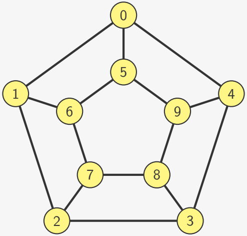
MC-202
Percurso em Grafos

Rafael C. S. Schouery
rafael@ic.unicamp.br

Universidade Estadual de Campinas

2º semestre/2023

Caminhos em Grafos



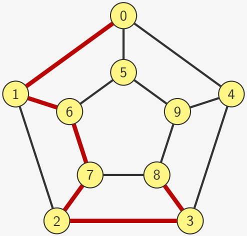
Um caminho de s para t em um grafo é:

- Uma sequência sem repetição de vértices vizinhos
- Começando em s e terminado em t

Por exemplo:

- $0, 1, 6, 7, 2, 3, 8$ é um caminho de 0 para 8
- $0, 5, 8$ não é um caminho de 0 para 8
- $0, 1, 2, 7, 6, 1, 2, 3, 8$ não é um caminho de 0 para 8

Caminhos em Grafos



Formalmente, um caminho de s para t em um grafo é:

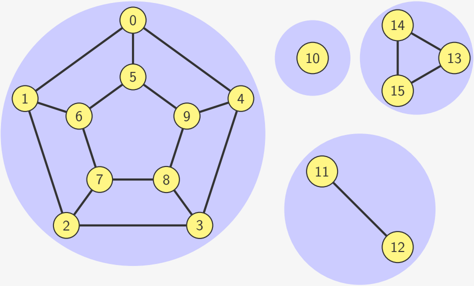
- Uma sequência de vértice $v_0, v_1, \ldots, v_k$ onde
- $v_0 = s$ e $v_k = t$
- $\{v_i, v_{i+1}\}$ é uma aresta para todo $0 \leq i \leq k-1$
- $v_i \neq v_j$ para todo $0 \leq i < j \leq k$

k é o comprimento do caminho

- $k = 0$ se e somente se $s = t$

Componentes Conexas

Um grafo pode ter várias “partes”



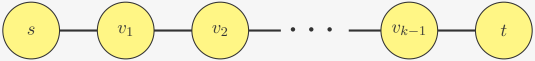
Chamamos essas partes de Componentes Conexas

- Um par de vértices está na mesma componente se e somente se existe caminho entre eles
  - Não há caminho entre vértices de componentes distintas
- Um grafo conexo tem apenas uma componente conexa

Existe caminho entre s e t?

Queremos saber se s e t estão na mesma componente conexa

Se estiverem, existe algum caminho de s até t



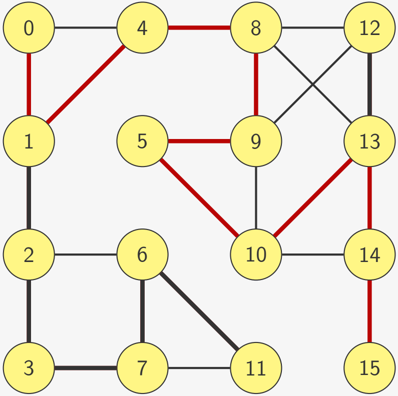
Se existe caminho e $s \neq t$, existe um segundo vértice v_1

- E v_1 é vizinho de s
- Então, ou $v_1 = t$, ou existe um terceiro vértice v_2
  - E v_2 é vizinho de v_1
- E assim por diante...

A dificuldade é acertar qual vizinho v_1 de s devemos usar...

- Solução: testar todos!

Exemplo — Existe caminho de 0 até 15?



Essa é uma busca em profundidade:

- Vá o máximo possível em uma direção
- Se não encontrarmos o vértice, volte o mínimo possível
- E pegue um novo caminho por um vértice não visitado

Implementação (com Matriz de Adjacências)

```c
int existe_caminho(p_grafo g, int s, int t) {
  int encontrou, i, *visitado = malloc(g->n * sizeof(int));
  for (i = 0; i < g->n; i++)
    visitado[i] = 0;
  encontrou = busca_rec(g, visitado, s, t);
  free(visitado);
  return encontrou;
}
```

```c
int busca_rec(p_grafo g, int *visitado, int v, int t) {
  int w;
  if (v == t)
    return 1; /*sempre existe caminho de t para t*/
  visitado[v] = 1;
  for (w = 0; w < g->n; w++)
    if (g->adj[v][w] && !visitado[w])
      if (busca_rec(g, visitado, w, t))
        return 1;
  return 0;
}
```

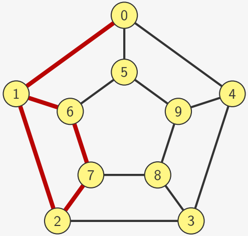
E se quisermos saber quais são as componentes conexas?

Componentes Conexas (Listas de Adjacência)

```c
int * encontra_componentes(p_grafo g) {
  int s, c = 0, *componentes = malloc(g->n * sizeof(int));
  for (s = 0; s < g->n; s++)
    componentes[s] = -1;
  for (s = 0; s < g->n; s++)
    if (componentes[s] == -1) {
      visita_rec(g, componentes, c, s);
      c++;
    }
  return componentes;
}
```

```c
void visita_rec(p_grafo g, int *componentes, int comp, int v) {
  p_no t;
  componentes[v] = comp;
  for (t = g->adj[v]; t != NULL; t = t->prox)
    if (componentes[t->v] == -1)
      visita_rec(g, componentes, comp, t->v);
}
```

Ciclos em Grafos



Um ciclo em um grafo é:

- Uma sequência de vértices vizinhos sem repetição exceto pelo primeiro e o último vértice que são idênticos

Por exemplo:

- $5, 6, 7, 8, 9, 5$ é um ciclo
- $1, 2, 3$ não é um ciclo
- $1, 2, 7, 6, 1$ é um ciclo
- $1, 2, 7, 6, 1, 0$ não é um ciclo (mas contém um ciclo)

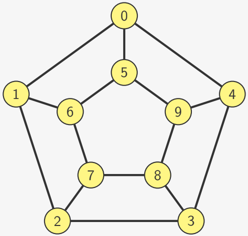
Árvores, Florestas e Subgrafos

Uma árvore é um grafo conexo acíclico

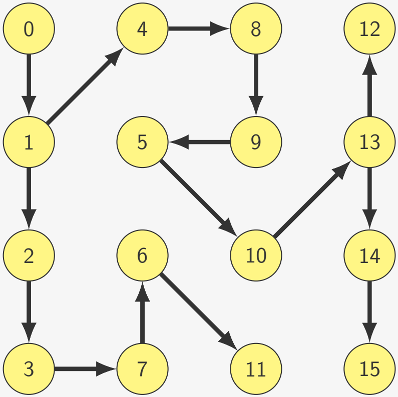
- Uma floresta é um grafo acíclico
- Suas componentes conexas são árvores

Um subgrafo é um grafo obtido a partir da remoção de vértices e arestas

- Podemos considerar também árvores/florestas que são subgrafos de um grafo dado



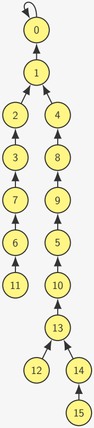
Caminhos de s para outros vértices da componente



As arestas usadas formam uma árvore!

- Essa árvore dá um caminho de qualquer vértice até a raiz
- Basta ir subindo na árvore

Caminhos de s para outros vértices da componente



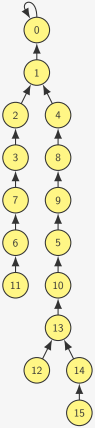

```c
int * encontra_caminhos(p_grafo g, int s) {
  int i, *pai = malloc(g->n * sizeof(int));
  for (i = 0; i < g->n; i++)
    pai[i] = -1;
  busca_em_profundidade(g, pai, s, s);
  return pai;
}
```

```c
void busca_em_profundidade(p_grafo g, int *pai, int p, int v) {
  p_no t;
  pai[v] = p;
  for(t = g->adj[v]; t != NULL; t = t->prox)
    if (pai[t->v] == -1)
      busca_em_profundidade(g, pai, v, t->v);
}
```

Imprimindo o caminho

```c
void imprimi_caminho_reverso(int v, int *pai) {
  printf("%d", v);
  if(pai[v] != v)
    imprimi_caminho_reverso(pai[v], pai);
}
```

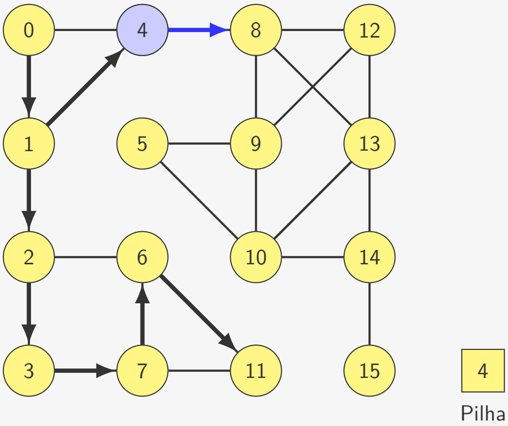
```c
void imprimi_caminho(int v, int *pai) {
  if(pai[v] != v)
    imprimi_caminho(pai[v], pai);
  printf("%d", v);
}
```



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

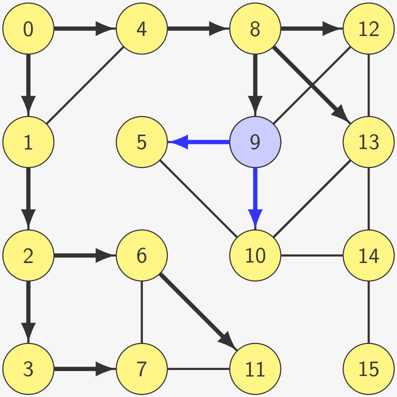
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Implementação

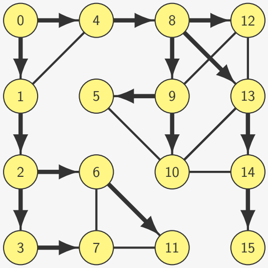
```c
int * busca_em_profundidade(p_grafo g, int s) {
  int w, v;
  int *pai = malloc(g->n * sizeof(int));
  int *visitado = malloc(g->n * sizeof(int));
  p_pilha p = criar_pilha();
  for (v = 0; v < g->n; v++) {
    pai[v] = -1;
    visitado[v] = 0;
  }
  empilhar(p,s);
  pai[s] = s;
  while(!pilha_vazia(p)) {
    v = desempilhar(p);
    visitado[v] = 1;
    for (w = 0; w < g->n; w++)
      if (g->adj[v][w] && !visitado[w]) {
        pai[w] = v;
        empilhar(p, w);
      }
  }
  destroi_pilha(p);
  free(visitado);
  return pai;
}
```

E se tivéssemos usado uma Fila?



Fila	12	13	7	11	5

E se tivéssemos usado uma Fila?



Usando uma fila, visitamos primeiro os vértices mais próximos

- Enfileiramos os vizinhos de 0 (que estão a distância 1)
- Desenfileiramos um de seus vizinhos
- E enfileiramos os vizinhos deste vértice
  - que estão a distância 2 de 0
- Assim por diante...

A árvore nos dá um caminho mínimo entre raiz e vértice

Implementação da Busca em Largura

```c
int * busca_em_largura(p_grafo g, int s) {
  int w, v;
  int *pai = malloc(g->n * sizeof(int));
  int *visitado = malloc(g->n * sizeof(int));
  p_fila f = criar_fila();
  for (v = 0; v < g->n; v++) {
    pai[v] = -1;
    visitado[v] = 0;
  }
  enfileira(f,s);
  pai[s] = s;
  visitado[s] = 1;
  while(!fila_vazia(f)) {
    v = desenfileira(f);
    for (w = 0; w < g->n; w++)
      if (g->adj[v][w] && !visitado[w]) {
        visitado[w] = 1;/*evita repetição na fila*/
        pai[w] = v;
        enfileira(f, w);
      }
  }
  destroi_fila(f);
  free(visitado);
  return pai;
}
```

Tempo para fazer a busca

Quanto tempo demora para fazer uma busca?

- em profundidade ou em largura
- em um grafo com n vértices e m arestas

Suponha que inserir e remover de pilha/fila leva $\mathrm{O}(1)$

- Podemos usar vetores ou listas ligadas

A busca percorre todos os vértices

- E empilha/enfileira seus vizinhos não visitados
- Se usarmos uma Matriz de Adjacências, leva $\mathrm{O}(n^2)$

E se usarmos Listas de Adjacência?

- Cada aresta é analisada apenas duas vezes
- Gastamos tempo $\mathrm{O}(\max\{n, m\}) = \mathrm{O}(n + m)$
  - Linear no tamanho do grafo