Técnicas de Programação Avançada

Árvores Geradoras Mínimas
PCV

Definição

- Árvore Geradora Mínima – Minimum Spanning Tree – MST (árvore espalhada mínima)

- Em um grafo não orientado ponderado, é um conjunto de arestas T cuja soma dos seus pesos $w(T) = \sum w(w,v)$, (u,v) E a T, é mínimo.
- Aplicações: Determinar conexões mínimas em sistemas com cabeamento.

Algoritmo Genérico

A ← {}

Enquanto A não formar uma MST faça

  Encontrar uma aresta (u,v) que seja segura para A

  Adicionar (u,v) ao conjunto A

fim enquanto

Retornar A

Definições

- **Corte**: é uma partição (S, V-S) do conjunto V de vértices de G.
- Uma aresta *e* **cruza** o corte quando cada um de seus vértices estiver de um lado do corte
- **Um corte respeita** um conjunto A de arestas se nenhuma dessas arestas cruza o corte.
- **Aresta leve** é uma aresta que cruza o corte e cujo peso é o menor dentre os pesos das arestas que cruzam o corte.

Determinando Arestas seguras

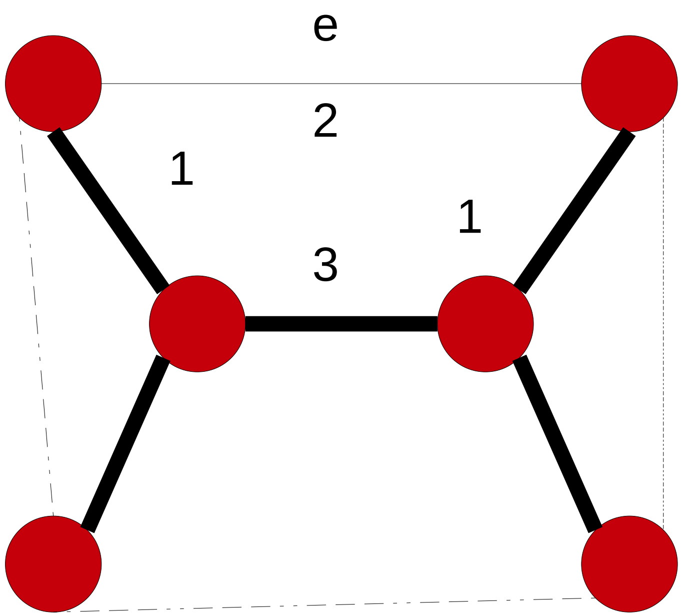
- Teorema: Se (S,V-S) é um corte que respeita o conjunto de arestas A, e (u,v) é uma aresta level cruzando (S,V-S), então (u,v) é uma aresta segura para A

Propriedade dos Ciclos

- Seja T uma árvore geradora de um grafo ponderado G
- Sejam e uma aresta de G fora de T e C um ciclo formado por e + T
- Se T é uma árvore geradora mínima, para cada aresta f de C, w(f) <= W(e)
- Se w(f) > w(e), é possível obter uma árvore geradora de menor peso total trocando-se f por e.

Propriedade do Ciclo

- No Exemplo ao lado, **e** não é máxima para o ciclo. Logo T não é uma árvore geradora mínima.



Propriedade da Partição

- Considere uma partição dos vértices de G em dois subconjuntos disjuntos U e V. Seja e a aresta de peso mínimo que conecta U e V (aresta leve).
- Então, existe uma árvore geradora mínima de G que contém e.

Algoritmo de Prim-Jarnik

- É um algoritmo parecido com o algoritmo de Dijkstra para encontrar menores caminhos.
- O algoritmo parte de um vértice arbitrário pertencente ao grafo e gera a MST:
  1. Escolhe um vértice para iniciar a MST
  2. Encontra a aresta mínima dentre as arestas que ligam a MST ao restante do grafo e adiciona o vértice oposto (que ainda não faz parte da MST) a ela.
  3. Repete o passo 2 até não haver mais vértices fora da MST.

Algoritmo de Prim-Jarnik

```
Prim(G,r, w)
 para cada vértice u de G faça
     custo[u] ← inf
     p[u] ← nil
 fim para
 custo[r]=0
  Q ← V[G]
 enquanto (Q.vazia == FALSO) faça
   u ← removeMenor(Q)
  para cada v adjacente a u faça
     Se v E Q e w(u,v) < custo[v]
      então p[v] = u
         custo[v] =w[u,v]
    fim se
  fim para
fim enquanto
```

Algoritmo de Prim-Jarnik

- Q é uma fila de prioridade onde são removidos sempre os vértices de menor custo.
- O custo de um vértice é o peso da menor aresta que liga o vértice à MST.
- Inicialmente o custo é infinito para todos os vértices exceto para o vértice escolhido para iniciar a MST que possuirá custo 0.
- O vetor p indica a qual vértice da MST um vértice novato vai ser ligado quando este for adicionado à MST.
- Neste algoritmo a partição fica mantida entre o vetor p e a fila Q.

Algoritmo de Kruskal

Criar um grafo T formado por todos os vértices de G e nem uma aresta. Ou seja uma floresta com V árvores cada uma com um vértice.

Criar uma lista ordenada em ordem não decrescente com todas as arestas de G.

Para cada aresta (u,v) tomadas respeitando a ordem da lista faça

  Se u e v não estão conectadas por um caminho em T, inserir (u,v) em T

Fim para

Algoritmo de Kruskal

- O conjunto A no algoritmo de Kruskal é uma floresta.
- O algoritmo encontra arestas seguras procurando por arestas mínimas que conectam duas árvores quaisquer da floresta sem criar ciclos.
- O algoritmo mantém estruturas de conjuntos disjuntos para guardar cada árvore da floresta atual

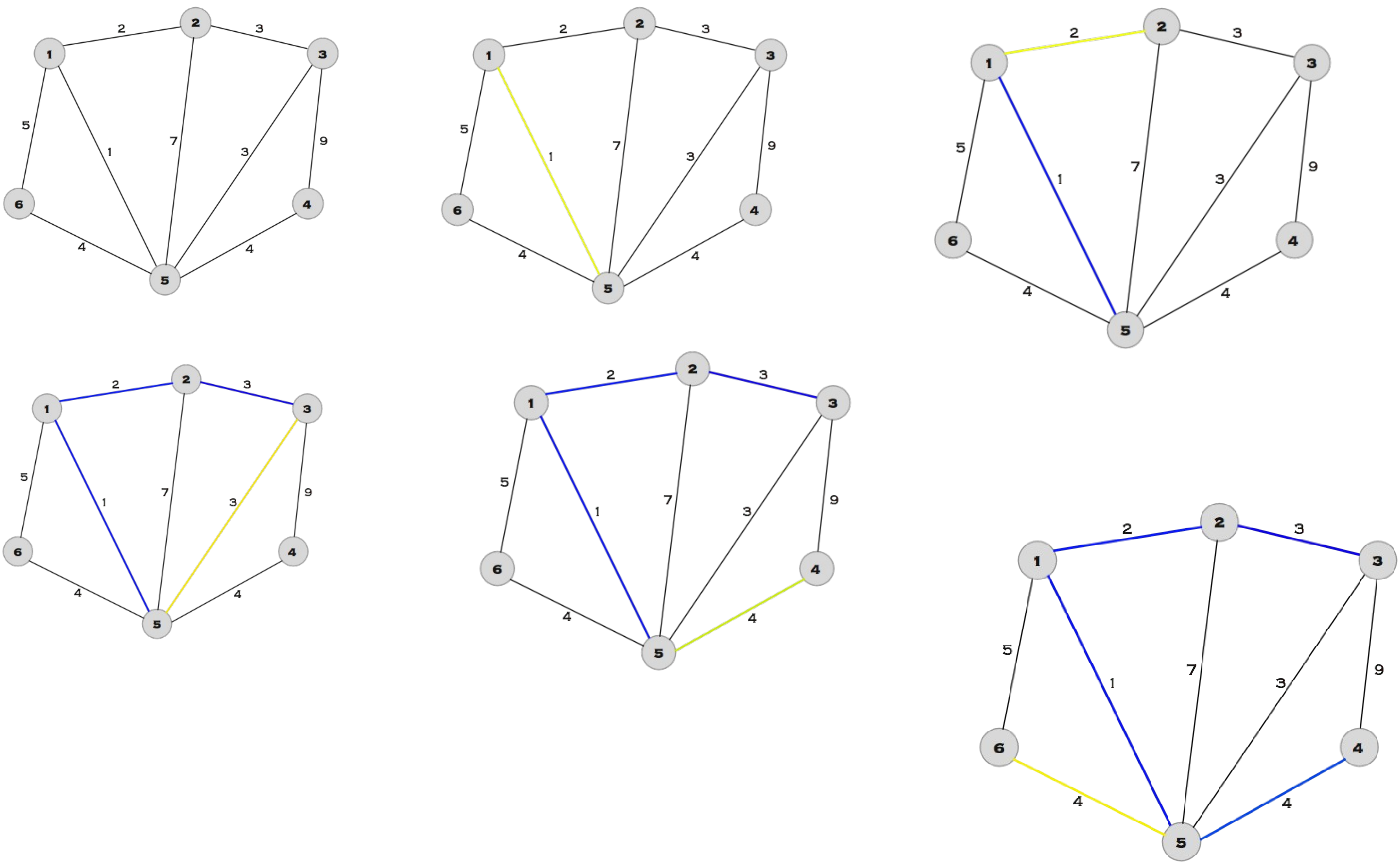
MST - KRUSKAL

- Obtém uma AGM adicionando uma aresta de cada vez à floresta e, a cada passo, usa a aresta de menor peso que não forma ciclo.

- Inicia com uma floresta de |V | árvores de um vértice: em |V | passos, une duas árvores até que exista apenas uma árvore na floresta.

-

Kruskal - Exemplo



Algoritmo MST-KRUSKAL

```
mst-kruskal(G,w)
A ← {}
para cada vértice v E V faça
  make-set(v)
ordenar as arestas de E pelo peso w  em ordem crescente
para cada aresta (u,v) de E em ordem crescente de peso
faça
  se find-set(u) <> find-set(v) então
    A ← A U {(u,v)}
  Union (u,v)
```

Problema do Caixeiro Viajante

- *Dados um conjunto de cidades c1..cn e uma conexão direta com distância d(ci,cj) para cada par de cidades, encontrar o trajeto para visitar todas as cidades, passando uma única vez em cada cidade, cujo comprimento total seja o menor possível.*
- *Ou, Dado um grafo ponderado completo G, encontrar um circuito hamiltoniano para G cujo custo total seja mínimo.*
- *Conhecido como PCV ou em inglês TSP (Travelling Salesman problem)*
- *Problema NP-Completo. Não existe uma solução conhecida para o mesmo que o resolva em tempo polinomial. A função de complexidade de tempo do algoritmo ótimo conhecido é exponencial.*

Problema do Caixeiro Viajante

- Aplicações
  - o problema do Caixeiro viajante está muito relacionado com aplicações de transporte
  - Mas sua aplicação mais importante é na otimização de sequências de operações de manufatura.
    - exemplo: Considere um braço de robô cuja função é soldar todas as conexões de uma placa de circuito impresso. O menor roteiro que visita todos os pontos de solda exatamente uma única vez define o roteiro mais eficiente para o robô.

Problema do Caixeiro Viajante

- Um algoritmo que forneça uma solução ótima para este problema deve percorrer todas as possíveis soluções afim de verificar qual a solução ótima, que é um percurso de menor custo possível.
- Para tanto, o algoritmo deve montar todos os possíveis caminhos a partir de uma dada cidade passando por todas as cidades uma única vez e retornando a cidade de origem, e computar seus custos.
- O menor custo vai sendo armazenado e comparado com o custo de cada novo percurso encontrado.

Problema do Caixeiro Viajante

- O algoritmo de busca em profundidade em grafos (Depth First Search -DFS), pode pode ser adaptado para determinar uma solução ótima do PCV.
- O DFS utiliza o esquema de cores para determinar vertices não visitados (brancos), visitados (cinzas) e com todos os adjacente visitados (pretos)
- Avaliando o comportamento do algoritmo DFS, verificamos que um vértice qualquer r, pintado de preto não será mais alcançado em uma busca através de um outro vértice v da lista de adjancência de um vértice qualquer s. Assim este algoritmo vai eliminando vértices alcançaveis e sua complexidade de tempo fica = O(V+E).

Problema do Caixeiro Viajante

- Para fazer com que DFS visite um vértice t, quantas vezes este for alcançado a partir de um vértice inicial s, passando por vértices intermediários diferentes, basta que ao término da lista de adjacências de cada vértice, estes passem a ter novamente a cor branca.
- Com esta alteração, se o mesmo vértice t, for adjacente a um outro vértice que não seja o seu predecessor atual, ele será novamente visitado.

Problema do Caixeiro Viajante

- Outra adaptação necessária é fazer com que o algoritmo compute o custo do percurso construído, verifique quando um novo percurso foi concluído e compare o custo deste novo percurso com o custo do menor percurso obtido até então.
- Além disso, para computar o percurso, uma cidade de partida é definida, dado que o percurso é um ciclo deve haver uma conexão entre a ultima cidade do percurso e a primeira.
- Como se trata de um ciclo, a partida poderá se dar de qualquer outra cidade dentro do percurso encontrado.

TSP – - Algoritmo

Algoritmo DFS(G: grafo,c)
  para cada vértice u de G faça
    cor[u]🡐 branco
    pred[u]=🡐-1
  fim para

  nVisitas ← 1
  MenorRota ← {}
  Custo = 0
  MenorCusto ← Infinito
  Rota <--{}
  visita(u)

TSP - Algoritmo

```
Visita(u: vertice)
   cor[u] ← cinza
   rota   ← rota U u
   Para cada v adjacente a u faça
      Se (cor[v] ! = branco) então
          nVisitas ← nVisitas+1
          custo=custo+peso(u,v)
          pred[v]←u
          visita(v)
     Fim se
  Fim Para
  Cor[u] ←  BRANCO
     Se visitas = |V|
        custo ← custo + peso(u,ci)
        Se custo< menorCusto
             MenorCusto ← custo
             MenorRota ← Rota
       Fim se
      custo ← custo – peso(u,ci)
   Fim se
   custo ← custo - peso(Pred[u],u)
   rota   ←  rota – u
   visitas ←  vistas -1

 fim Visita
```