Universidade Tecnológica Federal do Paraná
Professor Murilo V. G. da Silva
Notas de aula – Projeto e Análise de Algoritmos – PPGCA (Aula 06)
Conteúdos da aula: [DPV06: 5.1.4, 5.2, 5.4]

[Observação: Estas notas de aula são apenas um esboço do que foi visto em aula e não devem ser usadas como material principal de estudos. O(a) aluno(a) deve acompanhar os conteúdos cobertos nesta aula usando os livros indicados na página da disciplina.]

Algoritmos gulosos

1. Árvore geradora mínima

O algoritmo de Kruskal e algoritmo de Prim, vistos na aula passada, são exemplos clássicos de algoritmos que usam a estratégia gulosa (as vezes chamada de estratégia míope). Abaixo reapresentamos estes dois algoritmos, mas desta vez explicitamente fazendo referência as estruturas de dados usadas em suas implementações: a estrutura *union-find* no algoritmo de Kruskal e um *fila de prioridades* no algoritmo de Prim.

KRUSKAL (G, w)

for all $u \in V$ **do**
    MAKE_COMPONENT(u)
end for
Ordene E pelos pesos de entrada w
$X = \emptyset$
for all $uv \in E$ em ordem crescente **do**
    if FIND$(u) \neq$ FIND(v) **then**
        Insira uv em X
        UNION(u, v)
    end if
end for

PRIM (G, w)

for all $u \in V$ **do**
    $cost[u] = \infty$
    $prev[u] = NULL$
end for
$cost[s] = 0$ (escolha $s \in V$ arbitrariamente)
Inicializa fila de prioridade Q
while $u =$ DeleteMin(Q) **do**
    for all $\{u, v\} \in E$ **do**
        if $cost[v] > w(uv)$ **then**
            $cost[v] = w[uv]$
            $prev[v] = u$
            DecreaseKey(Q, v)
        end if
    end for
end while

2. Código de Huffman

Queremos armazenar em binário um código genético formado pelos símbolos A, C, T, G. Digamos que estes símbolos aparecem no código genético com as seguintes frequências, respectivamente: $f_A = 0.55$, $f_C = 0.05$, $f_T = 0.15$ e $f_G = 0.25$. Qual a maneira mais compacta de se representar o código genético?

- **Codificação de tamanho fixo:** Uma primeira ideia seria fazer $A = 00$, $C = 01$, $T = 10$ e $G = 11$.
- **Codificação de tamanho variável:** Outra ideia é usar uma quantidade variável de bits por símbolo. O ideal seria usar menos bits para símbolos que mais se repetem com mais frequência. Mas há um problema: a codificação binária pode ser ambígua (você consegue pensar em um exemplo de codificação ambígua?). Solução: Escolha uma codificação tal que nenhuma sequência de bits que represente um símbolo seja prefixo de alguma outra sequência de bits que represente algum outro símbolo. Esse tipo de codificação é chamada de *codificação livre de prefixos*. Exemplo: $A = 0$, $C = 100$, $T = 101$ e $G = 11$.

Pergunta: Na codificação fixa, o que significa 00001110011100?

Pergunta: Na codificação variável, o que significa 0011101100110?

- Quantidade esperada de bits esperada para representar n símbolos na codificação fixa: $2n$.
- Quantidade esperada de bits esperada para representar n símbolos na codificação variável: $n\ (0.55 \cdot 1 + 0.05 \cdot 3 + 0.15 \cdot 3 + 0.25 \cdot 2) = 1.65n$.

Pergunta: Como encontrar a codificação livre de prefixos ótima?

Resposta: Criando uma árvore de codificação usando uma estratégia gulosa (detalhes vistos em sala).

5. Cobertura por por conjuntos

- **Entrada:** Um conjunto B e m subconjuntos $S_1, ..., S_m \subseteq B$.
- **Saída:** O menor número de subconjuntos S_i tal que a união dos conjuntos S_i seja B.

O problema acima é NP-completo, ou seja, não esperamos uma solução polinomial ótima para ele. Entretanto, veremos que um algoritmo guloso retorna uma solução "razoável". A ideia é que o algoritmo sempre procura escolher o conjunto que cobre o maior número de elementos e nunca volta atrás nas decisões tomadas. A quantidade de subconjuntos encontrada por este algoritmo nunca será maior que $k \cdot \ln n$, onde k é o número de conjuntos de uma solução ótima e n é o número de elementos de B. Segue a análise:

Lema: *A cada passo o algoritmo guloso cobre pelo menos* $1/k$ *elementos restantes.*

Prova: Visto em sala.

Suponha que a solução ótima tenha k conjuntos e seja n_i o número de elementos ainda não cobertos depois da execução de i passos. Em particular, observe que $n_0 = n$. Aplicando o Lema visto acima, temos que:

Passo 1: Sobram $\leq (1 - 1/k)n_0$ elementos;

Passo 2: Sobram $\leq (1 - 1/k)n_1 \leq (1 - 1/k)^2 n_0$ elementos;

Passo k: Sobram $\leq (1 - 1/k)^k n_0$ elementos;

Depois de t passos, onde $t = k \ln n$, o número de elementos que sobram é

$$n_t \leq (1 - 1/k)^t n_0 < (e^{-1/k})^t n_0 = e^{-t/k} n = 1.$$

Como o conjunto de elementos é discreto, $n_t = 0$. Ou seja, o algoritmo garantidamente encontra uma cobertura depois de $k \ln n$ passos.

Pergunta: Qual é a complexidade deste algoritmo?

Como manipular os componentes do grafo no Algoritmo de Kruskal?

Vamos finalizar a aula voltando ao algoritmo de Kruskal. Vimos que este algoritmo usa uma estratégia gulosa, mas não explicamos como o algoritmo mantém de maneira eficiente as componentes conexas sendo construídas para testar se uma aresta "fecha um ciclo".

Para tal, o algoritmo faz uso da estrutura de dados bastante simples e útil, chamada *union-find*. Veremos agora que mesmo no caso de uma estrutura bastante simples e com operações (que são algoritmos que manipulam esta estrutura) também bastante simples a análise pode se tornar bastante complicada. A estrutura de dados *union-find* é usada para manter uma coleção de conjuntos de elementos, de maneira que seja possível responder rapidamente em qual destes conjuntos um determinado elemento está contido. Além disso a estruura também permite fazer a união de dois conjuntos de maneira eficiente. No nosso caso os elementos são vértices do grafo e os conjuntos são componentes conexas contendo os vértices da árvore geradora que vai sendo construída pelo algoritmo de Kruskal. Nossas operações básicas são:

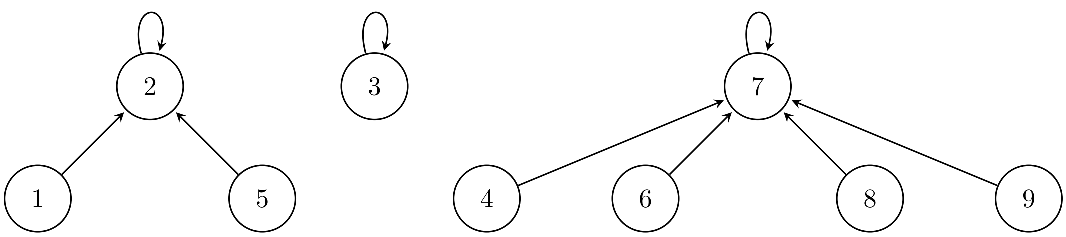
make_component(v): Cria uma contendo apenas o vértice v;

union(u, v): Junta os componentes C e C', onde $u \in C$ e $v \in C'$;

find(v): Retorna o componente em que o vértice v se encontra.

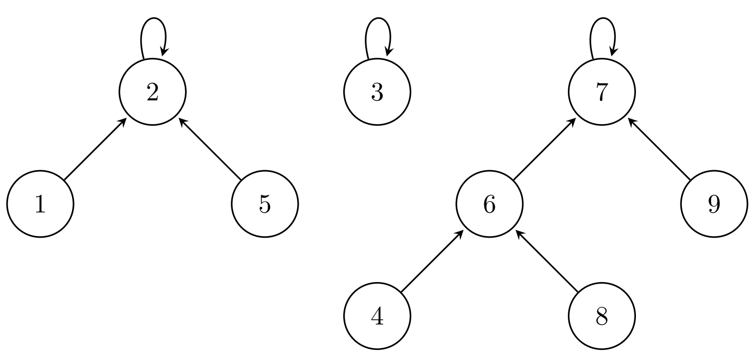
Aprsentamos abaixo duas estratégias simples de como implementar esta estrutura. Na primeira estratégia a operação $find$ é trivial e a operação *union* é mais trabalhosa. Na segunda estratégia a situação se inverte:

Elegendo um líder de componente (quick find)



2	2	3	7	2	7	7	7	7
1	2	3	4	5	6	7	8	9

União por rank (quick union)



2	2	3	6	2	7	7	6	7
1	2	3	4	5	6	7	8	9

Obs: Note que **estamos agora focando especificamente na análise da estrutura union-find**. As árvores representando a estrutura *union-find* mostradas no diagrama da página anterior, embora estejam associadas aos componentes originalmente sendo criados no algoritmo de Kruskal, não tem relação com a árvore geradora que o algoritmo de Kruskal encontra no final de sua execução. Em particular, note que arestas nesta estrutura de dados não necessariamente são arestas do grafo original.

Pergunta 1: Como melhorar o operação *union* na primeira estratégia (quick find)?
Pergunta 2: Como melhorar o operação *find* na segunda a estratégia (quick union)?

Resposta para pergunta 1: Na união de dois componentes, mudar o pai dos vértices do menor componente. Com isso, fazendo-se uma análise "vértice-cêntrica" temos o seguinte argumento amortizado: para cada vértice x, quando o representante de sua componente muda, a sua nova componente pelo menos dobra de tamanho. Portanto, como o número total de vértices é limitado em n, o ponteiro de x é mudado no máximo $\lg n$ vezes. Com isso, n operações de união custam no máximo $n \lg n$.

Resposta para pergunta 2: Diminuindo a profundidade das árvores. Como fazer isso? Na união, fazer o líder da árvore mais rasa tornar-se filho da árvore mais profunda. No caso de empate, fazer escolha arbitrária. Para fazer isso, introduzimos o conceito de *rank*. Se um vértice x é uma folha da árvore, então $rank(x) = 0$. Caso contrário, temos o seguinte: suponha que k é maior o rank entre os ranks de todos os filhos de x. Neste caso definimos $rank(x) = k + 1$.

Análise da estratégia de união por rank:

Seja x o vértice de maior rank. O custo das duas operações neste novo cenário é $\mathcal{O}(rank(x))$. Para provar isso, vamos mostrar que o número de vértices de rank r nunca ultrapassa $n/2^r$. Como o número de objetos é limitado a n, temos que $\forall x$, $rank(x) \leq \lg n$.

Lema ("Lema do rank baixo"): O número de vértices de rank r é $\leq \frac{n}{2^r}$.
Prova: Começamos provando as seguintes afirmações:

Afirmamção 1: Dados dois vértices x, y, se $rank(x) = rank(y)$, então as árvores contendo x e y são disjuntas

Provamos esta afirmação usando a contrapositiva. Suponha que as árvores de x e y contenham um mesmo vértices z. Então existem caminhos $z, ..., x$ e $z, ..., y$. Como em árvores caminhos são únicos, temos que x é ancestral do y ou vice-versa. Portanto $rank(x) \neq rank(y)$. Assim finalizamos a prova da Afirmação 1.

Afirmamção 2: Se $rank(x) = k$, então a árvore contendo x contém pelo menos 2^k vértices.

Vamos provar esta afirmação por indução no número de operações *union*.

Base: Nenhuma operação *union*: Neste caso $\forall x$, $rank(x) = 0$ e a árvore contendo x tem tamanho 1 ($2^0 = 1$).
H.I.: Com n operações *union*, se $rank(x) = k$, então a árvore contendo x contém pelo menos 2^k vértices.

Para concluir a prova suponha agora que realizamos a $(n+1)$-ésima operação *union*. Caso nenhum vértice mude de rank, não precisamos provar nada, pois antes desta operação todos os vértices respeitavam a propriedade que queremos provar e depois da operação as árvores dos vértices de rank k contém pelo menos tantos vértices quanto antes. Vamos nos preocupar agora com o caso em que algum vértice mude de rank. Digamos que na operação *union(x,y)* a raiz da árvore de y passa a apontar para a raiz da árvore de x. Seja $r_1 = find(x)$ e $r_2 = find(y)$ e $rank(r_1) = rank(r_2) = k$. Após a operação *union*, o único vértice em que o rank se modificou foi r_2 e seu novo rank é $k+1$ e portanto ele é único vértice em que possivelmente a propriedade agora não valha mais (i.e., sua árvore pode não ter vértices suficientes). Mas, pela Hipótese de Indução, o tamanho da árvore com raiz r_1 é $\geq 2^k$ e o tamanho da árvore com raiz r_2 é $\geq 2^k$. Portanto a nova árvore de r_2 tem $2^k + 2^k = 2^{k+1}$ vértices. Isso prova a Afirmação 2.

A prova do "Lema do rank baixo" segue das afirmações 1 e 2 e o fato de que existem apenas n elementos. □

Compressão de caminhos

Considere a seguinte ideia para otimizar a estrutura de dados: Quando fazemos uma operação $find(x)$ e obtemos r como raiz da árvore, setamos r como pai de x e também de todos os elementos percorridos durante o caminho entre x e r. Antes de fazer a análise de como isso melhora o desempenho da estrutura, observe que:

- Agora $rank(x)$ é um limitante superior para a distância entre x a folha mais distante.
- O "lema do rank baixo" ainda é verdadeiro.
- O fato que $rank(PAI(x)) > rank(x)$ ainda é verdadeiro.

Definição: A função $\log{*}n$ nos diz o número de vezes que temos que aplicar iterativamente a operação log em n para se chegar a 1. Para qualquer número que signifique algo no universo conhecido, $\log{*}$ deste número é ≤ 5. Exemplo: $\log{*}2^{65536} = 5$, uma vez que precisamos aplicar a função 5 vezes: $2^{65536} \rightarrow 65536 \rightarrow 16 \rightarrow 4 \rightarrow 2 \rightarrow 1$.

Teorema 1 [Hopcroft-Ullman, 73]: Utilizando união por rank e compressão de caminhos o custo de m operações *union/find* é $\mathcal{O}(m \cdot \log{*}n)$, para casos onde $m = \Omega(n)$.

Prova: Lembramos que como cada operação *union* se resume a duas operações *find*, sempre que quisermos provar que m operações, sejam elas *union* ou $find$, custam $\mathcal{O}(f)$, basta nos focarmos apenas em mostrar que m operações $\boldsymbol{find}$ custam $\mathcal{O}(f)$.

Vamos começar a prova particionando o conjunto $\{0, 1, 2, ..., n\}$ em "blocos" da seguinte maneira:

$\{0\}, \{1\}, \{2\}, \{3, 4\}, \{5, ..., 16\}, \{17, ..., 65536\}, \{65537, ..., 2^{65536}\}, ...\ ,\{..., n\}$

Note que se k é último elemento de um bloco, então o último elemento do bloco seguinte é 2^k. A ideia é que vamos analisar o $rank$ de cada vértice x na estrutura de dados e verificar em qual bloco $rank(x)$ estará. Chamaremos estes blocos de "blocos de rank".

- Observação: O númbero de blocos de rank é $\mathcal{O}(\log{*}n)$
- Ideia chave: Suponha que realizamos uma operação $find$ e para tal percorremos os vértices $x_1, x_2, ..., x_k$ no caminho em direção a raiz da estrutura. Cada vez que $rank(x_i)$ está em um bloco e $rank(PAI(x_{i+1}))$ está em um bloco posterior temos uma indição que o caminho sofreu grande compressão.

Vamos dividir os vértices agora em dois grupos: Vértices "simples" e vértices "problemáticos":

- **Vértices simples:** Todo vértice x que respeita alguma das 3 propriedades a seguir:
  (1) x é raiz;
  (2) $PAI(x)$ é raiz;
  (3) O $rank(PAI(x))$ está um bloco posterior ou $rank(x)$.
- **Vértices problemáticos:** Demais vértices.

Seja T o custo total de m operações $find$, S a quantidade de visitas a vértices simples, P a quantidade de visitas a véritices problemáticos e K a quantidade de blocos de rank.

Afirmação 1: $S = \mathcal{O}(m \cdot \log{*}n)$

Para provar isso, veja que em uma operação $find$, o máximo de vértices simples que podem se percorridos é $2 + K$. Isso vem do fato que para vértices simples temos no máximo uma raiz um filho da raiz e um vértice para cada bloco de rank. Portanto, em uma operação find, o número de vértices simples percorridos é no máximo $\mathcal{O}(\log{*}n)$ e, portanto, $S = \mathcal{O}(m \cdot \log{*}n)$.

Afirmação 2: $P = \mathcal{O}(m \cdot log{*}n)$

Aqui temos que ter mais cuidado e usar um argumento de amortização e ver quantos vértices problemáticos podem ser visitados durante a sequência de operações $find$ que estamos considerando. O ponto chave é que existe um número máximo n de vértices na estrutura e sempre que um vértice é visitado (em particular, os vértices problemáticos que nos interessam agora) ele ganha um novo pai com $rank$ maior do que o atual. Com isso, com o passar do tempo, os vértices vão deixando de ser problemáticos quando o rank de seus pais crescerem muito.

Seja B o bloco de rank $\{k+1, k+2, ..., 2^k\}$ e B' o bloco imediatamente posterior. Note que $|B| = 2^k$. Com isso, um vértice com rank em B não pode ser visitado mais do que 2^k vezes enquato ele for problemático, pois, a cada visita, o rank do seu pai vai sendo incrementado até que eventualmente o rank caia no bloco B' ou em algum bloco posterior. Quando isso acontece, o vértice passa a ser simples.

Seja agora um vértice problemático x tal que $rank(x) \in B$:

- Fato 1: Pelo raciocínio do parágrafo anterior x pode ser visistado no máximo 2^k vezes.
- Fato 2: Pelo lema do rank o número de vértices com rank (final) em B é $\leq \sum_{i=k+1}^{2^k} \frac{n}{2^i} = n \sum_{i=k+1}^{2^k} \frac{1}{2^i} \leq \frac{n}{2^k}$.

  Para ver o porquê da última desigualdade note que $\frac{1}{2^k} \geq \sum_{i=k+1}^{\infty} \frac{1}{2^i}$.
- Juntando os Fatos 1 e 2 temos que o total de visitas a vértices problemáticos com rank em B é $\leq 2^k \cdot \frac{n}{2^k} = n$.
- Como temos apenas $\mathcal{O}(log{*}n)$ blocos, o total de visitas a vértices problemáticos, contando todos os blocos, é $\mathcal{O}(n \cdot \log{*}n)$. Como $m = \Omega(n)$, então $P = \mathcal{O}(m \cdot \log{*}n)$. Isso finaliza a prova da Afirmação 2.

Colocando as duas afirmações juntas, temos que o total de operações T é dado por $T = S + P = \mathcal{O}(m \cdot log{*}n)$. Isso conclui a prova do Teorema. □

Esta análise mostra que a estrutura de dados opera de maneira quase linear. Ainda assim nossa análise não foi "apertada" o suficiente para nos dizer o quão próximo de linear custam as operações $union/find$ (amortizadamente). Caso tenha interesse, veja as notas de aula opcionais sobre o assunto para uma análise mais apertada.