Brazilian ICPC Summer School 2020

- Bipartite Matching
- Vertex Cover
- Independent Set
- Edge Cover
- Minimum Path Cover em DAG

André Amaral de Sousa

Autor

Sobre

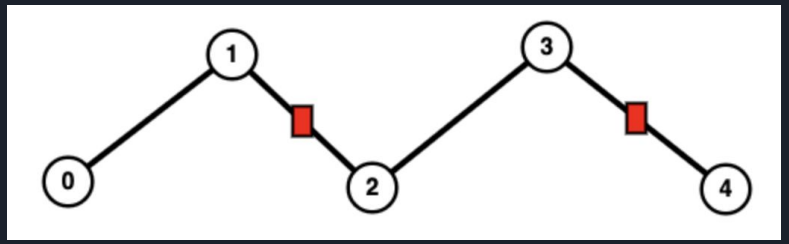
- Ex-Maratonista (ICPC 2013)
- Ex-Olímpico (IOI 2012)
- Problem Setter na Brasileira 2019
- Engenheiro de Computação (POLI-USP)
- Professor no colégio ETAPA
- Contato:
  - andremfq@gmail.com

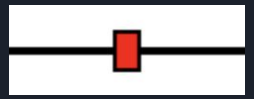
Matching

Emparelhamento (em um grafo qualquer)

Conjunto de arestas de tal forma que cada vértice incide a no máximo uma aresta deste conjunto.

Caminho alternante



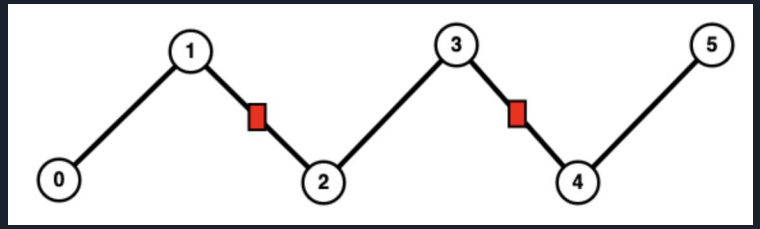


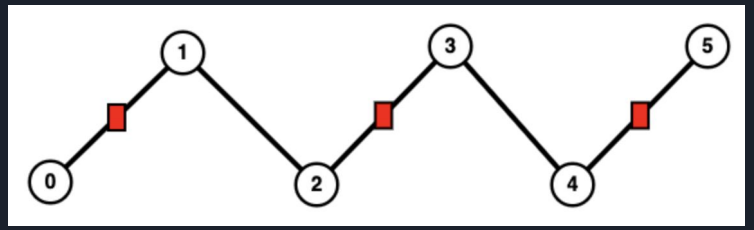
Aresta que pertence ao emparelhamento **M**

Dizemos que um caminho é alternante relativo à um emparelhamento **M** quando as arestas deste caminho se alteram em estar fora de **M** e estar dentro de **M**.

Matching

Caminho aumentante





Aresta que pertence ao emparelhamento **M**

Dizemos que um caminho é aumentante relativo à um emparelhamento **M** quando ele é alternante e começa e termina em vértices fora de **M** (vértices que não incidem em nenhuma aresta de **M**, chamaremos esses vértices de **unmatched**). Note que podemos trocar as arestas do caminho aumentante de forma a conseguir um emparelhamento com exatamente uma aresta a mais.

Matching

Teorema de Berge (para um grafo qualquer)

Um emparelhamento **M** é máximo (maior número de arestas) $\Leftrightarrow$ não possui caminho aumentante

$\Rightarrow$

Tranquilo, pois se **M** é máximo ele não pode ter nenhum caminho aumentante pois se tivesse conseguiríamos encontrar um emparelhamento maior que **M**, o que é absurdo.

Matching

Teorema de Berge (para um grafo qualquer)

Um emparelhamento **M** é máximo $\Leftrightarrow$ não possui caminho aumentante

$\Leftarrow$

- Considere um outro emparelhamento qualquer **N**. Queremos mostrar que **|M| ≥ |N|**.
- Lembre-se que em um emparelhamento o grau de todo vértice é no máximo 1, desta forma se considerarmos apenas as arestas pertencentes à **M** e **N** (vamos chamar esse grafo de **G**) todo vértice terá grau no máximo 2.
- Perceba que **G** é composto apenas por ciclos e caminhos.
- Todo ciclo de **G** é par, pois num ciclo ímpar teríamos duas arestas adjacentes pertencentes ao mesmo emparelhamento, o que é absurdo. Podemos desconsiderar os ciclos (que são pares) pois contribuem com a mesma quantidade de arestas tanto pra **M** quanto pra **N**.

Matching

Teorema de Berge (para um grafo qualquer)

Um emparelhamento **M** é máximo $\Leftrightarrow$ não possui caminho aumentante

$\Leftarrow$

- Desta forma resta analisarmos os caminhos de **G**. Perceba que estes caminhos não podem duas arestas no mesmo emparelhamento, então elas se alternam entre pertencente à **M** e pertencente à **N**.
- Nenhum destes caminhos pode começar em **N** e terminar em **N**, pois desta forma **M** teria um caminho aumentante (estamos assumindo que **M** não possui caminho aumentante).
- Assim, ou um caminho é par (quantidade par de arestas) e contribui igualmente para **M** e para **N**, ou ele é ímpar e começa e termina em **M**, e desta forma contribui com uma aresta a mais para **M**. Portanto temos que $|M| \geq |N|$.

Matching

Ideia de um algoritmo

Usando o teorema de Berge, podemos encontrar um emparelhamento máximo num grafo qualquer da seguinte forma:

- Começa com **M** sendo vazio (sem nenhuma aresta)
- Enquanto houver um caminho aumentante
  - Encontre o caminho aumentante
  - Troque as arestas deste caminho (quem estava fora passa a fazer parte do emparelhamento, e quem estava dentro passa para fora)
- Quando não houver mais caminho aumentante, o emparelhamento será máximo

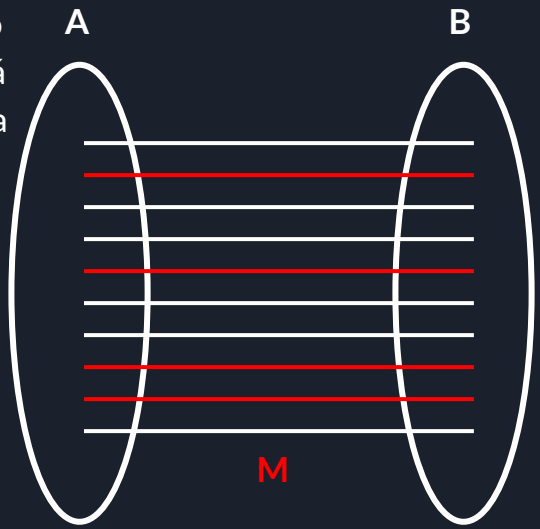
O problema é encontrar um caminho aumentante de forma eficiente num grafo qualquer (e também saber quando não existe mais caminho aumentante). Por isso focaremos em grafos bipartidos, onde esta tarefa é mais simples.

Bipartite Matching

Algoritmo de Kuhn - Ideia

Como encontrar um emparelhamento máximo num grafo Bipartido ?

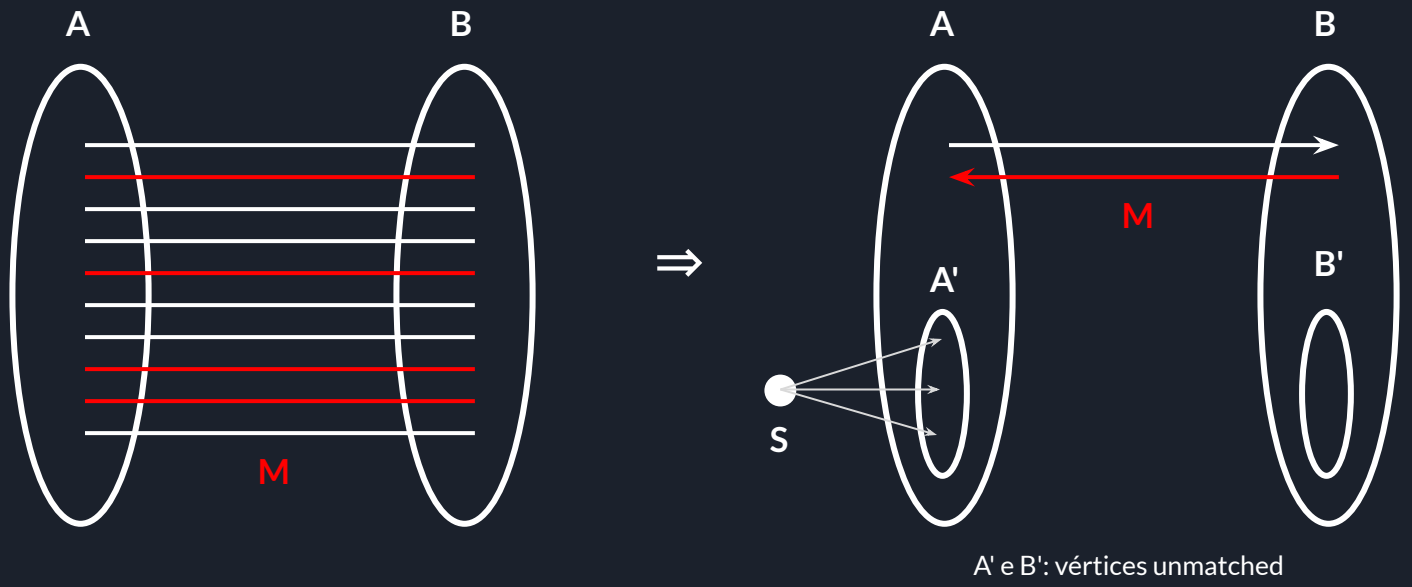
- Inicialmente teremos **M** vazio (sem nenhuma aresta)
- Iremos usar uma **DFS** para determinar se existe caminho aumentante. Se existir, a própria **DFS** retornará verdadeiro e já irá trocar as arestas do caminho aumentante. Se não existir, a **DFS** apenas retornará falso.
- Iremos mostrar que a complexidade é **O(V* (V + E))**.



Bipartite Matching

Algoritmo de Kuhn - DFS

Para a **DFS** iremos trabalhar com o seguinte grafo direcionado (vai fora e volta dentro):

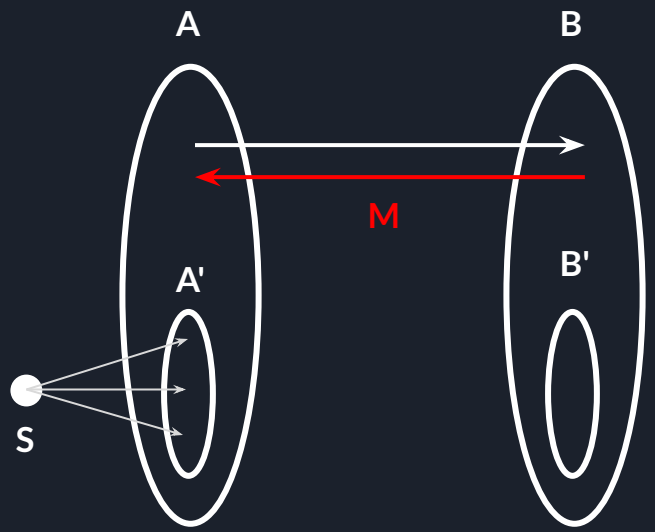


A' e B': vértices unmatched

Bipartite Matching

Algoritmo de Kuhn - DFS

Agora para verificar se existe um caminho aumentante, basta fazer uma **DFS** em **S** e checar se chega em algum vértice de **B'**.



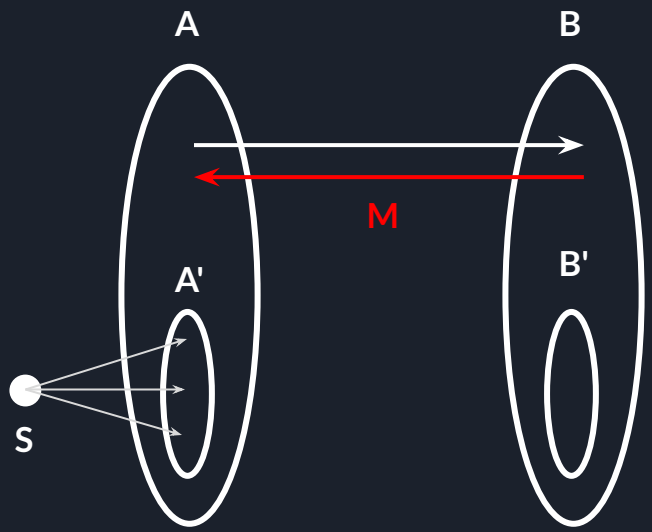
A' e B': vértices unmatched

Bipartite Matching

Algoritmo de Kuhn - DFS

Convenções:

- Os vértices de **A** são numerados de 0 à n - 1.
- Os vértices de **B** são numerados de 0 à m - 1.
- O vetor **matchA[]** será usado para guardar o vértice em **B** que está emparelhado com o respectivo vértice de **A** (no início devemos ter -1).
- O vetor **matchB[]** será usado para guardar o vértice em **A** que está emparelhado com o respectivo vértice de **B** (no início devemos ter -1).
- O vetor **marcB[]** será usado para marcar os vértices de **B** que já foram visitados pela **DFS**.
- O vetor de vectors **grafoA[][]** será usado para guardar a lista de adjacências dos vértices de **A** (contendo os vértices de **B**).



A' e B': vértices unmatched

Bipartite Matching

Algoritmo de Kuhn - DFS

Código:

```
bool dfs(int v) {
  for(int i = 0; i < grafoA[v].size(); i++) {
    int viz = grafoA[v][i];
    if(marcB[viz] == 1) continue;
    marcB[viz] = 1;

    if(matchB[viz] == -1 || dfs(matchB[viz])) {
      matchB[viz] = v;
      matchA[v] = viz;
      return true;
    }
  }
  return false;
}
```

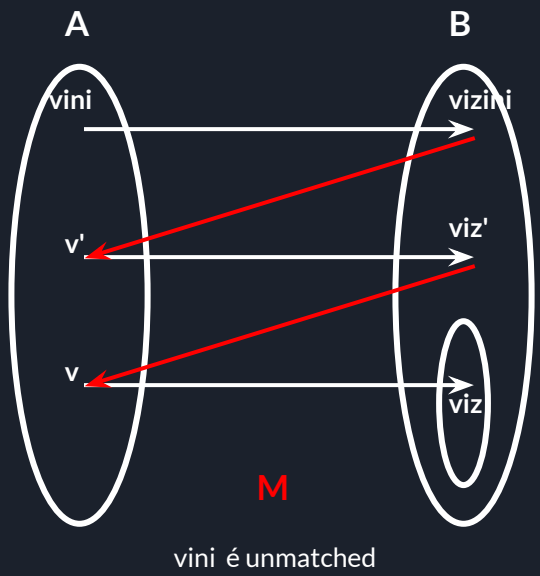
Bipartite Matching

Algoritmo de Kuhn - DFS

Entendendo o Código:

```
bool dfs(int v) {
  for(int i = 0; i < grafoA[v].size(); i++) {
    int viz = grafoA[v][i];
    if(marcB[viz] == 1) continue;
    marcB[viz] = 1;

    if(matchB[viz] == -1 || dfs(matchB[viz])) {
      matchB[viz] = v;
      matchA[v] = viz;
      return true;
    }
  }
  return false;
}
```

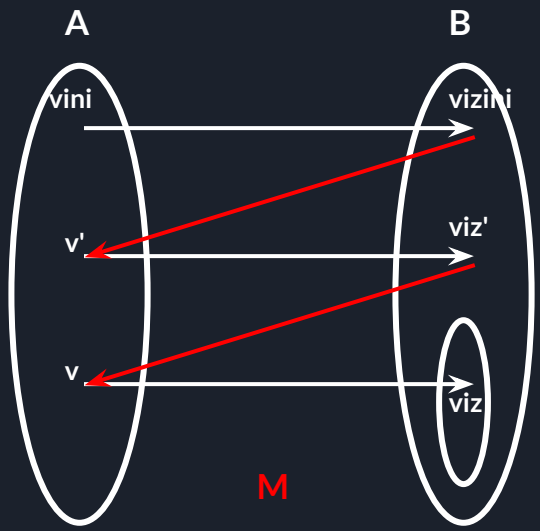


Bipartite Matching

Algoritmo de Kuhn - DFS

Entendendo o Código:

- Quando chegamos em um **viz** que não estava marcado e é unmatched, acabamos de encontrar um caminho aumentante.



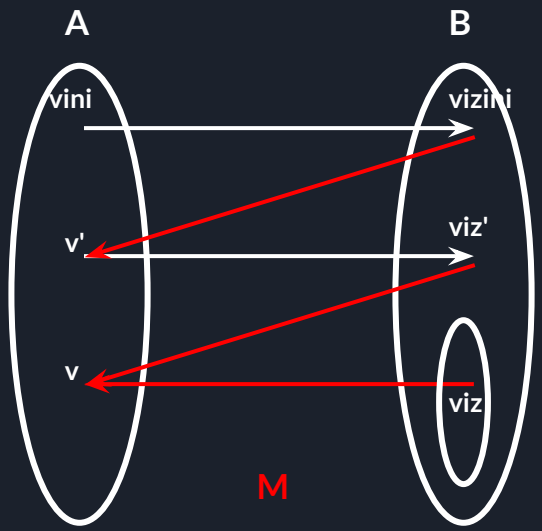
vini é unmatched

Bipartite Matching

Algoritmo de Kuhn - DFS

Entendendo o Código:

- Quando chegamos em um **viz** que não estava marcado e é unmatched, acabamos de encontrar um caminho aumentante.
- Portanto devemos colocar a aresta **(v, viz)** no emparelhamento **M** e retornar verdadeiro na dfs(v).



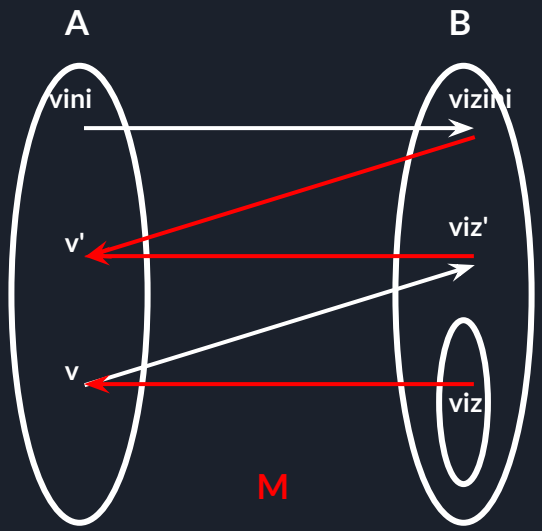
vini é unmatched

Bipartite Matching

Algoritmo de Kuhn - DFS

Entendendo o Código:

- Quando chegamos em um **viz** que não estava marcado e é unmatched, acabamos de encontrar um caminho aumentante.
- Portanto devemos colocar a aresta **(v, viz)** no emparelhamento **M** e retornar verdadeiro na dfs(v).
- Como a dfs(v') chamou a dfs(v) e esta retornou verdadeiro, então a dfs(v') irá retornar verdadeiro também e devemos colocar a aresta (v', viz') no emparelhamento **M** (o que automaticamente fará a aresta (v, viz') sair de **M** pelas nossas convenções).



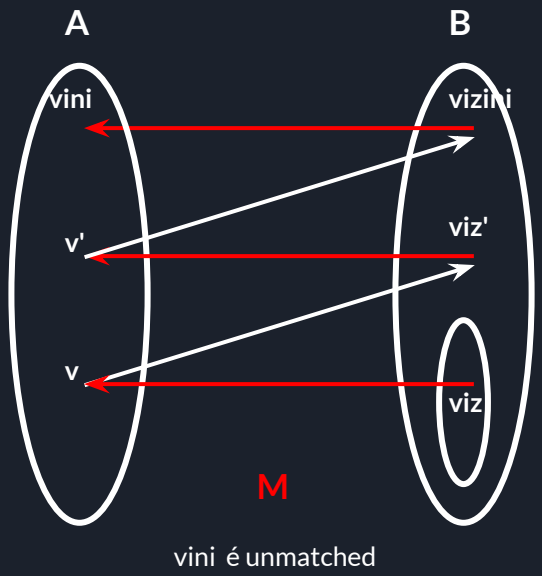
vini é unmatched

Bipartite Matching

Algoritmo de Kuhn - DFS

Entendendo o Código:

- Quando chegamos em um **viz** que não estava marcado e é unmatched, acabamos de encontrar um caminho aumentante.
- Portanto devemos colocar a aresta **(v, viz)** no emparelhamento **M** e retornar verdadeiro na dfs(v).
- Como a dfs(v') chamou a dfs(v) e esta retornou verdadeiro, então a dfs(v') irá retornar verdadeiro também e devemos colocar a aresta (v', viz') no emparelhamento **M** (o que automaticamente fará a aresta (v, viz') sair de **M** pelas nossas convenções).
- De forma análoga dfs(vini) chamou dfs(v') e irá retornar verdadeiro e a aresta (vini, vizini) será adicionada à **M**.



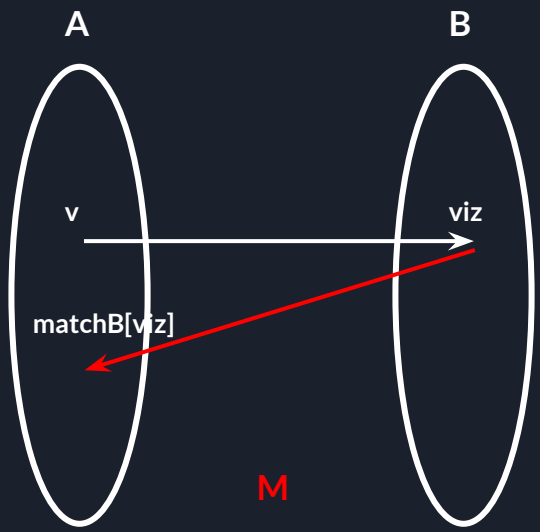
Bipartite Matching

Algoritmo de Kuhn - DFS

Entendendo o Código:

```
bool dfs(int v) {
  for(int i = 0; i < grafoA[v].size(); i++) {
    int viz = grafoA[v][i];
    if(marcB[viz] == 1) continue;
    marcB[viz] = 1;

    if(matchB[viz] == -1 || dfs(matchB[viz])) {
      matchB[viz] = v;
      matchA[v] = viz;
      return true;
    }
  }
  return false;
}
```

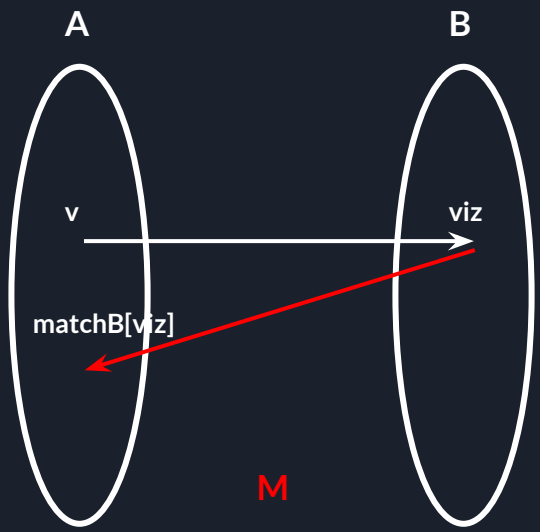


Bipartite Matching

Algoritmo de Kuhn - DFS

Entendendo o Código:

- Quando **viz** é matched, precisamos verificar se existe um caminho aumentante partindo do vértice que está emparelhado com **viz** (matchB[viz]).
- Isso é exatamente o que a **dfs** faz, mas ela supõe que o vértice onde ela começa é **unmatched**. Porém mesmo que **matchB[viz]** não seja **unmatched** podemos fingir que é e chamar a **dfs** partindo dele, e isso vai funcionar porque como acabamos de fazer **marcB[viz] = 1** a **dfs** em **matchB[viz]** não visitará **viz**, portanto é como se fosse **unmatched**.



Bipartite Matching

Algoritmo de Kuhn - usando a DFS

Código:

```
for (int i = 0; i < n; i++) matchA[i] = -1;
for (int j = 0; j < m; j++) matchB[j] = -1;

int resp = 0;
for (int i = 0; i < n; i++) {
  for (int j = 0; j < m; j++) marcB[j] = 0;
  if(dfs(i)) resp++;
}
```

Complexidade: Note que estamos chamando uma nova **dfs** para cada vértice de **A**, assim a complexidade fica:

O(N * (N+M+E)) = O(V*(V+E))

Bipartite Matching

Algoritmo de Kuhn - usando a DFS

A princípio para verificar se existe caminho aumentante seria necessário fazer uma **DFS** que partisse de todo vértice **unmatched** (ou do vértice **S** que estaria ligado apenas aos **unmatched**), mas fizemos apenas uma chamada da **dfs** para cada vértice em **A** e nunca mais voltamos a verificar ele.

Porque isso funciona ?

Para entender precisamos de algumas observações

Bipartite Matching

Algoritmo de Kuhn - usando a DFS

Observações:

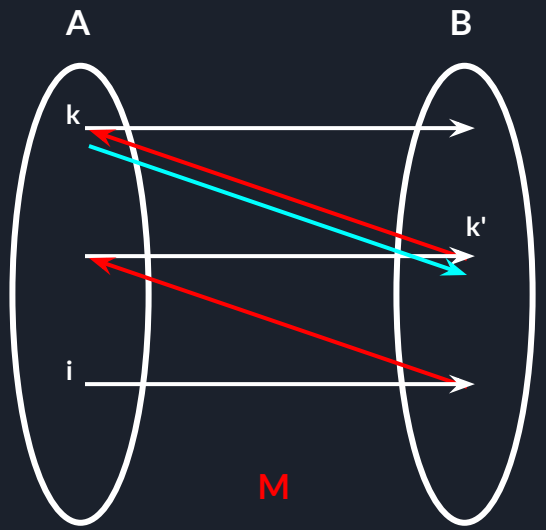
- Todo vértice intermediário (todos menos o primeiro e o último) de um caminho aumentante é **matched** (está emparelhado com algum outro vértice).
- Quando trocamos as arestas de um caminho aumentante **todos** os vértices passam a ser **matched**.
- Se a **dfs** chamada em **i** encontrar um caminho aumentante, este caminho não irá passar por nenhum vértice em **A** maior que **i**, pois todo vértice intermediário de um caminho aumentante é **matched** e no momento da chamada da **dfs** em **i**, todos os vértices maiores que **i** estão **unmatched**.
- Suponha que ao chamar a **dfs** em **i** encontramos um caminho aumentante, então a partir daí o vértice **i sempre** estará **matched** (pois ao realizar a troca de um caminho aumentante todos os vértices ficam **matched**) e portanto nunca mais existirá um caminho aumentante partindo de **i**.

Bipartite Matching

Algoritmo de Kuhn - usando a DFS

Observações:

- Suponha que ao chamar a **dfs** em **i** não encontramos um caminho aumentante.
  - Suponha que logo após chamar a dfs em j (j > i) passa a existir um caminho aumentante partindo de i.
  - A dfs em j tem que ter encontrado um caminho aumentante, senão nada mudaria.
  - Imagine o caminho aumentante de i que se formou, o caminho aumentante de j tem que passar por alguma aresta do caminho aumentante de i (senão nada mudaria).
  - Imagine que seja a aresta (k, k') que estava fora de M, e que após trocar o caminho aumentante de j, virou (k', k) e passou a ser de M. Isso significa que a dfs de j encontrou um caminho de k' a algum vértice unmatched de B, e portanto como existe um caminho de i a k', então existia um caminho aumentante partindo de i antes da dfs de j, o que é absurdo.

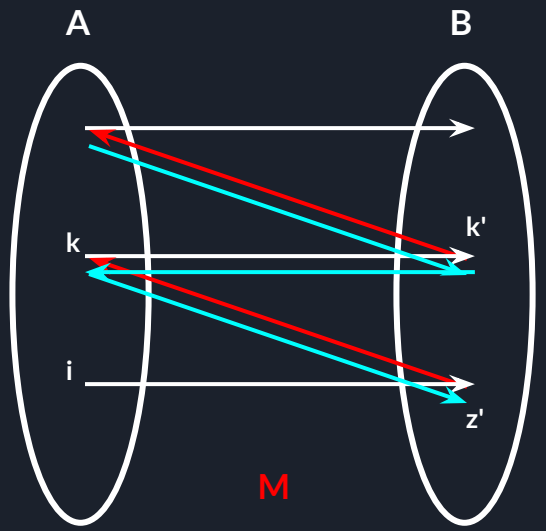


Bipartite Matching

Algoritmo de Kuhn - usando a DFS

Observações:

- Suponha que ao chamar a **dfs** em **i** não encontramos um caminho aumentante.
  - Suponha que logo após chamar a dfs em j (j > i) passa a existir um caminho aumentante partindo de i.
  - A dfs em j tem que ter encontrado um caminho aumentante, senão nada mudaria.
  - Imagine o caminho aumentante de i que se formou, o caminho aumentante de j tem que passar por alguma aresta do caminho aumentante de i (senão nada mudaria).
  - Imagine que seja a aresta (k', k) que estava dentro de M, e que após trocar o caminho aumentante de j, virou (k, k') e saiu de M. Isso significa que o caminho aumentante de j tem que ter passado por z' também (pois senão k teria duas arestas em M), e assim analogamente ao caso anterior, existiria um caminho aumentante partindo de i e passando por z', antes da dfs de j , o que é absurdo.



Bipartite Matching

Algoritmo de Kuhn - usando a DFS

Observações:

- Suponha que ao chamar a **dfs** em **i** não encontramos um caminho aumentante. Vimos que nunca mais existirá um caminho aumentante partindo de **i**.
- Juntando todas as observações vemos que o algoritmo de fato funciona.

Bipartite Matching

Algoritmo de Kuhn - melhorias

1ª: Zerar o vetor de marcação da **DFS** apenas quando encontrar um caminho aumentante

```
int resp = 0;
for (int j = 0; j < m; j++) marcB[j] = 0;
for (int i = 0; i < n; i++) {
  if(dfs(i)) {
    resp++;
    for (int j = 0; j < m; j++) marcB[j] = 0;
  }
}
```

Note que podemos zerar o **marcB[]** apenas quando encontramos um caminho aumentante, pois quando não encontramos nada muda, então podemos aproveitar a **DFS**.

A complexidade fica **O(|$M_{máximo}$| * E)**.

Bipartite Matching

Algoritmo de Kuhn - melhorias

2ª: Fast Kuhn

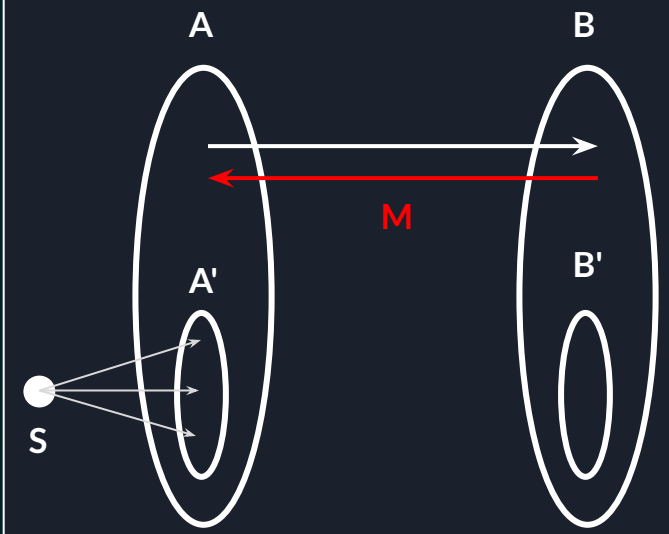
```
bool aux = true;
int resp = 0;
while(aux) {
  for (int j = 0; j < m; j++) marcB[j] = 0;
  aux = false;
  for(int i = 0; i < n; i++) {
    if(matchA[i] != -1) continue;
    if(dfs(i)) {
      resp++;
      aux = true;
    }
  }
}
```

Bipartite Matching

Algoritmo de Kuhn - melhorias

2ª: Fast Kuhn - Entendendo o Código

```
bool aux = true;
int resp = 0;
while(aux) {
  for (int j = 0; j < m; j++) marcB[j] = 0;
  aux = false;
  for(int i = 0; i < n; i++) {
    if(matchA[i] != -1) continue;
    if(dfs(i)) {
      resp++;
      aux = true;
    }
  }
}
```



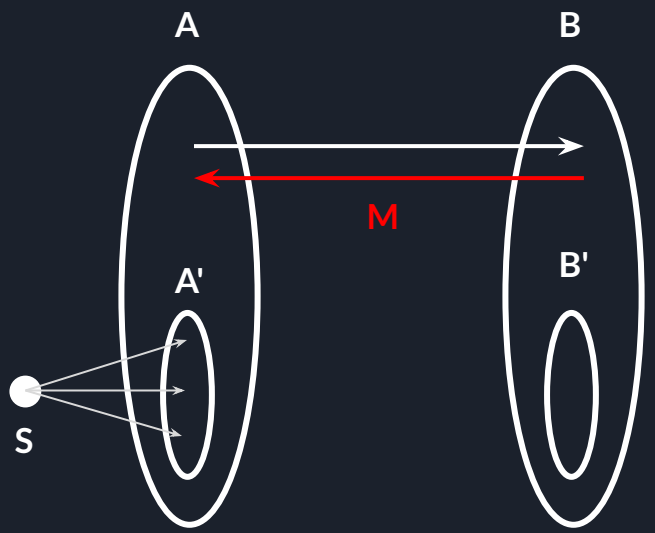
A' e B': vértices unmatched

Bipartite Matching

Algoritmo de Kuhn - melhorias

2ª: Fast Kuhn - Entendendo o Código

- Em uma **dfs** tentamos encontrar vários caminhos aumentantes, não apenas um (não é em uma chamada da **dfs**, mas sim sem zerar o vetor de marcação da **dfs**).
- Para isso passamos por todos os vértices em **A** e chamamos a **dfs** apenas quando ele é **unmatched**.
- Se após chamarmos a **dfs** em todos os vértices **unmatched** de **A** não encontrarmos nenhum caminho aumentante novo, é porque não há mais nenhum caminho aumentante, e portanto podemos parar pois já encontramos o matching máximo (essa é a função da variável auxiliar **aux**).



A' e B': vértices unmatched

Bipartite Matching

Algoritmo de Kuhn - melhorias

2ª: Fast Kuhn - Complexidade

- Nunca vi nenhuma prova de complexidade do **Fast Kuhn** (se souber me avise!)
- Testei em problemas que precisavam de **Hopcroft-Karp**, e passou em todos que testei até hoje.
- A complexidade de **Hopcroft-Karp** é **O(√V * E)**. Acredito que seja parecida.
- O código final é bem simples (acredito que seja mais rápido de codar que **Hopcroft-Karp**)

Vertex Cover

Vertex Cover (em um grafo qualquer)

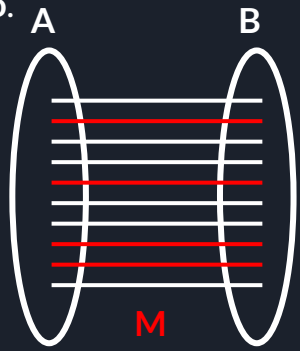
Conjunto de vértices, tal que toda aresta possui pelo menos um de seus vértice neste conjunto (é um conjunto de vértices que cobre todas as arestas).

Desigualdade

$$|V_{\text{mínimo}}| \geq |M_{\text{máximo}}|$$

O tamanho do menor **Vertex Cover** é maior ou igual ao tamanho do **Matching Máximo**.

Para cada aresta do Matching Máximo um Vertex Cover qualquer deve escolher pelo menos um dos vértices dela, mas como nenhuma aresta do Matching Máximo possui vértices em comum, então todos estes vértices serão distintos, por isso vale a desigualdade acima.



Vertex Cover

Teorema de Kőnig

Em um grafo bipartido vale a igualdade:

$$|V_{\text{mínimo}}| = |M_{\text{máximo}}|$$

Prova de forma construtiva, iremos descrever um algoritmo para determinar um conjunto de vértices, e depois provaremos que tal conjunto será um **Vertex Cover** e que seu tamanho é o mesmo de um **Matching Máximo**.

Vertex Cover

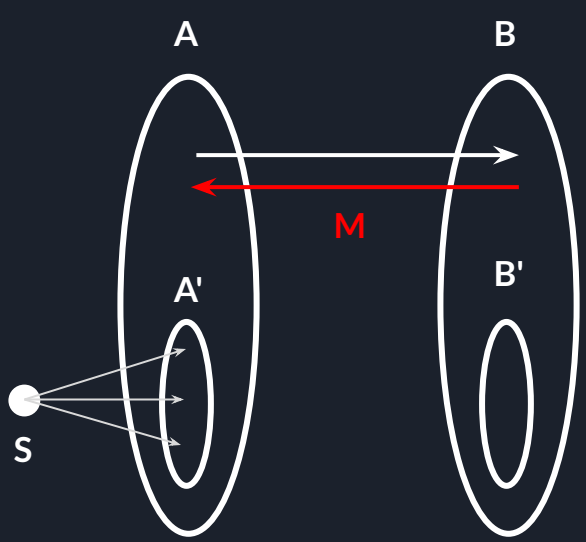
Teorema de Kőnig

- Achar o **Matching Máximo**, por exemplo usando **Kuhn**
- Trabalhando com a mesma convenção de direção das arestas (vai fora e volta dentro), fazer uma **dfs** partindo de todos os vértices **unmatched** de **A**. Mas agora marcando os vértices em **A** também (usar **marcA[]**)
- Seja $\mathbf{L_A}$ o conjunto de todos os vértices marcados pela **dfs** que estão em **A**.
- Seja $\mathbf{L_B}$ o conjunto de todos os vértices marcados pela **dfs** que estão em **B**.
- Tome $\mathbf{C = (A - L_A) \cup L_B}$.
- **C** será **Vertex Cover** (mostraremos).
- O tamanho de **C** será o mesmo do **Matching Máximo** (mostraremos).

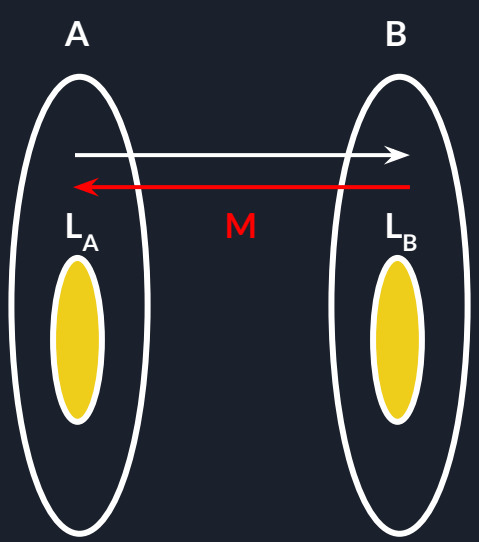
Vertex Cover

Teorema de Kőnig

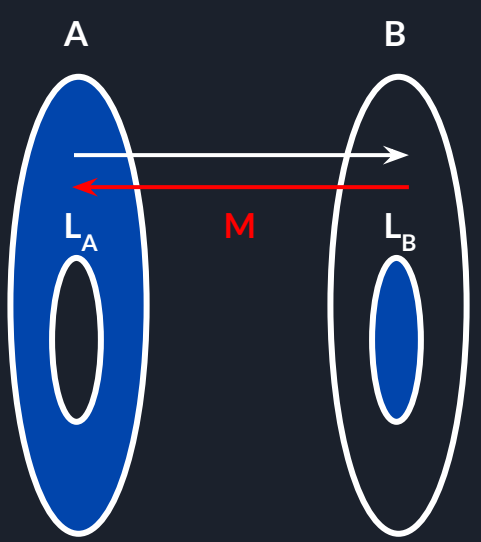
$$C = (A - L_A) \cup L_B$$



A' e B': vértices unmatched



L_A e L_B: vértices marcados

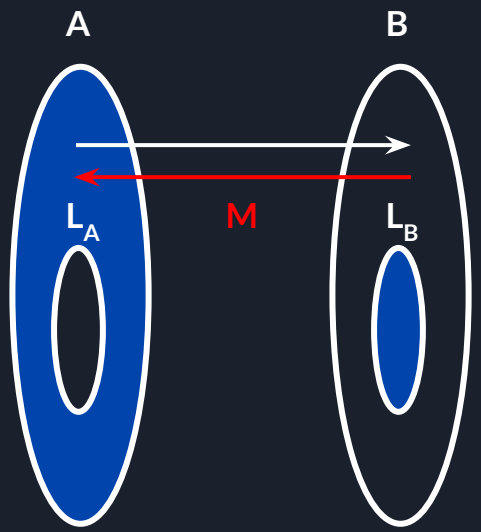


C: Vertex Cover

Vertex Cover

Teorema de Kőnig - C = (A - L_A) U L_B é Vertex Cover

- Para que **C** não seja **Vertex Cover** ele não pode cobrir todas as arestas, isso significa que precisa existir uma aresta **(a, b)** com **a** em L_A e b em **(B - L_B)**
- Existem duas possibilidades para esta aresta **(a, b)**:
  - **(a, b)** não está no **Matching Máximo**, e é direcionada de **a** para **b**.
  - **(a, b)** está no **Matching Maximo**, e é direcionada de **b** para **a**.
- Vamos analisar as duas possibilidades.

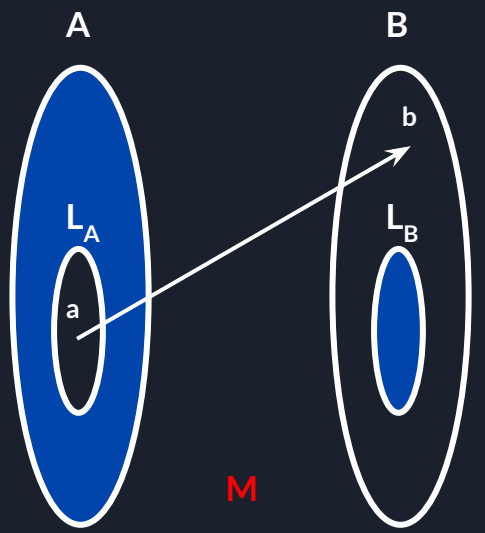


C: Vertex Cover

Vertex Cover

Teorema de Kőnig - C = (A - L_A) U L_B é Vertex Cover

- **(a, b)** não está no **Matching Máximo**, e é direcionada de **a** para **b**.
- Como **a** está em L_A , significa que a **dfs** (que marcou **L**) chegou em **a**, e como a direção é de **a** para **b**, então **b** deveria ter sido marcado também.
- Logo é absurdo, e portanto não pode existir esta aresta.



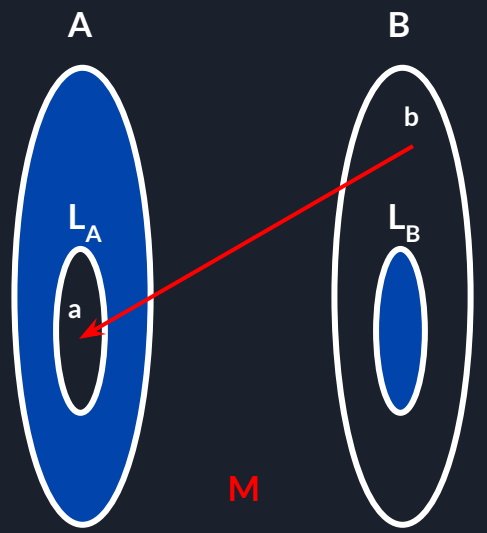
C: Vertex Cover

Vertex Cover

Teorema de Kőnig - C = (A - L_A) U L_B é Vertex Cover

- **(a, b)** está no **Matching Máximo**, e é direcionada de **b** para **a**.
- Como **(a, b)** está no **Matching**, então ambos são **matched**.
- Isso significa que a **dfs** (partiu dos **unmatched**) não partiu de **a**.
- Para a **dfs** alcançar **a** ela teve que chegar nele, mas como ele é **matched** só há uma aresta que chega nele, que é **(a, b)**.
- Portanto para a **dfs** chegar em **a**, ela tem que ter passado por **b**, mas **b** não foi marcado pela **dfs**.
- Logo é absurdo, e portanto não pode existir esta aresta.

Como analisamos as duas possibilidades e chegamos em contradições, então não existem arestas descobertas por **C** e assim é **Vertex Cover**.

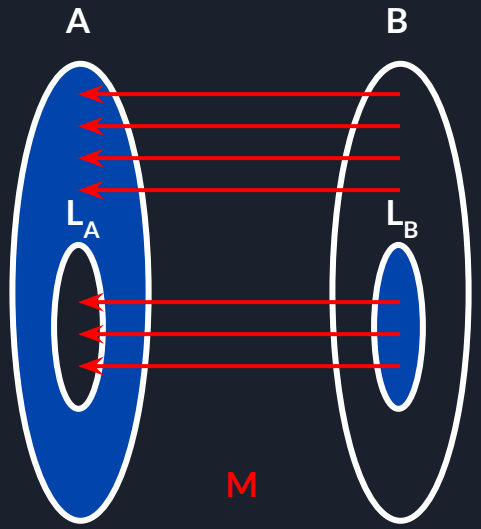


C: Vertex Cover

Vertex Cover

Teorema de Kőnig - $|C| = |M_{máximo}|$

- Se **a** está em **A - L_A** então **a** é **matched**, pois a **dfs** partiu de todos os **unmatched** de **A**. Note que **L_A** possui alguns **matched** de **A** e **todos** os **unmatched**.
- Se **b** está em **L_B** então **b** é **matched**, pois como o **Matching** é **Máximo**, não pode haver caminho aumentante, isso significa que a **dfs** (que partiu dos **unmatched** de **A**) não pode chegar em **unmatched** de **B**, portanto todos em **L_B** são **matched**.
- Todo **a** que está em **A - L_A** está emparelhado com algum vértice em **B - L_B**, pois se fosse de **L_B** então a deveria estar marcado e estaria em **L_A**.
- De forma análoga, todo b que está em **L_B** está emparelhado com algum vértice de **L_A**.
- Conforme vimos anteriormente não pode haver aresta entre **L_A** e **(B - L_B)**.

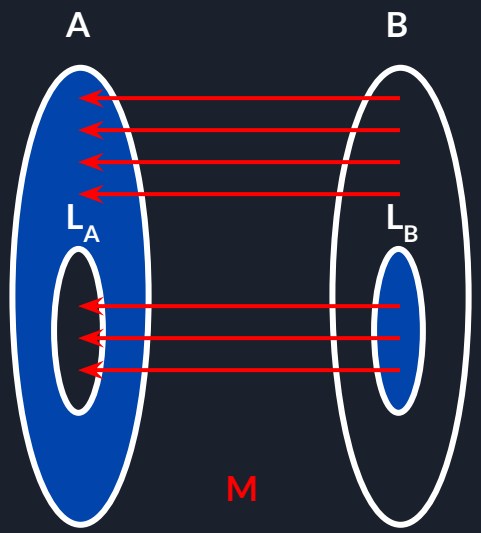


C: Vertex Cover

Vertex Cover

Teorema de Kőnig - $|C| = |M_{máximo}|$

- Todas estas observações juntas mostram que o tamanho de **C** é igual ao tamanho do **Matching Máximo**.
- Como $|V_{mínimo}| \geq |M_{máximo}|$ então além de **C** ser **Vertex Cover**, ele é mínimo.
- E assim mostramos que para todo grafo bipartido $|V_{mínimo}| = |M_{máximo}|$.



C: Vertex Cover

Independent Set

Independent Set (em um grafo qualquer)

Conjunto de vértices, tal que não exista aresta entre quaisquer dois vértices do conjunto.

Exercício

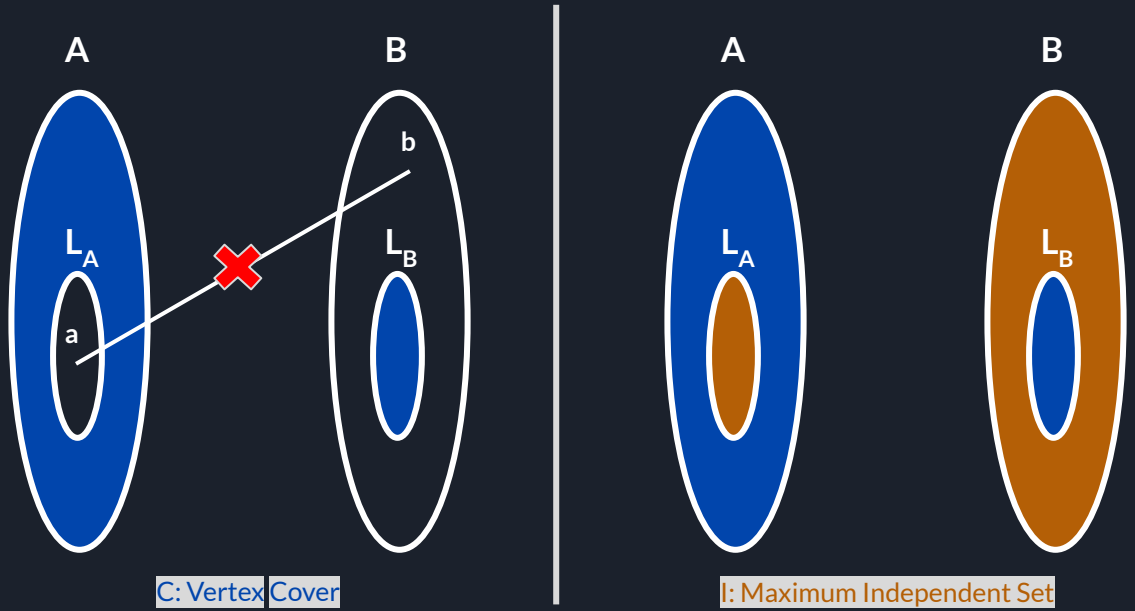
Qual o maior tamanho de um Independent Set em um grafo bipartido (**Maximum Independent Set**) ?

Independent Set

Maximum Independent Set para Grafo Bipartido

Basta pegar todos os vértices que **não** estão no **Minimum Vertex Cover.**

- $|I_{máximo}| = N - |V_{mínimo}| = N - |M_{máximo}|$



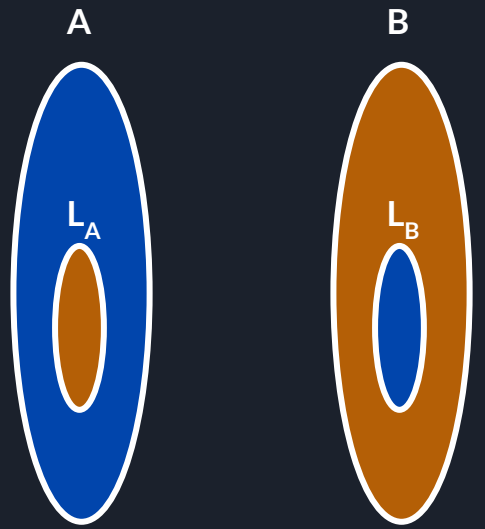
Independent Set

Maximum Independent Set para Grafo Bipartido

Porque funciona ?

- Podemos imaginar que queremos arrancar o número mínimo de vértices (ao arrancar um vértice, arrancamos também todas as suas arestas) de tal forma a arrancar todas as arestas.
- O conjunto de vértices que queremos arrancar é o **Minimum Vertex Cover**, pois ele cobre todas as arestas com o menor número de vértices.

Dado que é um grafo bipartido, tanto **A** quanto **B** são Independent Sets, porque pegar todos os vértices menos o **Minimum Vertex Cover** é sempre melhor (ou igual) a pegar **A** ou pegar **B** ?



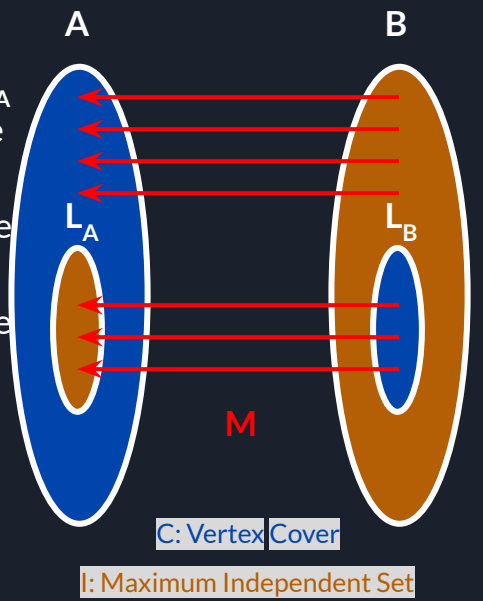
I: Maximum Independent Set

Independent Set

Maximum Independent Set para Grafo Bipartido

Dado que é um grafo bipartido, tanto **A** quanto **B** são Independent Sets, porque pegar todos os vértices menos o **Minimum Vertex Cover** é sempre melhor (ou igual) a pegar **A** ou pegar **B** ?

- Vimos que se **b** está em L_B então **b** é **matched** com algum vértice de L_A
- L_A possui todos os unmatched de A, e alguns matched que correspondem aos vértices em L_B, assim $|L_A| \geq |L_B|$.
- Vimos que se a está em A - L_A então a é matched com algum vértice de B - L_B.
- B - L_B possui todos os vértices unmatched de B, e alguns matched que correspondem aos vértices em A - L_A, assim $|B - L_B| \geq |A - L_A|$.
- $|I_{máximo}| = |L_A| + |B - L_B| \geq |L_B| + |B - L_B| = |B|$.
- $|I_{máximo}| = |L_A| + |B - L_B| \geq |L_A| + |A - L_A| = |A|$.



Edge Cover

Edge Cover (em um grafo qualquer)

Conjunto de arestas que cobrem todos os vértices (todo vértice possui alguma aresta deste conjunto, podemos desconsiderar os vértices com grau zero)

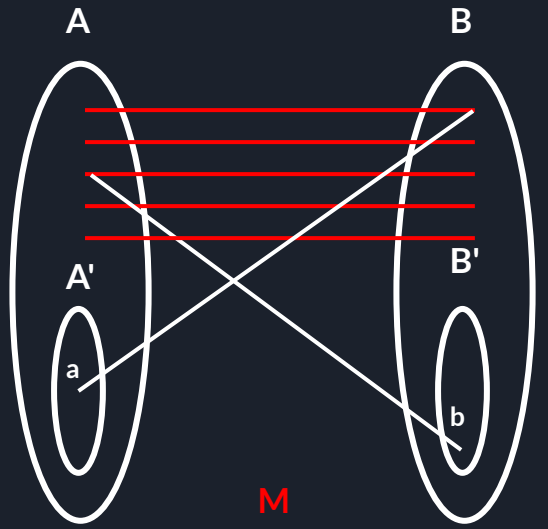
Exercício

Qual o menor tamanho de um Edge Cover em um grafo bipartido (**Minimum Edge Cover**) ?

Edge Cover

Minimum Edge Cover para Grafo Bipartido

- Cada aresta que pegamos para o Edge Cover contribui com 1 ou 2 vértices novos (se for 0 não faz sentido incluir pois queremos o mínimo).
- É sempre ótimo maximizar o número de arestas que contribuem com 2 vértices novos. Mas perceba que estas arestas não podem se interferir, ou seja, não devem ter vértices em comum. Note que nesta parte devemos pegar as arestas de um **Matching Máximo**.
- Ainda devemos cobrir os vértices de **A'** e **B'** (**unmatched**), e só conseguiremos com arestas que irão contribuir com apenas 1 vértice novo (pois se existisse uma aresta de **A'** para **B'**, o Matching não seria Máximo).

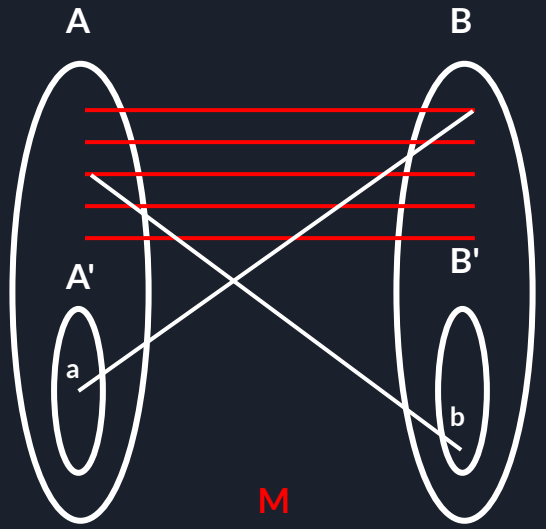


A' e B': vértices unmatched

Edge Cover

Minimum Edge Cover para Grafo Bipartido

- Então para cobrir os vértices de **A'** basta escolher uma aresta pra cada vértice (uma aresta qualquer, pois necessariamente cairá fora de **B'**). Idem para os vértices de **B'**.
- Desta forma teremos:
- $|E_{mínimo}| = |M_{máximo}| + |A'| + |B'|$
- $|A'| = |A| - |M_{máximo}|$
- $|B'| = |B| - |M_{máximo}|$
- $|E_{mínimo}| = |M_{máximo}| + |A| - |M_{máximo}| + |B| - |M_{máximo}|$
- $|E_{mínimo}| = |A| + |B| - |M_{máximo}|$
- $|E_{mínimo}| = N - |M_{máximo}|$

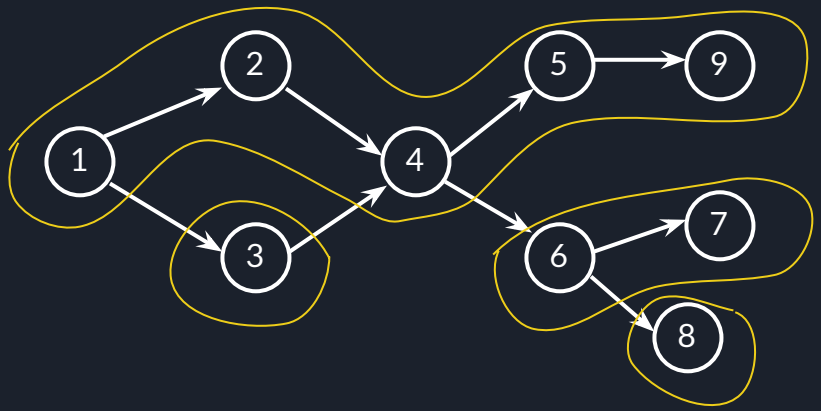


A' e B': vértices unmatched

Minimum Path Cover em DAG

Exercício

Dado um DAG (Directed Acyclic Graph) determine o menor número de caminhos de forma a cobrir todos os vértices (cada vértice só pode pertencer a um único caminho).

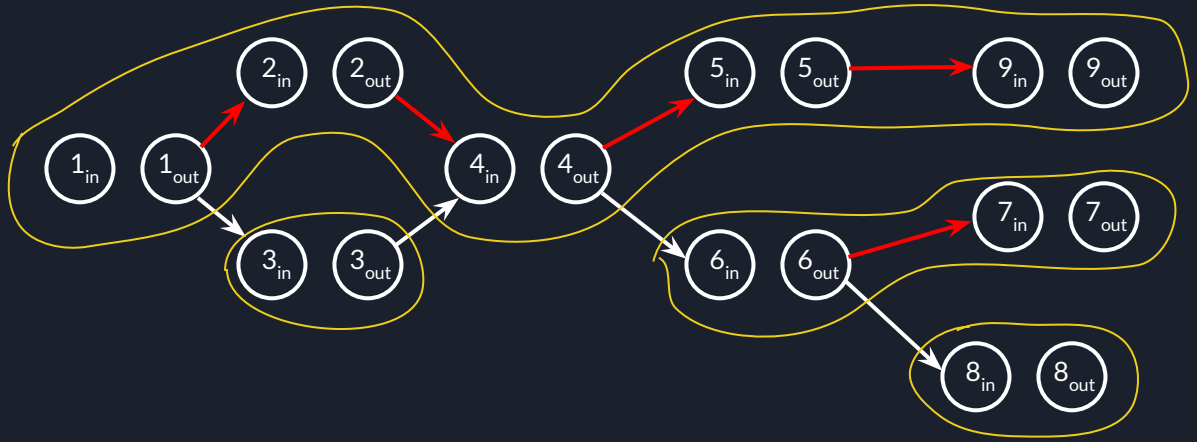


4 caminhos

Minimum Path Cover em DAG

Exercício - Solução

- Podemos expandir cada vértice em dois, seu vértice de entrada e seu vértice de saída.
- Perceba que ao ligar a saída de um vértice à entrada de outro, os dois fazem parte do mesmo caminho.

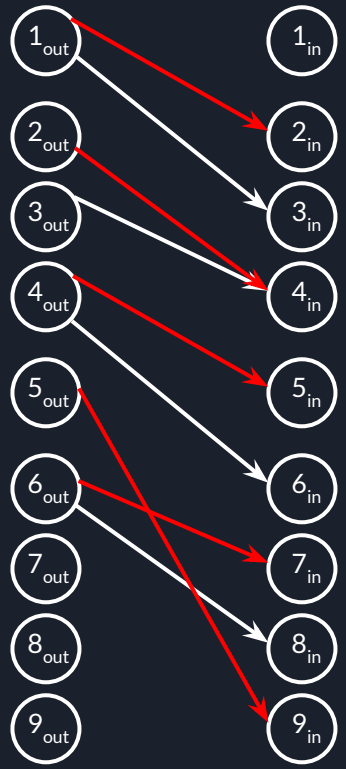


4 caminhos

Minimum Path Cover em DAG

Exercício - Solução

- Quanto mais arestas escolhemos, menos caminhos terão, logo queremos maximizar o número de arestas.
- Mas para manter a restrição que cada vértice pertença a exatamente um caminho, no DAG expandido cada vértice de saída só pode ter no máximo uma aresta escolhida, e também cada vértice de entrada só pode ter no máximo uma aresta escolhida.
- Note que o DAG expandido é bipartido, e maximizar as arestas escolhidas equivale a encontrar o **Matching Máximo**.
- Para calcular o tamanho do **Minimum Path Cover** podemos pensar que inicialmente cada vértice está num caminho isolado, e a cada aresta escolhida estamos unindo dois caminhos.
- $|P_{mínimo}| = N - |M_{máximo}|$



Bipartite Matching

Lista de Exercícios

- MATCHING
- Attacking Rooks - Brasileira 2013
- Angels and Devils
- Muddy Fields
- Technobabble
- No Cheating
- SAM I AM
- Bounty Hunter II
- Keep It Covered
- Piece It Together