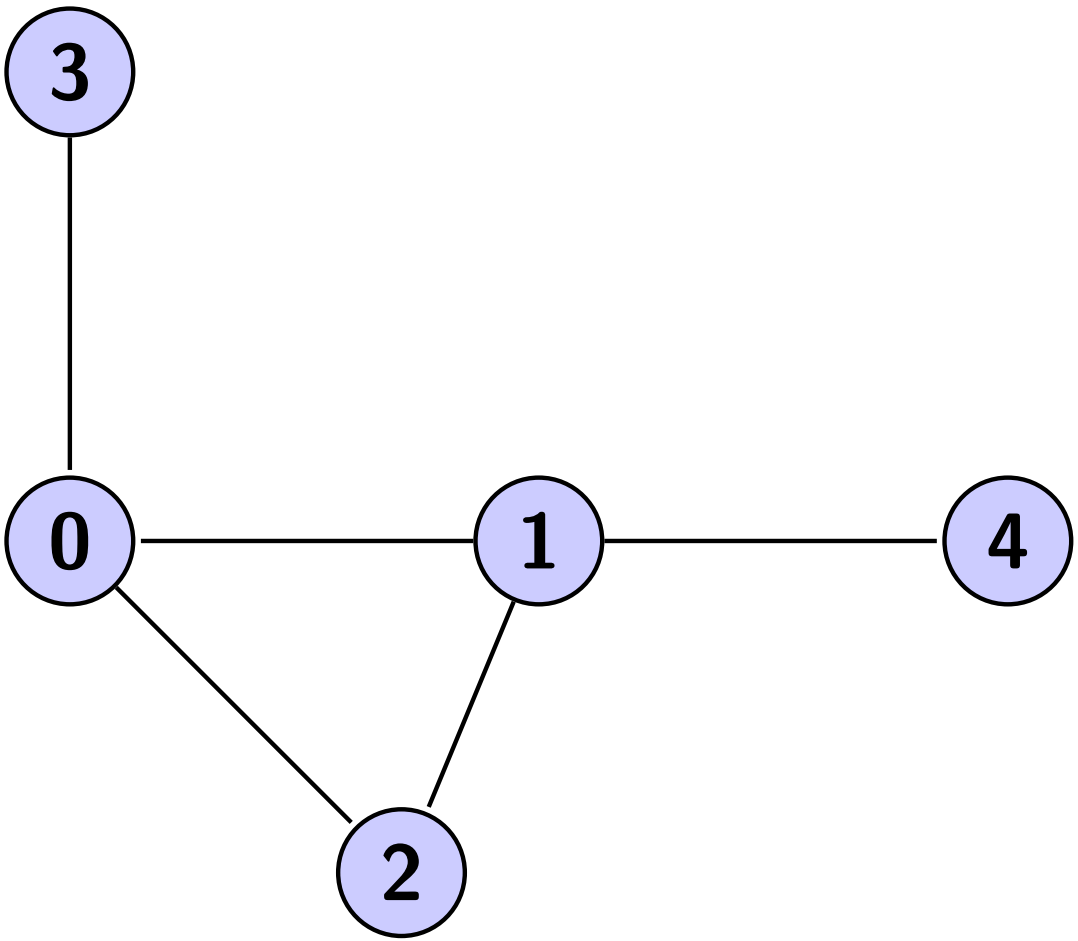
GBC052 - Análise de Algoritmos

Marcelo Keese Albertini

2 de Julho de 2018

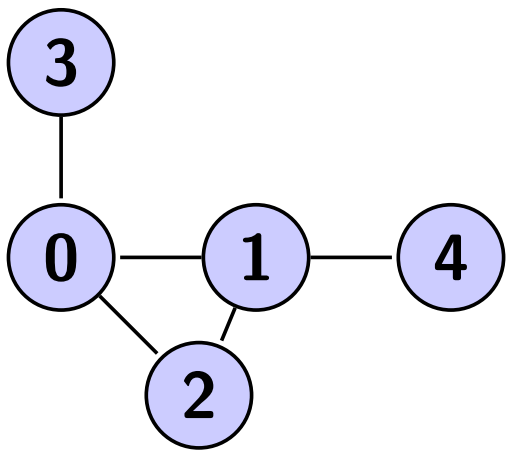
Conceitos básicos

- Vértice – uma entidade representada
- Aresta – ligação entre dois vértices
- Grafo – conjunto de vértices e arestas



Notação para escrever Grafos

- Um grafo é representado por $G = (V, A)$.
- A letra G é o nome do grafo.
- A letra V é o conjunto de vértices.
- A letra A é o conjuntos de arestas.



Exemplo

$G = (V, A)$,
$V = \{0, 1, 2, 3, 4\}$,
$A = \{(0, 3), (0, 1), (0, 2), (1, 2), (1, 4)\}$

Conceitos básicos

Exemplo

$G = (V, A)$,
$V = \{0, 1, 2, 3, 4\}$,
$A = \{(0, 3), (0, 4), (0, 1), (0, 2), (1, 2), (1, 4)\}$

Definições

- Ordem de um grafo: número de vértices
  - ordem de $G = 5$
- Número de vértices $|V|$ = cardinalidade/tamanho de V
  - $|V| = 5$
- Número de arestas $|A|$ = cardinalidade/tamanho de A
  - $|V| = 6$

Grafos direcionados ou Dígrafos

- Arestas tem direção, indicada por setas
- Arestas podem ser em laço ("loop"), ou seja, referenciar o próprio vértice de saída

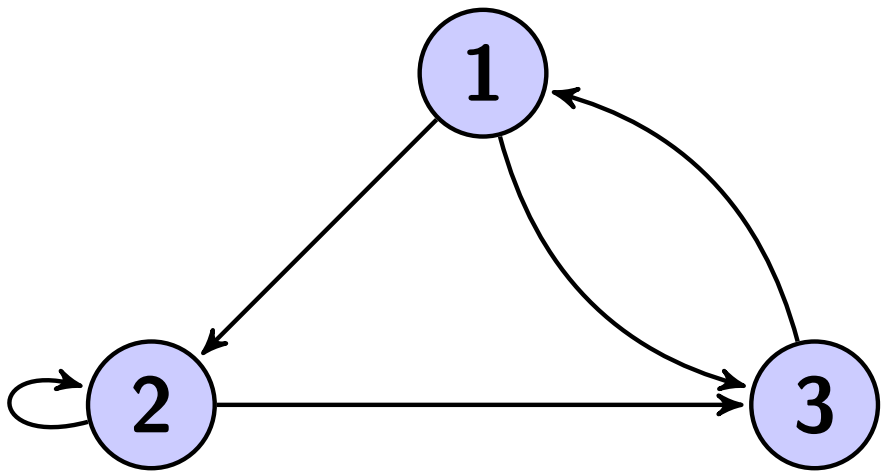
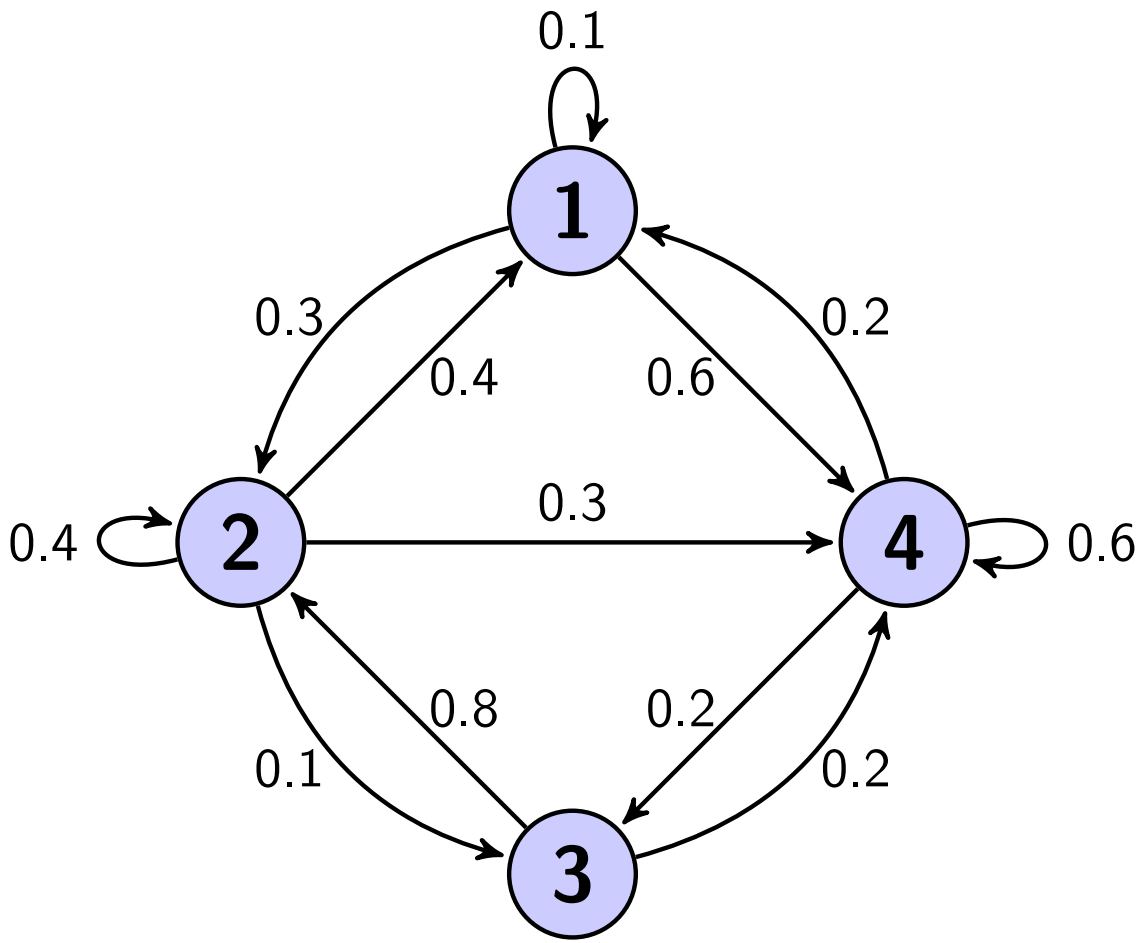


Figura: $G = (V, A)$, $V = \{1, 2, 3\}$, $A = \{(1, 2), (1, 3), (2, 2), (2, 3), (3, 1)\}$

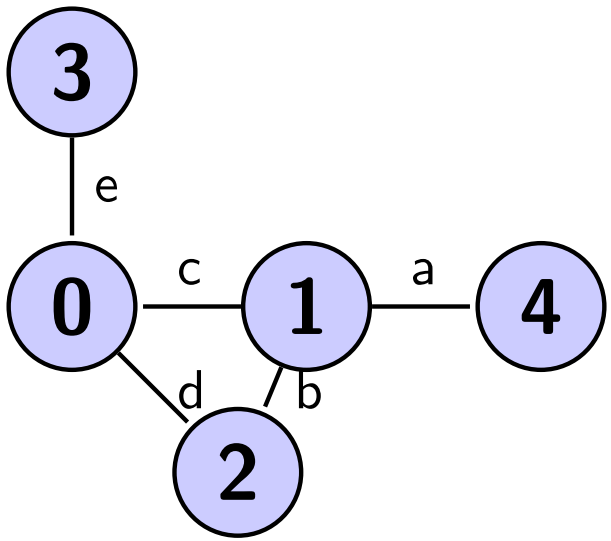
- Aresta – ligação entre dois vértices
  - pode ter direção
  - pode ter peso
- Aresta múltipla
  - pode haver mais de 1 aresta ligando os mesmos dois vértices



Vértices adjacentes (ou vizinhos)

Exemplo: grafo $G = (V, A)$

$V = \{0, 1, 2, 3, 4\}$,
$A = \{e = (0, 3), c = (0, 1), d = (0, 2), b = (1, 2), a = (1, 4)\}$



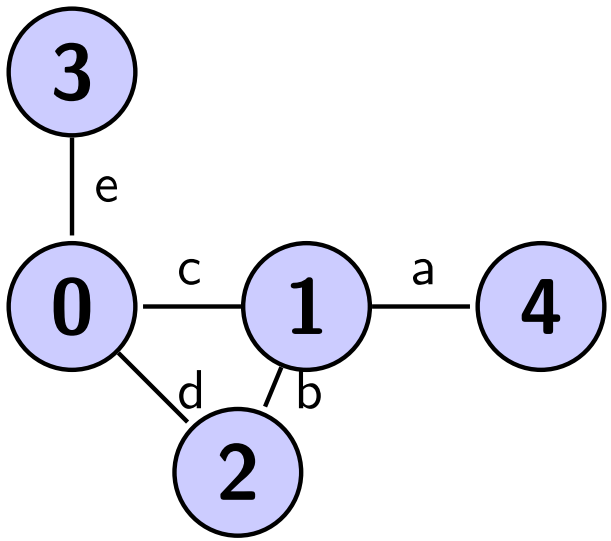
Vértices adjacentes

Vértice 0 é adjacente a 3, 1 e 2
Vértice 3 é adjacente a 0.

Arestas adjacentes (ou vizinhas)

Exemplo: grafo $G = (V, A)$

$V = \{0, 1, 2, 3, 4\}$,
$A = \{e = (0, 3), c = (0, 1), d = (0, 2), b = (1, 2), a = (1, 4)\}$



Arestas adjacentes

Aresta a é adjacente às arestas b e c.
Aresta c é adjacente às arestas e, d, a, b.

Outras definições

Definições

- Grafo trivial
  - grafo com apenas um vértice
- Grafo simples
  - grafo que não tem laços nem arestas múltiplas

Graus de vértices

Grafos direcionados

- Grau de entrada de um vértice
  - número de arestas que chegam ao vértice
- Grau de saída de um vértice
  - número de arestas que saem do vértice
- Grau de um vértice
  - número de arestas que chegam e saem

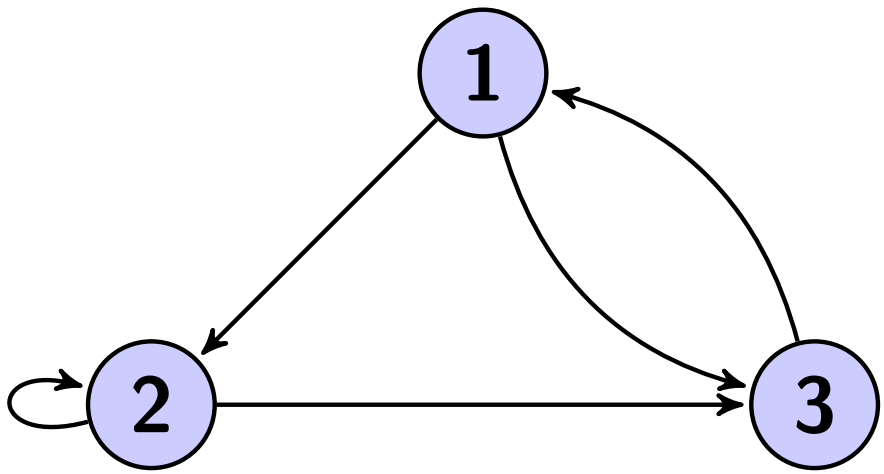


Figura: $grau(1) = 3$

Caminho em um grafo

Definições: caminho

- Caminho é uma sequência de vértices adjacentes
- Um vértice é alcançável se existe um caminho a ele.

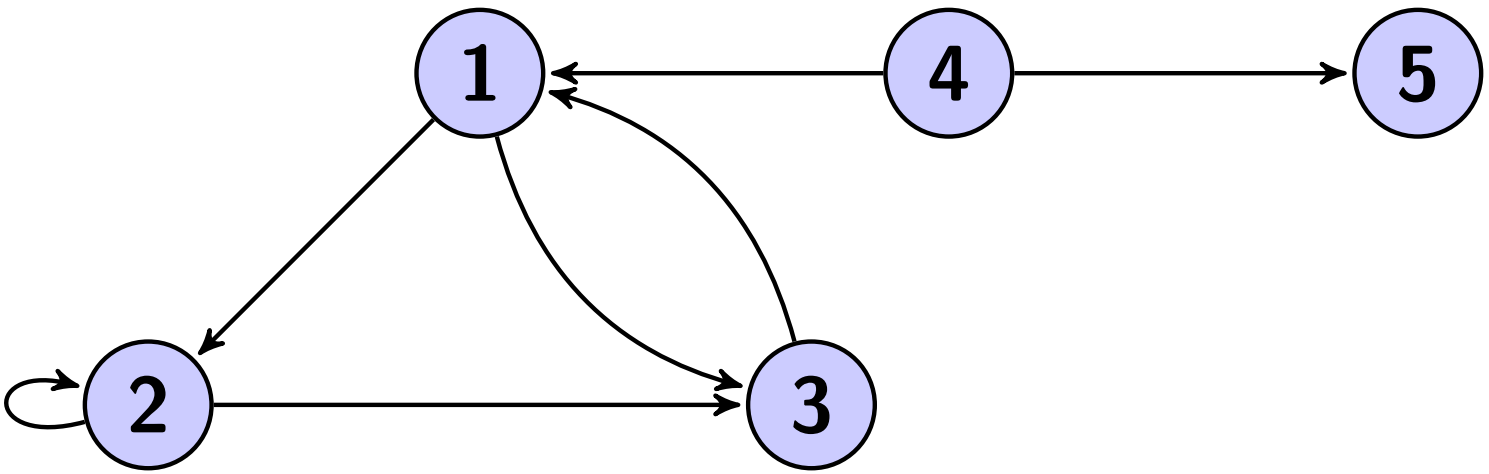


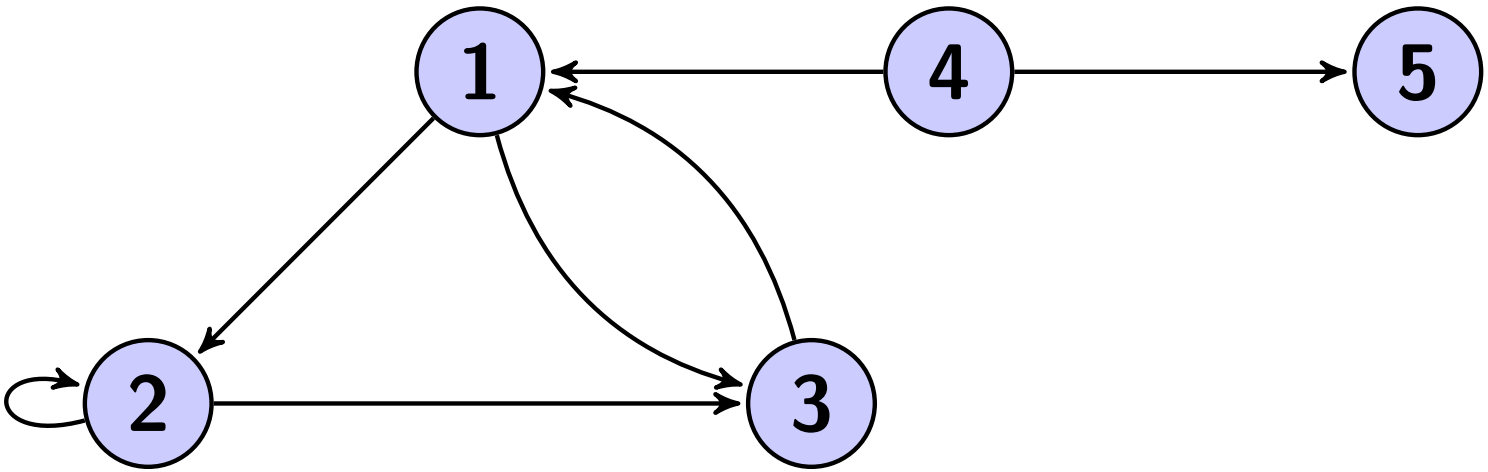
Figura: $(1, 2, 3, 1)$ é um caminho.

Vértice 4 não é alcançável a partir de nenhum outro vértice.
Vértice 5 só é alcançável a partir do vértice 4.

Caminho em um grafo

Definições: caminho

- Caminho é simples se todos os vértices são distintos

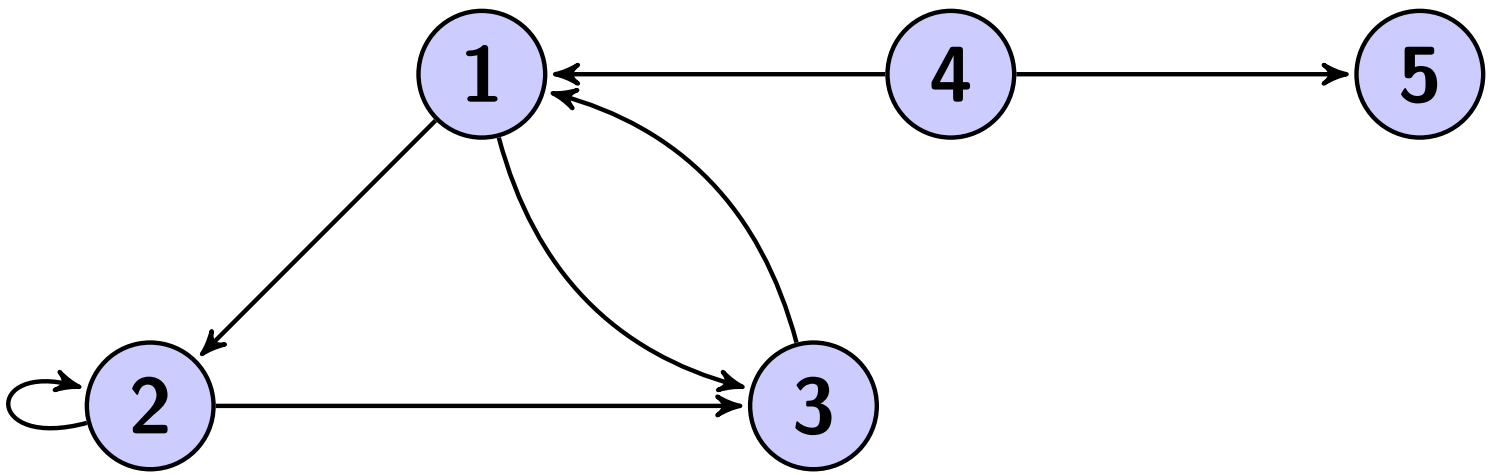


$(1, 2, 3)$ é um caminho simples.
$(1, 2, 3, 1)$ não é um caminho simples.

Caminhos cíclicos ou ciclos

Definições: ciclo

- Um ciclo é um caminho que começa e termina com o mesmo vértice
- Caso especial: um laço é ciclo com uma só aresta

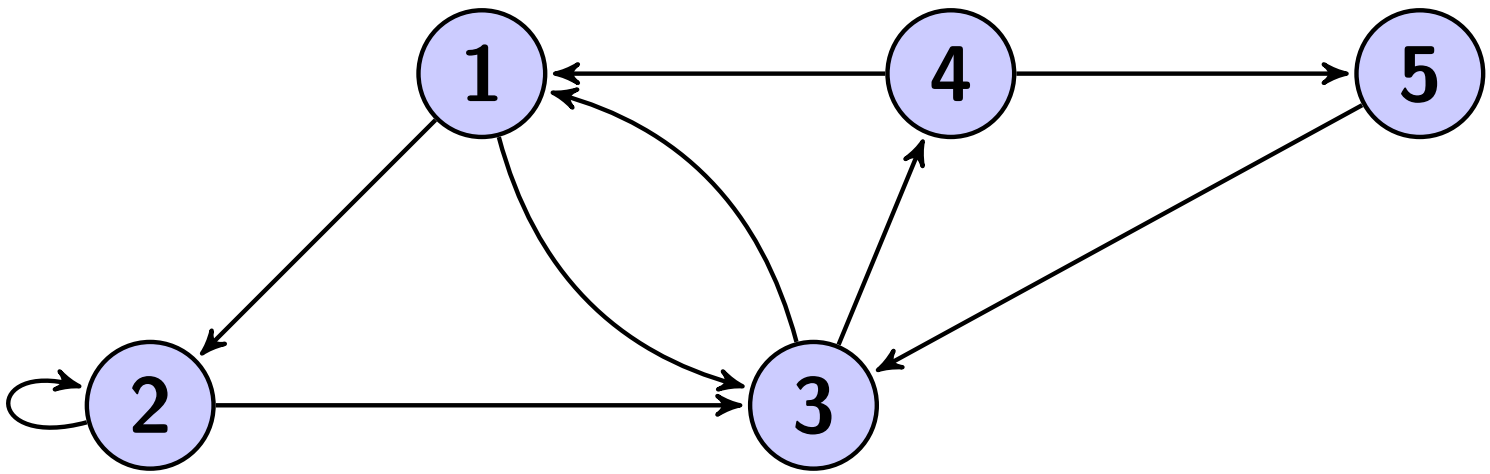


$(1, 2, 3, 1)$ é um ciclo.
$(2, 2)$ é um ciclo.

Caminhos hamiltonianos

Definições: caminho hamiltoniano

- Caminho hamiltoniano contém cada vértice do grafo exatamente uma vez

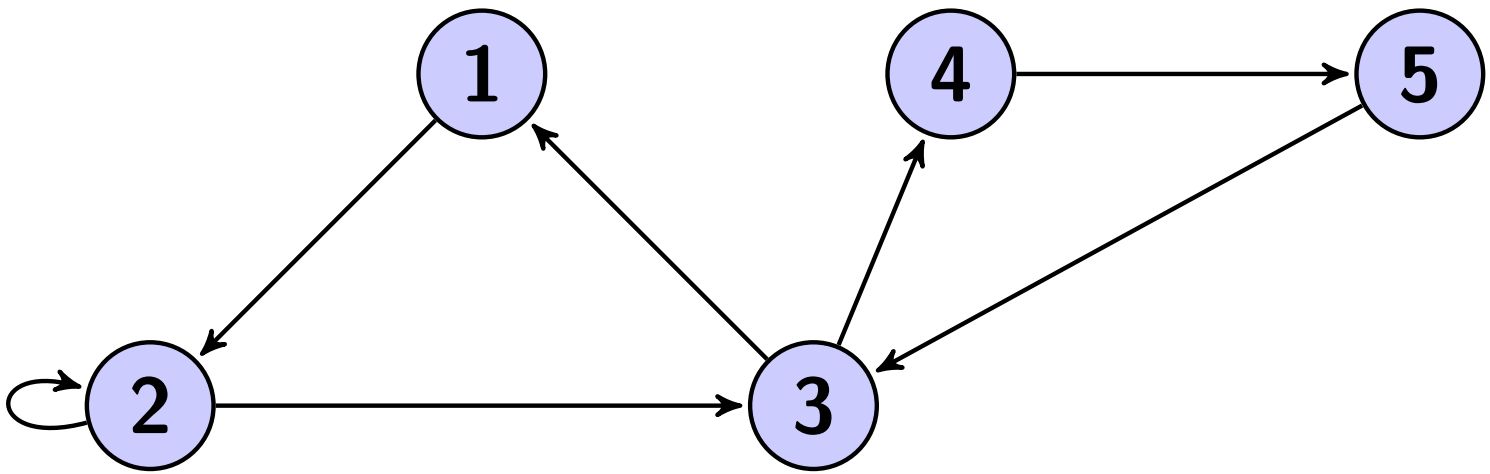


$(4, 5, 3, 1, 2)$ é um caminho hamiltoniano.
$(4, 5, 3, 1, 2, 3, 4)$ é um ciclo hamiltoniano.

Caminhos eulerianos

Definições: caminho euleriano

- Caminho euleriano contém cada aresta do grafo exatamente uma vez



$(4, 5, 3, 1, 2, 2, 3, 4)$ é um caminho euleriano.
Problema da 7 pontes visa encontrar um caminho euleriano.

Grafos conexos e desconexos

Definições: em um grafo não direcionado

- Um grafo é conexo se cada vértice tem pelo menos um caminho a qualquer outro vértice

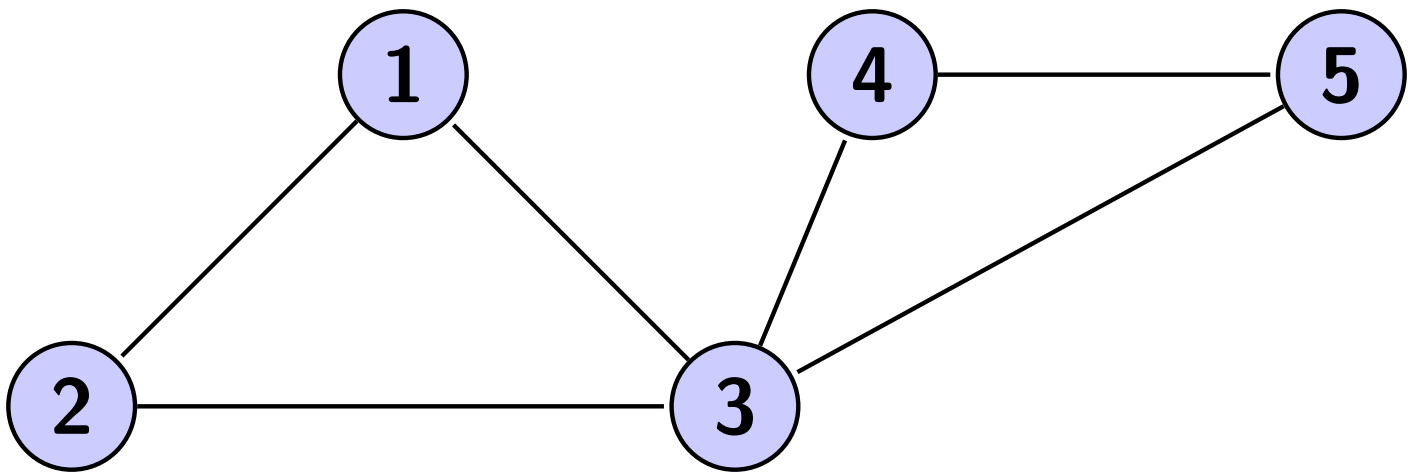


Figura: Grafo conexo

Grafos conexos e desconexos

Definições: em um grafo não direcionado

- Um grafo é conexo se cada vértice tem pelo menos um caminho a qualquer outro vértice

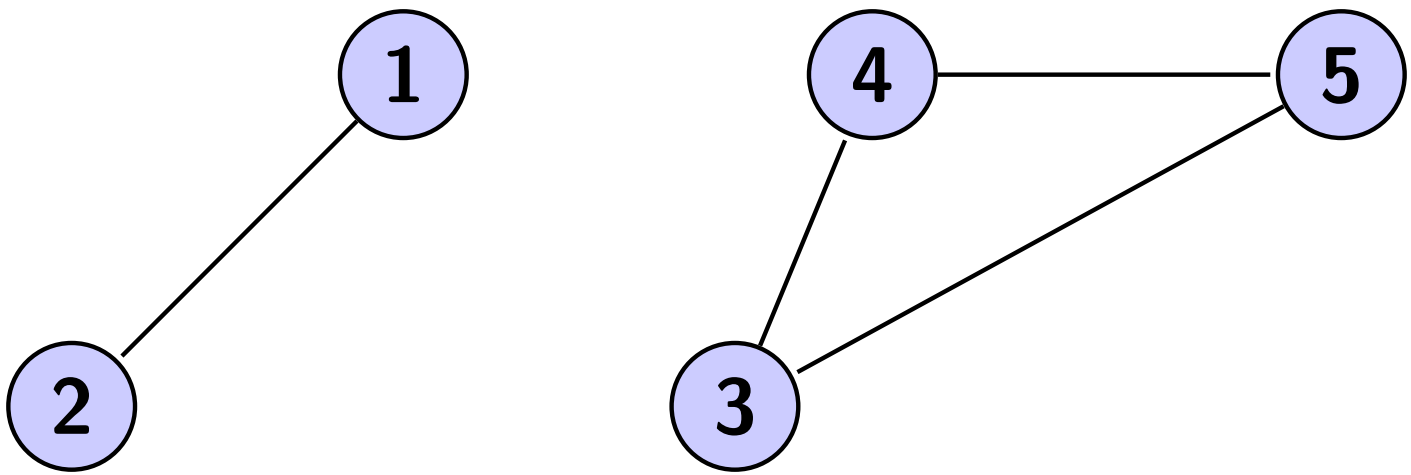


Figura: Grafo desconexo

Grafos completos

Definições: grafo completo

- Um grafo é completo se todos os vértices tiverem aresta ligando a todos os outros vértices
- Um grafo completo com n vértices é chamado K_n

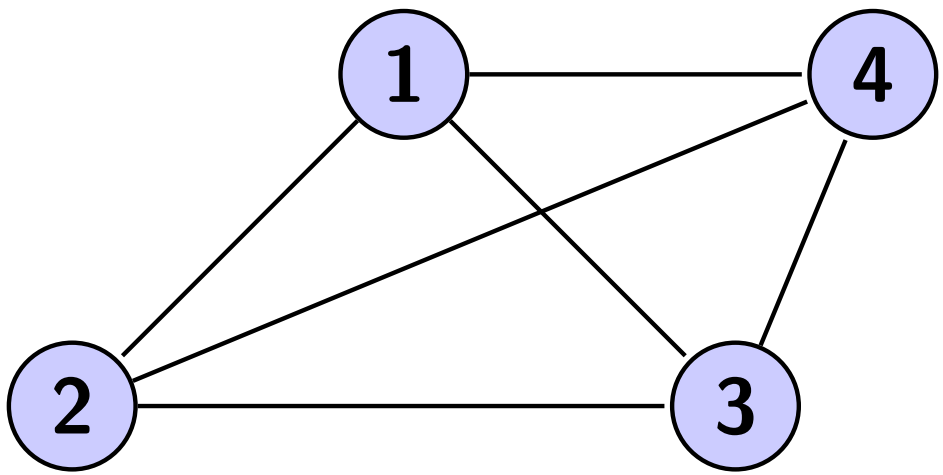


Figura: Grafo completo K_4

Aplicações de grafos

grafo	**vértice**	**aresta**
comunicações	telefone, computadores	fibra ótica
circuito	porta, registrador, processador	fio
finanças	ações, cotações	transações
transporte	cruzamento, aeroporto	rodovia, rota aéra
internet	rede classe C	conexão
jogos	posição no tabuleiro	movimento de peça
rede social	pessoa, ator	amizade, alenco
rede neural	neurônio	sinapse
rede proteica	proteína	interação de proteínas
composto químico	molécula	ligação

API de Grafos

public class	Grafo	
	Grafo(int V)	criar grafo vazio com V vértices
	Grafo(InputStream in)	criar grafo a partir de entradas
void	novaAresta(int v, int w)	criar aresta
Iterable<Integer>	adj(int v)	vértices adjacentes a *v*
int	V()	número de vértices
int	E()	número de arestas

```
1 public static void main(String args[]) {
2   Grafo G = new Grafo(System.in);
3
4   for (int v = 0; v < G.V(); v++)
5     for (int w: G.adj(v))
6       System.out.println(v+"--"+w);
7 }
```

Operações em grafos

A partir da API básica é possível construir outras operações e aplicações completas:

```java
/** Calcular o grau de um vértice */
public static int grau(Grafo G, int v) {
  int grau = 0;
  for (int w: G.adj(v)) grau++;
  return grau;
}
```

```java
/** Calcular o grau máximo de um grafo */
public static int grauMax(Grafo G) {
  int max = 0;
  for (int v = 0; v < G.V(); v++)
    if (grau(G, v) > max)
      max = grau(G, v);
  return grau;
}
```

Representação de grafos

- Opções
  - Lista de arestas
    - Lista que contém pares de arestas
  - Matriz de adjacências
    - Matriz (array de arrays)
  - Listas de adjacências
    - Array com listas das adjacências de cada vértice

Representação de grafos

Na prática: usar listas de adjacências

- Algoritmo em geral operam nos vértices
- Grafos reais tendem a ser esparsos (número grande de vértices, número pequeno de arestas por vértice)

representação	espaço	criar aresta	existe v-w?	iterar adj(v)
lista de arestas	E	1	E	E
matriz de adjacências	V^2	1	1	V
listas de adjacências	E+V	1	grau(v)	grau(v)

E = número de arestas
V = número de vértices

Busca em profundidade

- O algoritmo é a base para tarefas importantes
  - Verificação de grafos acíclicos
  - Ordenação topológica
  - Detecção de componentes fortemente conectados.

Exemplo de uso de Busca em profundidade

Desacoplamento

É possível fazer separação do Grafo da operação: desacoplamento.

Padrão de projeto para desacoplar o Grafo de suas operações

- Criar um objeto Grafo
- Passar o Grafo para uma rotina de processamento de uma operação
- Consultar a rotina por informações relacionadas à operação

public class	Caminhos	
	Caminhos(Grafo G, int v)	processar grafo a partir do vértice v
boolean	temLigacao(int w)	existe caminho entre v e w?
Iterable<Integer>	caminhoPara(int w)	obter caminho de v para w

API de Caminho

```
1  class Caminhos {
2    Grafo G; int origem;
3    boolean marcado[];//indica se foi visitado
4    int veioDe[];//indica por onde foi visitado
5
6    public Caminhos(Grafo G, int origem) {
7      this.origem = origem; this.G = G;
8      veioDe = new int[G.V()];
9      marcado = new boolean[G.V()];
10
11     dfs(G, origem);
12   }
13
14   //Obter o caminho pelo metodo de profundidade.
15   boolean dfs(Grafo G, int atual) {...}
16   //Vértice tem ligação à origem?
17   boolean temLigacao(int w) {...}
18   //qual é o caminho da origem para um vértice?
19   Iterable<Integer> caminhoPara(int w) {...}
20 }
```

Busca em profundidade: não recursivo

```
1  void dfs(Grafo G, int origem) {
2    Stack pilha = new Stack();// pilha substitui recursão
3    pilha.push(origem);
4
5    while (pilha.empty() == false) {
6      int atual = (int) pilha.pop();//pega mais recente
7      if (marcado[atual] == false) {
8          marcado[atual] = true;
9          for (int adjacente: G.adj(atual)) {
10             veioDe[adjacente] = atual; //marca caminho
11             pilha.push(adjacente);
12         }
13     }
14   }
15 }
```

Aplicação do DFS: consultas sobre conectividade

Definição

Vértices `v` e `w` são conectados se existe um caminho entre eles.

Meta

Pré-processar grafo para responder consultas da forma v é conectado a w? em tempo constante.

Definição

Um componente conectado é um conjunto maximal de vértices conectados

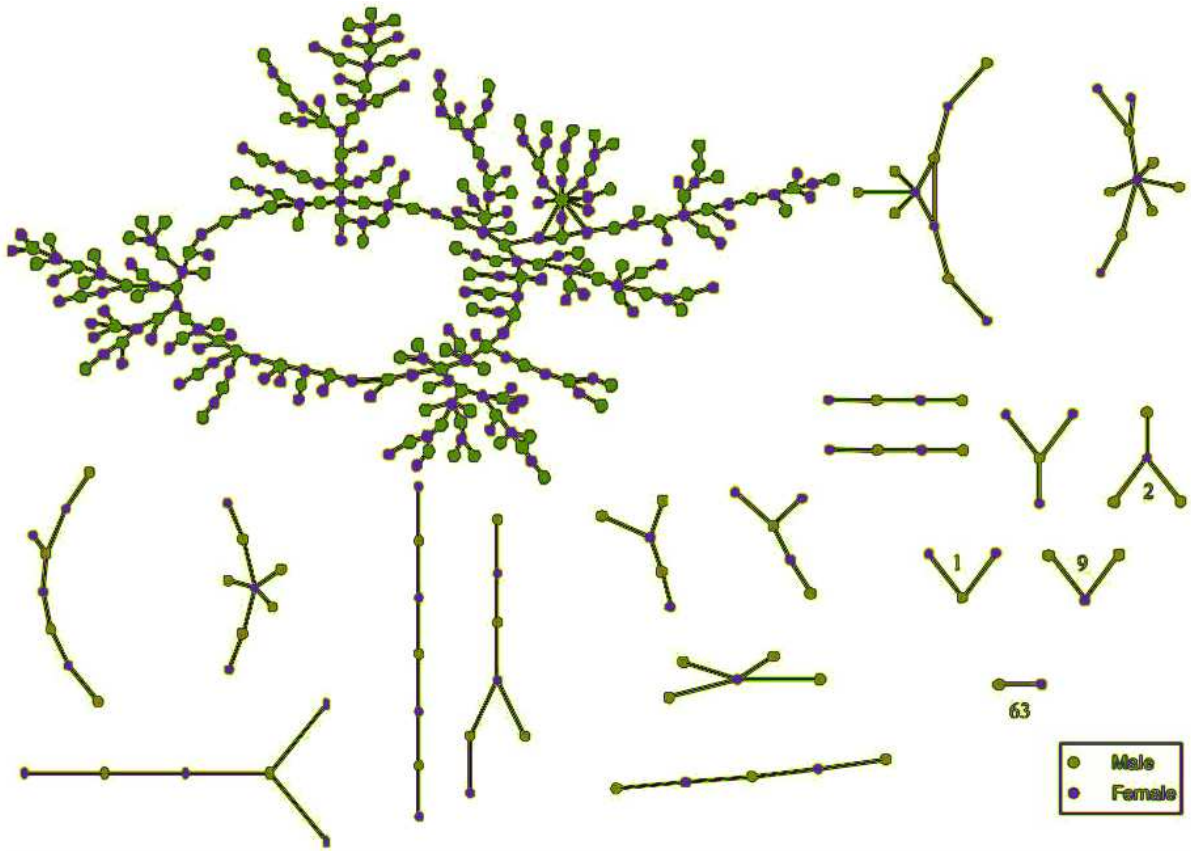


Figura: Peter Bearman, James Moody, and Katherine Stovel. Chains of affection: The structure of adolescent romantic and sexual networks. American Journal of Sociology, 110(1): 44-99, 2004.

Implementação para Componentes Conectados

```
1  class CompConectado {
2   boolean marcado[];
3   int id[];
4   int componentes;
5
6   public CompConectado(Grafo G) {
7     marcado = new boolean[G.V()];
8     id = new int[G.V()];
9
10    for (int v = 0; v < G.V(); v++) {
11      if (marcado[v] == false) {
12        dfs(G, v);
13        componentes++;
14      }
15    }
16  }
17
18  int count() {return componentes;}
19   int id(int v) {return id[v];}
20  void dfs(Grafo G, int v) {/* usar não recursivo */}
21 }
```

Busca em profundidade modificada

```
1  void dfs(Grafo G, int v) {
2   Stack pilha = new Stack();// pilha substitui recursão
3   pilha.push(origem);
4
5   while (pilha.empty() == false) {
6     int atual = (int) pilha.pop();//pega mais recente
7     if (marcado[atual] == false) {
8         marcado[atual] = true;
9         id[atual] = componentes; // MARCAR o vertice
10        for (int adjacente: G.adj(atual)) {
11            veioDe[adjacente] = atual; //marca caminho
12            pilha.push(adjacente);
13        }
14    }
15  }
16 }
```

Análise da Busca em Profundidade

- Análise de custos de `Caminhos`: bivariado em $|V|$ e $|E|$
- Custo de tempo
  - Quantos vértices são marcados?
  - Quanto custa operações de percorrer adjacentes?
  - Quanto custa operações de Stack?
- Custo de espaço

Algoritmo: UnionFind

- Problema: conjuntos disjuntos
  - Entrada: conjunto de arestas E
- Operações
  - Ligar u a v: `void uniao(int u, int v)`
  - `boolean existeCaminho(int u, int v)`

UnionFind: versão 1 – achar rápido

```
1  class UnionFindV1 {
2    int id[];
3
4    public UnionFindV1(int V) {
5      id = new int[V];
6      for (int i = 0; i < V; i++) id[i] = i;
7    }
8
9    boolean existeCaminho(int u, int v) {
10     return id[u]==id[v];
11   }
12
13   boolean uniao(int u, int v) {
14     int idU = id[u], idV = id[v];
15     for (int i = 0; i < id.length; i++)
16       if (id[i] == idU) id[i] = idV;
17   }
18 }
```

UnionFind: versão 1 – análise

- Quantos acessos ao array `id[]` são feitos para uma operação `uniao(u, v)`?
  - São $2V + 2$ acessos para cada união.
- Quantos acessos ao array `id[]` são feitos para uma operação `existeCaminho(u, v)`?
  - Operação de verificar se existe caminho é rápida: 2 acessos

UnionFind: versão 2 – árvores invertidas

```
1  class UnionFindV2 {
2    int id[]; // contém o id do pai na árvore invertida
3
4    public UnionFindV2(int V) {
5      id = new int[V];
6      for (int i = 0; i < V; i++) id[i] = i;
7    }
8
9    private int raiz(int u) {
10     while (u != id[u]) u = id[u]; // sobe na árvore
11     return u;
12   }
13
14   boolean existeCaminho(int u, int v) {
15     return raiz(u)==raiz(v);
16   }
17   boolean uniao(int u, int v) {
18     int rU = raiz(u), rV = raiz(v);
19     id[rU] = rV;// árvore de u é ligada à de v
20   }
21 }
```

UnionFind: versão 2 – análise

- Quantos acessos ao array `id[]` são feitos para uma operação `uniao(u, v)`?
  - Depende da altura de u e de v
- Quantos acessos ao array `id[]` são feitos para uma operação `existeCaminho(u, v)`?
  - Depende da altura de u e de v
- Como as árvores invertidas não são balanceadas: pior caso é N

UnionFind: versão 3 – árvores invertidas balanceadas

```
1  class UnionFindV3 {
2    int id[]; // contém o id do pai na árvore invertida
3    int tam[]; // tamanho da árvore
4
5    public UnionFindV3(int V) {// igual à V2 ...
6    private int raiz(int u)   {// igual à V2 ...
7    boolean existeCaminho(int u, int v){// igual à V2
8
9    // ligar raiz da menor arv. à maior
10   boolean uniao(int u, int v) {
11     int rU = raiz(u), rV = raiz(v);
12     if (tam[rU] < tam[rV])
13         { id[rU] = rV; tam[rV] += tam[rU]; }
14     else
15         { id[rV] = rU; tam[rU] += tam[rV]; }
16   }
17 }
```

UnionFind: versão 3 – análise

- Quantos acessos ao array `id[]` são feitos para uma operação `uniao(u, v)` ou `existeCaminho(u, v)`?
  - Depende da altura de u e de v
- Como árvore é dinamicamente balanceada, altura tende ser baixa
- Altura é no máximo $\lg N$ pois
  - Árvore de x é T_x
  - Altura de T_x aumenta quando sua árvore é conectada a outra T_y somente se $|T_y| \geq |T_x|$
  - Tamanho da árvore T_x dobra no máximo $\lfloor \lg N \rfloor$

UnionFind: versão 4 – árvores invertidas compactadas

```
1  class UnionFindV4 {
2    int id[]; // contém o id do pai na árvore invertida
3    int tam[]; // tamanho da árvore
4
5    public UnionFindV4(int V) {// igual à V3 ...
6    private int raiz(int u) {
7      int aux = u;
8      while (u != id[u]) {
9        // id[u] = id[id[u]]; //corta altura pela metade
10       u = id[u];
11     }
12     while (aux != id[aux]) {// compacta árvore
13       int tmp = aux;
14       aux = id[aux];
15       id[tmp] = u;
16     }
17     return u;
18   }
19   boolean existeCaminho(int u, int v){// igual à V3
20   boolean uniao(int u, int v) { // igual à V3
21 }
```

UnionFind: versão 4 – análise

- Quantos acessos ao array `id[]` são feitos para uma operação `uniao(u, v)` ou `existeCaminho(u, v)`?
  - Depende da altura de u e de v
- Como árvore é compactada, altura tende ser muito baixa
  - União custa $o(\lg^* N)$
    - $\lg^* N$ é número de vezes que lg é iterado em N antes de obter valor < 1

Árvore Geradora Mínima em Dígrafos Com Arestas Ponderadas

- Árvore Geradora Mínima (AGM) = Minimum Spanning Tree (MST)
- Arestas têm direção e peso
- Objetivo: obter árvore contendo todos os vértices do dígrafos com a menor soma de pesos das arestas

Algoritmo de Kruskal

- Algoritmo guloso:
  - Obtém melhor solução do problema completo a partir da melhor decisão a cada passo do algoritmo
- Algoritmo de Kruskal:
  - Fazer até todos vértices serem incluídos na árvore:
    - Adicionar próxima aresta com menor peso a menos que ela crie um laço

Algoritmo de Kruskal para AGM

```
1  class Kruskal {
2      Fila<Aresta> AGM = new Fila<Aresta>();
3
4      public Krukal(GrafoArestaPonderada G) {
5          MinHeap<Aresta> arvMin = new MinHeap<Aresta>();
6          for (Aresta a: G.arestas()) arvMin.insere(a);
7
8          UnionFind uf = new UnionFind(G.V());
9
10         while (!arvMin.vazia() && AGM.size() < G.V()-1)
               {
11           Aresta a = arvMin.removeMinimo();
12           int u = a.umLado(), v = a.outroLado();
13           if (!uf.conectados(u, v)) {
14             uf.unir(u, v);
15             AGM.enfilera(a);
16         }
17     }
18   }
19     public Iterable<Aresta> obterAGM() { return AGM; }
20 }
```

Algoritmo de Kruskal: análise

- Quanto custa para:
  - Manter arestas ordenadas?
    - Pode-se usar árvore `minHeap`: $O(\log |V|)$
  - Verificar se aresta pode criar laço na árvore sendo construída ?
    - Se usar DFS: $O(|V| + |E|)$
    - Se usar `unionFind`: $O(\log^* |V|)$

Extensão de relação de recorrência para grafos

- Grafos de dependência de subproblemas independentes
- Se subproblemas tiver ordenação topológica, então possível usar Programação Dinâmica
- Técnicas de Programação Dinâmica
  - Abordagem Top-Down vs. Bottom-up
  - Memoização
- Exemplos
  - Menor caminho em grafos
  - Distância de Edição
  - Parentização
  - Corte de Barras
  - Justificação de Textos
- Análise de Tempo vs. Espaço