DCC206 – Algoritmos 1

Aula 06 – Árvores geradoras mínimas – Parte 2

Professor Renato Vimieiro

Introdução

- Na aula anterior, discutimos o conceito de árvore geradora mínima e suas aplicações
- Vimos um algoritmo genérico baseado para computar a MST
- Essencialmente, o algoritmo genérico determinava que, a cada iteração, uma aresta segura deveria ser incluída na árvore em construção
- Estudamos o conceito de aresta segura e demonstramos que arestas de menor custo do cut-set de um corte do grafo pertencem a MST

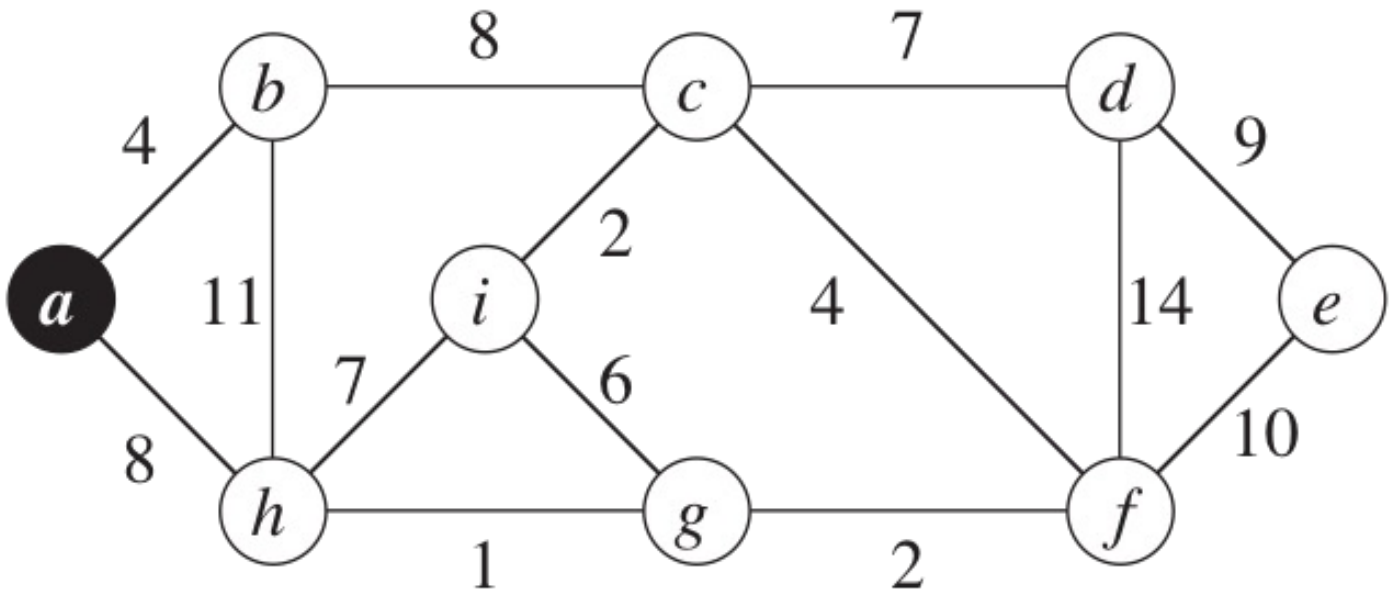
Introdução

- O algoritmo de Prim explorou essa definição expandindo a árvore na direção dos vértices “adjacentes de menor custo”
- Dessa forma, o algoritmo sempre considerava o corte como sendo a árvore atual e os demais vértices
- Na aula de hoje, vamos estudar outro método guloso com uma abordagem bastante diferente
- Estudaremos o algoritmo de Kruskal

Algoritmo de Kruskal

- O algoritmo de Kruskal considera que cada vértice do grafo forma, individualmente, uma árvore
  - Ou seja, ele considera que inicialmente temos uma floresta de árvores com um único vértice no grafo
- Em seguida, o objetivo do algoritmo é unir essas árvores disjuntas para formar um único componente conexo
- Para unir duas árvores disjuntas, escolhemos a aresta de menor custo ligando as duas
- Em seguida, atualizamos a informação de que os vértices de ambas pertencem à mesma árvore, e repetimos o processo

Algoritmo de Kruskal



Implementação do algoritmo de Kruskal

- Intuitivamente, o algoritmo de Kruskal parece ser bastante simples e fácil de implementar
- Contudo, para obter uma implementação eficiente do algoritmo, precisamos ter cuidado com a escolha das estruturas utilizadas
- A ideia do algoritmo é:
  - Criar conjuntos individuais para cada vértice
  - Enquanto número de árvores maior que 1
    - Escolher a aresta de menor custo ligando duas árvores distintas
    - Unir as árvores

Implementação do algoritmo de Kruskal

- O livro apresenta o seguinte pseudocódigo

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```

Implementação do algoritmo de Kruskal

- Veja que o custo do algoritmo é definido por
  - Ordenação das arestas
  - Verificação se os terminais de uma aresta pertencem a duas árvores distintas
  - União de duas árvores ligadas por uma aresta
- Retomaremos a discussão do custo do algoritmo posteriormente
- Agora vamos concentrar na escolha da estrutura de dados que dê suporte para as operações do algoritmo
- Precisamos de uma estrutura que:
  - Permita a criação de n conjuntos disjuntos (**make-set**)
  - Encontre o conjunto ao qual um elemento pertence (**find-set**). Em geral, retornamos somente um **representante** desse conjunto.
  - Una dois conjuntos disjuntos (**union**)

Estrutura de dados para conjuntos disjuntos

- Essa estrutura é chamada **estrutura para conjuntos disjuntos (união-busca)**
- Vamos impor certas restrições às operações que podem ser realizadas
- O universo de elementos é estático
  - No início, existem n elementos no universo e esses continuam ao longo da execução
- Os conjuntos armazenados na estrutura são disjuntos
- Dois conjuntos armazenados na estrutura podem ser unidos, mas não separados
  - Se dois conjuntos A e B (disjuntos) são unidos para formar o conjunto C, esse não poderá ser dividido em dois conjuntos disjuntos posteriormente
  - O mesmo se aplica para todo conjunto disjunto armazenado

Estrutura de dados para conjuntos disjuntos

- Temos duas abordagens principais para implementar essa estrutura de dados
  - **Quick-find**: prioriza o tempo de recuperação do conjunto ao qual um elemento pertence
  - **Quick-union**: prioriza o tempo para unir dois conjuntos disjuntos
- Vamos discutir ambas as implementações e analisar as vantagens e desvantagens de cada

Quick-find

- A implementação da abordagem quick-find é a mais simples das duas
- Podemos armazenar os conjuntos em uma lista encadeada
- Inicialmente, cada elemento é associado a uma lista que contém somente ele
  - Armazenamos um vetor de ponteiros para associar os elementos às suas respectivas listas
- O custo para criação dos conjuntos é O(n). Cada lista requer O(1) para ser criada.
- O custo para a operação find-set é O(1). Basta retornar o primeiro elemento da lista.
- O custo da união pode ser bastante caro

Quick-find

- Como possuímos um total de n elementos, podemos realizar, no máximo, n-1 operações de união
- A união de dois conjuntos requer a atualização dos ponteiros de cada elemento de um dos conjuntos
- Assim, podemos terminar com um custo $\Theta(n^2)$
- Uma heurística bastante comum para melhorar esse custo é fazer a **união ponderada**
  - Nesse caso, sempre uniremos a lista (conjunto) de menor tamanho à maior
- Veremos que o custo individual de uma operação de união ainda pode ser alto nessa solução, mas ele é amortizado ao longo das várias chamadas

Quick-find

- **Teorema**: Usar uma estrutura para conjuntos disjuntos com união ponderada com m operações make-set, find-set e union, sendo n o número de elementos possui custo O(m + n log n).
- **Prova**: Como possuímos n elementos, o número de uniões é limitado a n-1. Vamos considerar as atualizações relacionadas a um elemento x. Quando x é atualizado a primeira vez, temos como resultado um conjunto com, no mínimo, 2 elementos. Na segunda vez, terminamos com um conjunto com, no mínimo, 4 elementos. Na verdade, é fácil perceber que, a cada atualização, o conjunto resultante ao qual x pertence terá pelo menos o dobro de elementos do conjunto anterior de x. Ou seja, após i atualizações de x, sabemos que o conjunto resultante terá $2^i \leq n$ elementos. Portanto, esse elemento será atualizado, no máximo, log n vezes. Como são executadas, no máximo, n-1 uniões, o custo total das operações union é O(n log n). Dado que existem O(m) operações make-set e find-set, o custo total é O(m + n log n).

Quick-union

- A segunda alternativa de implementação é um pouco mais elaborada
- Vamos criar uma estrutura de ponteiros (tipo árvore) em que cada nó (elemento de um conjunto) aponta para seu pai (na árvore)
  - Caso o nó de um elemento seja a raiz da árvore, ele aponta para si mesmo
- A chamada a make-set cria uma árvore por elemento
  - Cuja raiz aponta para si mesma
- A chamada ao find-set deve seguir os nós da árvore, iniciando pelo elemento, até encontrar a raiz. A raiz é retornada como representante
- A união faz com que a raiz de uma árvore (conjunto) aponte para a raiz da outra
  - Como já encontramos os representantes (raízes) ao chamar o find-set, temos custo O(1) para unir os dois conjuntos

Quick-union

- O custo de find-set para um elemento será O(n), já que a árvore pode degenerar para uma lista encadeada
- Assim, foram propostas duas heurísticas para melhorar esse custo
  - União pela altura
  - Compressão de caminhos

Quick-union

- A união por altura funciona analogamente à união ponderada da implementação anterior
- O objetivo é manter a árvore mais balanceada (mais baixa), então unimos a árvore de menor altura à outra de maior altura
- Dessa forma, precisamos armazenar a altura de uma subárvore em cada nó
- Para unirmos as árvores x e y, comparamos as alturas:
  - Se h(x) > h(y), fazemos y.p = x
  - Se h(y) > h(x), fazemos x.p = y
  - Se h(x) = h(y), fazemos x.p = y e h(y) = h(y) + 1

Quick-union

- A segunda heurística de compressão de caminhos tenta fazer com que os elementos apontem diretamente para raiz o mais rápido possível
- Sempre que realizamos uma busca (find-set), percorremos as subárvores em direção à raiz.
- Dessa forma, no caminho de retorno ao nó original, podemos atualizar os pais apontando diretamente para aquele que representa a raiz

Quick-union

```
FIND-SET(x)
1  if x ≠ x.p
2        x.p = FIND-SET(x.p)
3  return x.p
```

Quick-union

- O uso combinado dessas heurísticas resulta em um custo O(m f(n)), em que f(n) é uma função com um crescimento “beeeeem” lento
  - n elementos
  - m operações make-set, find-set e union
- A demonstração dessa propriedade é bastante sofisticada e não será apresentada
- Contudo, para a maior parte das aplicações reais, o valor dessa função é desprezível, resultando em um custo linear no número de operações

Algoritmo de Kruskal (Análise)

- Retomando a análise do algoritmo de Kruskal
- A linha 4 ordena as arestas do grafo com custo O(|E| log |E|)
- Usando a estrutura de conjuntos disjuntos com quick-union e heurísticas, temos
  - |V| operações make-set
  - |E| operações find-set e union
- Logo, temos um custo agregado do laço O((|V|+|E|) f(|V|)
- Portanto, o custo do algoritmo é dominado pela ordenação
  - Como $|E| < |V|^2$, O(log |E|) = O(log |V|)
  - O custo do algoritmo é O(|E| log |V|) como o de Prim

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```

Leitura

- Seções 23.2, 21.1-21.3 (CLRS)