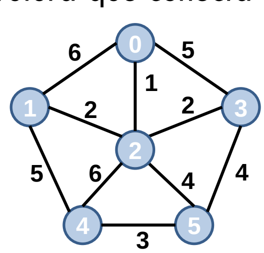
GRAFOS
ÁRVORE GERADORA MÍNIMA

Prof. André Backes

Árvore Geradora Mínima

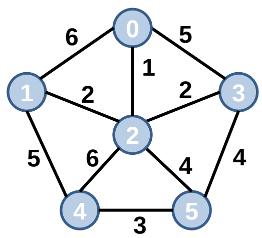
- Definição
  - Uma árvore geradora (do inglês, *spanning tree*) é um subgrafo que contenha todos os vértices do grafo original e um conjunto de arestas que permita conectar todos esses vértices na forma de uma árvore.
  - É a menor estrutura que conecta todos os vértices

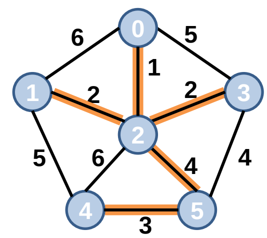


Árvore Geradora Mínima

3

- Dado um grafo ***G(V,A)***, a árvore geradora possui
  - todos os vértices ***V***
  - um total de arestas igual a **|V|-1** (o número de vértices menos um)

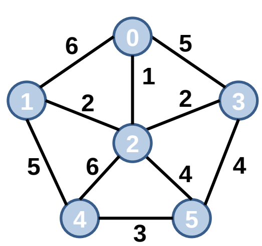


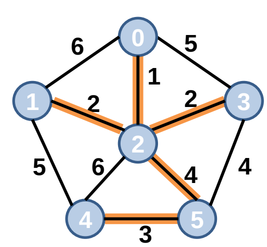


Árvore Geradora Mínima

4

- Se o grafo é ponderado (arestas com peso), podemos querer encontrar a árvore geradora mínima (do inglês, *minimum spanning tree*)
  - Procura o conjunto de arestas de menor custo





Árvore Geradora Mínima

5

- Condição para existir uma árvore geradora mínima
  - Para quaisquer dois vértices distintos, sempre deve existir um caminho que os une
  - Como todos os vértices estão conectados, calcular a árvore geradora não depende do vértice inicial
- Portanto, o grafo deve ser
  - Não-direcionado
  - Conexo
  - Ponderado

Árvore Geradora Mínima

6

- Aplicações
  - transporte aéreo: mapa de conexões de vôo
  - transporte terrestre: infra-estrutura das rodovias com o menor uso de material;
  - redes de computadores: conectar uma série de computadores com a menor quantidade de fibra ótica possível
  - redes elétricas e telefônicas: unir um conjunto de localidades com menor gasto
  - circuitos integrados
  - análise de clusters
  - armazenamento de informações

Árvore Geradora Mínima

- O problema pode ser resolvido usando uma estratégia gulosa que constrói a árvore incrementalmente
- Existem dois algoritmos clássicos para obter soluções ótimas
  - Algoritmo de Prim
  - Algoritmo de Kruskal
- A diferença entre eles está na regra usada para encontrar a aresta que fará parte da árvore

Algoritmo de Prim

- Funcionamento
  - Considera um vértice inicialmente na árvore
  - A cada iteração, o algoritmo procura a aresta de **menor peso** que conecte um vértice da árvore a outro que ainda não esteja na árvore.
  - Esse vértice é adicionado a árvore e o processo se repete.
  - Esse processo continua até que
    - Todos os vértices façam parte da árvore
    - Não se pode encontrar uma aresta que satisfaça essa condição (grafo desconexo)

Algoritmo de Prim

9

- Criando um grafo para teste

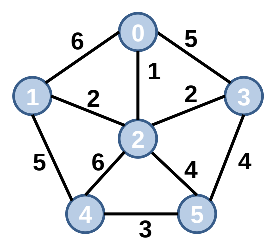
```c
#include <stdio.h>
#include <stdlib.h>
#include "Grafo.h"
int main(){
    int eh_digrafo = 0;
    Grafo* gr = cria_Grafo(6, 6, 1);
    insereAresta(gr, 0, 1, eh_digrafo, 6);
    insereAresta(gr, 0, 2, eh_digrafo, 1);
    insereAresta(gr, 0, 3, eh_digrafo, 5);
    insereAresta(gr, 1, 2, eh_digrafo, 2);
    insereAresta(gr, 1, 4, eh_digrafo, 5);
    insereAresta(gr, 2, 3, eh_digrafo, 2);
    insereAresta(gr, 2, 4, eh_digrafo, 6);
    insereAresta(gr, 2, 5, eh_digrafo, 4);
    insereAresta(gr, 3, 5, eh_digrafo, 4);
    insereAresta(gr, 4, 5, eh_digrafo, 3);

    int i, pai[6];

    arvoreGeradoraMinimaPRIM_Grafo(gr, 0, pai);

    libera_Grafo(gr);

    return 0;
}
```



Algoritmo de Prim

10

Vértices não tem pai, menos orig

Procura menor aresta ligando um vértice que está na árvore a outro fora da árvore

```c
void algPRIM(Grafo *gr, int orig, int *pai){
    int i, j, dest, primeiro, NV = gr->nro_vertices;
    double menorPeso;
    for(i=0; i < NV; i++)
      pai[i] = -1;// sem pai
    pai[orig] = orig;
    while(1){
        primeiro = 1;
        //percorre todos os vértices
        for(i=0; i < NV; i++){
            //achou vértices já visitado
            if(pai[i] != -1){
            // percorre os vizinhos do vértice visitado
                for(j=0; j<gr->grau[i]; j++){
                    //procurar menor peso: continua
                }
            }
        }
        if(primeiro == 1)
            break;

        pai[dest] = orig;
}
```

Algoritmo de Prim

11

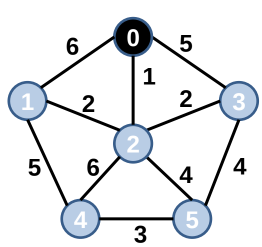
```c
//continuação
//achou vértice vizinho não visitado
if(pai[gr->arestas[i][j]] == -1){
    if(primeiro){//procura aresta de menor custo
        menorPeso = gr->pesos[i][j];
        orig = i;
        dest = gr->arestas[i][j];
        primeiro = 0;
    }else{
        if(menorPeso > gr->pesos[i][j]){
            menorPeso = gr->pesos[i][j];
            orig = i;
            dest = gr->arestas[i][j];
        }
    }
}
```

Algoritmo de Prim

12

- Passo a passo

Passo 1



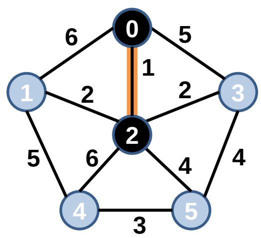
pai
0
-1
-1
-1
-1
-1

Inicia o cálculo com o vértice 0. Atribui seu próprio índice como pai. O restante dos vértices recebem pai igual a -1 (sem pai).

Algoritmo de Prim

- Passo a passo

Passo 2



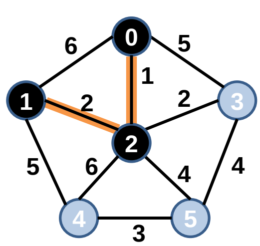
pai
0
-1
0
-1
-1
-1

Procura nos vértices com pai por um vértice sem pai e com menor peso: vértice 2.
Atribui vértice 0 como pai do vértice 2.

Algoritmo de Prim

- Passo a passo

Passo 3



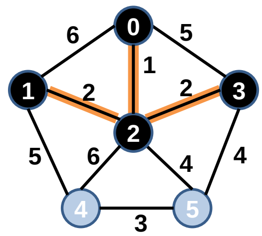
pai
0
2
0
-1
-1
-1

Procura nos vértices com pai por um vértice sem pai e com menor peso: vértice 1.
Atribui vértice 2 como pai do vértice 1.

Algoritmo de Prim

- Passo a passo

Passo 4



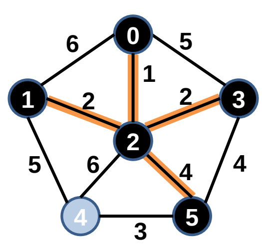
pai
0
2
0
2
-1
-1

Procura nos vértices com pai por um vértice sem pai e com menor peso: vértice 3.
Atribui vértice 2 como pai do vértice 3.

Algoritmo de Prim

- Passo a passo

Passo 5



pai
0
2
0
2
-1
2

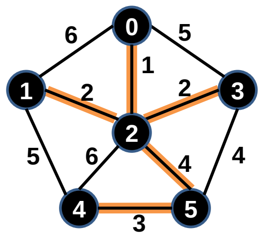
Procura nos vértices com pai por um vértice sem pai e com menor peso: vértice 5.
Atribui vértice 2 como pai do vértice 5.

Algoritmo de Prim

17

- Passo a passo

Passo 6



pai
0
2
0
2
5
2

Procura nos vértices com pai por um vértice sem pai e com menor peso : vértice 4.
Atribui vértice 5 como pai do vértice 4.
Fim do cálculo.

Algoritmo de Prim

18

- Complexidade
  - Considerando um grafo **G(V,A)**, onde **|V|** é o número de vértices e **|A|** é o número de arestas, a complexidade no pior caso é ***O(|V|*|A|)***. Como **|A|** é proporcional a **|V|**^{2}, seu custo é ***O(|V|^3)***

  - A eficiência depende da forma usada para procurar a aresta de menor peso. Usando uma fila de prioridade o custo pode ser reduzido para ***O(|A|log|V|)***

Algoritmo de Kruskal

19

- O algoritmo de Prim se inicia com um vértice e cresce uma única árvore a partir dele

- O algoritmo de Kruskal constrói uma floresta (várias árvores) ao longo do tempo, e que são unidas ao final do processo

Algoritmo de Kruskal

20

- Funcionamento
  - Considera cada vértice como uma árvore independente (floresta)
  - A cada iteração, o algoritmo procura a aresta de **menor peso** que conecta duas árvores diferentes
  - Os vértices das árvores selecionadas passam a fazer parte de uma mesma árvore
  - Esse processo continua até que
    - Todos os vértices façam parte da árvore
    - Não se pode encontrar uma aresta que satisfaça essa condição (grafo desconexo)

Algoritmo de Kruskal

- Criando um grafo para teste

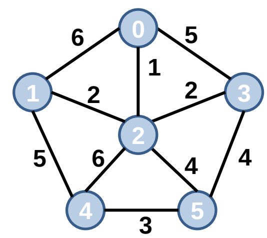
```c
#include <stdio.h>
#include <stdlib.h>
#include "Grafo.h"
int main(){
    int eh_digrafo = 0;
    Grafo* gr = cria_Grafo(6, 6, 1);
    insereAresta(gr, 0, 1, eh_digrafo, 6);
    insereAresta(gr, 0, 2, eh_digrafo, 1);
    insereAresta(gr, 0, 3, eh_digrafo, 5);
    insereAresta(gr, 1, 2, eh_digrafo, 2);
    insereAresta(gr, 1, 4, eh_digrafo, 5);
    insereAresta(gr, 2, 3, eh_digrafo, 2);
    insereAresta(gr, 2, 4, eh_digrafo, 6);
    insereAresta(gr, 2, 5, eh_digrafo, 4);
    insereAresta(gr, 3, 5, eh_digrafo, 4);
    insereAresta(gr, 4, 5, eh_digrafo, 3);

    int i, pai[6];

    arvoreGeradoraMinimaKruskal_Grafo(gr, 0, pai);

    libera_Grafo(gr);

    return 0;
}
```



Algoritmo de Kruskal

- Implementação

```c
void algKruskal(Grafo *gr, int orig, int *pai){
    int i,j,dest,primeiro,NV = gr->nro_vertices;
    double menorPeso;
    int *arv = (int*) malloc(NV * sizeof(int));
    for(i=0; i < NV; i++){
        arv[i] = i;
        pai[i] = -1;// sem pai
    }
    pai[orig] = orig;
    while(1){
        primeiro = 1;
        for(i=0; i < NV; i++){//percorre os vértices
            for(j=0; j<gr->grau[i]; j++){ //arestas
                //procura vértice menor peso: continua
            }
        }
        if(primeiro == 1) break;
        if(pai[orig] == -1) pai[orig] = dest;
        else pai[dest] = orig;

        for(i=0; i < NV; i++)
            if(arv[i] == arv[dest])
                arv[i] = arv[orig];
    }
    free(arv);
}
```

Cada vértice é uma árvore, sem pai

Procura menor aresta ligando árvores diferentes

Une as duas árvores da aresta selecionada

Algoritmo de Kruskal

23

- Implementação

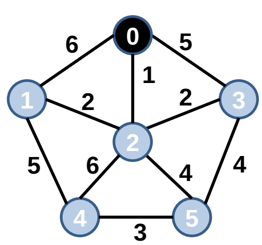
```
//continuação
//procura aresta de menor custo
if(arv[i] != arv[gr->arestas[i][j]]){
    if(primeiro){
        menorPeso = gr->pesos[i][j];
        orig = i;
        dest = gr->arestas[i][j];
        primeiro = 0;
    }else{
        if(menorPeso > gr->pesos[i][j]){
            menorPeso = gr->pesos[i][j];
            orig = i;
            dest = gr->arestas[i][j];
        }
    }
}
```

Algoritmo de Kruskal

24

- Passo a passo

Passo 1



pai	arv
0	0
-1	1
-1	2
-1	3
-1	4
-1	5

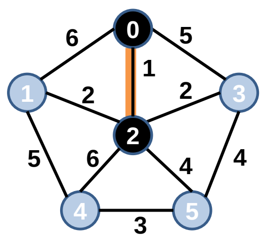
Inicia o cálculo com o vértice 0. Atribui seu próprio índice como pai. O restante dos vértice recebem pai igual a -1 (sem pai). Inicializa a árvore com o índice do vértice.

Algoritmo de Kruskal

25

- Passo a passo

Passo 2



pai	arv
0	0
-1	1
0	0
-1	3
-1	4
-1	5

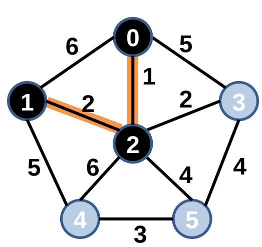
Procura a aresta com menor peso conectando vértices com árvores diferentes: vértices 0 e 2.
Atribui vértice 0 como pai do vértice 2.
Todos que possuem árvore igual ao vértice 2 passam a ter árvore igual ao vértice 0.

Algoritmo de Kruskal

26

- Passo a passo

Passo 3



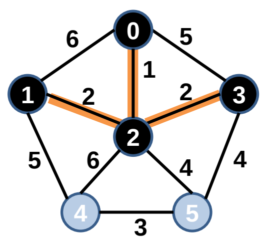
pai	arv
0	1
2	1
0	1
-1	3
-1	4
-1	5

Procura a aresta com menor peso conectando vértices com árvores diferentes: vértices 1 e 2.
Atribui vértice 2 como pai do vértice 1.
Todos que possuem árvore igual ao vértice 2 passam a ter árvore igual ao vértice 1.

Algoritmo de Kruskal

- Passo a passo

Passo 4



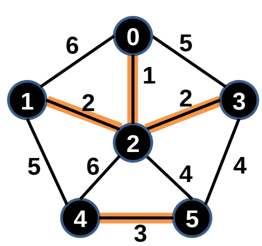
pai	arv
0	1
2	1
0	1
2	1
-1	4
-1	5

Procura a aresta com menor peso conectando vértices com árvores diferentes: vértices 2 e 3.
Atribui vértice 2 como pai do vértice 3.
Todos que possuem árvore igual ao vértice 3 passam a ter árvore igual ao vértice 2.

Algoritmo de Kruskal

- Passo a passo

Passo 5



pai	arv
0	1
2	1
0	1
2	1
5	4
-1	4

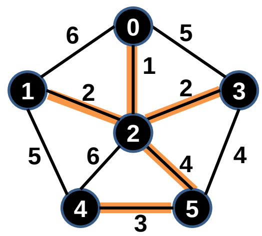
Procura a aresta com menor peso conectando vértices com árvores diferentes: vértices 4 e 5.
Atribui vértice 5 como pai do vértice 4.
Todos que possuem árvore igual ao vértice 5 passam a ter árvore igual ao vértice 4.

Algoritmo de Kruskal

29

- Passo a passo

Passo 6



pai	arv
0	1
2	1
0	1
2	1
5	1
2	1

Procura a aresta com menor peso conectando vértices com árvores diferentes: vértices 2 e 5.
Atribui vértice 2 como pai do vértice 5.
Todos que possuem árvore igual ao vértice 5 passam a ter árvore igual ao vértice 2.
Fim do cálculo

Algoritmo de Kruskal

30

- Complexidade
  - Considerando um grafo **G(V,A)**, onde **|V|** é o número de vértices e **|A|** é o número de arestas, a complexidade no pior caso é $O(|V|*|A|)$. Como **|A|** é proporcional a $|V|^2$, seu custo é $O(|V|^3)$

  - A eficiência depende da forma usada para procurar a aresta de menor peso. Usando uma estrutura de dados **união-busca** (*Union&Find*) o custo pode ser reduzido para $O(|A| \log |V|)$

Material Complementar

- Vídeo Aulas
  - Aula 112: Grafo - Árvore Geradora Mínima:
  - https://www.youtube.com/watch?v=eHC2tjQPX3A
  - Aula 113: Grafos – Algoritmo de Prim:
  - https://www.youtube.com/watch?v=bBq_Cu5doy0
  - Aula 114: Grafos – Algoritmo de Kruskal:
  - https://www.youtube.com/watch?v=EzMHc5xW6Pc