Brazilian ICPC Summer School 2020

- Functional Graphs
- MinQueue

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

Functional Graphs

$f(x) = y$	
x	y
1	8
2	17
3	16
4	5
5	10
6	3
7	3
8	4
9	10
10	14
11	3
12	8

$f(x) = y$	
x	y
13	15
14	2
15	9
16	4
17	11
18	3
19	21
20	19
21	20
22	22
23	22
24	22
25	24

Functional Graphs

Definição

Dada uma Função $f : \mathbb{N} \rightarrow \mathbb{N}$ vamos construir o Grafo dessa função, ou seja para cada $f(x) = y$ iremos criar uma aresta direcionada de x para y

Como deve ser o formato deste Grafo ?

Functional Graphs

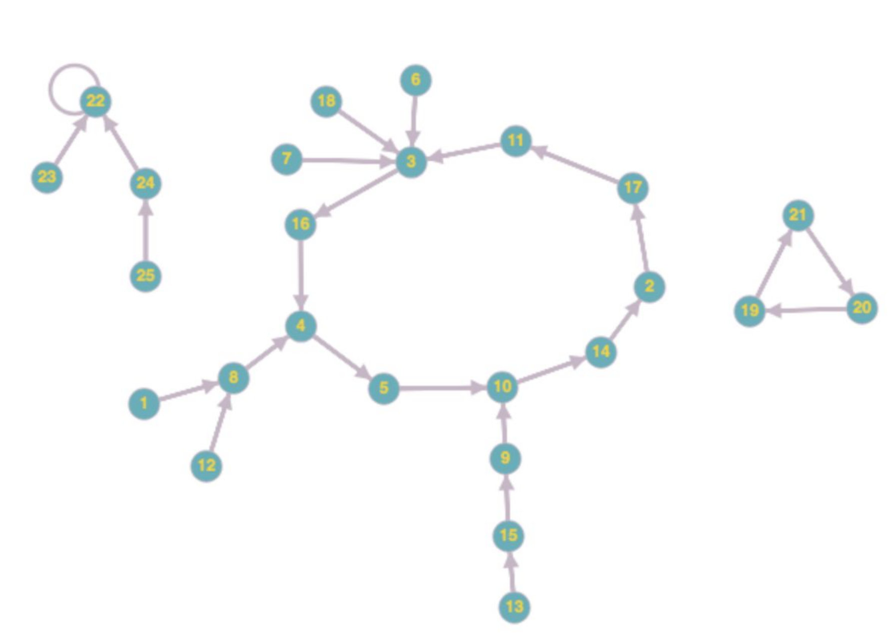
Características

- Para todo vértice temos EXATAMENTE uma aresta saindo dele
- Se partirmos de um vértice e formos andando pela aresta que sai do vértice atual, como sempre existe tal aresta e o número de vértices é finito, então em algum momento iremos chegar em um ciclo. Ou seja, TODO vértice chega em um ciclo
- Cuidado, nem todo vértice pertence à um ciclo
- Vale ressaltar que se para todo vértice, além de sair exatamente uma aresta, também chega exatamente uma aresta (em outras palavras se a função f for bijetora) aí teríamos que todo vértice pertence à um ciclo, e o grafo seria composto apenas por ciclos

Functional Graphs

Formato do Grafo

Pelas características anteriores, concluímos que o grafo é composto por ciclos, e árvores ligadas à estes ciclos:



Fonte: Graph Online

Functional Graphs

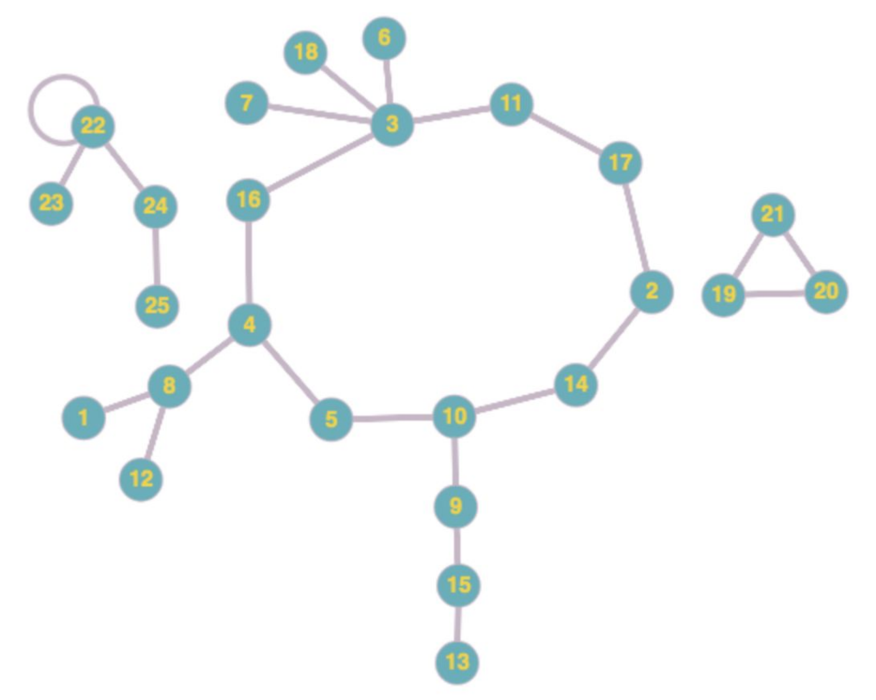
Não Direcionado

Existe um tipo de grafo que possui o mesmo formato geral, mas não é direcionado.

Podemos descrever como um grafo não direcionado onde cada componente conexa possui a mesma quantidade de vértices e arestas. Podemos imaginar também cada componente conexa deste grafo como uma árvore + uma aresta.

Functional Graphs

Não Direcionado



Functional Graphs

Estruturar o Grafo

Para lidarmos com este tipo de grafo, é necessário saber calcular algumas informações, elas variam de acordo com o que o problema pede, mas vamos discutir as mais comuns:

Informações relativas aos vértices:

- **pai[v]** - vértice vizinho de v que está mais próximo de um ciclo que v.
- **paiCiclo[v]** - vértice mais próximo de v, que está num ciclo (primeiro vértice que v alcança que está num ciclo)
- **ciclo[v]** - Índice do ciclo que o v alcança (colocaremos um índice para cada ciclo para diferenciá-los)
- **noCiclo[v]** - Guarda se v está num ciclo ou não (V/F ou 0/1)
- **idNoCiclo[v]** - Índice de v no ciclo. Caso v não esteja no ciclo podemos colocar -1
- **prof[v]** - Profundidade na árvore, ou quantas arestas até v chegar no paiCiclo[v]
- **sub[v]** - Quantidade de vértices na subárvore de v
- **Informações da Árvore (por ex DP na árvore)**

Functional Graphs

Estruturar o Grafo

Para lidarmos com este tipo de grafo, é necessário saber calcular algumas informações, elas variam de acordo com o que o problema pede, mas vamos discutir as mais comuns:

Informações relativas aos ciclos:

- **tam[c]** - Tamanho do ciclo de índice c
- **ini[c]** - Vértice de índice 0 do ciclo de índice c
- **ciclos[c][]** - Vector contendo todos os vértices do ciclo c na ordem correta.
- **qtdCiclos** - Quantidade de ciclos

Functional Graphs

Primeira Estratégia - Whiles

Esta estratégia funciona bem no caso direcionado, pois como já é dada a aresta que sai de cada vértice (já temos o pai[v]) fica muito fácil "ir andando" no grafo.

Mas é ruim para o caso não direcionado, e também é ruim para calcular informações da árvore que dependam dos filhos (por exemplo sub[v]).

Ela consiste em marcar os vértices (criar um **marc[v]**) com a seguinte convenção:

- 0: vértice ainda não encontrado
- 1: vértice já encontrado, mas ainda não determinado, ou seja, estamos calculando
- 2: vértice que está em algum ciclo
- 3: vértice que não está em um ciclo

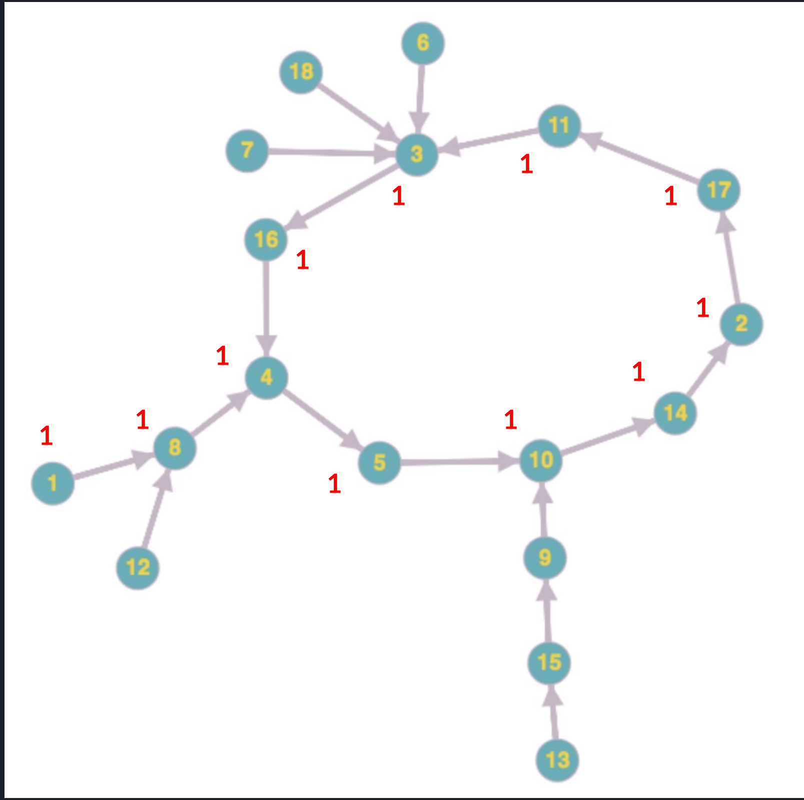
Functional Graphs

Primeira Estratégia - Whiles

O processo de marcar os vértices consiste em:

- Passa por todos os vértices, sempre que encontrar algum vértice marcado como 0, começa o processo nele
- Enquanto o vértice atual estiver marcado como 0, marque ele como 1 e vai para o pai dele

```
void achei(int v) {
  int vini = v;
  while(marc[v] == 0) {
    marc[v] = 1;
    v = pai[v];
  }
```

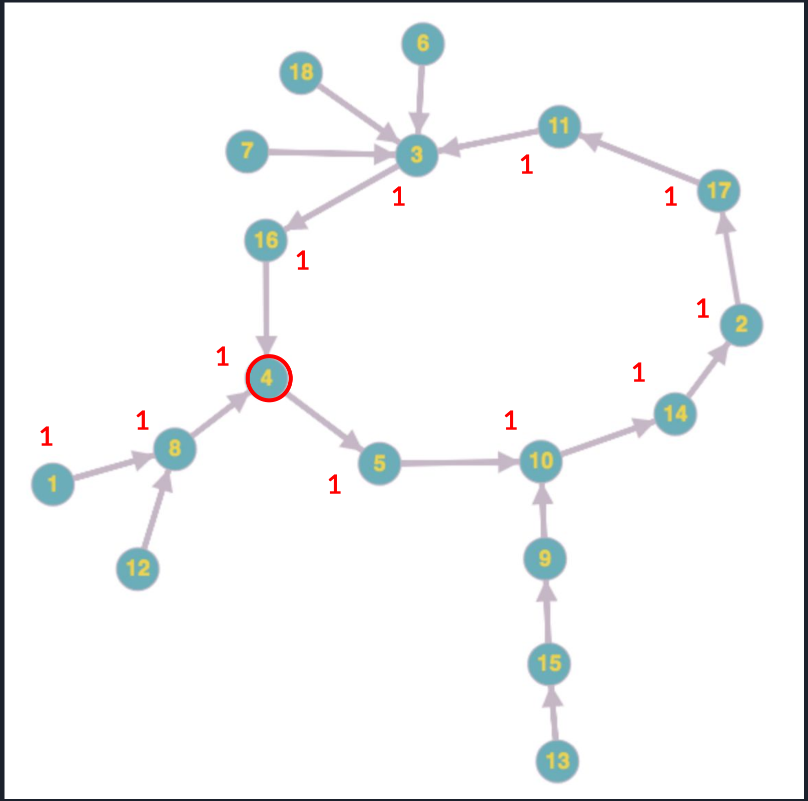


Functional Graphs

Primeira Estratégia - Whiles

Agora caso a marcação do vértice que você chegou seja 1, significa que você acabou de chegar em um vértice que você estava calculando, assim sendo, achou um ciclo novo.

```
void achei(int v) {
  int vini = v;
  while(marc[v] == 0) {
    marc[v] = 1;
    v = pai[v];
  }

  if(marc[v] == 1) acheiCiclo(v);
```

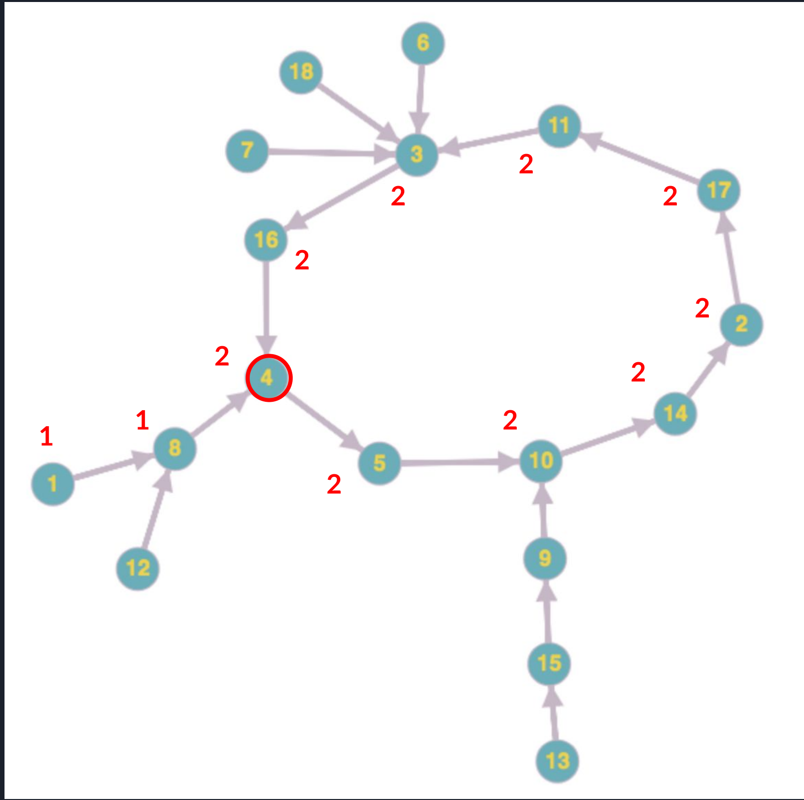


Functional Graphs

Primeira Estratégia - Whiles

Quando encontramos um ciclo novo, vamos marcando com 2 enquanto o vértice atual não estiver marcado com 2 (pois como é um ciclo, iremos voltar no início do ciclo).

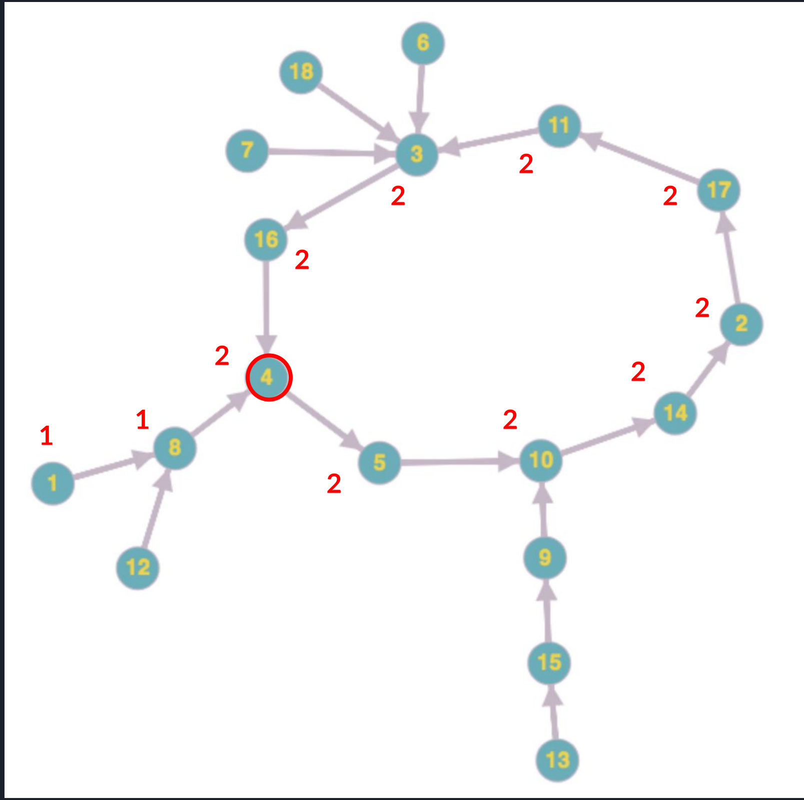
Neste processo vamos calculando tudo que precisamos para os vértices do ciclo, e para o ciclo novo



Functional Graphs

Primeira Estratégia - Whiles

```
void acheiCiclo(int v) {
    int idCiclo = ++qtdCiclos;
    int curId = 0;
    ini[idCiclo] = v;
    tam[idCiclo] = 0;
    ciclos[idCiclo].clear();
    while(marc[v] != 2) {
        marc[v] = 2;
        paiCiclo[v] = v;
        ciclo[v] = idCiclo;
        noCiclo[v] = 1;
        idNoCiclo[v] = curId;
        ciclos[idCiclo].push_back(v);
        tam[idCiclo]++;
        prof[v] = 0;

        v = pai[v];
        curId++;
    }
}
```



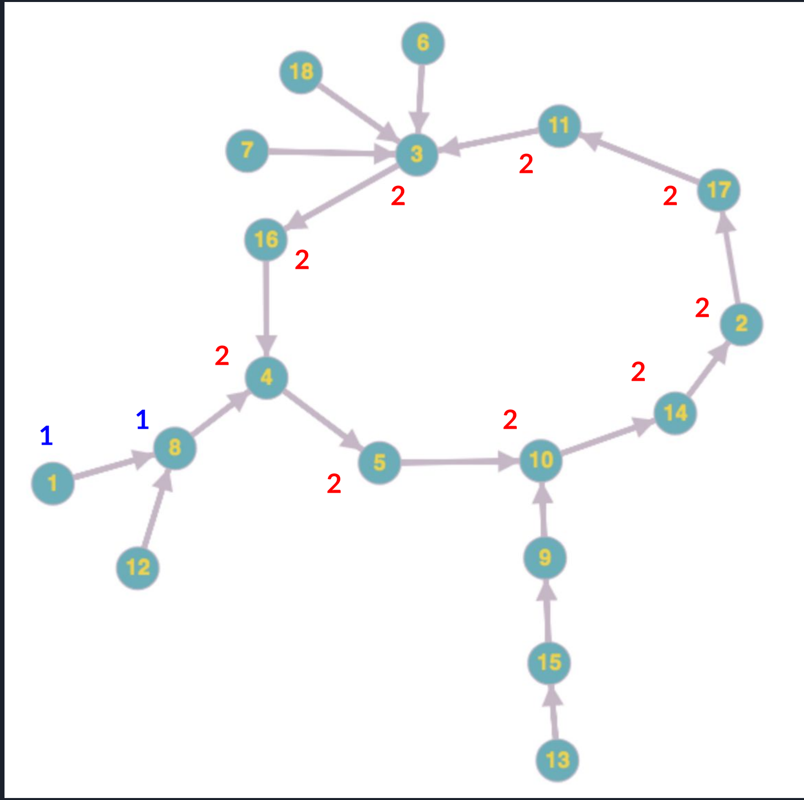
Functional Graphs

Primeira Estratégia - Whiles

Não podemos esquecer dos vértices que marcamos como 1.

Agora que já marcamos com 2 os vértices do ciclo, podemos voltar ao vértice inicial, e marcar todos os 1s como 3, pois temos certeza que eles não estão em nenhum ciclo.

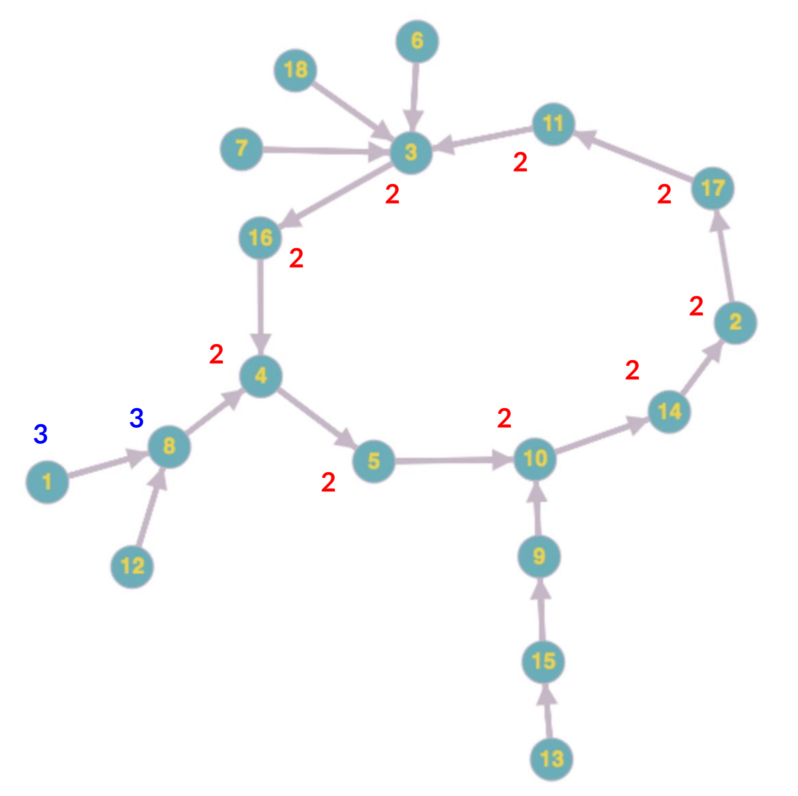
Como algumas informações dependem que o pai esteja calculado primeiro, teremos que salvar estes vértices num vector auxiliar e passar de trás pra frente calculando.



Functional Graphs

Primeira Estratégia - Whiles

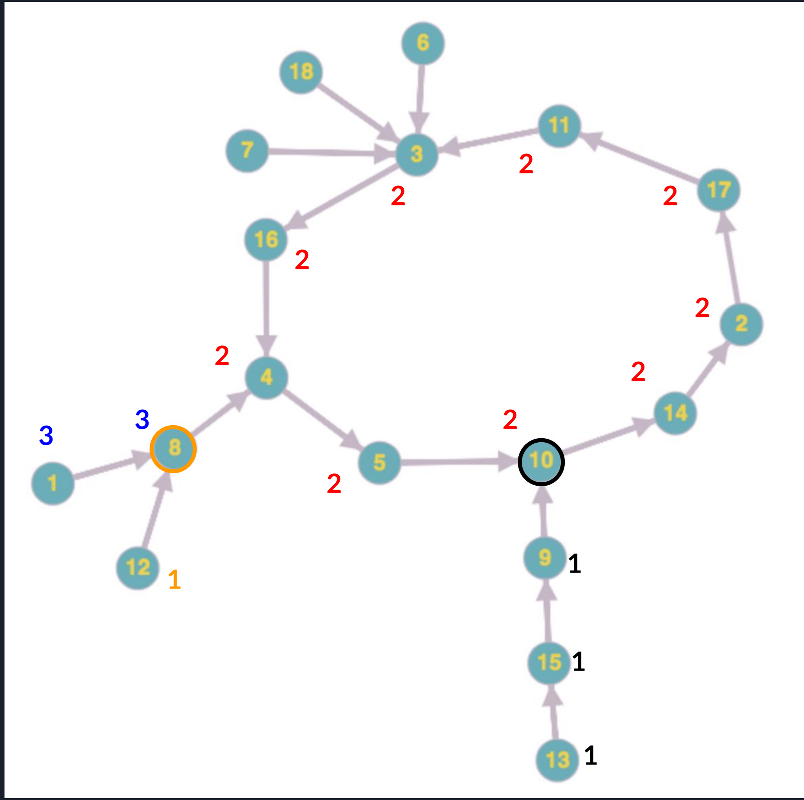
```
void acheiFora(int v) {
  vector<int> aux;
  while(marc[v] == 1) {
    marc[v] = 3;
    aux.push_back(v);

    v = pai[v];
  }

  for(int i = aux.size() - 1; i >= 0; i--) {
    int vcur = aux[i];
    int paicur = pai[vcur];

    paiCiclo[vcur] = paiCiclo[paicur];
    ciclo[vcur] = ciclo[paicur];
    noCiclo[vcur] = 0;
    idNoCiclo[vcur] = -1;
    prof[vcur] = prof[paicur] + 1;
  }
}
```



Functional Graphs

Primeira Estratégia - Whiles

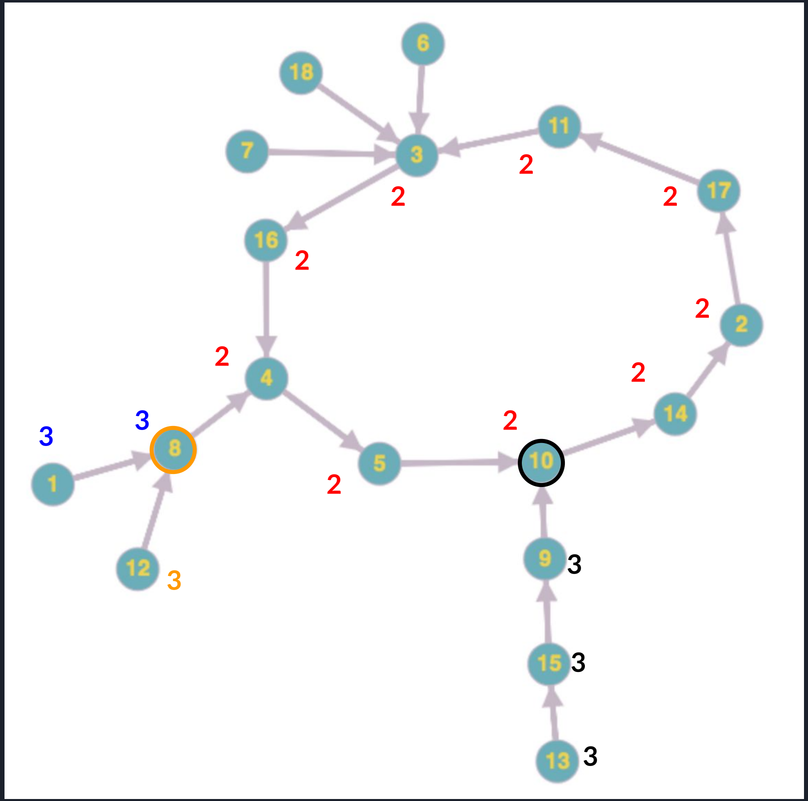
Tratamos o caso quando vamos marcando com 1 e chegamos num vértice marcado com 1. Os outros casos, usamos apenas a última parte de voltar ao vértice inicial e marcar com 3



Functional Graphs

Primeira Estratégia - Whiles

```
void achei(int v) {
  int vini = v;
  while(marc[v] == 0) {
    marc[v] = 1;
    v = pai[v];
  }

  if(marc[v] == 1) acheiCiclo(v);

  if(marc[vini] == 1) acheiFora(vini);
}
```



Functional Graphs

Segunda Estratégia - Lenhadora

Esta estratégia funciona bem nos dois casos, e também é boa para calcular informações da árvore que dependam dos filhos (por exemplo sub[v]).

Ela consiste em ir "cortando as folhas", até que não se tenha mais folhas e sobre apenas os ciclos.

Para isso manteremos uma fila de processamento das folhas, e para processar uma folha (simular cortá-la) basta subtrair 1 do ingrau do pai desta folha, e sempre que o ingrau de algum vértice virar zero colocamos nesta fila de processamento

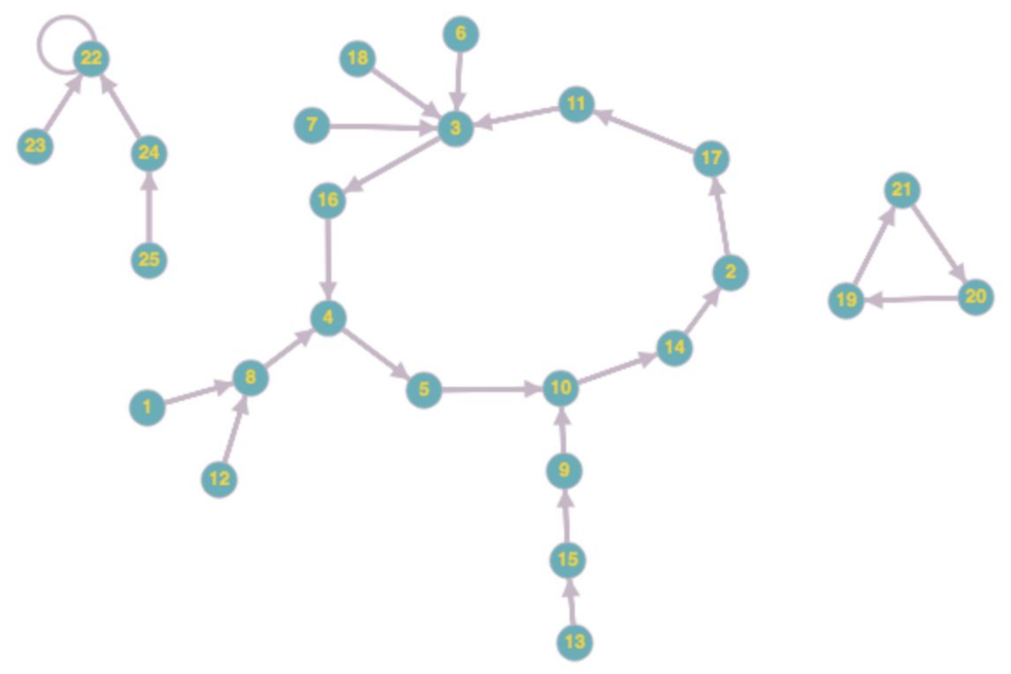
Note que durante esse processo passamos por todos os filhos antes de passar por um vértice, e portanto podemos calcular informações que dependam dos filhos, como por exemplo sub[v]

Functional Graphs

Segunda Estratégia - Lenhadora

Fila de Processamento das Folhas:

1	23	13	15	6	12	18	7	25	9	8	24



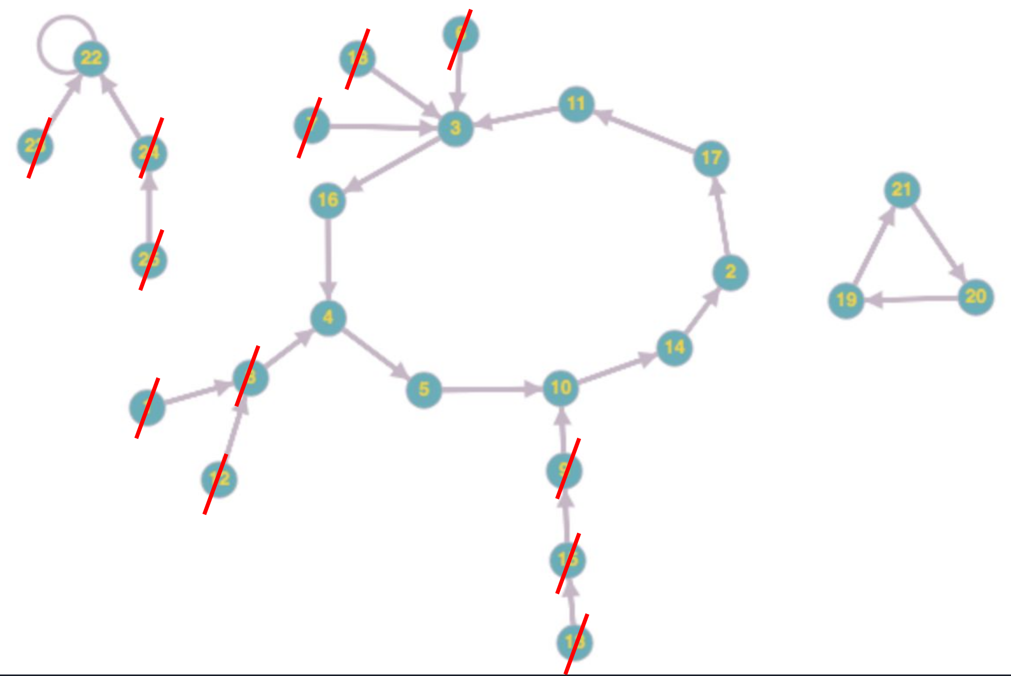
Functional Graphs

Segunda Estratégia - Lenhadora

Fila de Processamento das Folhas:

1	23	13	15	6	12	18	7	25	9	8	24

```cpp
void lenhadora() {
  queue<int> fila;
  for(int i = 1; i <= n; i++)
    if(ingrau[i] == 0) fila.push(i), marc[i] = 1;

  while(!fila.empty()) {
    int cur = fila.front(); fila.pop();
    processados.push_back(cur);
    sub[cur]++;

    int paicur = pai[cur];
    ingrau[paicur]--;
    sub[paicur] += sub[cur];
    if(ingrau[paicur] == 0)
      fila.push(paicur), marc[paicur] = 1;
  }
```



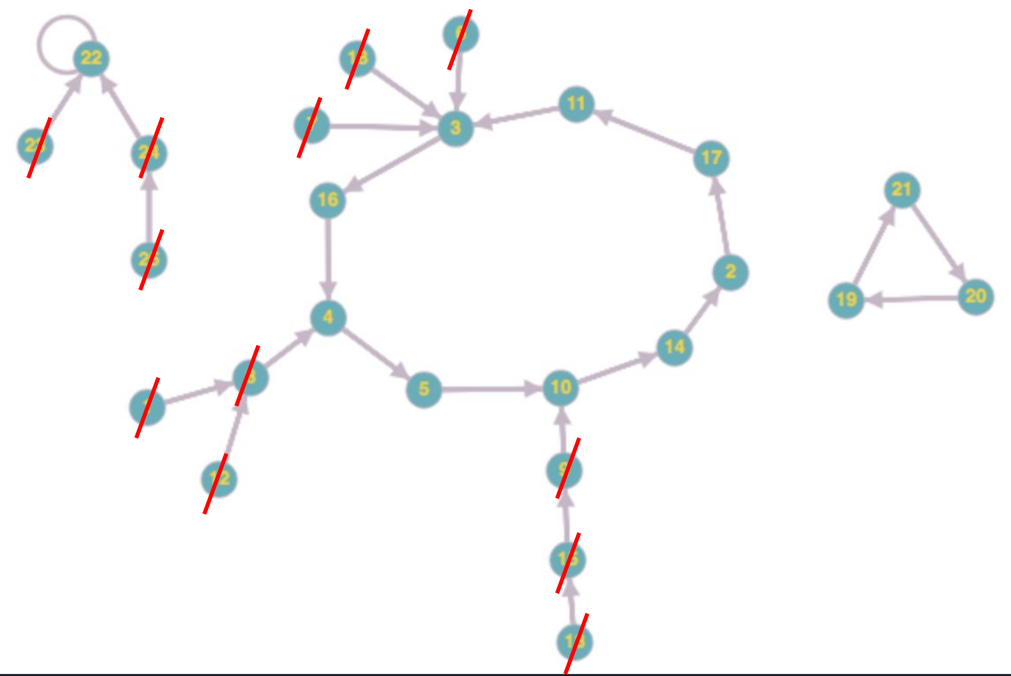
Functional Graphs

Segunda Estratégia - Lenhadora

Após processar todas as folhas, sobram apenas os ciclos

No caso direcionado podemos usar praticamente a mesma função acheiCiclo(v) nos vértices não marcados. A única mudança é que temos que somar 1 no sub[v] de cada vértice do ciclo, para contá-lo.

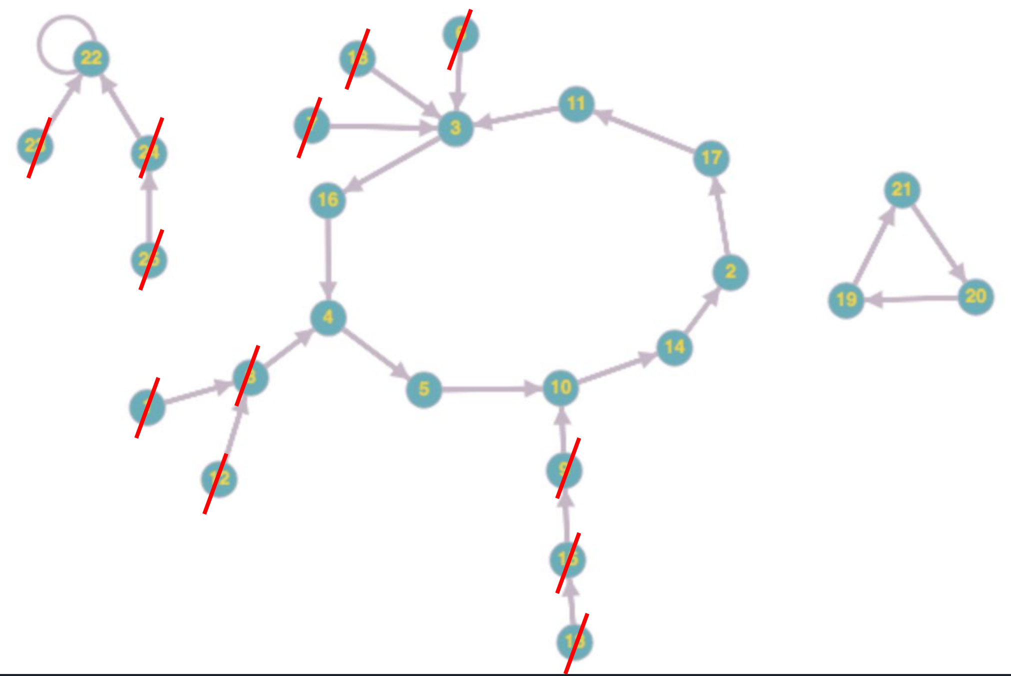
```
qtdCiclos = 0;
for(int i = 1; i <= n; i++)
  if(marc[i] == 0) acheiCiclo(i);
```



Functional Graphs

Segunda Estratégia - Lenhadora

```
void acheiCiclo(int v) {
    int idCiclo = ++qtdCiclos;
    int curId = 0;
    ini[idCiclo] = v;
    tam[idCiclo] = 0;
    ciclos[idCiclo].clear();
    while(marc[v] == 0) {
        marc[v] = 1;
        paiCiclo[v] = v;
        ciclo[v] = idCiclo;
        noCiclo[v] = 1;
        idNoCiclo[v] = curId;
        ciclos[idCiclo].push_back(v);
        tam[idCiclo]++;
        prof[v] = 0;
        sub[v]++;

        v = pai[v];
        curId++;
    }
}
```

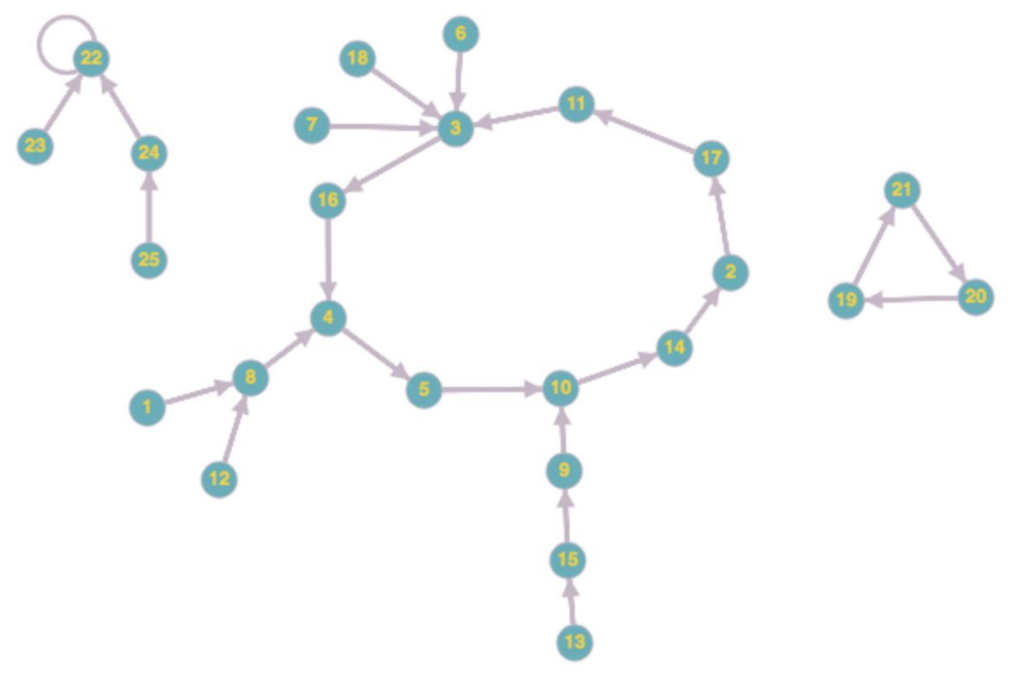


Functional Graphs

Segunda Estratégia - Lenhadora

Por fim temos ainda que calcular as informações que dependem do pai para os vértices fora dos ciclos

Para isso basta passar na ordem contrária à da fila de processamento e calcular, pois desta forma garantiremos que o pai de v foi computado antes de v.



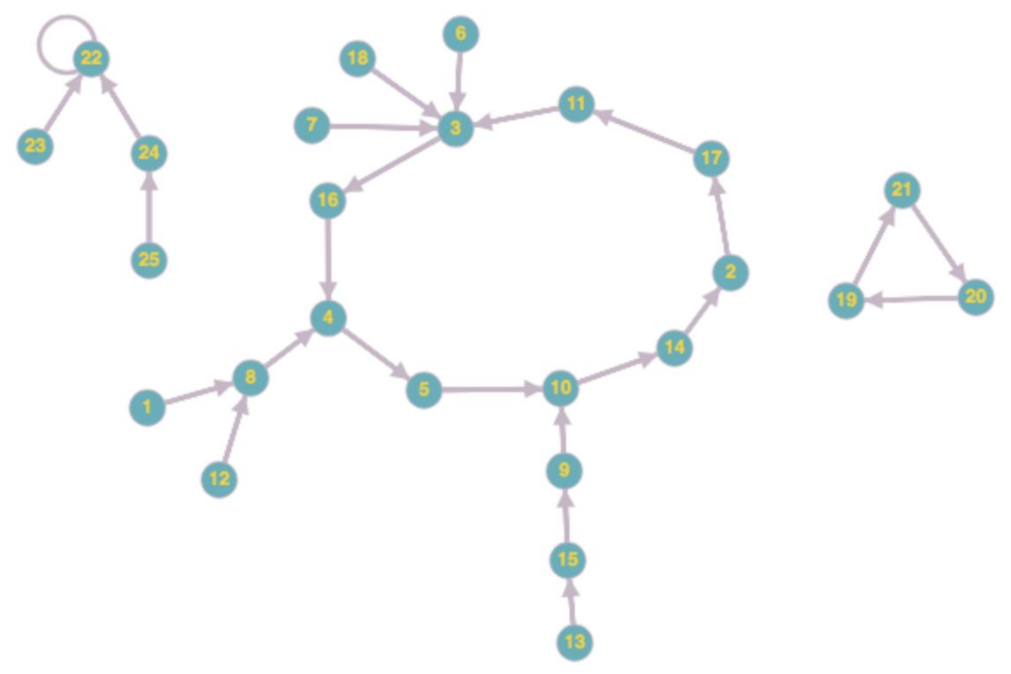
Fila de Processamento das Folhas:

1	23	13	15	6	12	18	7	25	9	8	24

Functional Graphs

Segunda Estratégia - Lenhadora

```
void lenhadora() {
  queue<int> fila;
  for(int i = 1; i <= n; i++)
    if(ingrau[i] == 0) fila.push(i), marc[i] = 1;

  while(!fila.empty()) {
    int cur = fila.front(); fila.pop();
    processados.push_back(cur);
    sub[cur]++;

    int paicur = pai[cur];
    ingrau[paicur]--;
    sub[paicur] += sub[cur];
    if(ingrau[paicur] == 0)
      fila.push(paicur), marc[paicur] = 1;
  }

  qtdCiclos = 0;
  for(int i = 1; i <= n; i++)
    if(marc[i] == 0) acheiCiclo(i);

  for(int i = processados.size() - 1; i >= 0; i--) {
    int vcur = processados[i];
    int paicur = pai[vcur];

    paiCiclo[vcur] = paiCiclo[paicur];
    ciclo[vcur] = ciclo[paicur];
    noCiclo[vcur] = 0;
    idNoCiclo[vcur] = -1;
    prof[vcur] = prof[paicur] + 1;
  }
}
```



Functional Graphs

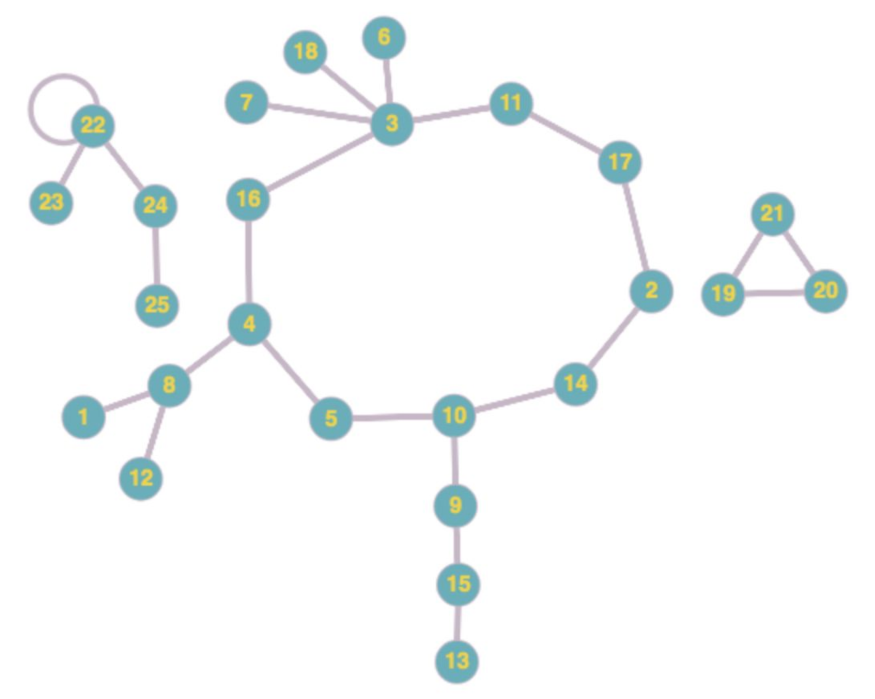
Segunda Estratégia - Lenhadora para Grafo não direcionado

Quando o grafo não é direcionado o pai[v] não é dado, e portanto temos que determiná-lo

Observe que durante o processo de cortar uma folha, apenas um de seus vizinhos não estará marcado, este será o pai.

Assim como durante o descobrimento de um ciclo só haverá um vizinho não marcado. Exceto para o primeiro do ciclo, que terá dois vizinhos, mas não importa qual dos dois escolhemos.

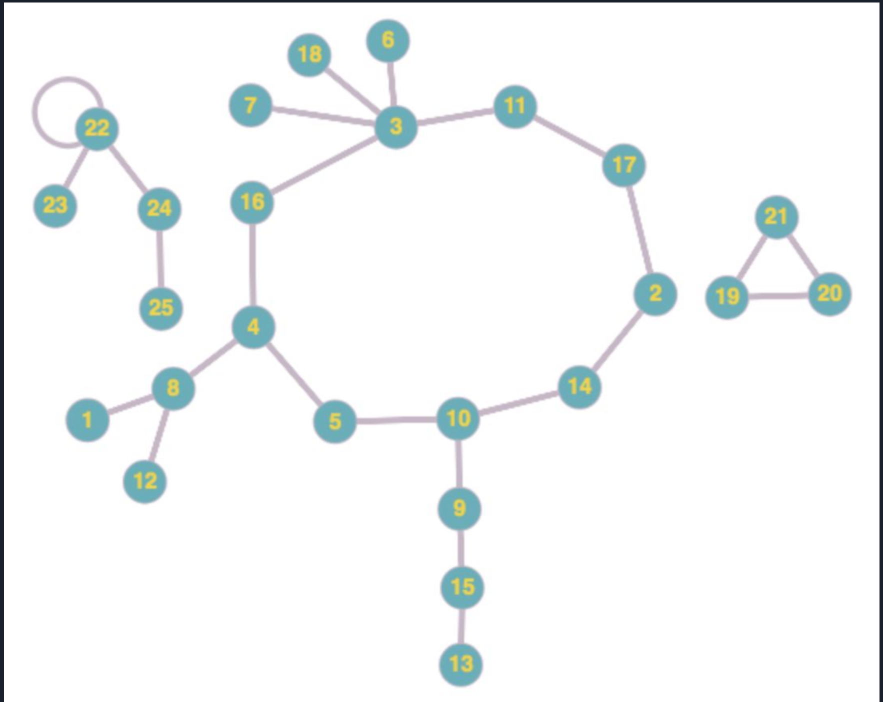
Portanto basta criarmos uma função que Encontra o pai, e fazer as alterações para usá-la



Functional Graphs

Segunda Estratégia - Lenhadora para Grafo não direcionado

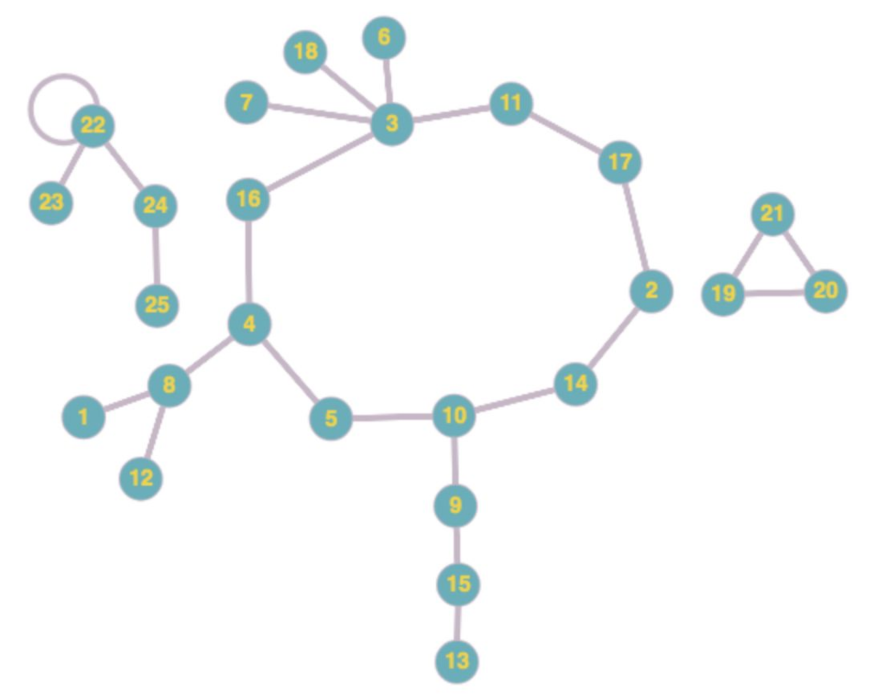
```
int achaPai(int v) {
  for(int i = 0; i < grafo[v].size(); i++) {
    int viz = grafo[v][i];
    if(marc[viz] == 0) return viz;
  }
  return -1;
}
```



Functional Graphs

Segunda Estratégia - Lenhadora para Grafo não direcionado

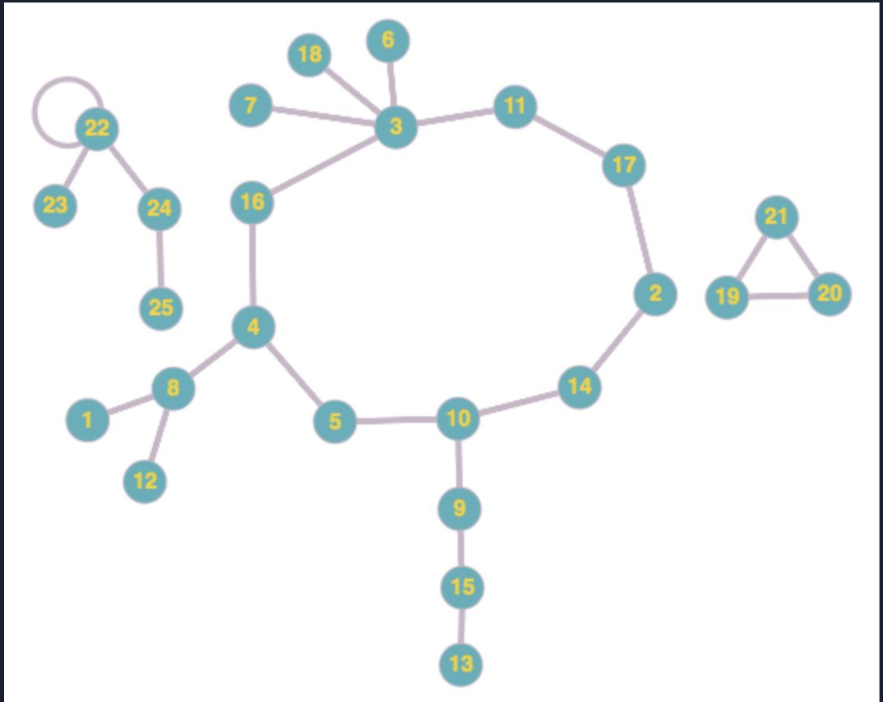
```
void lenhadora() {
  queue<int> fila;
  for(int i = 1; i <= n; i++)
    if(grau[i] == 1) fila.push(i), marc[i] = 1;

  while(!fila.empty()) {
    int cur = fila.front(); fila.pop();
    processados.push_back(cur);
    sub[cur]++;

    int paicur = achaPai(cur);
    pai[cur] = paicur;
    grau[paicur]--;
    sub[paicur] += sub[cur];
    if(grau[paicur] == 1)
      fila.push(paicur), marc[paicur] = 1;
  }

  qtdCiclos = 0;
  for(int i = 1; i <= n; i++)
    if(marc[i] == 0) acheiCiclo(i);

  for(int i = processados.size() - 1; i >= 0; i--) {
    int vcur = processados[i];
    int paicur = pai[vcur];

    paiCiclo[vcur] = paiCiclo[paicur];
    ciclo[vcur] = ciclo[paicur];
    noCiclo[vcur] = 0;
    idNoCiclo[vcur] = -1;
    prof[vcur] = prof[paicur] + 1;
  }
}
```



Functional Graphs

Segunda Estratégia - Lenhadora para Grafo não direcionado

```cpp
void acheiCiclo(int v) {
  int vini = v;
  int idCiclo = ++qtdCiclos;
  int curId = 0;
  ini[idCiclo] = v;
  tam[idCiclo] = 0;
  ciclos[idCiclo].clear();
  while(marc[v] == 0) {
    marc[v] = 1;
    pai[v] = achaPai(v);
    if(pai[v] == -1) pai[v] = vini;
    paiCiclo[v] = v;
    ciclo[v] = idCiclo;
    noCiclo[v] = 1;
    idNoCiclo[v] = curId;
    ciclos[idCiclo].push_back(v);
    tam[idCiclo]++;
    prof[v] = 0;
    sub[v]++;

    v = pai[v];
    curId++;
  }
}
```



Functional Graphs

Lista de Exercícios

Caminhos do Reino **-** OBI2016 P1 Fase 2
Joining Couples - Final Brasileira 2012
Scheme - Round 22 Div2 E
BFF's - Codejam 2016 Round 1A
Amigo Secreto - Seletiva IOI 2018
Idempotent functions - VK Cup 2015 - Round 3
Analysis of Pathes in Functional Graph - Educational Round 15 E
Tropical Garden - IOI 2011
Islands - IOI 2008
Association for Cool Machineries - Asia Singapore Regional Contest 2015
Stable Sets - ITMO Programming Contest Summer School 2014

MinQueue

MinQueue

Definição

Estrutura que suporta

- Inserir elemento no final
- Remover elemento do início
- Obter o menor elemento ativo na estrutura

MinQueue

Versão 1 - Usando MinStacks

Vamos pensar primeiro como implementar uma MinStack

Estrutura que suporta

- Inserir elemento no topo
- Remover elemento do topo
- Obter o menor elemento ativo na estrutura

MinQueue

Versão 1 - Usando MinStacks

Basta fazer uma stack (pilha) e além de guardar o valor, guardar também o menor elemento dali para baixo:

Valor	Mínimo
1	1
2	2
7	5
5	5

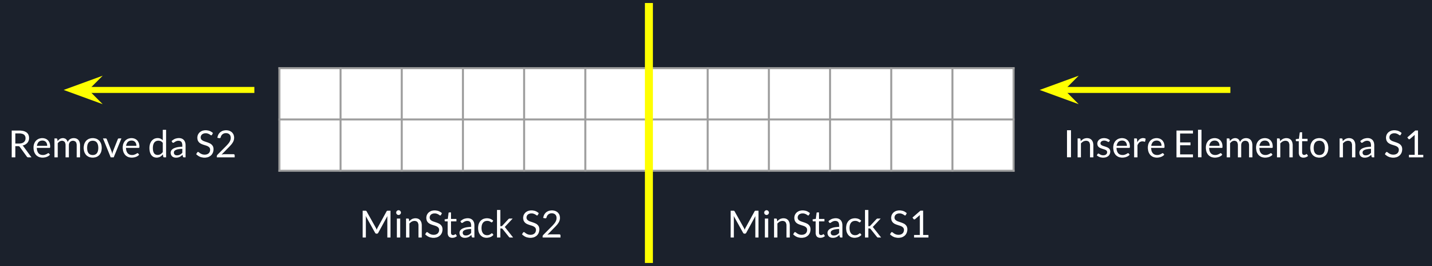
Insere Elemento:

Ao inserir um novo elemento calcula o mínimo entre ele e o mínimo no topo

MinQueue

Versão 1 - Usando MinStacks

Agora que sabemos construir uma MinStack, basta criar uma queue com 2 stacks



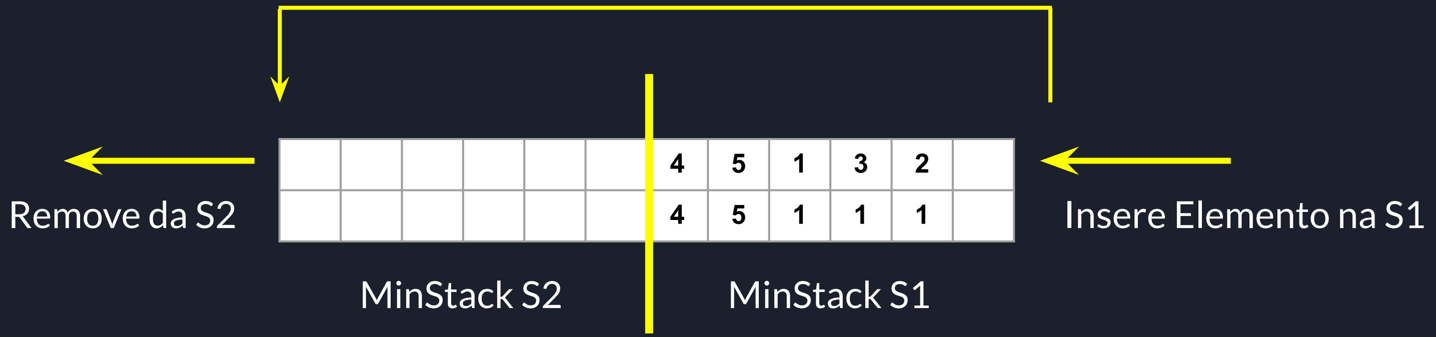
Mínimo: é o mínimo entre o mínimo das duas MinStacks

Mas e quando S2 estiver vazia ? Removermos de onde ?

MinQueue

Versão 1 - Usando MinStacks

Quando S2 estiver vazia, retira TODOS os elementos de S1 e coloca em S2, note que inverterá a ordem deles.

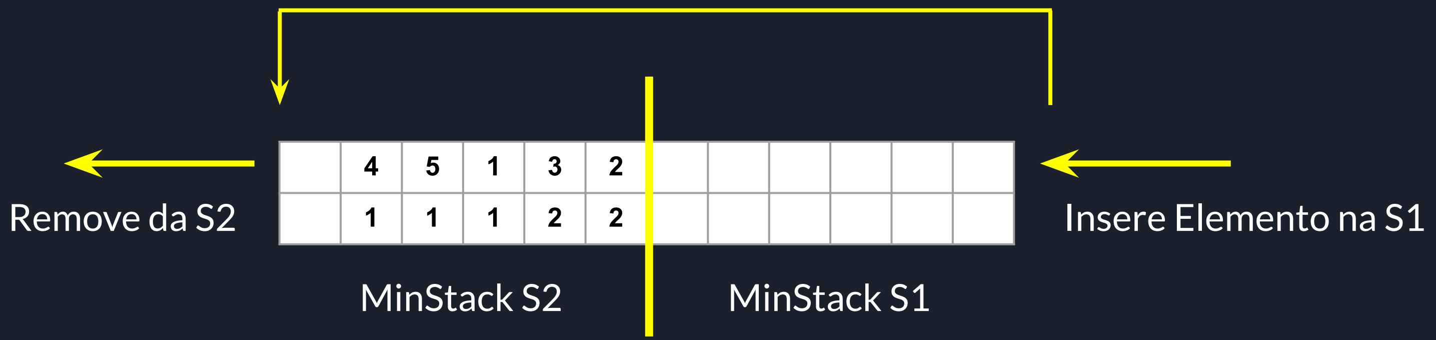


Mínimo: é o mínimo entre o mínimo das duas MinStacks

MinQueue

Versão 1 - Usando MinStacks

Quando S2 estiver vazia, retira TODOS os elementos de S1 e coloca em S2, note que inverterá a ordem deles.



Mínimo: é o mínimo entre o mínimo das duas MinStacks

MinQueue

Versão 1 - Usando MinStacks

Perceba que a complexidade é O(1) para todas as operações

Na verdade a operação de Mínimo é Amortizado O(1), ou seja, uma operação de mínimo é O(N) já que todos podem passar de S1 para S2, mas todas as operações de mínimo juntas também são O(N) pois cada elemento só pode ir de S1 para S2 uma única vez.

MinQueue

Versão 2 - Usando Deque

Outra forma de pensar é manter na estrutura somente os elementos que ainda podem ser mínimo

Ex.: 2 3 7 5

Quando o 5 entra na estrutura, o 7 nunca mais poderá ser o mínimo, pois o 5 aparece depois então será removido depois que o 7, então sempre que o 7 estiver na estrutura o 5 também estará, logo o 7 é inútil para o cálculo, e podemos removê-lo.

Portanto quando chega um elemento removemos todos os elementos maiores que ele e que estejam no final da fila. Note que precisamos de uma estrutura que suporte remover do fim também (além de remover do início por conta da instrução de remover), por isso usaremos a Deque (Double Ended Queue) que insere e remove tanto do início quanto do final.

MinQueue

Versão 2 - Usando Deque

Mas se simplesmente jogarmos os elementos fora, teremos um problema ao remover

Ex.: 2 3 7 5 1 9

Neste exemplo apenas os elementos 1 e 9 estarão na nossa estrutura (o 1 arrancou todos os anteriores quando entrou na estrutura). Se então após colocarmos todos os elementos, removermos um elemento da estrutura iremos remover o 1, quando na realidade deveríamos remover o elemento 2

Como consertar ?

MinQueue

Versão 2 - Usando Deque

Mas se simplesmente jogarmos os elementos fora, teremos um problema ao remover

Ex.: 2 3 7 5 1 9

Neste exemplo apenas os elementos 1 e 9 estarão na nossa estrutura (o 1 arrancou todos os anteriores quando entrou na estrutura). Se então após colocarmos todos os elementos, removermos um elemento da estrutura iremos remover o 1, quando na realidade deveríamos remover o elemento 2

Basta manter para cada elemento o índice em que ele entrou na estrutura, e manter dois índices informando quais índices estão ativos na estrutura.

Ex.:							
	valor	2	3	7	5	1	9
	Índice	1	2	3	4	5	6

|---------------|
inativos

ini = 1 e fim = 6

Ao remover vemos se o primeiro elemento tem índice igual ao ini, se sim removemos ele, e sempre aumentamos o ini

MinQueue

Versão 2 - Usando Deque

```
struct MinQueue {
  deque<pair<int, int> > d;
  int ini, fim;

  MinQueue() { ini = 1; fim = 0; }

  void push(int v) {
    while(!d.empty() && d.back().first >= v) d.pop_back();

    d.push_back(make_pair(v , ++fim));
  }

  void pop() {
    if(!d.empty() && d.front().second == ini++)
      d.pop_front();
  }

  int getMin() {
    return d.front().first ;
  }
};
```

MinQueue

Exercício

Pense em como fazer uma MinQueue que também suporte a seguinte operação:

- Add(val) - Some o valor val em todos os elementos ativos da Minqueue

Ex.: 2 3 7 5 1 9

Add(3)

5 6 10 8 4 12

Insere(7)

5 6 10 8 4 12 7

Remove() -> retorna 5

6 10 8 4 12 7

Add(2)

8 12 10 6 14 9

MinQueue

Exercício - Solução

Basta manter uma variável soma, dizendo quanto devemos somar em todos os elementos ativos na estrutura.

Mas isso dá problema com os novos elementos que chegarem depois dessa instrução de soma, pois podemos somar em elementos que não deveríamos.

Para lidar com este erro, basta ao invés de inserir o elemento v, inserir o elemento v - soma

MinQueue

Exercício - Solução

Ex.: 2 3 7 5 1 9 soma = 0
Add(3)
2 3 7 5 1 9 soma = 3
Insere(7)
2 3 7 5 1 9 4 soma = 3
Remove() -> retorna 2 + 3 = 5
3 7 5 1 9 4 soma = 3
Add(2)
3 7 5 1 9 4 soma = 5

MinQueue

Exercício - Solução (TEM UM ERRO, ENCONTRE)

```
struct MinQueue {
  deque<pair<int, int> > d;
  int ini, fim, soma;

  MinQueue() { ini = 1; fim = 0; soma = 0; }

  void push(int v) {
    while(!d.empty() && d.back().first >= v) d.pop_back();

    d.push_back(make_pair(v - soma , ++fim));
  }

  void pop() {
    if(!d.empty() && d.front().second == ini++)
      d.pop_front();
  }

  void add(int val) {
    soma += val;
  }

  int getMin() {
    return d.front().first + soma ;
  }
};
```

MinQueue

Exercício - Solução (TEM UM ERRO, ENCONTRE)

```
struct MinQueue {
  deque<pair<int, int> > d;
  int ini, fim, soma;

  MinQueue() { ini = 1; fim = 0; soma = 0; }

  void push(int v) {
    while(!d.empty() && d.back().first >= v) d.pop_back();

    d.push_back(make_pair(v - soma , ++fim));
  }

  void pop() {
    if(!d.empty() && d.front().second == ini++)
      d.pop_front();
  }

  void add(int val) {
    soma += val;
  }

  int getMin() {
    return d.front().first + soma ;
  }
};
```

MinQueue

Lista de Exercícios

Aquatic Surf
Sound
Pilots
Desk Ordering
Trous de loup
Little Bird