimativa $d[v]$ para cada vértice $v \in V$ do grafo, até alcançar a menor distância.

Essa é a ideia geral:

1 **insira** v_0 **em** T
2 **defina** $d[v_0] = 0$
3 **execute V - 1 vezes:**
4 | **para cada arsta** (v_i, v_j)**:**
5 | | **aplique o relaxamento**

6 **execute o relaxamento sobre todas as arestas**

7 | **se alguma distância $d[v_k]$ for reduzida, ciclo negativo**

O algoritmo executa as primeiras $V-1$ iterações construindo caminhos com 1 aresta, 2 arestas, até $V-1$ arestas, pois sabemos que um caminho simples pode ter no máximo $V-1$ arestas. Antes de implementarmos o algoritmo, vejamos como ele ocorreria para o grafo de exemplo.

0	0	-	-	-	-	-
1	0	4	9	9	10	12
2	0	2	9	6	8	10
3	0	0	9	4	6	8
4	0	-2	9	2	4	6
5	0	-4	9	0	2	4
6	0	-4	9	0	0	4

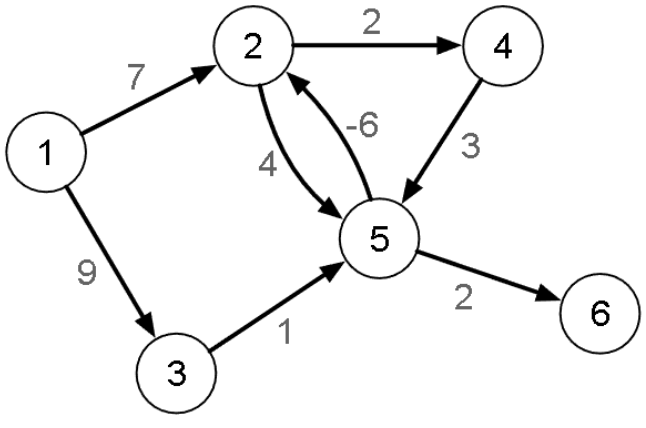


Figura 40: Exemplo do estado do algoritmo Bellman-Ford

Note que temos 6 iterações, 5 dos $V-1$ vértices e o último é a verificação do ciclo negativo. Vamos analisar de perto da primeira iteração: Indo em ordem e sendo o vértice 0 como raiz, temos a distância para ele é 0. Vendo seus filhos, ele adiciona que a distância para o vértice 2 é 7, e para o 3 9, então temos o vetor de distâncias como $[0, 7, 9, -, -, -]$.

Na próxima iteração, temos vamos analisar o vértice 2. Como ele tem uma distância, significa que podemos chegar nele dos vértices que já descobrimos, então podemos realizar a análise. Ele coloca distância 9 no vértice 4, e distância 11 no vértice 5. Temos então $[0, 7, 9, 9, 11, -]$.

Com $i = 2$, estamos no vértice 3 e atualizamos a distância para o 5, ficando com $[0, 7, 9, 9, 10, -]$.

No vértice 4 não conseguimos mudar o valor de nada, já que só mudaríamos no 5, que já é um valor menor.

No vértice 5, conseguimos mudar o valor do vértice 2 e ao vértice 6, e somando -6 ao custo do 5 (para o 2) e 2 ao custo do 5(para o 6), chegamos em $[0, 4, 9, 9, 10, 12]$.

Como o último vértice não tem nenhuma aresta de saída, a primeira iteração se encerra como $[0, 4, 9, 9, 10, 12]$.

Vamos ver como programar esse algoritmo:

C++

```cpp
bool cptBellmanFord(vertex v0, vertex * parent, int * distance) {
    for (int v=0; v < m_numVertices; v++) {
        parent[v] = -1;
        distance[v] = INT_MAX;
    }
    parent[v0] = v0;
    distance[v0] = 0;
    for (int i=1; i <= m_numVertices - 1; i++) {
        for (int v1 = 0; v1 < m_numVertices; v1++) {
            EdgeNode * edge = m_edges[v1];
```

```
            while (edge) {
                vertex v2 = edge->otherVertex();
                int cost = edge->cost();
                if (distance[v1] + cost < distance[v2]) {
                    parent[v2] = v1;
                    distance[v2] = distance[v1] + cost;
                }
                edge = edge->next();
            }
        }
    }
    for (int v1=0; v1 < m_numVertices; v1++) {
        EdgeNode * edge = m_edges[v1];
        while (edge) {
            vertex v2 = edge->otherVertex();
            int cost = edge->cost();
            if (distance[v1] + cost < distance[v2]) {
                return false;
            }
            edge = edge->next();
        }
    }
    return true;
}
```

A explicação já foi explicada, o algoritmo apenas passa por todas as arestas do grafo (pois o for de dentro passa por todos os vértices acessando todas as arestas) $V - 1$, ou seja, temos uma complexidade fácil de $O(VE)$.

Implementação em Python

```py
def bellman_ford(v0, list_adj):
    num_vertices = len(list_adj)
    parent = [-1] * num_vertices
    distance = [float('inf')] * num_vertices
    parent[v0] = v0
    distance[v0] = 0
    for i in range(num_vertices - 1):
        for j in range(num_vertices):
            for vizinho, custo in list_adj[j]:
                if distance[j] + custo < distance[vizinho]:
                    parent[vizinho] = j
                    distance[vizinho] = distance[j] + custo

    for j in range(num_vertices):
        for vizinho, custo in list_adj[j]:
            if distance[j] + custo < distance[vizinho]:
```

```
17                 return False
18
19     return parent, distance
```

Show! Mas, como achar a árvore mais barata que gera o grafo??

Árvore Geradora Minima

5.1 Árvore Geradora Mínima

Uma árvore geradora de um grafo $G = (V, E)$ é um subgrafo T que não possua ciclos e que contenha todos os vértices de G. Se um grafo G possui uma árvore geradora, então ele é conexo (claro, uma árvore é conexa) e possui sempre $V - 1$ arestas. Considere abaixo grafos conexos e não-dirigidos.

Um **corte** é um conjunto de arestas que conecta duas partes de um grafo.Dado um conjunto A de vértices, as arestas que possuem uma ponta em A e a outra ponta no complemento de $\tilde{\text{A}}$ representam um corte.

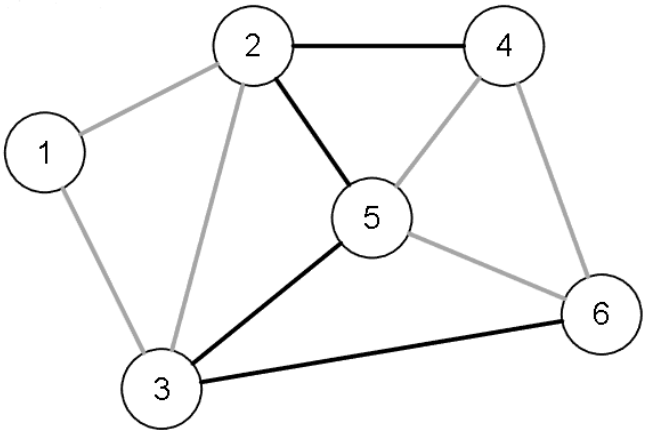


Figura 41: Exemplo de corte em grafo

Dada uma árvore geradora, temos duas operações básicas:
- A adição de uma aresta em uma árvore geradora cria um ciclo.
- A remoção de uma aresta em uma árvore geradora cria um corte.

Dada uma árvore geradora T e um grafo $G = (V, E)$, a propriedade dos ciclos consiste em :
- Se $e_i \notin T$, o grafo $T + e_i$ possui um único ciclo C.
- Se $e_j \in C$, o grafo $T + e_i - e_j$ é uma árvore geradora.

Claro que, se existir um $e_k \in T$, $T - e_k$ produz uma florest com duas componentes conexas em G.

Dada as mesmas coisas, a propriedade dos cortes consiste em:
- Dado $e_i \in T$ e $e_j \in \text{corte}(T - e_i)$.
- $T - e_i + e_j$ é uma árvore geradora.

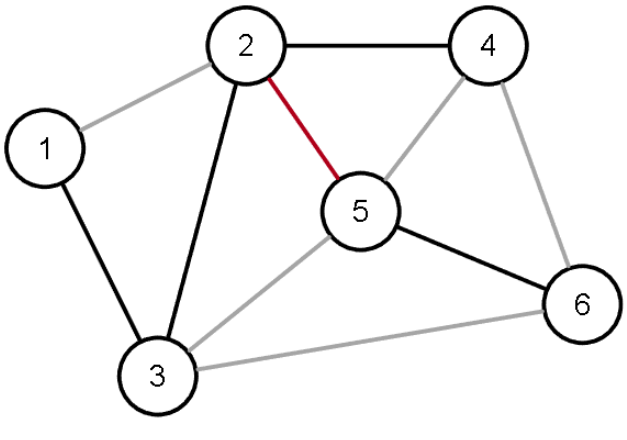


Figura 42: Exemplo da propriedade dos cortes: Se retirarmos a aresta $(2, 5)$, o corte que conectaria as duas árvores geradoras seria qualquer aresta $[(3, 5), (3, 6), (4, 5), (4, 6)]$.

Se $G = (V, E)$ for um grafo não-dirigido com custos nas arestas (com valores positivos e negativos), sabemos que o custo de um subgrafo H de G é calculado pelo somatório de custos das

arestas em H. Com isso, podemos definir que uma **Árvore Geradora Mínima** (Mininum Spamming Tree - MST) de um grafo G é qualquer árvore geradora cujo custo seja mínimo.

Problema: dado $G = (V, E)$ não-dirigido com custos nas arestas encontre uma árvore geradora mínima.

5.2 Algoritmo de Prim

Esse algoritmo é capaz de encontrar a MST de um grafo $G = (V, E)$.

5.2.1 Prim Slow

Dada uma árvore T de G, considere a franja de T como o corte cuja margem é composta pelos vértices em T. Ideia geral do algoritmo:

1 **Escolha a raiz de T**
2 **Enquanto a franja não estiver vazia:**
3 | **Escolha a aresta de menor custo**
4 | **Insira a aresta e o vértice em T**

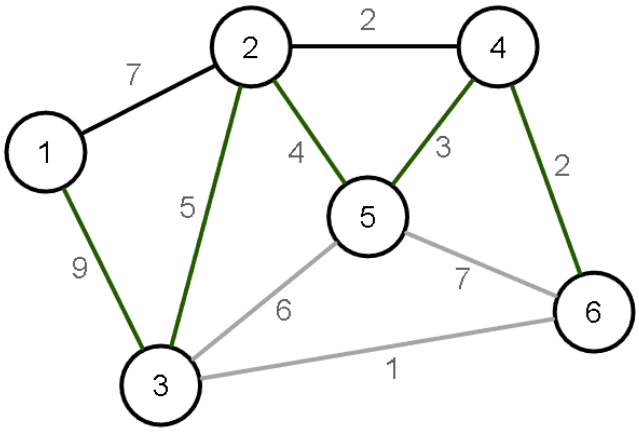


Figura 43: Exemplo da franja para $T = \{1, 2, 4\}$, com as arestas em verde sendo a franja.

Vamos ver sua implementação:

C++

```
void mstPrimSlow(vertex * parent) {
    for (vertex v=0; v < m_numVertices; v++) { parent[v] = -1; }
    parent[0] = 0;
    while (true) {
        int min = INT_MAX;
        vertex treeV, newV = -1;
        for (vertex v1=0; v1 < m_numVertices; v1++) {
            if (parent[v1] == -1) { continue; }
            EdgeNode * edge = m_edges[v1];
            while (edge) {
                vertex v2 = edge->otherVertex();
                int cost = edge->cost();
                if (parent[v2] == -1 && cost < min) {
                    min = cost;
                    treeV = v1;
                    newV = v2;
                }
                edge = edge->next();
            }
        }
        if (min == INT_MAX) { break; }
        parent[newV] = treeV;
    }
}
```

O código preenche o vetor de `parent` e abrimos um while true que só para de funcionar quando o `min` não mudou durante toda uma interação. Iniciamos o min, e as variáveis `treeV`, que é o vértice pertencente a T, e `newV`, que é o vértice a ser avaliado.

Então, percorremos todos os vértices que já estão na árvore (chamamos de v_1 e fazemos a verificação vendo se eles já foram incluidos no vetor `parent`), e para cada aresta de `newV` (que chamamos de v_2), verificamos o custo de adicionar a árvore. Se for menor do que o mínimo, marcamos o respectivo pai e filho, e depois do for acabar, adicionamos essa aresta no vetor de `parent`.

Implementação em Python

```py
def mst_prim_slow(v0, list_adj):
    num_vertices = len(list_adj)
    parent = [-1] * num_vertices
    parent[v0] = v0
    while True:
        min = float('inf')
        treeV, newV = -1, -1
        for i in range(num_vertices):
            if parent[i] == -1:
                continue
            for vizinho, custo in list_adj[i]:
                if parent[vizinho] == -1 and custo < min:
                    min = custo
                    treeV = i
                    newV = vizinho
        if min == float('inf'):
            break
        parent[newV] = treeV

    return parent
```

A ideia é a mesma, claro. Olhando para complexidade, a cada iteração, adicionamos um novo vértice à árvore. No pior caso, onde todos os vértices tem pai, o for passa por todas as arestas para adicionar um único vértice novo, e como rodamos até V vezes, o algoritmo é $O(V + E)$.

A **corretude** do algoritmo pode ser avaliada através do critério de minimalidade baseado em cortes. Uma árvore geradora T é uma MST se e somente se cada aresta $e_k \in T$ apresentar o menor custo no corte fundamental de e_k relativo à T.

Uma coisa que podemos observar no algoritmo é que essa versão calcula a franja a cada iteração, em vez de modificar gradualmente à medida que a árvore é construída. Como melhoramos isso?

5.2.2 Prim Fast

Uma estratégia consiste em manter uma estrutura de dados com o custo de cada vértice que pode ser adicionado a T. Considere como **fronteira** todos os vértices que podem ser acessados à partir da franja e não pertencem a T. A nova ideia é:

1 **Escolha a raiz de T**
2 **Enquanto a franja não estiver vazia:**

Procure v_k na fronteira com o menor custo

Insira V_k em T

Atualize a fronteira adicionando os acessíveis de v_k

C++

```cpp
void mstPrimFastV1(vertex * parent) {
    bool inTree[m_numVertices];
    int vertexCost[m_numVertices];
    for (vertex v = 0; v < m_numVertices; v++) {
        parent[v] = -1;
        inTree[v] = false;
        vertexCost[v] = INT_MAX;
    }
    parent[0] = 0;
    inTree[0] = true;
    EdgeNode * edge = m_edges[0];
    while (edge) {
        vertex v2 = edge->otherVertex();
        parent[v2] = 0;
        vertexCost[v2] = edge->cost();
        edge = edge->next();
    }
    while (true) {
        int minCost = INT_MAX;
        vertex v1 = -1;
        for (vertex v = 0; v < m_numVertices; v++) {
            if (!inTree[v] && vertexCost[v] < minCost) {
                minCost = vertexCost[v];
                v1 = v;
            }
        }
        if (minCost == INT_MAX) {
            break;
        }
        inTree[v1] = true;
        edge = m_edges[v1];
        while (edge) {
            vertex v2 = edge->otherVertex();
            int cost = edge->cost();
            if (!inTree[v2] && cost < vertexCost[v2]) {
                vertexCost[v2] = cost;
                parent[v2] = v1;
            }
            edge = edge->next();
        }
    }
```

```
42  }
```

No código, usaremos três listas, e declaramos e preenchemos elas inicialmente. Após declararmos para o vértice inicial 0 em `parent[0] = 0` e `inTree[0] = 1`, fazemos um while especificamente para as arestas do vértice inicial para preencher o array `vertexCost` com os respectivos pesos de cada vizinho.

Iniciamos um while para fazer a verificação de cada vértice, e iniciamos `minCost` e `v1`. Para cada vértice, verificamos se ele não está na árvore e se o custo desse vértice é menor que o mínimo. Se for, atualizamos o custo e o vértice. Verificamos a condição de parada e marcamos `v1` como true na árvore, pois é o menor que achamos.

Passamos no outro while por cada aresta do vértice selecionado para atualizar os custos dos seus vizinhos (`v2`). Verificamos se esse vizinho ainda não está incluído na árvore e se o peso da aresta atual é menor do que o custo que já tínhamos registrado para ele em `vertexCost`. Se for menor, atualizamos o `vertexCost[v2]` com esse novo peso (pois achamos uma conexão mais barata) e definimos o `parent[v2]` como sendo o `v1` (isso é a atualização da franja). Ao fim, temos o menor custo possível para cada vértice em `VertexCost` e a árvore mínima pelo `parent`.

Implementação em Python

```py
def mst_prim_fastv1(list_adj):
    num_vertices = len(list_adj)
    parent = [-1] * num_vertices
    intree = [False] * num_vertices
    vertexcost = [float('inf')] * num_vertices
    parent[0] = 0
    intree[0] = True
    for vizinho, custo in list_adj[0]:
        parent[vizinho] = 0
        vertexcost[vizinho] = custo

    while True:
        mincost = float('inf')
        v1 = -1
        for v in range(num_vertices):
            if not intree[v] and vertexcost[v] < mincost:
                mincost = vertexcost[v]
                v1 = v
        if mincost == float('inf'):
            break
        intree[v1] = True
        for v2,custo in list_adj[v1]:
             if not intree[v2] and custo < vertexcost[v2]:
                vertexcost[v2] = custo
                parent[v2] = v1

    return parent
```

É análoga a ideia em C++. Analisando a complexidade, vemos que a cada iteração do while de fora, passamos por todos os vértices, e a cada aresta desse vértice. Como queremos passar por todos os vértices, sabemos que temos que fazer isso V vezes. Isso intuitivamente dá, então, $O(V^2 + E)$. Como $E < V^2$(ou $E \propto V^2$), isso é simplesmente $O(V^2)$. Será que tem como ficar ainda melhor?

5.2.3 Prim Fast V2

E se mantessemos os vértices da fronteira em ordem crescente de custo? Dessa forma, não seria necessário procurar o vértice que apresenta o menor custo à cada iteração (basta usar o famoso heap mínimo).

C++

```
void mstPrimFastV2(vertex * parent) {
    bool inTree[m_numVertices];
    int vertexCost[m_numVertices];
    for (vertex v = 0; v < m_numVertices; v++) {
        parent[v] = -1;
        inTree[v] = false;
        vertexCost[v] = INT_MAX;
    }
    parent[0] = 0;
    inTree[0] = true;
    EdgeNode * edge = m_edges[0];
    while (edge) {
        vertex v2 = edge->otherVertex();
        parent[v2] = 0;
        vertexCost[v2] = edge->cost();
        edge = edge->next();
    }
    Heap heap;
    for (vertex v = 1; v < m_numVertices; v++) {
        heap.insert_or_update(vertexCost[v], v);
    }
    while (!heap.empty()) {
        vertex v1 = heap.top().second;
        heap.pop();
        if (vertexCost[v1] == INT_MAX) {
            break;
        }
        inTree[v1] = true;
        edge = m_edges[v1];
        while (edge) {
            vertex v2 = edge->otherVertex();
            int cost = edge->cost();
            if (!inTree[v2] && cost < vertexCost[v2]) {
                vertexCost[v2] = cost;
                parent[v2] = v1;
                heap.insert_or_update(vertexCost[v2], v2);
            }
```

```
            edge = edge->next();
        }
    }
}
```

A implementação é muito parecida, e o que muda é que, antes de fazer o while principal, iniciamos o heap, e preenchemos com os custos iniciais de cada vértice (incluindo os que já atualizamos que vem da raiz). Ainda, no while principal, pegamos o topo diretamente do heap e depois retiramos ele ($\log(V)$), e, na hora de atualizarmos a franja, também inserimos no heap.

```py
import heapq

def mst_prim_fastv2(v0, list_adj):
    num_vertices = len(list_adj)
    parent = [-1] * num_vertices
    intree = [False] * num_vertices
    vertexcost = [float('inf')] * num_vertices
    parent[v0] = v0
    intree[v0] = True
    for vizinho, custo in list_adj[v0]:
        parent[vizinho] = v0
        vertexcost[vizinho] = custo
    heap = []
    for v in range(num_vertices):
        if v != v0:
            heapq.heappush(heap, (vertexcost[v], v))

    while heap:
        custo_v1, v1 = heapq.heappop(heap)
        if intree[v1] or custo_v1 > vertexcost[v1]:
            continue
        if custo_v1 == float('inf'):
            break
        intree[v1] = True
        for v2, custo in list_adj[v1]:
            if not intree[v2] and custo < vertexcost[v2]:
                vertexcost[v2] = custo
                parent[v2] = v1
                heapq.heappush(heap, (vertexcost[v2], v2))

    return parent
```

Para a complexidade, muito parecido com o djikstra, o que vemos aqui é que temos um heap pop quando passamos por todos os vértices e temos um heap push quando temos que adicionar(ao passarmos pelas arestas). Ou seja, juntando com a explicação de complexidades anteriores, isso dá simplesmente $O((V+E)\log(V))$.

5.3 Algoritmo de Kruskal

A estratégia desse algoritmo consiste em crescer uma floresta $F = (V', E')$ até que ela se torne uma árvore geradora $F = (V, E')$, diferente de crescer arbitrariamente como o Prim.

5.3.1 Kruskal Slow

Uma aresta e_k é externa a floresta F se $e_k \notin F$ e o grafo $F + e_k$ é uma floresta. Ideia geral do algoritmo:

1 **Inicialize a floresta com todos os vértices e nenhuma aresta**
2 **Escolha a aresta $e_k = (v_i, v_j)$ de G que possua o menor custo**
3 **Insira e_k em F.**

Vamos ver como seria a implementação disso:

C++

```cpp
void mstKruskalSlow(Edge * edges) {
    vertex group[m_numVertices];
    for (vertex v=0; v < m_numVertices; v++) { group[v] = v; }
    int k = 0;
    while (true) {
        int minCost = INT_MAX;
        vertex minV1, minV2 = -1;
        for (vertex v1=0; v1 < m_numVertices; v1++) {
            EdgeNode * edge = m_edges[v1];
            while (edge) {
                vertex v2 = edge->otherVertex();
                int cost = edge->cost();
                if (v1 < v2 && group[v1] != group[v2] && cost < minCost) {
                    minCost = cost;
                    minV1 = v1;
                    minV2 = v2;
                }
                edge = edge->next();
            }
        }
        if (minCost == INT_MAX) return;
        edges[k++] = Edge(minV1, minV2, minCost);
        vertex leaderV1 = group[minV1];
        vertex leaderV2 = group[minV2];
        for (vertex v=0; v < m_numVertices; v++) {
            if (group[v] == leaderV2) {
                group[v] = leaderV1;
            }
        }
    }
}
```

No algoritmo, recebemos uma lista vazia denominada `edges`, onde iremos colocar as arestas da forma `(vértice1, vértice2, custo)`. Criamos um vetor para entendermos de que parte da floresta

cada vértice pertence, o `group`. E o `k` vai servir como contador de arestas. Após preenchermos cada vértice a cada grupo próprio, iniciamos o while com o `mincost` e o `minV1` e `minV2`, pra marcar a aresta.

Então, para cada vértice eu pego suas arestas, e faço 3 verificações para atualizar parâmetros:

- `v1 < v2` - Para evitar duplicatas em caso de grafo não dirigido, por exemplo a aresta $(0, 1)$ é a mesma que $(1, 0)$.
- `group[v2] != group[v1]` - as arestas devem ser de grupos diferentes, caso contrário certamente já pegamos a menor para a árvore.
- e o custo tem que ser menor que o último.

Se passou nessas verificações, então ele deve ser atualizado, e atualizamos a aresta de menor custo, em `minV1` e `minV2`. Após a verificação de contorno e a declaração de da aresta no edges. Por fim, salvamos os líderes de cada grupo e atualizamos o grupo de um dos grupos.

Que ideia do caramba!

Implementação em Python

```py
def mst_kruskal_slow(list_adj):
    num_vertices = len(list_adj)
    group = [v for v in range(num_vertices)]
    edges = []
    while True:
        mincost = float('inf')
        minv1, minv2 = -1, -1
        for v in range(num_vertices):
            for vizinho, custo in list_adj[v]:
                if v < vizinho and group[v] != group[vizinho] and custo < mincost:
                    mincost = custo
                    minv1 = v
                    minv2 = vizinho

        if mincost == float('inf'):
            break
        edges.append((minv1, minv2, mincost))
        leader1 = group[minv1]
        leader2 = group[minv2]
        for v in range(num_vertices):
            if group[v] == leader2:
                group[v] = leader1

    return edges
```

A explicação é análoga à anterior, e, olhando para a complexidade, o while True passa em cada vértice V vezes, e os dois primeiros fors passam por todas as arestas (a cada iteração!). Por fim, o for final percorre todos os vértices novamente, trazendo uma complexidade de $O(V(V + E))$.

A corretude do algoritmo pode ser avaliada através do critério de minimalidade baseado em ciclos. Uma árvore geradora T é uma MST de G se e somente se cada aresta $e_k \notin T$ apresentar o maior custo no ciclo fundamental de e_k relativo à T (não entendi, e espero que vocês acreditem).

Essa versão é um pouco lenta pois:

- A cada iteração todas as arestas são verificadas em busca da com menor custo.
- A cada iteração todos os vértices são verificados para avaliar se é necessário atualizar seu grupo.

Como melhorar isso?

5.3.2 Kruskal Fast

Podemos ordenar as arestas por seu custo e utilizar uma estrutura de dados mais eficiente para fazer a busca e união dos vértices.

Na estrutura **union-find**, todo elemento é associado à um conjunto:

- `group[v] = v`, se for o líder do grupo;
- `group[v] = v'`, se não for o líder.

Essa representação força uma estrutura de árvore entre os elementos de mesmo conjunto (também armazenamos o tamanho).

Seguindo essa abordagem podemos comparar se dois elementos pertencem ao mesmo grupo verificando se o líder de cada grupo é o mesmo.

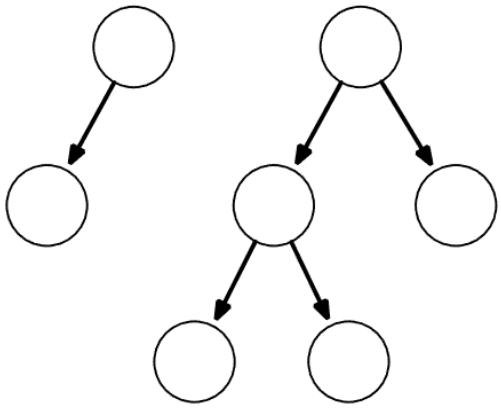


Figura 44: Exemplo bobo do tal do union-find.

Como precisamos chegar na raiz, isso consome até $O(\log(n))$ no pior caso. A união de grupos passa a ser realizada definindo como pai do menor conjunto o pai do maior conjunto.

Voltando ao Kruskal, podemos otimizá-lo usando o que acabamos de aprender. Veja a ideia:

Inicialize a floresta com todos os vértices e sem nenhuma aresta
Crie a lista de arestas ordenando-a pelo custo
Para cada aresta $e_k = (v_i, v_j)$
    Obtenha os líderes dos vértices
    Se forem diferentes, una-os
    insira e_k em F
    Pare ao encontrar $V - 1$ arestas

```cpp
void mstKruskalFast(Edge * mstEdges) {
    vector<Edge> edges(m_numEdges);
    int currentEdge = 0;
    for (vertex v1=0; v1 < m_numVertices; v1++) {
        EdgeNode * edge = m_edges[v1];
        while (edge) {
            vertex v2 = edge->otherVertex();
            if (v1 < v2) {
                edges[currentEdge++] = Edge(v1, v2, edge->cost());
            }
            edge = edge->next();
        }
    }
    sort(edges.begin(), edges.end(), compareEdges);
```

```
15      UnionFind uf(m_numVertices);
16      currentEdge = 0;
17      for (int e=0; currentEdge < m_numVertices - 1; e++) {
18          Edge & edge = edges[e];
19          vertex leaderV1 = uf.findE(edge.v1());
20          vertex leaderV2 = uf.findE(edge.v2());
21          if (leaderV1 != leaderV2) {
22              uf.unionE(leaderV1, leaderV2);
23              mstEdges[currentEdge++] = edge;
24          }
25      }
26  }
```

Após montar a estrutura edges bonitinha e ordenada pelo custo, fazemos literalmente o que foi dito no pseudocódigo - passamos por cada vértice, identificamos os líderes e, se os líderes forem diferentes, adicionamos na lista de retorno a aresta escolhida.

Não vou implementar isso em Python por falta de tempo