Programação Competitiva

Aula 6 - BFS

Emanuel Juliano

Universidade Federal de Minas Gerais

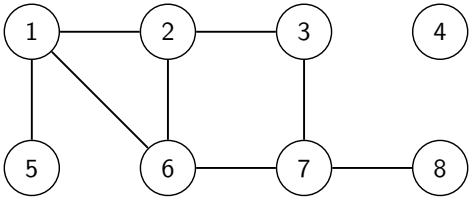
4 de Setembro de 2020

Motivação: Message Route

- Dada uma rede de N computadores e M conexões, queremos saber se é possível enviar uma mensagem do computador 1 para o computador N. Caso seja possível, qual o menor número de computadores nessa rota e quais são eles?

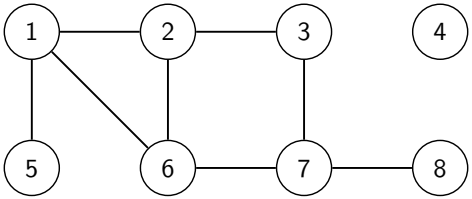
Motivação: Message Route

- Dada uma rede de N computadores e M conexões, queremos saber se é possível enviar uma mensagem do computador 1 para o computador N. Caso seja possível, qual o menor número de computadores nessa rota e quais são eles?
- Exemplo: N=8 e M=8



Motivação: Message Route

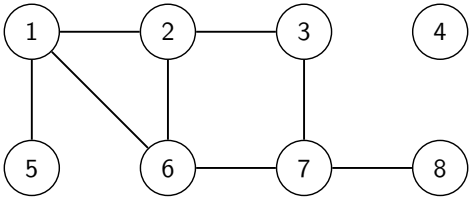
- Dada uma rede de N computadores e M conexões, queremos saber se é possível enviar uma mensagem do computador 1 para o computador N. Caso seja possível, qual o menor número de computadores nessa rota e quais são eles?
- Exemplo: N=8 e M=8



- Possíveis caminhos: $\{1, 2, 3, 7, 8\}$, $\{1, 2, 6, 7, 8\}$, $\{1, 6, 7, 8\}$

Motivação: Message Route

- Dada uma rede de N computadores e M conexões, queremos saber se é possível enviar uma mensagem do computador 1 para o computador N. Caso seja possível, qual o menor número de computadores nessa rota e quais são eles?
- Exemplo: N=8 e M=8



- Possíveis caminhos: $\{1, 2, 3, 7, 8\}$, $\{1, 2, 6, 7, 8\}$, $\{1, 6, 7, 8\}$
- Como fazer um algoritmo que descobre o menor?

BFS

- Da necessidade de um algoritmo que descobre o menor caminho entre dois vértices, surge a **Breadth First Search (BFS)**, ou **Busca em Largura**

BFS

- Da necessidade de um algoritmo que descobre o menor caminho entre dois vértices, surge a **Breadth First Search (BFS)**, ou **Busca em Largura**
- O algoritmo pode ser entendido como um fogo se espalhando pelo grafo: partindo de um vértice s, visitamos todos os vizinhos, depois todos os vizinhos dos vizinhos, e assim sucessivamente.

BFS

- Da necessidade de um algoritmo que descobre o menor caminho entre dois vértices, surge a **Breadth First Search (BFS)**, ou **Busca em Largura**
- O algoritmo pode ser entendido como um fogo se espalhando pelo grafo: partindo de um vértice s, visitamos todos os vizinhos, depois todos os vizinhos dos vizinhos, e assim sucessivamente.
- Sempre visitamos os vértices a uma distância $d - 1$ antes de visitar os vértices a uma distância d.

BFS

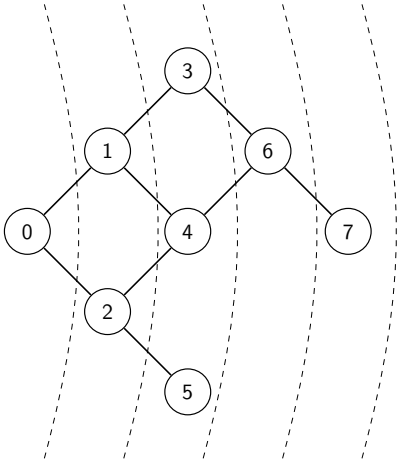
- Da necessidade de um algoritmo que descobre o menor caminho entre dois vértices, surge a **Breadth First Search (BFS)**, ou **Busca em Largura**
- O algoritmo pode ser entendido como um fogo se espalhando pelo grafo: partindo de um vértice s, visitamos todos os vizinhos, depois todos os vizinhos dos vizinhos, e assim sucessivamente.
- Sempre visitamos os vértices a uma distância $d - 1$ antes de visitar os vértices a uma distância d.
- Complexidade: $\mathcal{O}(|V|+|E|)$

Exemplo

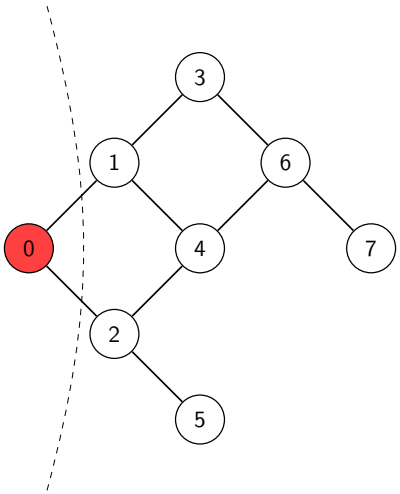
- A partir de um grafo qualquer, sempre podemos desenhá-lo a partir da sua distância da fonte:

Exemplo

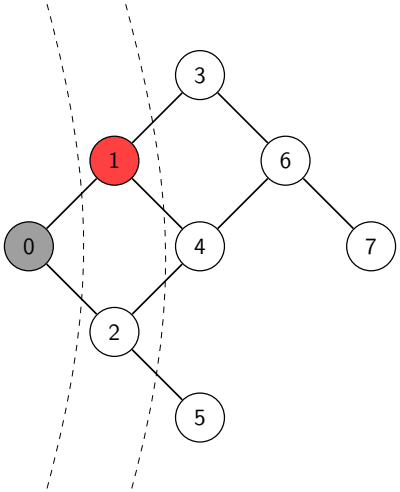
- A partir de um grafo qualquer, sempre podemos desenhá-lo a partir da sua distância da fonte:



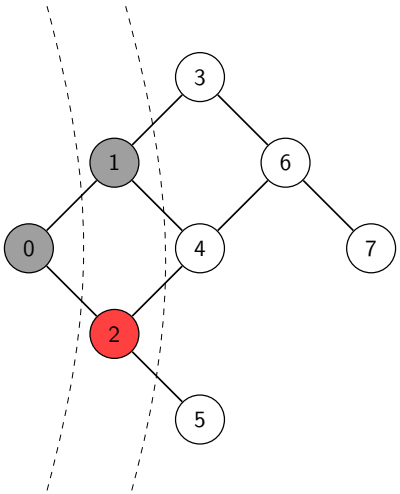
Exemplo



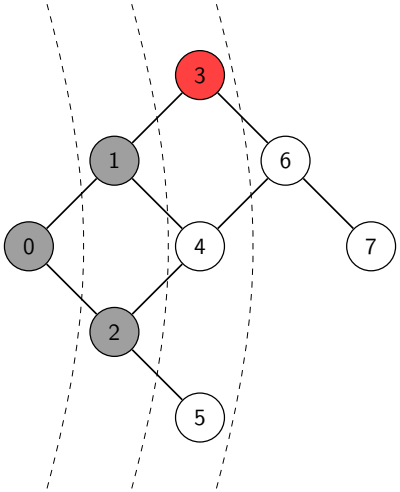
Exemplo



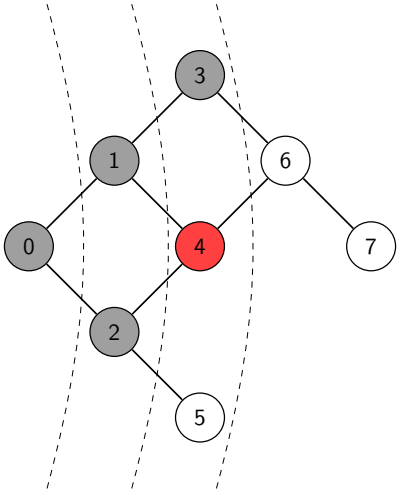
Exemplo



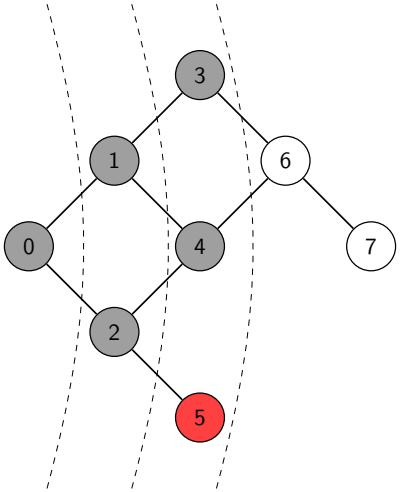
Exemplo



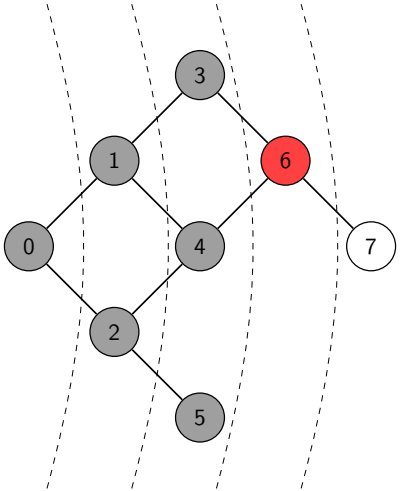
Exemplo



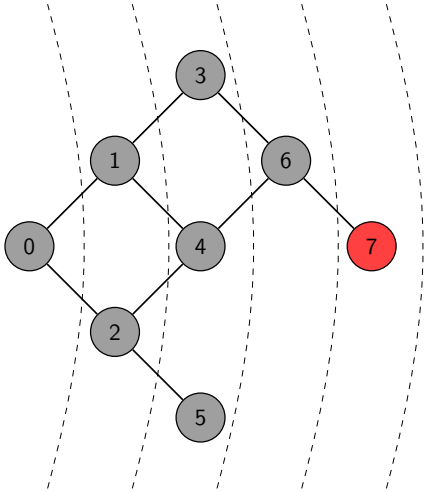
Exemplo



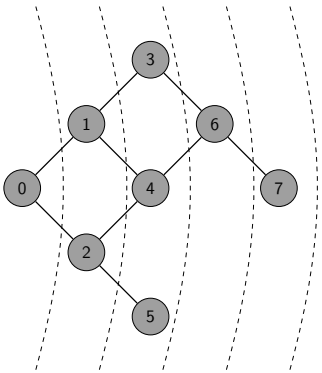
Exemplo



Exemplo



Exemplo



- Como visitamos primeiro os vértices a uma distância d da fonte antes de visitar os vértices a uma distância $d + 1$, precisamos de uma estrutura que prioriza os elementos que colocamos primeiro, tal qual uma fila.

queue

- queue (fila) é uma estrutura muito similar ao vector, porém, aqui os elementos são inseridos no final da fila e removidos no começo.

queue

- queue (fila) é uma estrutura muito similar ao `vector`, porém, aqui os elementos são inseridos no final da fila e removidos no começo.

```
1  queue<int> q;
2
3  q.push(1), q.push(2), q.push(3);
4  # q = {1, 2, 3};
5  cout << q.front() << endl;
6
7  q.pop();
8  # q = {2, 3};
9  cout << q.front() << endl;
```

- Saída:

  1

  2

O Algoritmo

- Criamos uma fila `q` que conterá os vértices a serem processados e nosso vetor de visitados `vis`.

O Algoritmo

- Criamos uma fila `q` que conterá os vértices a serem processados e nosso vetor de visitados `vis`.
- Inicialmente, colocamos nossa fonte `s` na fila (`q.push(s)`) e a marcamos como visitada (`vis[s] = true`).

O Algoritmo

- Criamos uma fila `q` que conterá os vértices a serem processados e nosso vetor de visitados `vis`.
- Inicialmente, colocamos nossa fonte `s` na fila (`q.push(s)`) e a marcamos como visitada (`vis[s] = true`).
- Enquanto a fila não estiver vazia, vamos processar a frente da nossa fila (chamaremos esse vértice de `v`):

O Algoritmo

- Criamos uma fila `q` que conterá os vértices a serem processados e nosso vetor de visitados `vis`.
- Inicialmente, colocamos nossa fonte `s` na fila (`q.push(s)`) e a marcamos como visitada (`vis[s] = true`).
- Enquanto a fila não estiver vazia, vamos processar a frente da nossa fila (chamaremos esse vértice de `v`):
  - Removemos `v` da fila (`q.pop()`) e o marcamos como visitado (`vis[v] = true`).

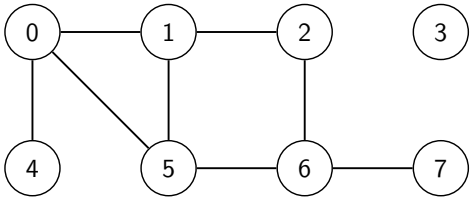
O Algoritmo

- Criamos uma fila `q` que conterá os vértices a serem processados e nosso vetor de visitados `vis`.
- Inicialmente, colocamos nossa fonte `s` na fila (`q.push(s)`) e a marcamos como visitada (`vis[s] = true`).
- Enquanto a fila não estiver vazia, vamos processar a frente da nossa fila (chamaremos esse vértice de `v`):
  - Removemos `v` da fila (`q.pop()`) e o marcamos como visitado (`vis[v] = true`).
  - Olhamos para todos os vizinhos não vistos de `v` e colocamos eles na fila.

Executando o Algoritmo

Vértice	Vizinhos	Visitado
0	{1,4,5}	N
1	{0,2,5}	N
2	{1,6}	N
3	{}	N
4	{0}	N
5	{0,1,6}	N
6	{2,5,7}	N
7	{6}	N

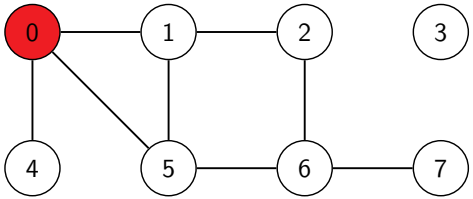
Fila $= \{\}$



Executando o Algoritmo

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	N
2	{1,6}	N
3	{}	N
4	{0}	N
5	{0,1,6}	N
6	{2,5,7}	N
7	{6}	N

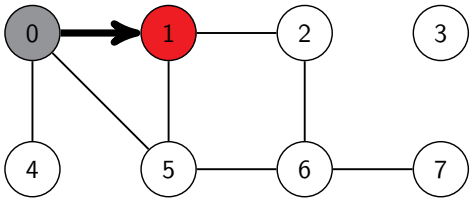
Fila $= \{0\}$



Executando o Algoritmo

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,6}	N
3	{}	N
4	{0}	N
5	{0,1,6}	N
6	{2,5,7}	N
7	{6}	N

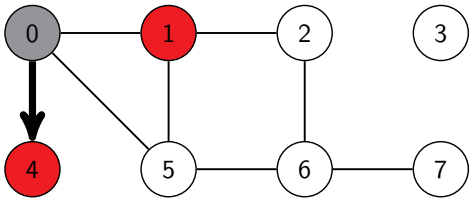
Fila $= \{1\}$



Executando o Algoritmo

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,6}	N
3	{}	N
4	{0}	S
5	{0,1,6}	N
6	{2,5,7}	N
7	{6}	N

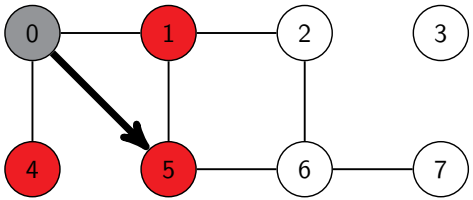
Fila $= \{1, 4\}$



Executando o Algoritmo

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,6}	N
3	{}	N
4	{0}	S
5	{0,1,6}	S
6	{2,5,7}	N
7	{6}	N

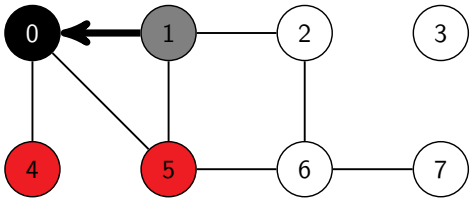
Fila $= \{1, 4, 5\}$



Executando o Algoritmo

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,6}	N
3	{}	N
4	{0}	S
5	{0,1,6}	S
6	{2,5,7}	N
7	{6}	N

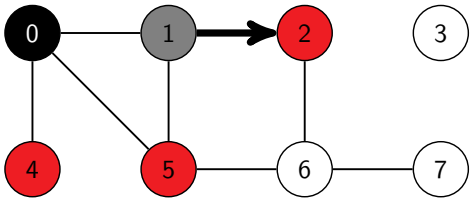
Fila $= \{4, 5\}$



Executando o Algoritmo

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,6}	S
3	{}	N
4	{0}	S
5	{0,1,6}	S
6	{2,5,7}	N
7	{6}	N

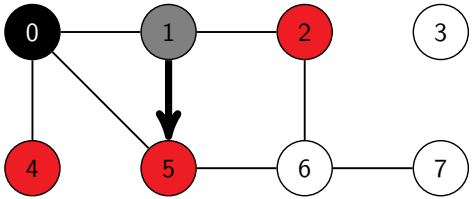
Fila = {4, 5, 2}



Executando o Algoritmo

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,6}	S
3	{}	N
4	{0}	S
5	{0,1,6}	S
6	{2,5,7}	N
7	{6}	N

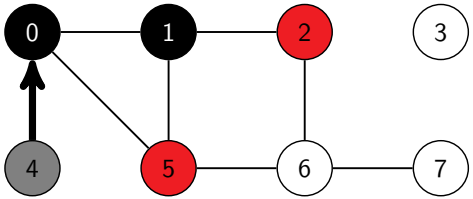
Fila = {4, 5, 2}



Executando o Algoritmo

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,6}	S
3	{}	N
4	{0}	S
5	{0,1,6}	S
6	{2,5,7}	N
7	{6}	N

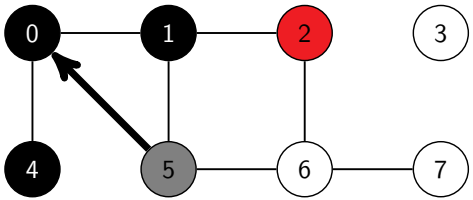
Fila $= \{5, 2\}$



Executando o Algoritmo

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,6}	S
3	{}	N
4	{0}	S
5	{0,1,6}	S
6	{2,5,7}	N
7	{6}	N

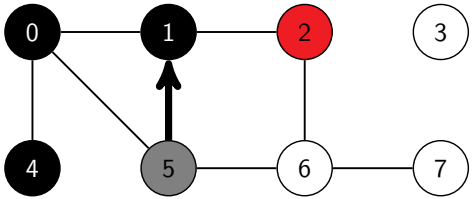
Fila $= \{2\}$



Executando o Algoritmo

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,6}	S
3	{}	N
4	{0}	S
5	{0,1,6}	S
6	{2,5,7}	N
7	{6}	N

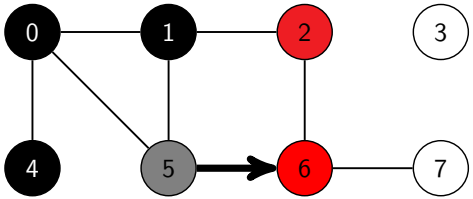
Fila $= \{2\}$



Executando o Algoritmo

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,6}	S
3	{}	N
4	{0}	S
5	{0,1,6}	S
6	{2,5,7}	S
7	{6}	N

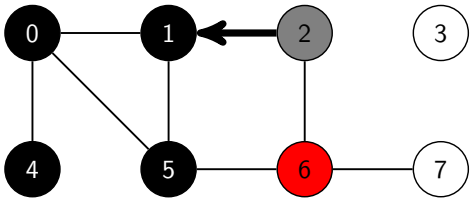
Fila $= \{2, 6\}$



Executando o Algoritmo

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,6}	S
3	{}	N
4	{0}	S
5	{0,1,6}	S
6	{2,5,7}	S
7	{6}	N

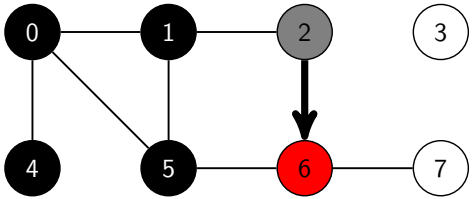
Fila $= \{6\}$



Executando o Algoritmo

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,6}	S
3	{}	N
4	{0}	S
5	{0,1,6}	S
6	{2,5,7}	S
7	{6}	N

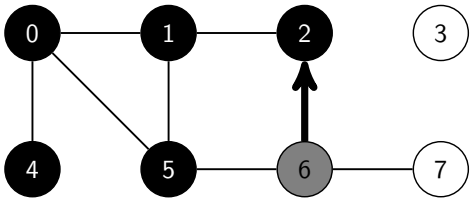
Fila $= \{6\}$



Executando o Algoritmo

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,6}	S
3	{}	N
4	{0}	S
5	{0,1,6}	S
6	{2,5,7}	S
7	{6}	N

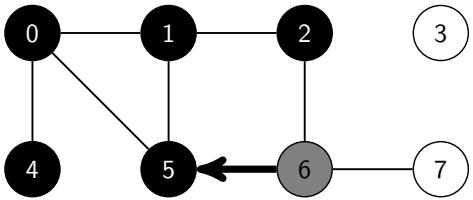
Fila $= \{\}$



Executando o Algoritmo

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,6}	S
3	{}	N
4	{0}	S
5	{0,1,6}	S
6	{2,5,7}	S
7	{6}	N

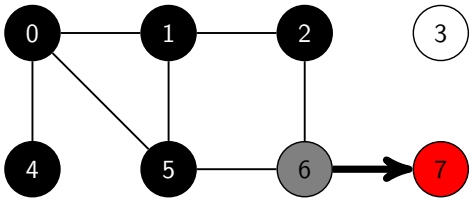
Fila $= \{\}$



Executando o Algoritmo

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,6}	S
3	{}	N
4	{0}	S
5	{0,1,6}	S
6	{2,5,7}	S
7	{6}	S

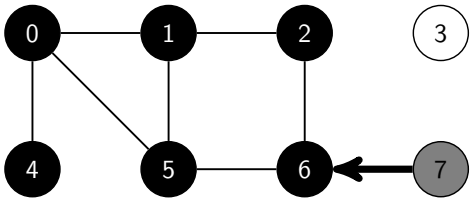
Fila $= \{7\}$



Executando o Algoritmo

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,6}	S
3	{}	N
4	{0}	S
5	{0,1,6}	S
6	{2,5,7}	S
7	{6}	S

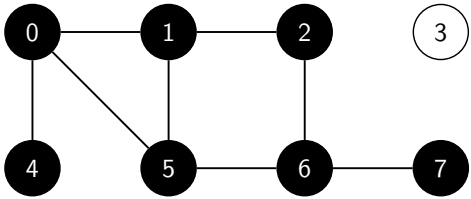
Fila $= \{\}$



Executando o Algoritmo

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,6}	S
3	{}	N
4	{0}	S
5	{0,1,6}	S
6	{2,5,7}	S
7	{6}	S

Fila $= \{\}$



Implementação

```
1  const int MAX = 1e3+10;
2
3  vector<bool> vis(MAX);
4  vector<vector<int>> g(MAX);
5
6  void bfs(int s) {
7      queue<int> q;
8      q.push(s), vis[s] = 1;
9
10     while(!q.empty()){
11         int v = q.front(); q.pop();
12         for(auto u : g[v].size()) if(!vis[u]){
13             q.push(u), vis[u] = 1;
14         }
15     }
16 }
```

Calcular Distância

- Adicionamos um vetor `dist[]` que nos diz a distância de um vértice até `s`

```
1  vector<int> dist(MAX, -1);
```

Calcular Distância

- Adicionamos um vetor `dist[]` que nos diz a distância de um vértice até `s`

```
1  vector<int> dist(MAX, -1);
```

```
1  dist[s] = 0;
2  while(!q.empty()){
3      v = q.front(); q.pop();
4      for(auto u : g[v]) if(!vis[u]){
5          dist[u] = dist[v]+1;
6          q.push(u);
7      }
8  }
```

Recuperar Caminho

- Adicionamos um vetor `pai[]` que nos diz o primeiro vértice nos chamou na BFS

```
1  vector<int> pai(MAX, -1);
```

Recuperar Caminho

- Adicionamos um vetor `pai[]` que nos diz o primeiro vértice nos chamou na BFS

```
1  vector<int> pai(MAX, -1);
```

- E assim como fizemos com a distância, definimos os valores desses vetores dentro da BFS: `pai[u] = v`

Recuperar Caminho

- Adicionamos um vetor `pai[]` que nos diz o primeiro vértice nos chamou na BFS

```
1  vector<int> pai(MAX, -1);
```

- E assim como fizemos com a distância, definimos os valores desses vetores dentro da BFS: `pai[u] = v`
- Para recuperar o menor caminho de `s` até algum vértice `v` fazemos:

Recuperar Caminho

- Adicionamos um vetor `pai[]` que nos diz o primeiro vértice nos chamou na BFS

```
1  vector<int> pai(MAX, -1);
```

- E assim como fizemos com a distância, definimos os valores desses vetores dentro da BFS: `pai[u] = v`
- Para recuperar o menor caminho de `s` até algum vértice `v` fazemos:

```
1  //pai[s] = s;
2  vector<int> path;
3  if(!vis[v]) return path;
4  while(pai[v] != v){
5      v = pai[v];
6      path.push_back(v);
7  }
```

Exercício Resolvido: Message Route

Exercício Resolvido: Message Route

- Global e BFS

```
1  const int MAX = 1e5+10;
2  vector<int> vis(MAX, 0), pai(MAX, -1);
3  vector<vector<int>> g(MAX);
4
5  void bfs(int s){
6      queue<int> q;
7      q.push(s), pai[s] = s, vis[s] = 1;
8      while(!q.empty()){
9          int v = q.front(); q.pop();
10         for(auto u : g[v]) if(!vis[u]){
11             q.push(u), vis[u] = 1, pai[u]=v;
12         }
13     }
14 }
```

Exercício Resolvido: Message Route

- Recuperando caminho

```
1  vector<int> path(int v){
2      vector<int> ret;
3      while(pai[v]!=-1){
4          ret.push_back(v);
5          if(v==pai[v]) break;
6          else v = pai[v];
7      }
8      reverse(ret.begin(), ret.end());
9      return ret;
10 }
```

Exercício Resolvido: Message Route

- Main

```
1  int n, m; cin >> n >> m;
2  for(int i=0;i <m; i++){
3      int a, b; cin >> a >>b; a--, b--;
4      g[a].push_back(b), g[b].push_back(a);
5  }
6
7  bfs(0);
8  vector<int> ans = path(0, n-1);
9
10 if(ans.empty()) cout << "IMPOSSIBLE" << endl;
11 else{
12     cout << ans.size() << endl;
13     for(auto u : ans) cout << u+1 << " ";
14     cout << endl;
15 }
```

Motivação: Mapa

- Harry ganhou um mapa mágico no qual ele pode ver o trajeto realizado por seus amigos. Ele precisa de sua colaboração para determinar onde Hermione está.

Motivação: Mapa

- Harry ganhou um mapa mágico no qual ele pode ver o trajeto realizado por seus amigos. Ele precisa de sua colaboração para determinar onde Hermione está.
- O mapa tem N linhas e M colunas de caracteres: ‘.’, ‘o’ ou ‘H’. A posição inicial de Hermione no mapa é indicada pela letra ‘o’. A letra ‘H’ indica uma posição em que Hermione pode ter passado. Todas as posições pelas quais Hermione passou são representadas pela letra ‘H’ no mapa

Motivação: Mapa

- Harry ganhou um mapa mágico no qual ele pode ver o trajeto realizado por seus amigos. Ele precisa de sua colaboração para determinar onde Hermione está.
- O mapa tem N linhas e M colunas de caracteres: '.', 'o' ou 'H'. A posição inicial de Hermione no mapa é indicada pela letra 'o'. A letra 'H' indica uma posição em que Hermione pode ter passado. Todas as posições pelas quais Hermione passou são representadas pela letra 'H' no mapa

	1	2	3	4	5	6	7
1	.	.	.	**H**	**H**	**H**	.
2	H	H	H	.	.	.	**H**
3	H	.	H	H	H	.	.
4	H	.	.	.	H	H	.
5	H	.	o	.	.	.	.
6	H	H	H	.	.	**H**	**H**

Motivação: Mapa

- Harry ganhou um mapa mágico no qual ele pode ver o trajeto realizado por seus amigos. Ele precisa de sua colaboração para determinar onde Hermione está.
- O mapa tem N linhas e M colunas de caracteres: '.', 'o' ou 'H'. A posição inicial de Hermione no mapa é indicada pela letra 'o'. A letra 'H' indica uma posição em que Hermione pode ter passado. Todas as posições pelas quais Hermione passou são representadas pela letra 'H' no mapa

	1	2	3	4	5	6	7
1	.	.	.	**H**	**H**	**H**	.
2	H	H	H	.	.	.	**H**
3	H	.	H	H	H	.	.
4	H	.	.	.	H	H	.
5	H	.	o	.	.	.	.
6	H	H	H	.	.	**H**	**H**

- No exemplo, a Hermione começa na posição (5, 3) e termina na posição (4, 6).

Caminhando em Grids

- Qual a relação desse problema com BFS?

Caminhando em Grids

- Qual a relação desse problema com BFS?
- A princípio nenhuma, mas veremos que muitos problemas em Grid precisam de BFS para serem resolvidos, por exemplo, quando envolvem labirintos ou achar a menor distância entre dois pontos.

Caminhando em Grids

- Qual a relação desse problema com BFS?
- A princípio nenhuma, mas veremos que muitos problemas em Grid precisam de BFS para serem resolvidos, por exemplo, quando envolvem labirintos ou achar a menor distância entre dois pontos.
- Mas não só isso, conseguimos resolver esse problema usando BFS!

Representação Implícita de Grafos

- No exercício anterior usamos lista de adjacência para representar nosso grafo.

Representação Implícita de Grafos

- No exercício anterior usamos lista de adjacência para representar nosso grafo.
- Porém, podemos representar grafos de outras formas, a exemplo da representação implicita.

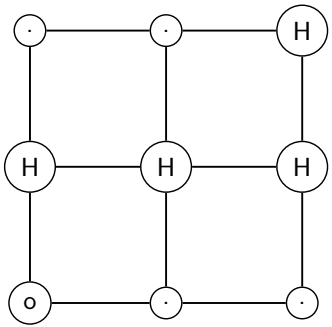
Representação Implícita de Grafos

- No exercício anterior usamos lista de adjacência para representar nosso grafo.
- Porém, podemos representar grafos de outras formas, a exemplo da representação implicita.
- Toda grid pode ser vista como um grafo, em que temos uma relação de vizinhança

Representação Implícita de Grafos

- No exercício anterior usamos lista de adjacência para representar nosso grafo.
- Porém, podemos representar grafos de outras formas, a exemplo da representação implicita.
- Toda grid pode ser vista como um grafo, em que temos uma relação de vizinhança

```
. . H
H H H
o . .
```



Dicas de implementação: Vetor de movimentos

- Podemos declarar um vetor que nos diz quais são os possível passos que podemos tomar a partir de uma certa posição.

Dicas de implementação: Vetor de movimentos

- Podemos declarar um vetor que nos diz quais são os possível passos que podemos tomar a partir de uma certa posição.
- Assim, no lugar de adicionarmos condições no nosso código, podemos iterar por esse vetor e somar suas coordenadas na nossa posição atual.

Dicas de implementação: Vetor de movimentos

- Podemos declarar um vetor que nos diz quais são os possível passos que podemos tomar a partir de uma certa posição.
- Assim, no lugar de adicionarmos condições no nosso código, podemos iterar por esse vetor e somar suas coordenadas na nossa posição atual.

```
1  # Movimentos: cima, baixo, esquerda, direita
2  vector<pair<int, int>> mov = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
```

Dicas de implementação: Vetor de movimentos

- Podemos declarar um vetor que nos diz quais são os possível passos que podemos tomar a partir de uma certa posição.
- Assim, no lugar de adicionarmos condições no nosso código, podemos iterar por esse vetor e somar suas coordenadas na nossa posição atual.

```
1  # Movimentos: cima, baixo, esquerda, direita
2  vector<pair<int, int>> mov = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
```

- E o usamos da seguinte forma:

```
1  #Posição atual: (i, j)
2  for(auto u : mov) {
3    int new_i = i + u.f, new_j = j + u.s;
4    # ...
5  }
```

Dicas de implementação: Função de validação

- Para evitar encher o código de verificações, podemos fazer uma única função que verifica se a posição para a qual queremos mover é válida

Dicas de implementação: Função de validação

- Para evitar encher o código de verificações, podemos fazer uma única função que verifica se a posição para a qual queremos mover é válida

```
1  bool val(int i, int j){
2      return i>=0 and j>=0 and i<n and j<m;
3  }
```

Dicas de implementação: Função de validação

- Para evitar encher o código de verificações, podemos fazer uma única função que verifica se a posição para a qual queremos mover é válida

```
1  bool val(int i, int j){
2      return i>=0 and j>=0 and i<n and j<m;
3  }
```

- `n` e `m` são os tamanhos da nossa matriz, perceba que eles são declarados globalmente para podermos usar.

Dicas de implementação: Função de validação

- Para evitar encher o código de verificações, podemos fazer uma única função que verifica se a posição para a qual queremos mover é válida

```
1  bool val(int i, int j){
2      return i>=0 and j>=0 and i<n and j<m;
3  }
```

- `n` e `m` são os tamanhos da nossa matriz, perceba que eles são declarados globalmente para podermos usar.
- Outras condições podem ser adicionadas a essa função, por exemplo se a posição é diferente de certo caracter, ou se ela ainda não foi visitada.

Exercício Resolvido: Mapa

Exercício Resolvido: Mapa

- BFS (código completo)

```
1  pair<int, int> bfs(int i, int j){
2      queue<pair<int, int>> q;
3      q.push({i, j}), vis[i][j] = 1;
4      pair<int, int> ret;
5      while(!q.empty()){
6          ret = q.front(); q.pop();
7          for(int k=0; k<4; k++){
8              int new_i = ret.f+mov[k].f, new_j = ret.s+mov[k].s;
9              if(val(new_i, new_j))
10                 q.push({new_i, new_j}), vis[new_i][new_j]=1;
11         }
12     }
13     return ret;
14 }
```

Motivação: Fire

- Você está preso em um castelo e em algum pontos começou um incêndio. Você precisa fugir do fogo e correr para a saída.

Motivação: Fire

- Você está preso em um castelo e em algum pontos começou um incêndio. Você precisa fugir do fogo e correr para a saída.
- A cada segundo, o fogo se espalha pelas direções norte, sul, leste e oeste. Felizmente, as paredes não queimam e mantém o fogo dentro do castelo, por isso se você escapar estará tudo bem.

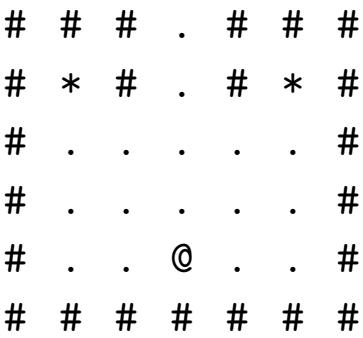
Motivação: Fire

- Você está preso em um castelo e em algum pontos começou um incêndio. Você precisa fugir do fogo e correr para a saída.
- A cada segundo, o fogo se espalha pelas direções norte, sul, leste e oeste. Felizmente, as paredes não queimam e mantém o fogo dentro do castelo, por isso se você escapar estará tudo bem.
- Seus movimentos são, assim como o fogo, norte, sul, leste e oeste e você demora 1 segundo para se mover. Você não pode atravessar paredes, ou correr pelo fogo.

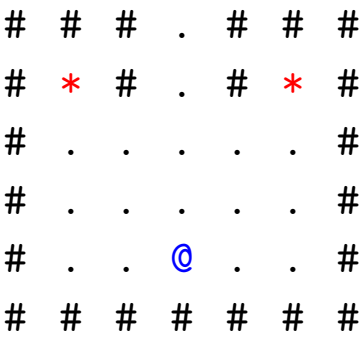
Motivação: Fire

- Você está preso em um castelo e em algum pontos começou um incêndio. Você precisa fugir do fogo e correr para a saída.
- A cada segundo, o fogo se espalha pelas direções norte, sul, leste e oeste. Felizmente, as paredes não queimam e mantém o fogo dentro do castelo, por isso se você escapar estará tudo bem.
- Seus movimentos são, assim como o fogo, norte, sul, leste e oeste e você demora 1 segundo para se mover. Você não pode atravessar paredes, ou correr pelo fogo.
- Dado um mapa do castelo, descrubra o quão rápido você consegue escapar da construção.

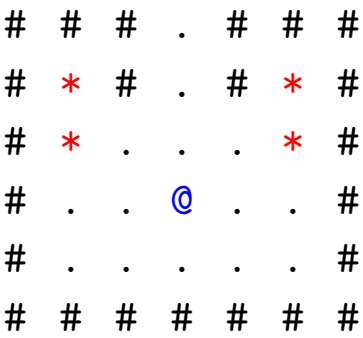
Escapando do Fogo



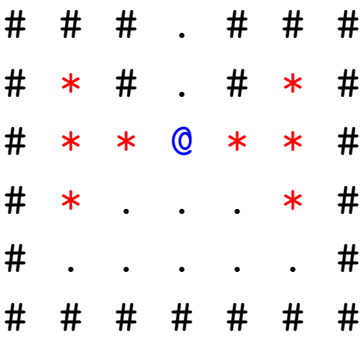
Escapando do Fogo



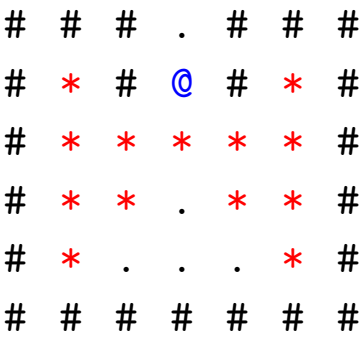
Escapando do Fogo



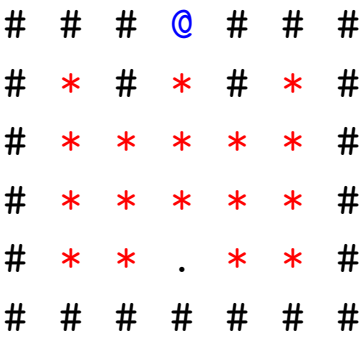
Escapando do Fogo



Escapando do Fogo



Escapando do Fogo



BFS Multisource

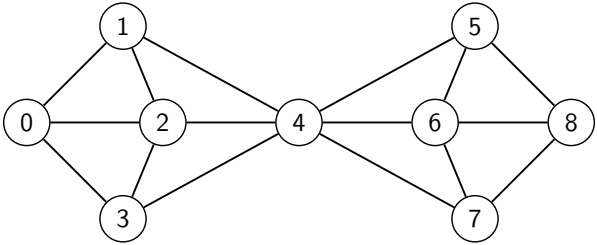
- Uma vantagem da BFS é que não precisamos nos restringir à distância dos vértices até um único vértice. Podemos calcular a distância dos vértices do nosso grafo a todo um conjunto de vértices.

BFS Multisource

- Uma vantagem da BFS é que não precisamos nos restringir à distância dos vértices até um único vértice. Podemos calcular a distância dos vértices do nosso grafo a todo um conjunto de vértices.
- Para isso, basta adicionar todos os vértices do conjunto na nossa fila da BFS.

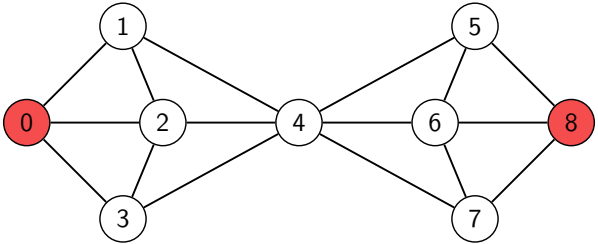
BFS Multisource: Exemplo

- Qual o vértice mais distante do conjuto de vértices $\{0, 8\}$?



BFS Multisource: Exemplo

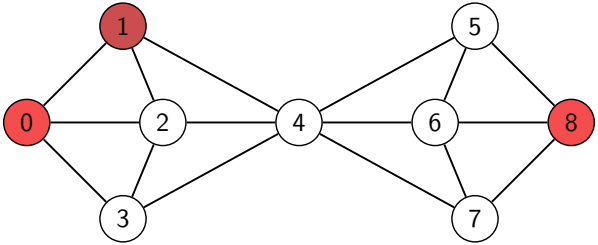
- Qual o vértice mais distante do conjuto de vértices $\{0, 8\}$?



- Os vértices 0 e 8 estão a uma distância 0 das nossas fontes.

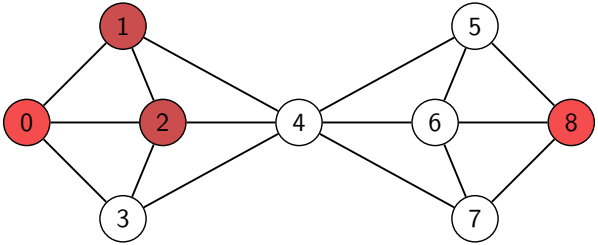
BFS Multisource: Exemplo

- Exemplo: Qual o vértice mais distante do conjuto de vértices $\{0, 8\}$?



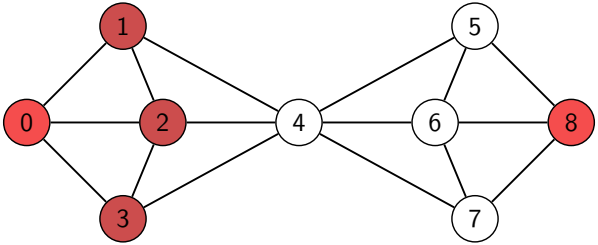
BFS Multisource: Exemplo

- Exemplo: Qual o vértice mais distante do conjuto de vértices $\{0, 8\}$?



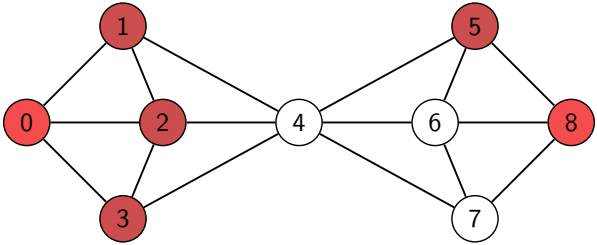
BFS Multisource: Exemplo

- Exemplo: Qual o vértice mais distante do conjuto de vértices $\{0, 8\}$?



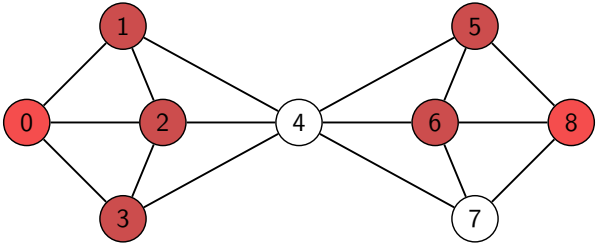
BFS Multisource: Exemplo

- Exemplo: Qual o vértice mais distante do conjuto de vértices $\{0, 8\}$?



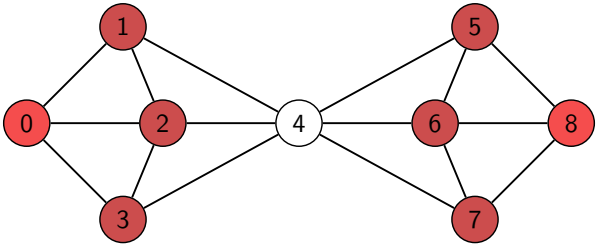
BFS Multisource: Exemplo

- Exemplo: Qual o vértice mais distante do conjuto de vértices $\{0, 8\}$?



BFS Multisource: Exemplo

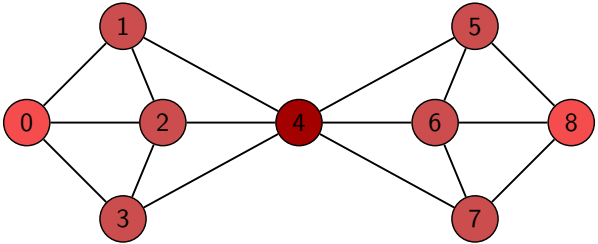
- Exemplo: Qual o vértice mais distante do conjuto de vértices $\{0, 8\}$?



- Os vértices 1, 2, 3, 5, 6 e 7 estão a uma distância 1 das nossas fontes

BFS Multisource: Exemplo

- Exemplo: Qual o vértice mais distante do conjuto de vértices $\{0, 8\}$?



- O vértice 4 está a uma distância 2 das nossas fontes e é o vértice mais distante.

Exercício Resolvido: Fire

Exercício Resolvido: Fire

- BFS Multisource (código completo)

```
1  void bfs_ms(vector<pair<int, int>> mult_s){
2      queue<pair<int, int>> q;
3      for(auto s: mult_s) q.push(s), vis[s.f][s.s]=1, dist[s.f][s.s]=0;
4
5      while(!q.empty()){
6          pair<int, int> v = q.front(); q.pop();
7          for(int k=0; k<4; k++){
8              pair<int, int> u(v.f+mov[k].f, v.s+mov[k].s);
9              if(val(u))
10                 q.push(u), vis[u.f][u.s]=1, dist[u.f][u.s]=dist[v.f][v.s]+1;
11         }
12     }
13 }
```

Material

- Lista de Exercícios (“BFS”)
- Tutorial de BFS CP-algorithms
- Aula Erik Demaine