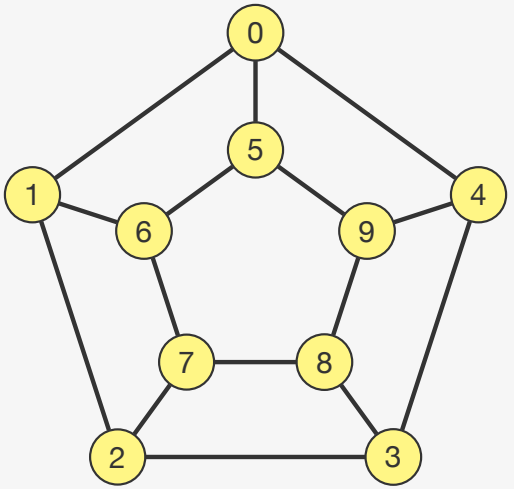
MC-202
Percurso em Grafos

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

Universidade Estadual de Campinas

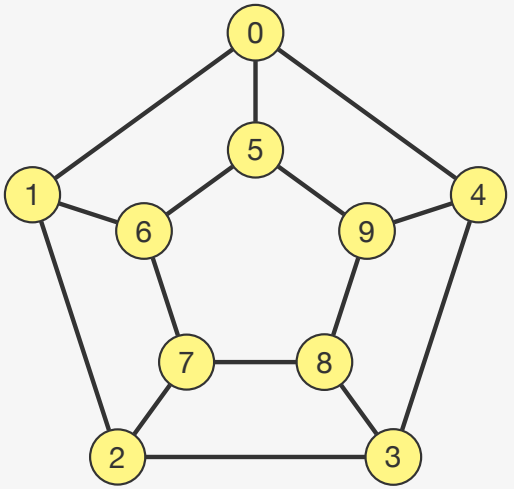
2º semestre/2018

Caminhos em Grafos



Um caminho de s para t em um grafo é:

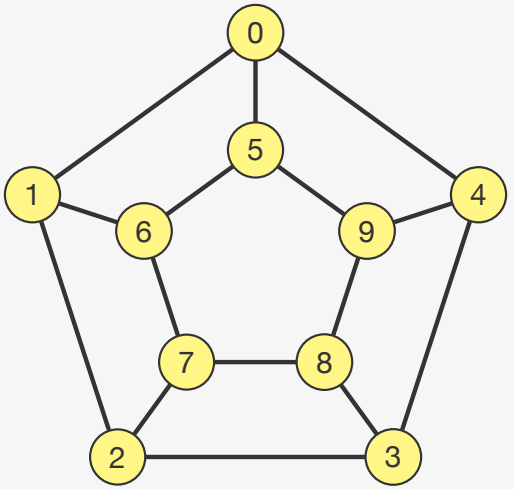
Caminhos em Grafos



Um caminho de s para t em um grafo é:

- Uma sequência sem repetição de vértices vizinhos

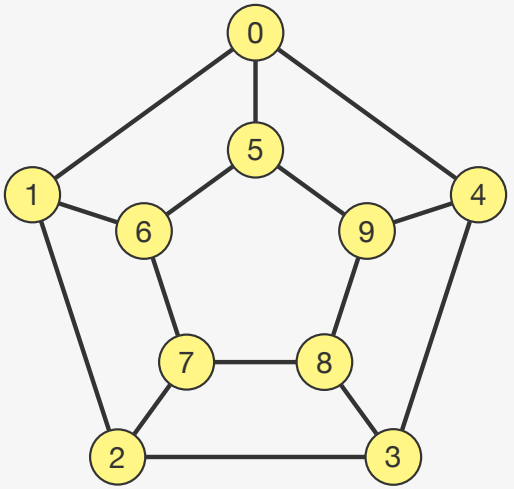
Caminhos em Grafos



Um caminho de s para t em um grafo é:

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

Caminhos em Grafos

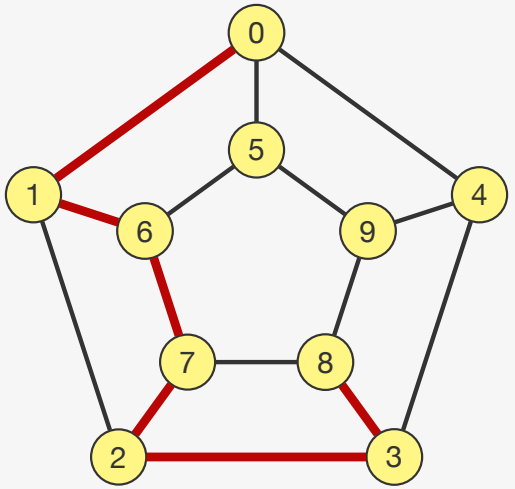


Um caminho de s para t em um grafo é:

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

Por exemplo:

Caminhos em Grafos



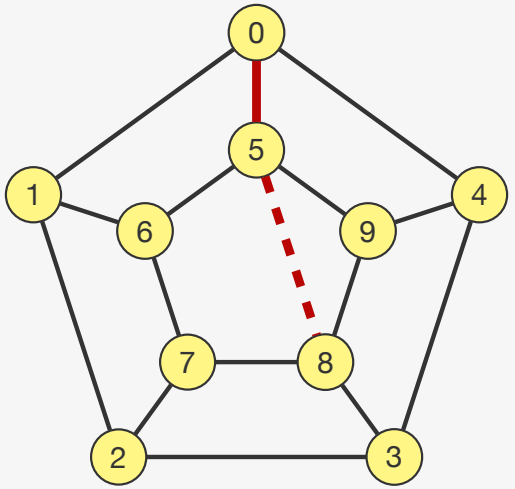
Um caminho de s para t em um grafo é:

- 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

Caminhos em Grafos



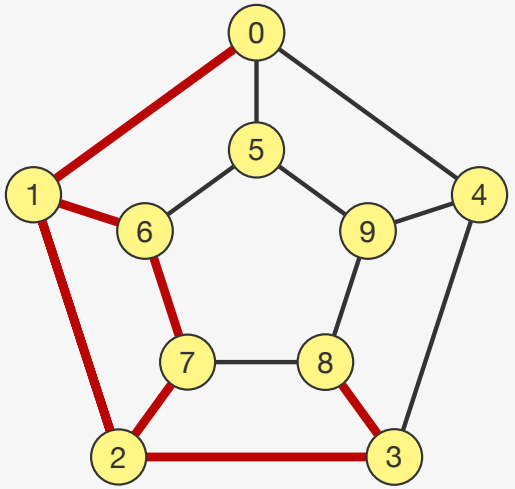
Um caminho de s para t em um grafo é:

- 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

Caminhos em Grafos



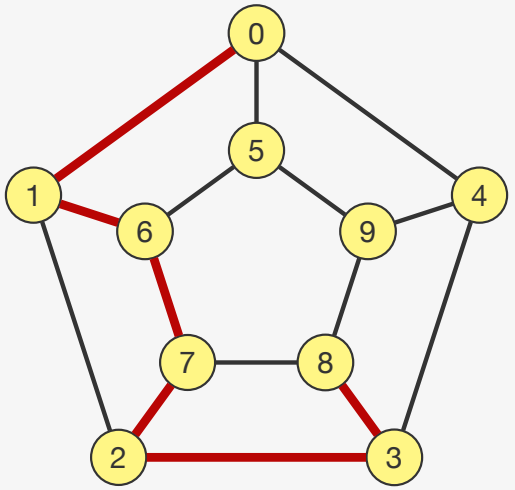
Um caminho de s para t em um grafo é:

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

Por exemplo:

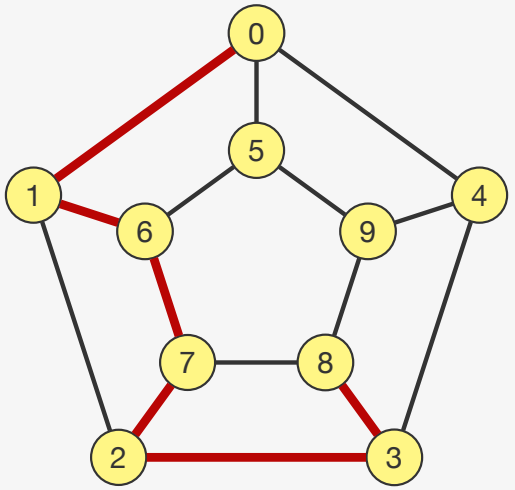
- $0, 1, 6, 7, 2, 3, 8$ é um caminho de 0 para 8
- $0, 5, 8$ não é um caminho de 0 para 8
- $0, 1, 2, 7, 6, 1, 2, 3, 8$ não é um caminho de 0 para 8

Caminhos em Grafos



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

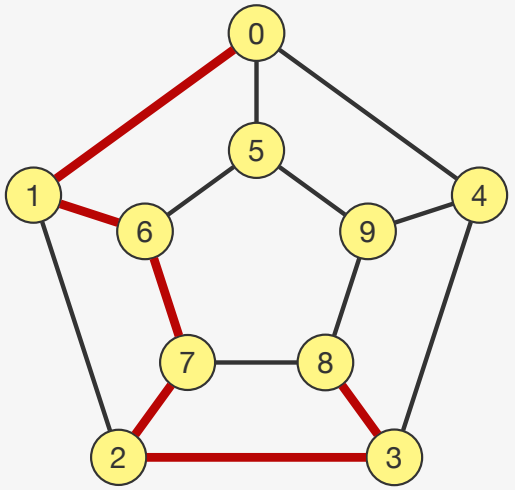
Caminhos em Grafos



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

- Uma sequência de vértice $v_0, v_1, \ldots, v_k$ onde

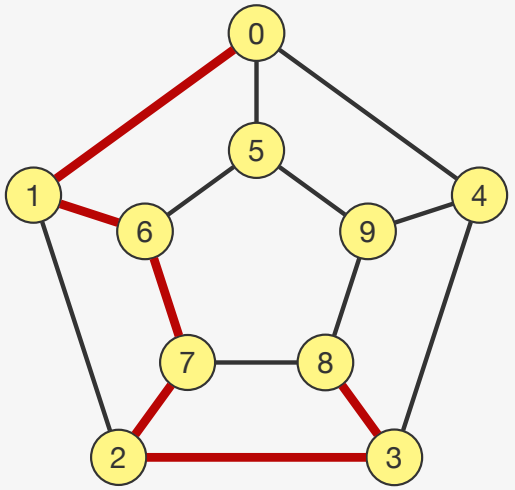
Caminhos em Grafos



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 $v_k = t$

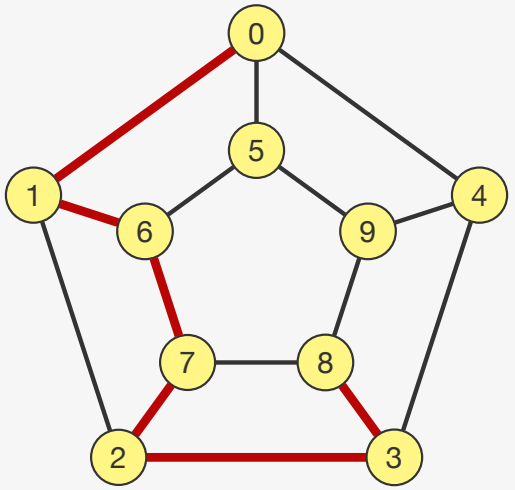
Caminhos em Grafos



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 $0 \leq i \leq k-1$

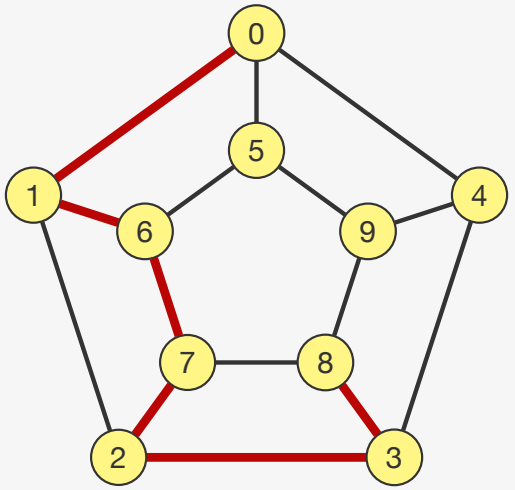
Caminhos em Grafos



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 $0 \leq i < j \leq k$

Caminhos em Grafos

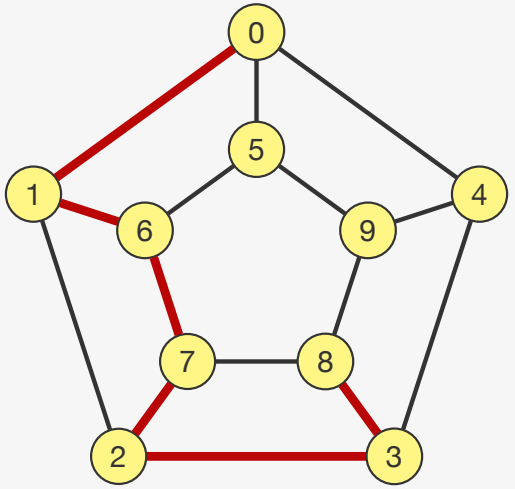


Formalmente, um caminho de s para t em um grafo é:

- 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

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

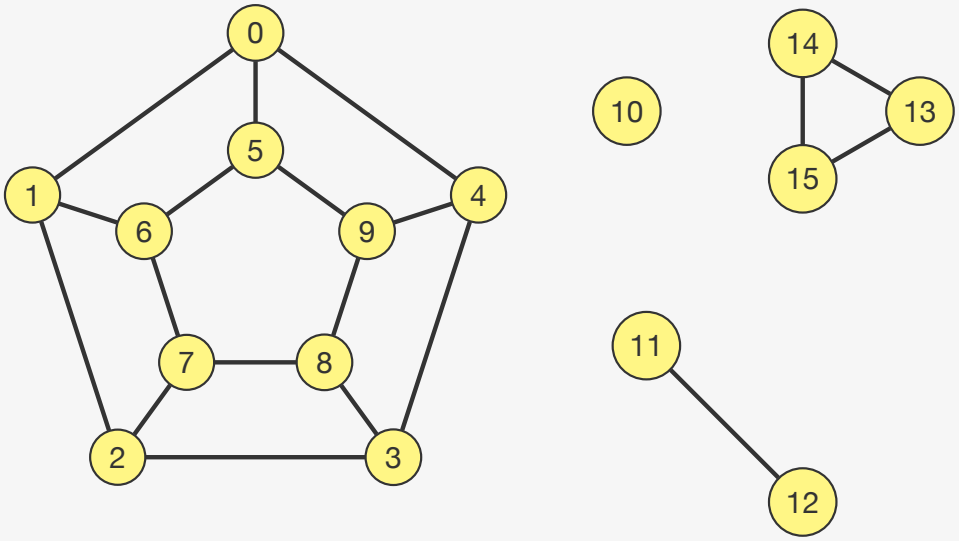
- $k = 0$ se e somente se $s = t$

Componentes Conexas

Um grafo pode ter várias “partes”

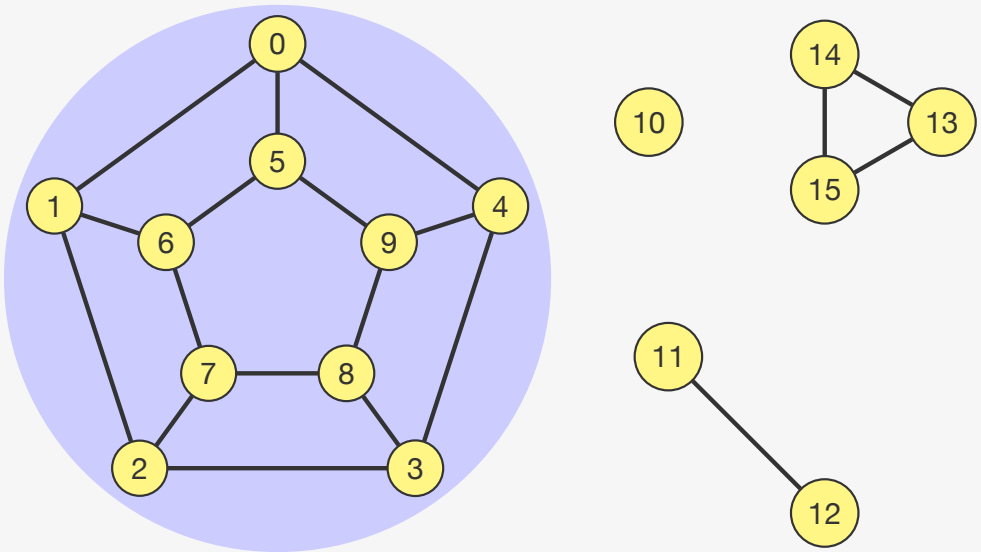
Componentes Conexas

Um grafo pode ter várias “partes”



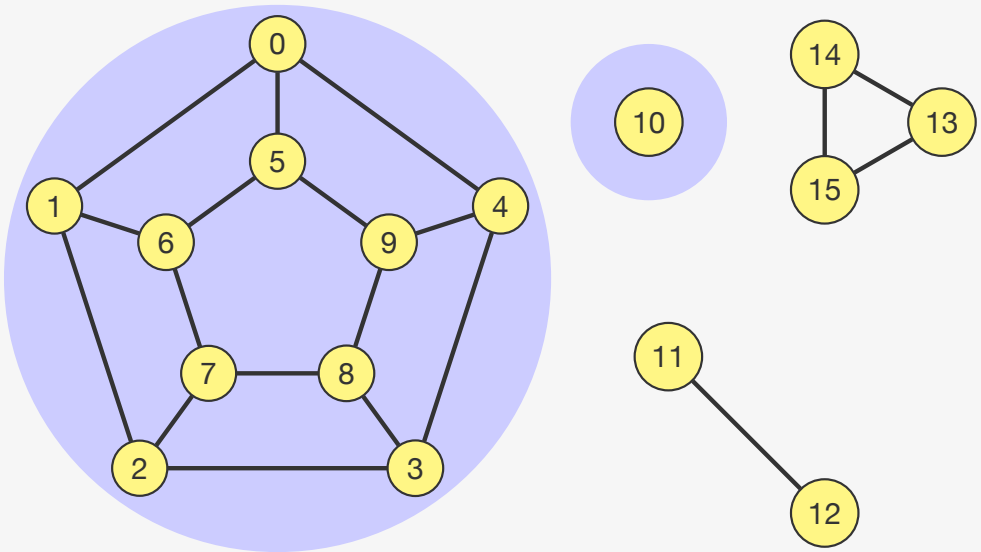
Componentes Conexas

Um grafo pode ter várias “partes”



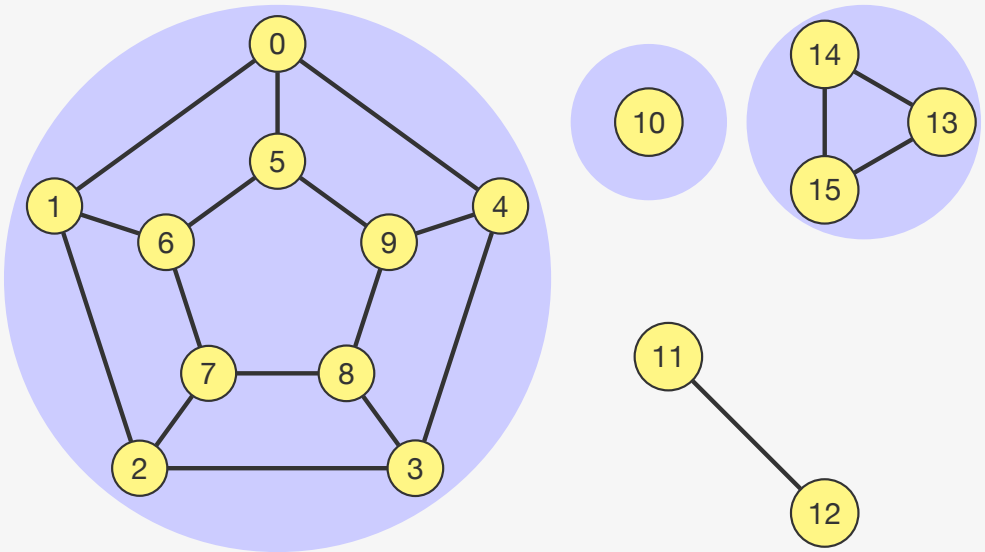
Componentes Conexas

Um grafo pode ter várias “partes”



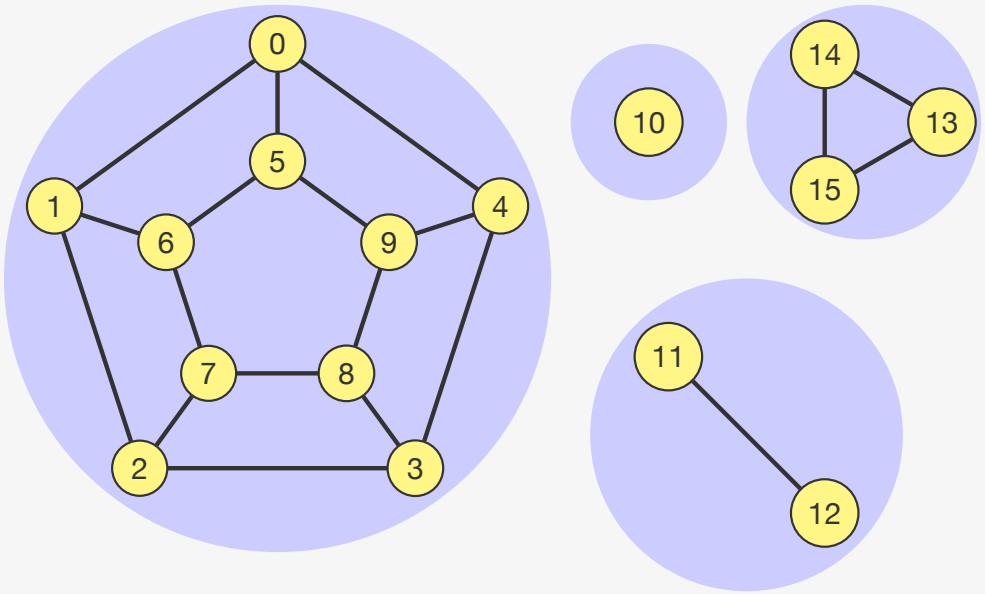
Componentes Conexas

Um grafo pode ter várias “partes”



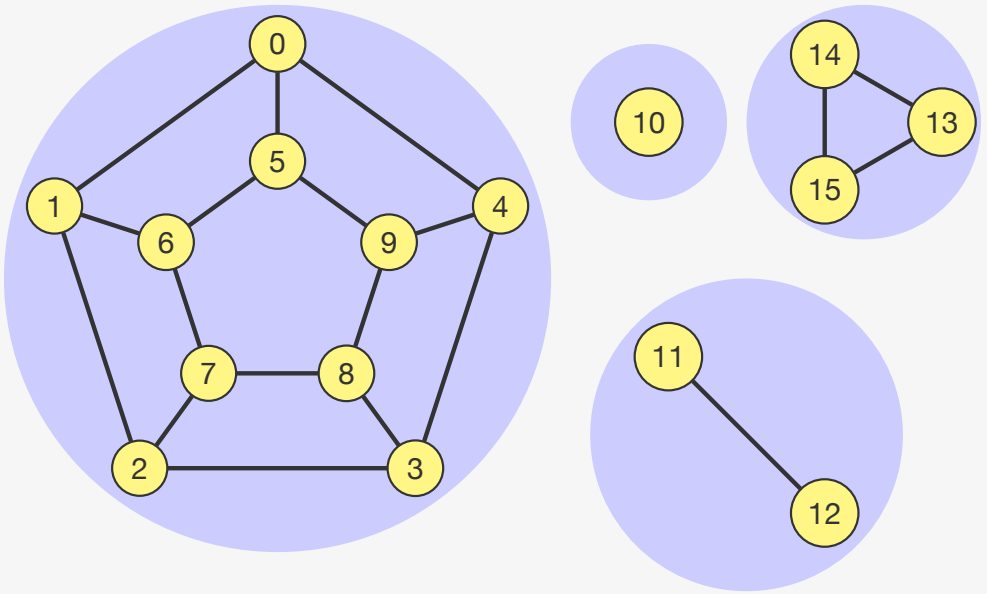
Componentes Conexas

Um grafo pode ter várias “partes”



Componentes Conexas

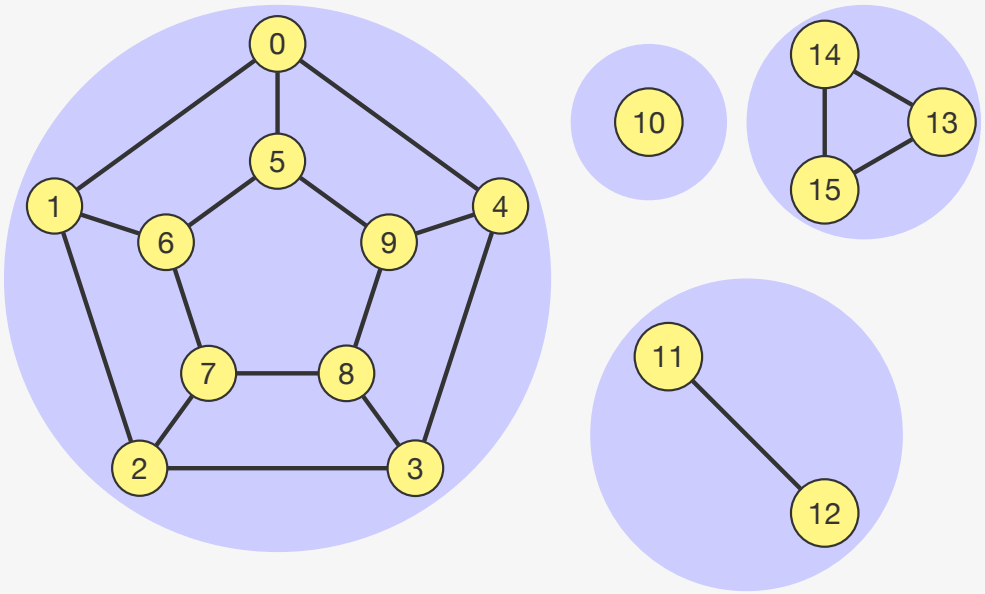
Um grafo pode ter várias “partes”



Chamamos essas partes de Componentes Conexas

Componentes Conexas

Um grafo pode ter várias “partes”

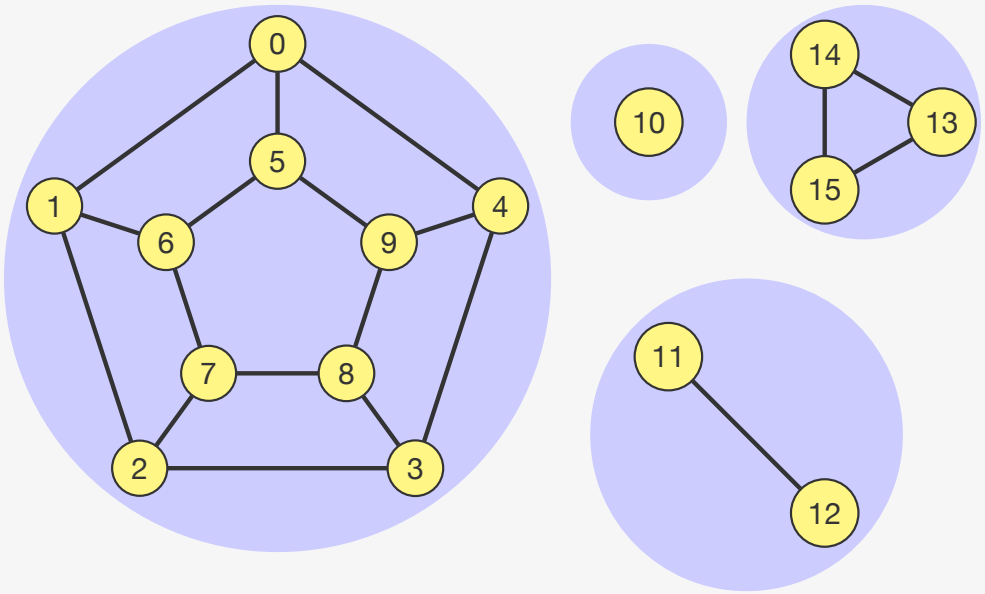


Chamamos essas partes de Componentes Conexas

- Um par de vértices está na mesma componente se e somente se existe caminho entre eles

Componentes Conexas

Um grafo pode ter várias “partes”

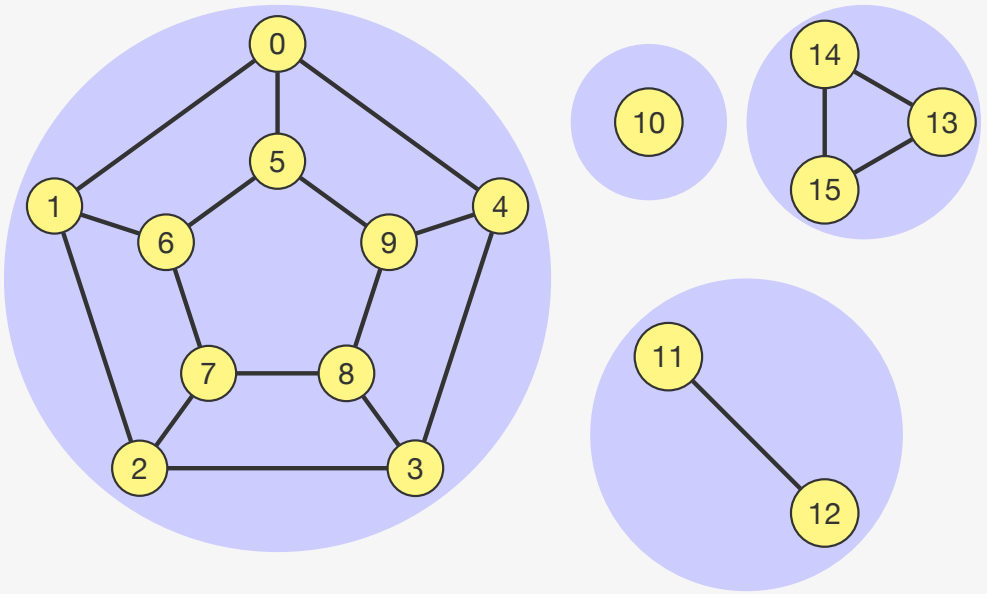


Chamamos essas partes de Componentes Conexas

- Um par de vértices está na mesma componente se e somente se existe caminho entre eles
  - Não há caminho entre vértices de componentes distintas

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

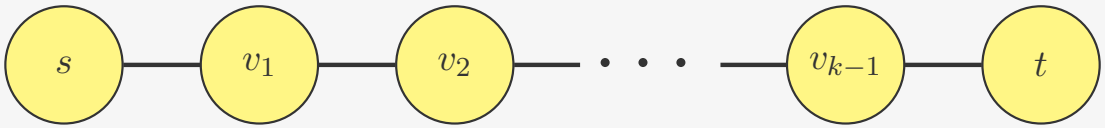
Existe caminho entre s e t?

Queremos saber se s e t estão na mesma componente conexa

Existe caminho entre s e t?

Queremos saber se s e t estão na mesma componente conexa

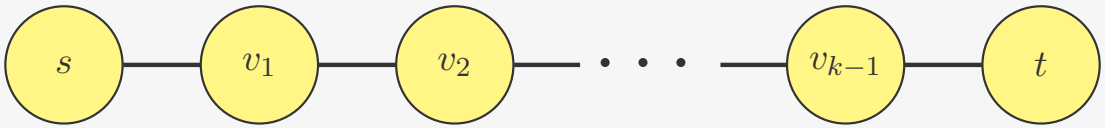
Se estiverem, existe algum caminho de s até t



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

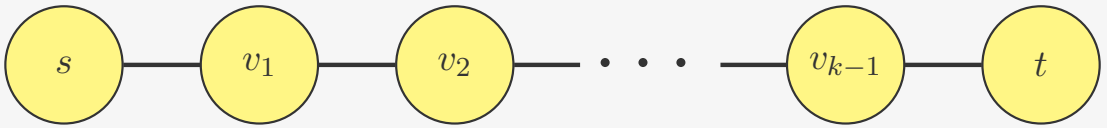


Se existe caminho e $s \neq t$, existe um segundo vértice v_1

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



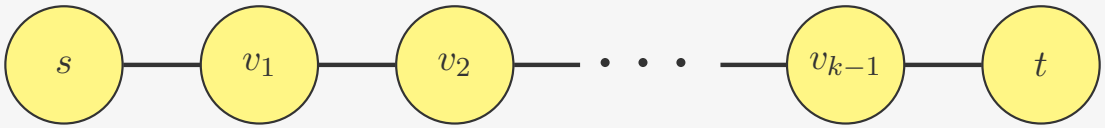
Se existe caminho e $s \neq t$, existe um segundo vértice v_1

- E v_1 é vizinho de s

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



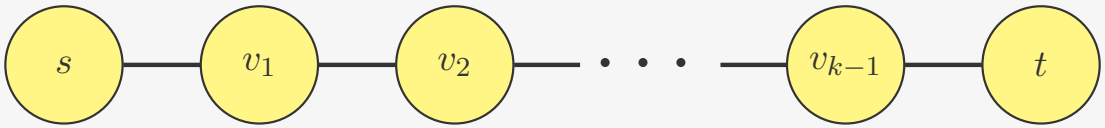
Se existe caminho e $s \neq t$, existe um segundo vértice v_1

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

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



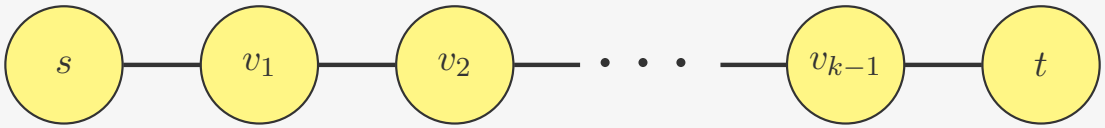
Se existe caminho e $s \neq t$, existe um segundo vértice v_1

- 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

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



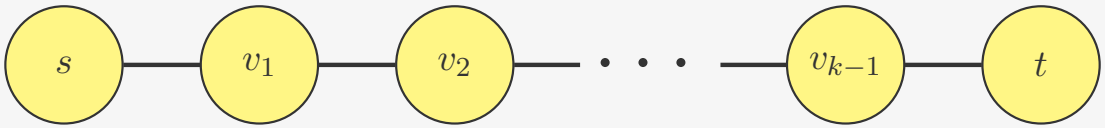
Se existe caminho e $s \neq t$, existe um segundo vértice v_1

- 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...

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

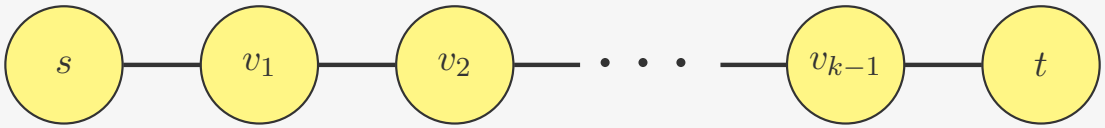
- E v_1 é vizinho de s
- Então, ou $v_1 = t$, ou existe um terceiro vértice v_2
  - E v_2 é vizinho de v_1
- E assim por diante...

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

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



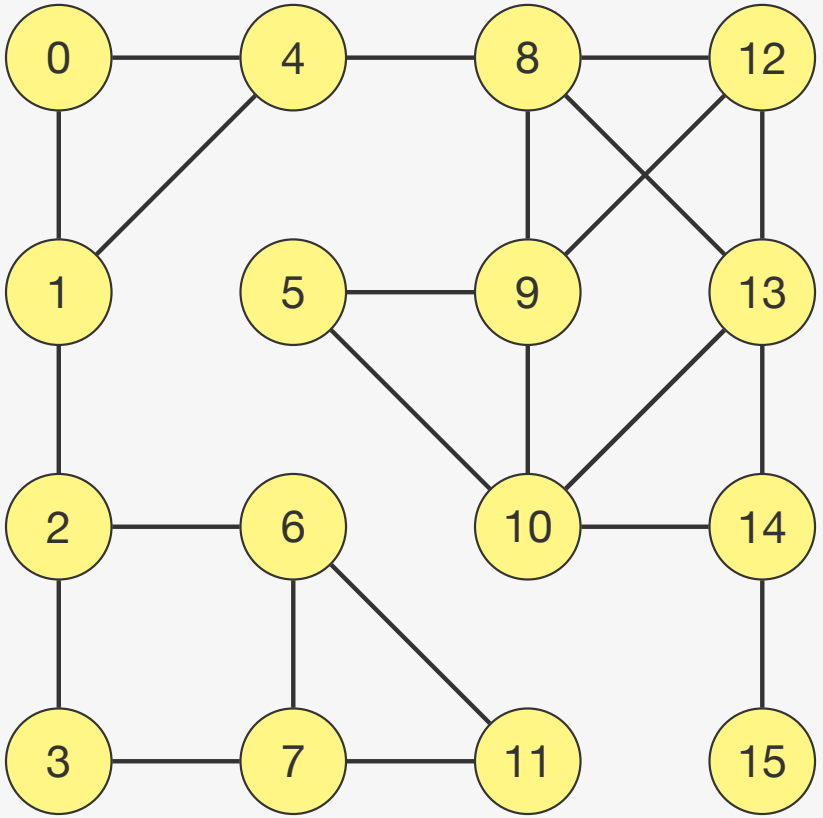
Se existe caminho e $s \neq t$, existe um segundo vértice v_1

- 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...

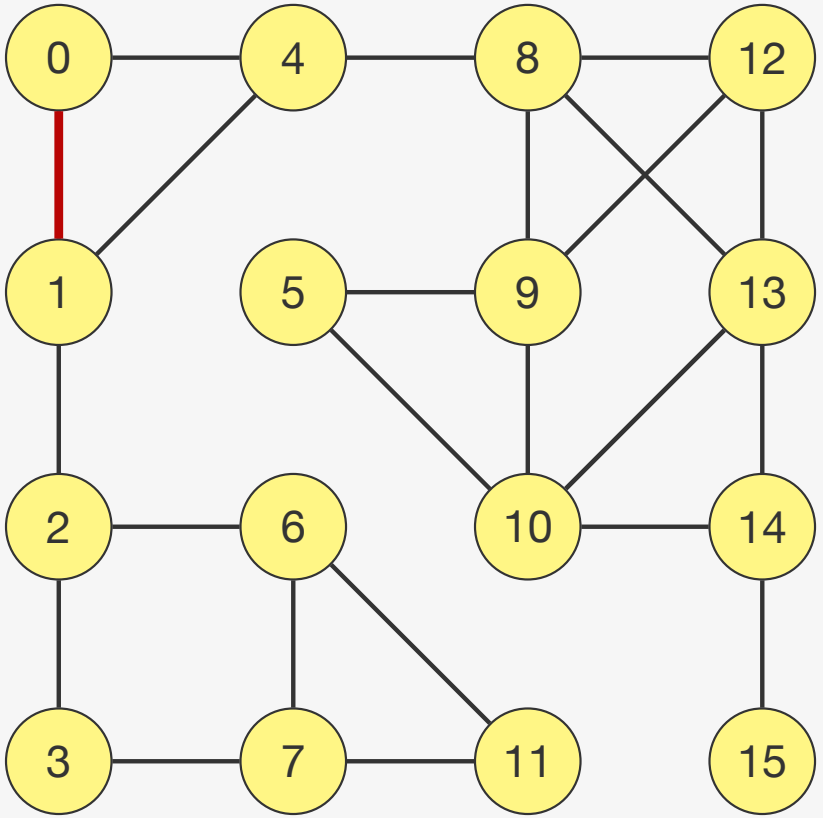
A dificuldade é acertar qual vizinho v_1 de s devemos usar...

- Solução: testar todos!

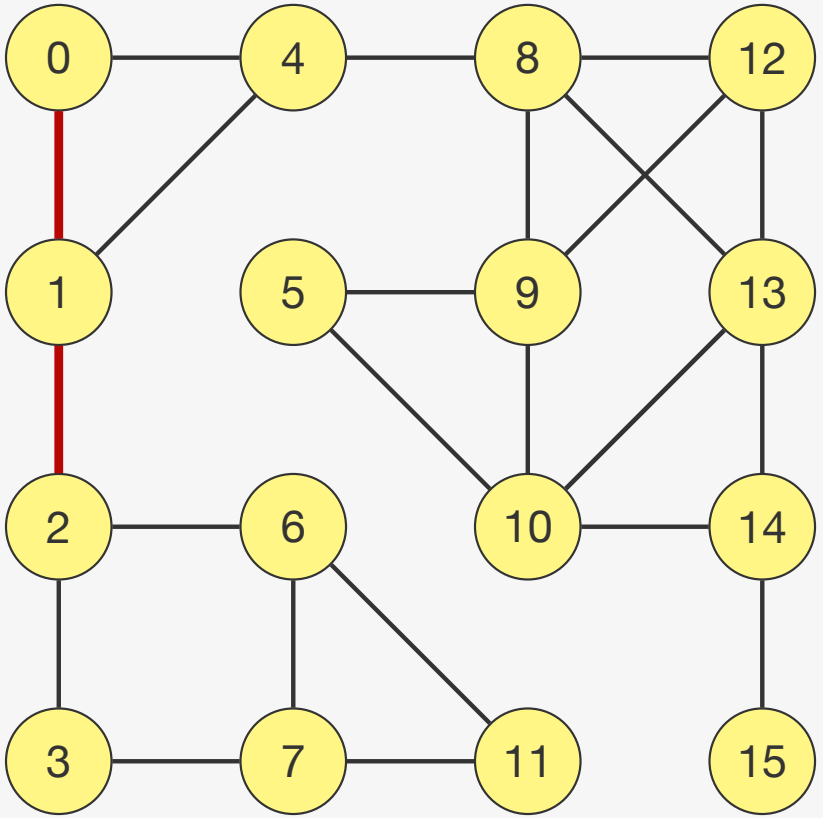
Exemplo - Existe caminho de 0 até 15?



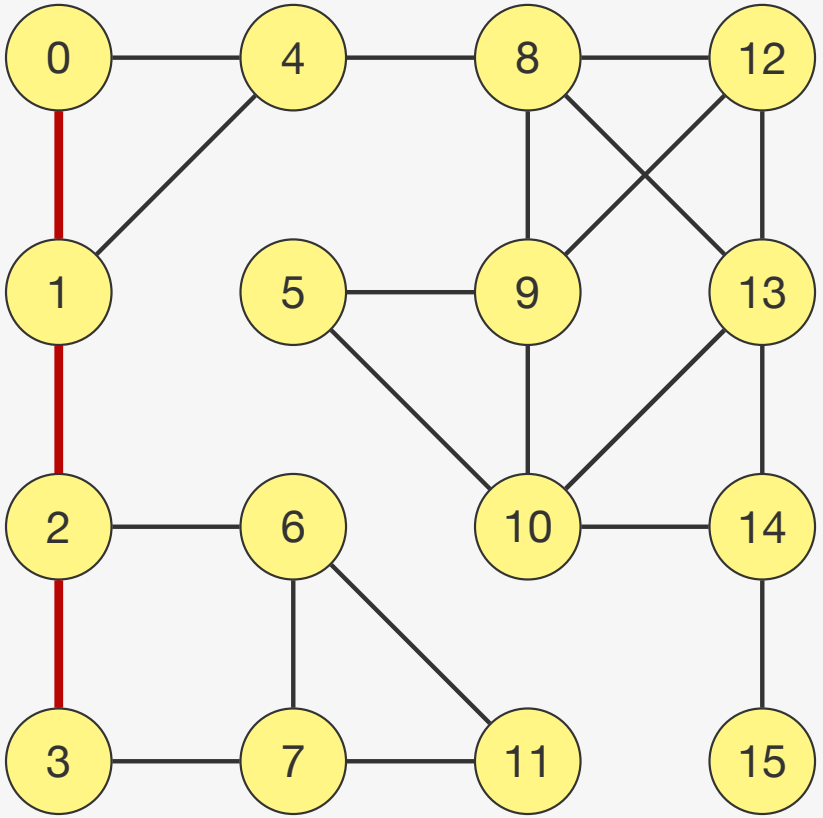
Exemplo - Existe caminho de 0 até 15?



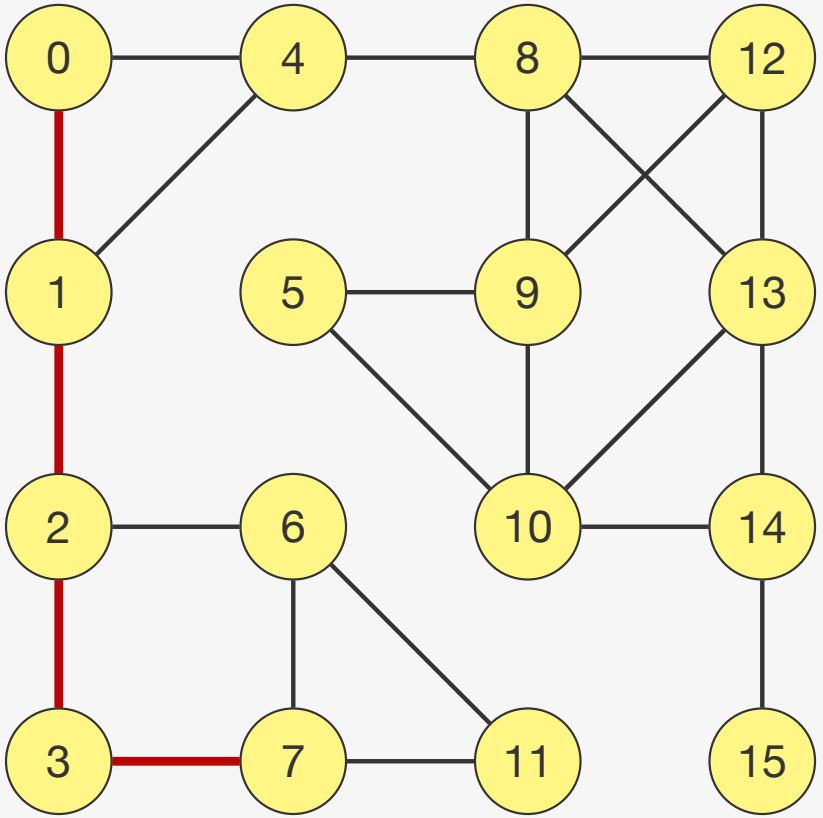
Exemplo - Existe caminho de 0 até 15?



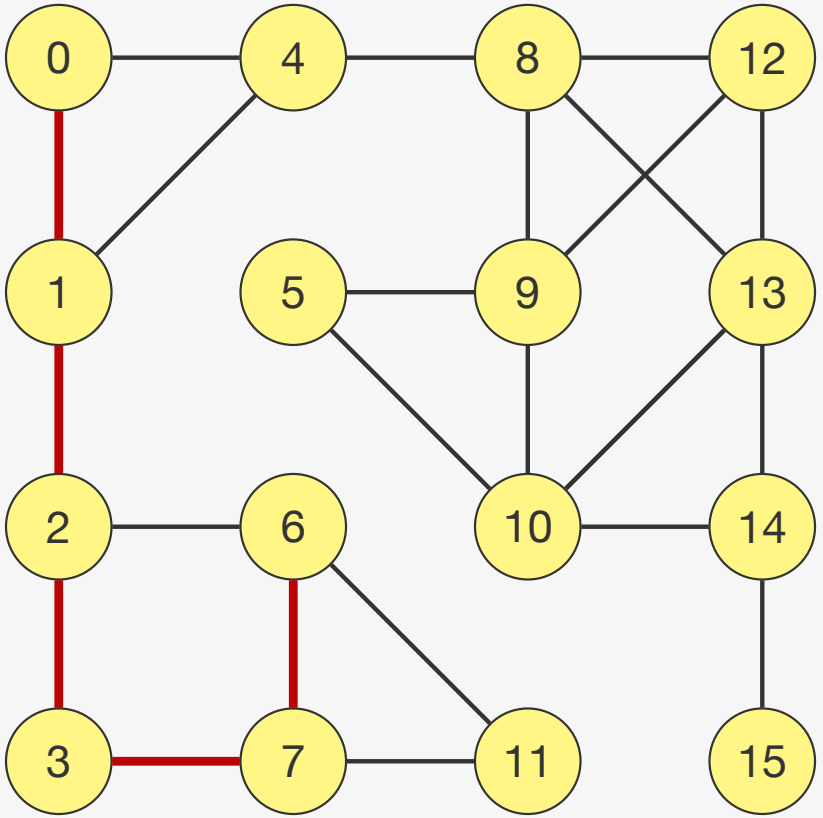
Exemplo - Existe caminho de 0 até 15?



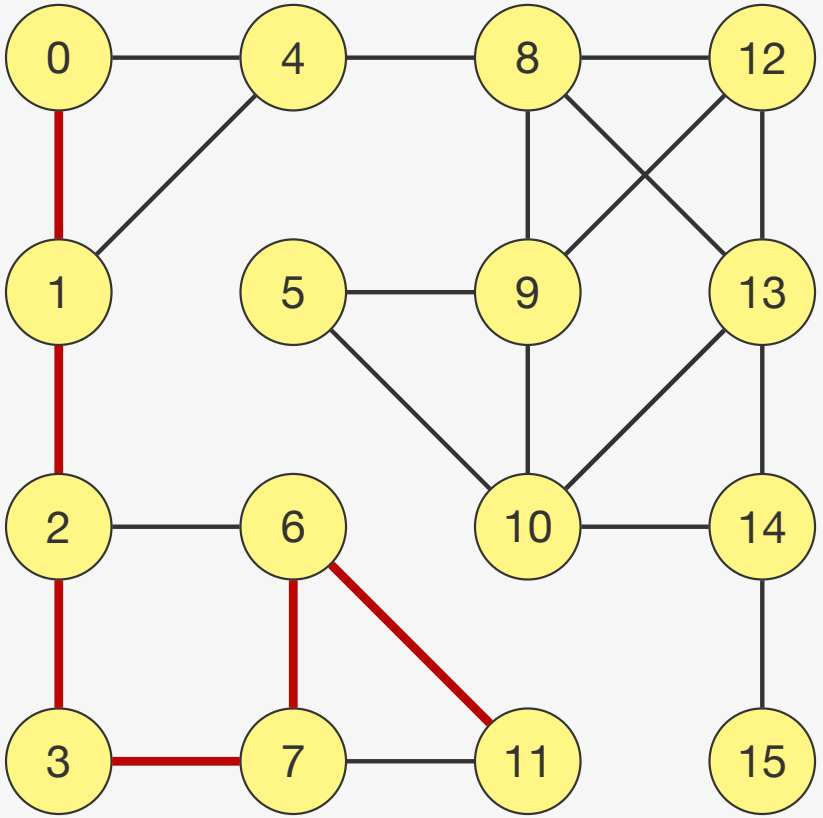
Exemplo - Existe caminho de 0 até 15?



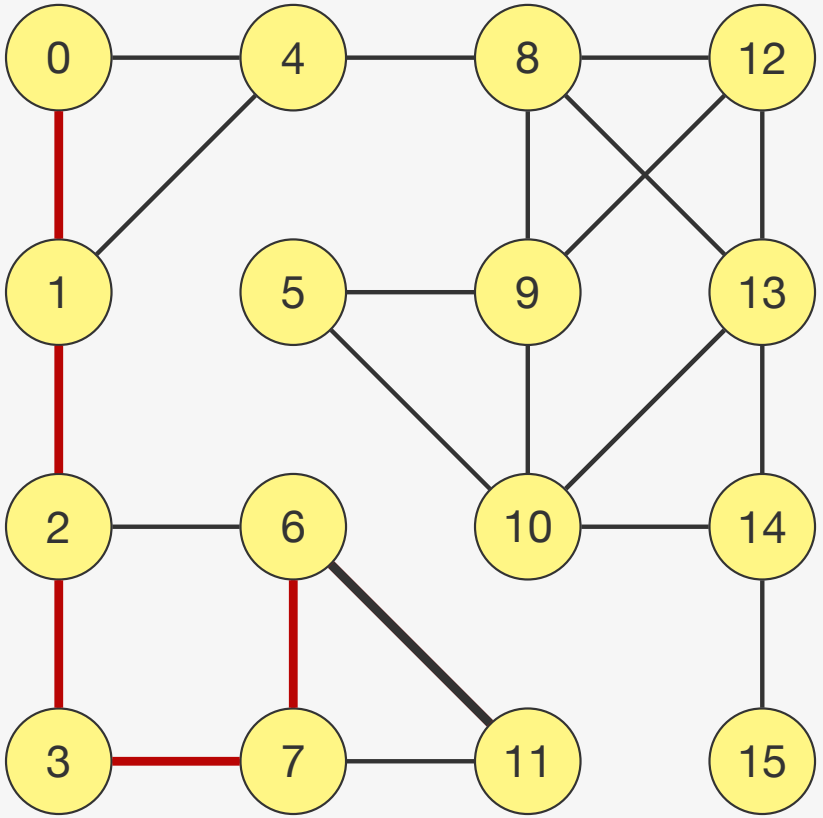
Exemplo - Existe caminho de 0 até 15?



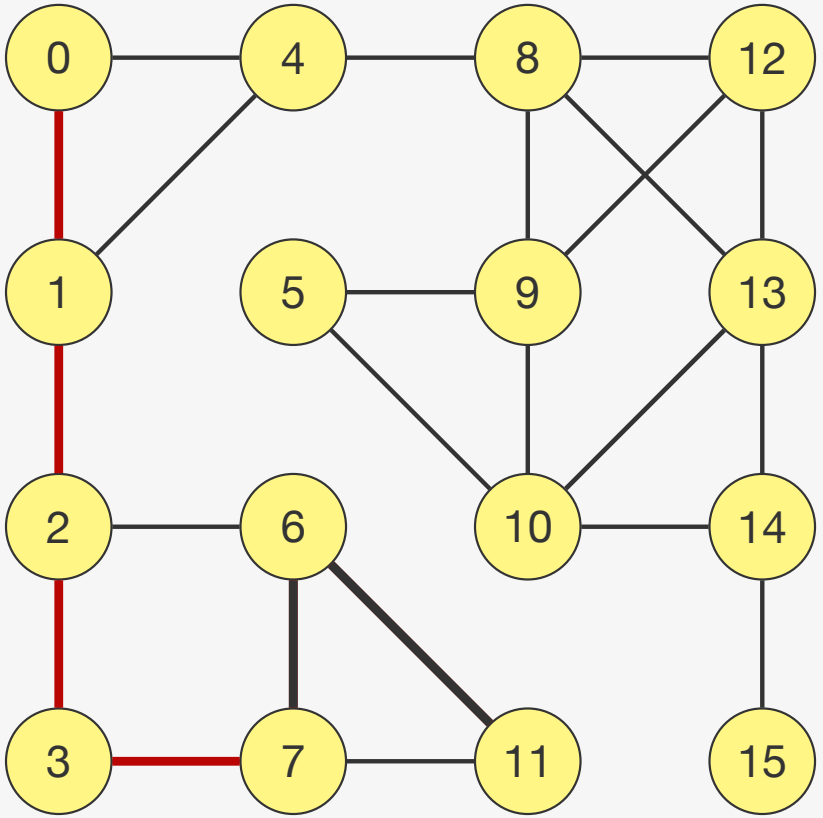
Exemplo - Existe caminho de 0 até 15?



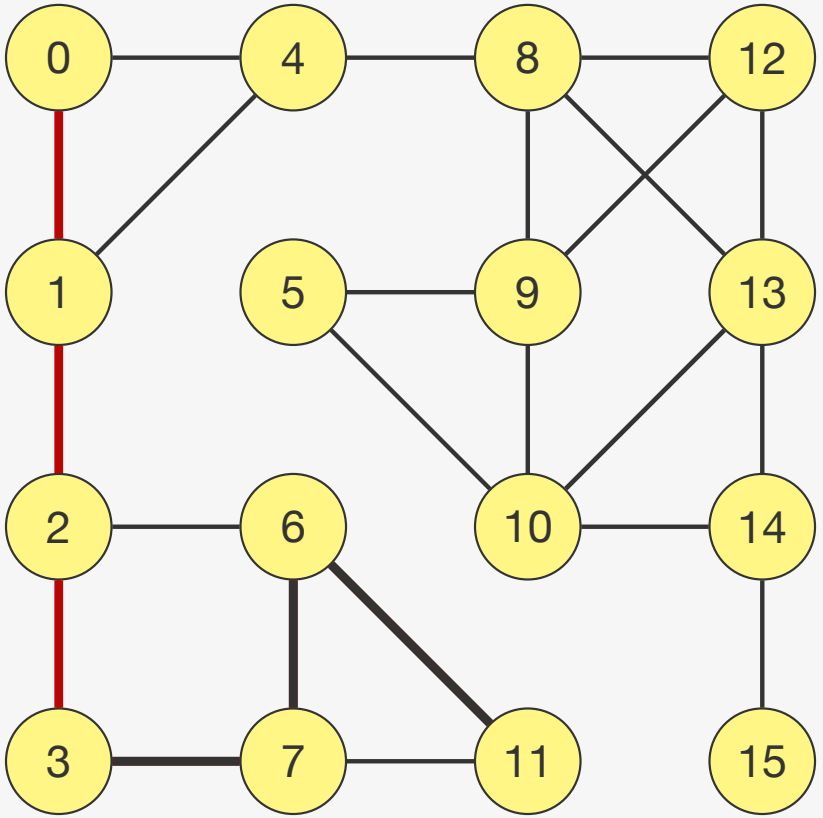
Exemplo - Existe caminho de 0 até 15?



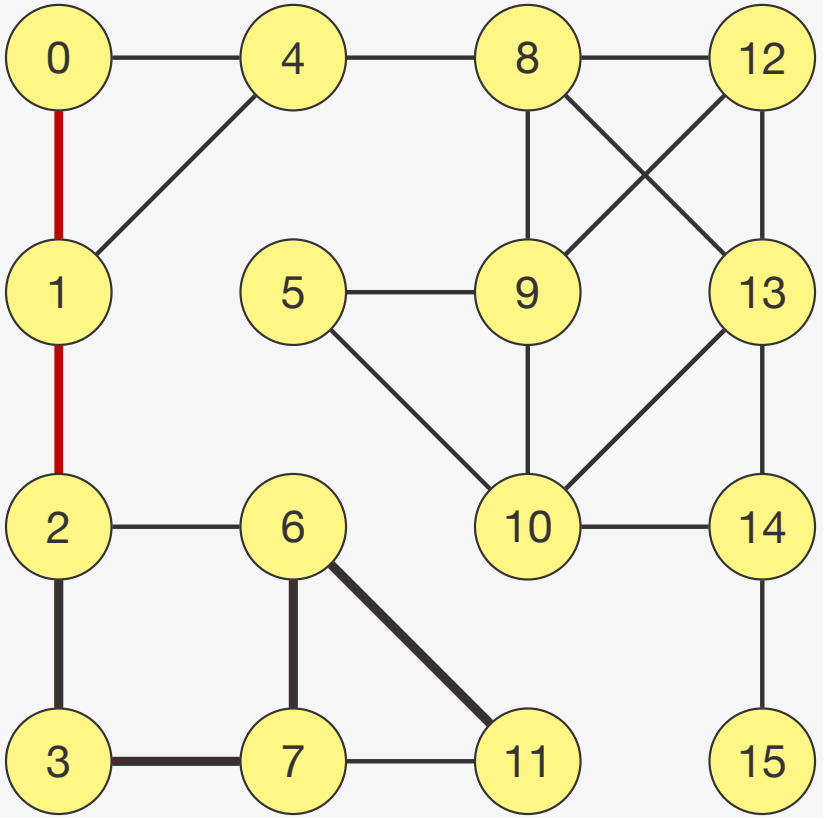
Exemplo - Existe caminho de 0 até 15?



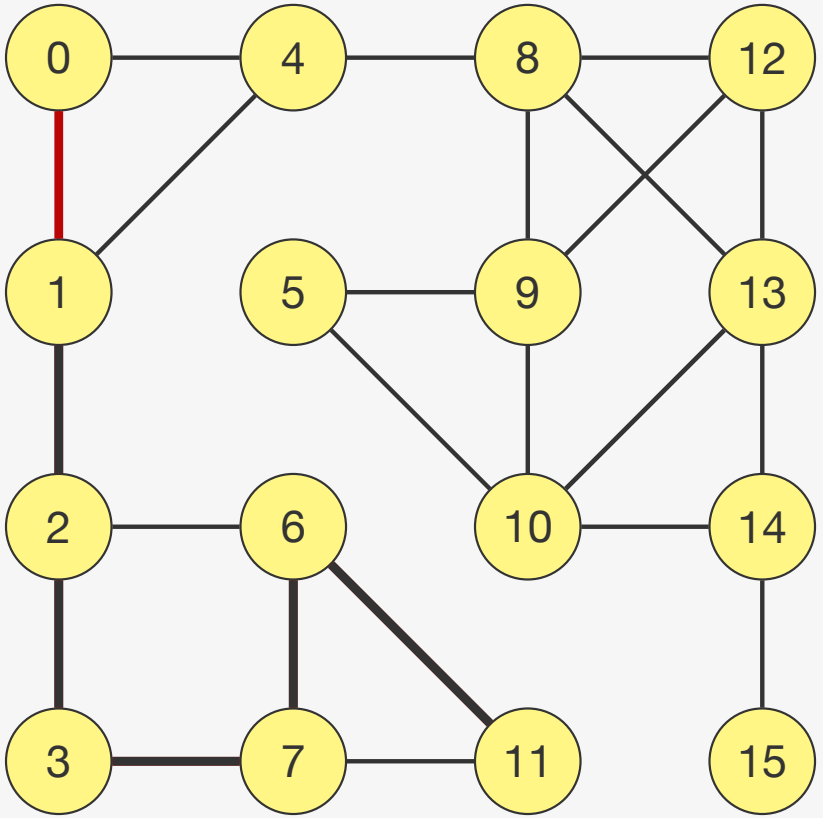
Exemplo - Existe caminho de 0 até 15?



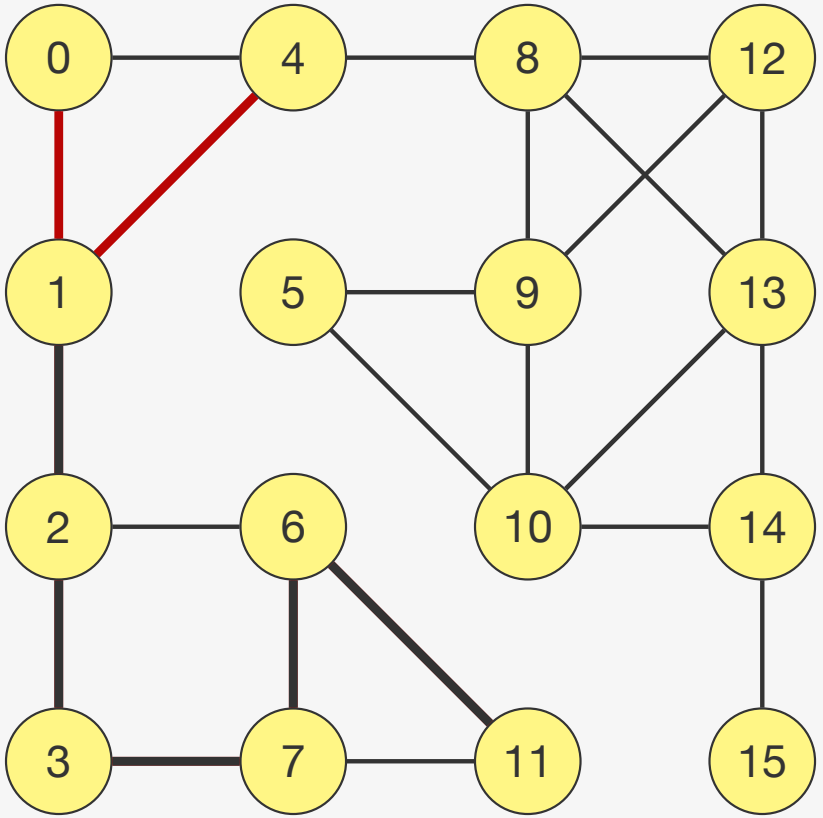
Exemplo - Existe caminho de 0 até 15?



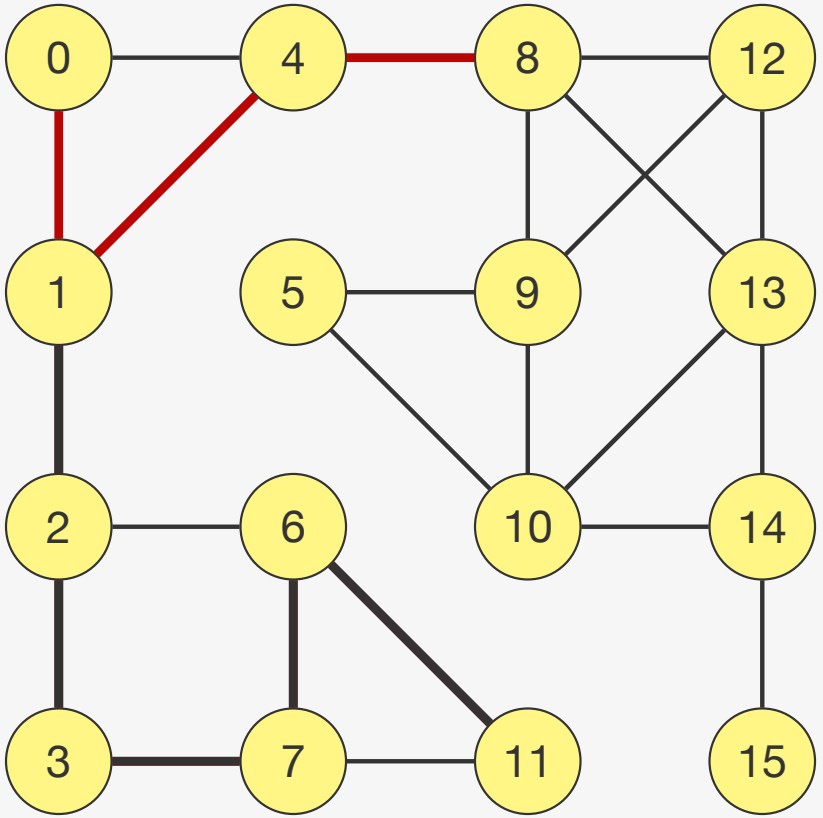
Exemplo - Existe caminho de 0 até 15?



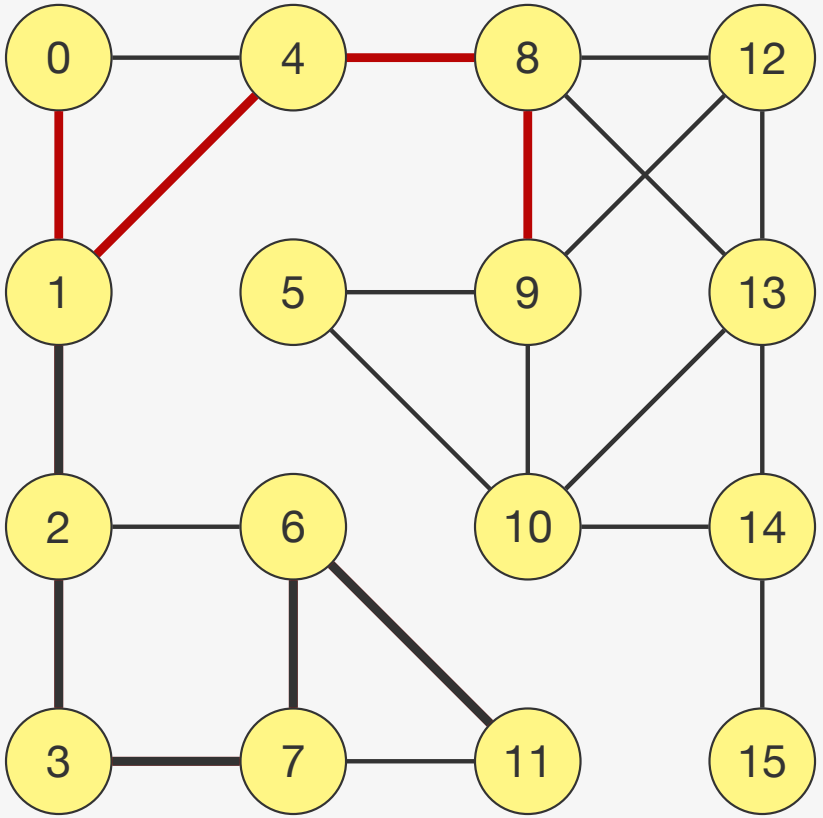
Exemplo - Existe caminho de 0 até 15?



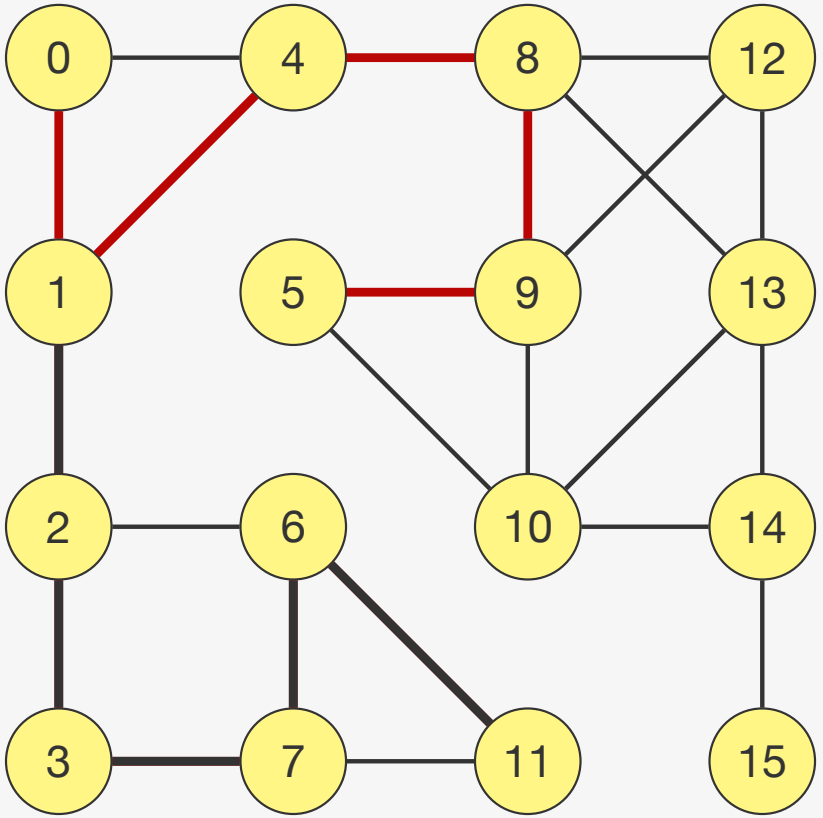
Exemplo - Existe caminho de 0 até 15?



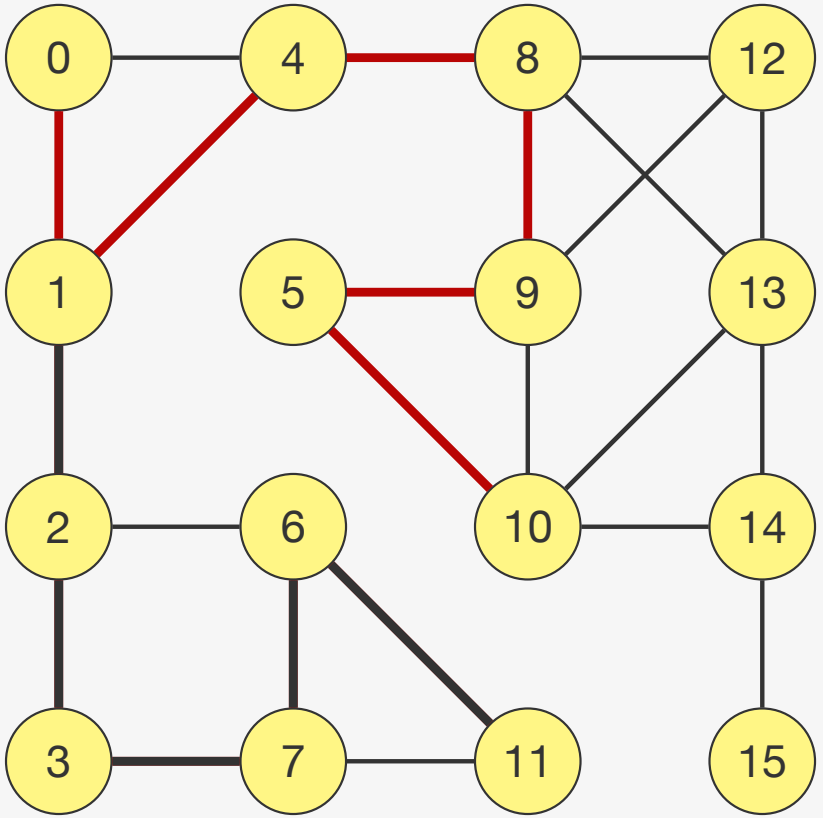
Exemplo - Existe caminho de 0 até 15?



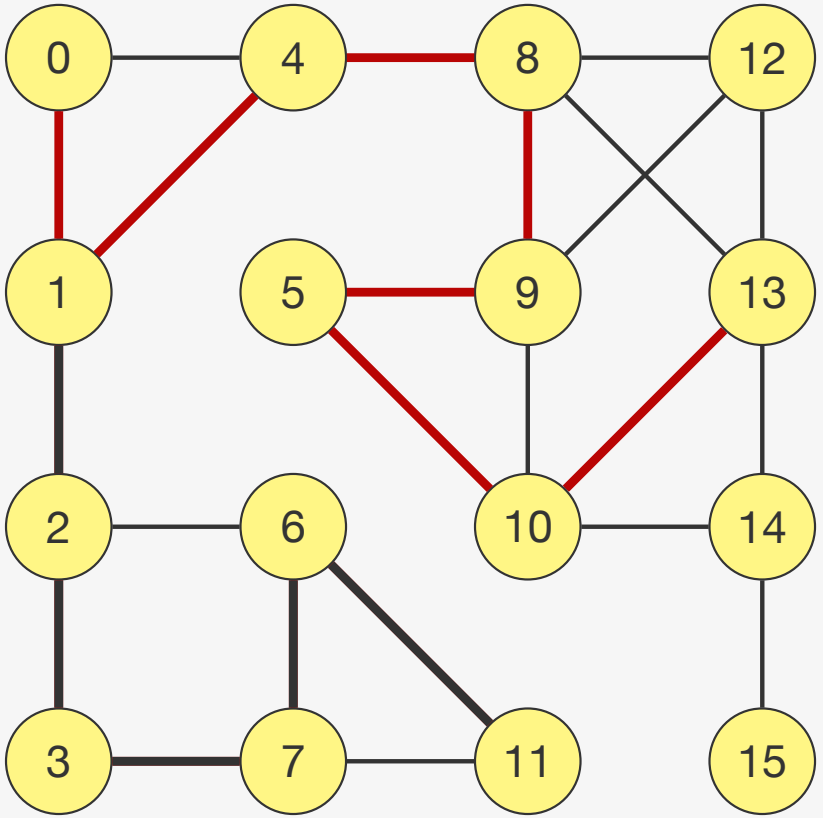
Exemplo - Existe caminho de 0 até 15?



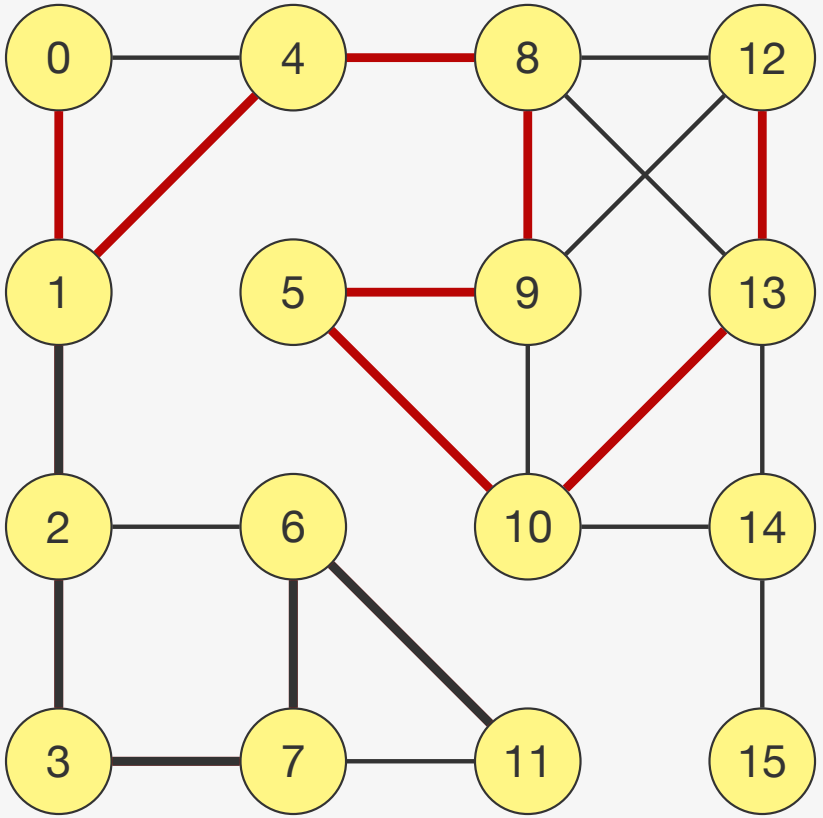
Exemplo - Existe caminho de 0 até 15?



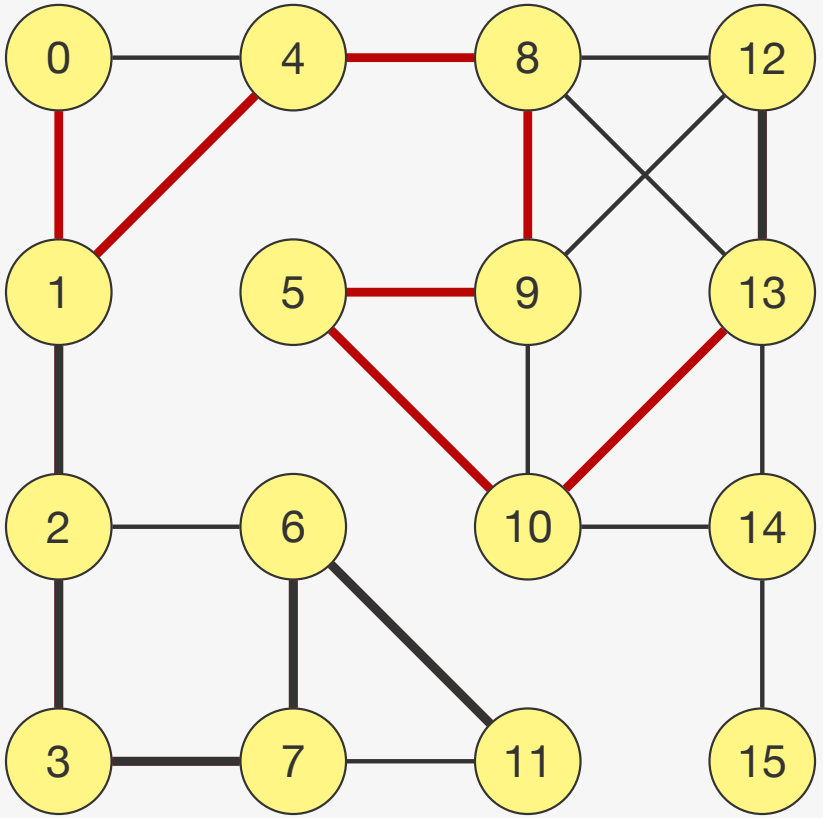
Exemplo - Existe caminho de 0 até 15?



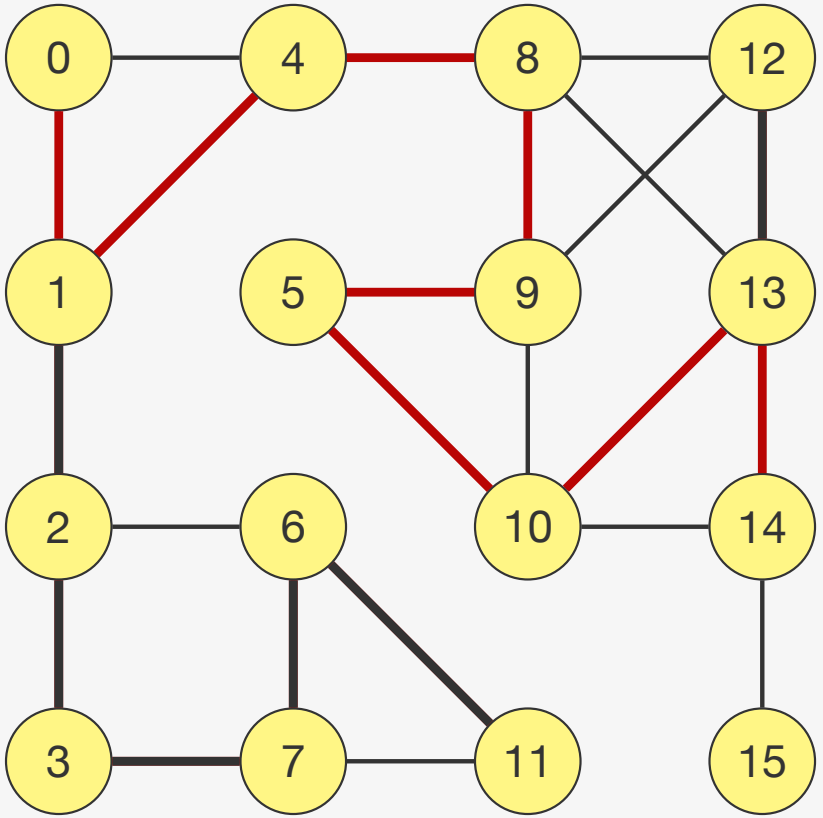
Exemplo - Existe caminho de 0 até 15?



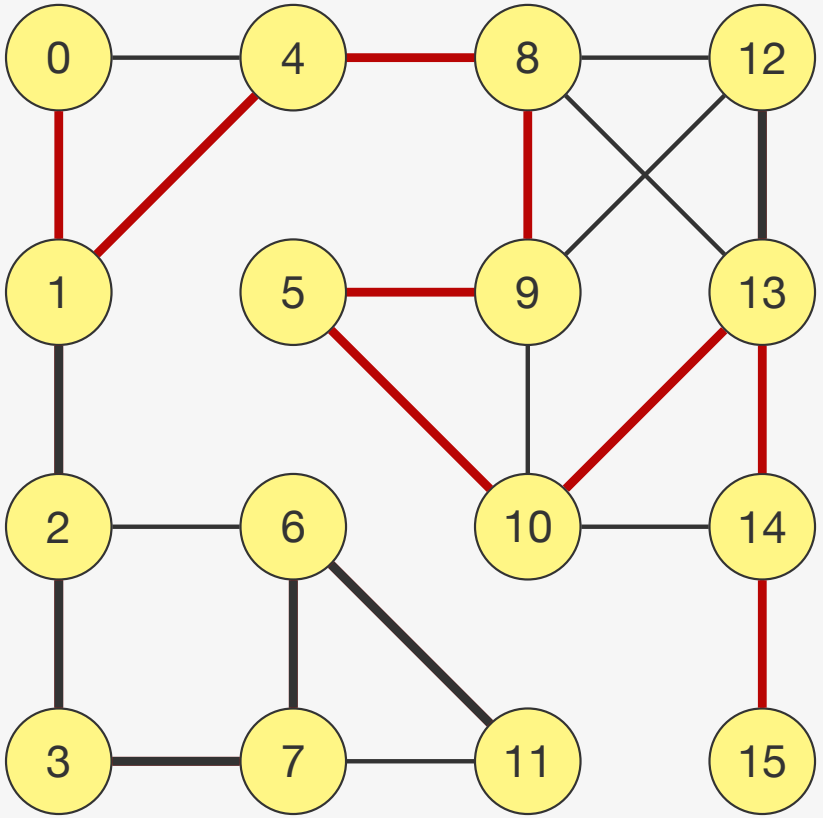
Exemplo - Existe caminho de 0 até 15?



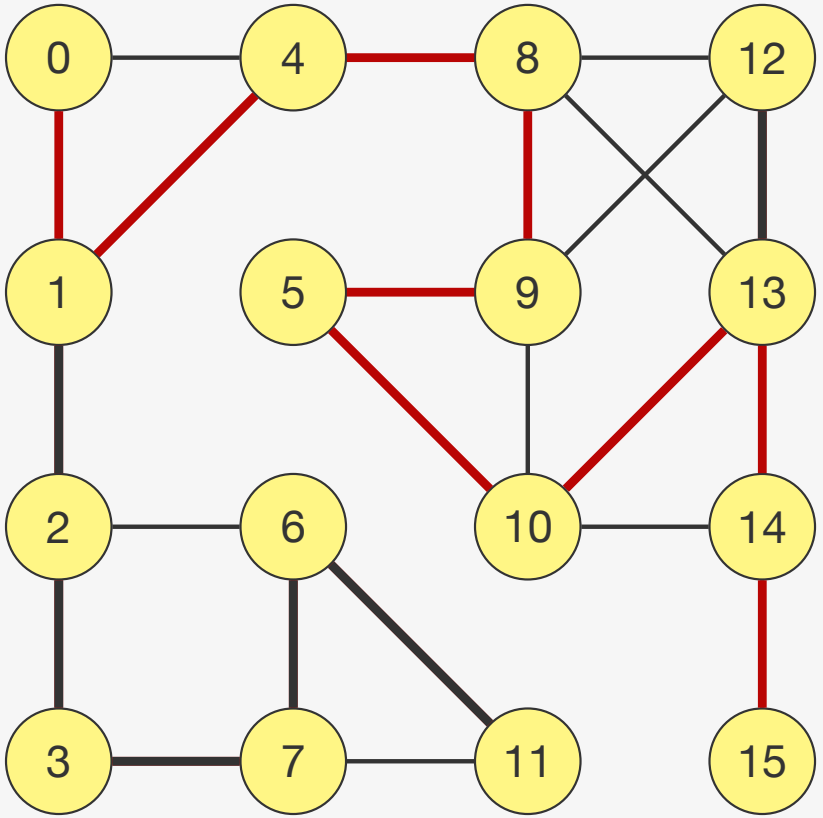
Exemplo - Existe caminho de 0 até 15?



Exemplo - Existe caminho de 0 até 15?

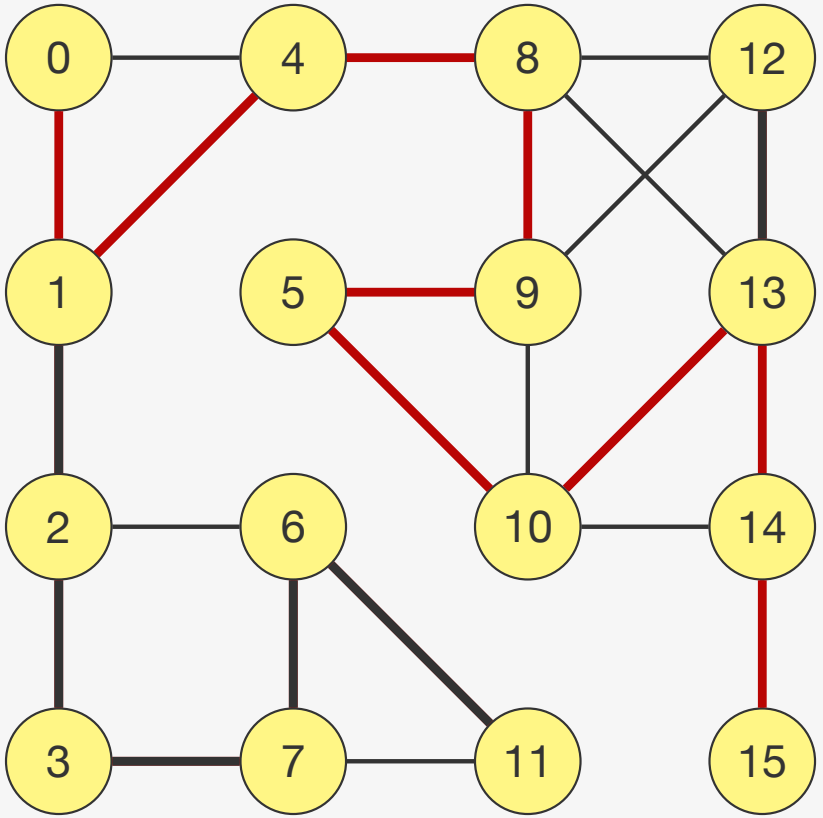


Exemplo - Existe caminho de 0 até 15?



Essa é uma busca em profundidade:

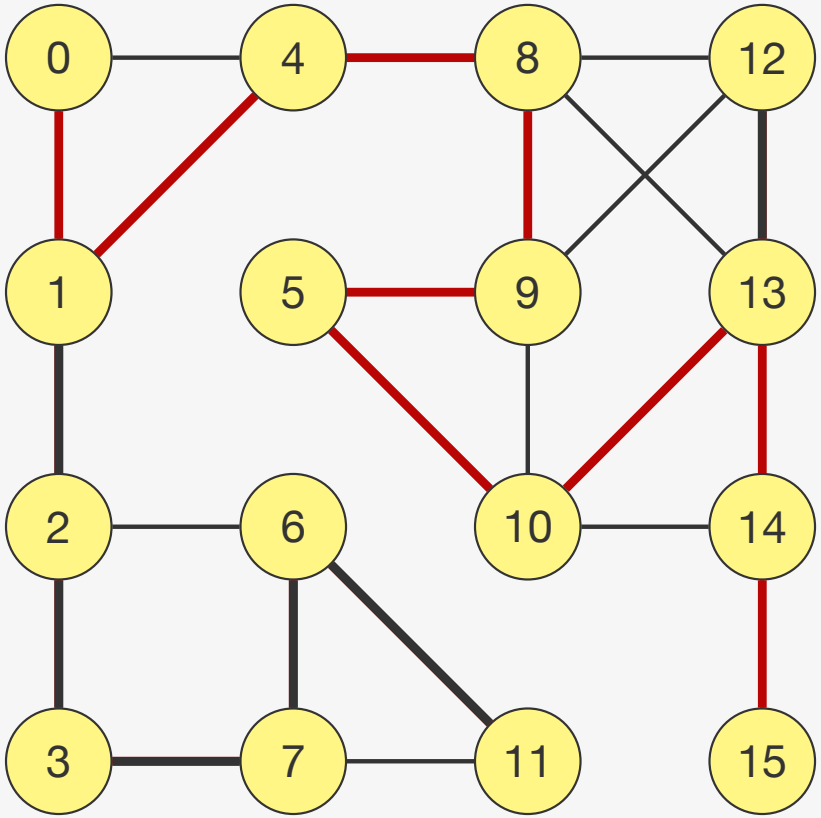
Exemplo - Existe caminho de 0 até 15?



Essa é uma busca em profundidade:

- Vá o máximo possível em uma direção

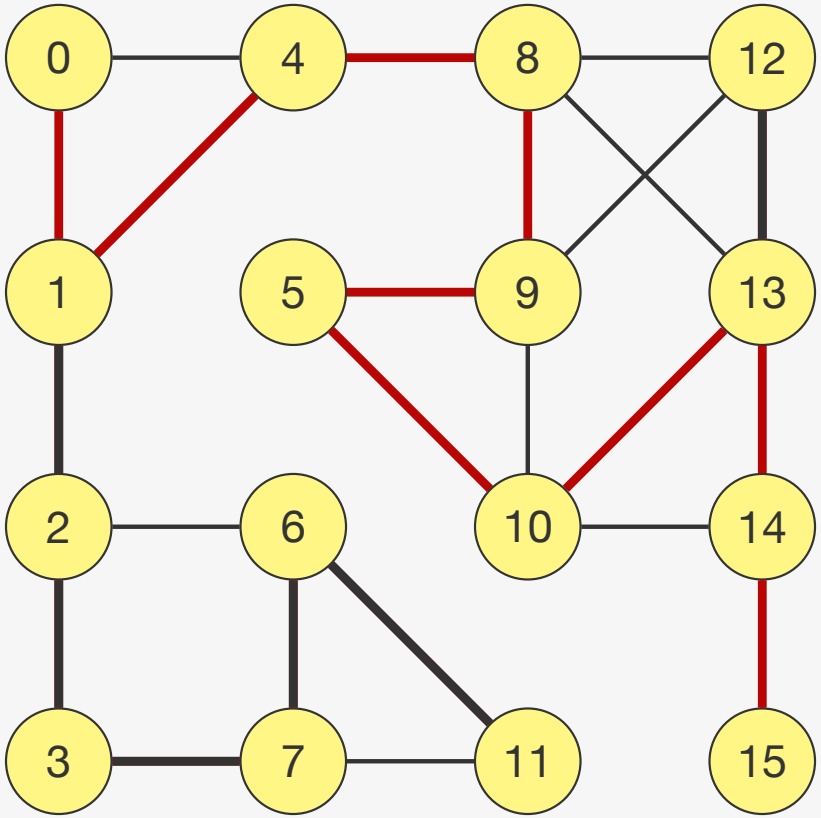
Exemplo - Existe caminho de 0 até 15?



Essa é uma busca em profundidade:

- Vá o máximo possível em uma direção
- Se não encontrarmos o vértice, volte o mínimo possível

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)

```
1 int existe_caminho(p_grafo g, int s, int t) {
2   int encontrou, i, *visitado = malloc(g->n * sizeof(int));
3   for (i = 0; i < g->n; i++)
4     visitado[i] = 0;
5   encontrou = busca_rec(g, visitado, s, t);
6   free(visitado);
7   return encontrou;
8 }
```

Implementação (com Matriz de Adjacências)

```
1 int existe_caminho(p_grafo g, int s, int t) {
2   int encontrou, i, *visitado = malloc(g->n * sizeof(int));
3   for (i = 0; i < g->n; i++)
4     visitado[i] = 0;
5   encontrou = busca_rec(g, visitado, s, t);
6   free(visitado);
7   return encontrou;
8 }
```

```
1 int busca_rec(p_grafo g, int *visitado, int v, int t) {
```

Implementação (com Matriz de Adjacências)

```
1 int existe_caminho(p_grafo g, int s, int t) {
2   int encontrou, i, *visitado = malloc(g->n * sizeof(int));
3   for (i = 0; i < g->n; i++)
4     visitado[i] = 0;
5   encontrou = busca_rec(g, visitado, s, t);
6   free(visitado);
7   return encontrou;
8 }
```

```
1 int busca_rec(p_grafo g, int *visitado, int v, int t) {
2   int w;
3   if (v == t)
4     return 1; /*sempre existe caminho de t para t*/
```

Implementação (com Matriz de Adjacências)

```
1 int existe_caminho(p_grafo g, int s, int t) {
2   int encontrou, i, *visitado = malloc(g->n * sizeof(int));
3   for (i = 0; i < g->n; i++)
4     visitado[i] = 0;
5   encontrou = busca_rec(g, visitado, s, t);
6   free(visitado);
7   return encontrou;
8 }
```

```
1 int busca_rec(p_grafo g, int *visitado, int v, int t) {
2   int w;
3   if (v == t)
4     return 1; /*sempre existe caminho de t para t*/
5   visitado[v] = 1;
```

Implementação (com Matriz de Adjacências)

```
1 int existe_caminho(p_grafo g, int s, int t) {
2   int encontrou, i, *visitado = malloc(g->n * sizeof(int));
3   for (i = 0; i < g->n; i++)
4     visitado[i] = 0;
5   encontrou = busca_rec(g, visitado, s, t);
6   free(visitado);
7   return encontrou;
8 }
```

```
1 int busca_rec(p_grafo g, int *visitado, int v, int t) {
2   int w;
3   if (v == t)
4     return 1; /*sempre existe caminho de t para t*/
5   visitado[v] = 1;
6   for (w = 0; w < g->n; w++)
```

Implementação (com Matriz de Adjacências)

```
1 int existe_caminho(p_grafo g, int s, int t) {
2   int encontrou, i, *visitado = malloc(g->n * sizeof(int));
3   for (i = 0; i < g->n; i++)
4     visitado[i] = 0;
5   encontrou = busca_rec(g, visitado, s, t);
6   free(visitado);
7   return encontrou;
8 }
```

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

Implementação (com Matriz de Adjacências)

```
1 int existe_caminho(p_grafo g, int s, int t) {
2   int encontrou, i, *visitado = malloc(g->n * sizeof(int));
3   for (i = 0; i < g->n; i++)
4     visitado[i] = 0;
5   encontrou = busca_rec(g, visitado, s, t);
6   free(visitado);
7   return encontrou;
8 }
```

```
1 int busca_rec(p_grafo g, int *visitado, int v, int t) {
2   int w;
3   if (v == t)
4     return 1; /*sempre existe caminho de t para t*/
5   visitado[v] = 1;
6   for (w = 0; w < g->n; w++)
7     if (g->adj[v][w] && !visitado[w])
8       if (busca_rec(g, visitado, w, t))
9         return 1;
```

Implementação (com Matriz de Adjacências)

```
1  int existe_caminho(p_grafo g, int s, int t) {
2    int encontrou, i, *visitado = malloc(g->n * sizeof(int));
3    for (i = 0; i < g->n; i++)
4      visitado[i] = 0;
5    encontrou = busca_rec(g, visitado, s, t);
6    free(visitado);
7    return encontrou;
8  }
```

```
1  int busca_rec(p_grafo g, int *visitado, int v, int t) {
2    int w;
3    if (v == t)
4      return 1; /*sempre existe caminho de t para t*/
5    visitado[v] = 1;
6    for (w = 0; w < g->n; w++)
7      if (g->adj[v][w] && !visitado[w])
8        if (busca_rec(g, visitado, w, t))
9          return 1;
10   return 0;
11 }
```

Implementação (com Matriz de Adjacências)

```
1  int existe_caminho(p_grafo g, int s, int t) {
2    int encontrou, i, *visitado = malloc(g->n * sizeof(int));
3    for (i = 0; i < g->n; i++)
4      visitado[i] = 0;
5    encontrou = busca_rec(g, visitado, s, t);
6    free(visitado);
7    return encontrou;
8  }
```

```
1  int busca_rec(p_grafo g, int *visitado, int v, int t) {
2    int w;
3    if (v == t)
4      return 1; /*sempre existe caminho de t para t*/
5    visitado[v] = 1;
6    for (w = 0; w < g->n; w++)
7      if (g->adj[v][w] && !visitado[w])
8        if (busca_rec(g, visitado, w, t))
9          return 1;
10   return 0;
11 }
```

E se quisermos saber quais são as componentes conexas?

Componentes Conexas (Listas de Adjacência)

```
1 int * encontra_componentes(p_grafo g) {
```

Componentes Conexas (Listas de Adjacência)

```
1 int * encontra_componentes(p_grafo g) {
2   int s, c = 0, *componentes = malloc(g->n * sizeof(int));
```

Componentes Conexas (Listas de Adjacência)

```
1 int * encontra_componentes(p_grafo g) {
2   int s, c = 0, *componentes = malloc(g->n * sizeof(int));
3   for (s = 0; s < g->n; s++)
4     componentes[s] = -1;
```

Componentes Conexas (Listas de Adjacência)

```
1 int * encontra_componentes(p_grafo g) {
2   int s, c = 0, *componentes = malloc(g->n * sizeof(int));
3   for (s = 0; s < g->n; s++)
4     componentes[s] = -1;
5   for (s = 0; s < g->n; s++)
6     if (componentes[s] == -1) {
```

Componentes Conexas (Listas de Adjacência)

```
1 int * encontra_componentes(p_grafo g) {
2   int s, c = 0, *componentes = malloc(g->n * sizeof(int));
3   for (s = 0; s < g->n; s++)
4     componentes[s] = -1;
5   for (s = 0; s < g->n; s++)
6     if (componentes[s] == -1) {
7       visita_rec(g, componentes, c, s);
```

Componentes Conexas (Listas de Adjacência)

```
1 int * encontra_componentes(p_grafo g) {
2   int s, c = 0, *componentes = malloc(g->n * sizeof(int));
3   for (s = 0; s < g->n; s++)
4     componentes[s] = -1;
5   for (s = 0; s < g->n; s++)
6     if (componentes[s] == -1) {
7       visita_rec(g, componentes, c, s);
8       c++;
```

Componentes Conexas (Listas de Adjacência)

```
 1 int * encontra_componentes(p_grafo g) {
 2   int s, c = 0, *componentes = malloc(g->n * sizeof(int));
 3   for (s = 0; s < g->n; s++)
 4     componentes[s] = -1;
 5   for (s = 0; s < g->n; s++)
 6     if (componentes[s] == -1) {
 7       visita_rec(g, componentes, c, s);
 8       c++;
 9     }
10   return componentes;
11 }
```

Componentes Conexas (Listas de Adjacência)

```
 1 int * encontra_componentes(p_grafo g) {
 2   int s, c = 0, *componentes = malloc(g->n * sizeof(int));
 3   for (s = 0; s < g->n; s++)
 4     componentes[s] = -1;
 5   for (s = 0; s < g->n; s++)
 6     if (componentes[s] == -1) {
 7       visita_rec(g, componentes, c, s);
 8       c++;
 9     }
10   return componentes;
11 }
```

```
1 void visita_rec(p_grafo g, int *componentes, int comp, int v) {
```

Componentes Conexas (Listas de Adjacência)

```
1  int * encontra_componentes(p_grafo g) {
2    int s, c = 0, *componentes = malloc(g->n * sizeof(int));
3    for (s = 0; s < g->n; s++)
4      componentes[s] = -1;
5    for (s = 0; s < g->n; s++)
6      if (componentes[s] == -1) {
7        visita_rec(g, componentes, c, s);
8        c++;
9      }
10   return componentes;
11 }
```

```
1  void visita_rec(p_grafo g, int *componentes, int comp, int v) {
2    p_no t;
3    componentes[v] = comp;
```

Componentes Conexas (Listas de Adjacência)

```
1  int * encontra_componentes(p_grafo g) {
2    int s, c = 0, *componentes = malloc(g->n * sizeof(int));
3    for (s = 0; s < g->n; s++)
4      componentes[s] = -1;
5    for (s = 0; s < g->n; s++)
6      if (componentes[s] == -1) {
7        visita_rec(g, componentes, c, s);
8        c++;
9      }
10   return componentes;
11 }
```

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

Componentes Conexas (Listas de Adjacência)

```
1  int * encontra_componentes(p_grafo g) {
2    int s, c = 0, *componentes = malloc(g->n * sizeof(int));
3    for (s = 0; s < g->n; s++)
4      componentes[s] = -1;
5    for (s = 0; s < g->n; s++)
6      if (componentes[s] == -1) {
7        visita_rec(g, componentes, c, s);
8        c++;
9      }
10   return componentes;
11 }
```

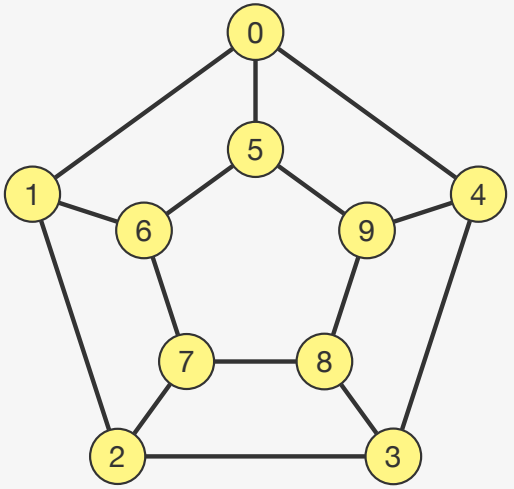
```
1  void visita_rec(p_grafo g, int *componentes, int comp, int v) {
2    p_no t;
3    componentes[v] = comp;
4    for (t = g->adj[v]; t != NULL; t = t->prox)
5      if (componentes[t->v] == -1)
```

Componentes Conexas (Listas de Adjacência)

```
1  int * encontra_componentes(p_grafo g) {
2    int s, c = 0, *componentes = malloc(g->n * sizeof(int));
3    for (s = 0; s < g->n; s++)
4      componentes[s] = -1;
5    for (s = 0; s < g->n; s++)
6      if (componentes[s] == -1) {
7        visita_rec(g, componentes, c, s);
8        c++;
9      }
10   return componentes;
11 }
```

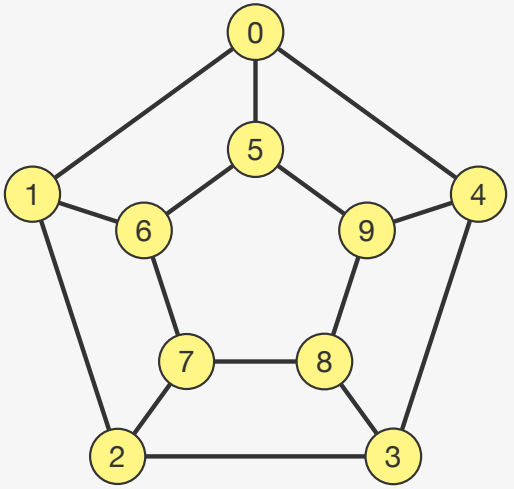
```
1  void visita_rec(p_grafo g, int *componentes, int comp, int v) {
2    p_no t;
3    componentes[v] = comp;
4    for (t = g->adj[v]; t != NULL; t = t->prox)
5      if (componentes[t->v] == -1)
6        visita_rec(g, componentes, comp, t->v);
7  }
```

Ciclos em Grafos



Um ciclo em um grafo é:

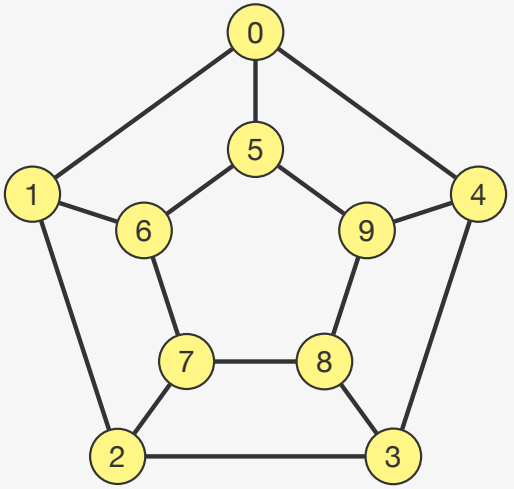
Ciclos em Grafos



Um ciclo em um grafo é:

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

Ciclos em Grafos

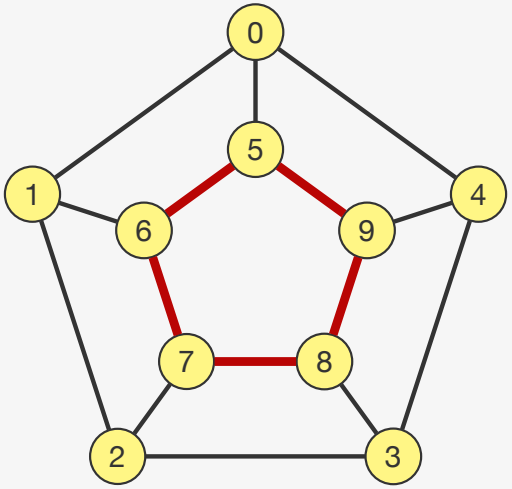


Um ciclo em um grafo é:

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

Por exemplo:

Ciclos em Grafos



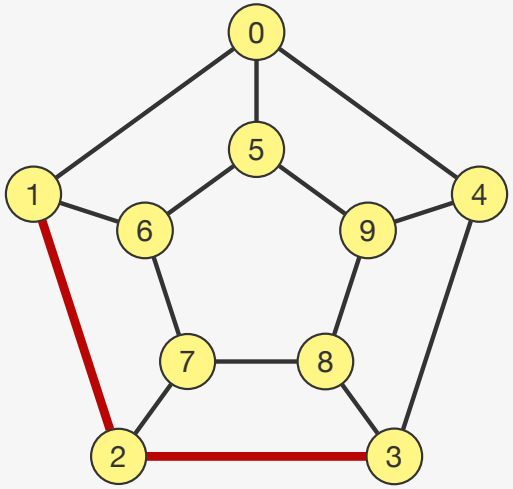
Um ciclo em um grafo é:

- 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

Ciclos em Grafos



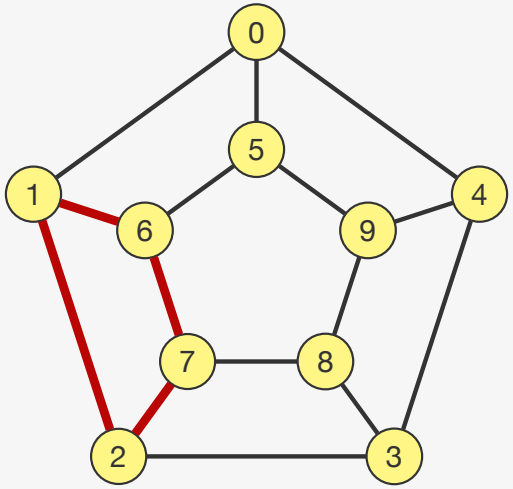
Um ciclo em um grafo é:

- 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

Ciclos em Grafos



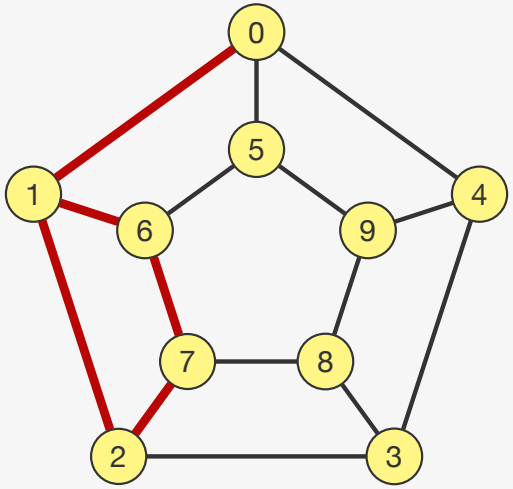
Um ciclo em um grafo é:

- 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

Ciclos em Grafos



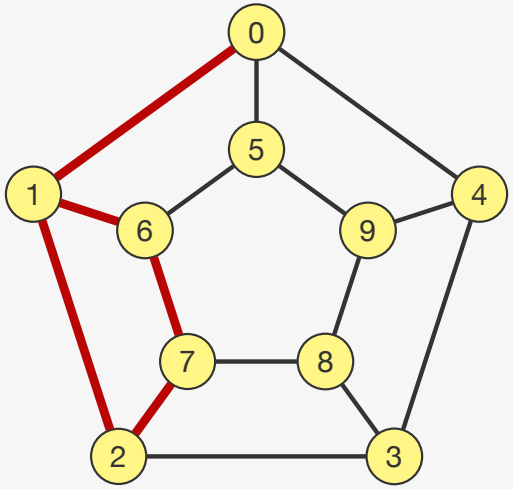
Um ciclo em um grafo é:

- 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

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

Árvores, Florestas e Subgrafos

Uma árvore é um grafo conexo acíclico

- Uma floresta é um grafo acíclico

Árvores, Florestas e Subgrafos

Uma árvore é um grafo conexo acíclico

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

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

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

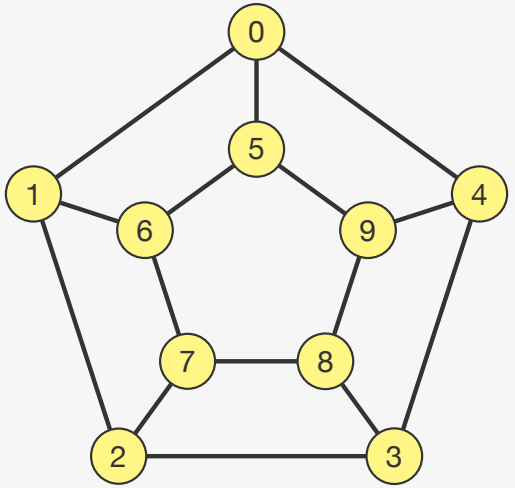
Árvores, Florestas e Subgrafos

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



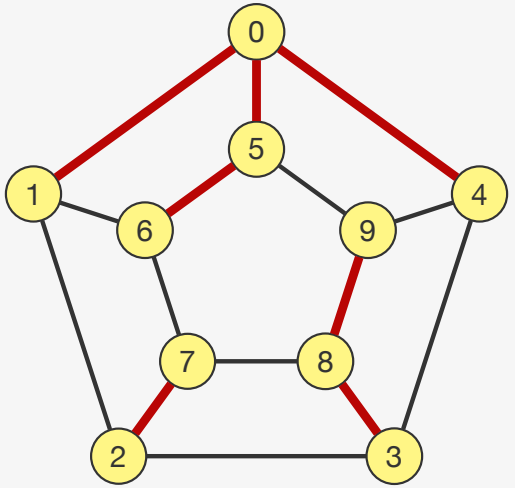
Árvores, Florestas e Subgrafos

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



Um subgrafo que é uma floresta

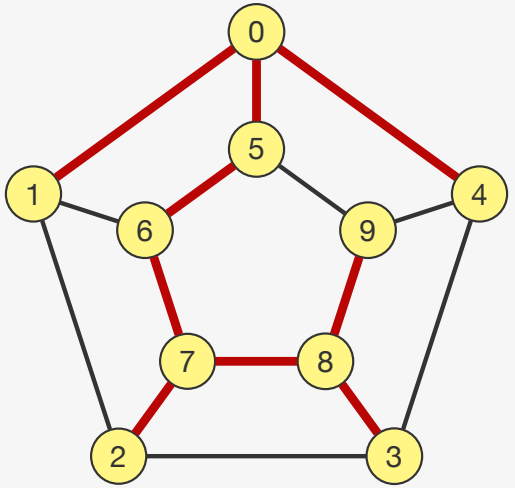
Árvores, Florestas e Subgrafos

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



Um subgrafo que é uma árvore

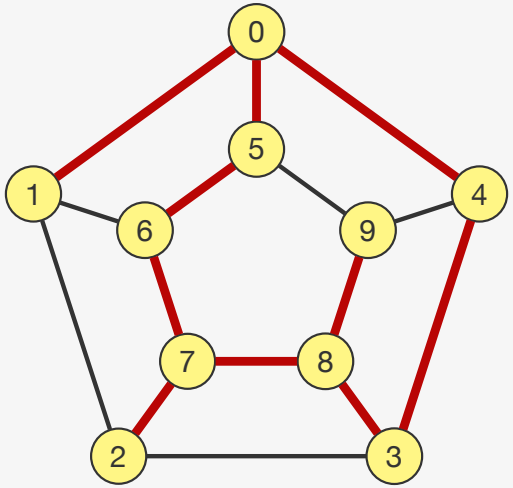
Árvores, Florestas e Subgrafos

Uma árvore é um grafo conexo acíclico

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

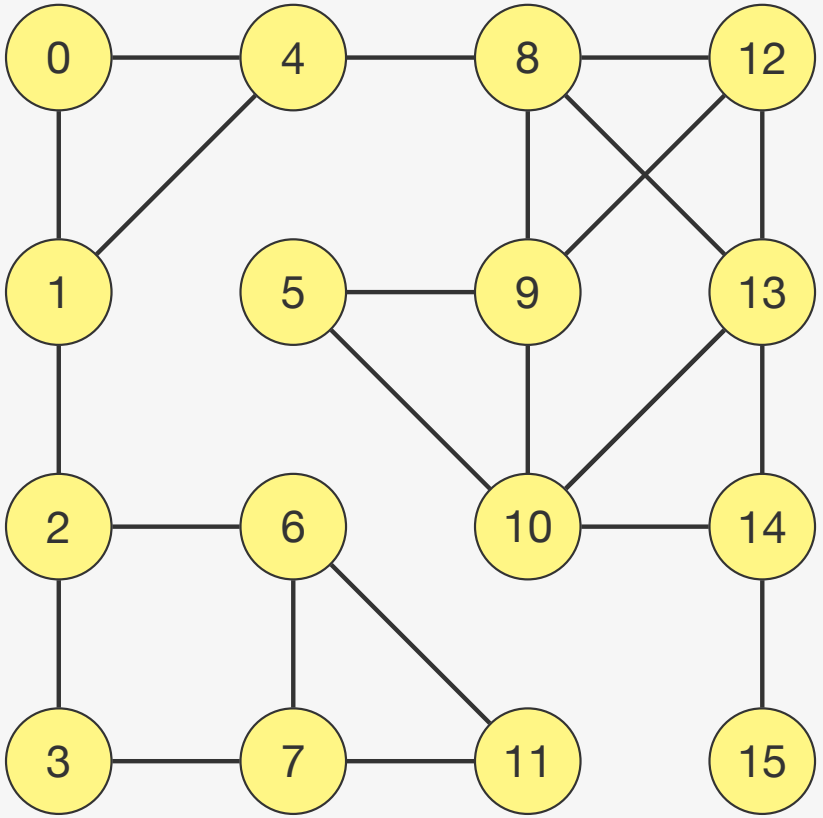
Um subgrafo é um grafo obtido a partir da remoção de vértices e arestas

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

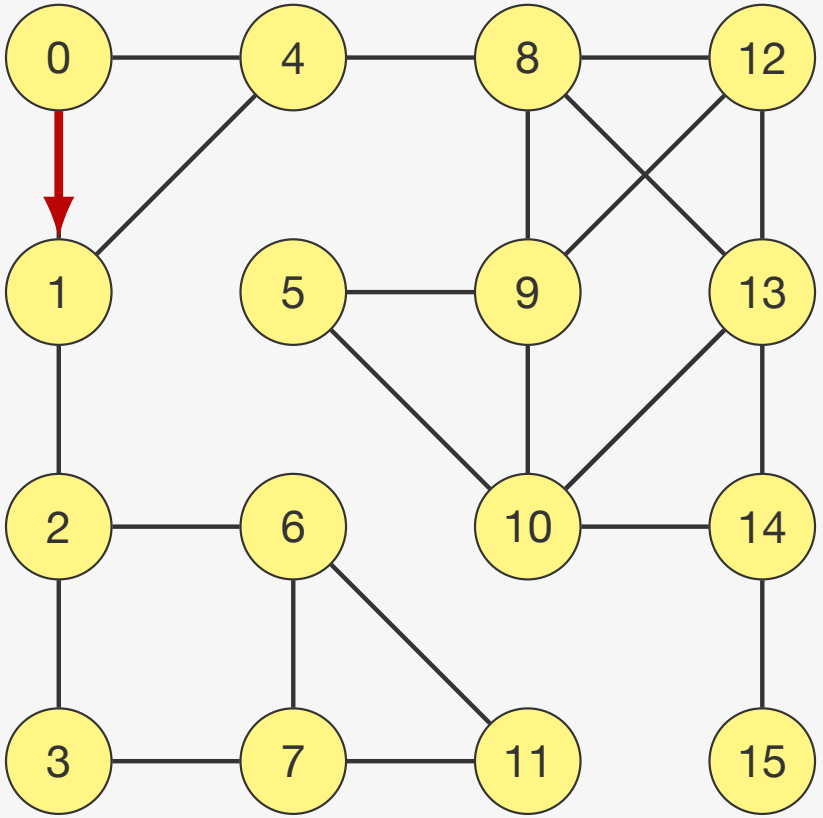


Um subgrafo com ciclo

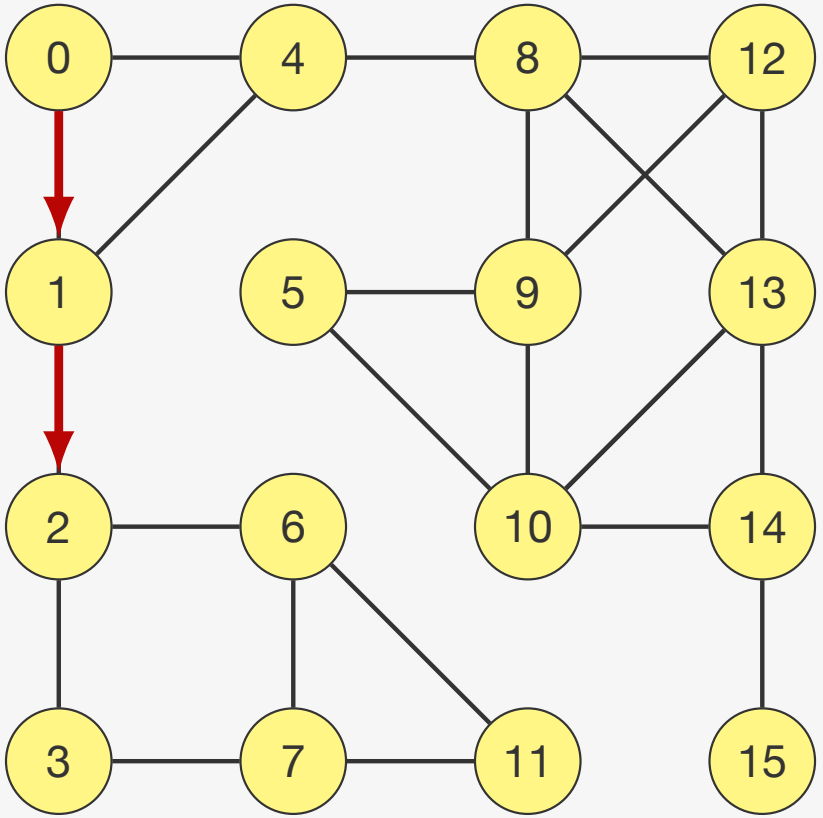
Caminhos de s para outros vértices da componente



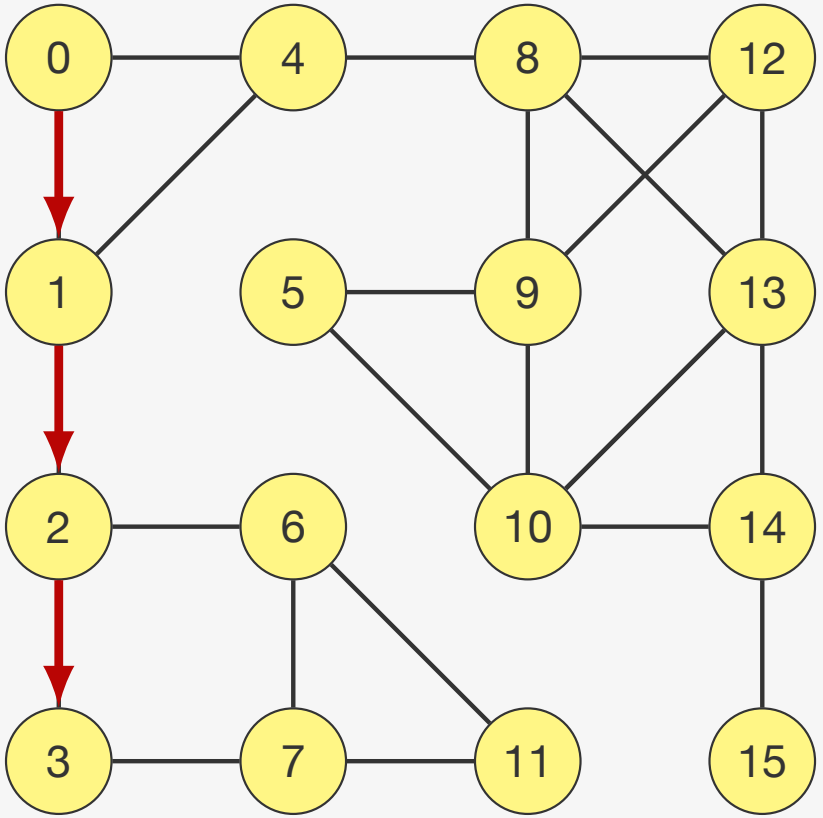
Caminhos de s para outros vértices da componente



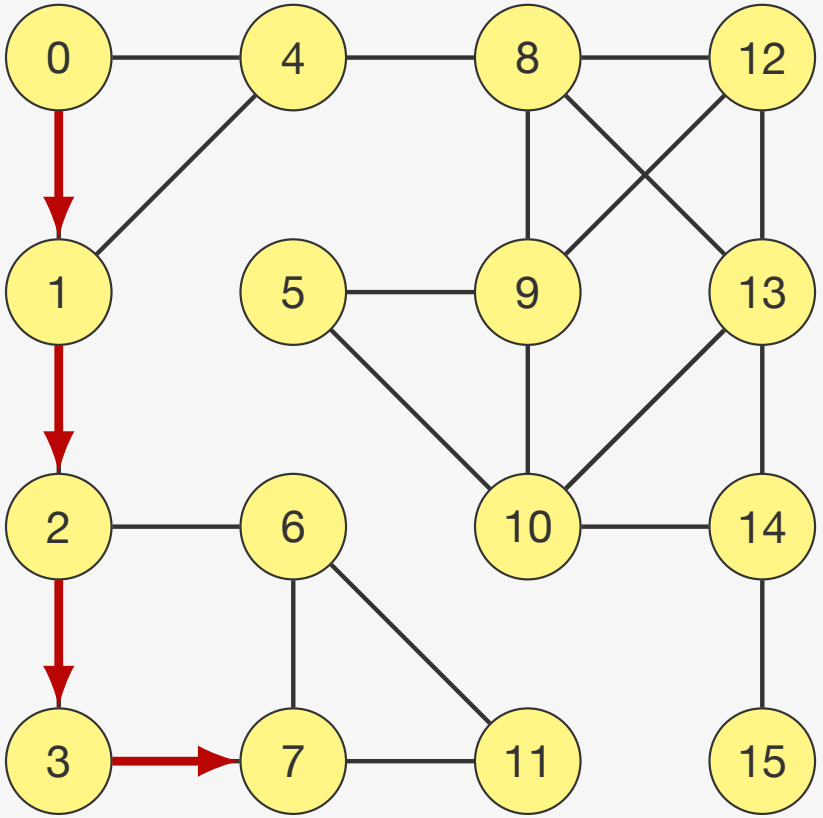
Caminhos de s para outros vértices da componente



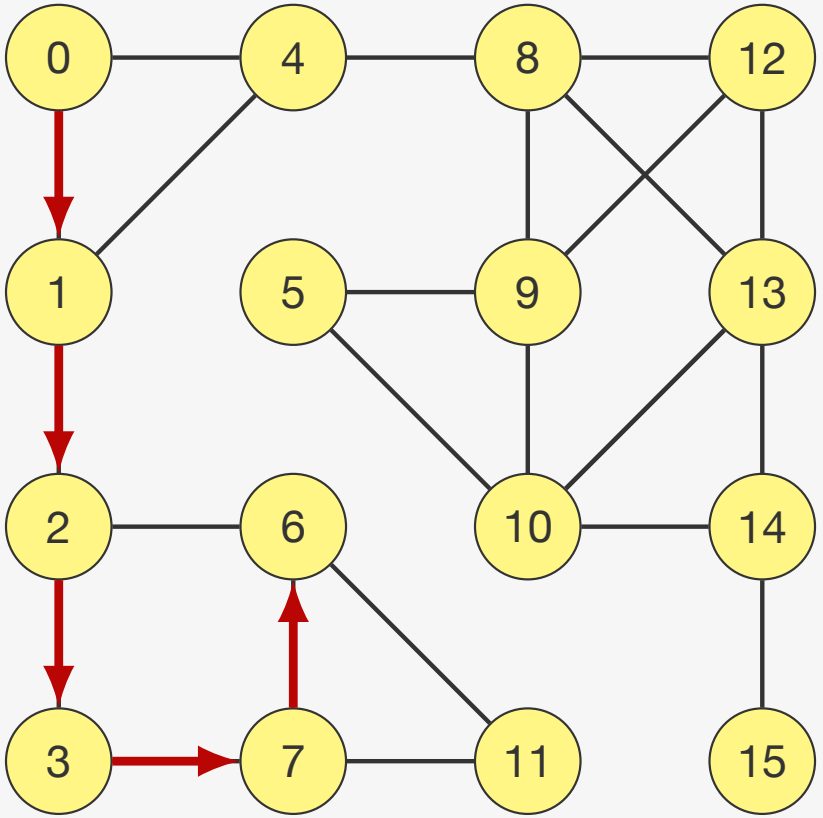
Caminhos de s para outros vértices da componente



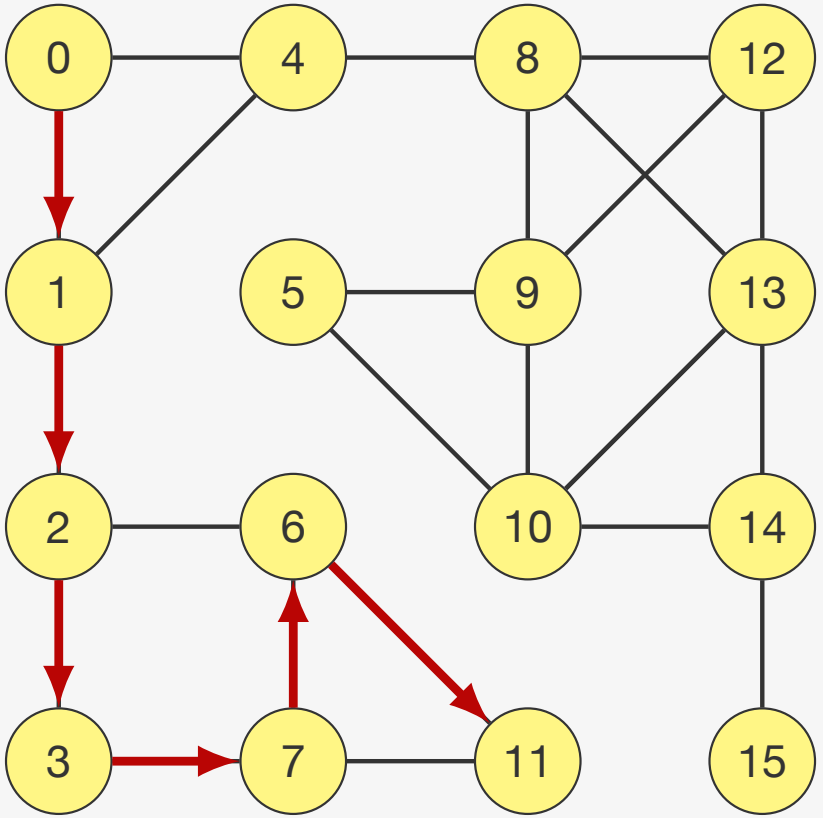
Caminhos de s para outros vértices da componente



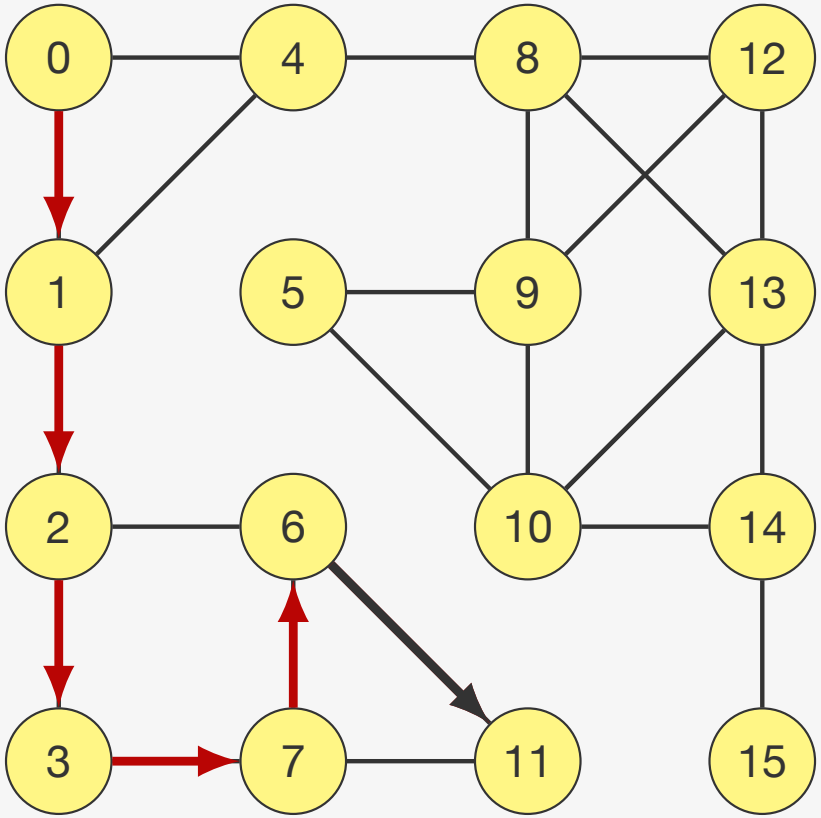
Caminhos de s para outros vértices da componente



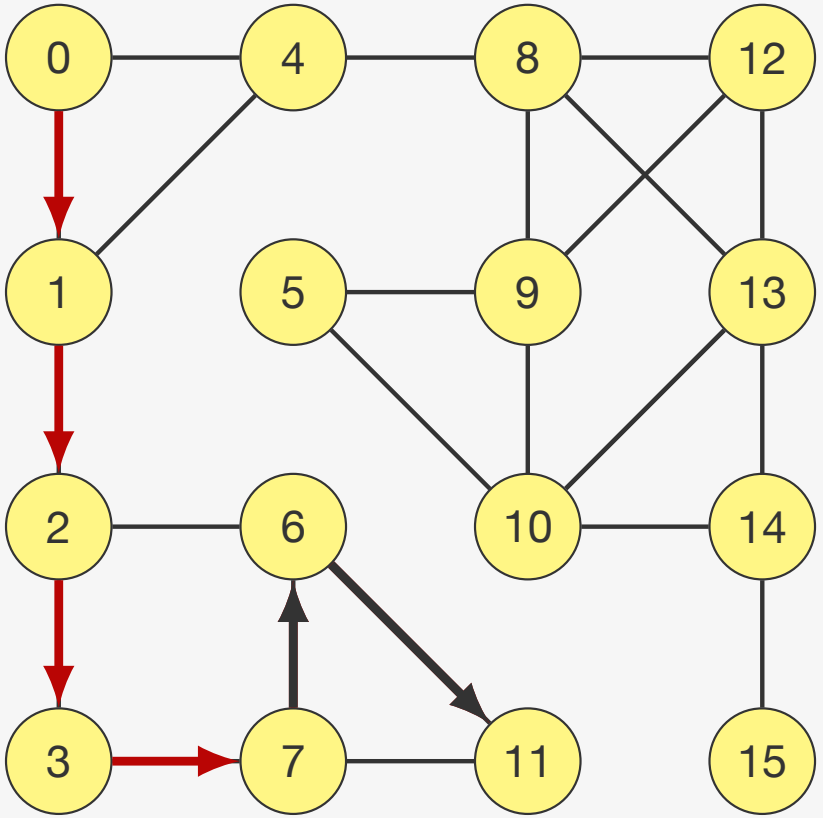
Caminhos de s para outros vértices da componente



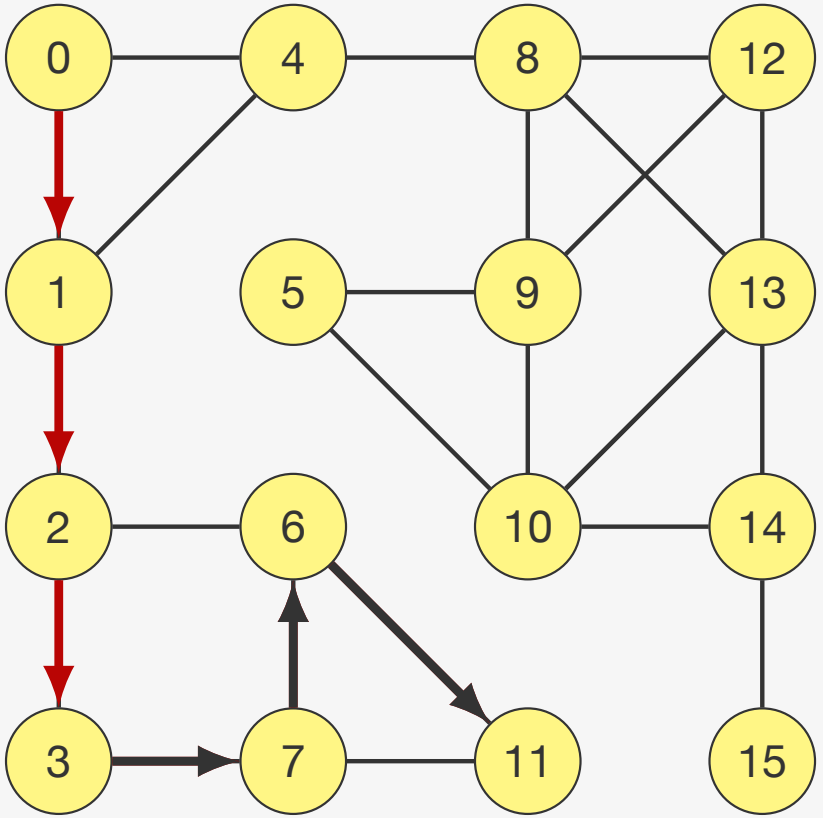
Caminhos de s para outros vértices da componente



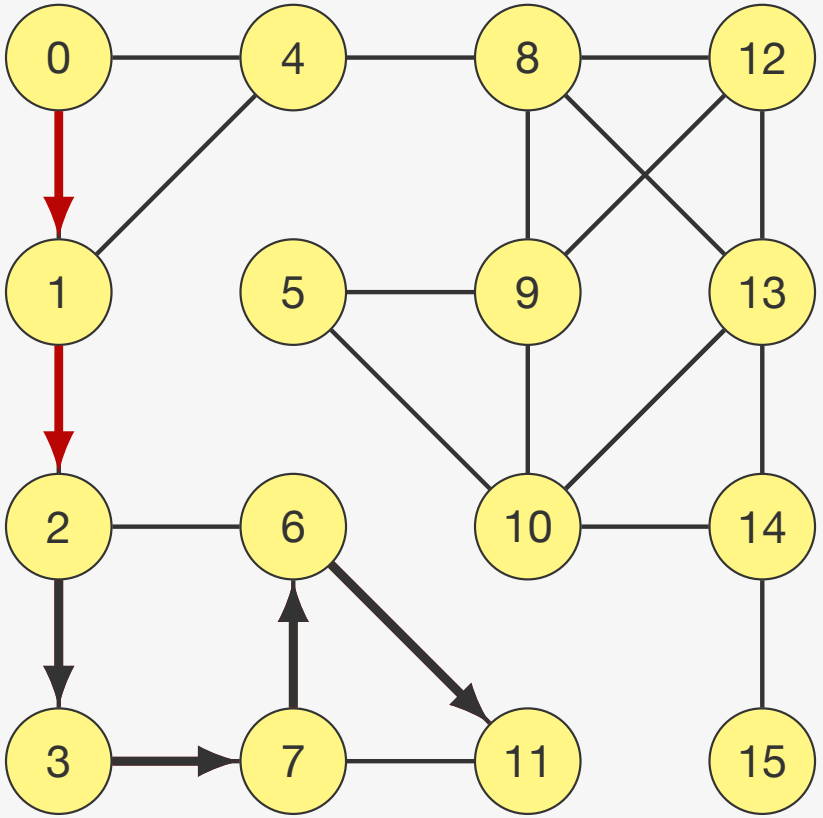
Caminhos de s para outros vértices da componente



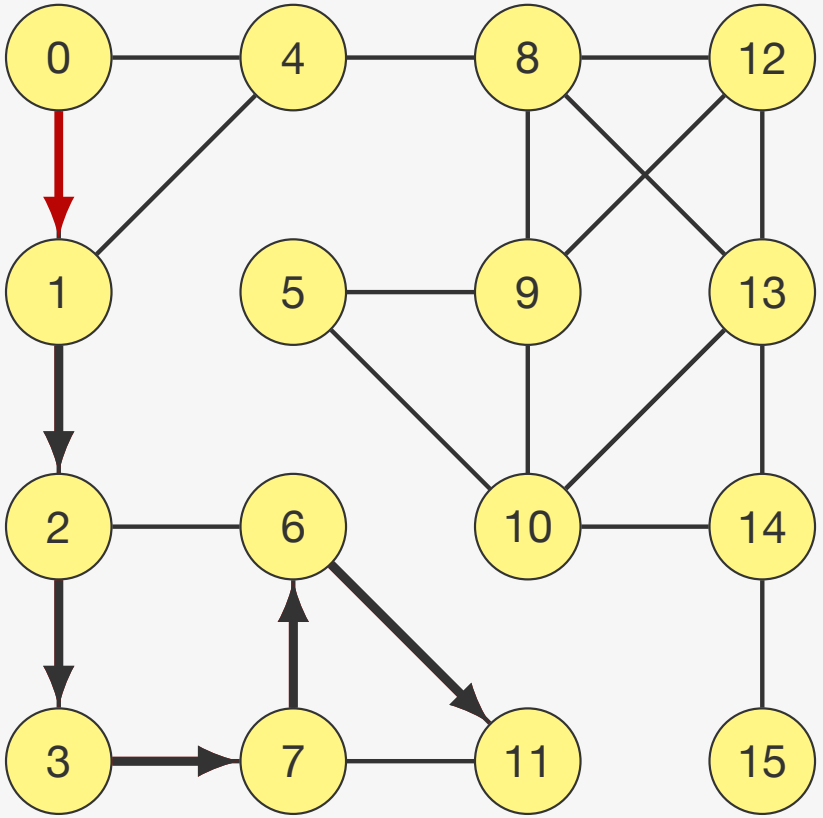
Caminhos de s para outros vértices da componente



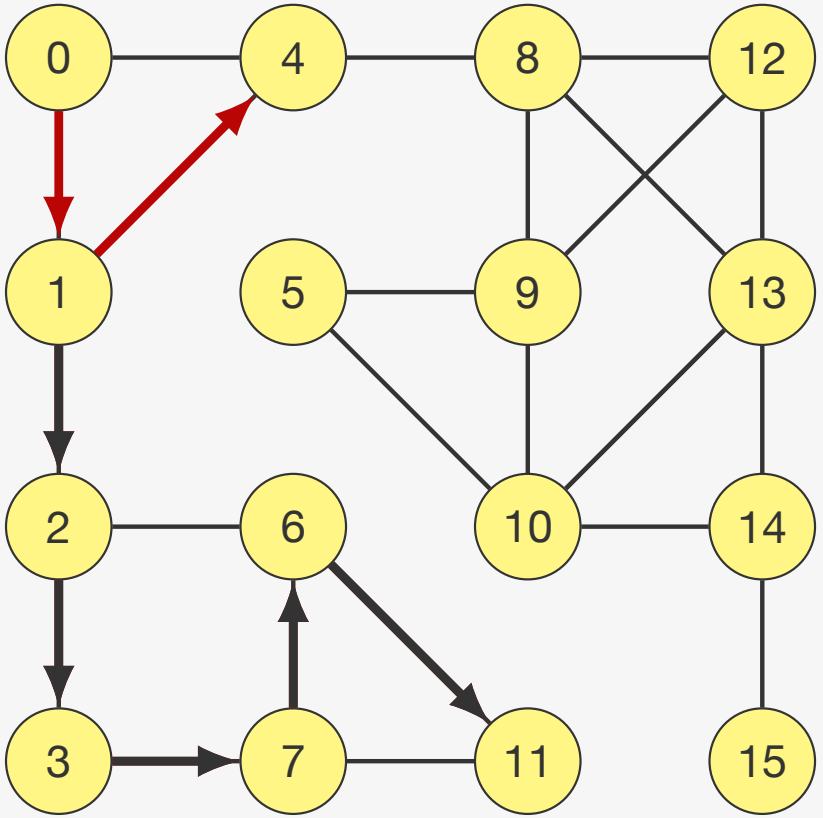
Caminhos de s para outros vértices da componente



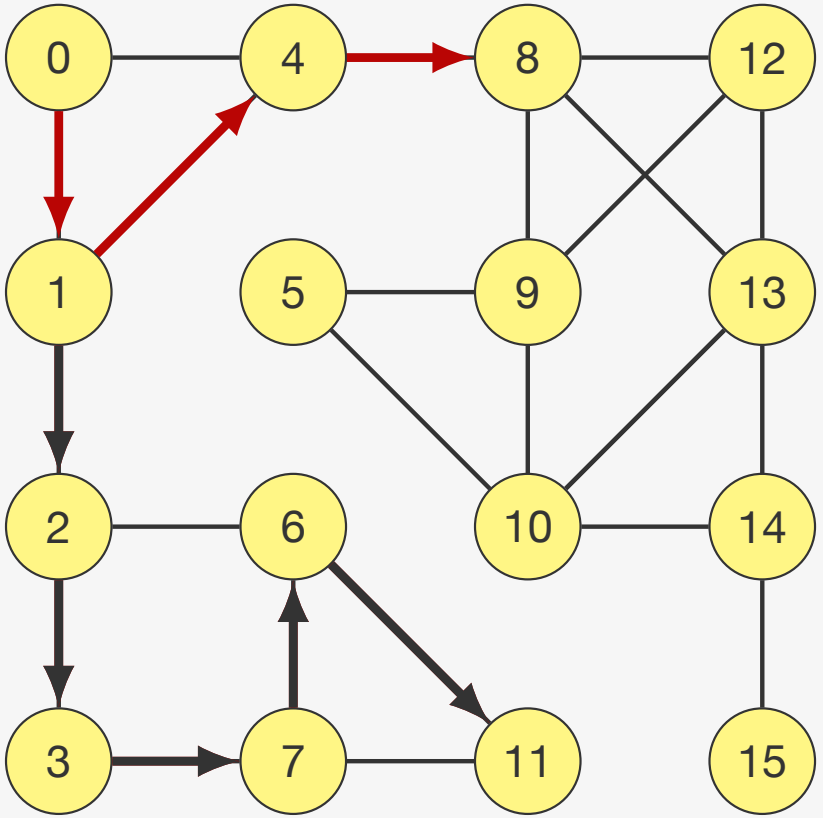
Caminhos de s para outros vértices da componente



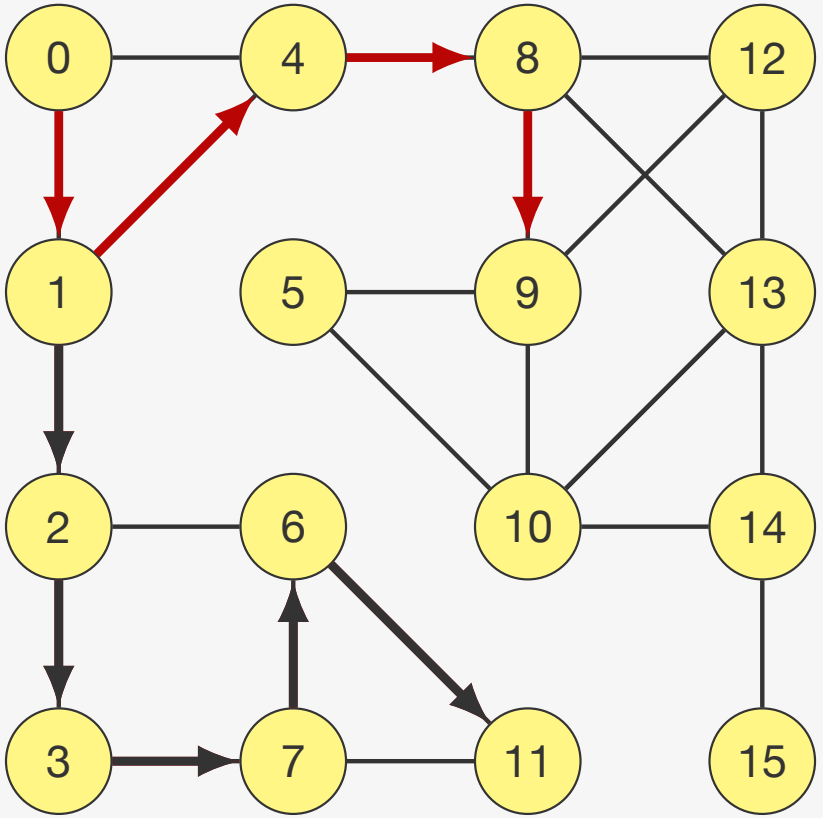
Caminhos de s para outros vértices da componente



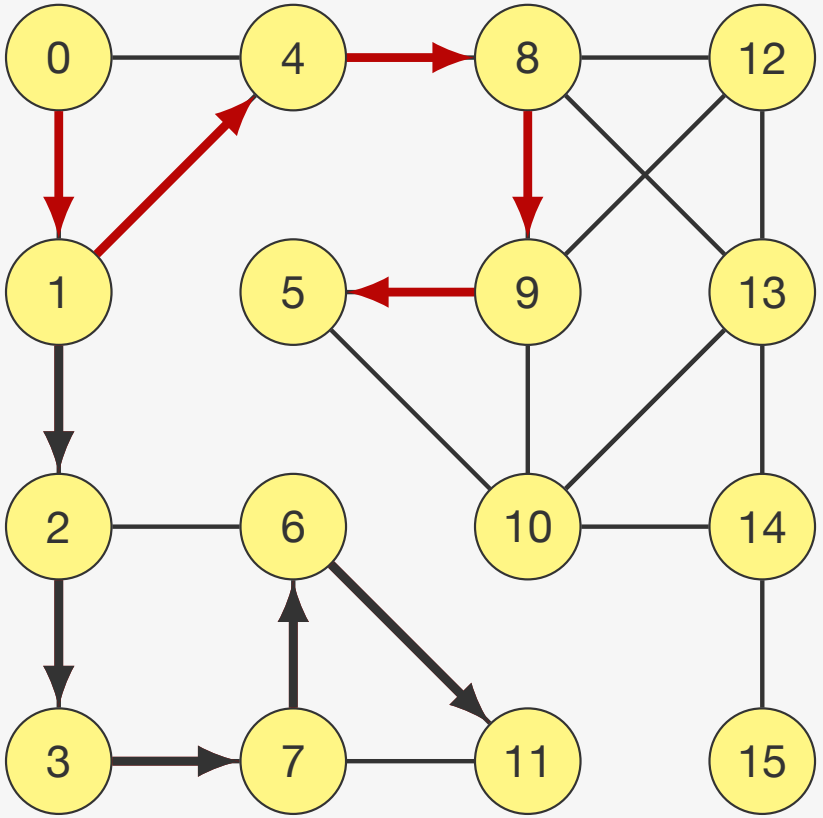
Caminhos de s para outros vértices da componente



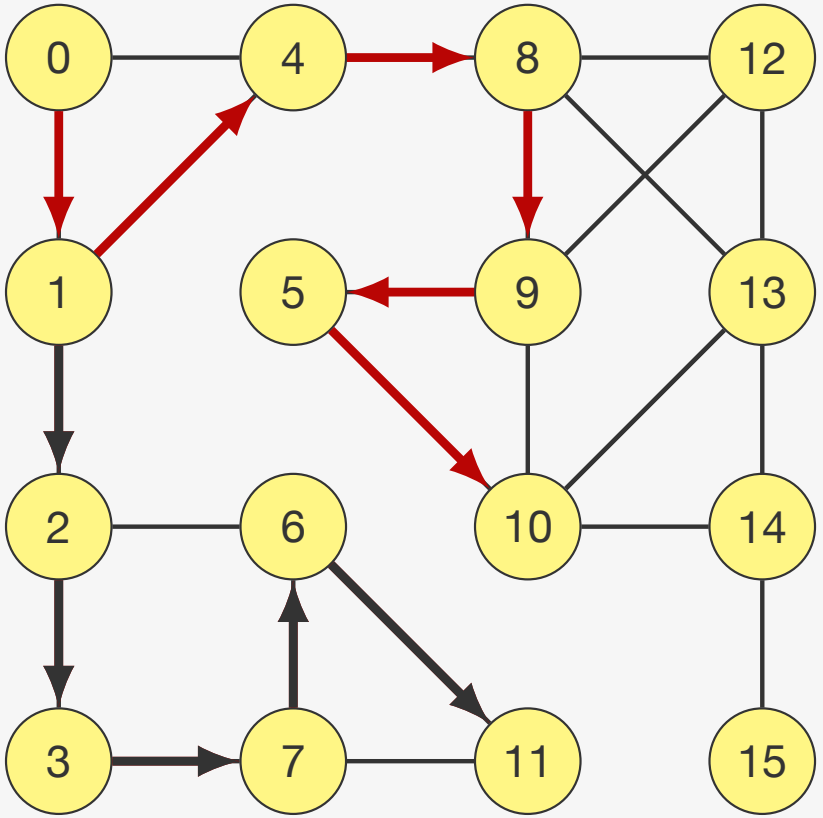
Caminhos de s para outros vértices da componente



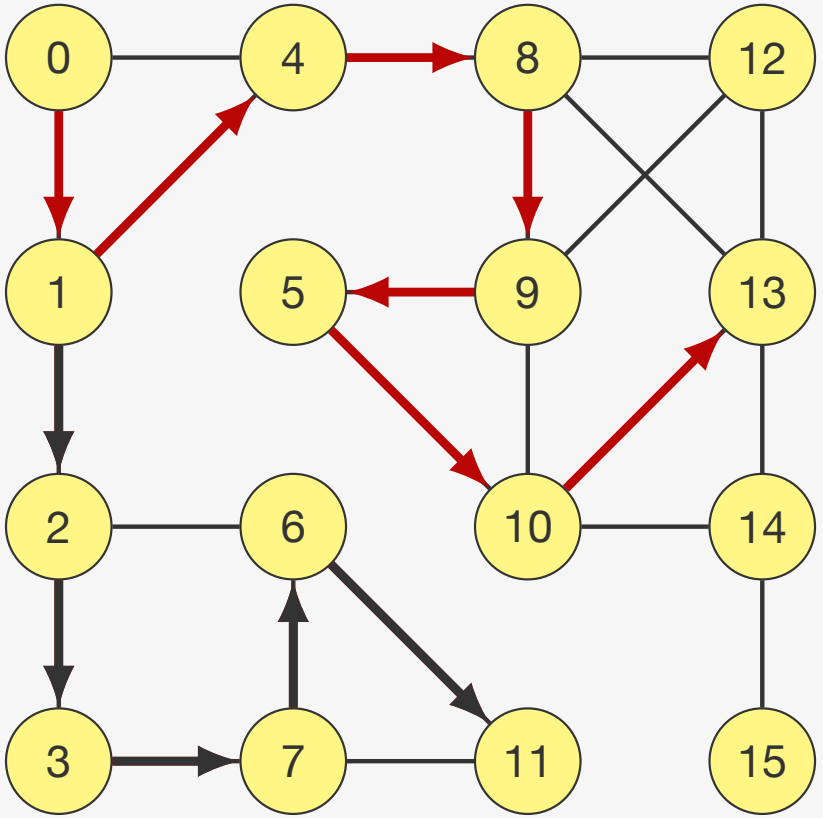
Caminhos de s para outros vértices da componente



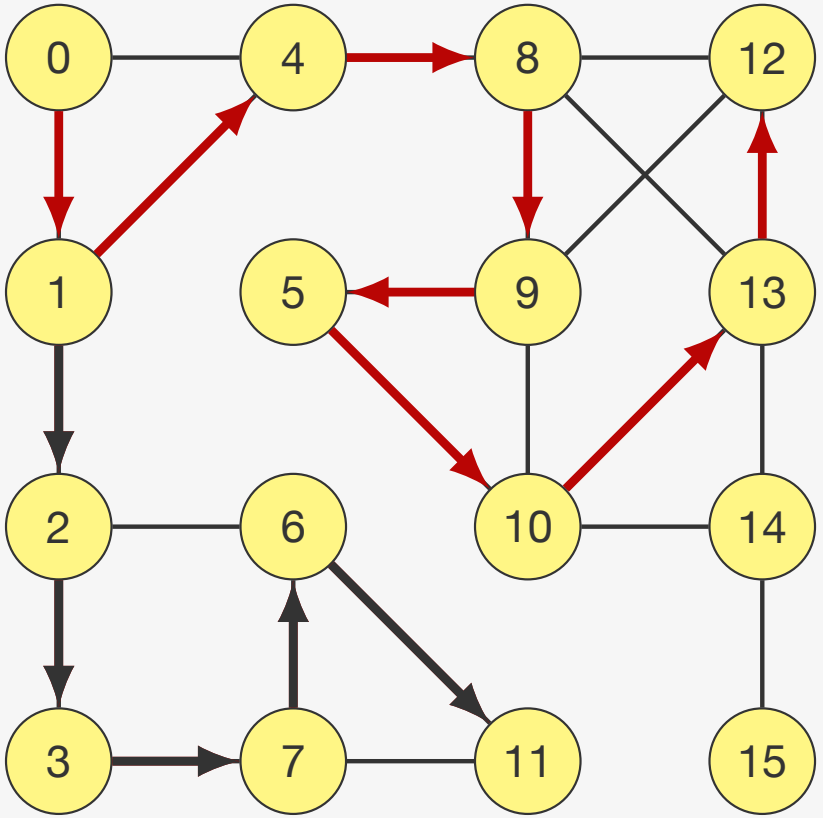
Caminhos de s para outros vértices da componente



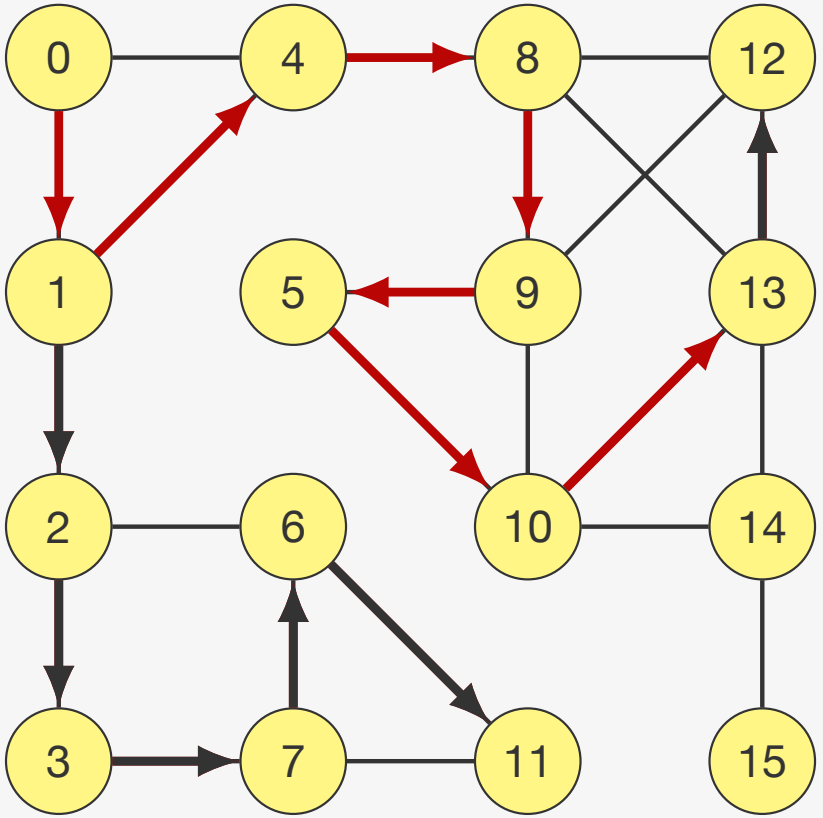
Caminhos de s para outros vértices da componente



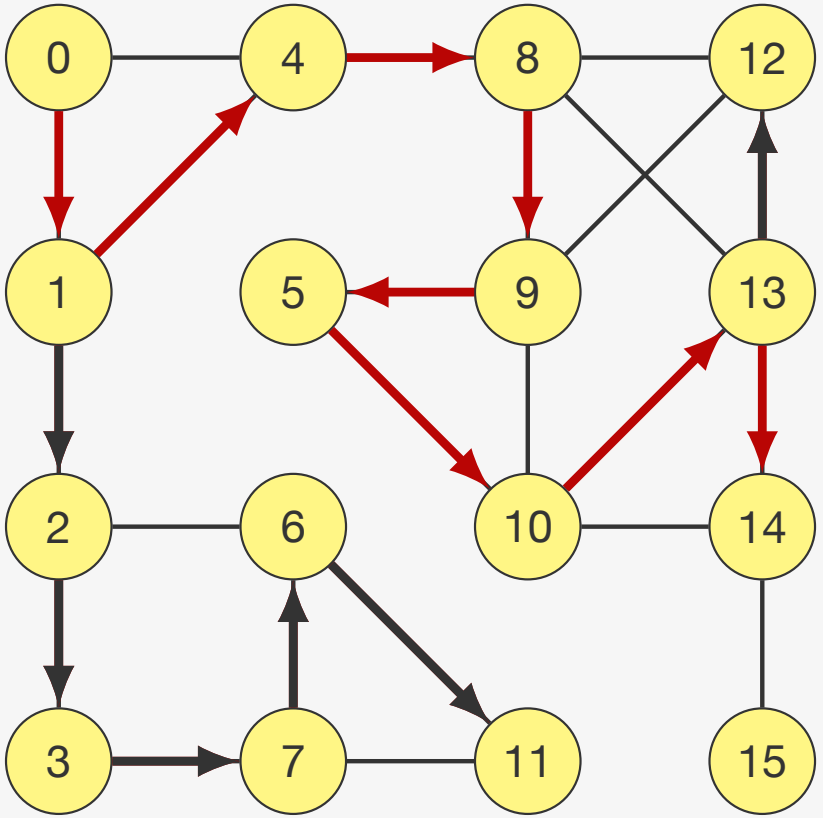
Caminhos de s para outros vértices da componente



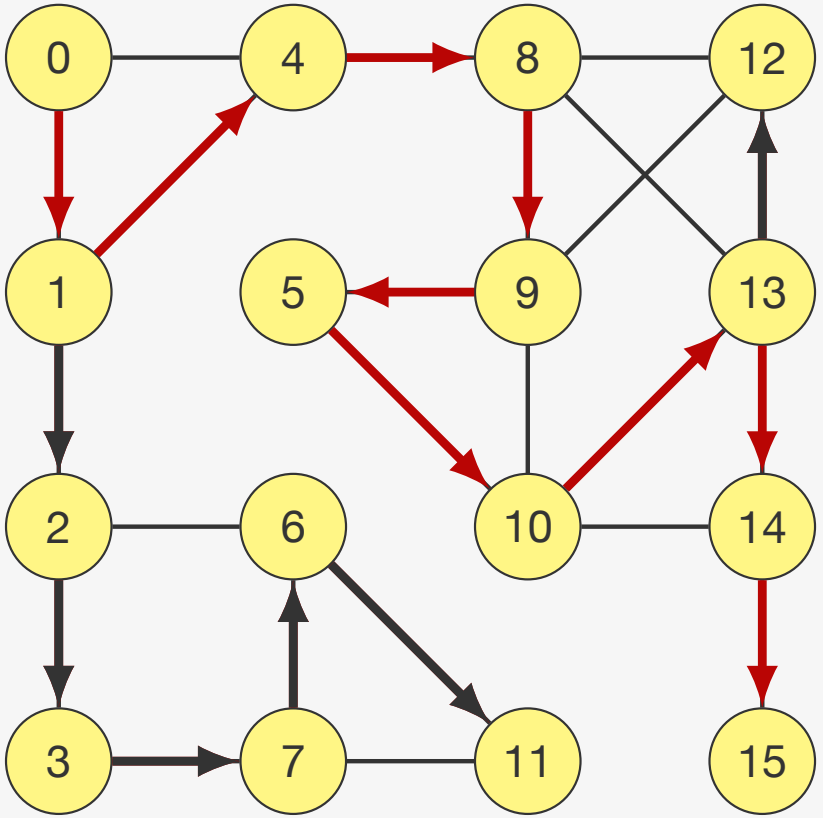
Caminhos de s para outros vértices da componente



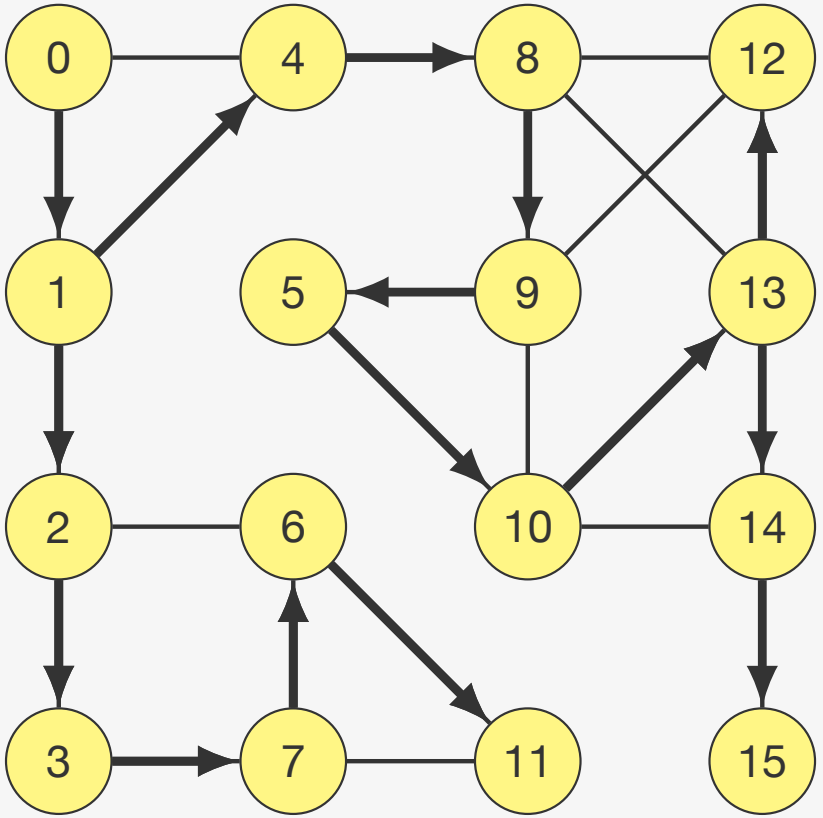
Caminhos de s para outros vértices da componente



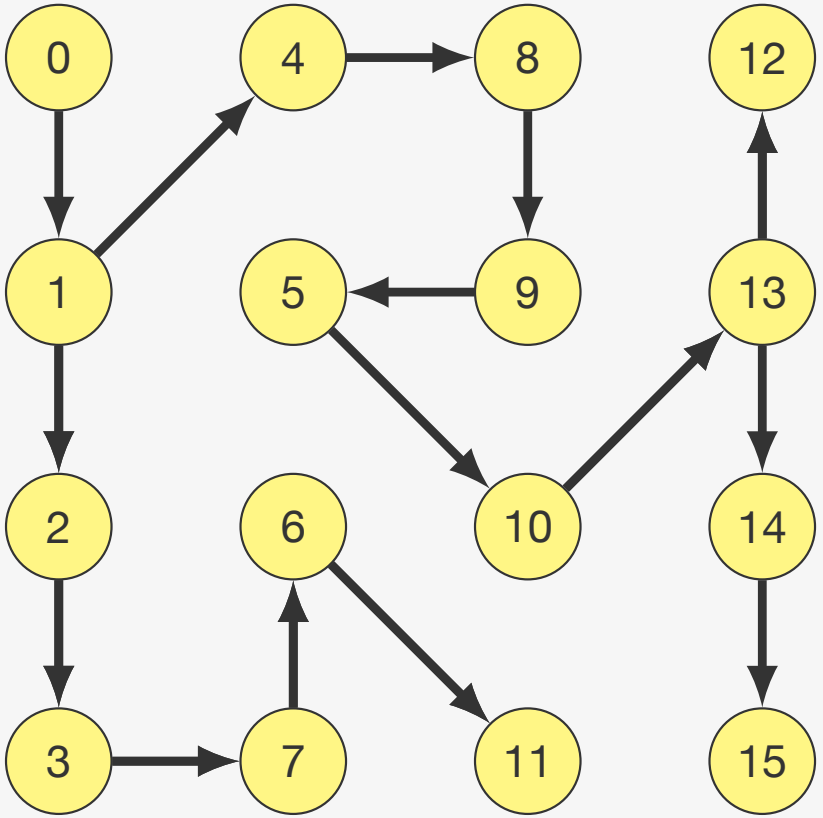
Caminhos de s para outros vértices da componente



Caminhos de s para outros vértices da componente

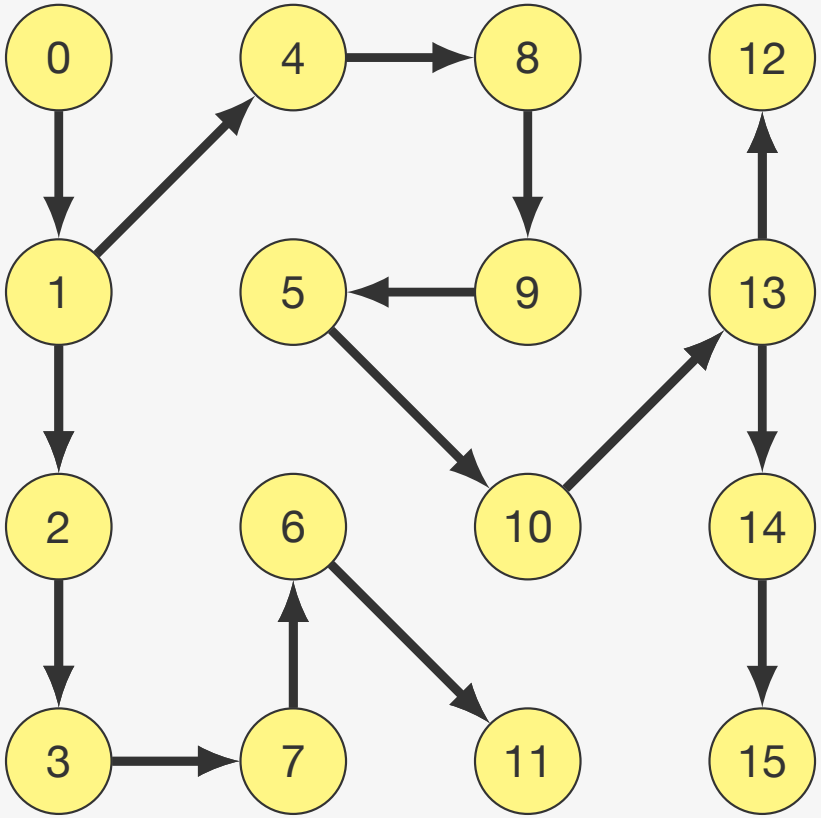


Caminhos de s para outros vértices da componente



As arestas usadas formam uma árvore!

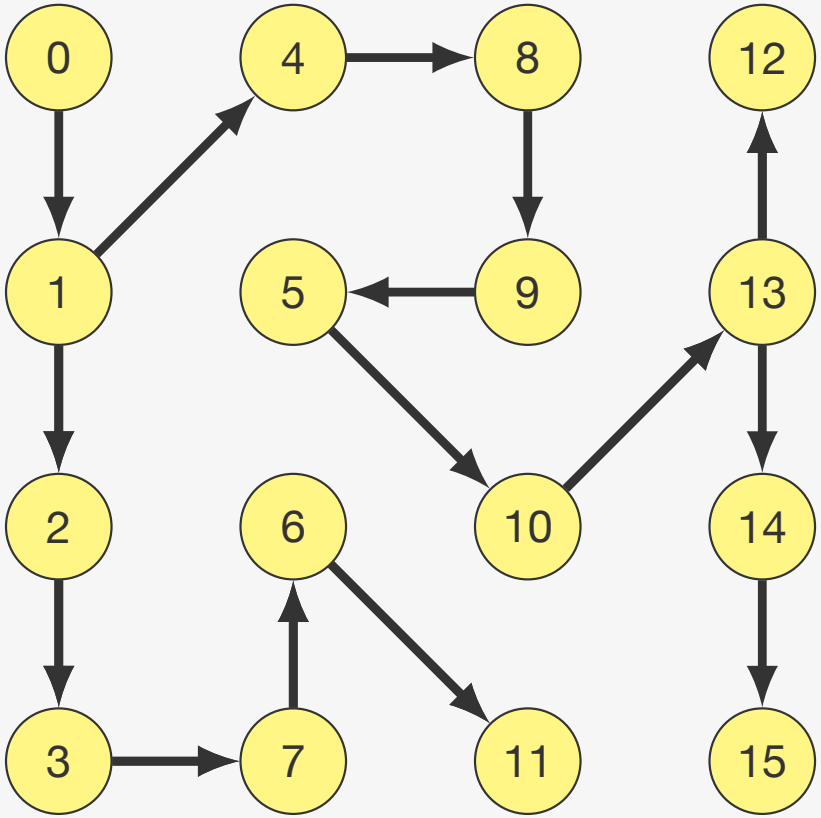
Caminhos de s para outros vértices da componente



As arestas usadas formam uma árvore!

- Essa árvore dá um caminho de qualquer vértice até a raiz

Caminhos de s para outros vértices da componente



As arestas usadas formam uma árvore!

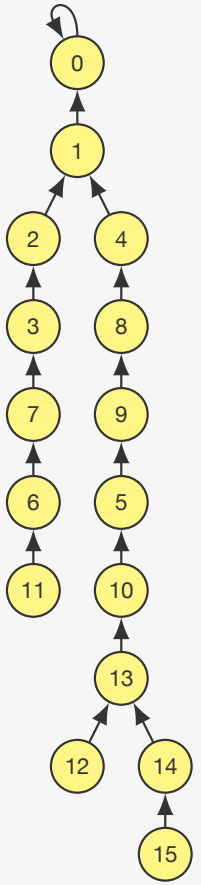
- Essa árvore dá um caminho de qualquer vértice até a raiz
- Basta ir subindo na árvore

Caminhos de s para outros vértices da componente

```
1 int * encontra_caminhos(p_grafo g, int s) {
2   int i, *pai = malloc(g->n * sizeof(int));
3   for (i = 0; i < g->n; i++)
4     pai[i] = -1;
5   busca_em_profundidade(g, pai, s, s);
6   return pai;
7 }
```

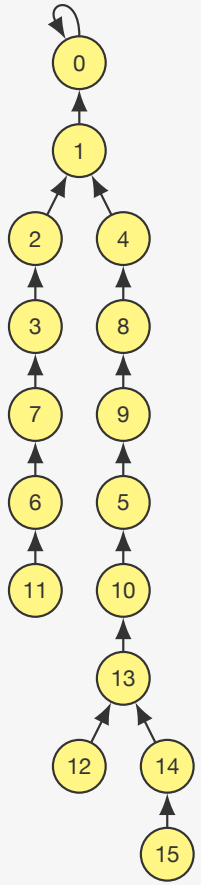
Caminhos de s para outros vértices da componente

```
1 int * encontra_caminhos(p_grafo g, int s) {
2   int i, *pai = malloc(g->n * sizeof(int));
3   for (i = 0; i < g->n; i++)
4     pai[i] = -1;
5   busca_em_profundidade(g, pai, s, s);
6   return pai;
7 }
```



Caminhos de s para outros vértices da componente

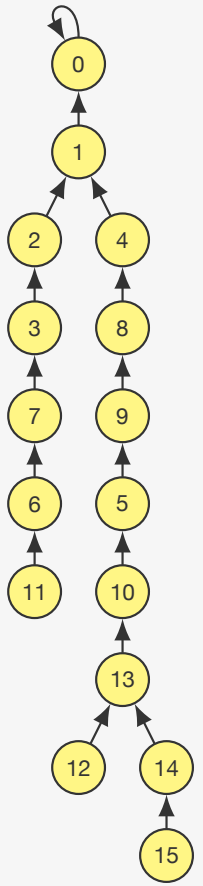
```
1 int * encontra_caminhos(p_grafo g, int s) {
2   int i, *pai = malloc(g->n * sizeof(int));
3   for (i = 0; i < g->n; i++)
4     pai[i] = -1;
5   busca_em_profundidade(g, pai, s, s);
6   return pai;
7 }
```




```
1 void busca_em_profundidade(p_grafo g, int *pai, int p, int v) {
2   p_no t;
3   pai[v] = p;
4   for(t = g->adj[v]; t != NULL; t = t->prox)
5     if (pai[t->v] == -1)
6       busca_em_profundidade(g, pai, v, t->v);
7 }
```

Imprimindo o caminho

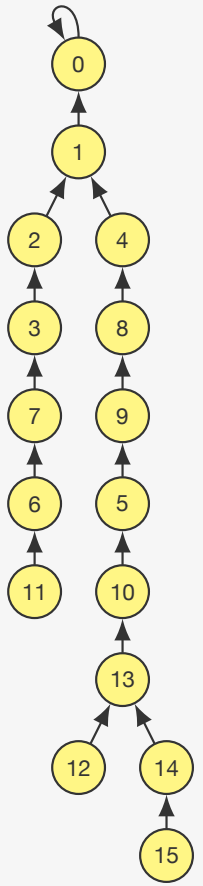
```
1 void imprimi_caminho_reverso(int v, int *pai) {
2   printf("%d", v);
3   if(pai[v] != v)
4     imprimi_caminho_reverso(pai[v], pai);
5 }
```



Imprimindo o caminho

```
1 void imprimi_caminho_reverso(int v, int *pai) {
2   printf("%d", v);
3   if(pai[v] != v)
4     imprimi_caminho_reverso(pai[v], pai);
5 }
```

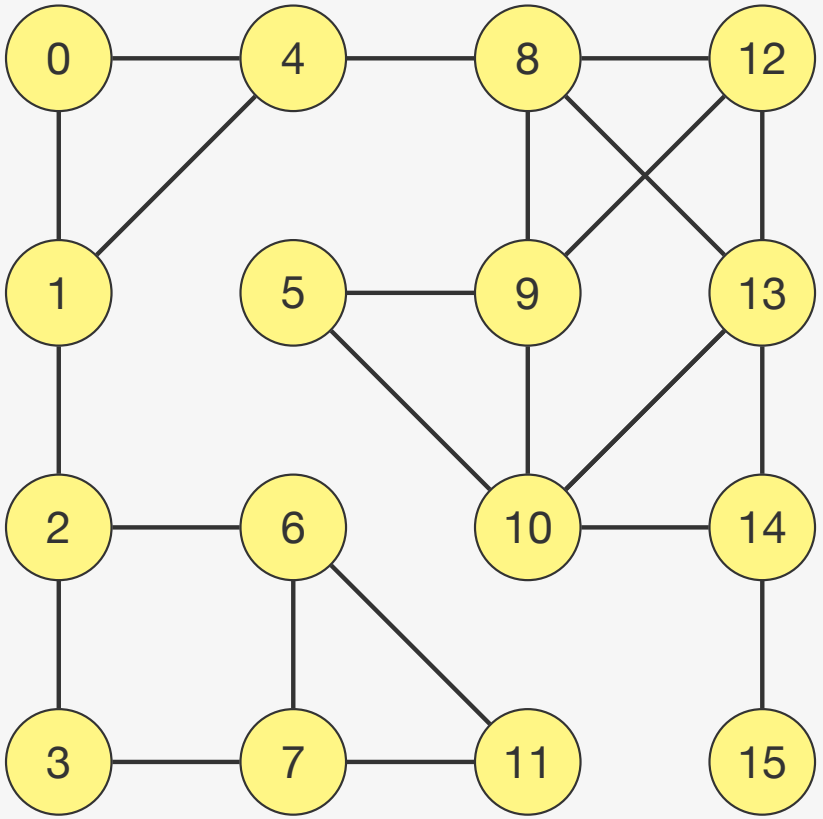
```
1 void imprimi_caminho(int v, int *pai) {
2   if(pai[v] != v)
3     imprimi_caminho(pai[v], pai);
4   printf("%d", v);
5 }
```



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

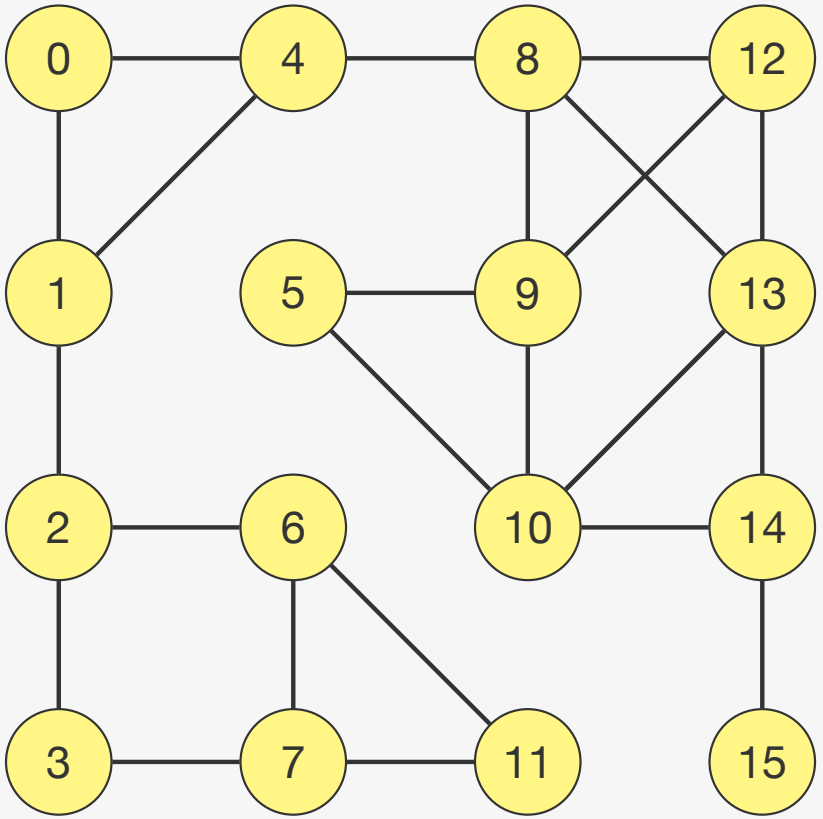


Pilha

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

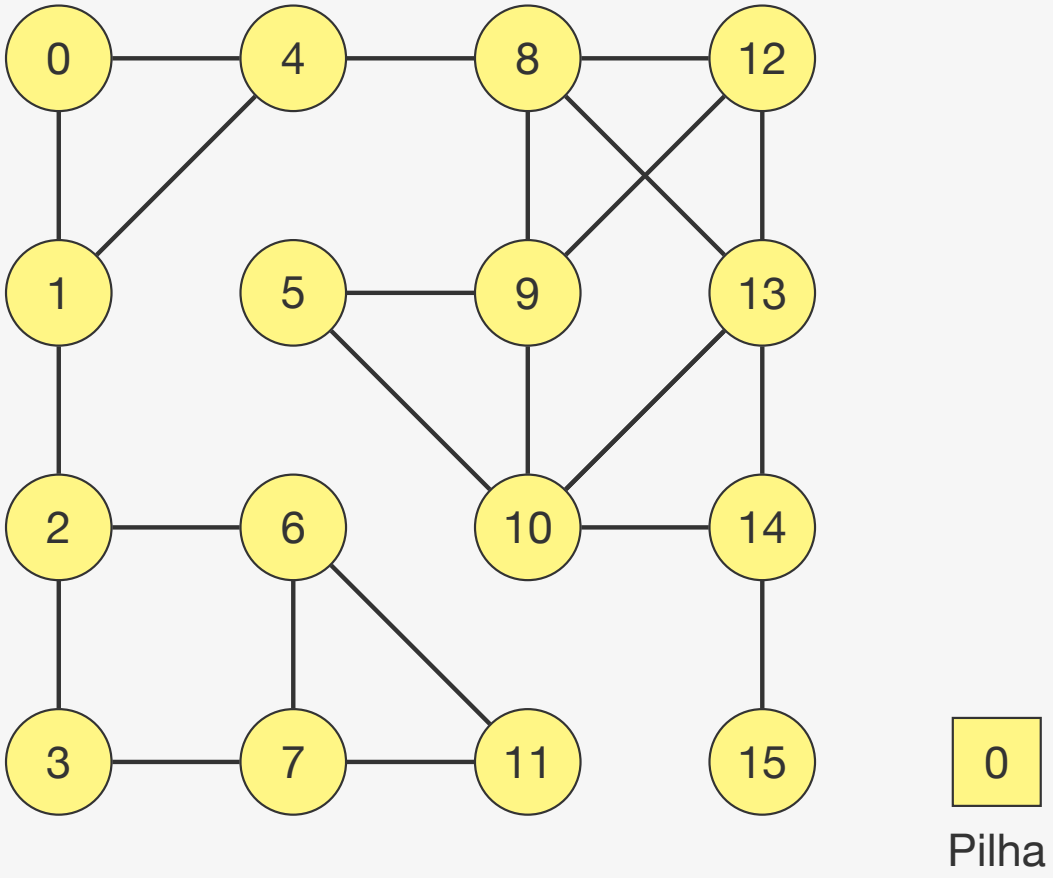


Pilha

Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

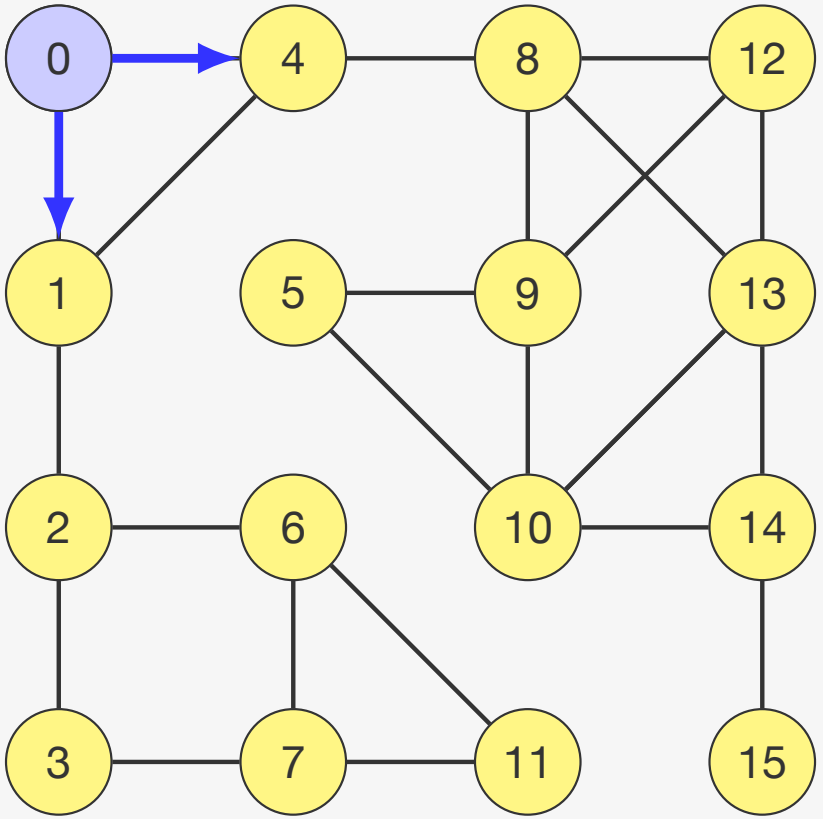
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



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

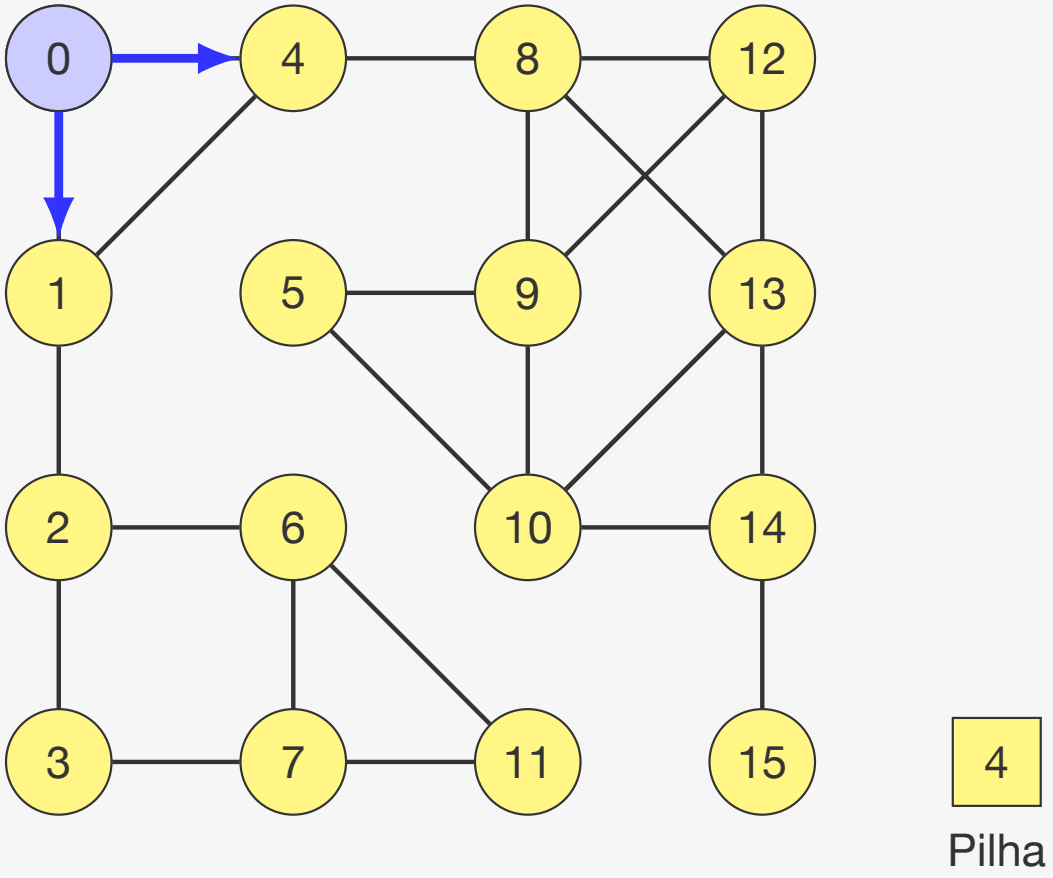


Pilha

Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

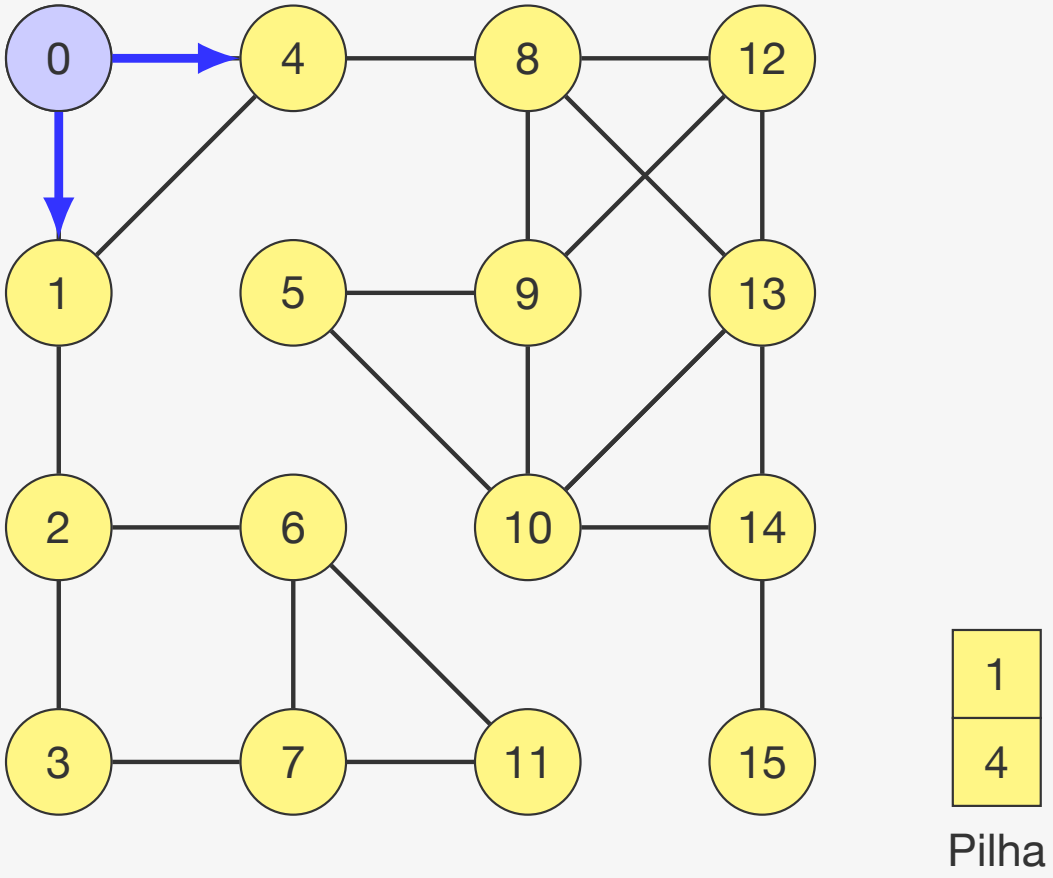
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

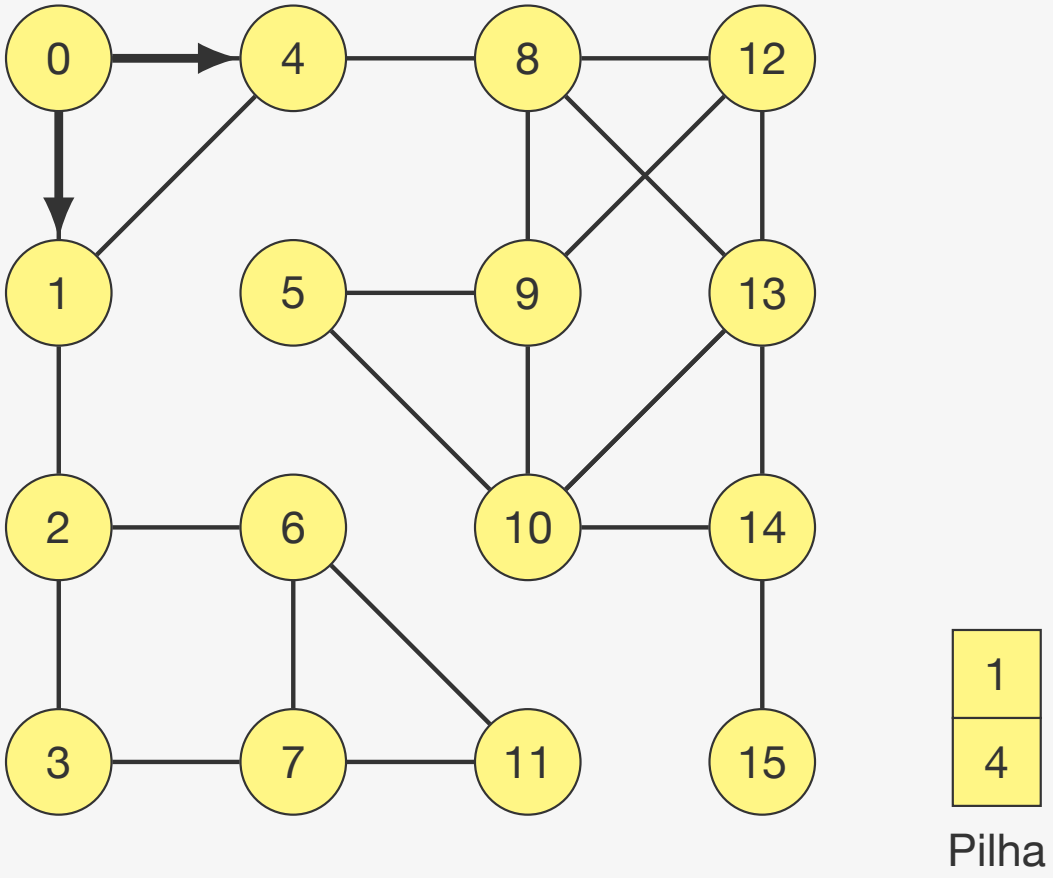
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

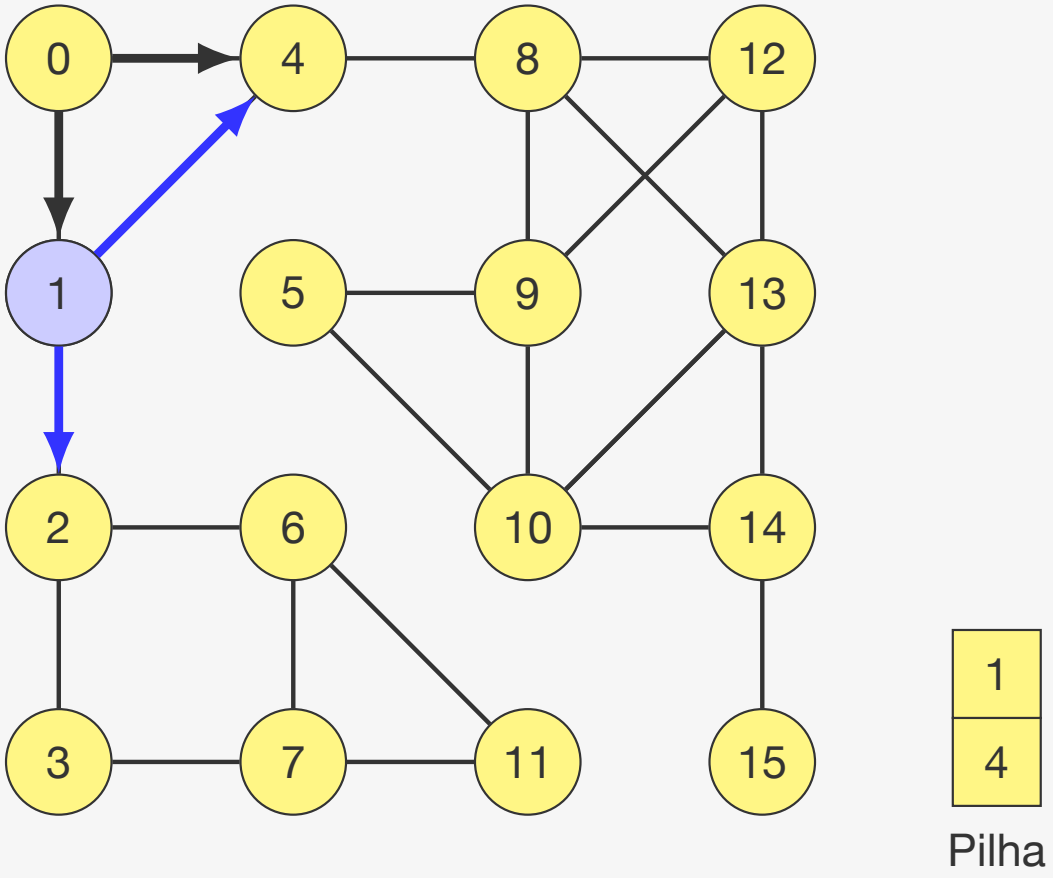
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

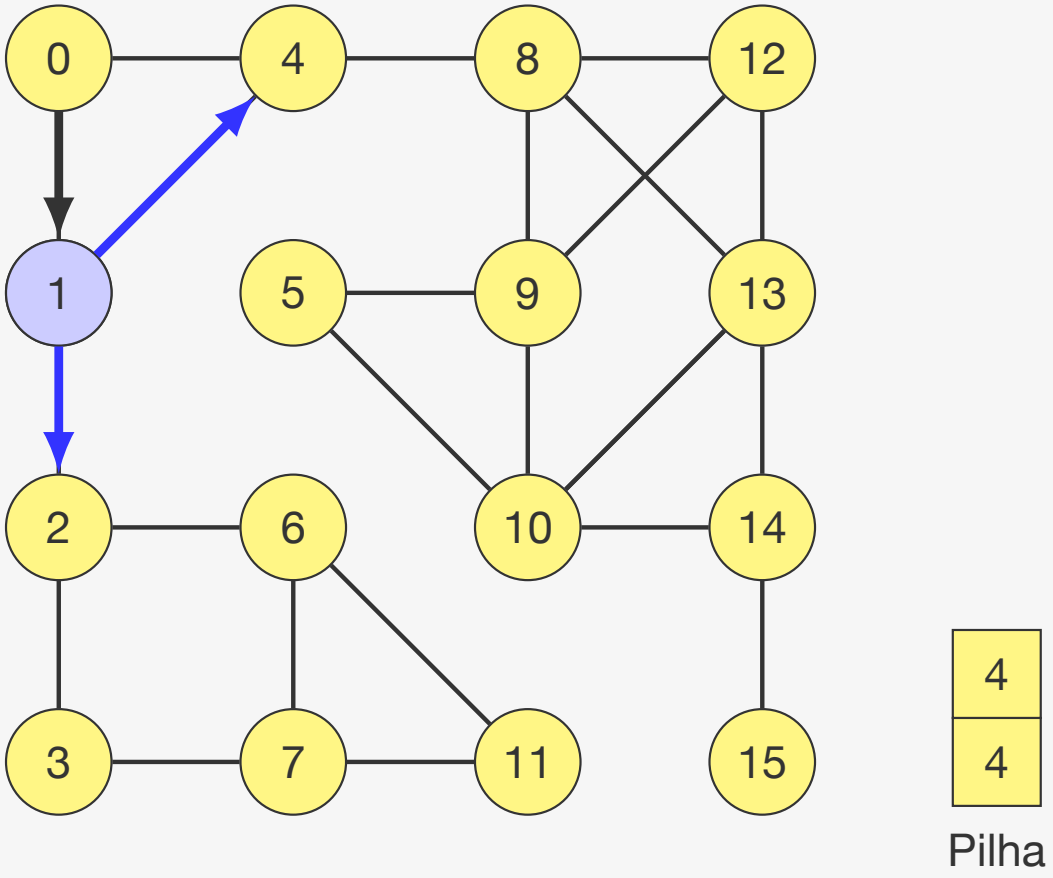
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

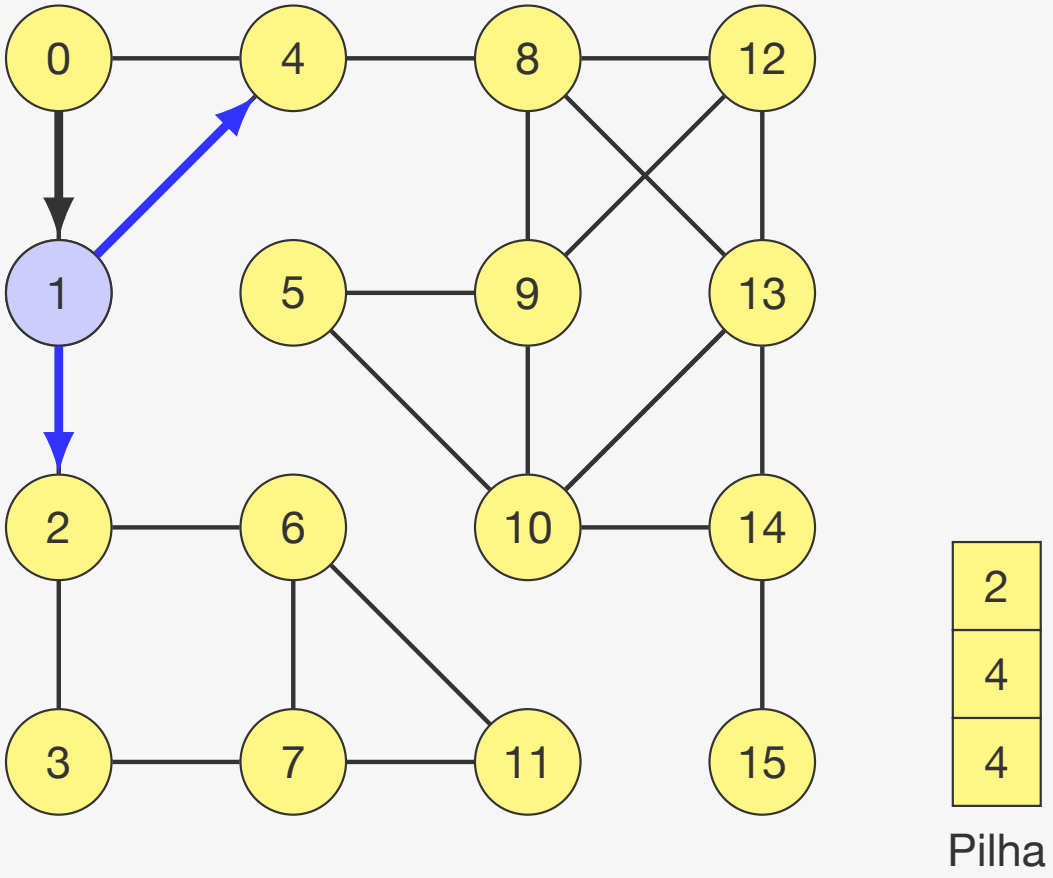
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

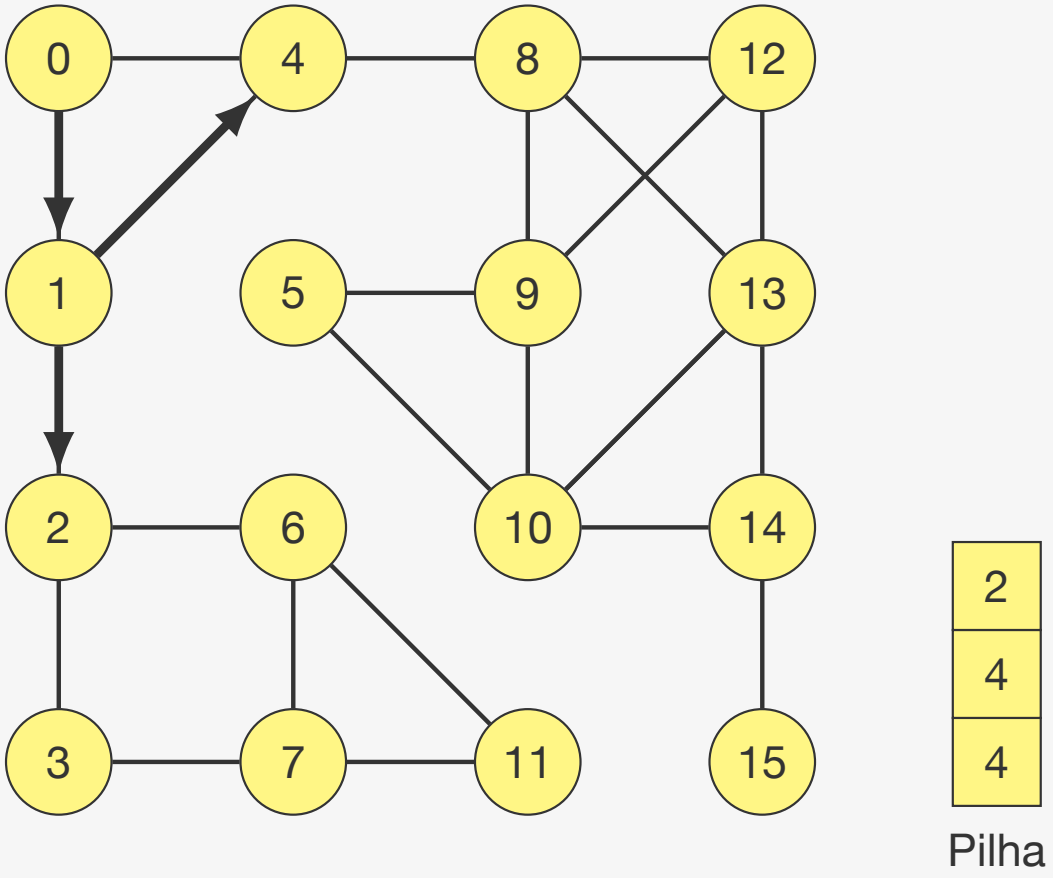
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

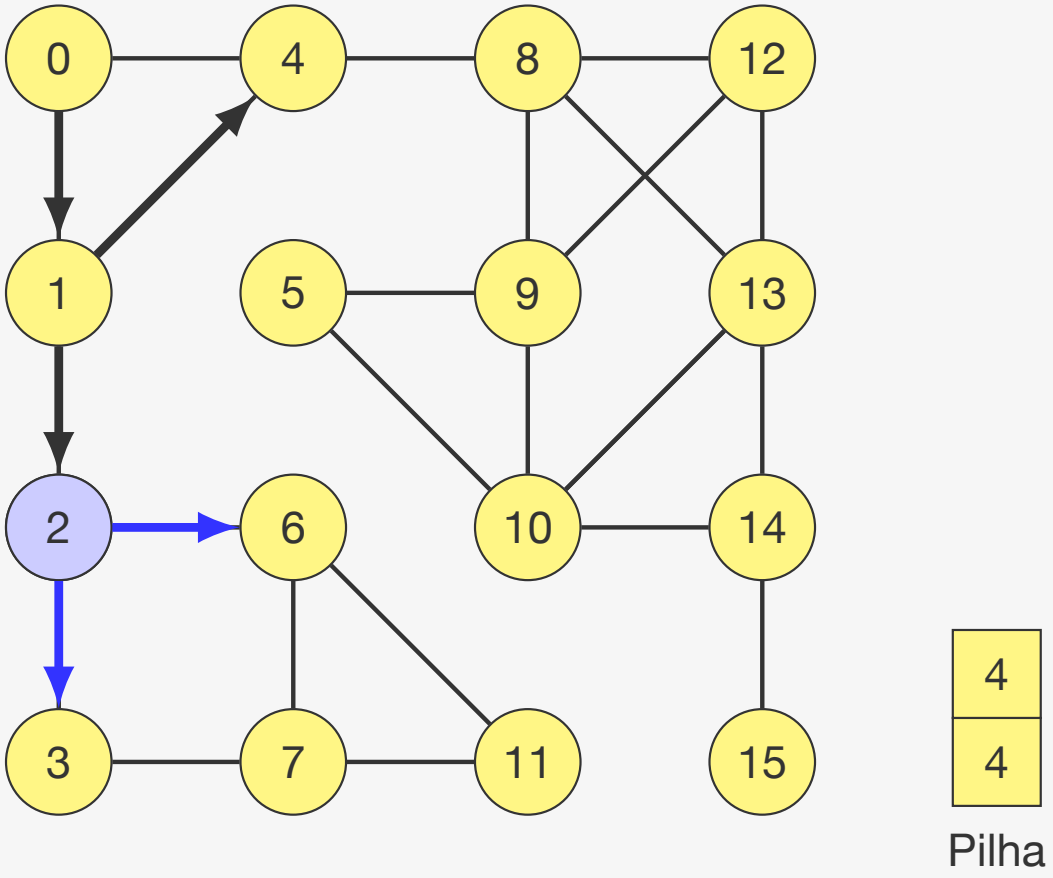
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

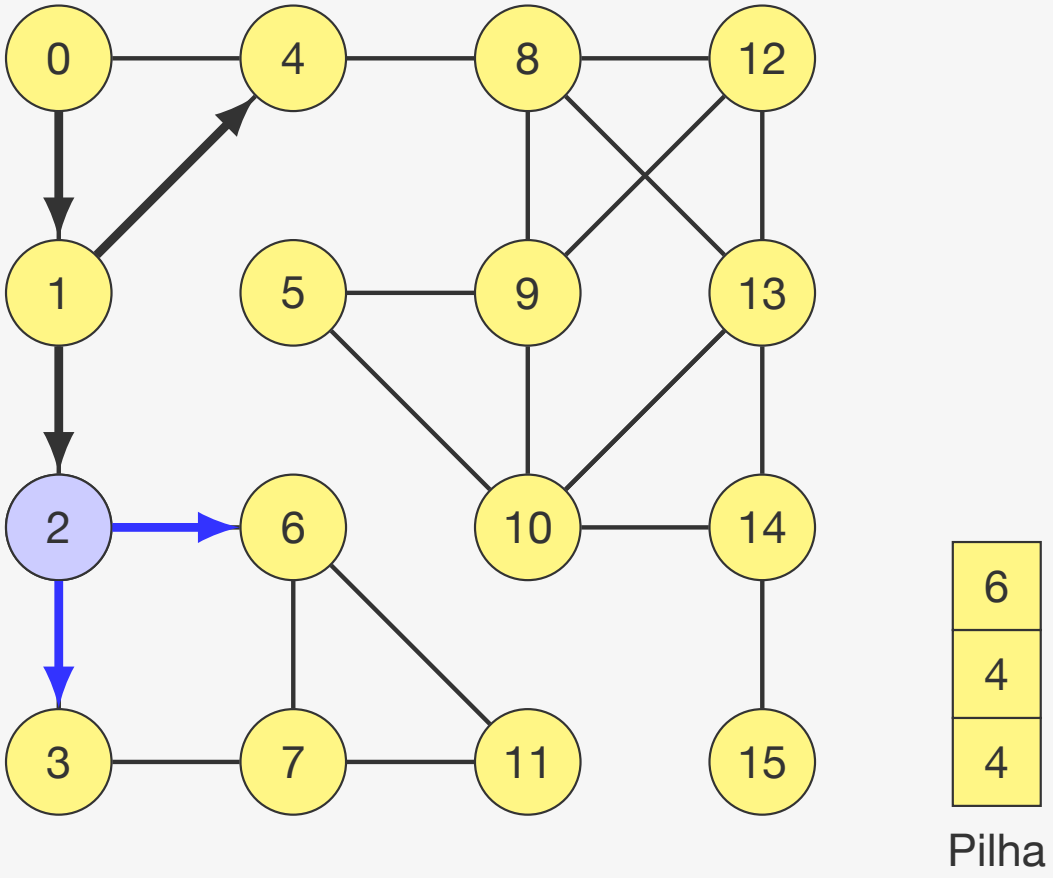
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

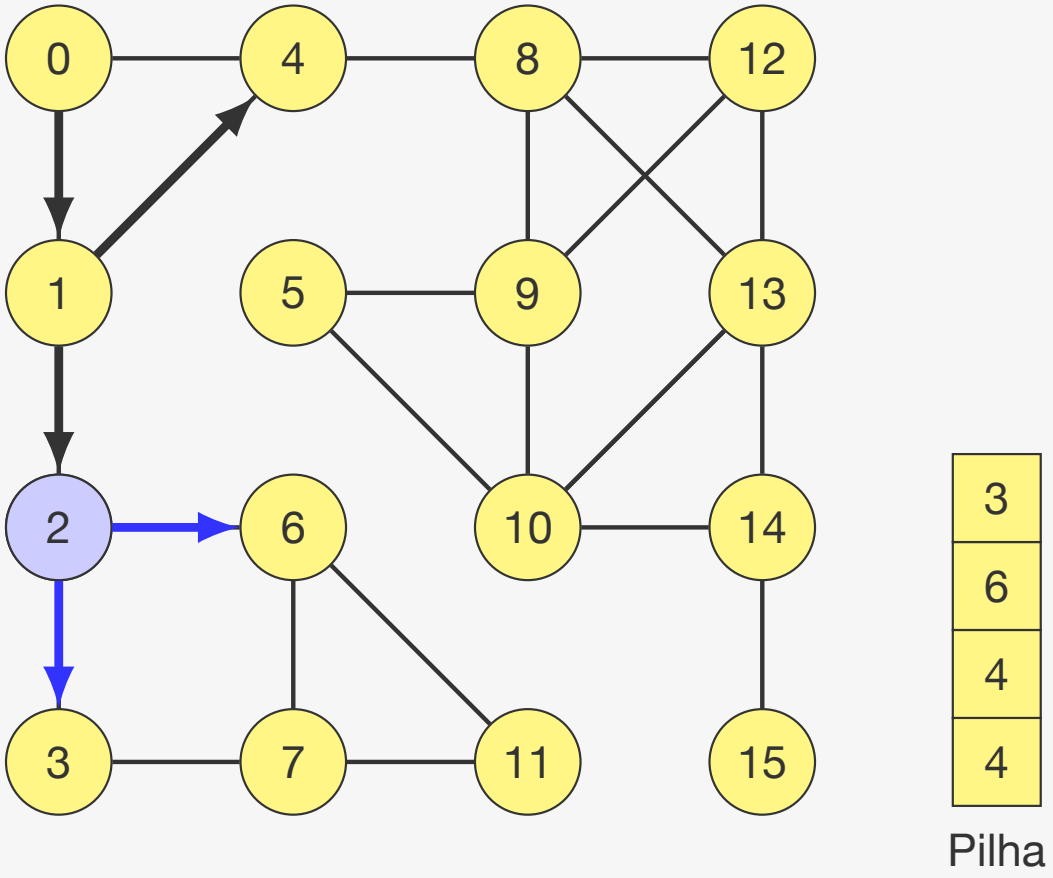
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

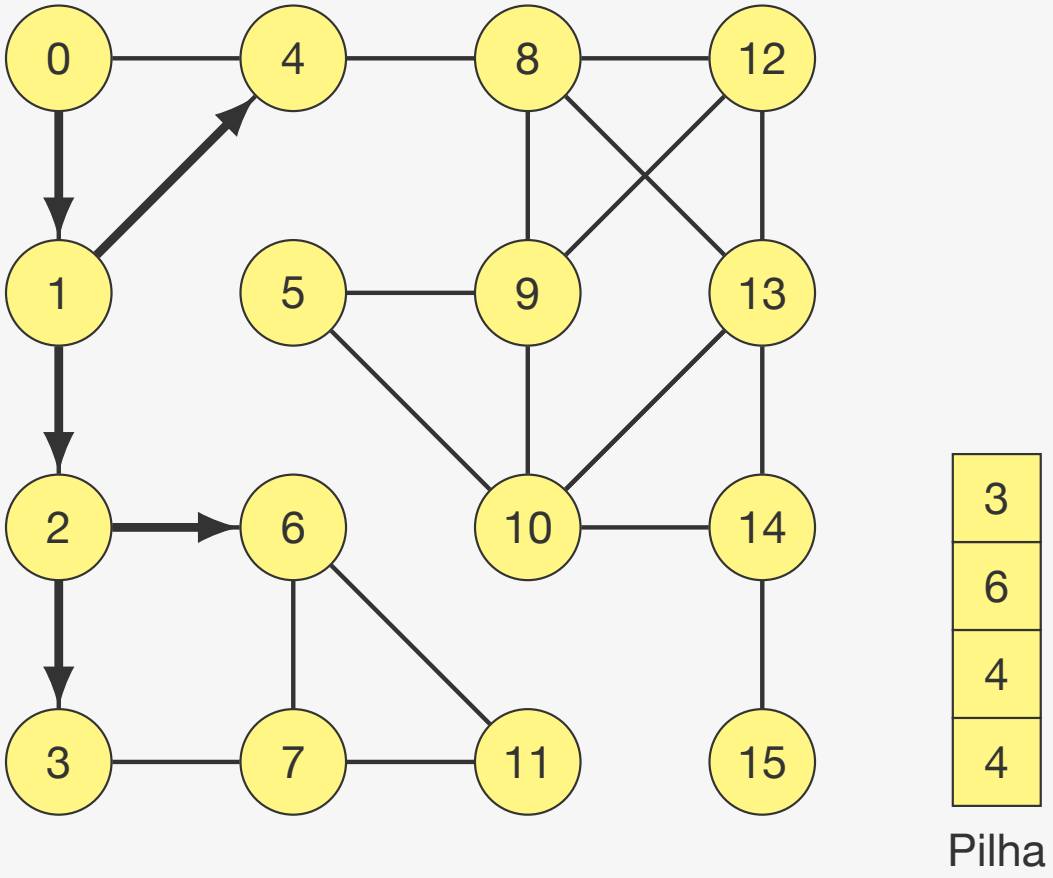
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

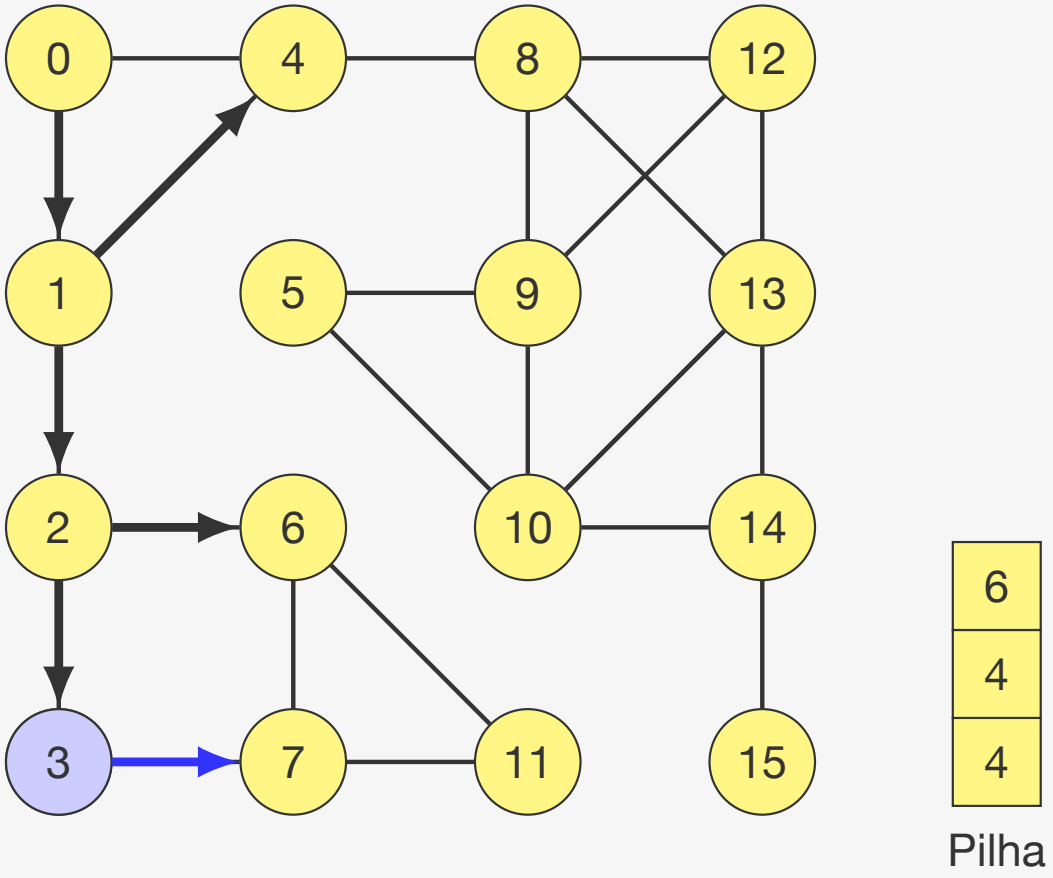
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

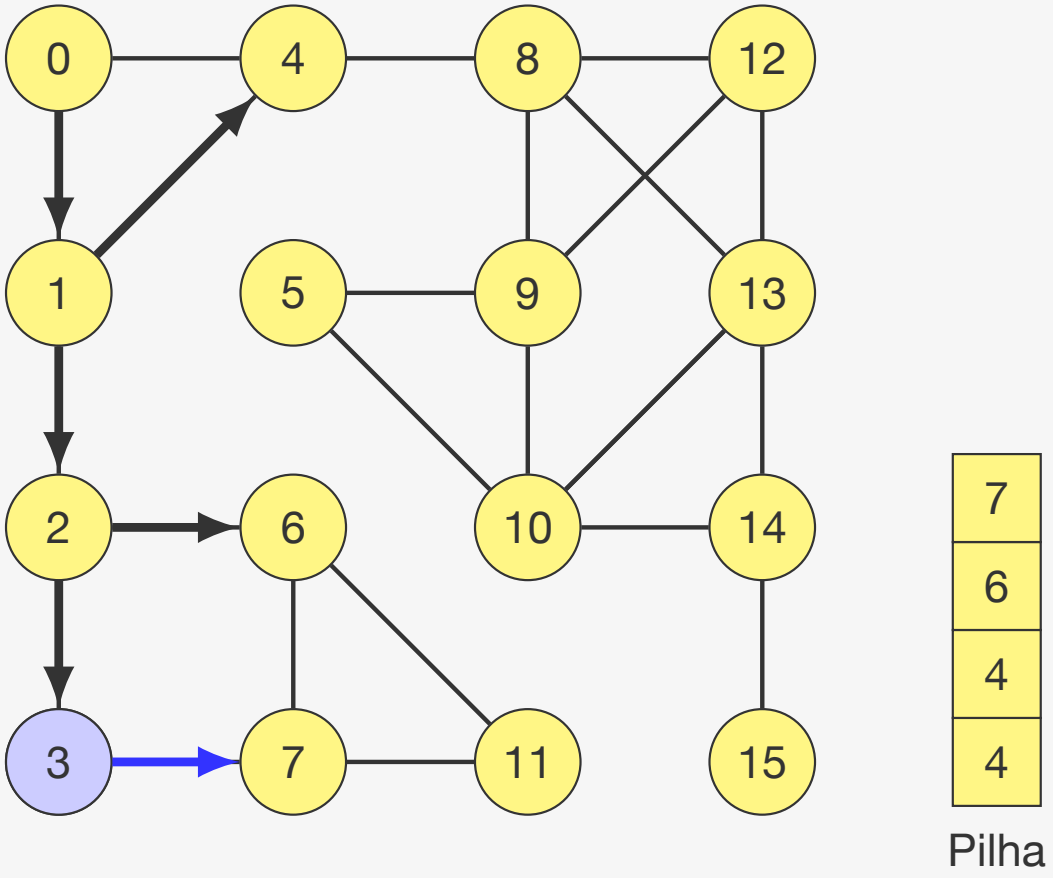
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

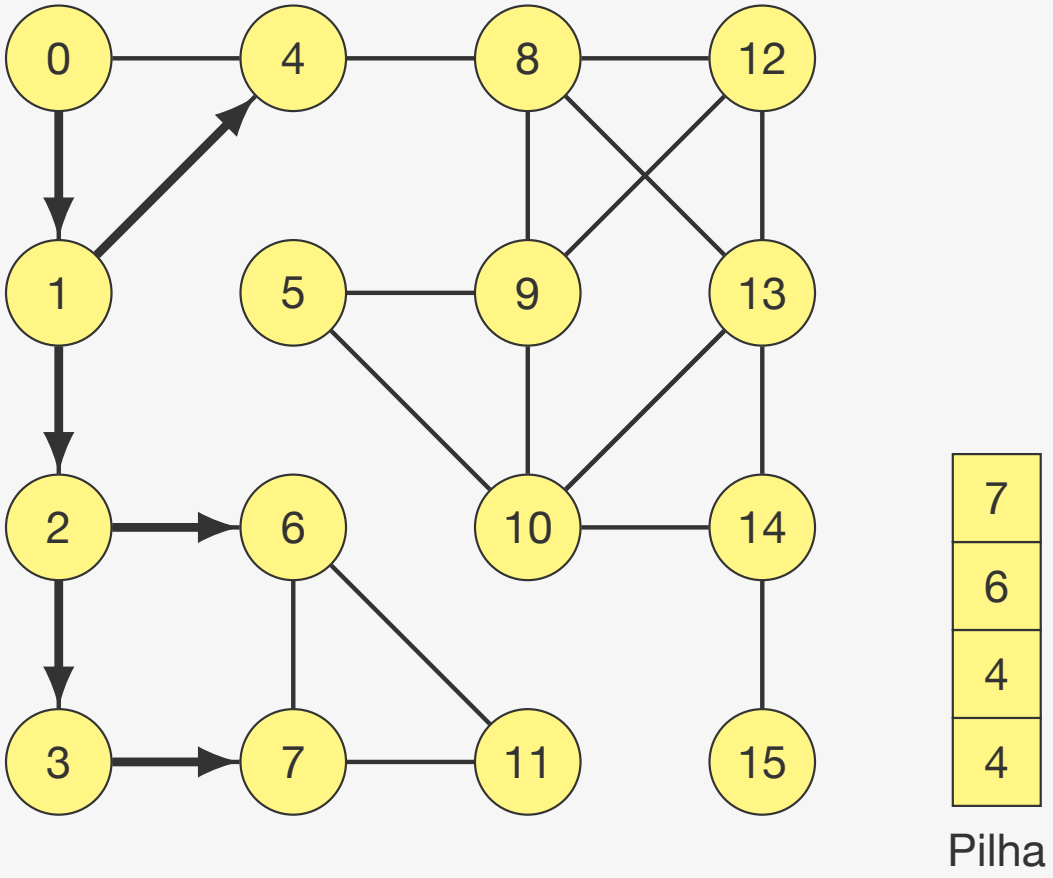
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

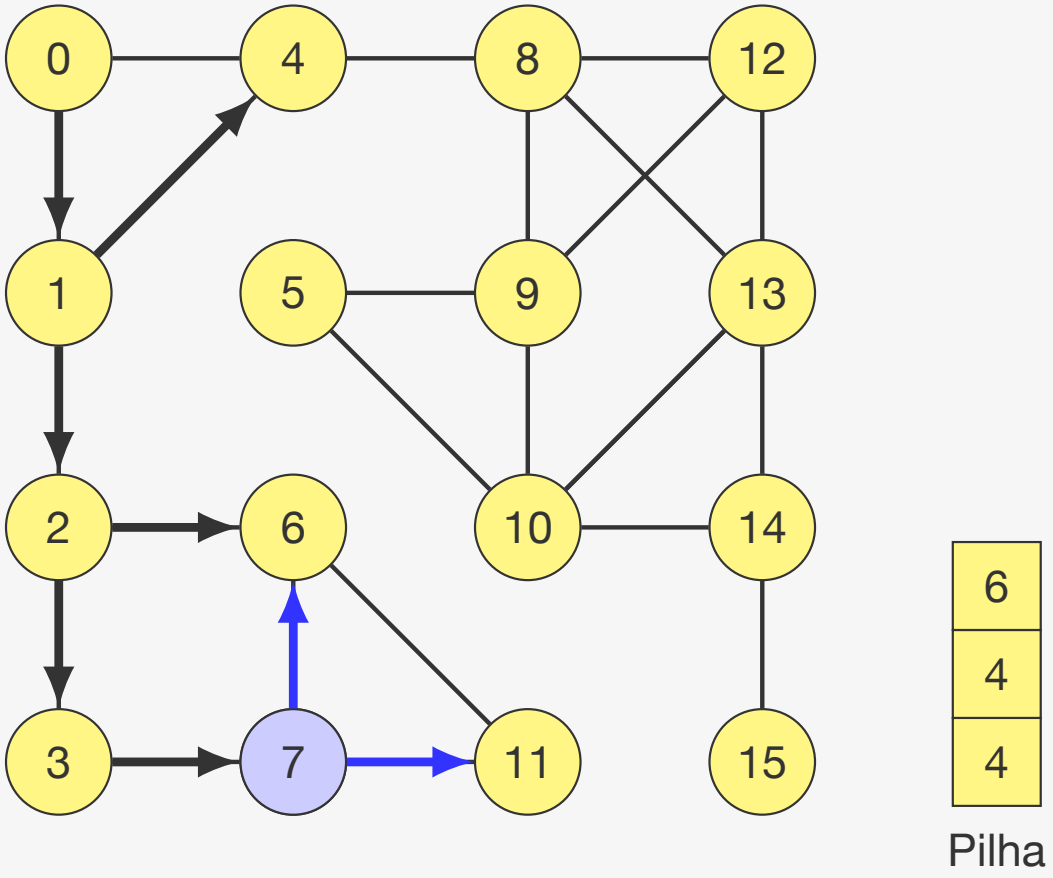
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

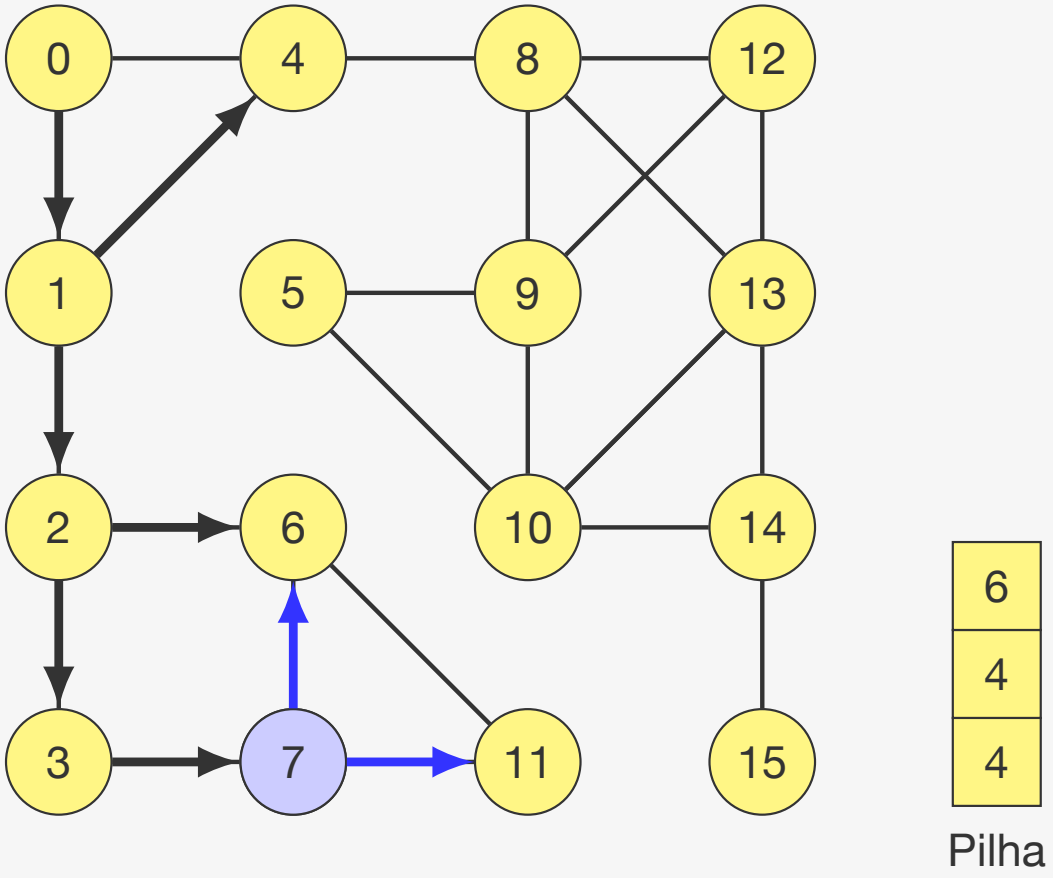
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

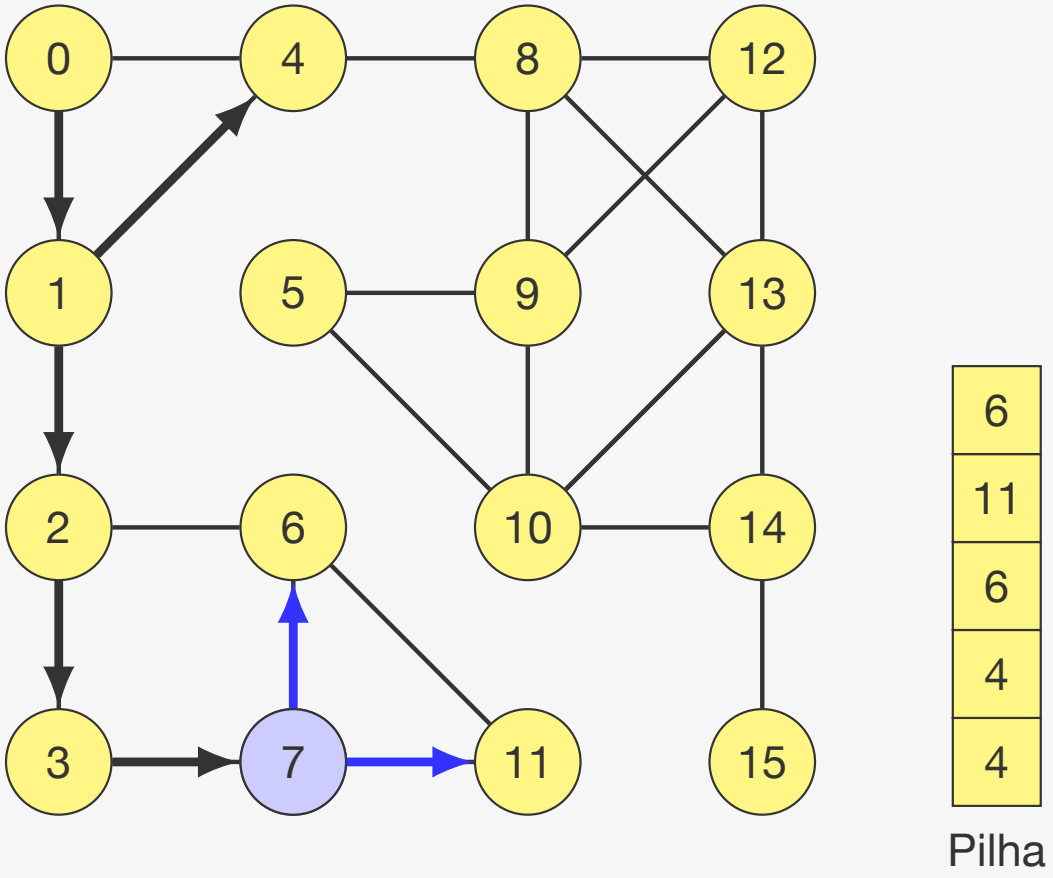
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

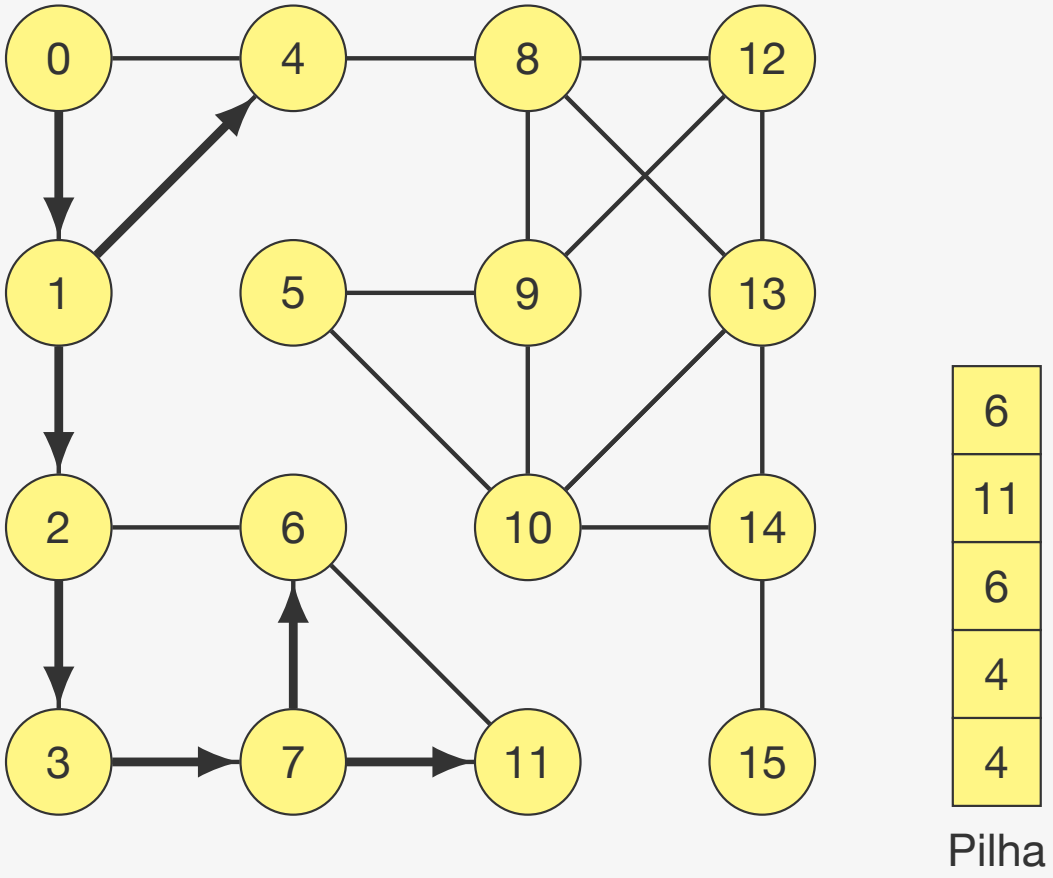
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

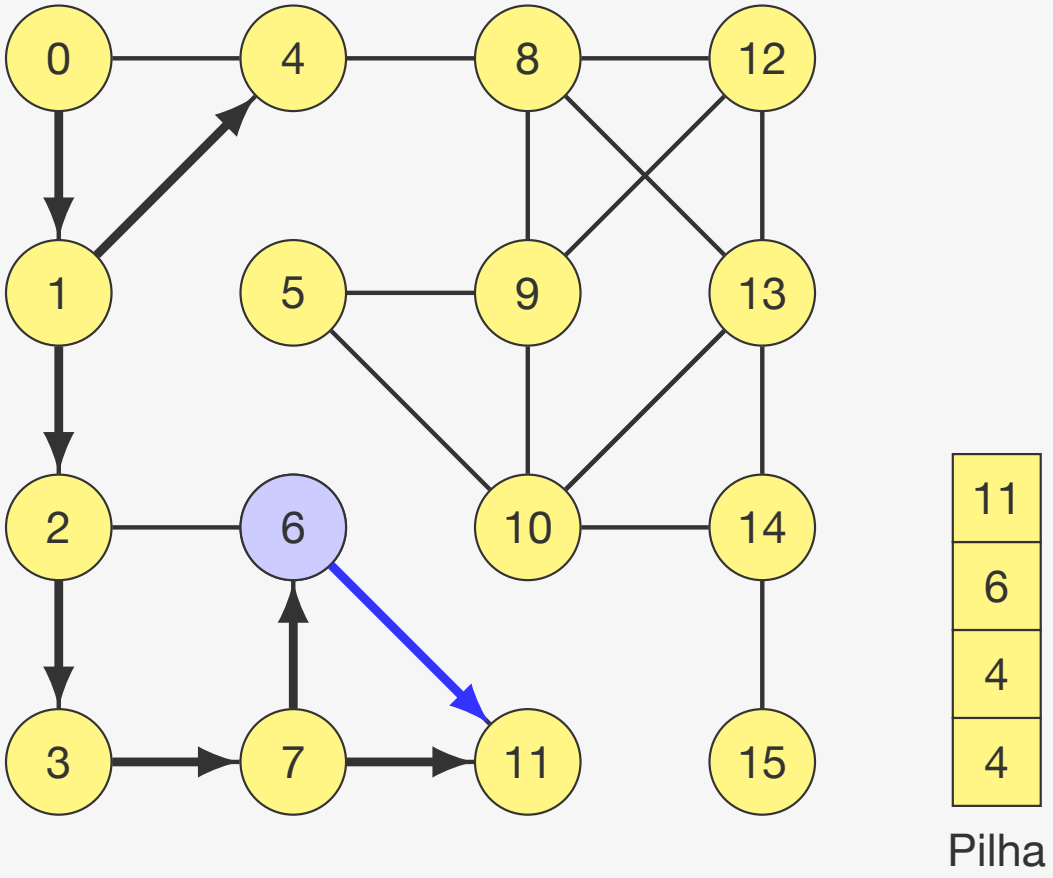
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

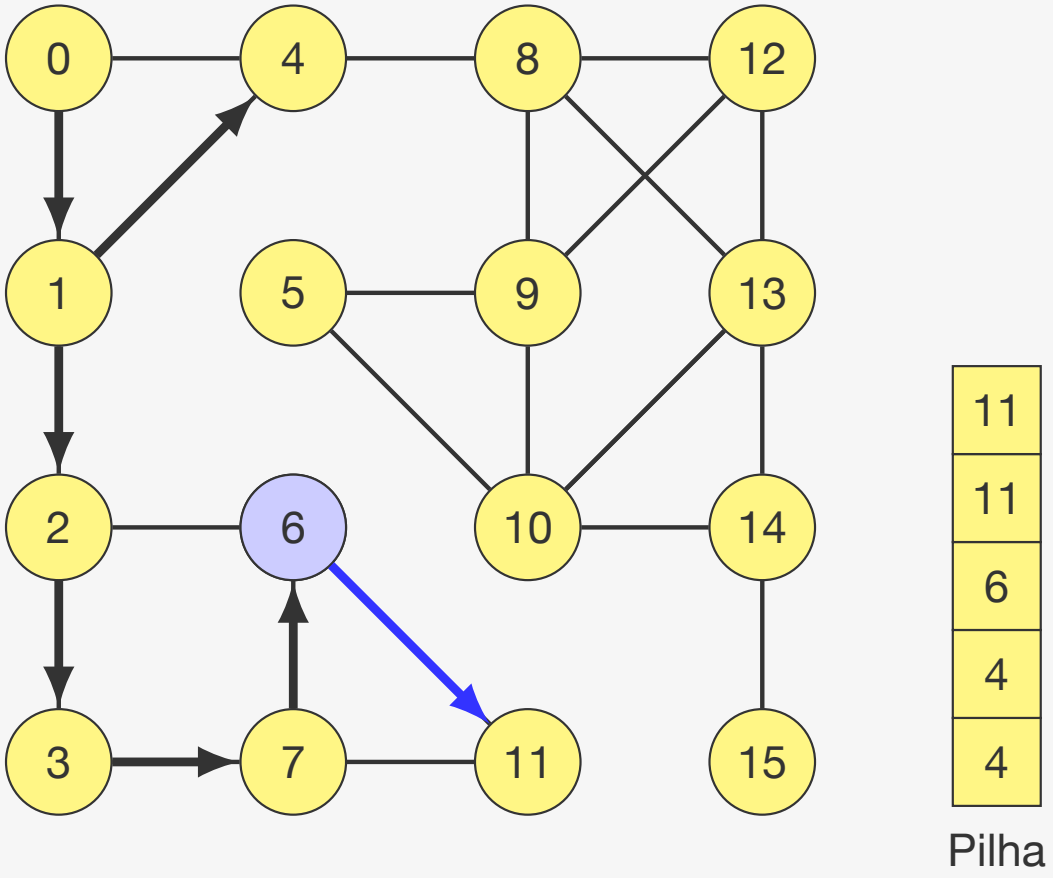
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

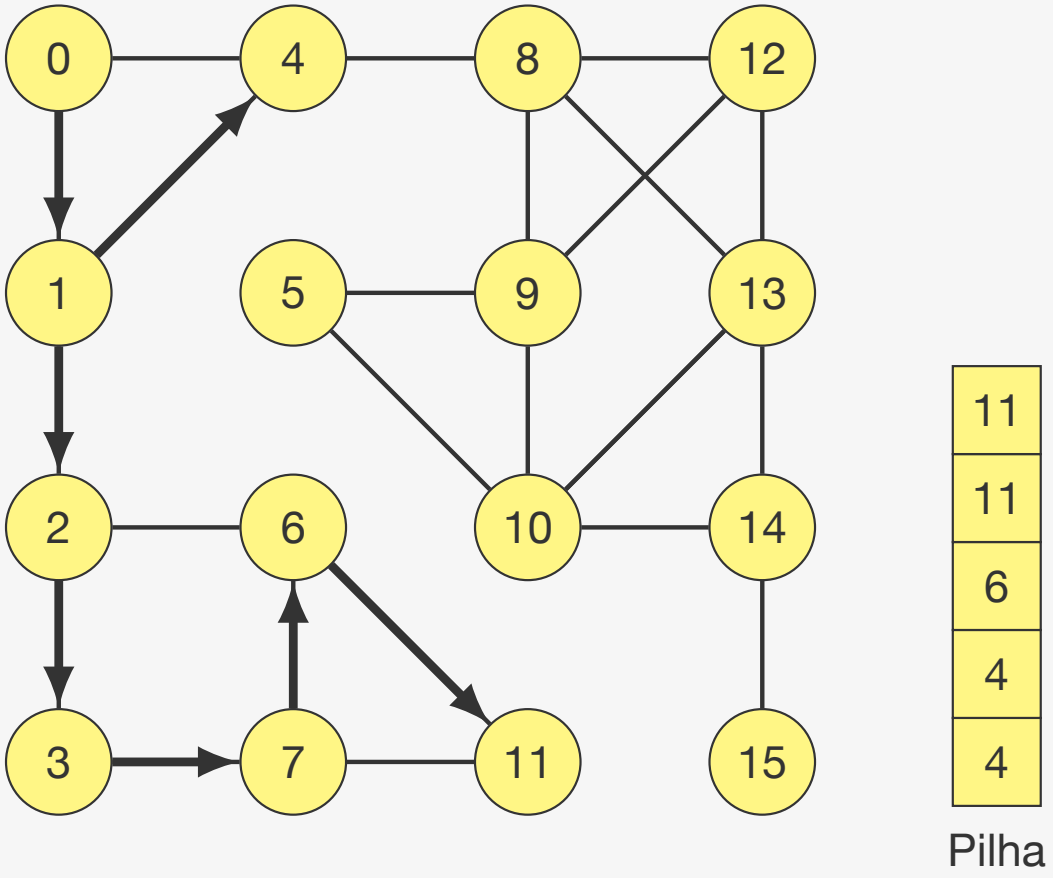
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

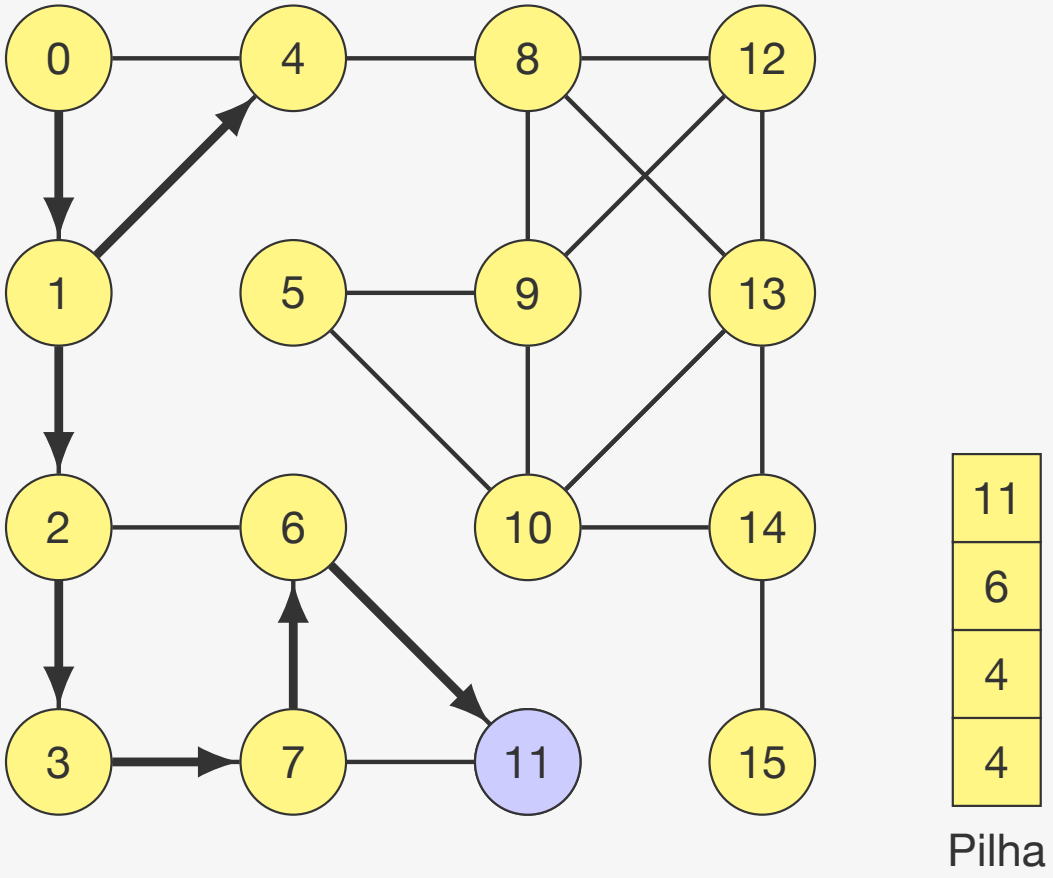
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

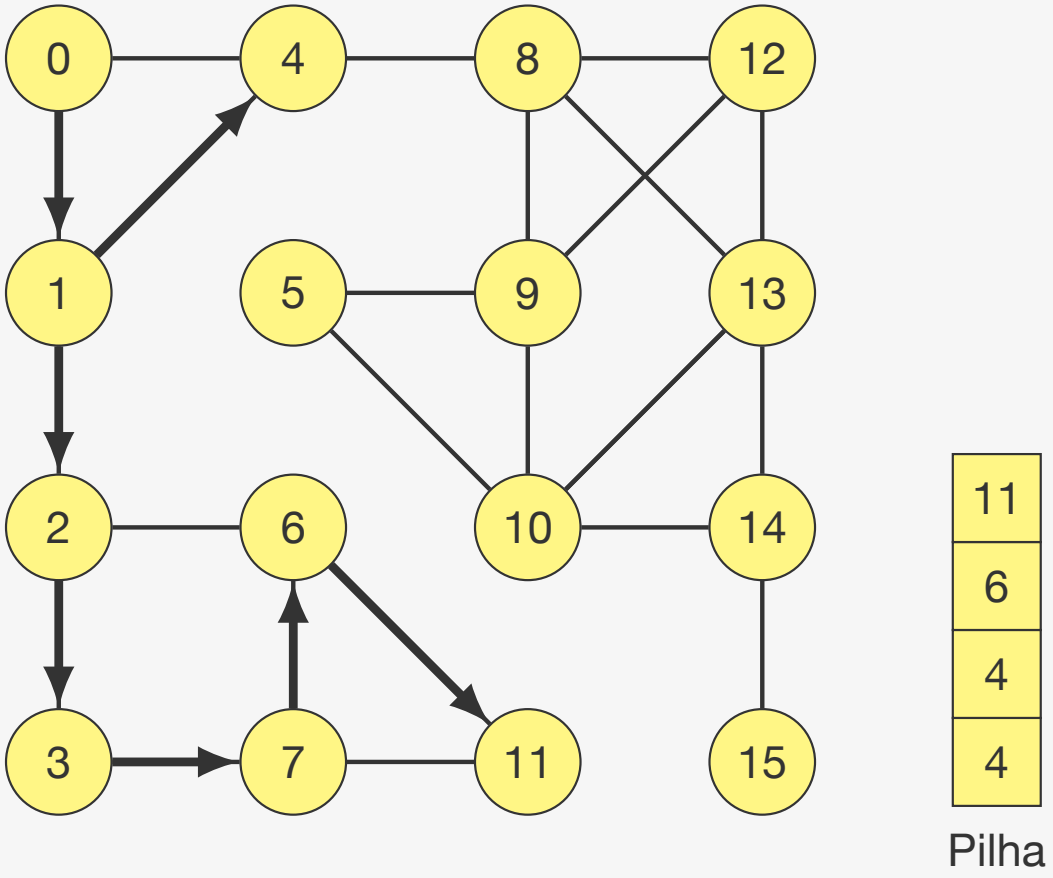
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

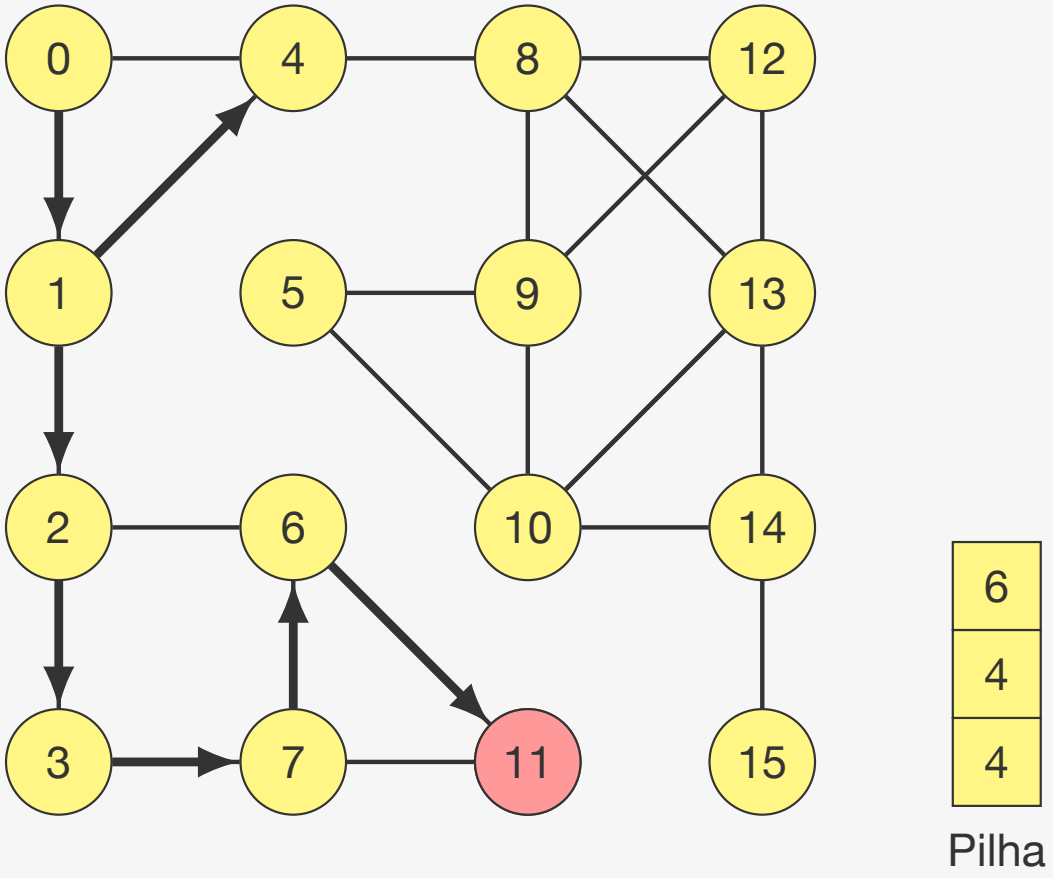
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

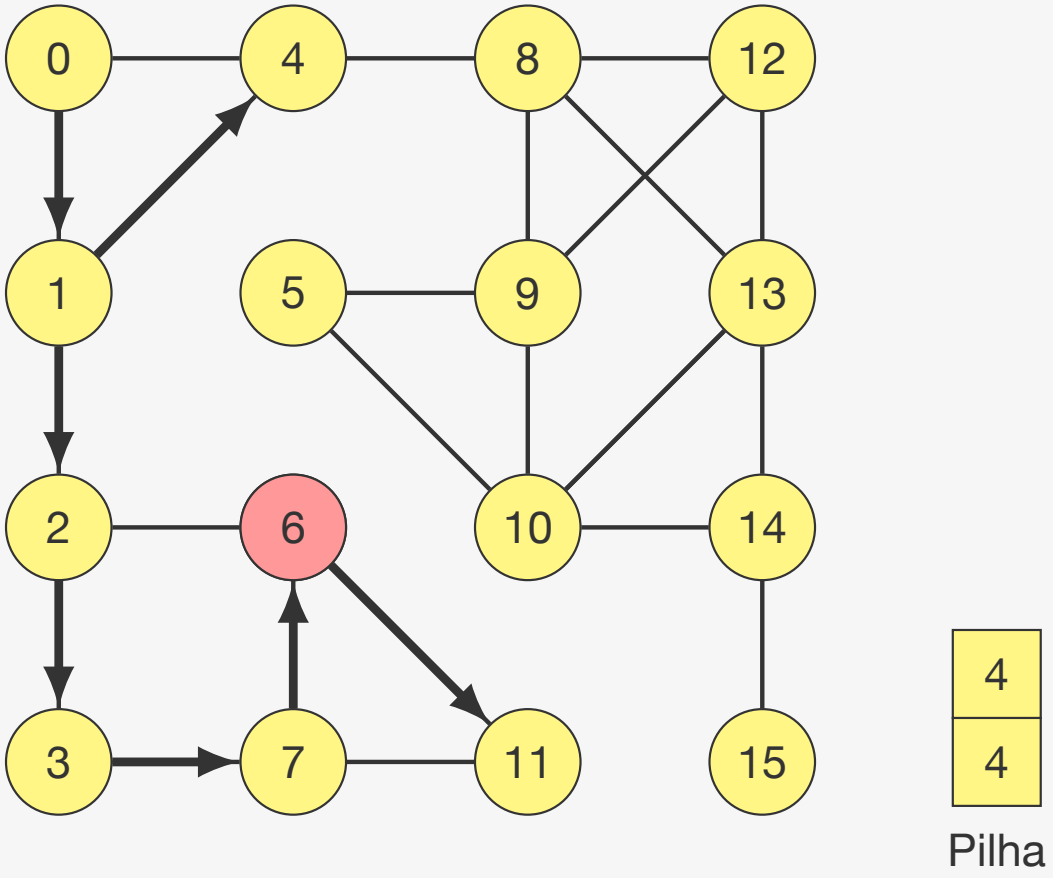
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

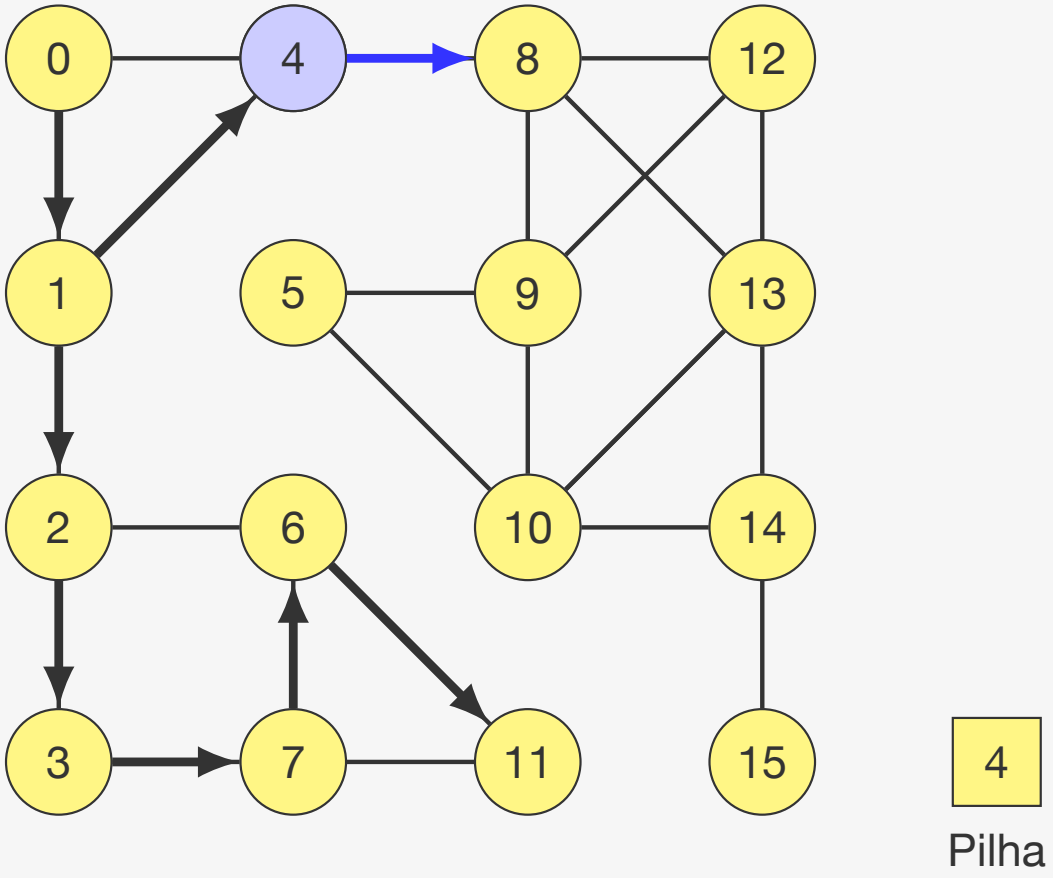
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

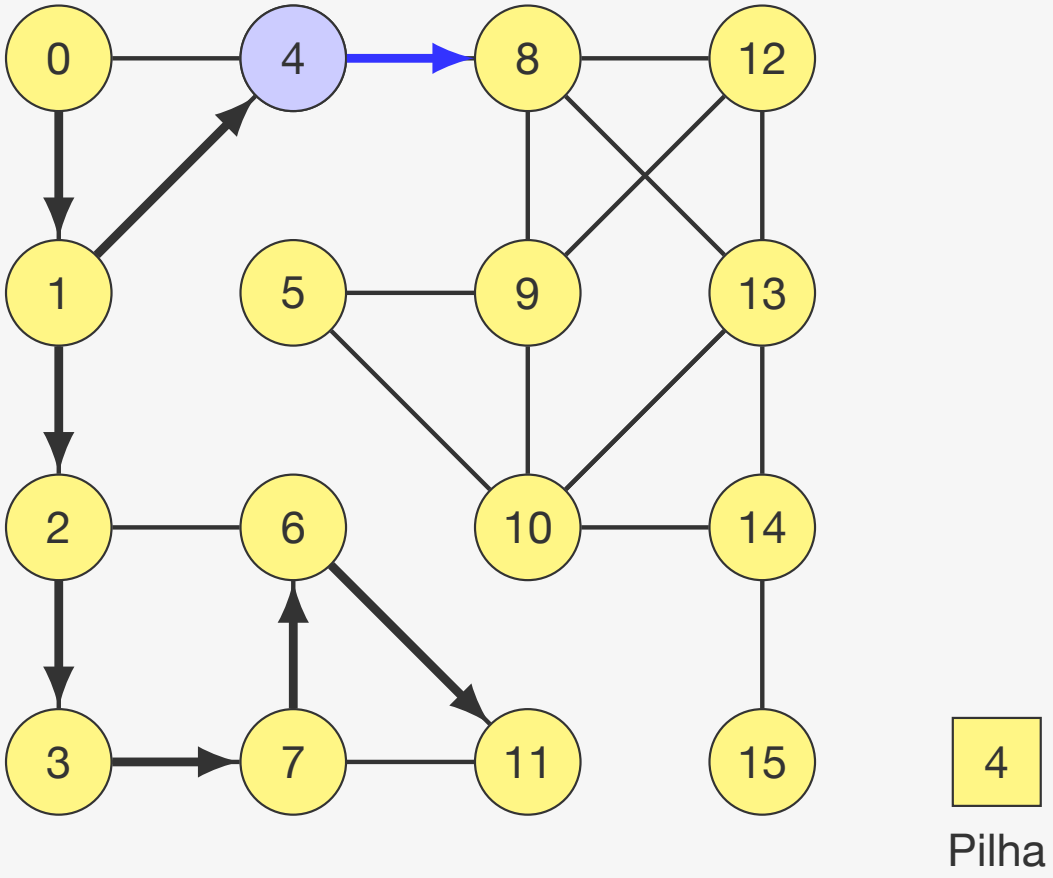
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

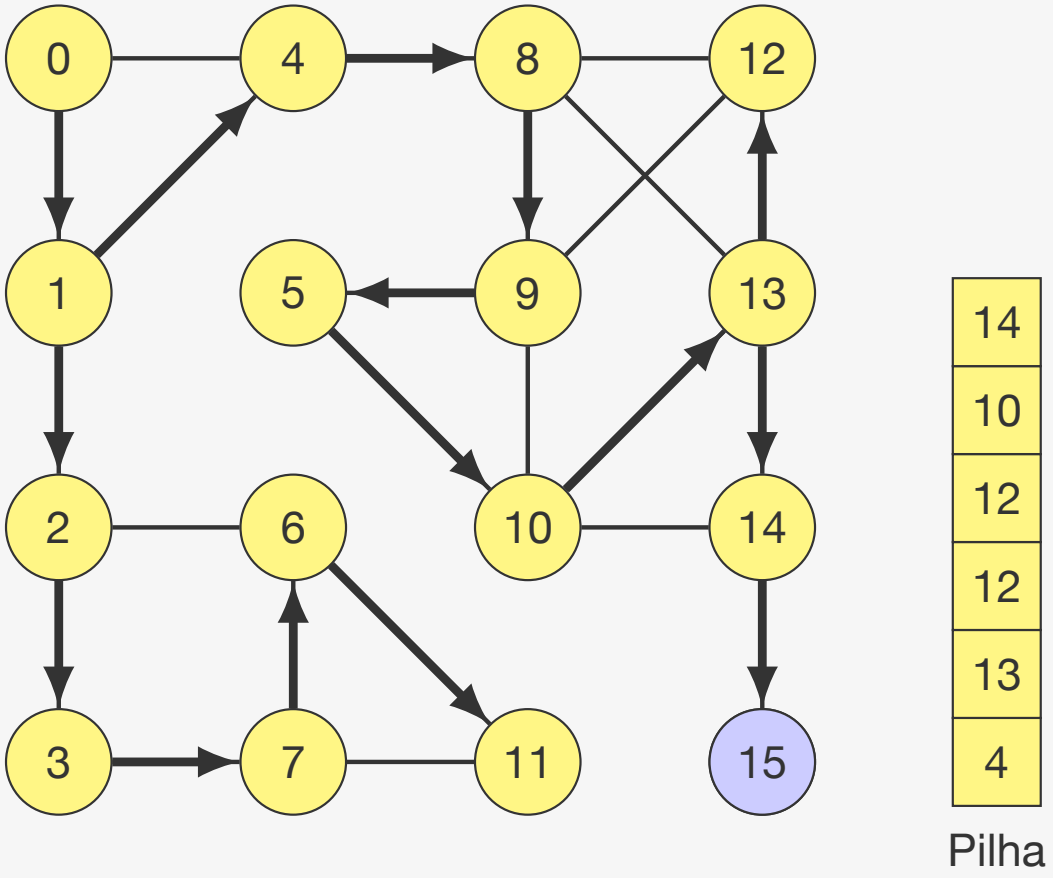
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

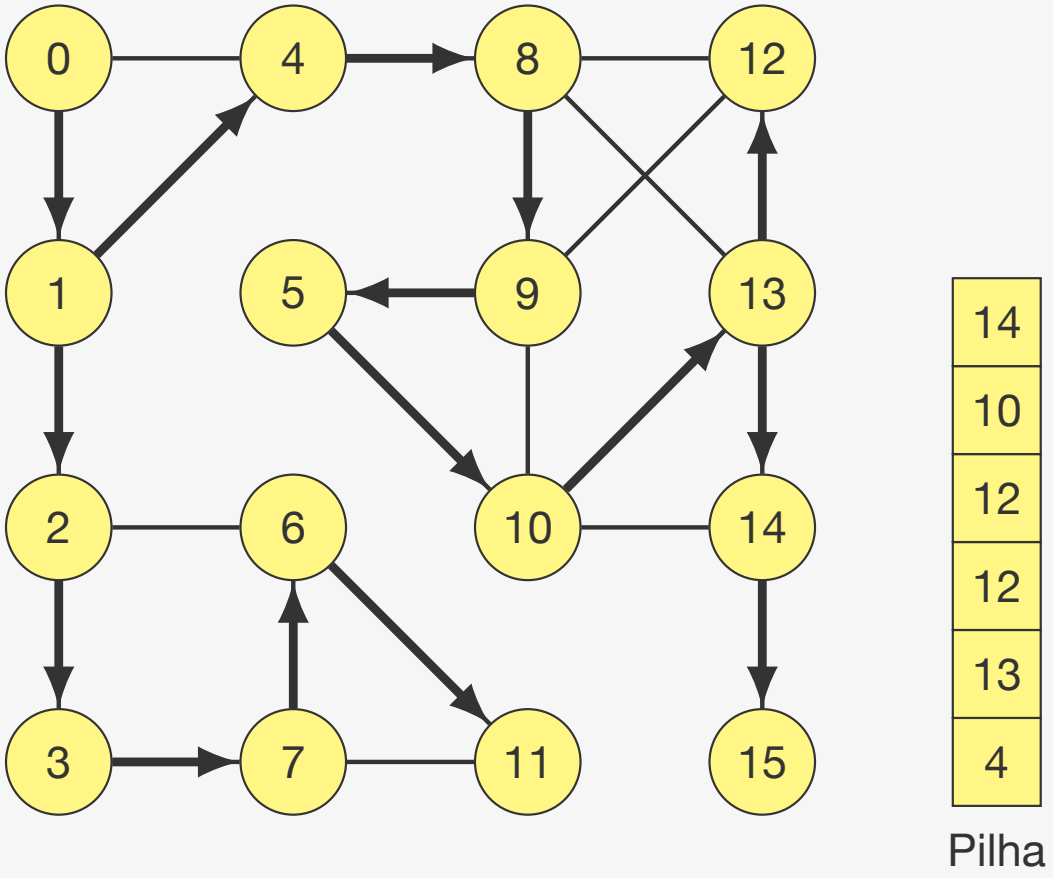
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

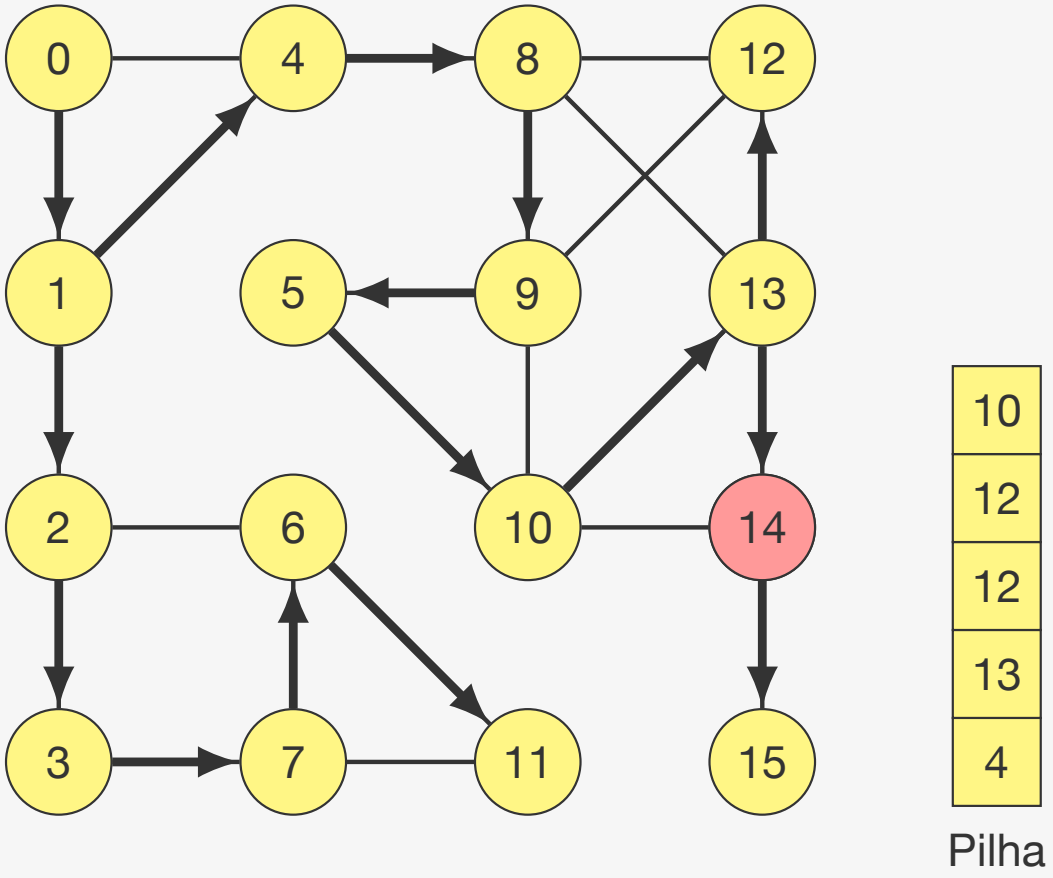
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

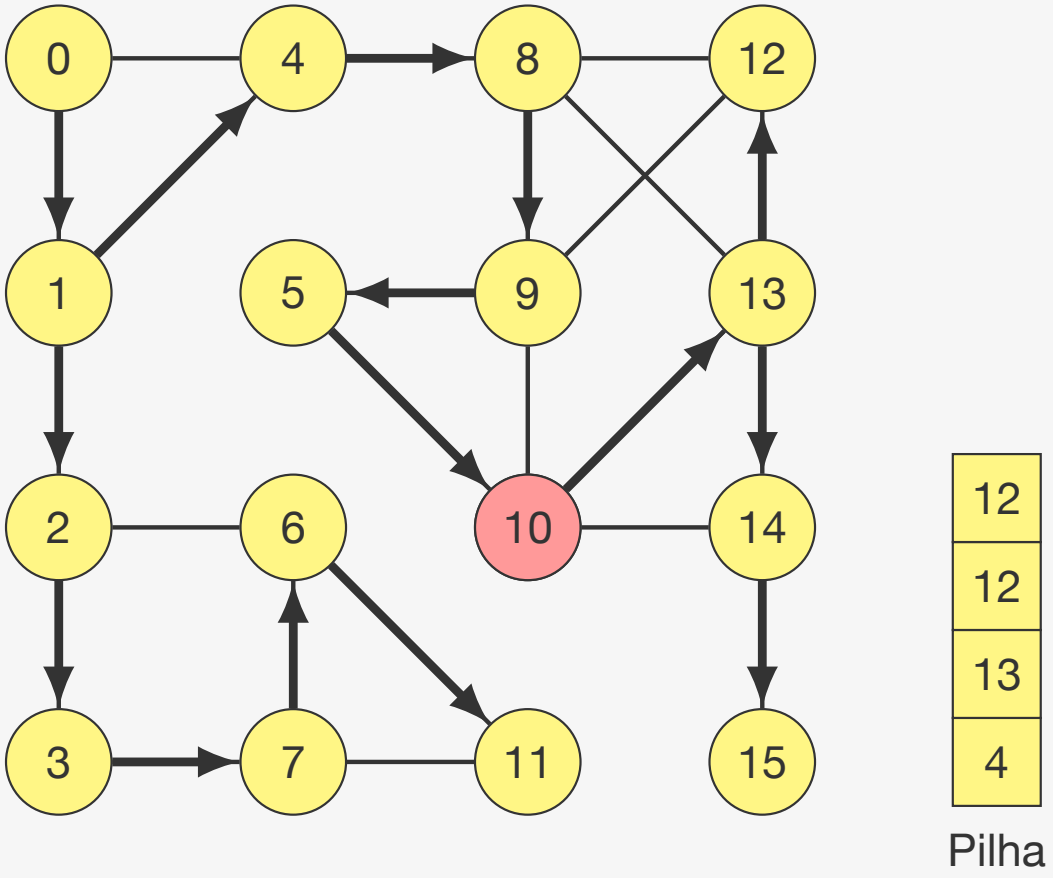
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

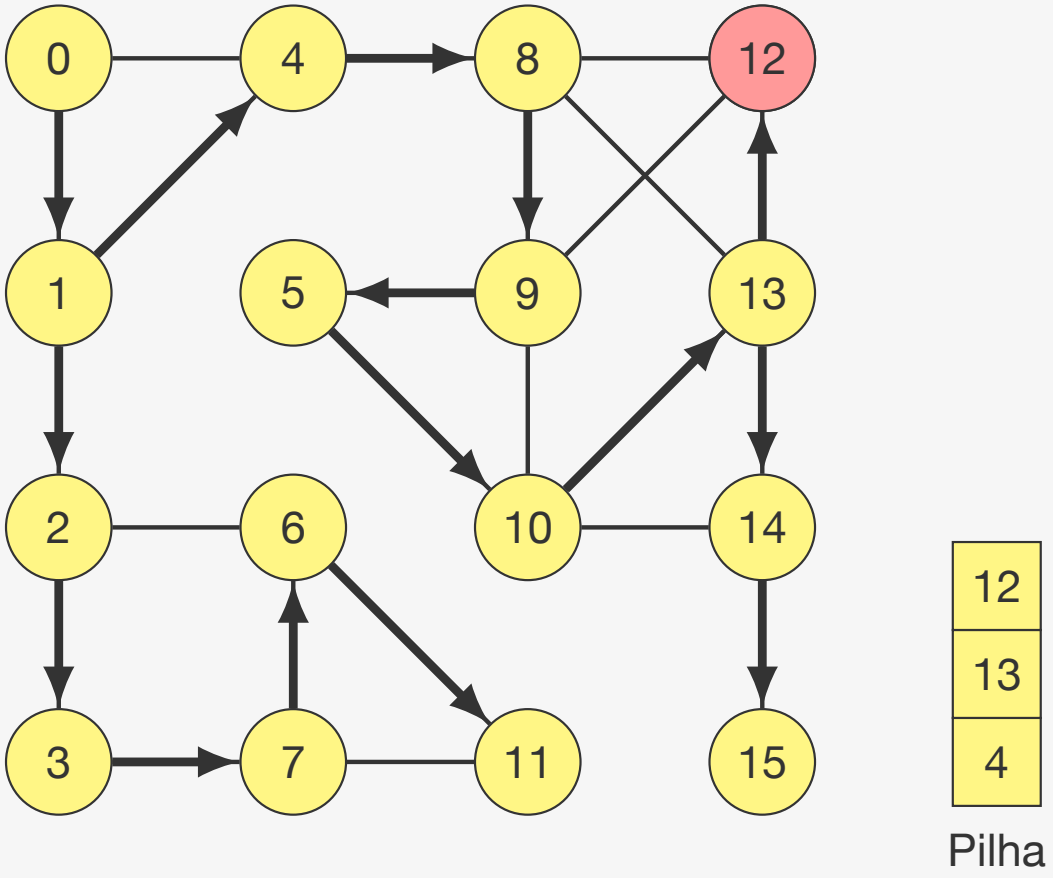
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

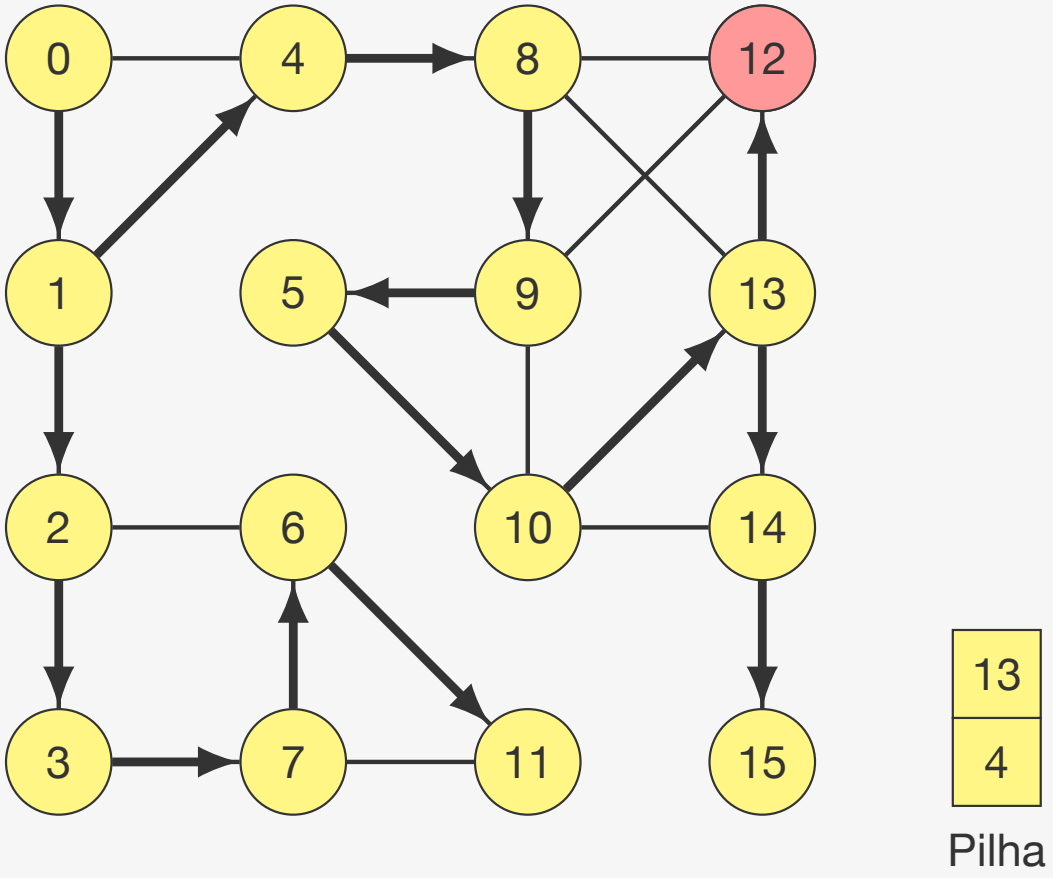
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

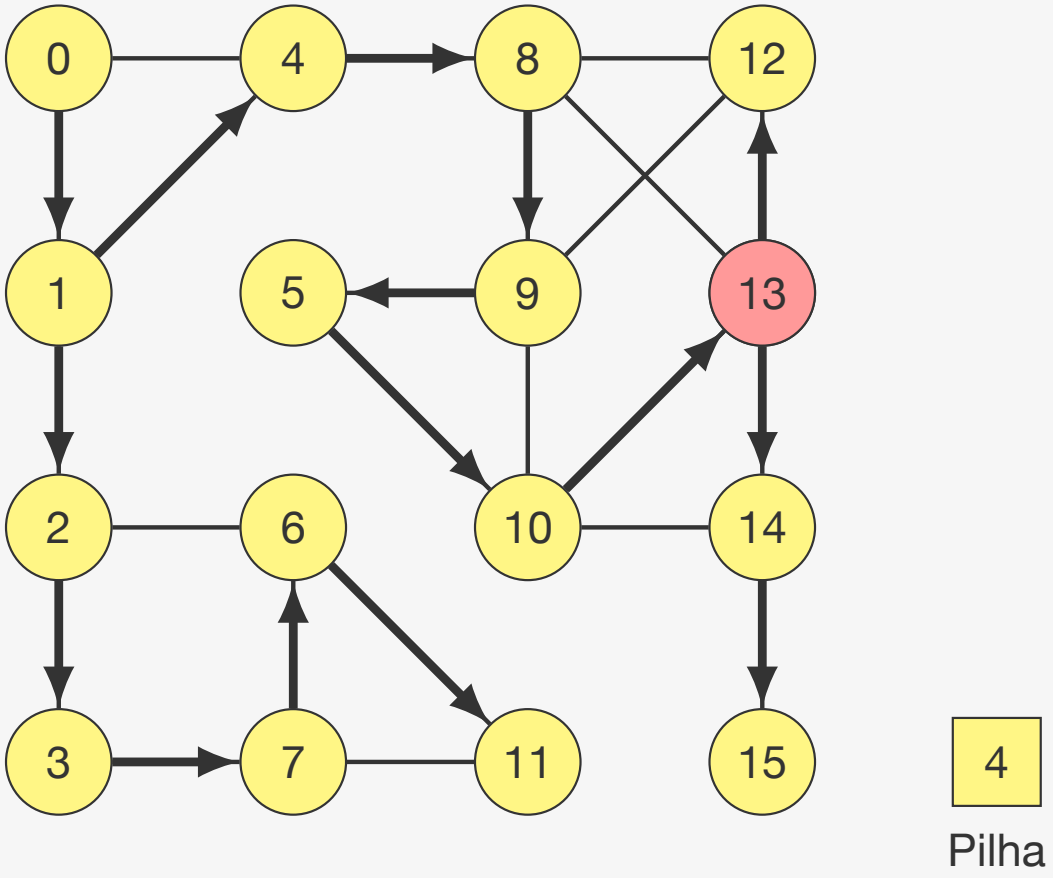
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

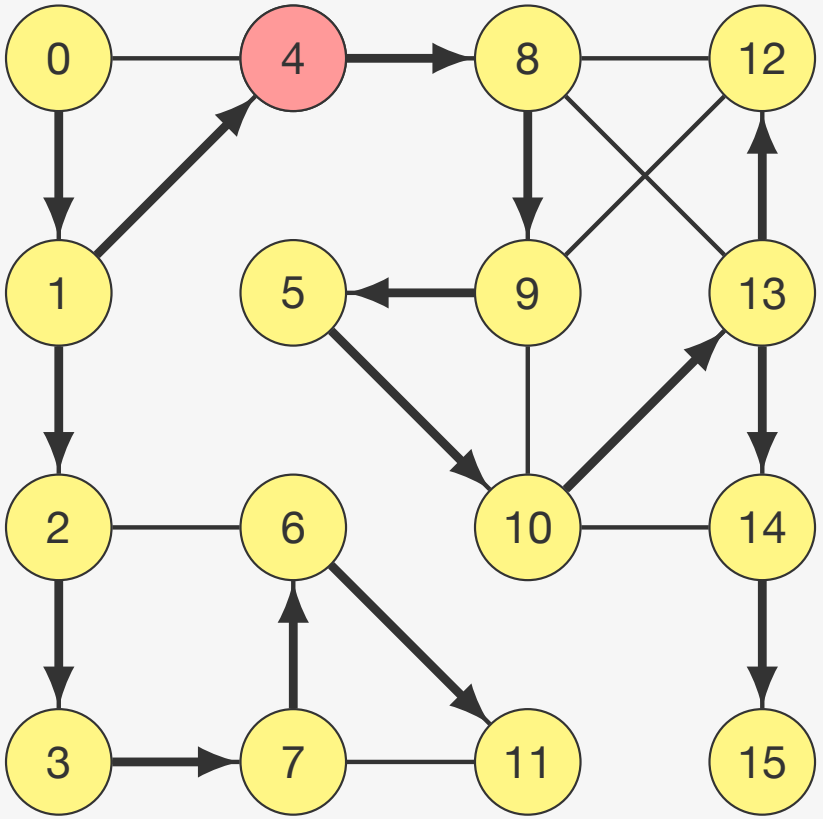
- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha



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



Pilha

Implementação

```
1 int * busca_em_profundidade(p_grafo g, int s) {
```

Implementação

```
1 int * busca_em_profundidade(p_grafo g, int s) {
2   int w, v;
3   int *pai = malloc(g->n * sizeof(int));
4   int *visitado = malloc(g->n * sizeof(int));
```

Implementação

```
1 int * busca_em_profundidade(p_grafo g, int s) {
2   int w, v;
3   int *pai = malloc(g->n * sizeof(int));
4   int *visitado = malloc(g->n * sizeof(int));
5   p_pilha p = criar_pilha();
```

Implementação

```
1 int * busca_em_profundidade(p_grafo g, int s) {
2   int w, v;
3   int *pai = malloc(g->n * sizeof(int));
4   int *visitado = malloc(g->n * sizeof(int));
5   p_pilha p = criar_pilha();
6   for (v = 0; v < g->n; v++) {
7     pai[v] = -1;
8     visitado[v] = 0;
9   }
```

Implementação

```
1  int * busca_em_profundidade(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_pilha p = criar_pilha();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   empilhar(p,s);
```

Implementação

```
1  int * busca_em_profundidade(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_pilha p = criar_pilha();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   empilhar(p,s);
11   pai[s] = s;
```

Implementação

```
1  int * busca_em_profundidade(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_pilha p = criar_pilha();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   empilhar(p,s);
11   pai[s] = s;
12   while(!pilha_vazia(p)) {
```

Implementação

```
1  int * busca_em_profundidade(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_pilha p = criar_pilha();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   empilhar(p,s);
11   pai[s] = s;
12   while(!pilha_vazia(p)) {
13     v = desempilhar(p);
```

Implementação

```
1  int * busca_em_profundidade(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_pilha p = criar_pilha();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   empilhar(p,s);
11   pai[s] = s;
12   while(!pilha_vazia(p)) {
13     v = desempilhar(p);
14     visitado[v] = 1;
```

Implementação

```
1  int * busca_em_profundidade(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_pilha p = criar_pilha();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   empilhar(p,s);
11   pai[s] = s;
12   while(!pilha_vazia(p)) {
13     v = desempilhar(p);
14     visitado[v] = 1;
15     for (w = 0; w < g->n; w++)
```

Implementação

```
1  int * busca_em_profundidade(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_pilha p = criar_pilha();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   empilhar(p,s);
11   pai[s] = s;
12   while(!pilha_vazia(p)) {
13     v = desempilhar(p);
14     visitado[v] = 1;
15     for (w = 0; w < g->n; w++)
16       if (g->adj[v][w] && !visitado[w]) {
```

Implementação

```
1  int * busca_em_profundidade(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_pilha p = criar_pilha();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   empilhar(p,s);
11   pai[s] = s;
12   while(!pilha_vazia(p)) {
13     v = desempilhar(p);
14     visitado[v] = 1;
15     for (w = 0; w < g->n; w++)
16       if (g->adj[v][w] && !visitado[w]) {
17         pai[w] = v;
```

Implementação

```
1  int * busca_em_profundidade(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_pilha p = criar_pilha();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   empilhar(p,s);
11   pai[s] = s;
12   while(!pilha_vazia(p)) {
13     v = desempilhar(p);
14     visitado[v] = 1;
15     for (w = 0; w < g->n; w++)
16       if (g->adj[v][w] && !visitado[w]) {
17         pai[w] = v;
18         empilhar(p, w);
```

Implementação

```
1  int * busca_em_profundidade(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_pilha p = criar_pilha();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   empilhar(p,s);
11   pai[s] = s;
12   while(!pilha_vazia(p)) {
13     v = desempilhar(p);
14     visitado[v] = 1;
15     for (w = 0; w < g->n; w++)
16       if (g->adj[v][w] && !visitado[w]) {
17         pai[w] = v;
18         empilhar(p, w);
19       }
20   }
```

Implementação

```
1  int * busca_em_profundidade(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_pilha p = criar_pilha();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   empilhar(p,s);
11   pai[s] = s;
12   while(!pilha_vazia(p)) {
13     v = desempilhar(p);
14     visitado[v] = 1;
15     for (w = 0; w < g->n; w++)
16       if (g->adj[v][w] && !visitado[w]) {
17         pai[w] = v;
18         empilhar(p, w);
19       }
20   }
21   destroi_pilha(p);
```

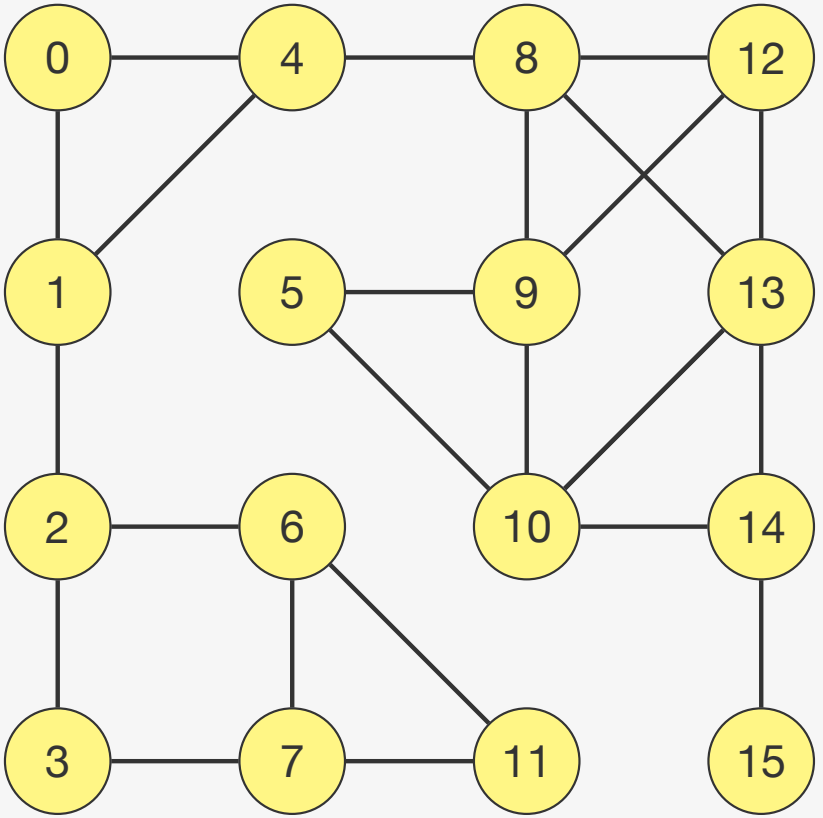
Implementação

```
1  int * busca_em_profundidade(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_pilha p = criar_pilha();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   empilhar(p,s);
11   pai[s] = s;
12   while(!pilha_vazia(p)) {
13     v = desempilhar(p);
14     visitado[v] = 1;
15     for (w = 0; w < g->n; w++)
16       if (g->adj[v][w] && !visitado[w]) {
17         pai[w] = v;
18         empilhar(p, w);
19       }
20   }
21   destroi_pilha(p);
22   free(visitado);
```

Implementação

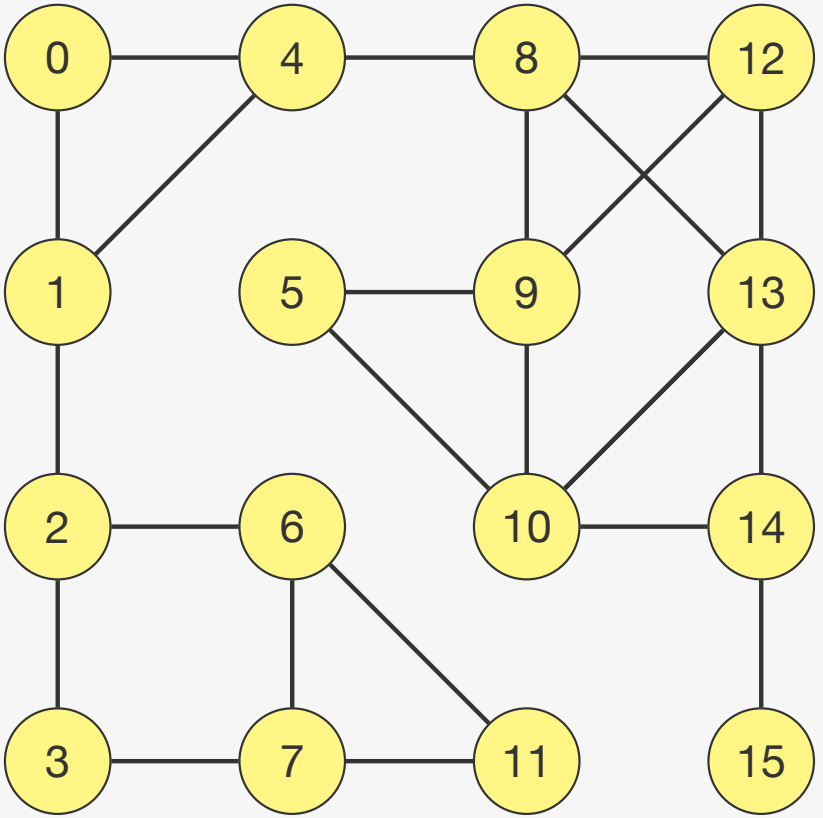
```
1  int * busca_em_profundidade(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_pilha p = criar_pilha();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   empilhar(p,s);
11   pai[s] = s;
12   while(!pilha_vazia(p)) {
13     v = desempilhar(p);
14     visitado[v] = 1;
15     for (w = 0; w < g->n; w++)
16       if (g->adj[v][w] && !visitado[w]) {
17         pai[w] = v;
18         empilhar(p, w);
19       }
20   }
21   destroi_pilha(p);
22   free(visitado);
23   return pai;
24 }
```

E se tivéssemos usando uma Fila?



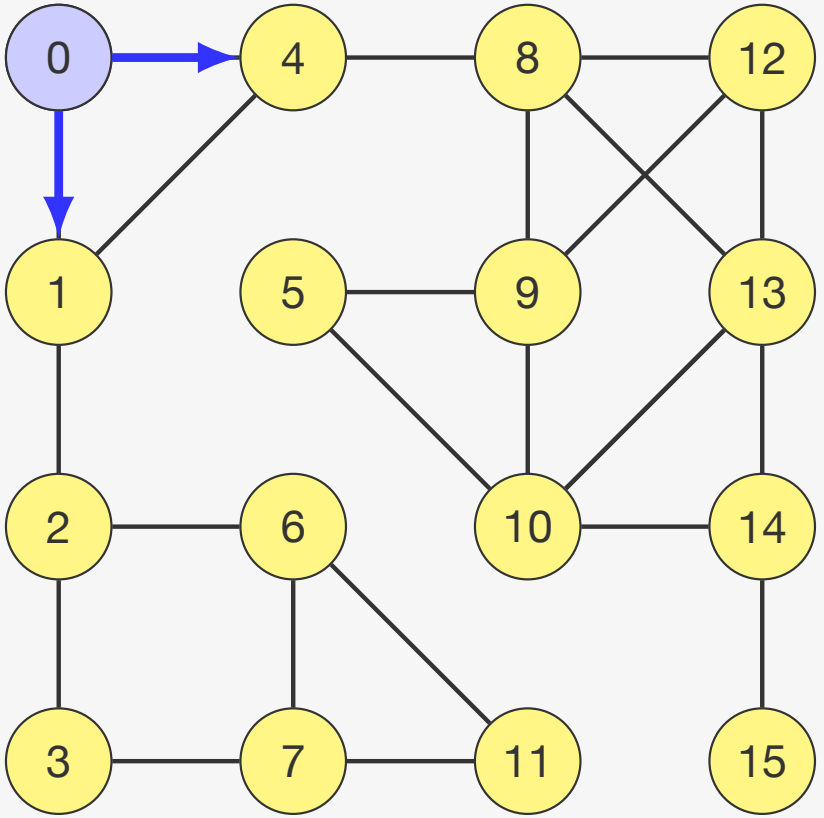
Fila

E se tivéssemos usando uma Fila?



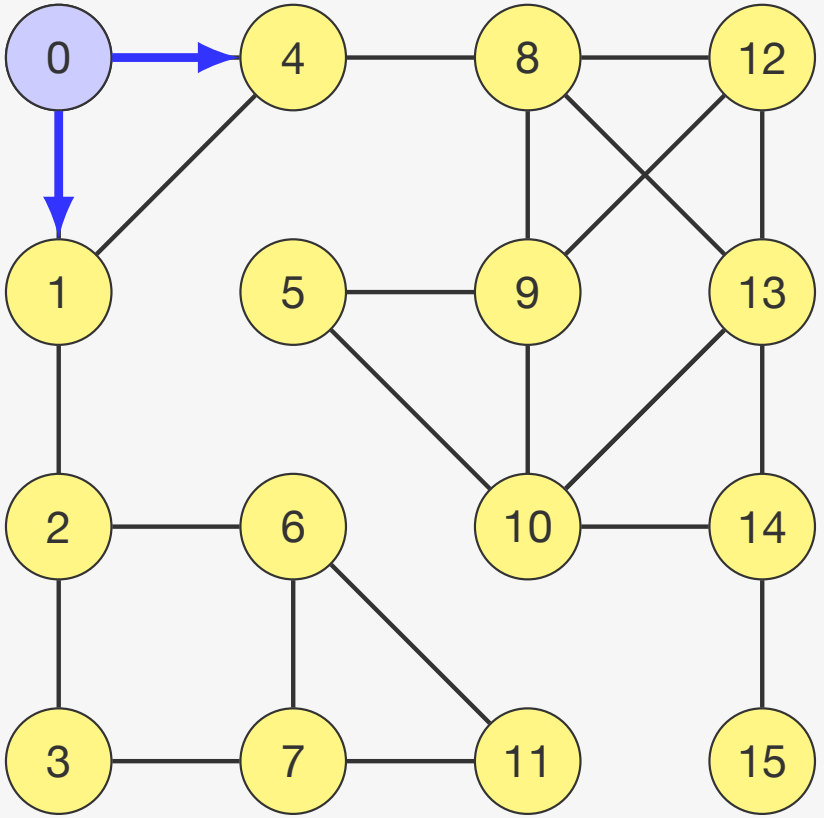
Fila | 0 |

E se tivéssemos usando uma Fila?



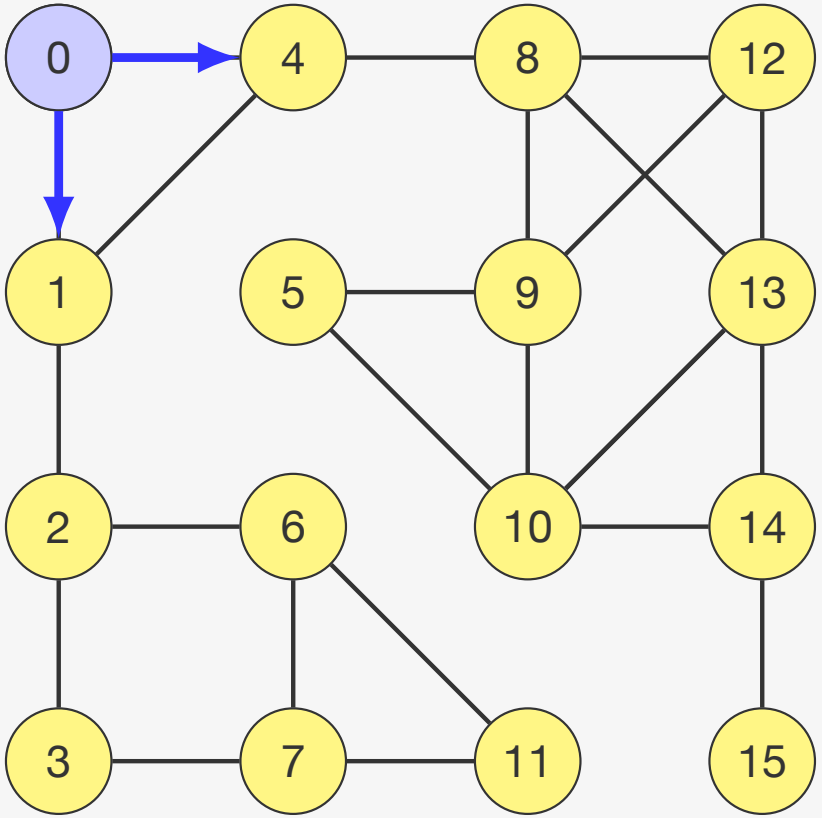
Fila

E se tivéssemos usando uma Fila?



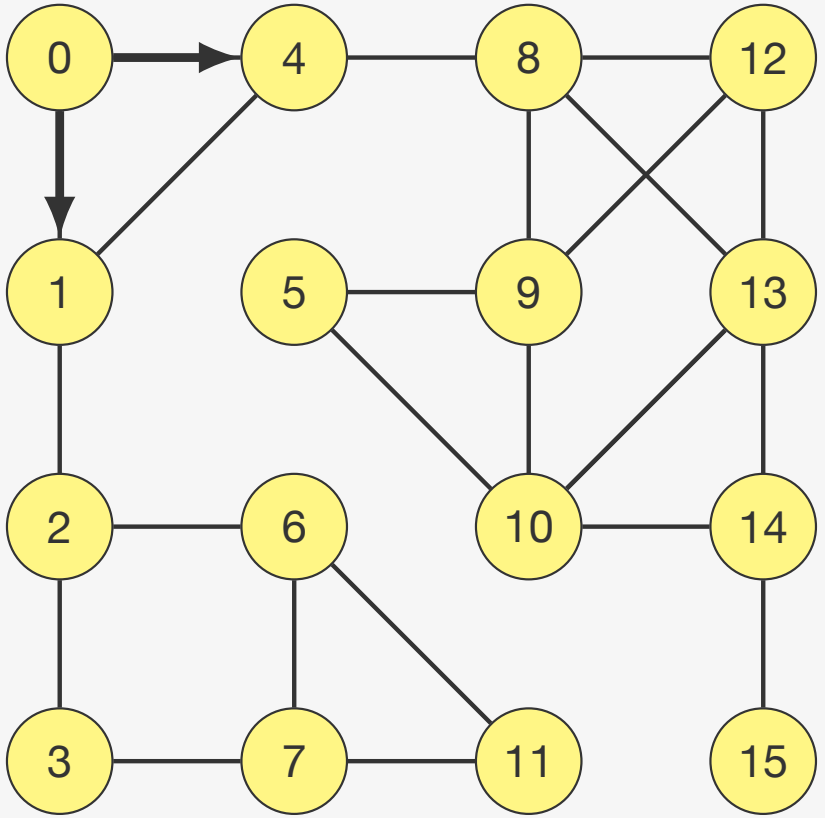
Fila | 1 |

E se tivéssemos usando uma Fila?



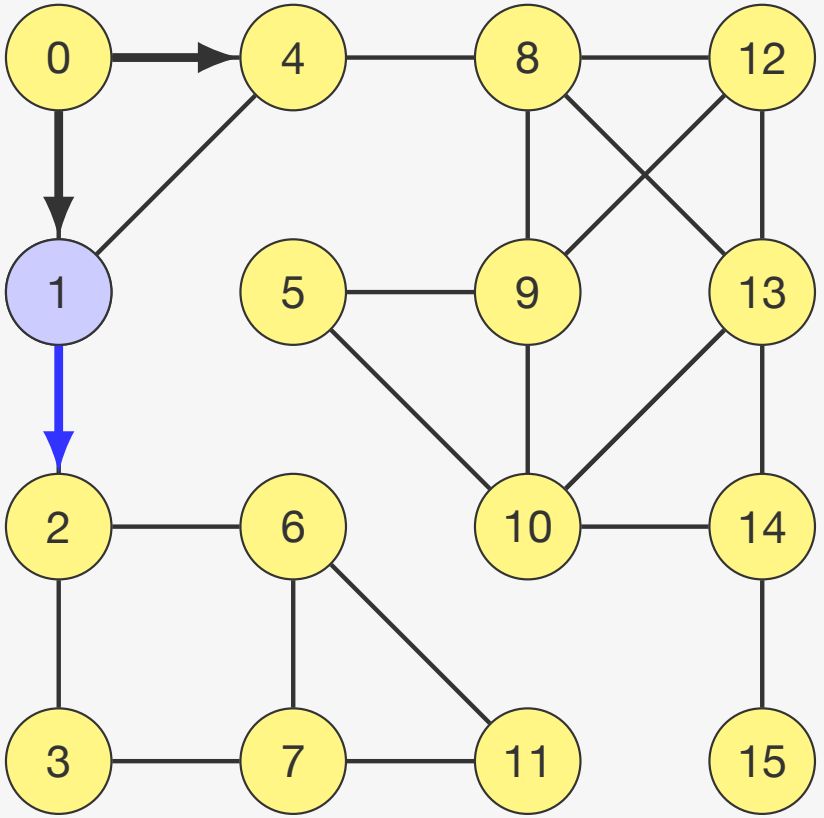
Fila | 1 | 4 |

E se tivéssemos usando uma Fila?



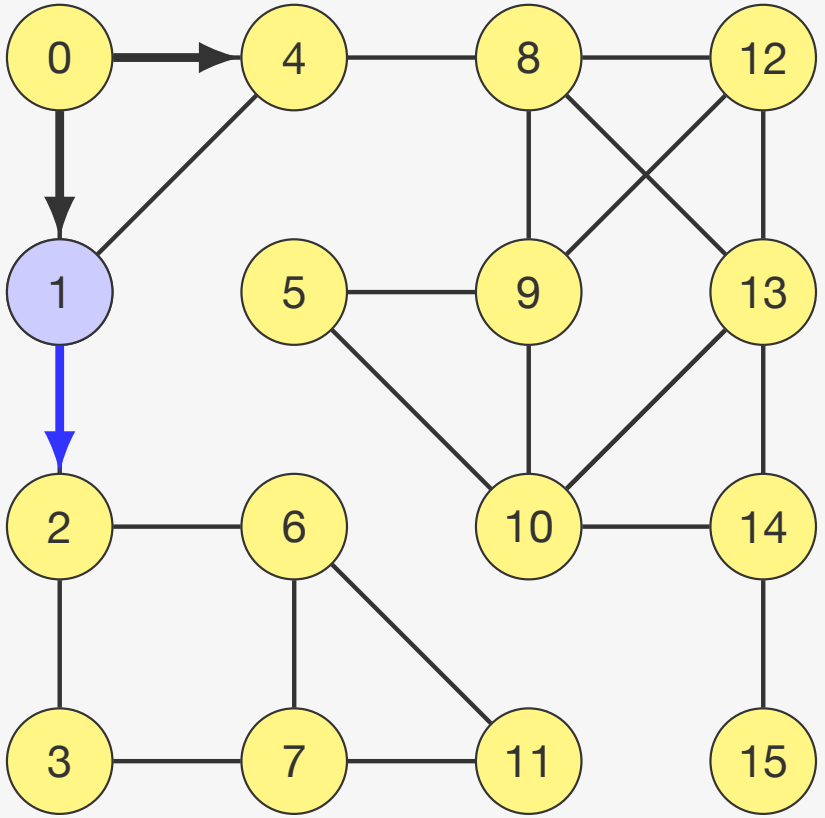
Fila | 1 | 4 |

E se tivéssemos usando uma Fila?



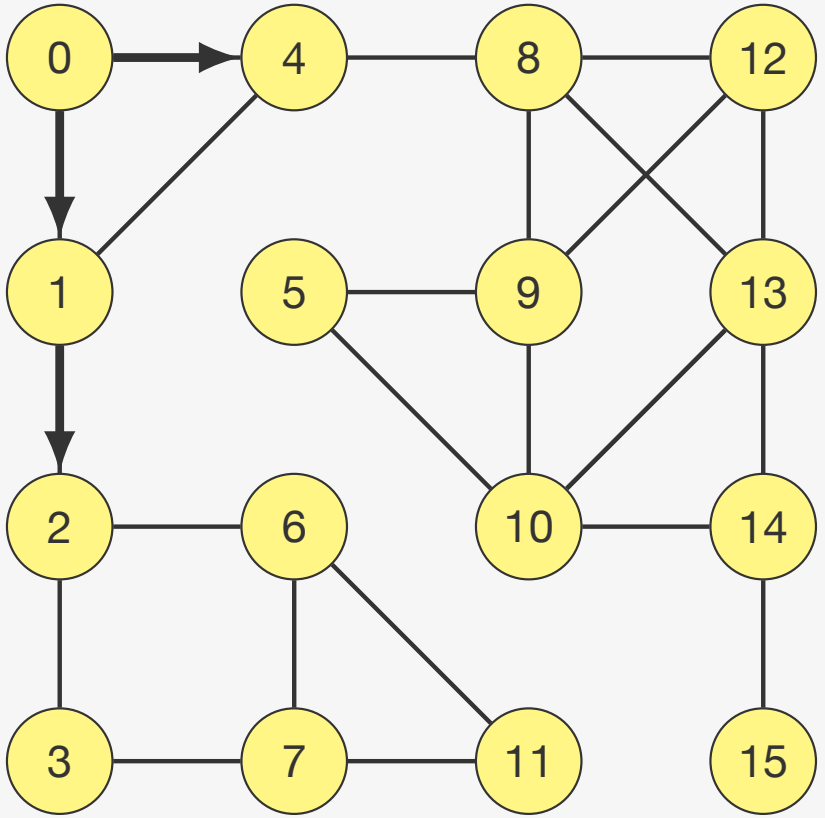
Fila | 4 |

E se tivéssemos usando uma Fila?



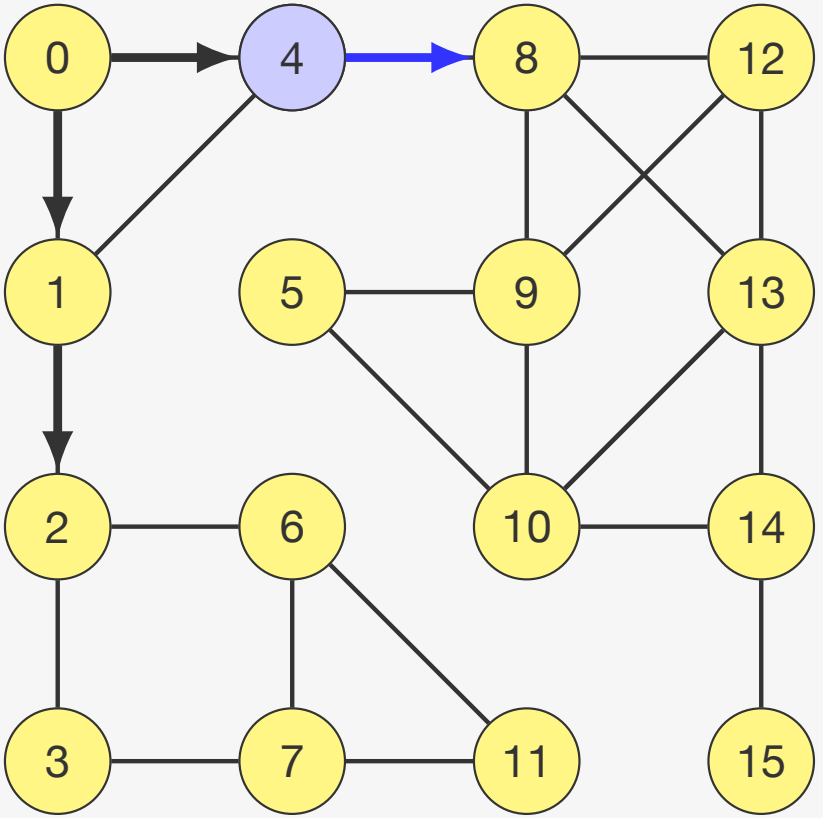
Fila | 4 | 2 |

E se tivéssemos usando uma Fila?



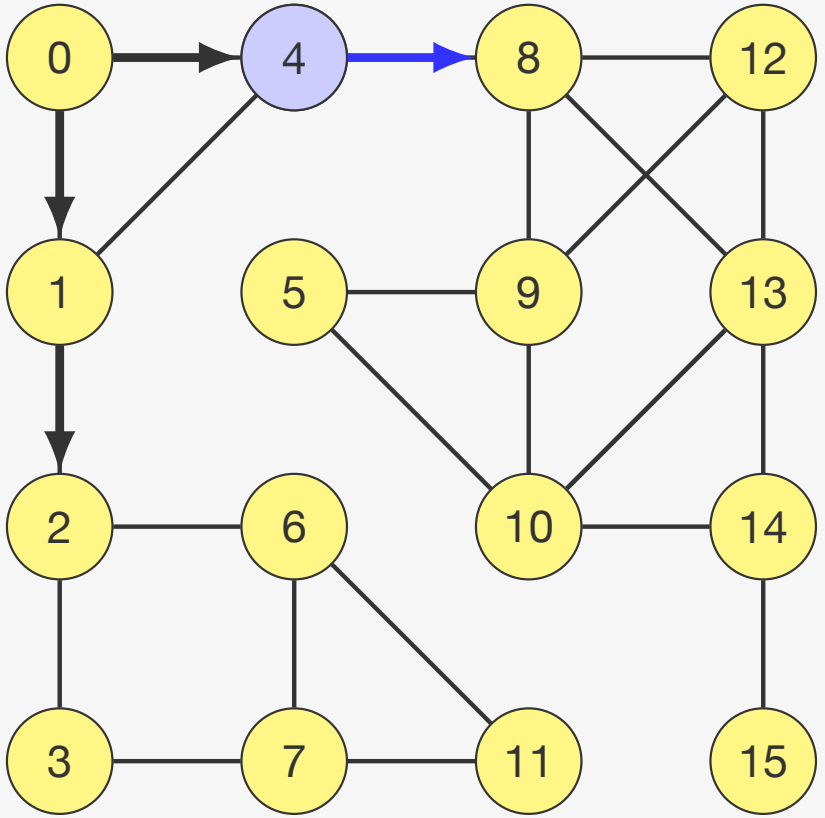
Fila | 4 | 2 |

E se tivéssemos usando uma Fila?



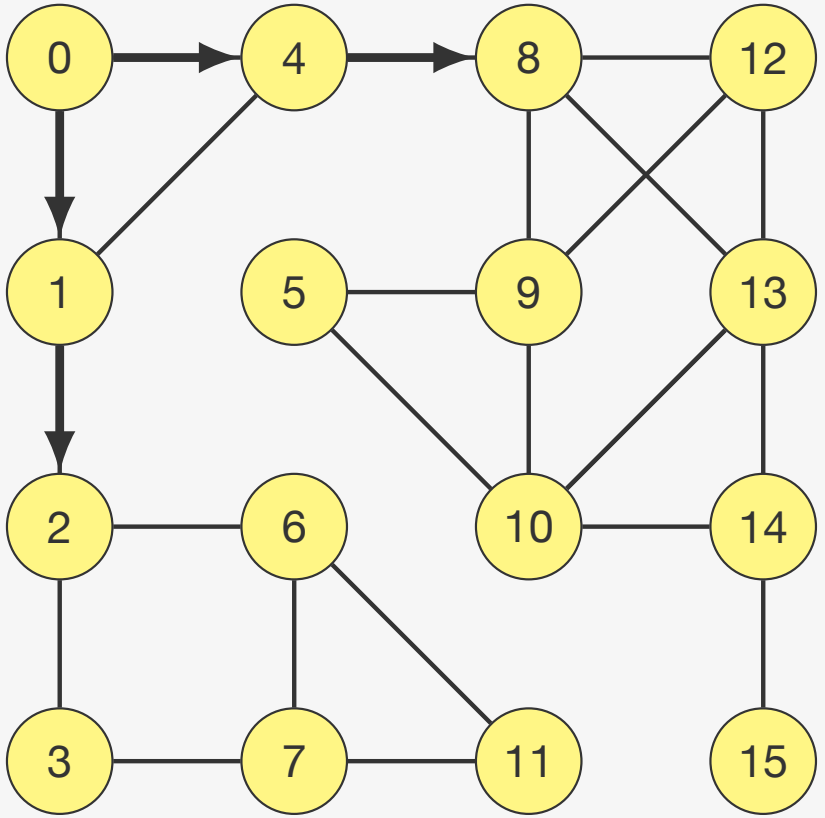
Fila | 2 |

E se tivéssemos usando uma Fila?



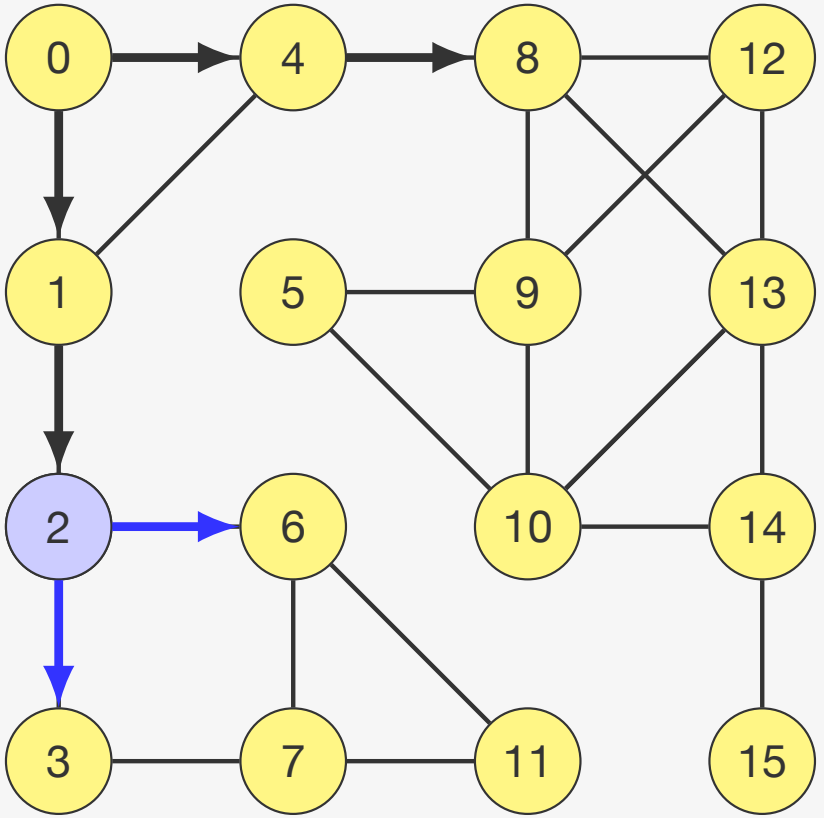
Fila | 2 | 8 |

E se tivéssemos usando uma Fila?



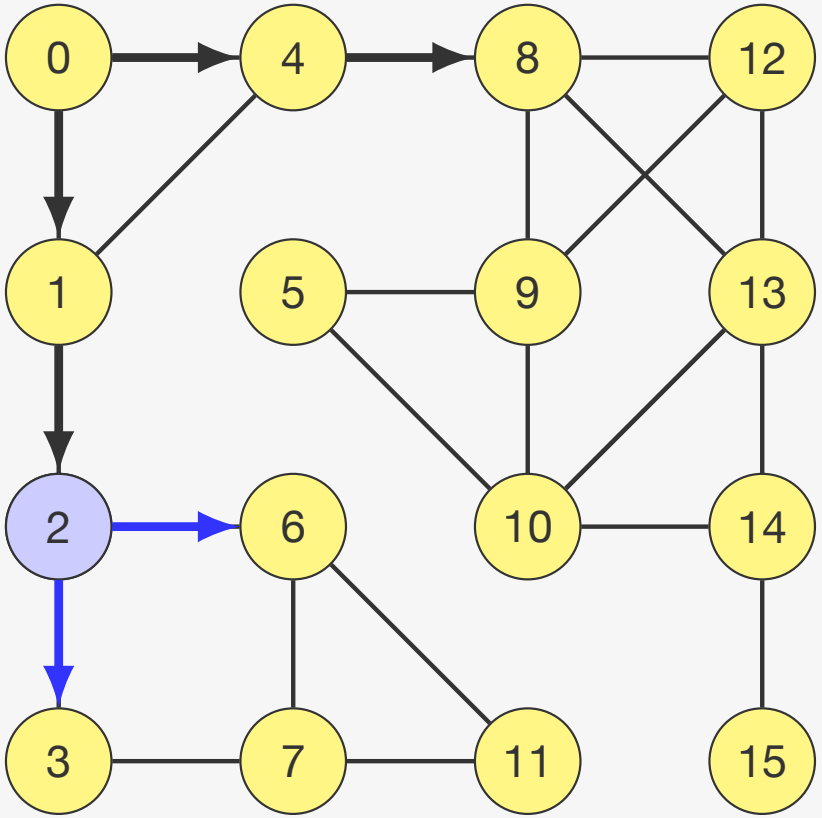
Fila | 2 | 8 |

E se tivéssemos usando uma Fila?



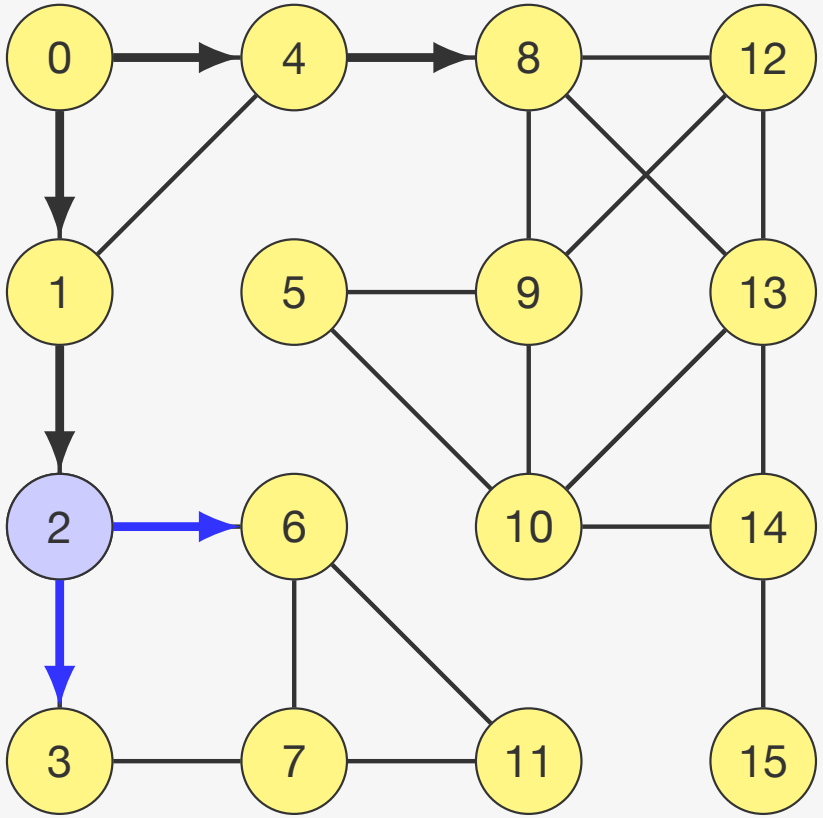
Fila | 8

E se tivéssemos usando uma Fila?



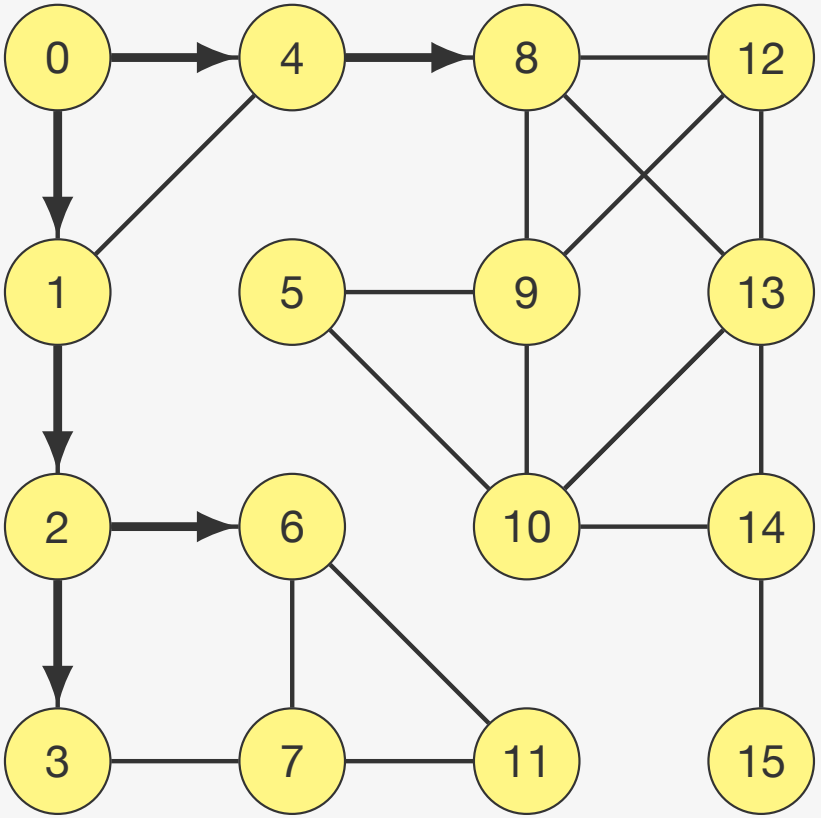
Fila | 8 | 3 |

E se tivéssemos usando uma Fila?



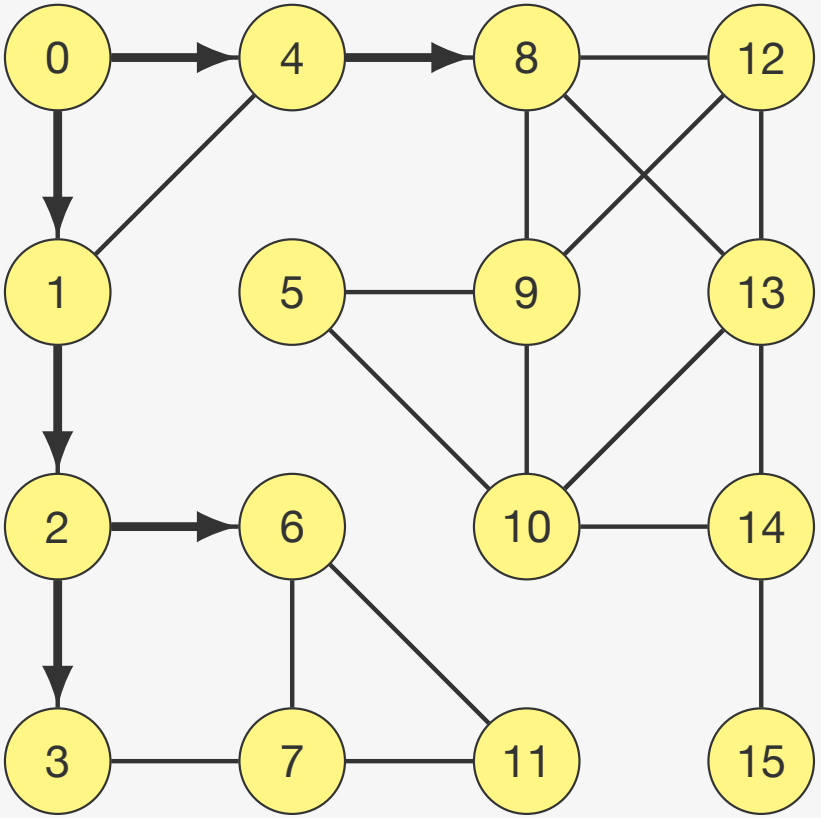
Fila | 8 | 3 | 6 |

E se tivéssemos usando uma Fila?



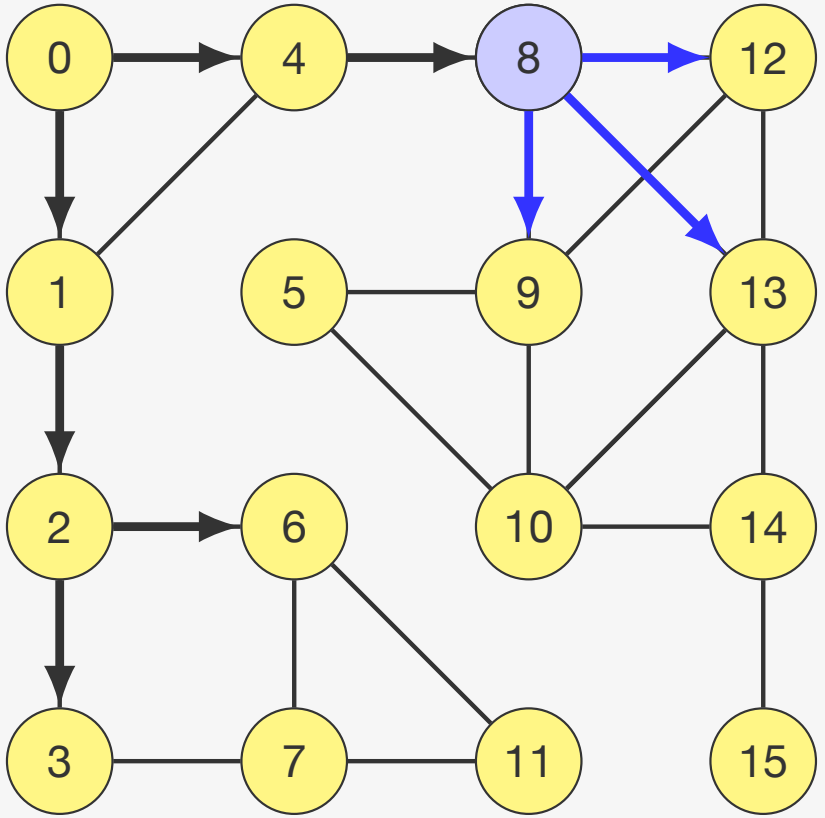
Fila | 8 | 3 | 6 |

E se tivéssemos usando uma Fila?



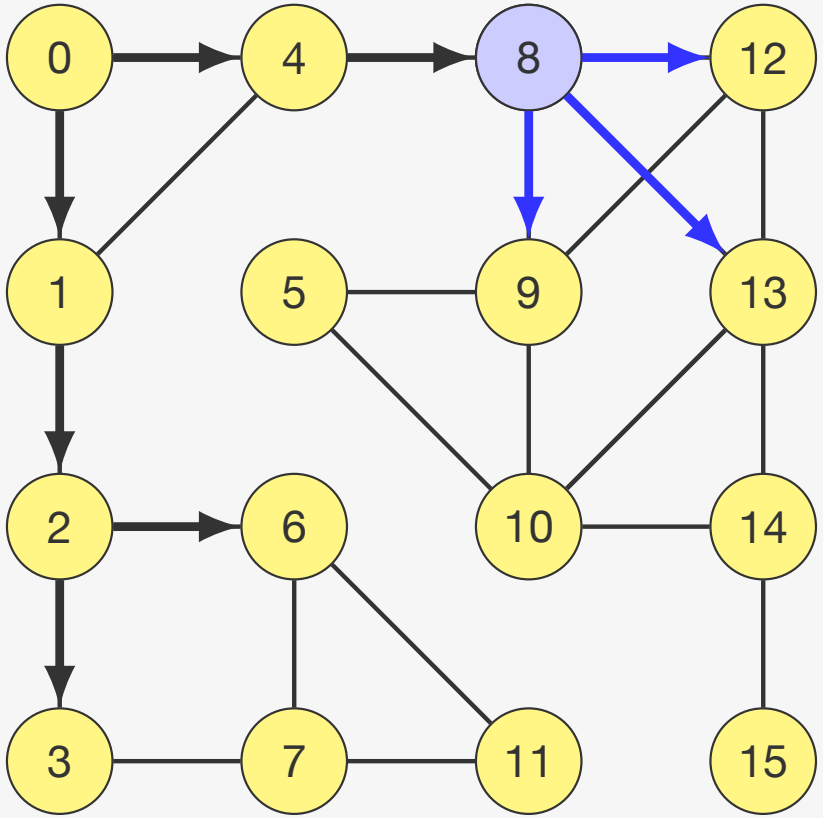
Fila | 8 | 3 | 6 |

E se tivéssemos usando uma Fila?



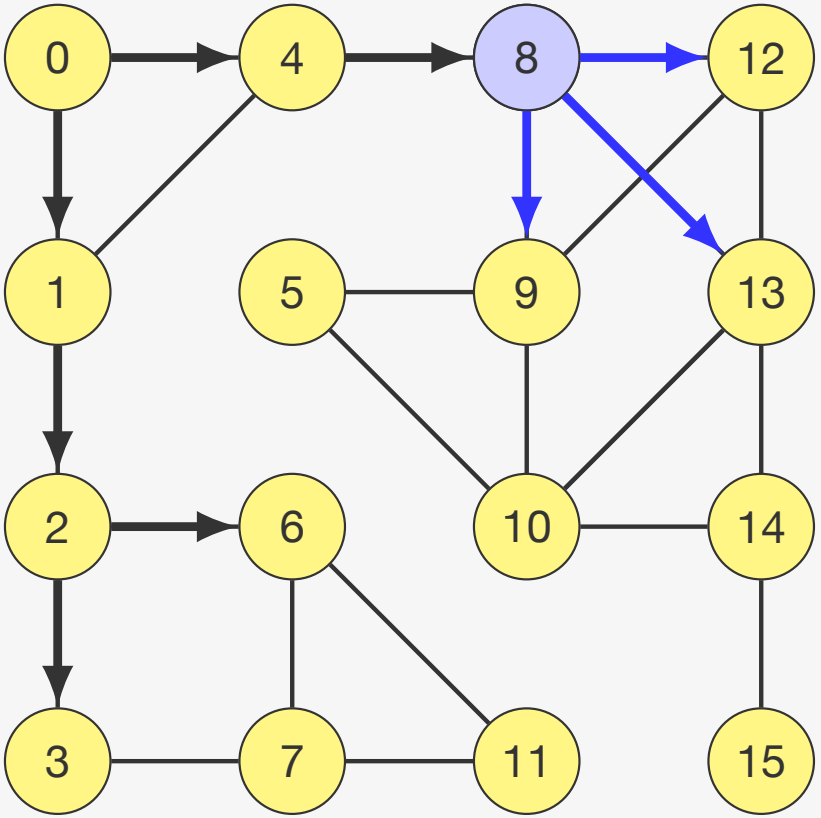
Fila | 3 | 6 |

E se tivéssemos usando uma Fila?



Fila | 3 | 6 | 9 |

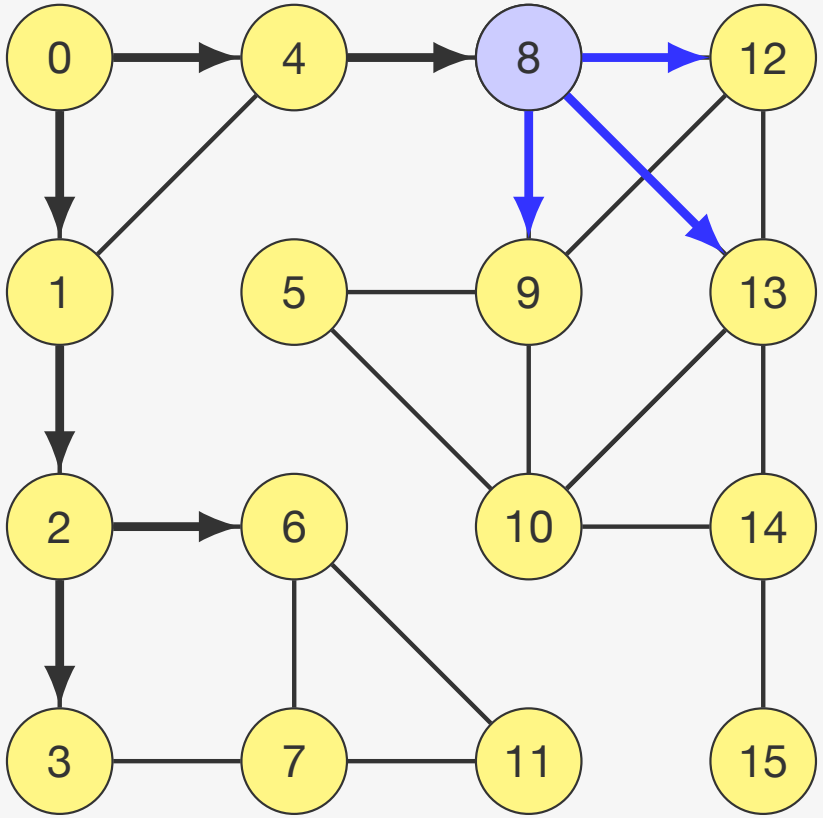
E se tivéssemos usando uma Fila?



Fila

3	6	9	12

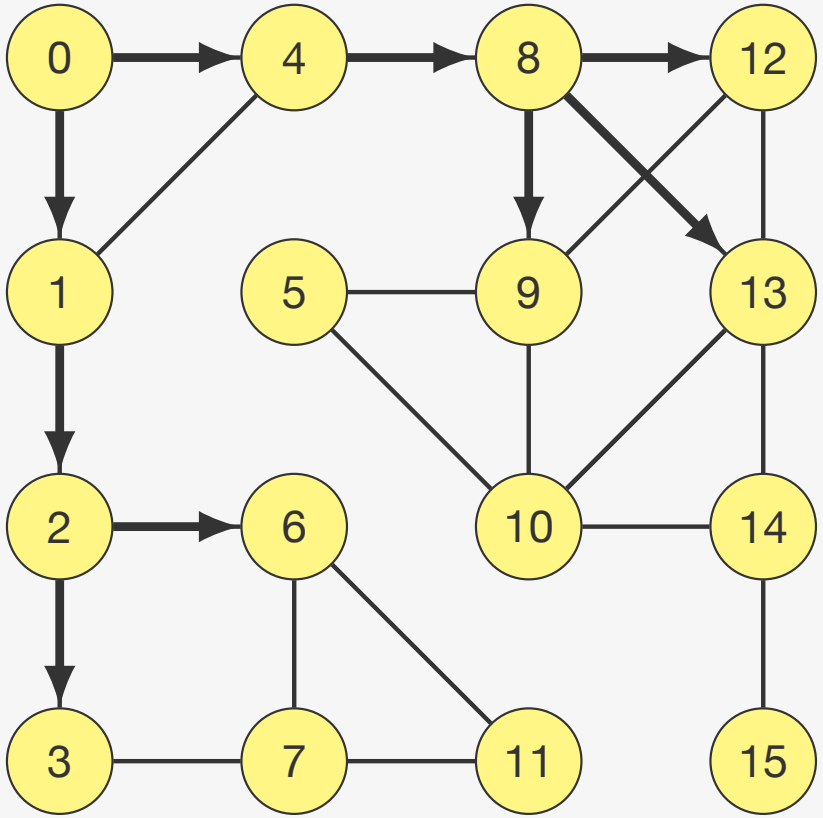
E se tivéssemos usando uma Fila?



Fila

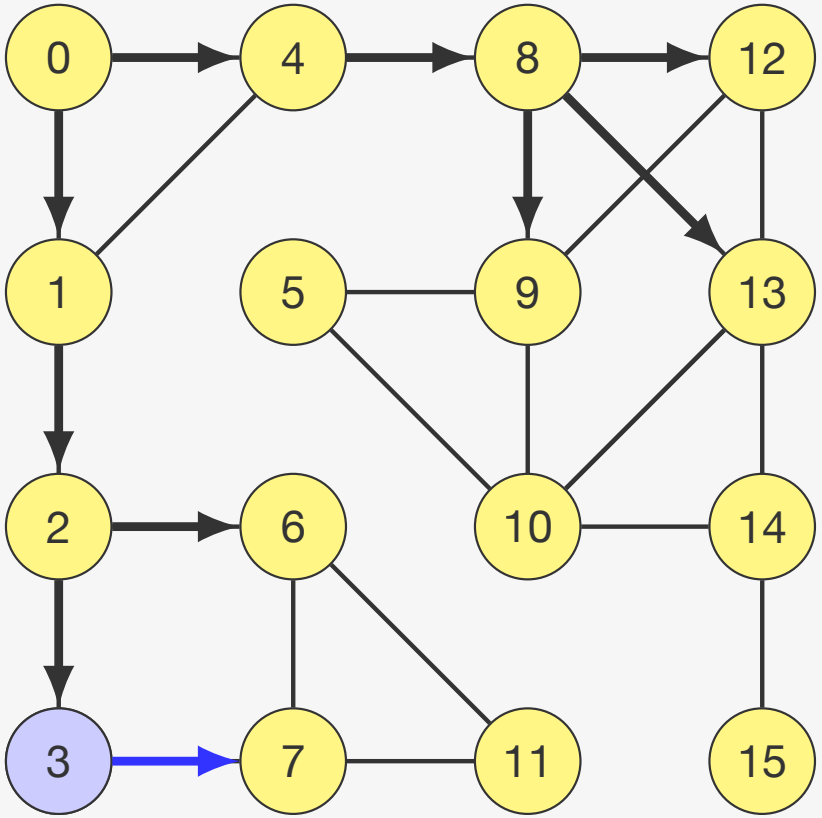
3	6	9	12	13

E se tivéssemos usando uma Fila?



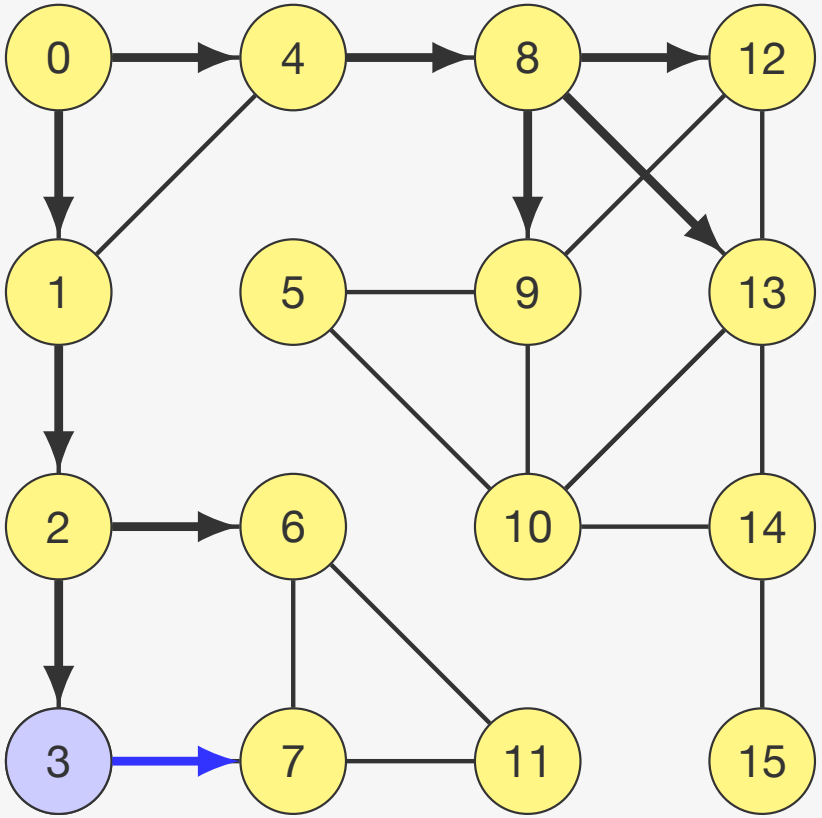
Fila | 3 | 6 | 9 | 12 | 13 |

E se tivéssemos usando uma Fila?



Fila | 6 | 9 | 12 | 13 |

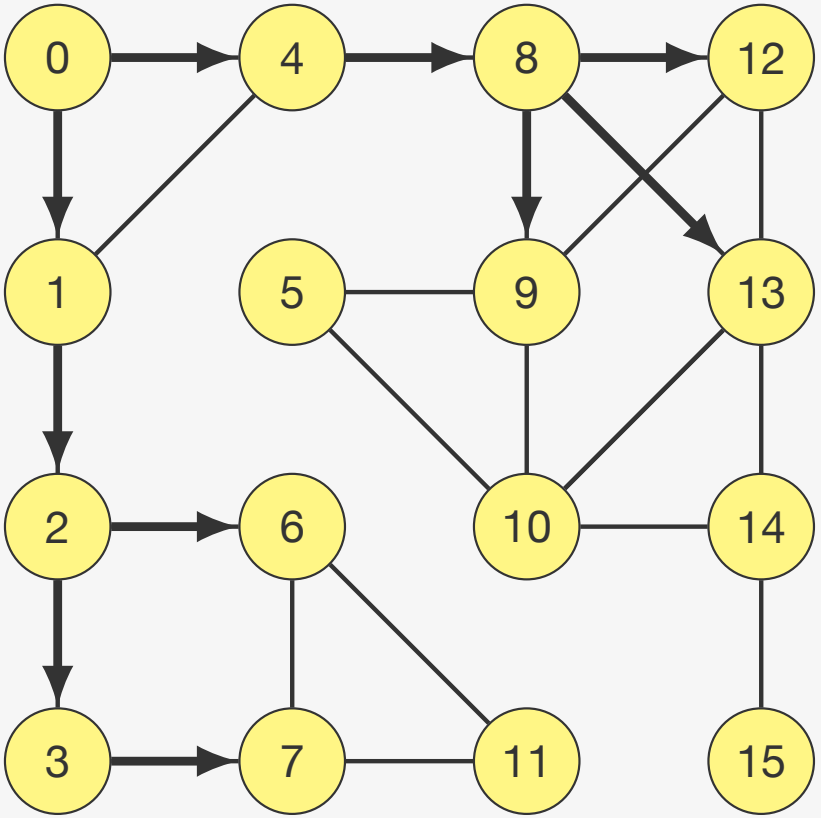
E se tivéssemos usando uma Fila?



Fila

6	9	12	13	7

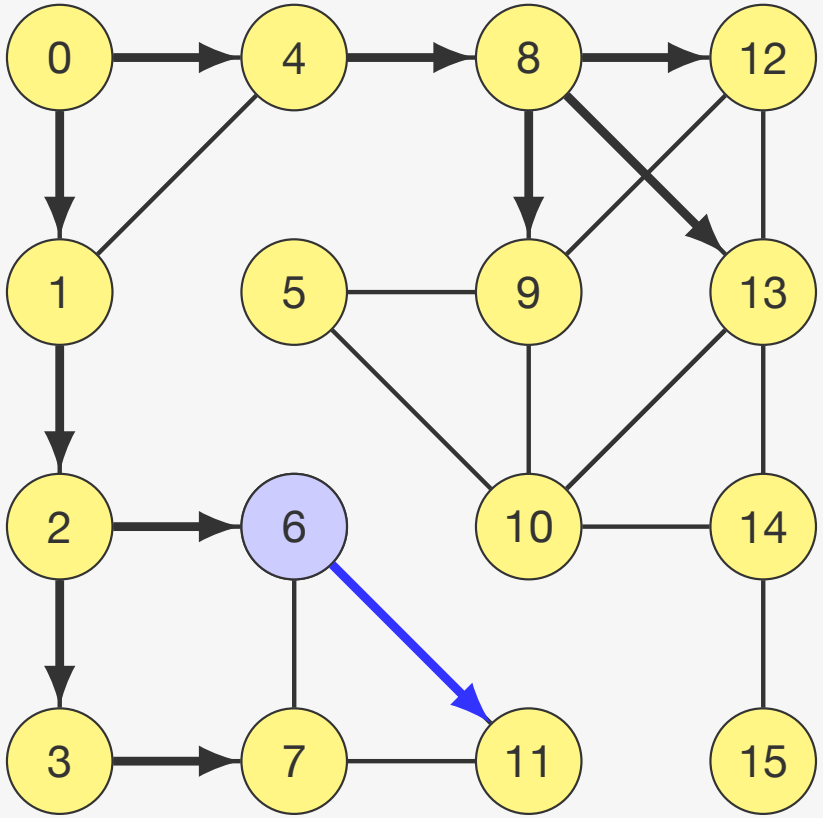
E se tivéssemos usando uma Fila?



Fila

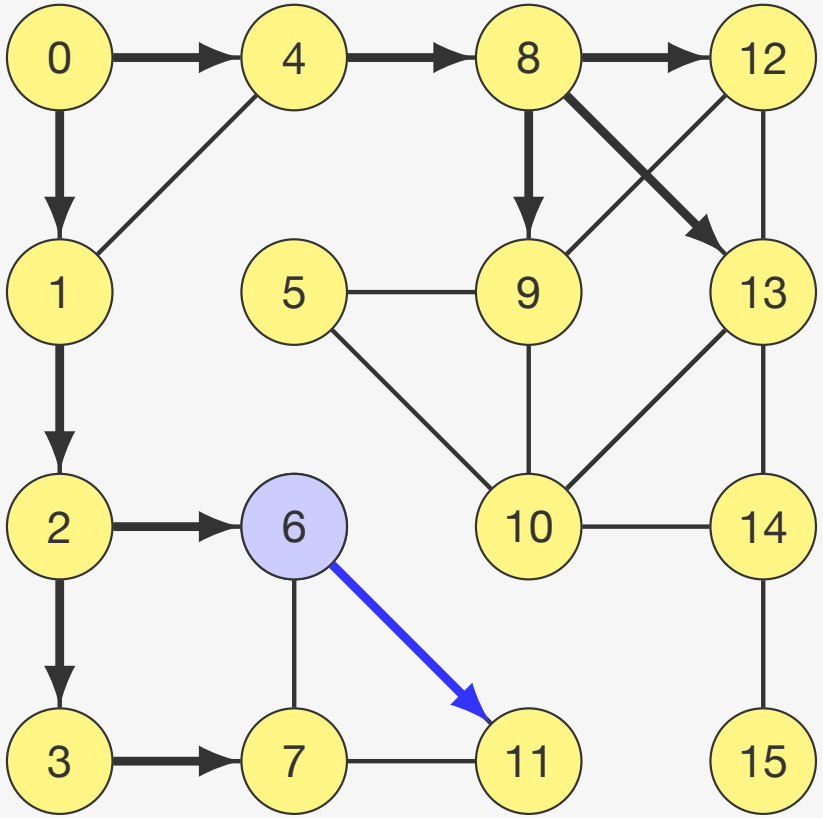
6	9	12	13	7

E se tivéssemos usando uma Fila?



Fila | 9 | 12 | 13 | 7 |

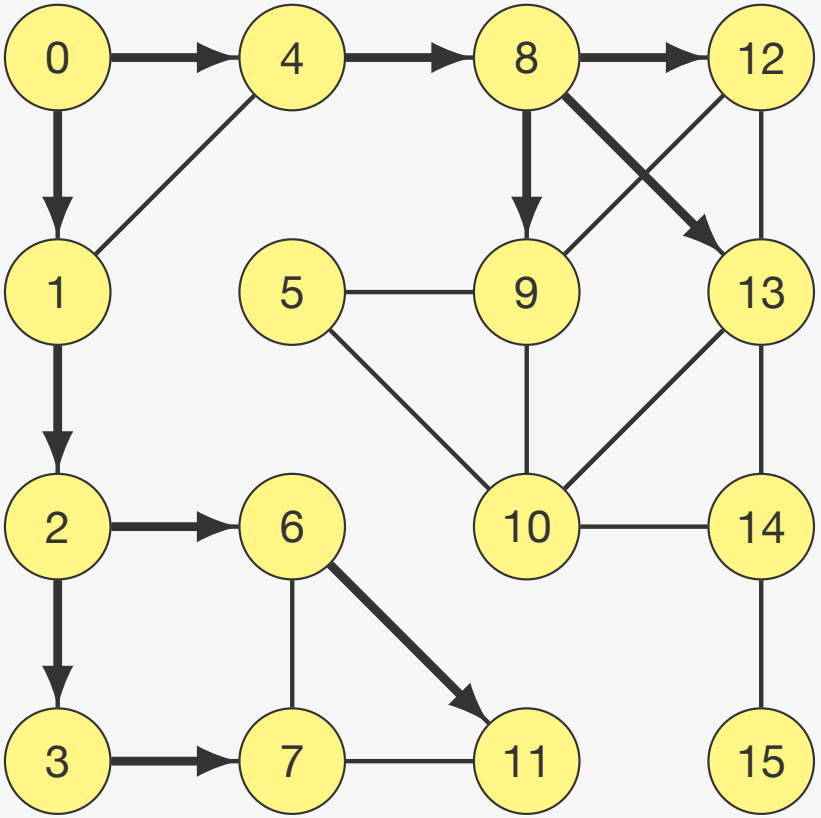
E se tivéssemos usando uma Fila?



Fila

9	12	13	7	11

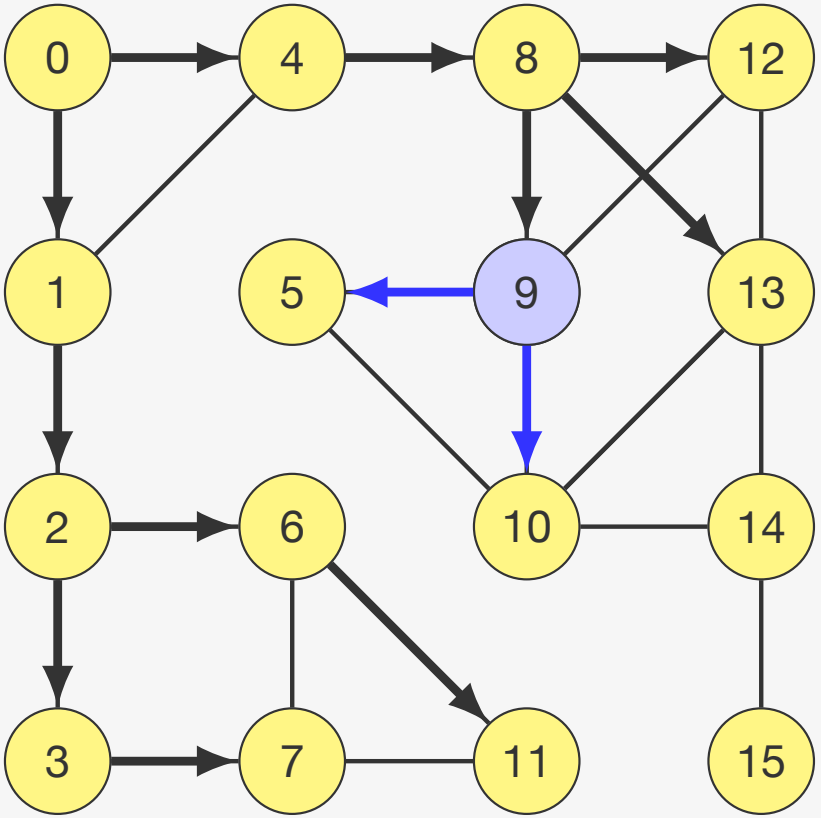
E se tivéssemos usando uma Fila?



Fila

9	12	13	7	11

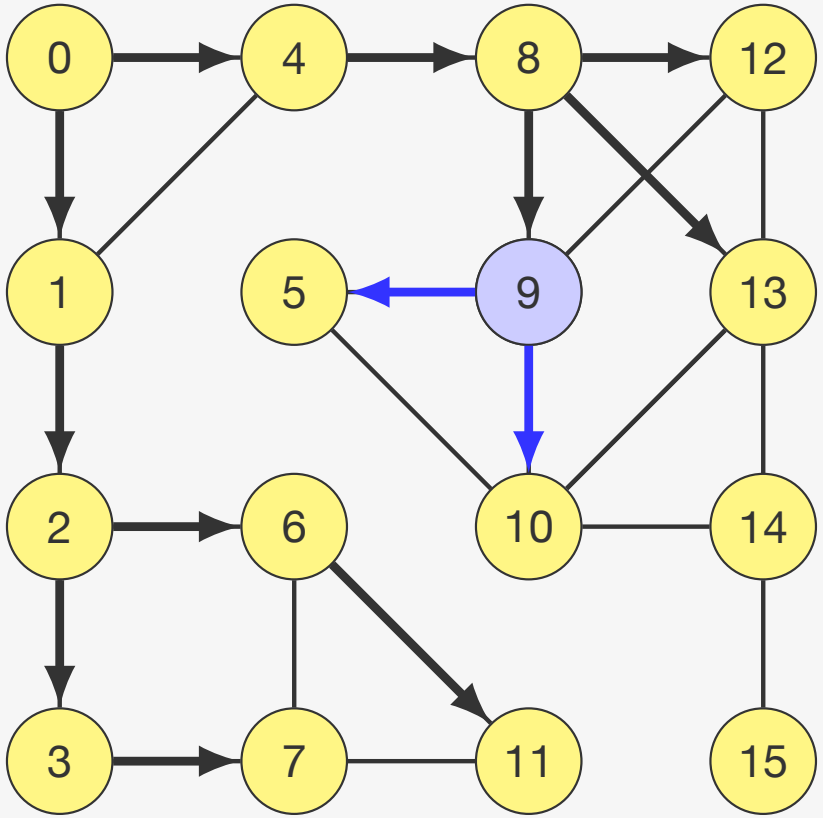
E se tivéssemos usando uma Fila?



Fila

12	13	7	11

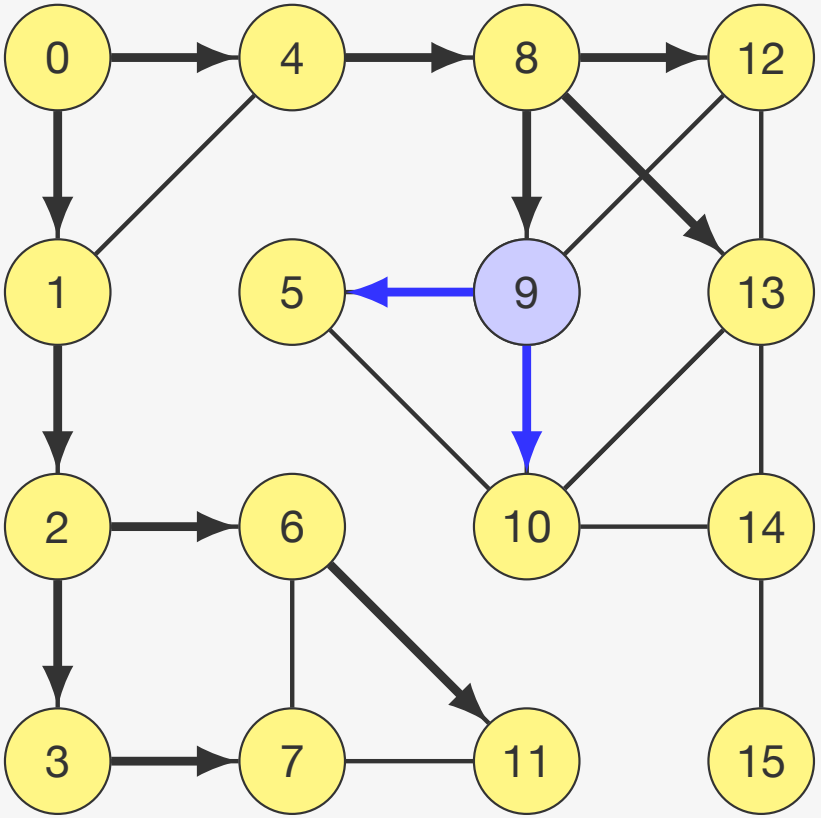
E se tivéssemos usando uma Fila?



Fila

12	13	7	11	5

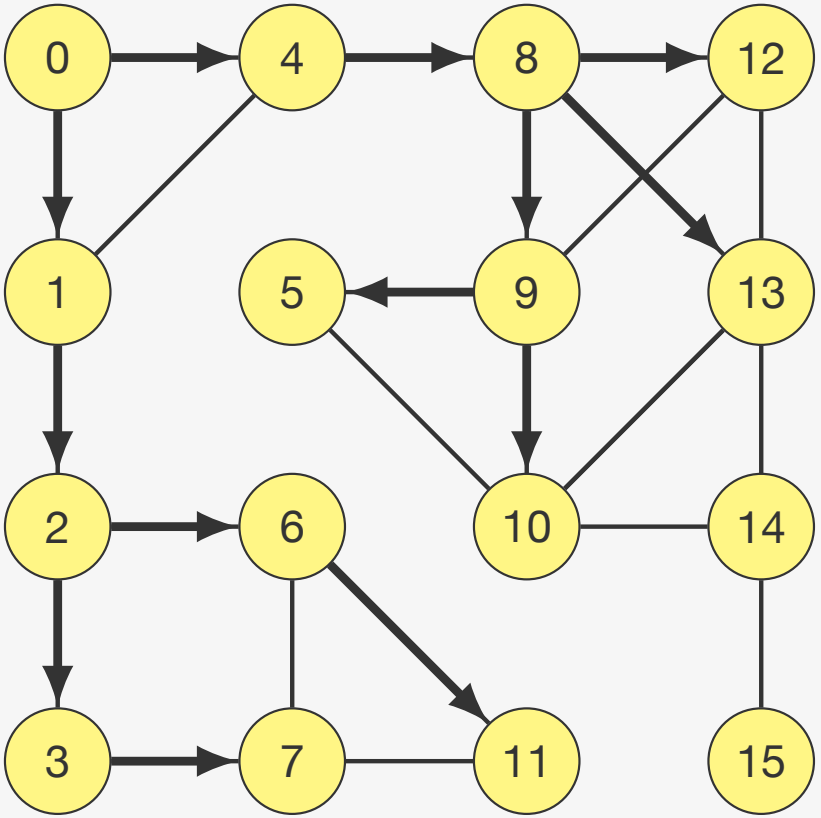
E se tivéssemos usando uma Fila?



Fila

12	13	7	11	5	10

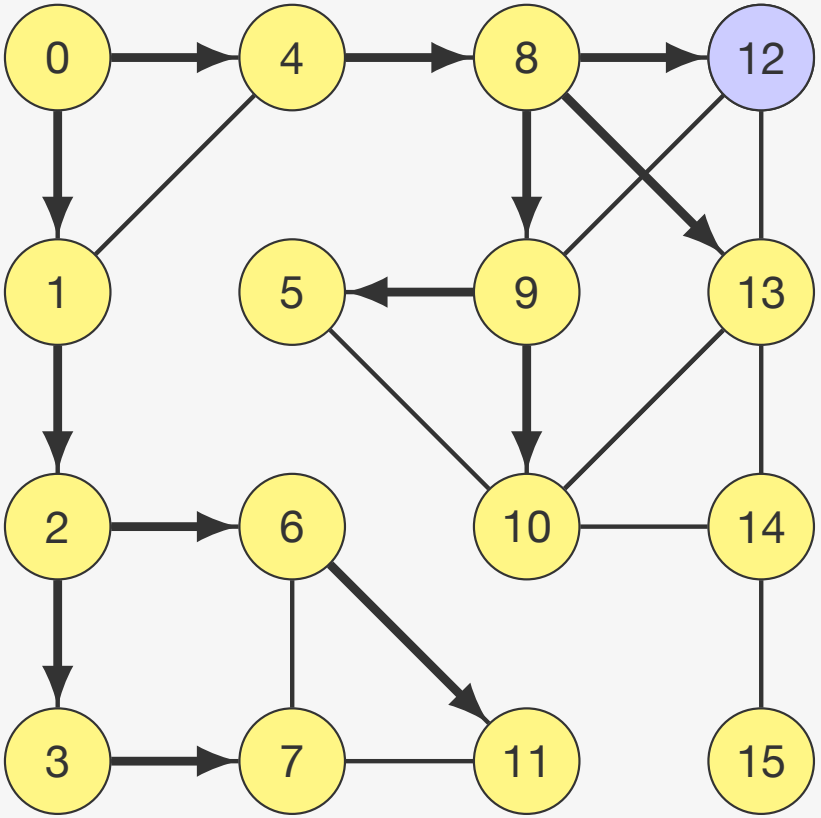
E se tivéssemos usando uma Fila?



Fila

12	13	7	11	5	10

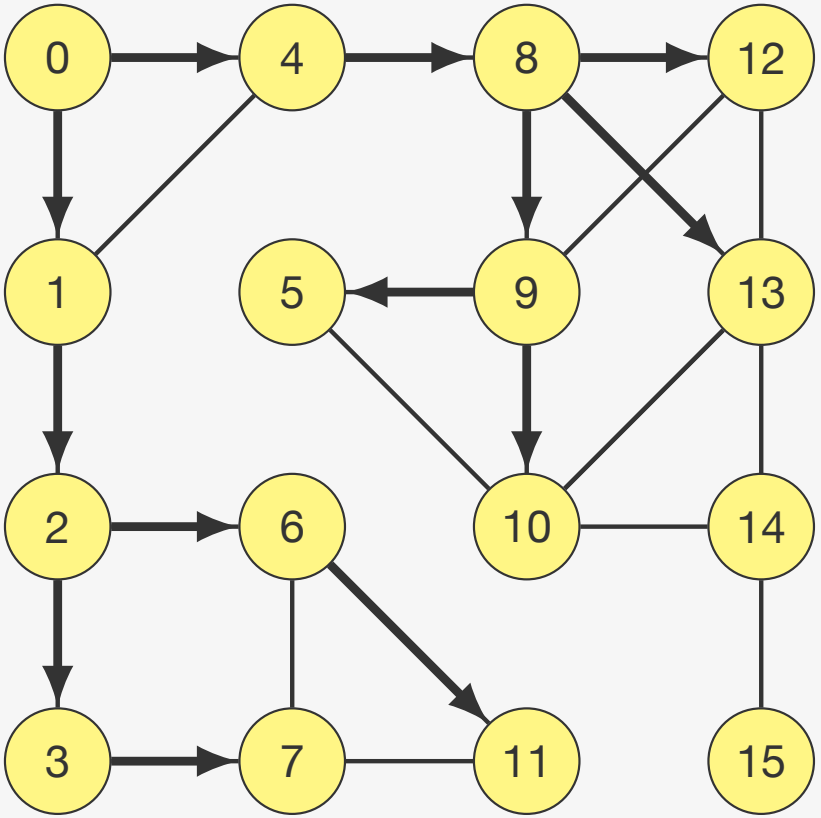
E se tivéssemos usando uma Fila?



Fila

13	7	11	5	10

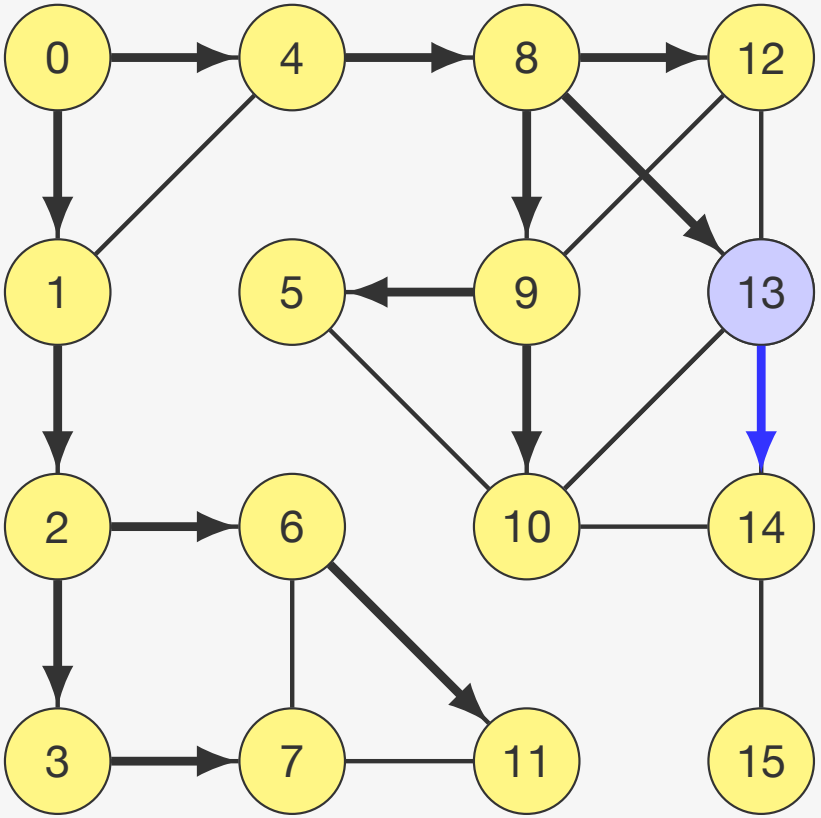
E se tivéssemos usando uma Fila?



Fila

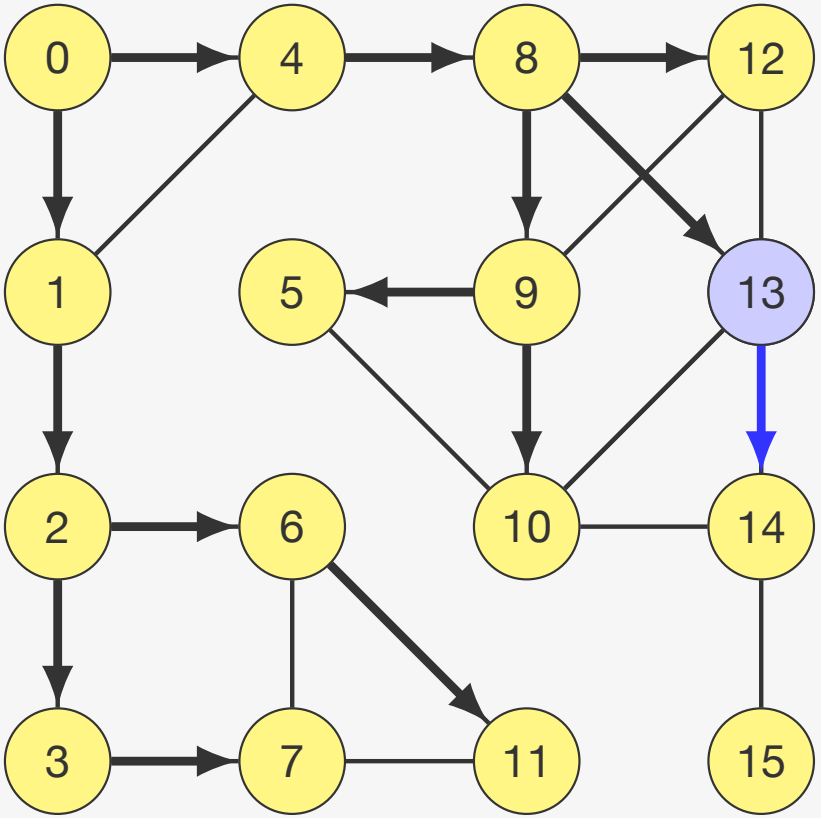
13	7	11	5	10

E se tivéssemos usando uma Fila?



Fila | 7 | 11 | 5 | 10 |

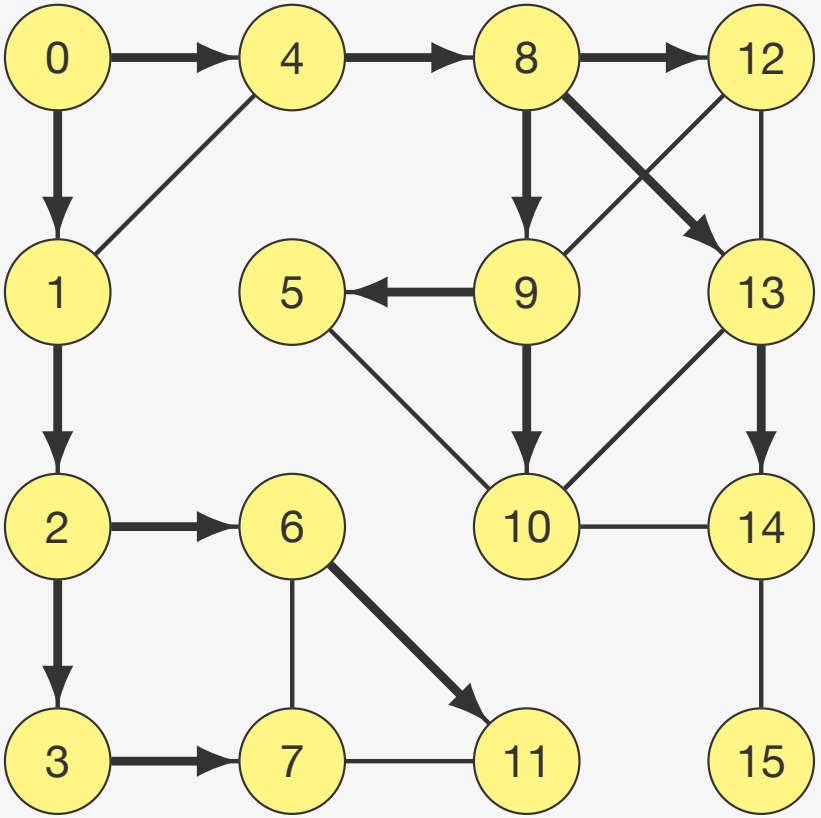
E se tivéssemos usando uma Fila?



Fila

7	11	5	10	14

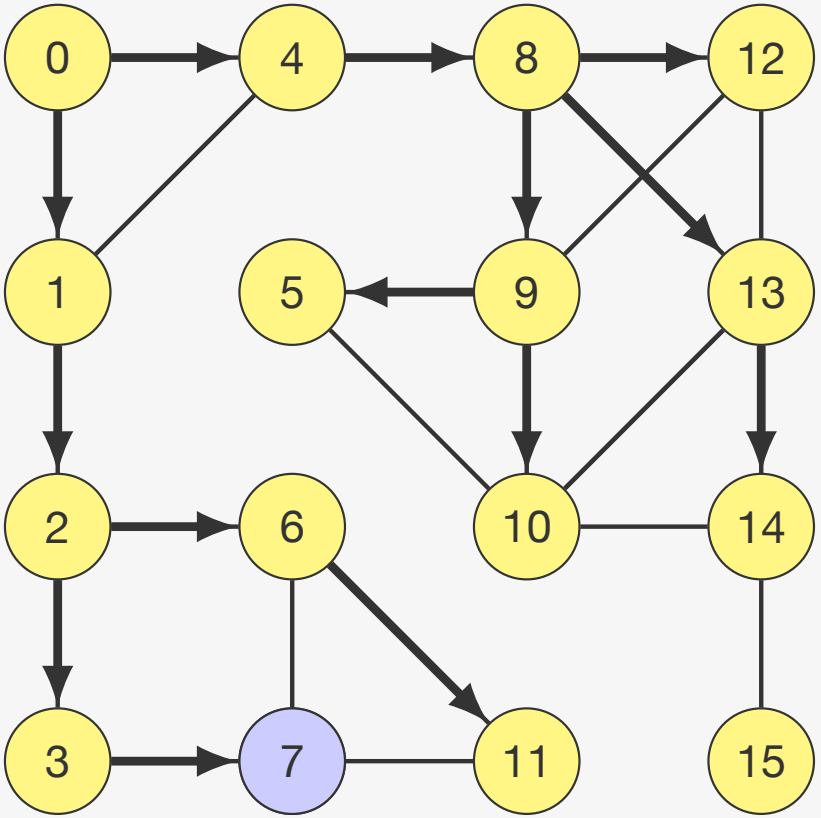
E se tivéssemos usando uma Fila?



Fila

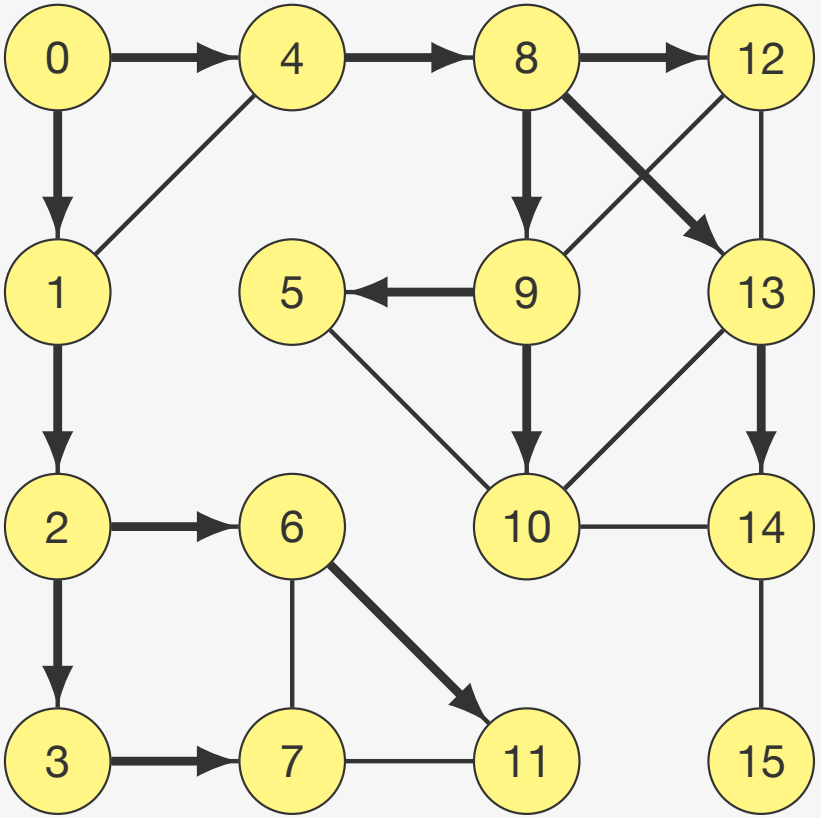
7	11	5	10	14

E se tivéssemos usando uma Fila?



Fila | 11 | 5 | 10 | 14 |

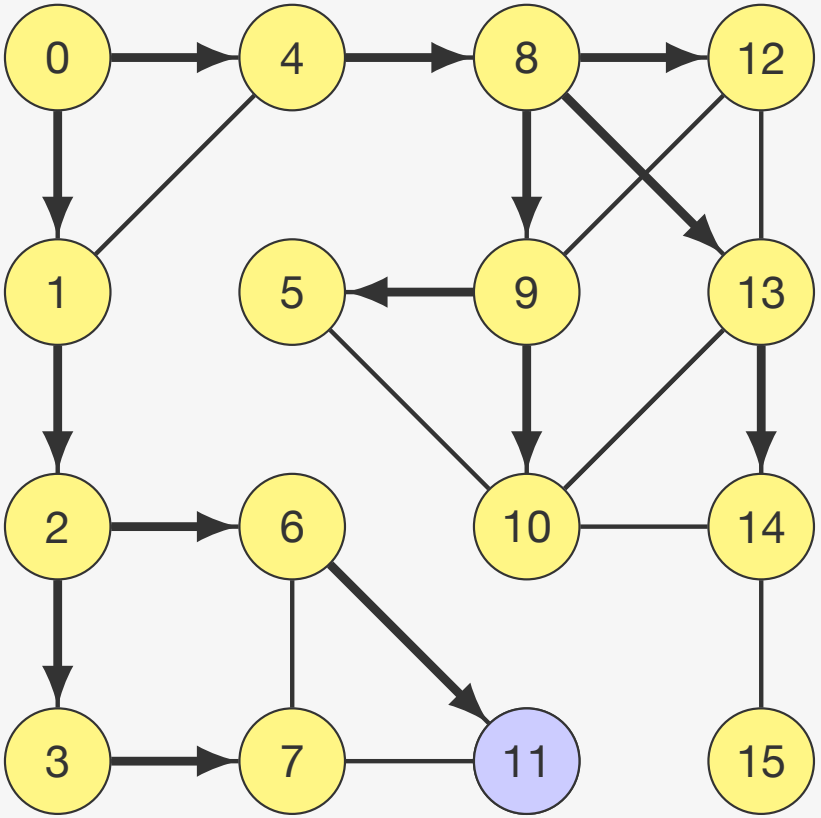
E se tivéssemos usando uma Fila?



Fila

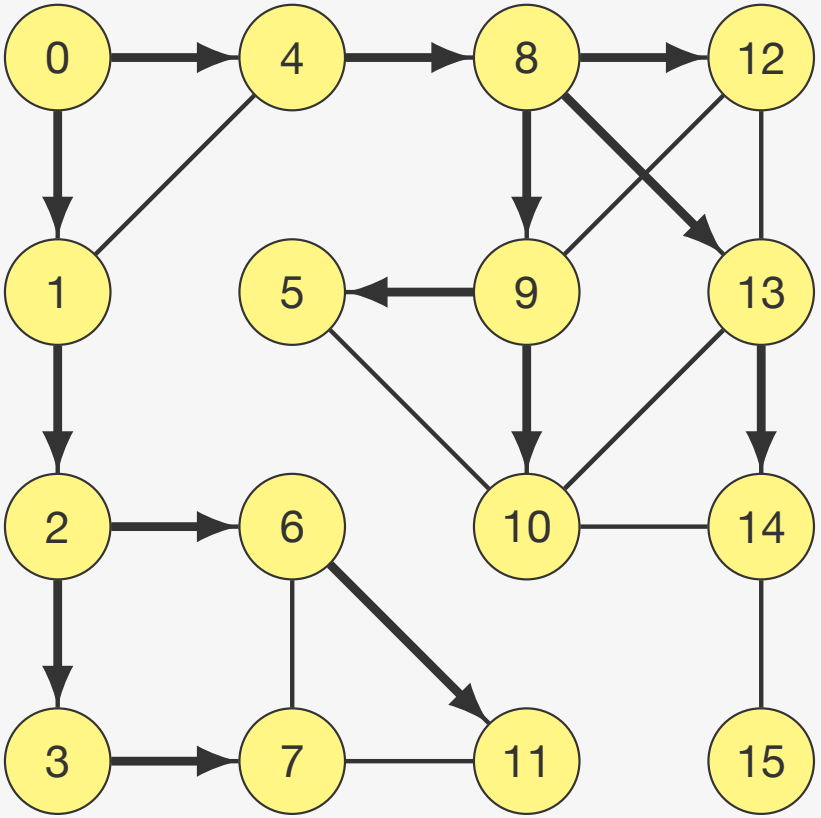
11	5	10	14

E se tivéssemos usando uma Fila?



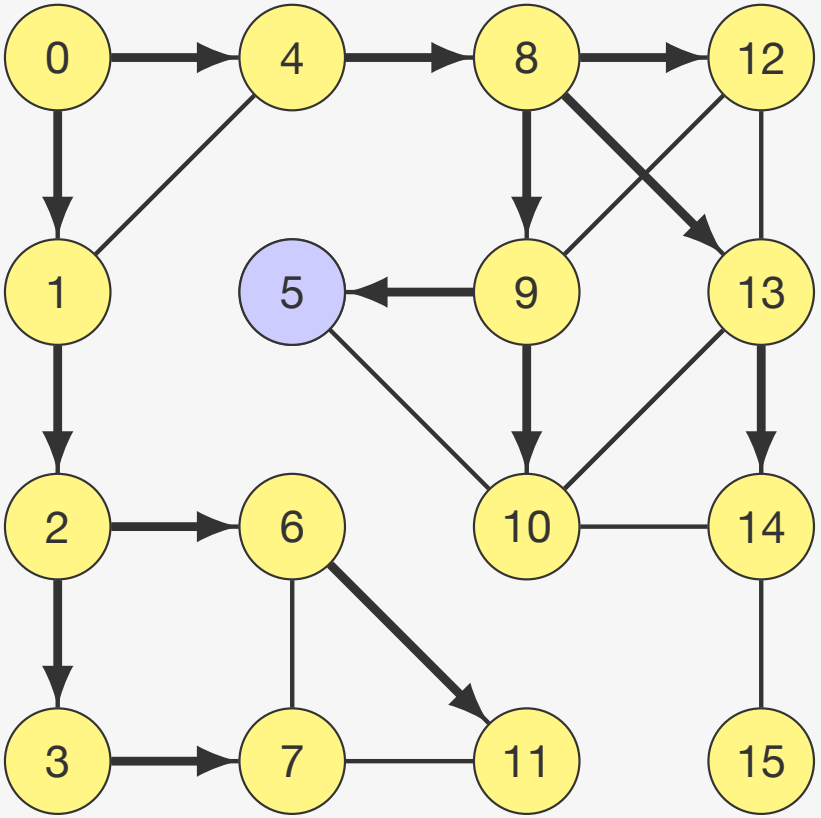
Fila | 5 | 10 | 14 |

E se tivéssemos usando uma Fila?



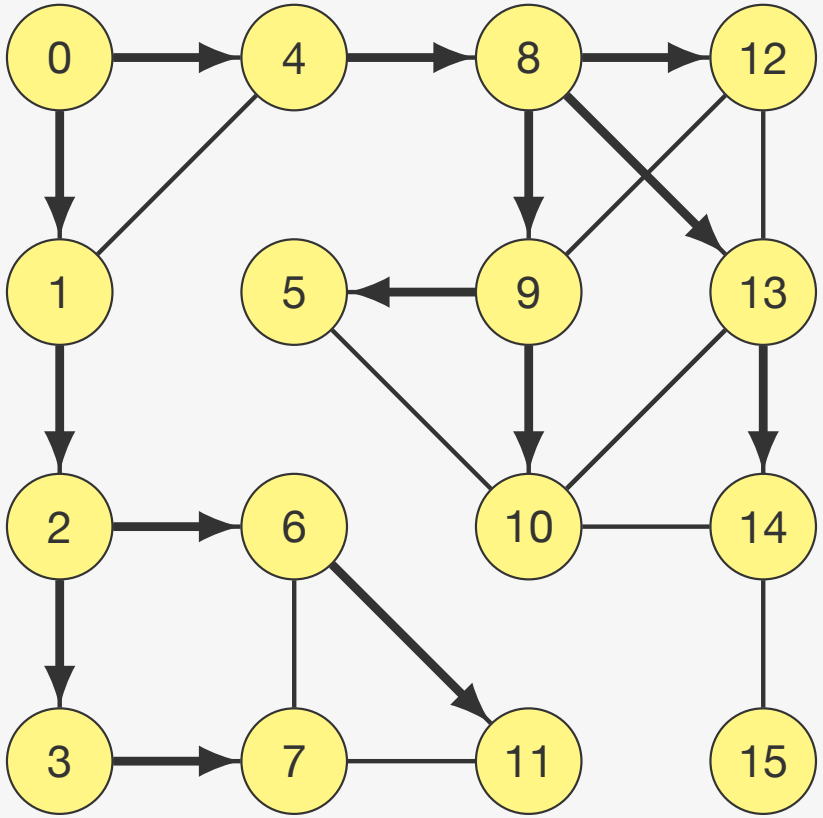
Fila | 5 | 10 | 14 |

E se tivéssemos usando uma Fila?



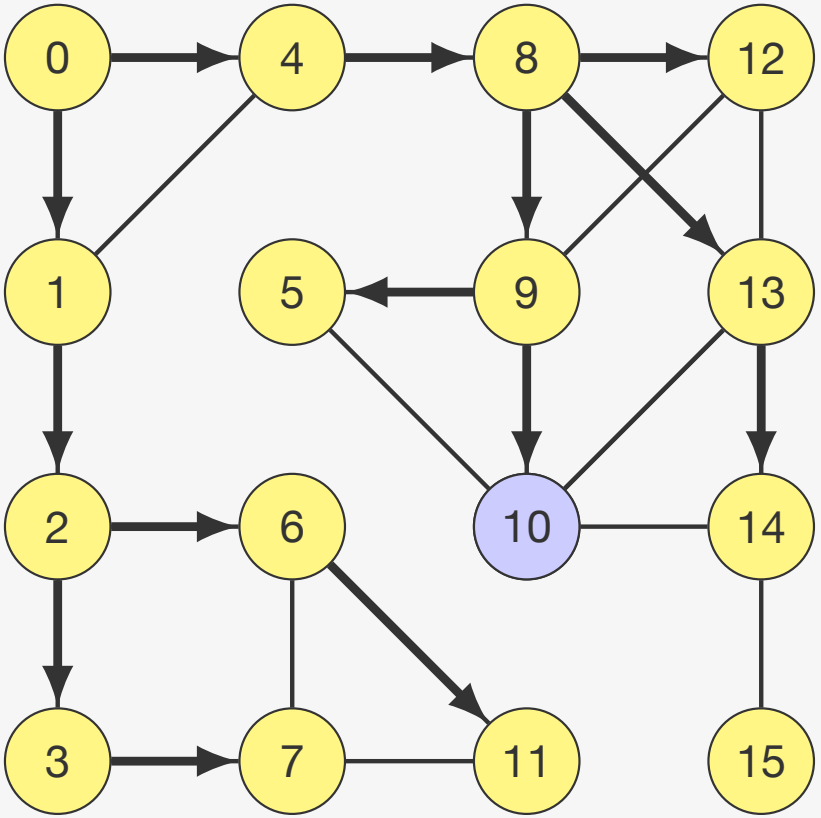
Fila | 10 | 14 |

E se tivéssemos usando uma Fila?



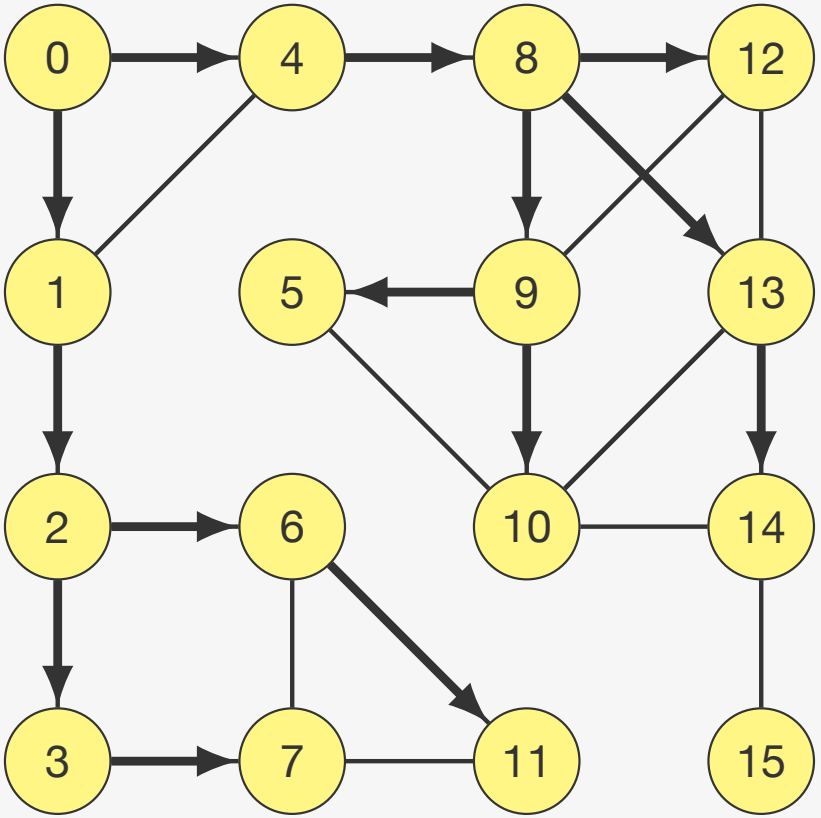
Fila | 10 | 14 |

E se tivéssemos usando uma Fila?



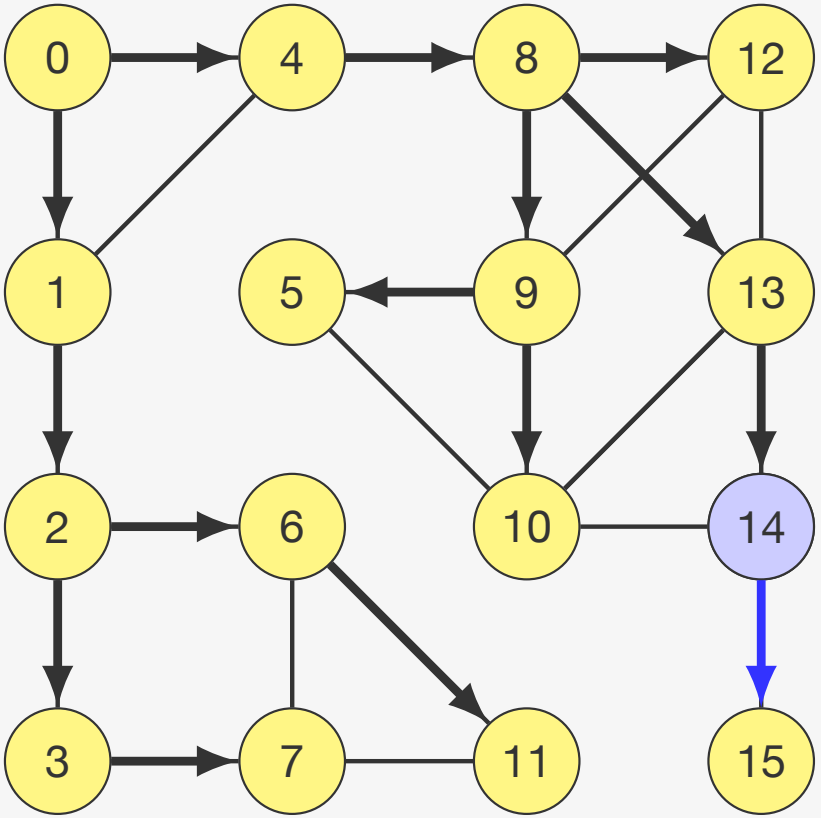
Fila | 14

E se tivéssemos usando uma Fila?



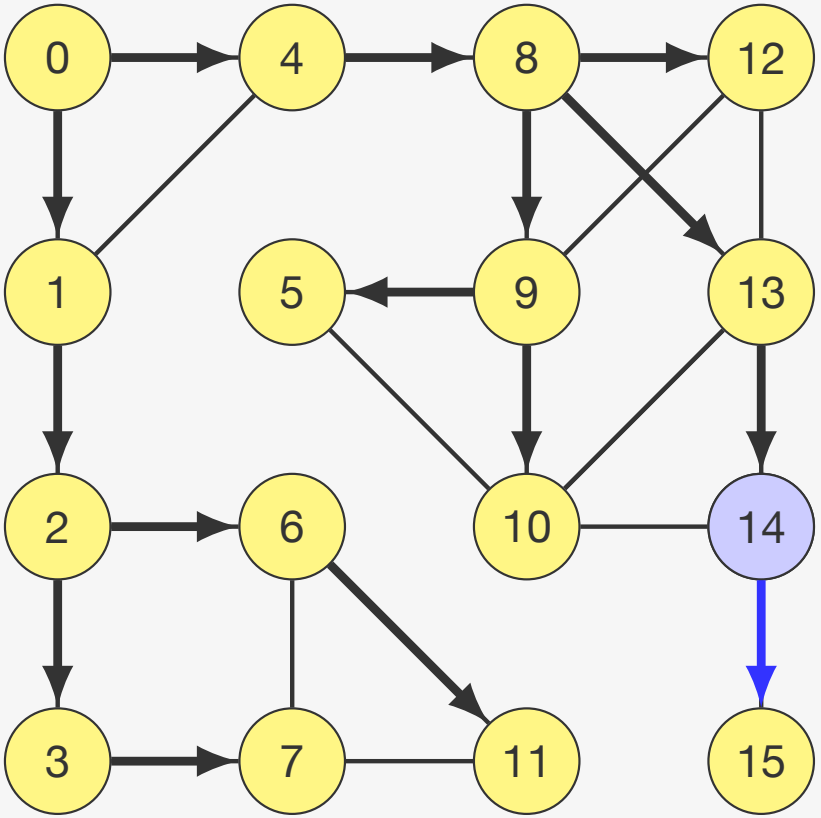
Fila | 14

E se tivéssemos usando uma Fila?



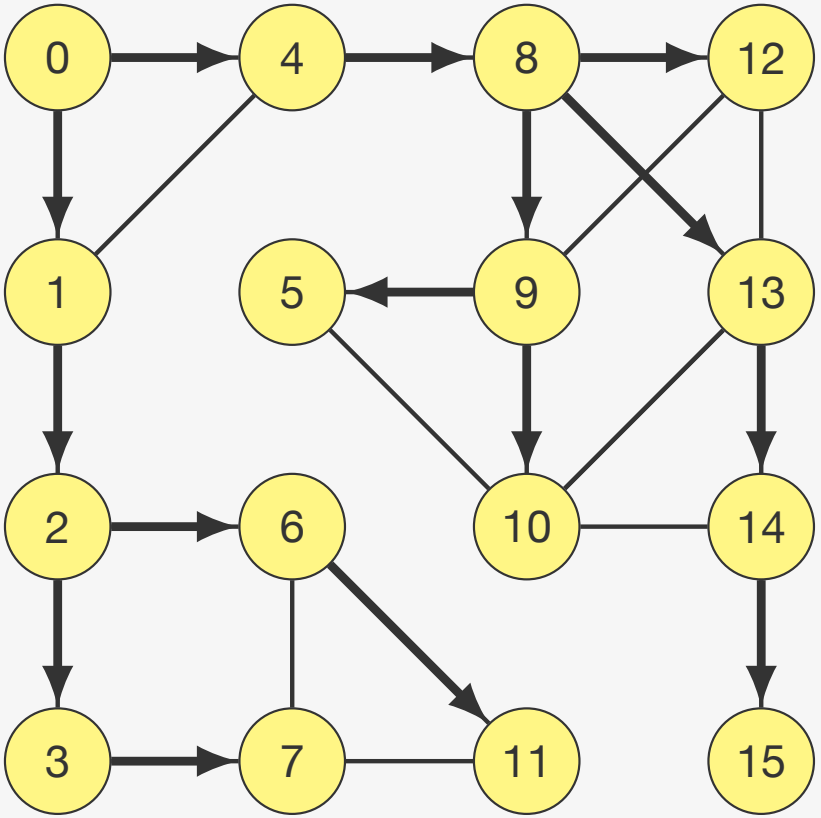
Fila

E se tivéssemos usando uma Fila?



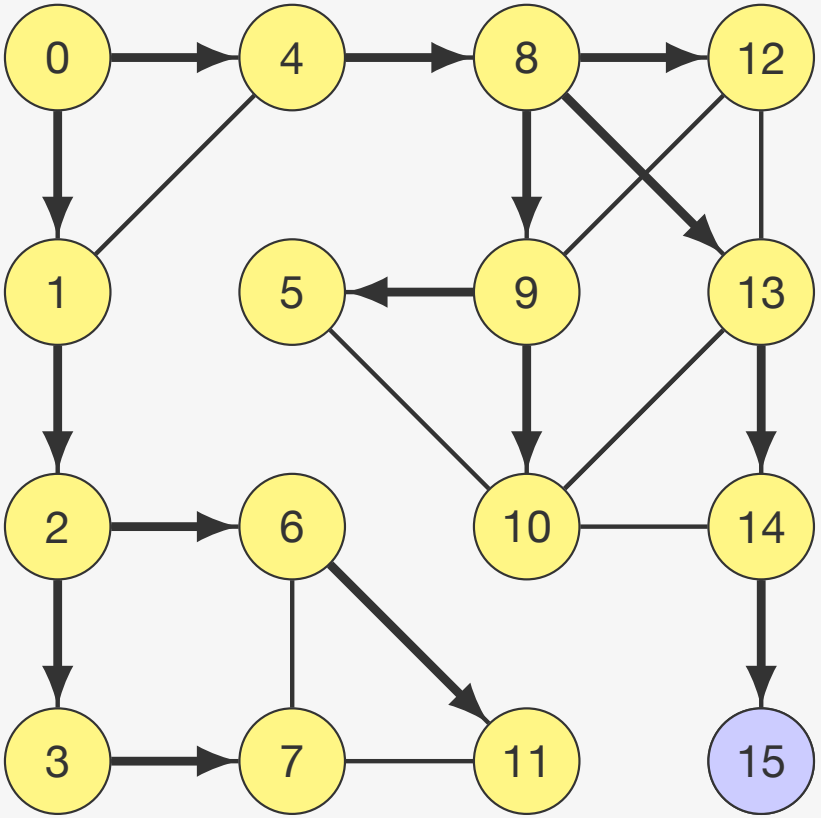
Fila | 15 |

E se tivéssemos usando uma Fila?



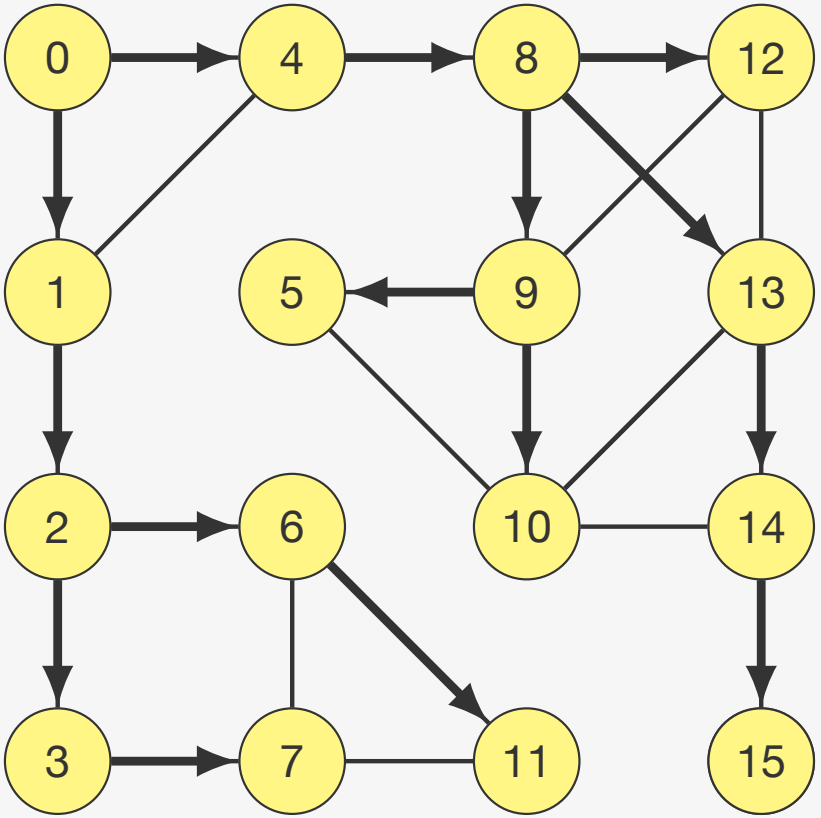
Fila | 15 |

E se tivéssemos usando uma Fila?



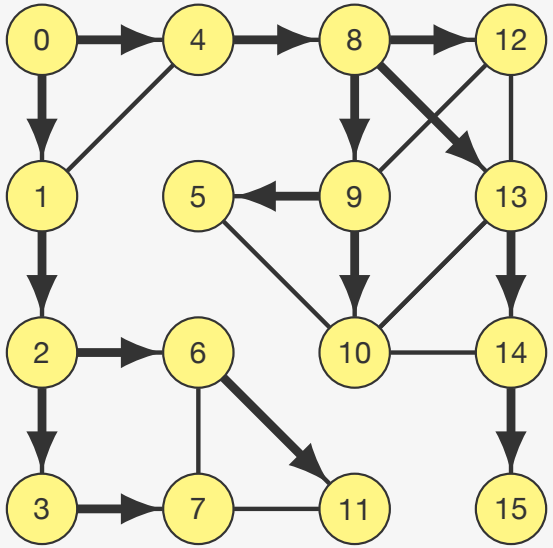
Fila

E se tivéssemos usando uma Fila?

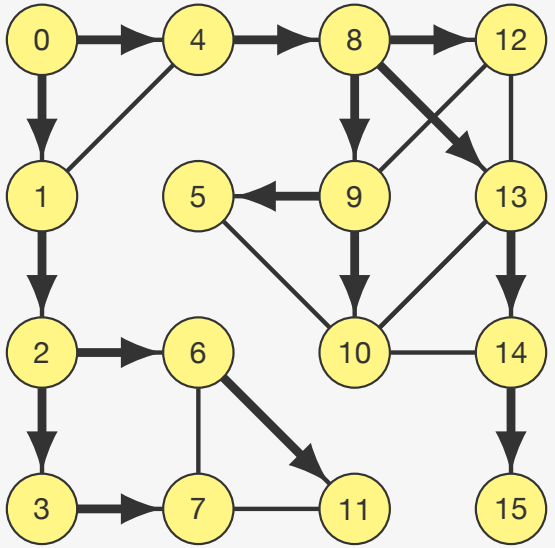


Fila

E se tivéssemos usando uma Fila?

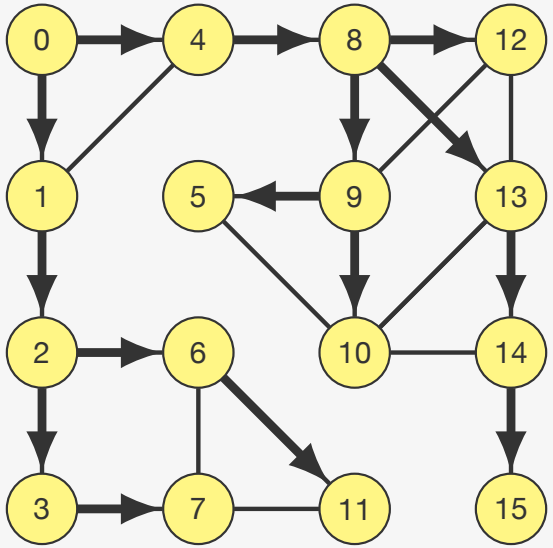


E se tivéssemos usando uma Fila?



Usando uma fila, visitamos primeiro os vértices mais próximos

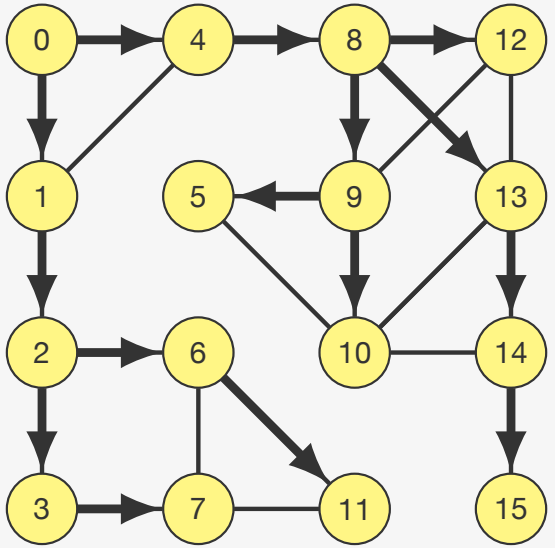
E se tivéssemos usando uma Fila?



Usando uma fila, visitamos primeiro os vértices mais próximos

- Enfileiramos os vizinhos de 0 (que estão a distância 1)

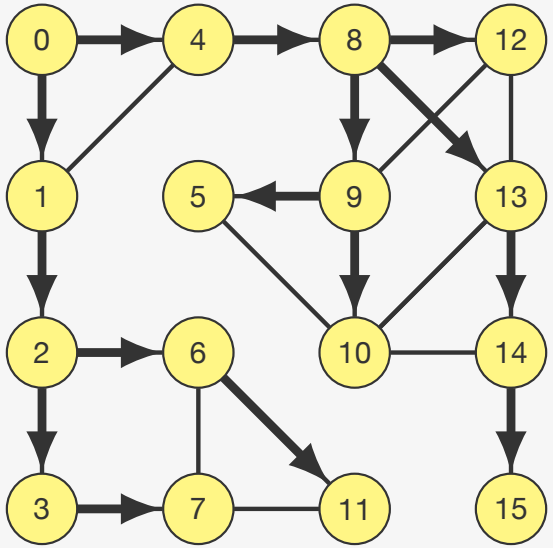
E se tivéssemos usando uma Fila?



Usando uma fila, visitamos primeiro os vértices mais próximos

- Enfileiramos os vizinhos de 0 (que estão a distância 1)
- Desenfileiramos um de seus vizinho

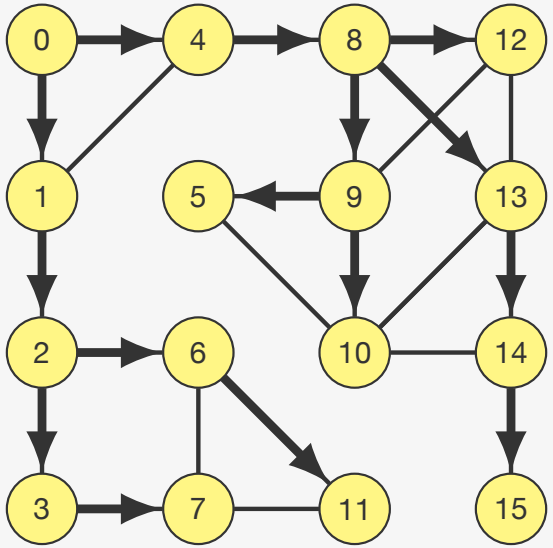
E se tivéssemos usando uma Fila?



Usando uma fila, visitamos primeiro os vértices mais próximos

- Enfileiramos os vizinhos de 0 (que estão a distância 1)
- Desenfileiramos um de seus vizinho
- E enfileiramos os vizinhos deste vértice

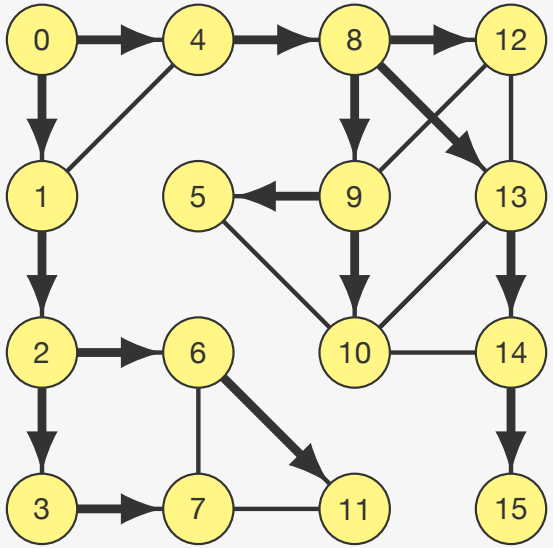
E se tivéssemos usando uma Fila?



Usando uma fila, visitamos primeiro os vértices mais próximos

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

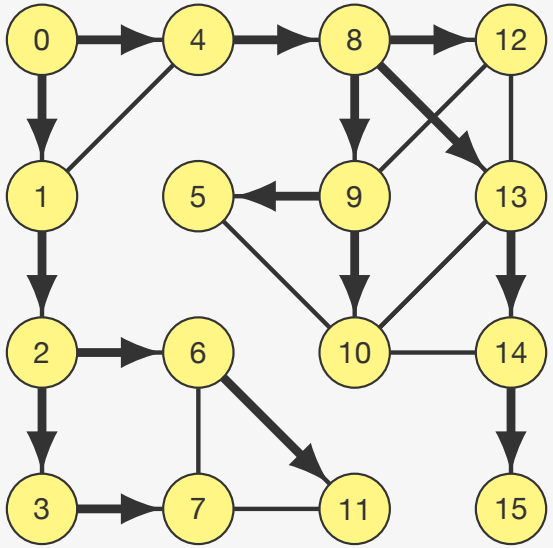
E se tivéssemos usando uma Fila?



Usando uma fila, visitamos primeiro os vértices mais próximos

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

E se tivéssemos usando 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 vizinho
- 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

```
1 int * busca_em_largura(p_grafo g, int s) {
```

Implementação da Busca em Largura

```
1 int * busca_em_largura(p_grafo g, int s) {
2   int w, v;
3   int *pai = malloc(g->n * sizeof(int));
4   int *visitado = malloc(g->n * sizeof(int));
```

Implementação da Busca em Largura

```
1 int * busca_em_largura(p_grafo g, int s) {
2   int w, v;
3   int *pai = malloc(g->n * sizeof(int));
4   int *visitado = malloc(g->n * sizeof(int));
5   p_fila f = criar_fila();
```

Implementação da Busca em Largura

```
1 int * busca_em_largura(p_grafo g, int s) {
2   int w, v;
3   int *pai = malloc(g->n * sizeof(int));
4   int *visitado = malloc(g->n * sizeof(int));
5   p_fila f = criar_fila();
6   for (v = 0; v < g->n; v++) {
7     pai[v] = -1;
8     visitado[v] = 0;
9   }
```

Implementação da Busca em Largura

```
1  int * busca_em_largura(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_fila f = criar_fila();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   enfileira(f,s);
```

Implementação da Busca em Largura

```
1  int * busca_em_largura(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_fila f = criar_fila();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   enfileira(f,s);
11   pai[s] = s;
```

Implementação da Busca em Largura

```
1  int * busca_em_largura(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_fila f = criar_fila();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   enfileira(f,s);
11   pai[s] = s;
12   visitado[s] = 1;
```

Implementação da Busca em Largura

```
1  int * busca_em_largura(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_fila f = criar_fila();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   enfileira(f,s);
11   pai[s] = s;
12   visitado[s] = 1;
13   while(!fila_vazia(f)) {
```

Implementação da Busca em Largura

```
1  int * busca_em_largura(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_fila f = criar_fila();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   enfileira(f,s);
11   pai[s] = s;
12   visitado[s] = 1;
13   while(!fila_vazia(f)) {
14     v = desenfileira(f);
```

Implementação da Busca em Largura

```
1  int * busca_em_largura(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_fila f = criar_fila();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   enfileira(f,s);
11   pai[s] = s;
12   visitado[s] = 1;
13   while(!fila_vazia(f)) {
14     v = desenfileira(f);
15     for (w = 0; w < g->n; w++)
```

Implementação da Busca em Largura

```
1  int * busca_em_largura(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_fila f = criar_fila();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   enfileira(f,s);
11   pai[s] = s;
12   visitado[s] = 1;
13   while(!fila_vazia(f)) {
14     v = desenfileira(f);
15     for (w = 0; w < g->n; w++)
16       if (g->adj[v][w] && !visitado[w]) {
```

Implementação da Busca em Largura

```
1  int * busca_em_largura(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_fila f = criar_fila();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   enfileira(f,s);
11   pai[s] = s;
12   visitado[s] = 1;
13   while(!fila_vazia(f)) {
14     v = desenfileira(f);
15     for (w = 0; w < g->n; w++)
16       if (g->adj[v][w] && !visitado[w]) {
17         visitado[w] = 1;/*evita repetição na fila*/
```

Implementação da Busca em Largura

```
1  int * busca_em_largura(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_fila f = criar_fila();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   enfileira(f,s);
11   pai[s] = s;
12   visitado[s] = 1;
13   while(!fila_vazia(f)) {
14     v = desenfileira(f);
15     for (w = 0; w < g->n; w++)
16       if (g->adj[v][w] && !visitado[w]) {
17         visitado[w] = 1;/*evita repetição na fila*/
18         pai[w] = v;
```

Implementação da Busca em Largura

```
1  int * busca_em_largura(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_fila f = criar_fila();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   enfileira(f,s);
11   pai[s] = s;
12   visitado[s] = 1;
13   while(!fila_vazia(f)) {
14     v = desenfileira(f);
15     for (w = 0; w < g->n; w++)
16       if (g->adj[v][w] && !visitado[w]) {
17         visitado[w] = 1;/*evita repetição na fila*/
18         pai[w] = v;
19         enfileira(f, w);
```

Implementação da Busca em Largura

```
1  int * busca_em_largura(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_fila f = criar_fila();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   enfileira(f,s);
11   pai[s] = s;
12   visitado[s] = 1;
13   while(!fila_vazia(f)) {
14     v = desenfileira(f);
15     for (w = 0; w < g->n; w++)
16       if (g->adj[v][w] && !visitado[w]) {
17         visitado[w] = 1;/*evita repetição na fila*/
18         pai[w] = v;
19         enfileira(f, w);
20       }
21   }
```

Implementação da Busca em Largura

```
1  int * busca_em_largura(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_fila f = criar_fila();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   enfileira(f,s);
11   pai[s] = s;
12   visitado[s] = 1;
13   while(!fila_vazia(f)) {
14     v = desenfileira(f);
15     for (w = 0; w < g->n; w++)
16       if (g->adj[v][w] && !visitado[w]) {
17         visitado[w] = 1;/*evita repetição na fila*/
18         pai[w] = v;
19         enfileira(f, w);
20       }
21   }
22   destroi_fila(f);
```

Implementação da Busca em Largura

```
1  int * busca_em_largura(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_fila f = criar_fila();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   enfileira(f,s);
11   pai[s] = s;
12   visitado[s] = 1;
13   while(!fila_vazia(f)) {
14     v = desenfileira(f);
15     for (w = 0; w < g->n; w++)
16       if (g->adj[v][w] && !visitado[w]) {
17         visitado[w] = 1;/*evita repetição na fila*/
18         pai[w] = v;
19         enfileira(f, w);
20       }
21   }
22   destroi_fila(f);
23   free(visitado);
```

Implementação da Busca em Largura

```
1  int * busca_em_largura(p_grafo g, int s) {
2    int w, v;
3    int *pai = malloc(g->n * sizeof(int));
4    int *visitado = malloc(g->n * sizeof(int));
5    p_fila f = criar_fila();
6    for (v = 0; v < g->n; v++) {
7      pai[v] = -1;
8      visitado[v] = 0;
9    }
10   enfileira(f,s);
11   pai[s] = s;
12   visitado[s] = 1;
13   while(!fila_vazia(f)) {
14     v = desenfileira(f);
15     for (w = 0; w < g->n; w++)
16       if (g->adj[v][w] && !visitado[w]) {
17         visitado[w] = 1;/*evita repetição na fila*/
18         pai[w] = v;
19         enfileira(f, w);
20       }
21   }
22   destroi_fila(f);
23   free(visitado);
24   return pai;
25 }
```

Tempo para fazer a busca

Quanto tempo demora para fazer uma busca?

Tempo para fazer a busca

Quanto tempo demora para fazer uma busca?

- em profundidade ou em largura

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

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 $\mathrm{O}(1)$

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

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

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

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 $\mathrm{O}(n^2)$

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?

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

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 $\mathrm{O}(\max\{n, m\}) = \mathrm{O}(n + m)$

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