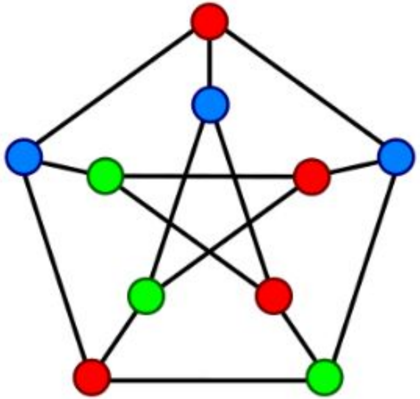
Emparelhamento em grafos bipartidos

Laboratório de Programação Competitiva - 2020

Pedro Henrique Paiola

Coloração de vértices

- Um **problema de coloração** em grafos consiste em atribuir cores a certos elementos do grafo sujeito a determinadas condições.
- Uma **coloração dos vértices** de um grafo é uma atribuição de cores aos vértices tal que cada vértice recebe uma e só uma cor.
- Uma coloração de um grafo é **válida** se duas pontas de cada aresta têm cores diferentes. Ou seja, se não existem vértices adjacentes da mesma cor.

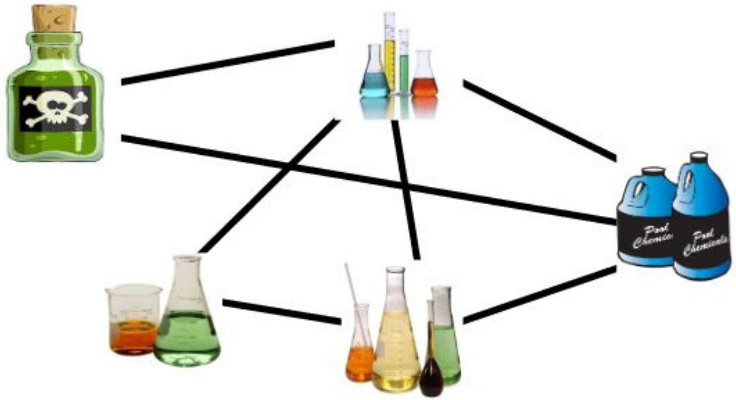


Coloração de vértices

- Dizemos que um grafo é **k-colorível** se tem uma coloração válida com até k cores.
- Problema da coloração mínima de vértices: dado um grafo não-dirigido G, encontrar uma coloração válida de G com o menor número de cores possível.
  - Problema NP-difícil
  - Na verdade, até hoje não se conhece nenhum bom algoritmo para checar se um grafo é k-colorível com $k \geq 3$

Aplicações

- Os vértices representam produtos químicos necessários em algum processo de produção.
- Produtos que podem explodir se combinados são ligados por uma aresta.
- O número cromático representa o número mínimo de compartimentos para guardar estes produtos químicos em segurança.



Aplicações

- O sudoku é uma variação do problema da coloração de vértices.
- Cada célula representa um vértice, e existe uma aresta entre dois vértices se eles estão em uma mesma linha, mesma coluna ou no mesmo bloco.

	6		1		4		5	
		8	3		5	6		
								1
8			4		7			6
		6				3		
7			9		1			4
5								2
		7	2		6	9		
	4		5		8		7	

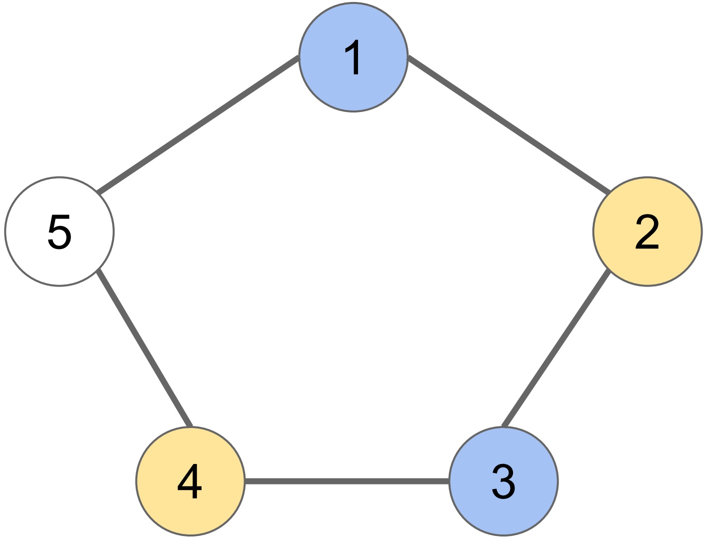
9	6	3	1	7	4	2	5	8
1	7	8	3	2	5	6	4	9
2	5	4	6	8	9	7	3	1
8	2	1	4	3	7	5	9	6
4	9	6	8	5	2	3	1	7
7	3	5	9	6	1	8	2	4
5	8	9	7	1	3	4	6	2
3	1	7	2	4	6	9	8	5
6	4	2	5	9	8	1	7	3

Bicoloração

- Verificar se um grafo é **bicolorível** é um processo mais simples.
- Basicamente, vamos realizar uma busca em largura, alternando as cores por camada. Se neste processo encontrarmos um vértice já colorido e tentarmos colorir ele com outra cor, então o grafo não é bicolorível.

Bicoloração

- Teorema: Dado um grafo G, existe uma bicoloração para G se, e somente se, não existe um ciclo de comprimento ímpar

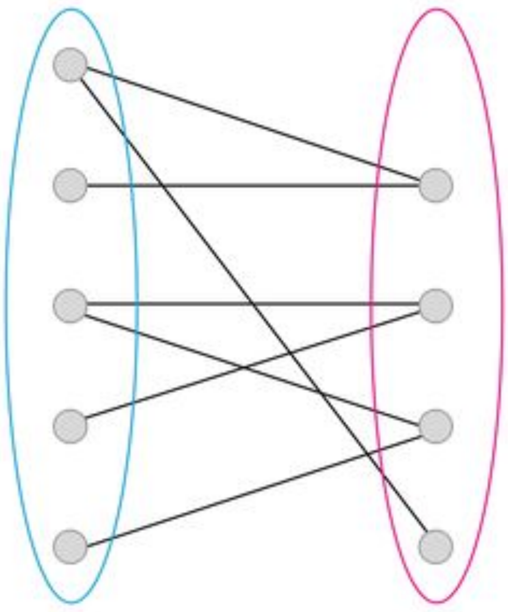


Bicoloração

```
bicoloração(G, s)
    fila.push(s);
    enquanto tiver elementos na fila faça
        v = fila.front(); fila.pop();
        para cada w vizinho de v faça
            se d[w] == -1 então
                d[w] = d[v] + 1
                c[w] = d[w] % 2
                fila.push(w)
            senão se (c[w] == c[v])
                Não é bicolorível
```

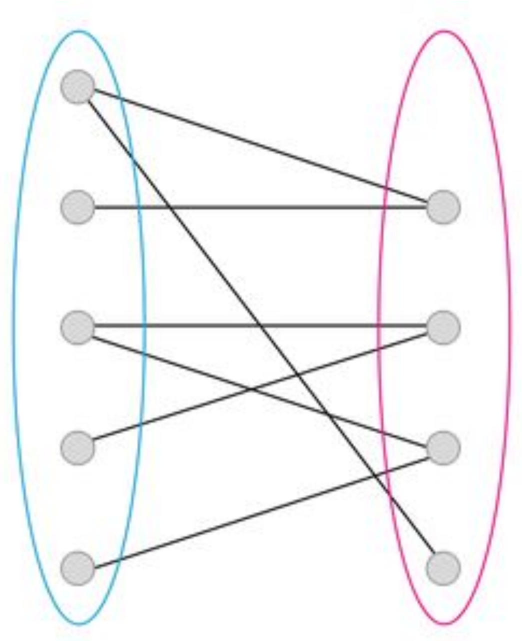
Grafos bipartidos

- Um grafo bipartido é um grafo não direcionado G(V,E) em que é possível particionar os vértices em dois conjuntos L e R tal que toda aresta (l,r) possui l ∈ L e r ∈ R.

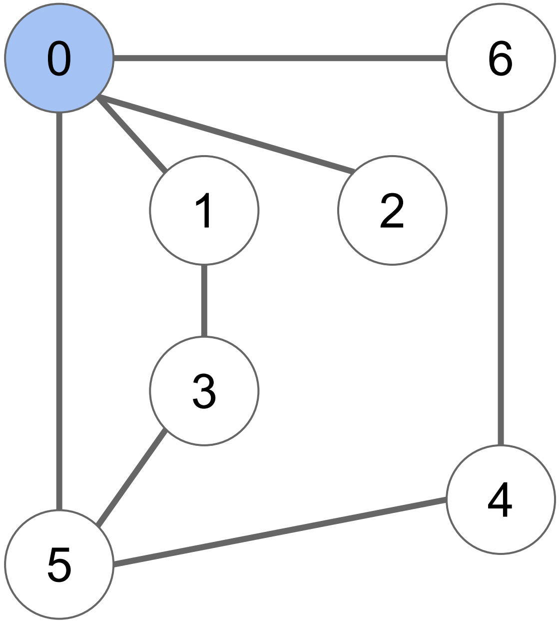


Grafos bipartidos

- Um grafo é bipartido se, e somente se, seu número cromático é menor ou igual a 2.



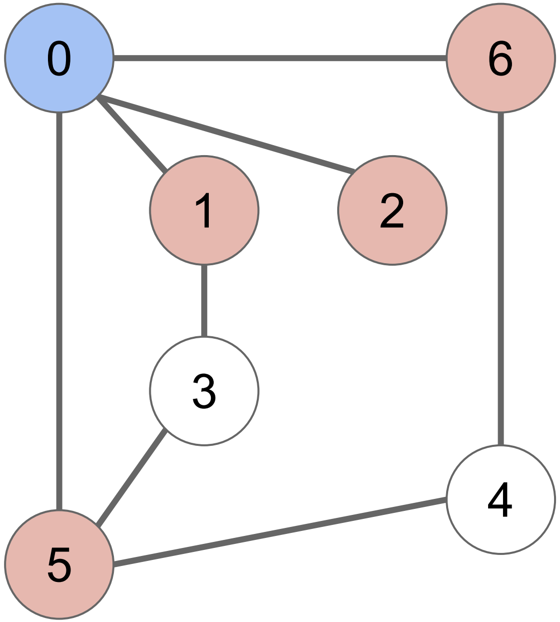
Grafos bipartidos



Fila: 0

Vértice	Distância
0	0
1	
2	
3	
4	
5	
6	

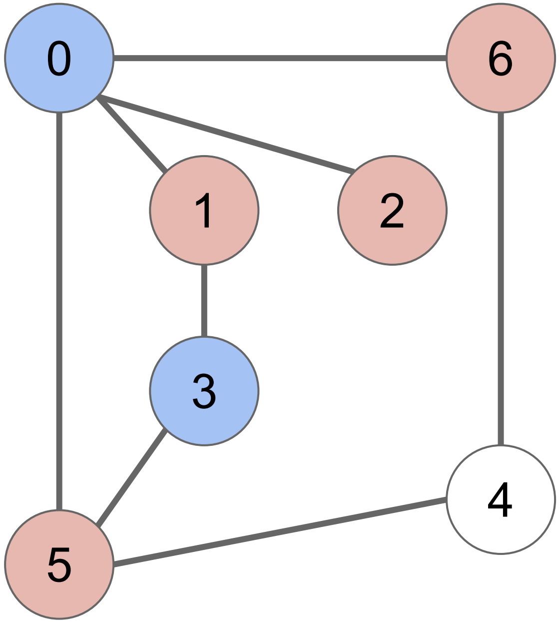
Grafos bipartidos



Fila: **1 2 5 6**

Vértice	Distância
0	0
1	1
2	1
3	
4	
5	1
6	1

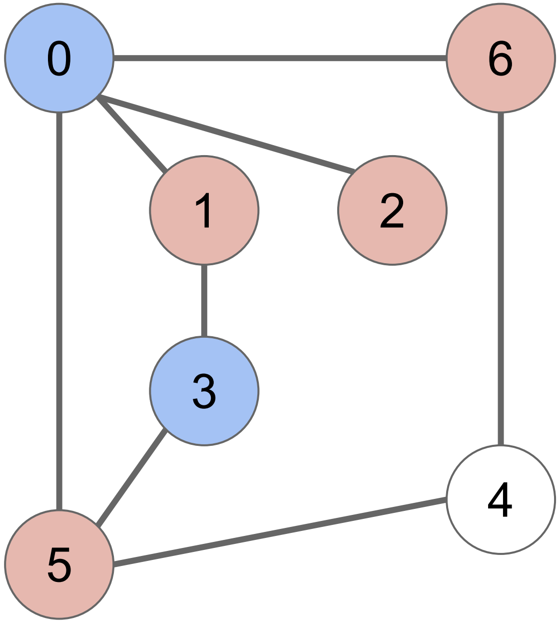
Grafos bipartidos



Fila: 2 5 6 **3**

Vértice	Distância
0	0
1	1
2	1
3	2
4	
5	1
6	1

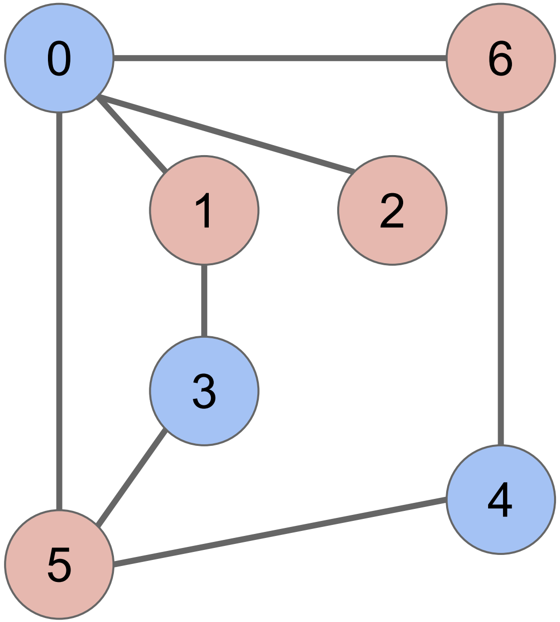
Grafos bipartidos



Fila: 5 6 3

Vértice	Distância
0	0
1	1
2	1
3	2
4	
5	1
6	1

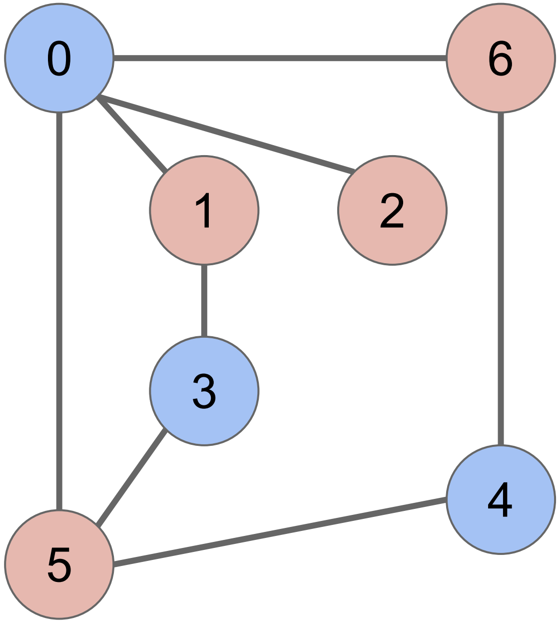
Grafos bipartidos



Vértice	Distância
0	0
1	1
2	1
3	2
4	3
5	1
6	1

Fila: 6 3 **4**

Grafos bipartidos



Fila: 3 4

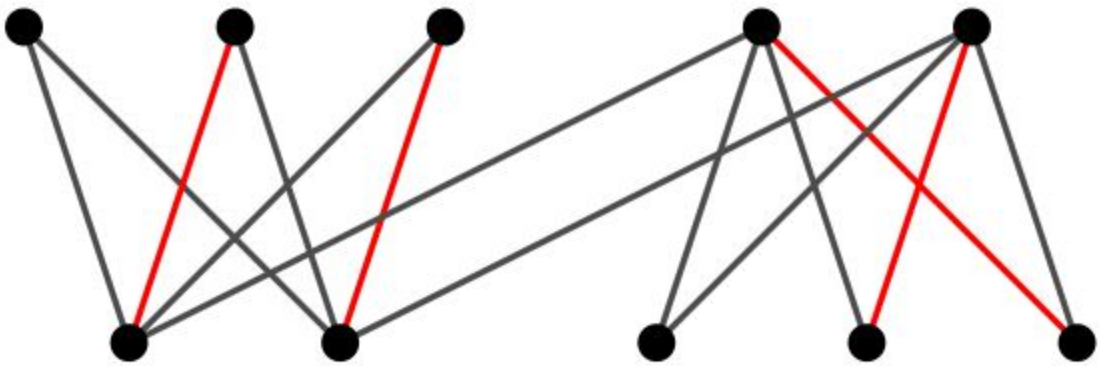
Vértice	Distância
0	0
1	1
2	1
3	2
4	3
5	1
6	1

Emparelhamento

- Um emparelhamento (*matching*) em um grafo G não direcionado é um conjunto M de arestas dotado da seguinte propriedade: todo vértice de G incide em no máximo um elemento de M
- Ou seja, no emparelhamento, um vértice possui no máximo grau 1
- Um emparelhamento M é **máximo** se não existe um emparelhamento M' tal que | M' | > | M |

Emparelhamento

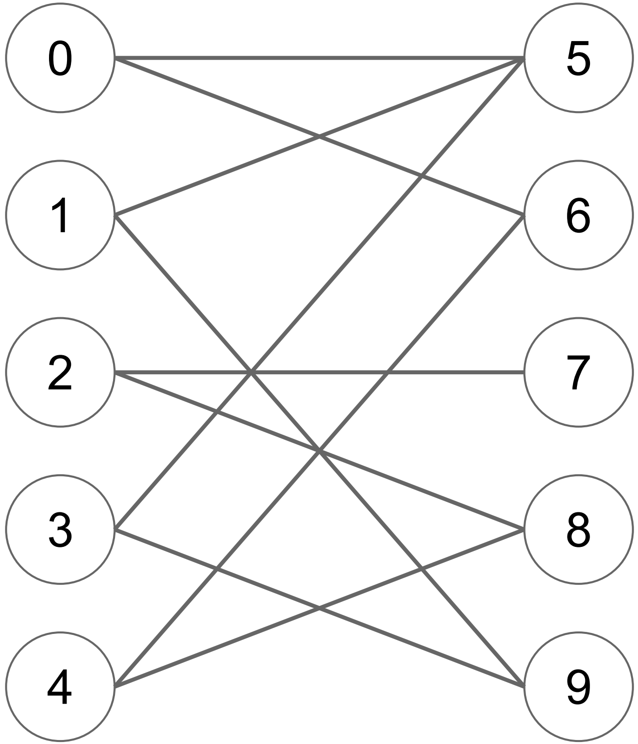
- Existem algoritmos que encontram o emparelhamento máximo em grafos arbitrários, porém são bastante complicados.
- Sendo assim, vamos nos limitar ao emparelhamento em grafos bipartidos.



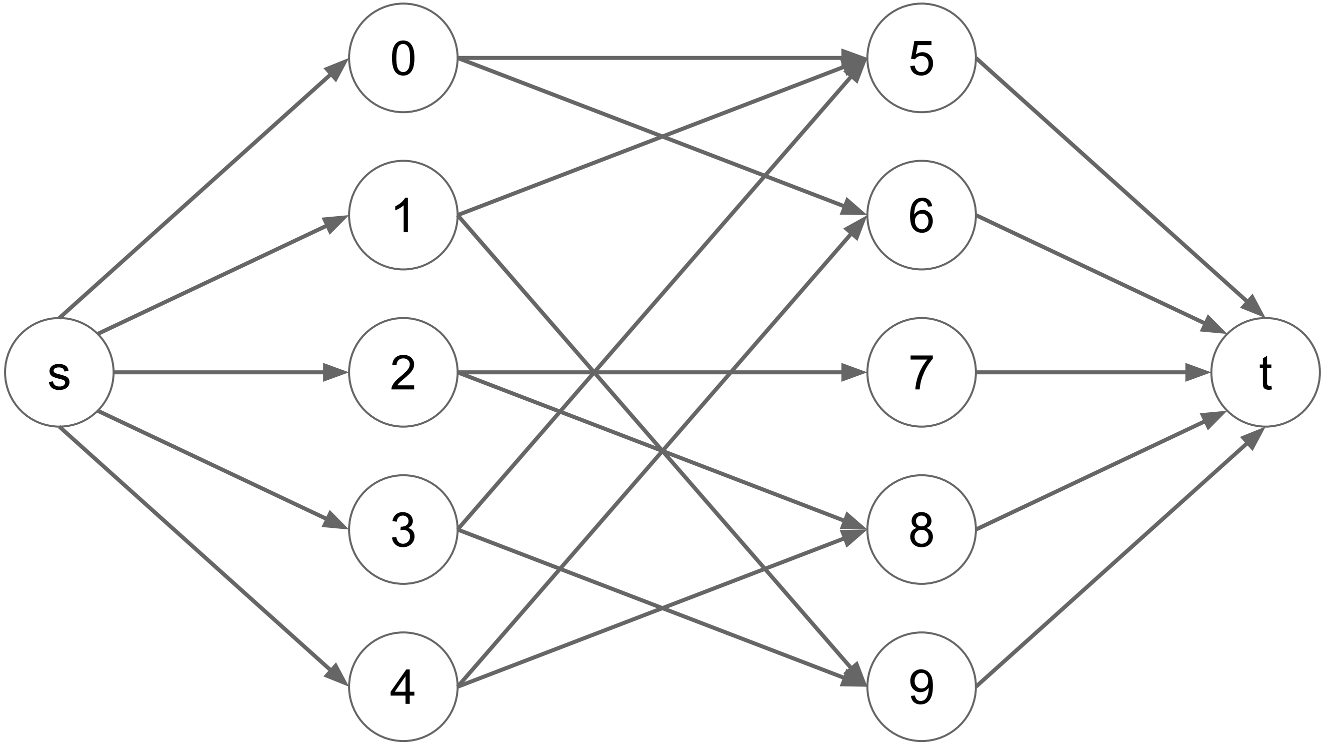
Modelando como problema de fluxo

- Uma forma simples de resolver o problema do emparelhamento em grafos bipartidos é reduzindo-o para um problema de fluxo máximo.
- Para resolver o emparelhamento de um grafo não orientado G(V,E) vamos construir um grafo orientado G'(V',E') em que
  - $V' = V \cup \{s,t\}$
  - $E' = \{(l,r) \mid \forall \{l,r\} \in E\} \cup \{(s,l) \mid \forall l \in L\} \cup \{(r,t) \mid \forall r \in R\}$
- E todas as capacidades serão iguais a 1

Modelando como problema de fluxo



Modelando como problema de fluxo

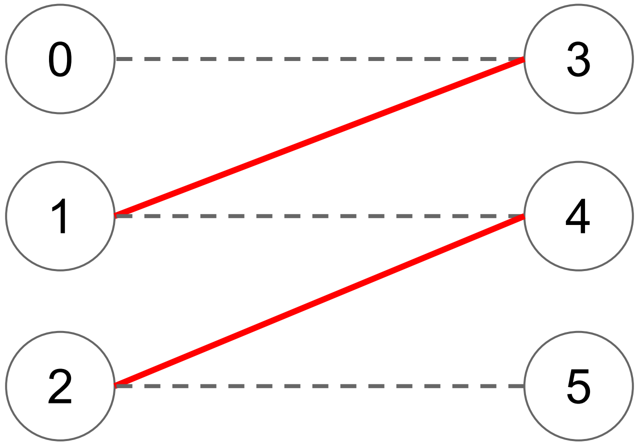


Caminhos alternantes

- Para obtermos um algoritmo mais eficiente, precisamos de algumas definições.
- Suponha dado um grafo G e um emparelhamento M
    - Arestas em M são chamadas de **emparelhadas,** as outras são ditas **livres**
    - Se uma aresta v-w pertence a M, dizemos que v e w estão **emparelhados**
    - Se nenhuma aresta de M incide em v, então v é **livre**
- Um **caminho alternante** é um caminho simples em que suas arestas são, alternadamente, livres e emparelhadas.
- Um caminho é **aumentante** se ele é alternante, começa e termina em um vértice livre e tem comprimento maior que 0.

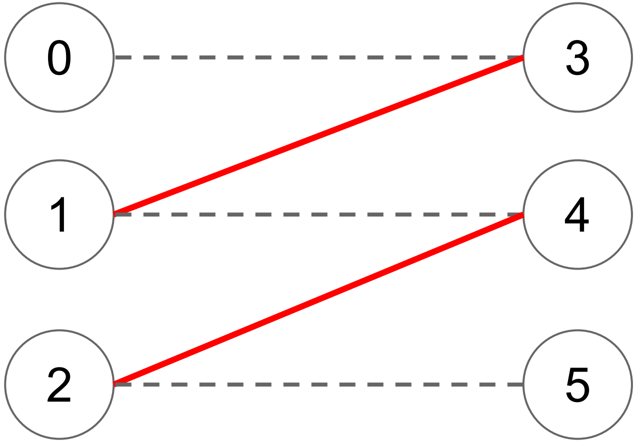
Caminhos alternantes

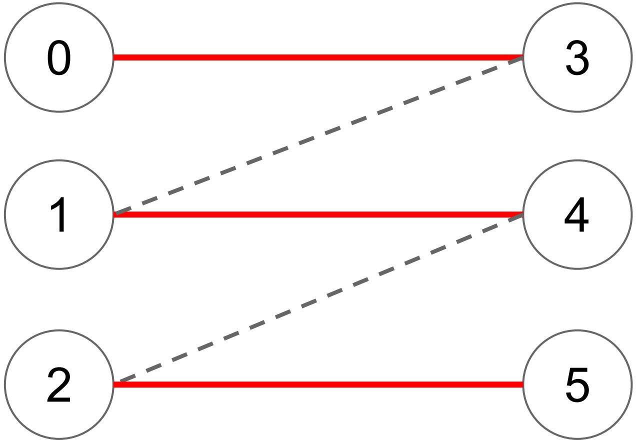
- Exemplo de caminho aumentante



Caminhos alternantes

- E por que este caminho é considerando “aumentante”? Pois se invertermos o estado das arestas, encontramos um emparelhamento válido e maior que o anterior.

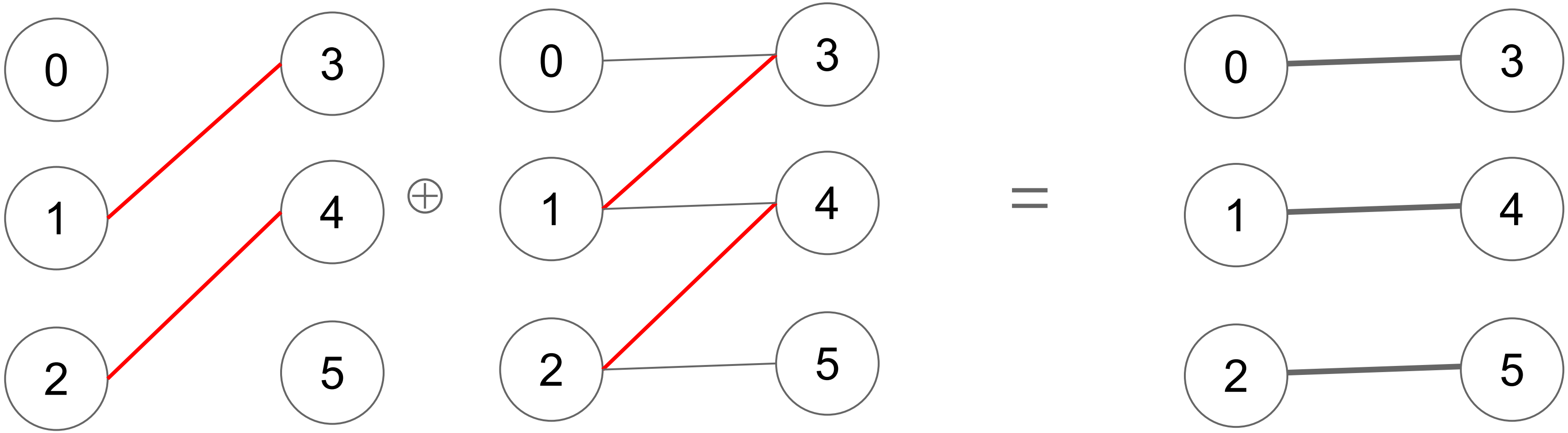




Caminhos alternantes

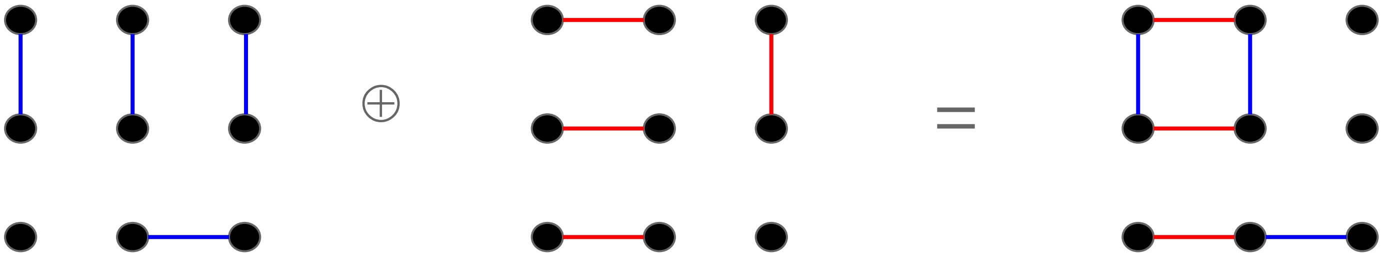
- Formalmente este processo de “inversão” é definido por uma diferença simétrica entre M e o caminho aumentante P

$$M \oplus P = (M - P) \cup (P - M)$$



Teorema de Berge

- Propriedade 1: Sejam dois emparelhamentos M_1 e M_2, ao fazer $M_1 \oplus M_2$ temos como resultado um grafo G' composto por componentes conexas de um dos seguintes tipos:
  a. Vértices isolados
  b. Caminhos alternantes
  c. Ciclos alternantes pares



Teorema de Berge

- Teorema de Berge: Um emparelhamento M em G é máximo se, e somente se, não houver nenhum caminho aumentante em G e M.

Demonstração:

$\Longrightarrow$) $A \Rightarrow B \Leftrightarrow \neg B \Rightarrow \neg A$

Se tiver caminho aumentante, então M não é máximo

Já observamos isso pela própria definição de caminho aumentante. Quando realizamos a diferença simétrica entre M e o caminho aumentante obtemos um emparelhamento M' em que $| M' | > | M |$

Teorema de Berge

- Teorema de Berge: Um emparelhamento M em G é máximo se, e somente se, não houver nenhum caminho aumentante em G e M.

Demonstração:

$\Leftarrow$) B $\Rightarrow$ A (por contradição, B $\wedge$ $\neg$A)

Suponha um emparelhamento M tal que existe outro emparelhamento M' tal que | M' | > | M |. Suponha, para fins de contradição, que não existe caminho aumentante em G e M.

Ao fazer M$\oplus$M', teremos componentes conexas conforme as descritas na Propriedade 1.

Teorema de Berge

- Teorema de Berge: Um emparelhamento M em G é máximo se, e somente se, não houver nenhum caminho aumentante em G e M.

Demonstração:

$\Longleftarrow$) $B \Rightarrow A$ (por contradição, $B \land \neg A$)

Como $| M' | > | M |$, ao menos alguma componente conexa deve ter mais arestas de M’ do que M. Isso só é possível na componente conexa do tipo b (caminho alternante)

Mas isso implica no surgimento de um caminho aumentante em M, o que é uma contradição ■

Algoritmo de Caminhos Aumentantes

```
emparelhamento_maximo(G)
    M = {}
    enquanto existir caminho aumentante P em G e M
        M = M⊕P
    retorne M
```

Algoritmo de Hopcraft-Karp

- De forma semelhante aos algoritmos de fluxo, um caminho aumentante pode ser encontrado com uma DFS ou com uma BFS.
- O algoritmo de Hopcraft-Karp, por sua vez, é considerado um caso especial do algoritmo de Dinic.

```
HopcraftKarp(G)
    M = {}
    enquanto existir caminho aumentante P em G e M
        P = {P_1,...,P_k} //conjunto maximal de c.a. disjuntos
                          //de menor comprimento
        M = M⊕P_1⊕...⊕P_k
    retorne M
```

Algoritmo de Hopcraft-Karp

- De forma semelhante aos algoritmos de fluxo, um caminho aumentante pode ser encontrado com uma DFS ou com uma BFS.
- O algoritmo de Hopcraft-Karp, por sua vez, é considerado um caso especial do algoritmo de Dinic.

```
HopcraftKarp(G)
    M = {}
    enquanto existir caminho aumentante P em G e M
        P = {P_1,...,P_k} //conjunto maximal de c.a. disjuntos
                          //de menor comprimento
        M = M⊕P_1⊕...⊕P_k
    retorne M
```

Algoritmo de Hopcraft-Karp

```
int m, n; //qtde de vértices de L e R
int adj[M+N+1];
int dist[M+1]; //somente para os vértices de L
int matchL[M+1], matchR[N+1];
queue<int> Q;
```

Algoritmo de Hopcraft-Karp

```
bool bfs(){
    for(l = 1; l <= m; l++){
        if (matchL[l] == 0){
            dist[l] = 0;
            Q.push(l);
        }
        else
            dist[l] = INF;
    }
    dist[0] = INF; //"Sorvedouro"
```

Algoritmo de Hopcraft-Karp

```
    while(!Q.empty()){
        int l = Q.front(); Q.pop();
        if (dist[l] < dist[0]){
            for(int r: adj[l]){
                if (dist[matchR[r]] == INF){
                    dist[matchR[r]] = dist[l] + 1;
                    Q.push(matchR[r]);
                }
            }
        }
    }
    return dist[0] < INF;
}
```

Algoritmo de Hopcraft-Karp

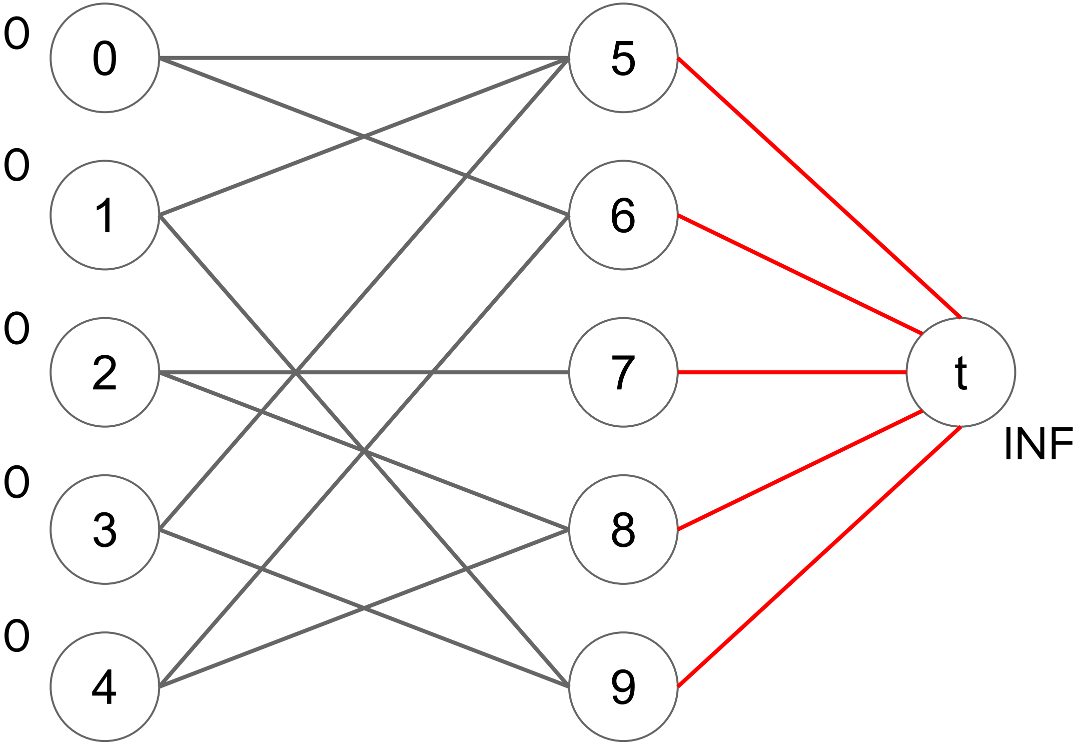
```
bool dfs(int l){
    if (l == 0) return true;
    for(int r: adj[l]){
        if (dist[matchR[r]] == dist[l] + 1){
            if (dfs(matchR[r])){
                matchR[r] = l;
                matchL[l] = r;
                return true;
            }
        }
    }
    return true;
}
```

Algoritmo de Hopcraft-Karp

```
int hopcraft_karp(){
    for(int l = 1; l <= m; l++)
        matchL[l] = 0;
    for(int r = 1; r <= n; r++)
        matchR[r] = 0;
    int matching = 0;
    while(bfs()){
        for(int l = 1; l <= m; l++)
            if (matchL[l] == 0 && dfs(l))
                matching++;
    }
    return matching;
}
```

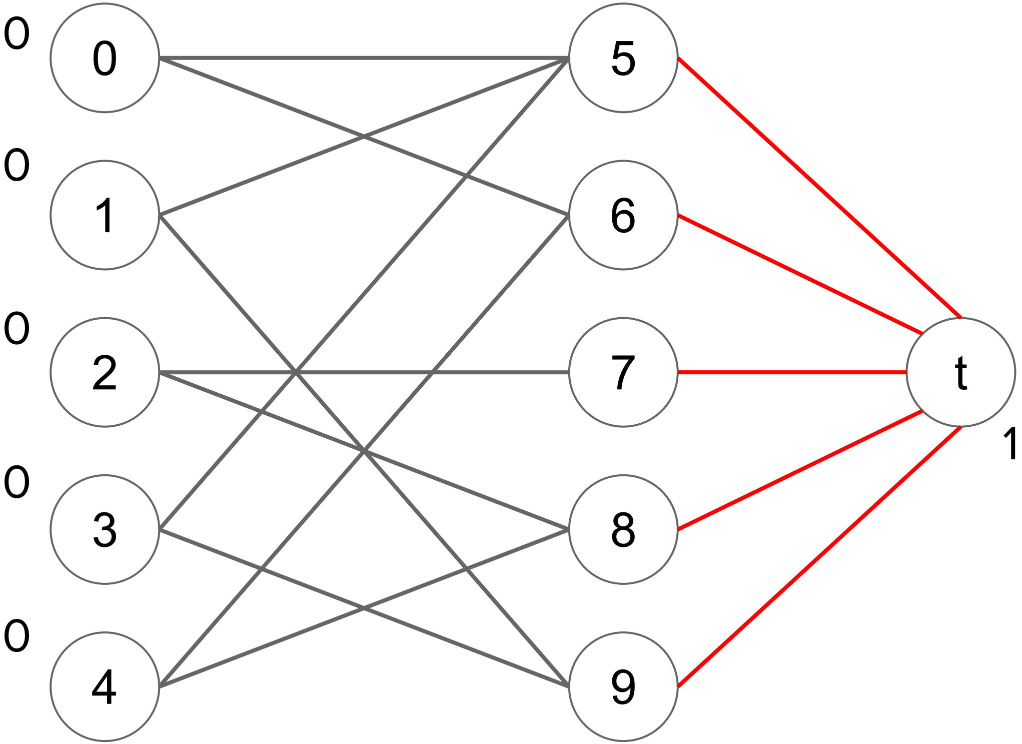
Exemplo

Iteração 1
Fase BFS



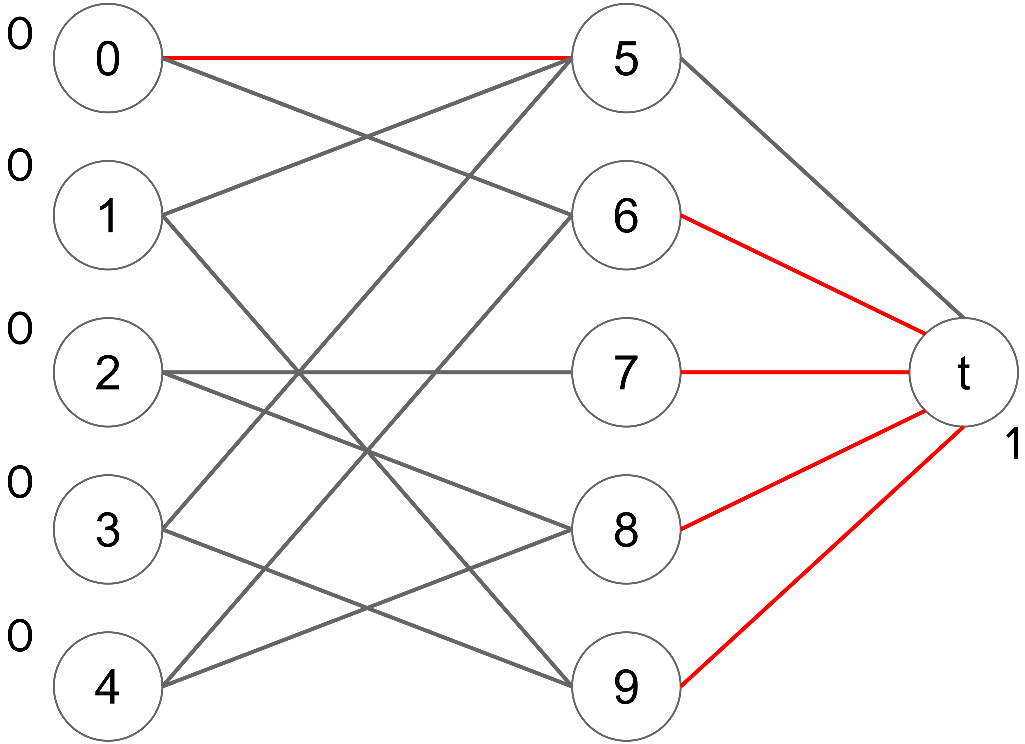
Exemplo

Iteração 1
Fase BFS



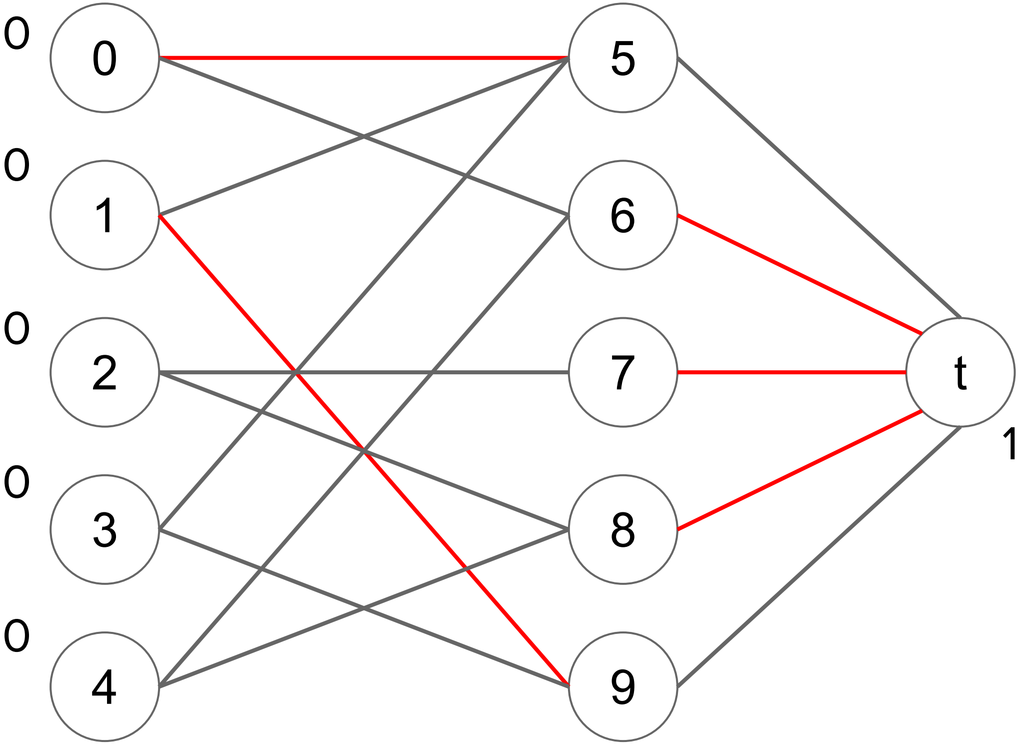
Exemplo

Iteração 1
Fase DFS
vértice 0



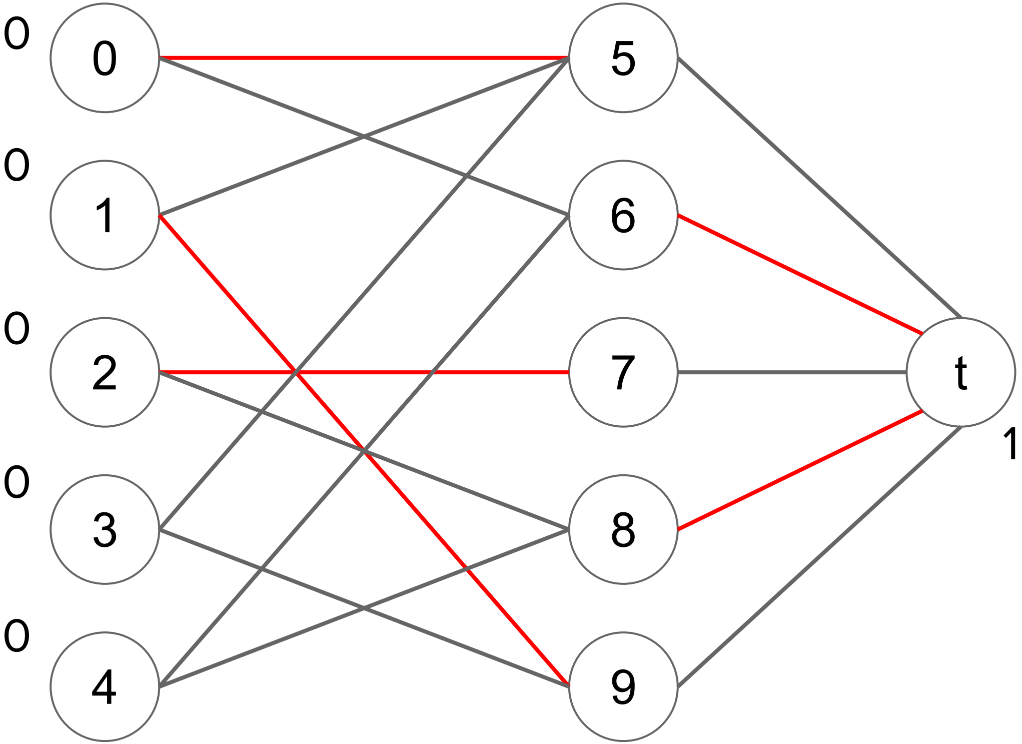
Exemplo

Iteração 1
Fase DFS
vértice 1



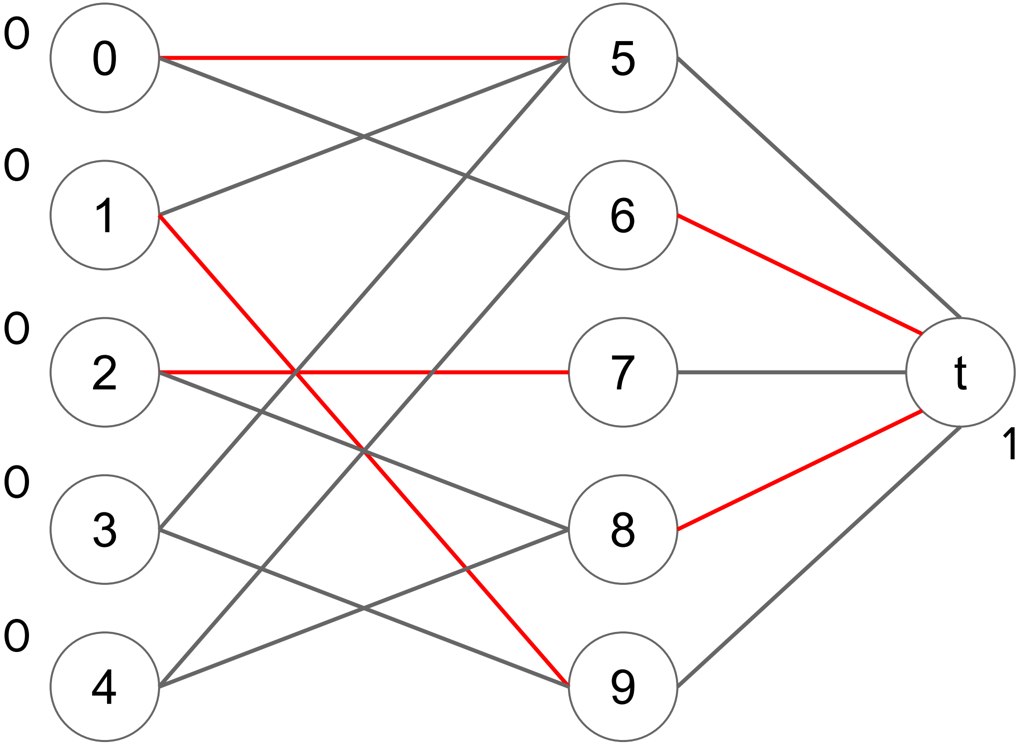
Exemplo

Iteração 1
Fase DFS
vértice 2



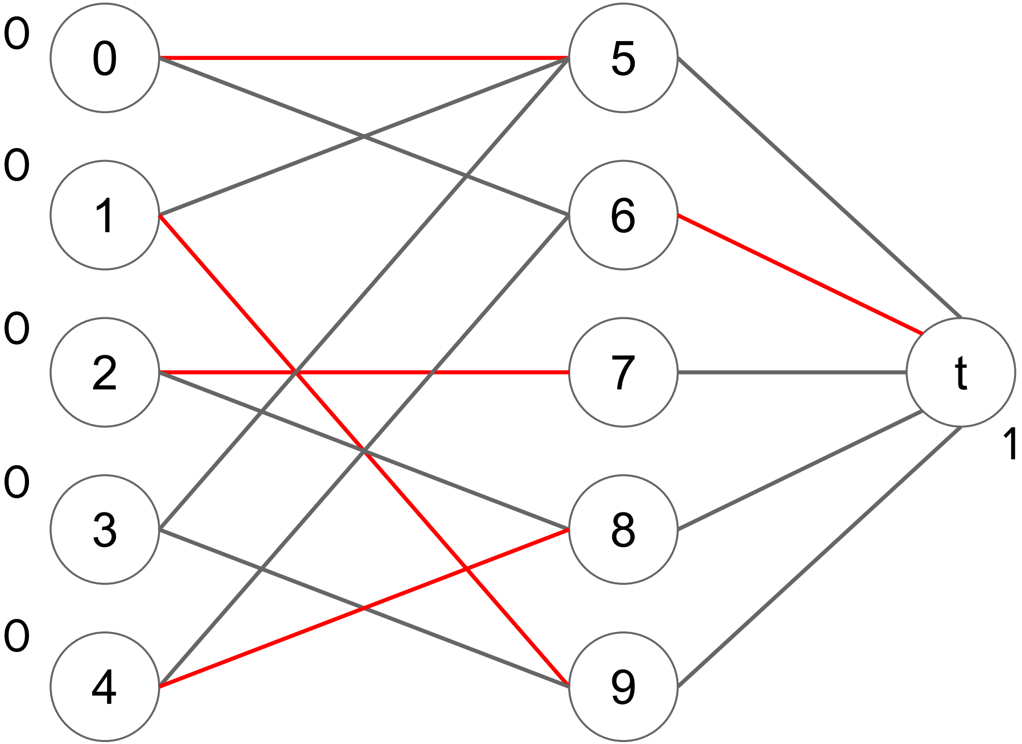
Exemplo

Iteração 1
Fase DFS
vértice 3



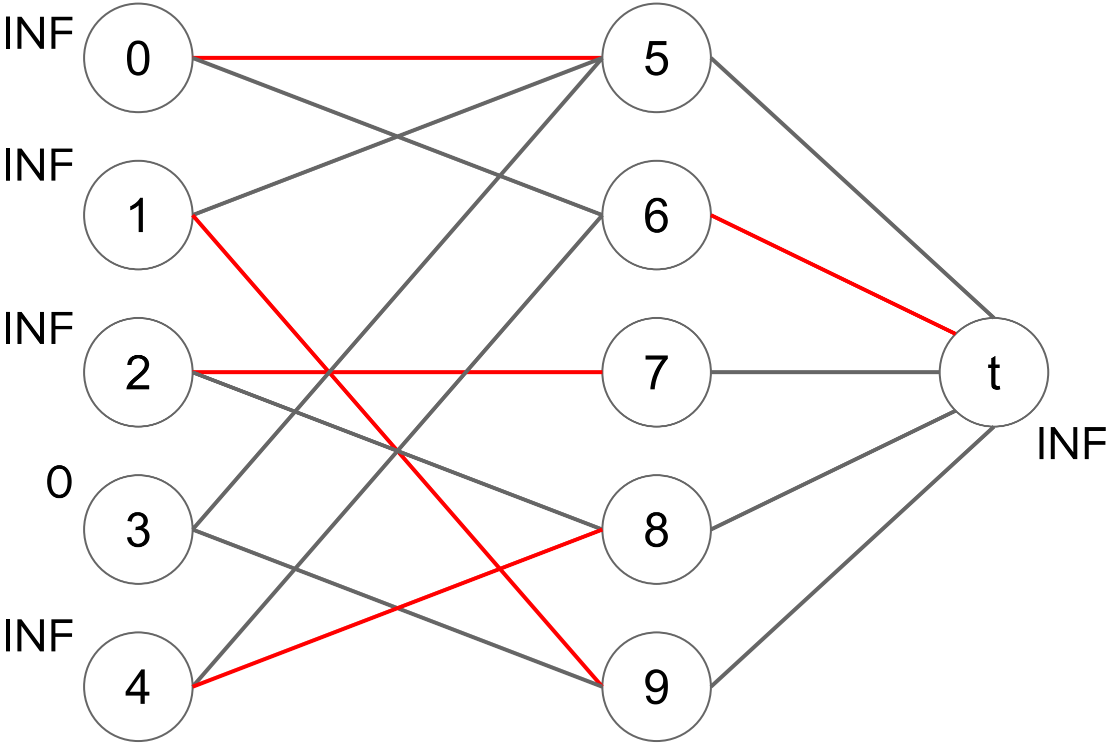
Exemplo

Iteração 1
Fase DFS
vértice 4



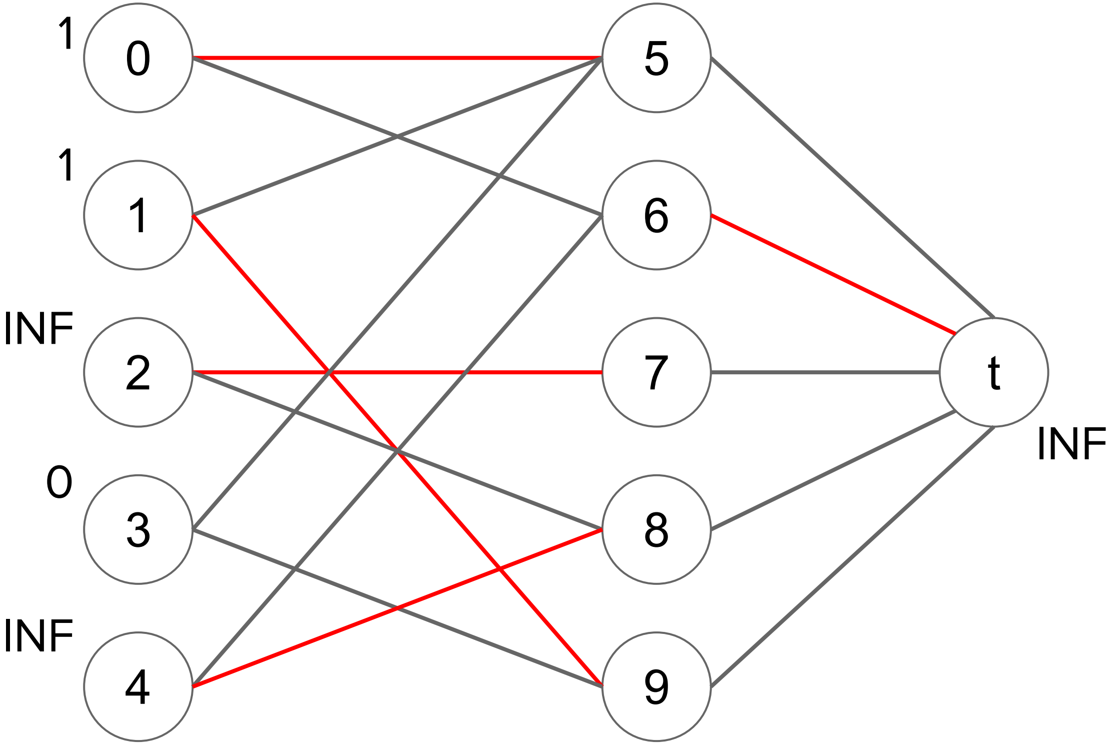
Exemplo

Iteração 2
Fase BFS
Q = {3}



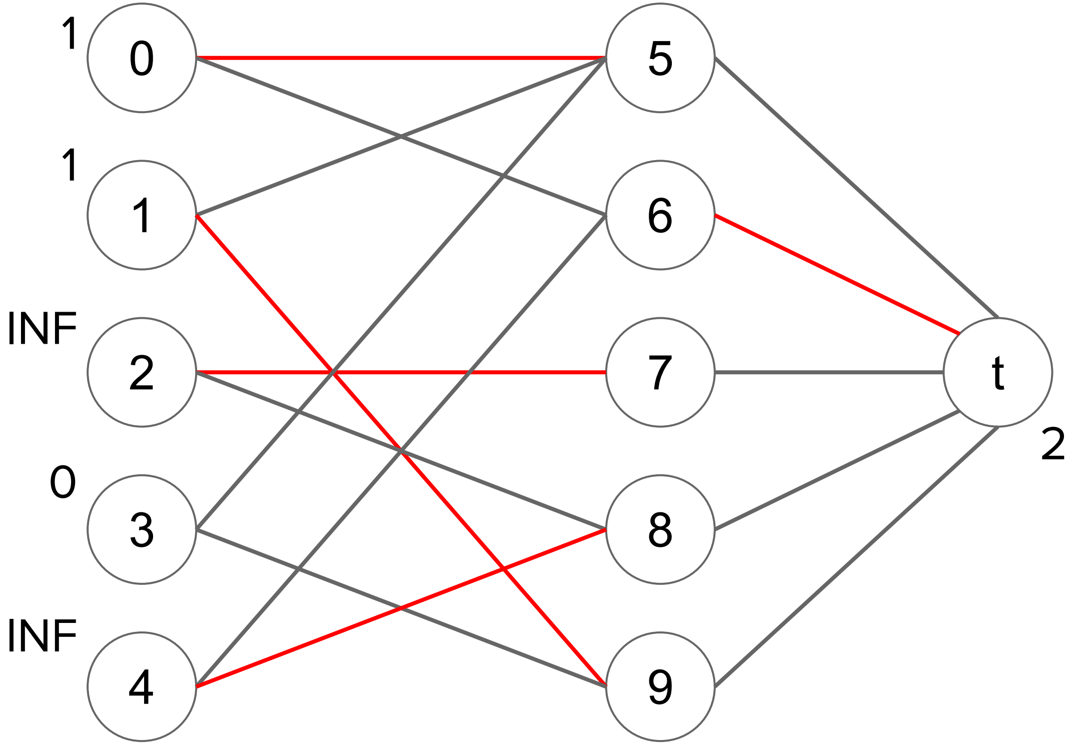
Exemplo

Iteração 2
Fase BFS
Q = {0, 1}



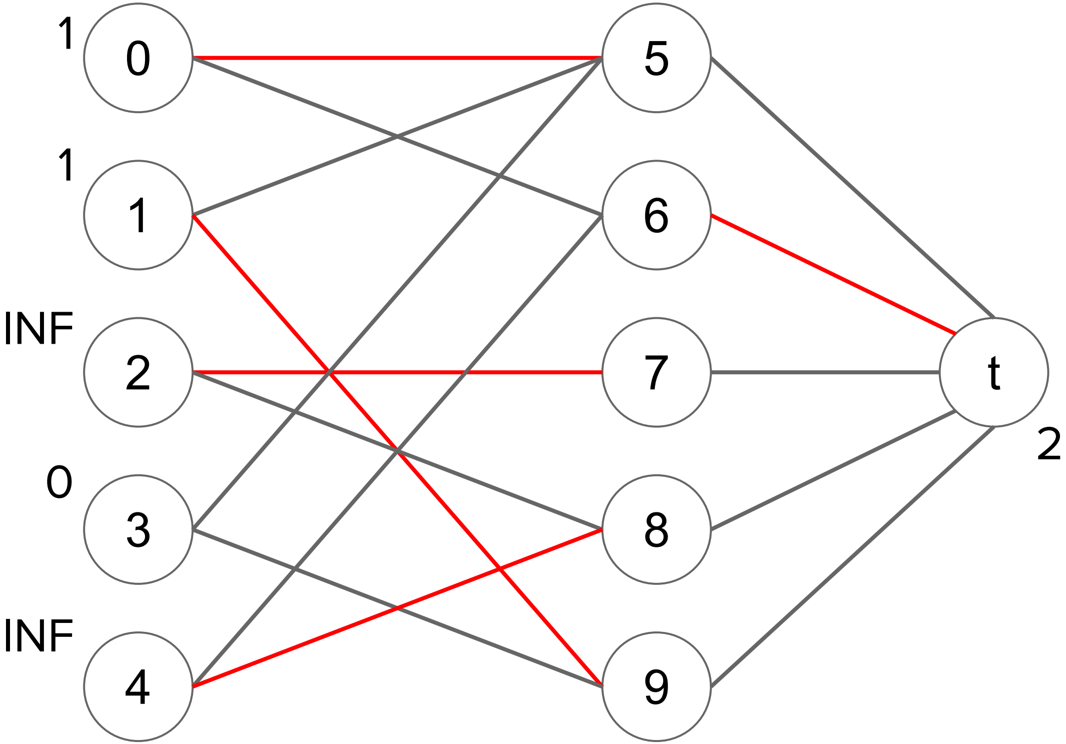
Exemplo

Iteração 2
Fase BFS
Q = {1}



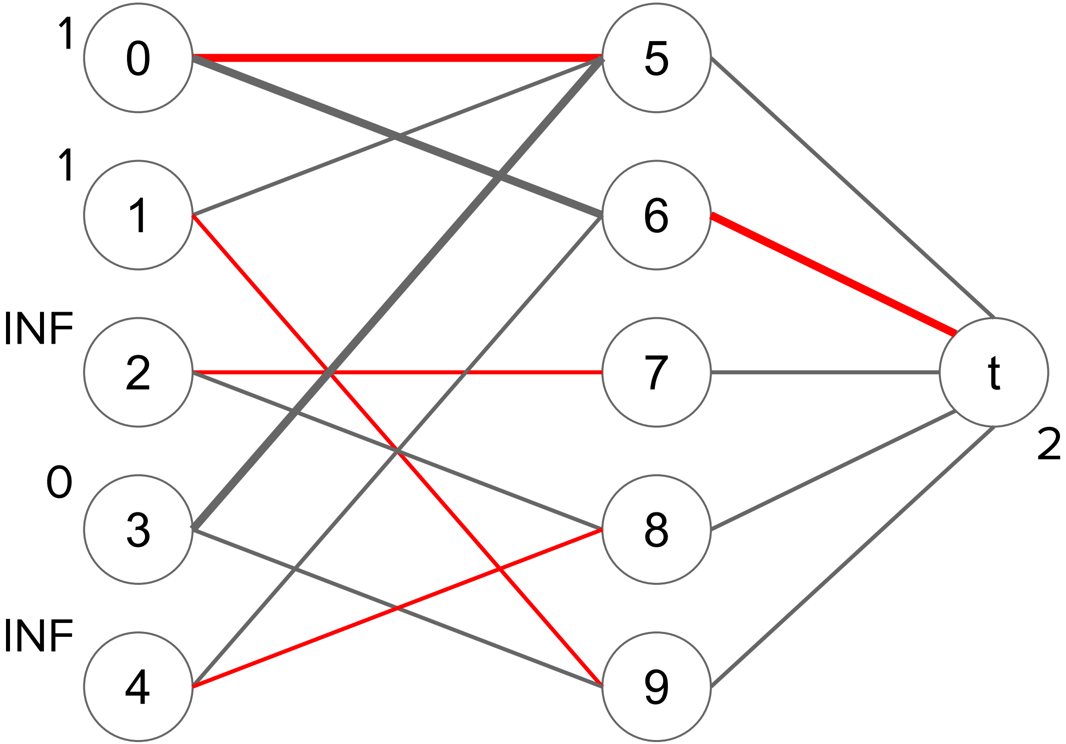
Exemplo

Iteração 2
Fase BFS
Q = {}



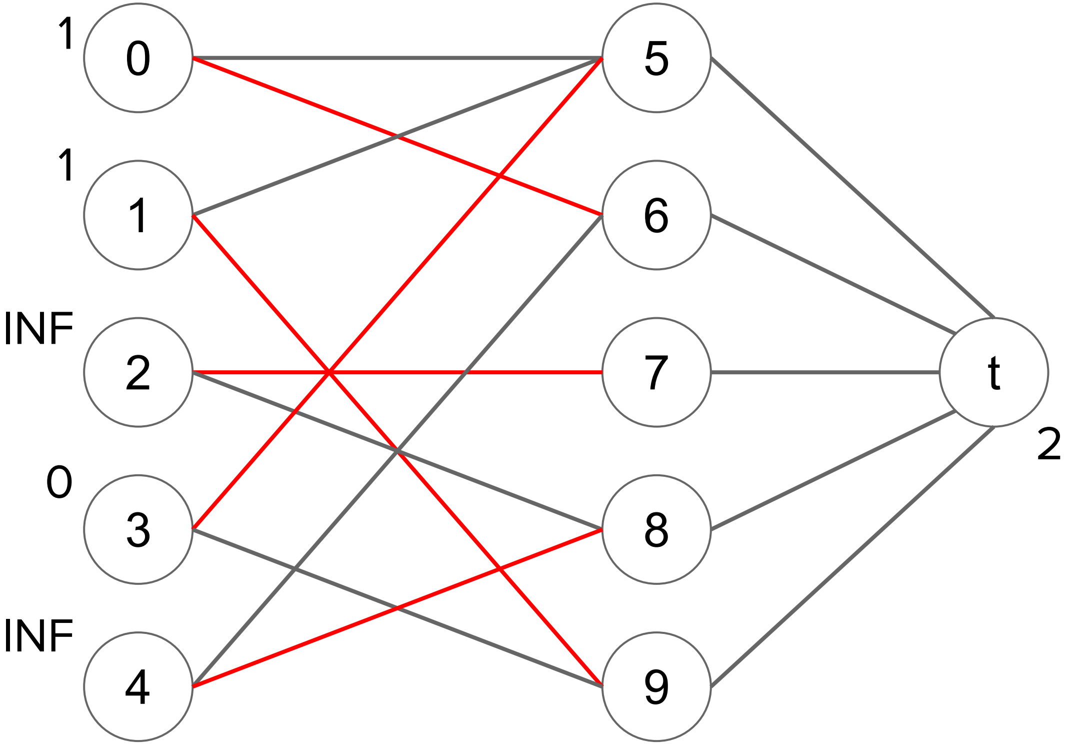
Exemplo

Iteração 2
Fase DFS
vértice 3



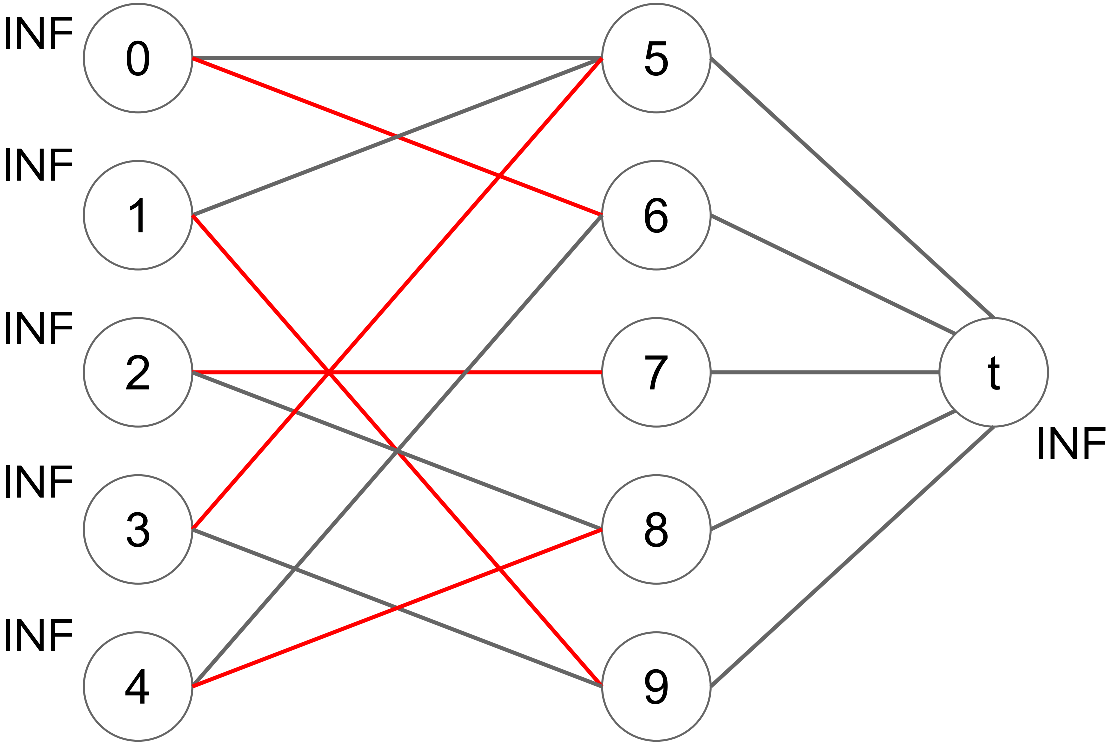
Exemplo

Iteração 2
Fase DFS
vértice 3

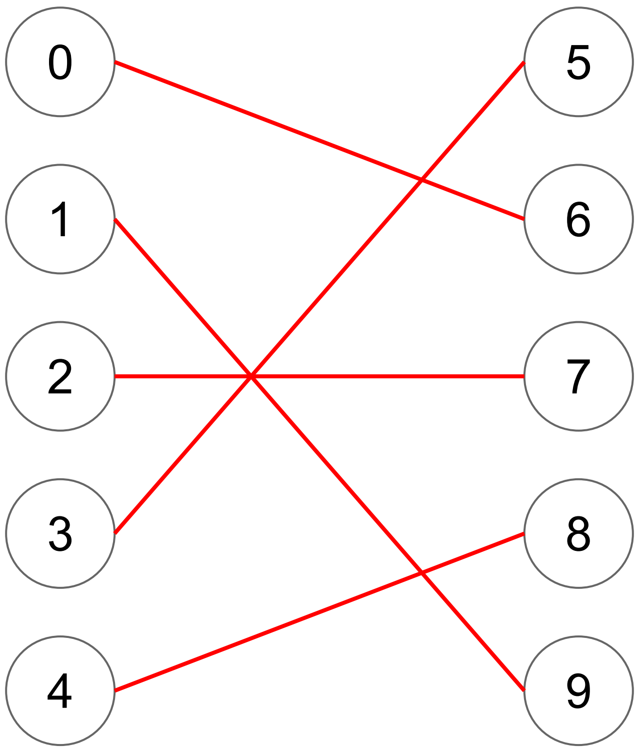


Exemplo

Iteração 3
Fase BFS
fim



Exemplo



Attacking rooks (UVALive - 6525)

- **Objetivo**: posicionar o maior número de torres possíveis em um tabuleiro N x N.
- Neste tabuleiro, algumas posições estão ocupadas por peões.

X				
X				
		X		
	X			
				X

Attacking rooks (UVALive - 6525)

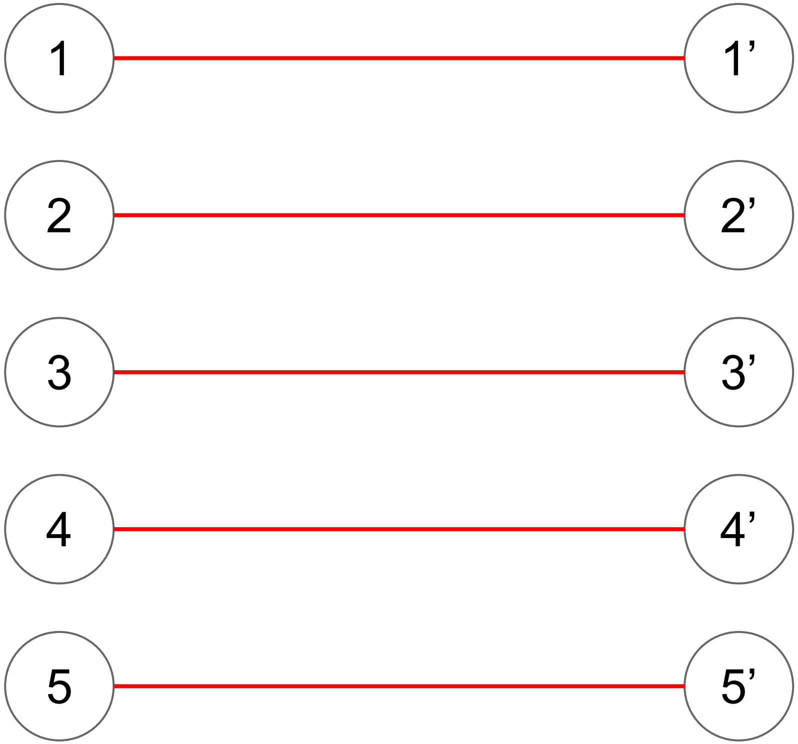
- Vamos pensar, inicialmente, no caso em que não temos peões.
- Nesta situação, podemos resolver de forma gulosa, enchendo a diagonal com torres.

T				
	T			
		T		
			T	
				T

Attacking rooks (UVALive - 6525)

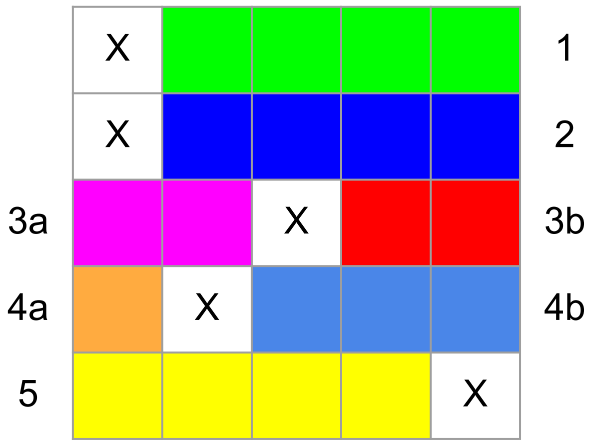
- MAS também poderíamos modelar como um problema de emparelhamento, onde criamos vértices para cada linha e cada coluna.
- Uma torre na posição (l,c) é representada pela aresta que conecta os vértice l e c.
- O emparelhamento garante que não teremos torres em conflito.
    - Como cada vértice do emparelhamento pode ter no máximo uma aresta incidente, então cada linha ou coluna pode ter no máximo uma torre.

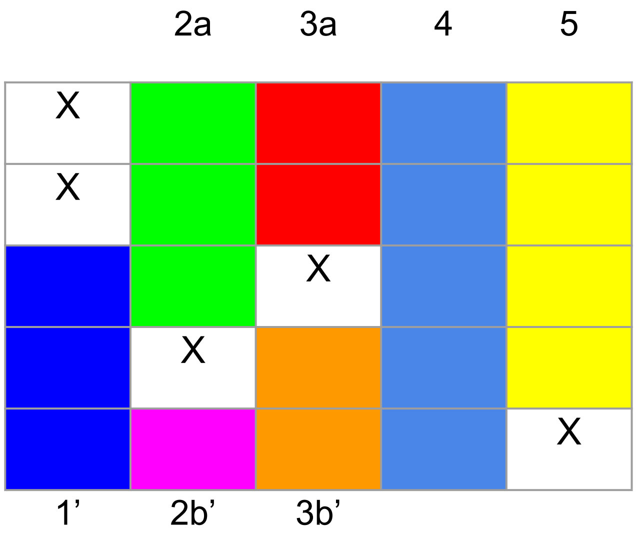
Attacking rooks (UVALive - 6525)



Attacking rooks (UVALive - 6525)

- E quando inserimos os peões? É como se dividíssemos as linhas e colunas em várias partes. Cada parte será representada por um vértice
- Criaremos arestas entre cada par de vértices que representa uma possível posição da torre





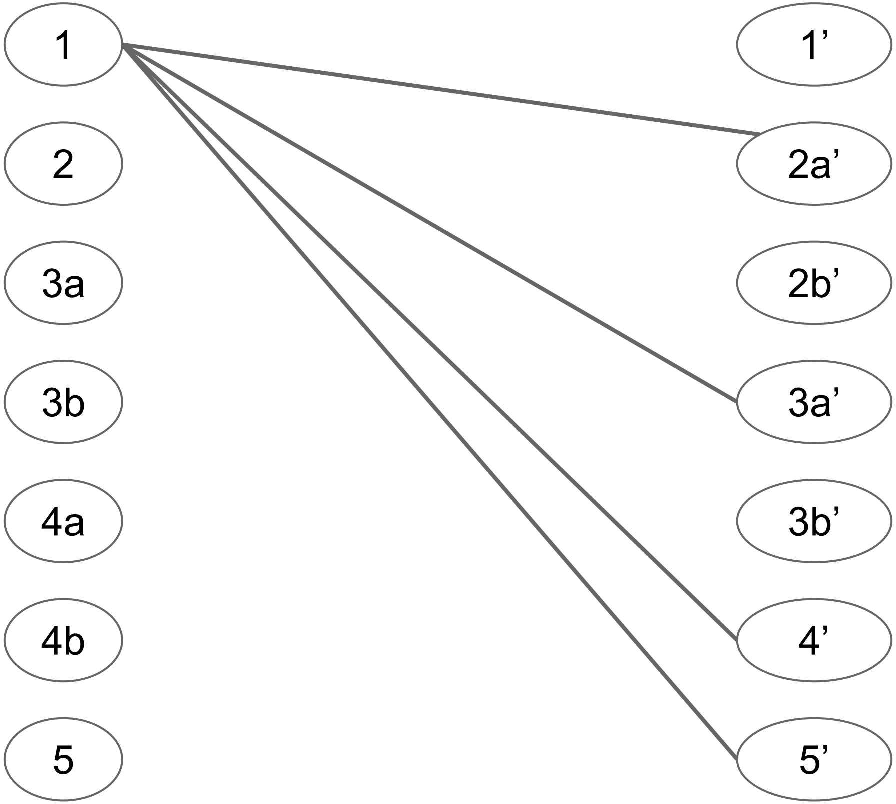
Attacking rooks (UVALive - 6525)

- E quando inserimos os peões? É como se dividíssemos as linhas e colunas em várias partes. Cada parte será representada por um vértice
- Criaremos arestas entre cada par de vértices que representa uma possível posição da torre

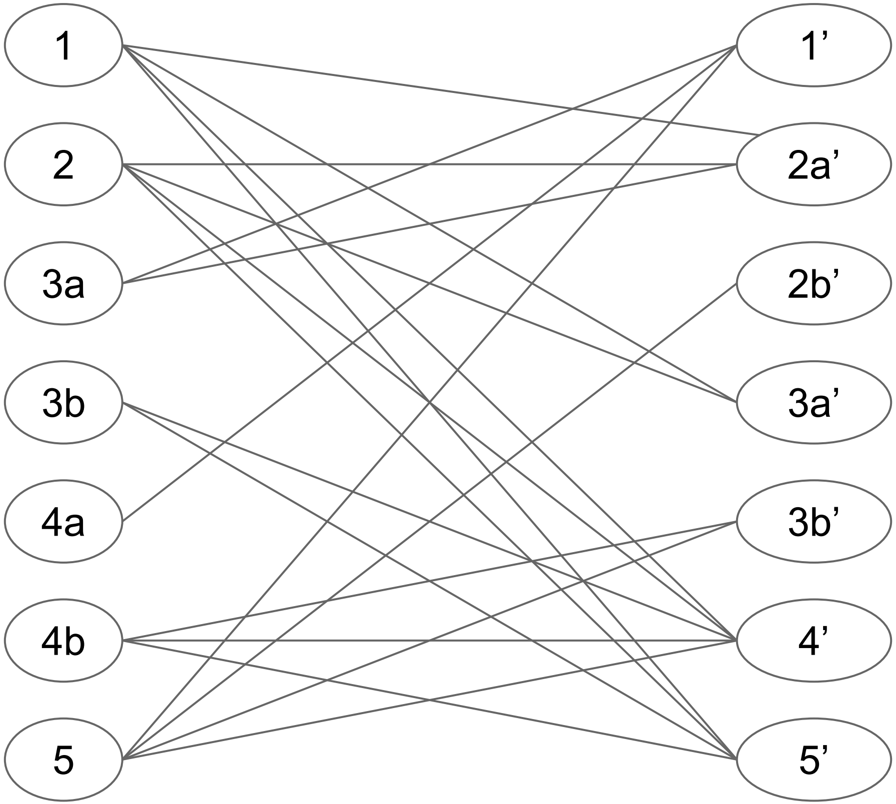
	X	2a’	3a’	4’	5’	1
	X	2a’	3a’	4’	5’	2
3a	1’	2a’	X	4’	5’	3b
4a	1’	X	3b’	4’	5’	4b
5	1’	2b’	3b’	4’	X	

	2a’	3a’	4’	5’
X	1	1	1	1
X	2	2	2	2
3a	3a	X	3b	3b
4a	X	4b	4b	4b
5	5	5	5	X
1’	2b’	3b’		

Attacking rooks (UVALive - 6525)



Attacking rooks (UVALive - 6525)



Referências

https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/vertex-coloring.html

https://www.ic.unicamp.br/~atilio/slidesWtisc.pdf

https://sites.google.com/site/maratonaufabc/topicos/grafos/bicolor

https://neps.academy/lesson/202

https://cp-algorithms-brasil.com/grafos/bipartido.html

http://www.land.ufrj.br/~classes/grafos/jai99.pdf

https://www.ime.usp.br/~pf/algoritmos_em_grafos/aulas/emparelha.html

https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/matching-bipartite.html

https://www.cin.ufpe.br/~hcs/if775/Hopcroft-Karp.pdf