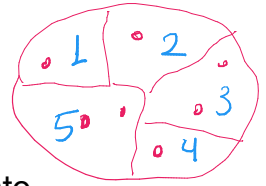
PAA - Aula ~~16~~ 17

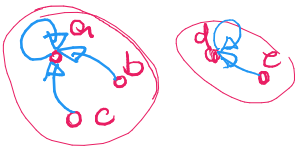
Union Find, k-Clusterização com Espalhamento Máximo



Union Find é uma estrutura especializada em registrar partições de um conjunto,
- i.e., em manter o registro das partes em que se divide tal conjunto.

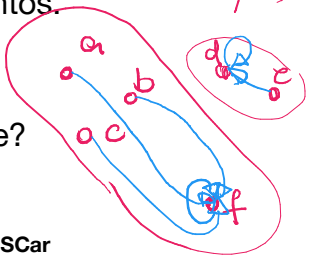
- Na última aula usamos essa estrutura no algoritmo de Kruskal,
  - para consultar a que componente conexa pertence cada vértice.

Union Find suporta três operações:
- makeSet(e) - cria uma parte (subconjunto) apenas com o elemento e
- find(e) - devolve o representante da parte em que está e
- union(e, f) - une a parte de e com a parte de f



Em uma das implementações mais simples dessa estrutura,
- todo elemento de uma parte aponta para o representante da mesma.
  - Com isso, as operações makeSet() e find() levam tempo $O(1)$
- No entanto, a operação union() pode levar tempo $O(n)$
  - o que acontece se os conjuntos envolvidos tem muitos elementos.

union(c, f)

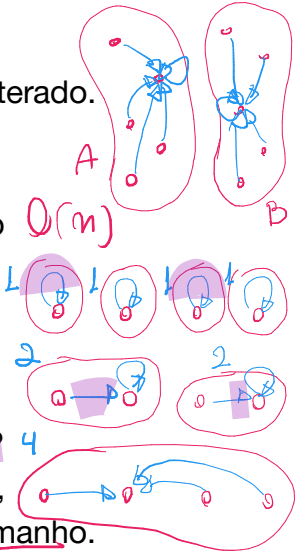


Quiz1: Curiosamente, um pequeno detalhe da implementação
  - pode tornar essa implementação eficiente. Qual é esse detalhe?
- Dica: você só precisa trocar o representante de um dos conjuntos.

Para tornar essa primeira implementação do Union Find mais eficiente,

- na união apenas o representante da menor parte envolvida é alterado.

- Para isso é preciso aumentar a estrutura
  - guardando (e atualizando) o tamanho de cada parte.
- Note que, mesmo nesse caso uma operação union() pode levar tempo $O(n)$
  - caso cada conjunto tenha $n/2$ elementos, por exemplo.
- No entanto, o custo amortizado da união passa a ser $\log n$
  - i.e., o custo médio de n operações union() é $O(n \log n)$



Para ver isso, responda: Quantas vezes um representante pode ser alterado?

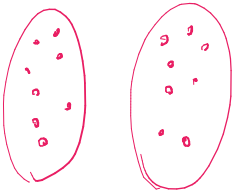
- Dado um elemento e, cada vez que o representante de e é alterado,
  - temos que uma união fez a parte de e ao menos dobrar de tamanho.
- Sendo k o número de alterações que o representante de e sofreu,
  - sabemos que o número de elementos no conjunto de e $\geq 2^k$
    - já que no início temos $2^k = 2^0 = 1$ elemento por conjunto.
- Como n é o número máximo de elementos, temos $n \geq \#elem_{conj}(e) \geq 2^k$ $\Rightarrow k \leq \lg n$
- Somando ao longo de todos os n elementos,
  - o total de alterações de representantes é $O(n \lg n)$

Essa implementação é suficiente para garantir

- que o algoritmo de Kruskal rode em tempo $O(m \lg n)$
- Mas existem impressionantes melhorias na implementação do Union Find.

Continuando o estudo de implementações para o Union Find,

- vamos ver as melhorias: Lazy Unions + Union by Rank + Path Compression.

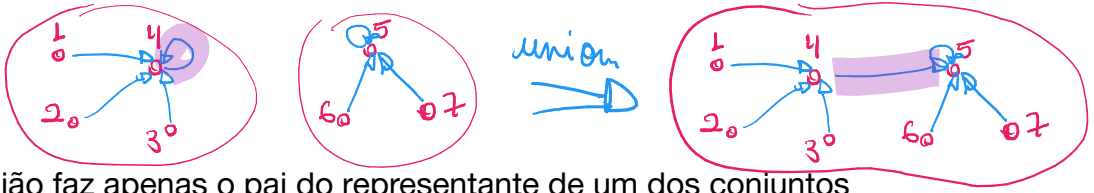


Lazy Unions (Uniões Preguiçosas)

Na nossa primeira implementação find() custa $O(1)$ e union() custa $O(n)$

- Vamos tentar reduzir o trabalho realizado pelas uniões.

Exemplo:



Ideia: a união faz apenas o pai do representante de um dos conjuntos

- se tornar o representante do outro conjunto.

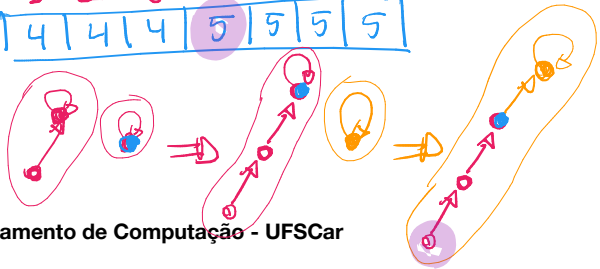
- Quiz2: Como essa mudança afeta o find()? representante ≠ pai

Pseudocódigo do union(e, f): $r_1 = find(e)$; $r_2 = find(f)$; $r_1 \rightarrow pai = r_2$

Implementação com vetor:

Eficiência de pior caso:

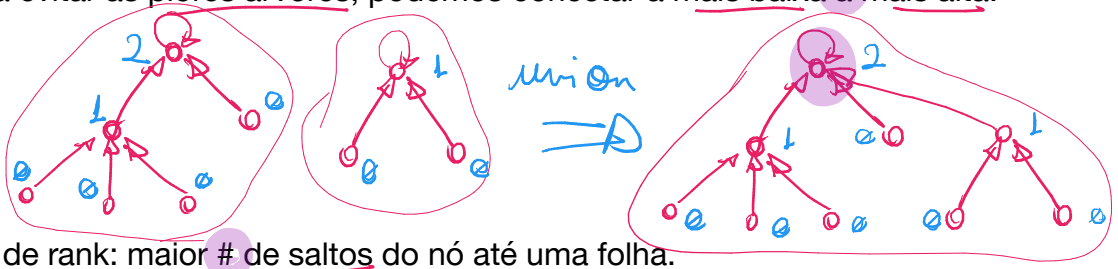
- Tanto find() quanto union() podem custar $O(n)$



Union by Rank (União baseada em Posto)

Ideia: para evitar as piores árvores, podemos conectar a mais baixa à mais alta.

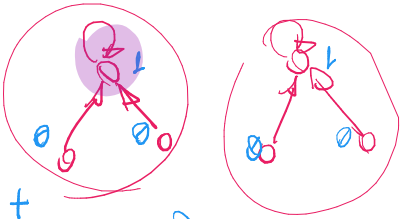
Exemplo:



Definição de rank: maior # de saltos do nó até uma folha.

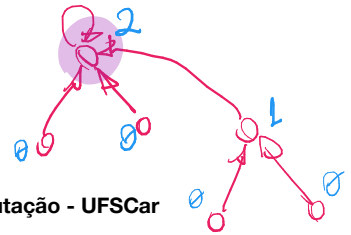
- No início todo elemento tem rank igual a 0

Invariante: rank(e) = $1 + \max_{f \text{ é filho de } e} \{\text{rank}(f)\}$



Pseudocódigo do union(e, f):

```
r1 = find(e); r2 = find(f);
Se rank(r1) = rank(r2): rank(r1)++
Se rank(r1) > rank(r2): r2 -> pai = r1
Senão: r1 -> pai = r2
```



Como os ranks mudam com as operações?

- Só aumentam na união de conjuntos de mesmo rank.

Lema Tamanho vs. Rank: um conjunto de rank k tem tamanho $\geq 2^k$

- A intuição é que cada união que aumenta o rank de um elemento
  - também dobra o tamanho do subconjunto resultante.
- Vamos provar por indução em k

Base: $k=0$ temos $2^k = 2^0 = 1$ elemento.

H.I.: Resultado vale para $k-1$

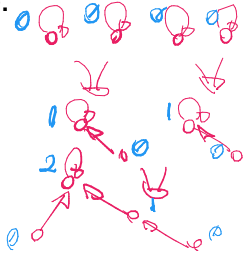
Passo: Se o rank de um elemento se tornou k
  - é porque ocorreu uma união de 2 conjuntos de rank $k-1$
- Pela H.I., cada conjunto de rank $k-1$ tem tamanho $\geq 2^{k-1}$
- O tamanho do conjunto resultante
  - é igual à soma dos tamanhos dos conjuntos originais,
- i.e., tamanho resultante $\geq 2^{k-1} + 2^{k-1} = 2 \cdot 2^{k-1} = 2^k$

Consequências: note que $n \geq$ tam. de qualquer conj. de rank $k \geq 2^k$

- Assim, $k \leq \lg n$ e as operações union() e find() tem eficiência $O(\log n)$
  - porque o tempo delas depende do # de saltos até o representante.

Uma conclusão interessante desse lema (com mais algumas propriedades)

- é que são poucos os nós com rank alto.

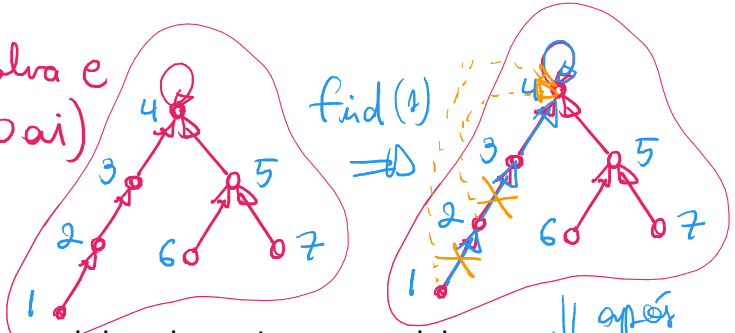


Path Compression (Compressão de Caminho)

Vamos olhar a operação find() mais de perto.

Pseudocódigo do find(e):

se e->pai = e : devolva e
devolva find(e->pai)

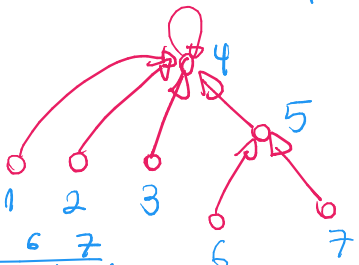


Exemplo de compressão:

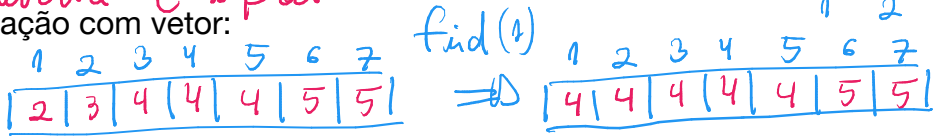
Ideia: find() pode aproveitar para trocar o pai dos elementos no caminho.

Pseudocódigo do find(e) com compressão de caminhos:

se e->pai = e : devolva e
e->pai = find(e->pai)
devolva e->pai



Implementação com vetor:



Eficiência do Union-Find com Lazy Unions + Union by Rank + Path Compression:

- na prática, tempo médio constante por operação, i.e., $O(1)$

≤ 5 p/ qualquer valor relevante

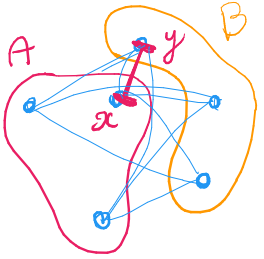
Problema da k-clusterização com espalhamento máximo

Entrada:

- um conjunto de pontos S
- uma função $d()$ de distância entre os pontos,
  - indicando quão diferentes são dois elementos de S
- e um inteiro positivo k indicando o número de clusters desejado.

Solução:

- agrupar os pontos em k clusters,
  - de modo a maximizar a menor distância entre clusters.
- A distância entre dois clusters A e B é dada
  - pela distância entre os pontos separados mais próximos,
    - i.e., $d(A,B) = \min_{p \in A, q \in B} \{d(p,q)\}$
- Assim, o objetivo é encontrar uma solução que
  - maximize $\min \{d(p,q)\}$ (p e q estão separados)
- sendo que dois pontos p e q estão separados
  - se pertencem a clusters distintos.

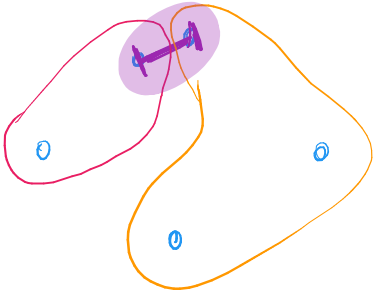


$d(A,B) = d(x,y)$

Estratégia gulosa:

A função objetivo é a menor distância entre qualquer par de pontos separados,

- i.e., que não pertencem ao mesmo cluster.



Por isso, nossa estratégia gulosa é unir, em cada iteração,

- o par de pontos separados mais próximo,
  - o que deve aumentar o valor da função objetivo.

Observe que cada união funde dois clusters.

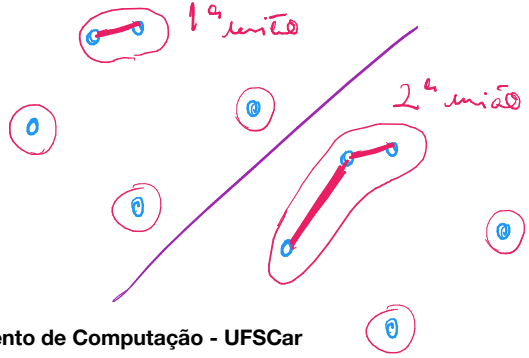
- Começamos com cada um dos n pontos isolados e
  - queremos terminar com k clusters.
- Assim, o algoritmo termina depois de $n-k$ iterações/fusões.

Note que este algoritmo é uma variante do algoritmo de Kruskal, em que

- pontos são vértices,
- pares de pontos são arestas,
- distância entre pares são custos das arestas,

e que termina quando obtém k componentes conexas,

- que são os clusters.



Prova de corretude da estratégia gulosa

Seja $C_1, \ldots, C_k$ uma clusterização gulosa

- com valor de espaçamento (função objetivo) L

Seja $C'_1, \ldots, C'_k$ uma clusterização qualquer com valor de espaçamento L'

⇒ Vamos mostrar que $L' \leq L$

- Como $C'_1, \ldots, C'_k$ é uma solução arbitrária
  - e o problema é de maximização,
- isso implica a otimalidade de $C_1, \ldots, C_k$

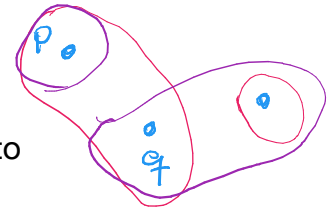
Caso 1: Se $C_1, \ldots, C_k$ e $C'_1, \ldots, C'_k$ são iguais,

- então elas tem o mesmo espaçamento e o resultado segue.

Gulosa

Caso 2: Se as soluções são diferentes, seja p e q um par

- que está no mesmo cluster na solução gulosa $(C_1, \ldots, C_k)$
  - e em clusters diferentes na outra solução $(C'_1, \ldots, C'_k)$.
- Note que um par assim deve existir, já que ao menos um elemento
  - deve ter vizinhos diferentes nas duas clusterizações.



Outra

Agora temos um subcaso fácil e um difícil.

Subcaso fácil: p e q foram unidos diretamente pelo algoritmo guloso.

- Como o algoritmo guloso une sempre o par separado mais próximo,
  - neste caso o espaçamento L da solução gulosa é pelo menos $d(p,q)$
- i.e., $L \geq d(p,q)$, pois para chegar a unir p e q
  - o algoritmo guloso já uniu todos os pares mais próximos antes.

Mas, como p e q estão em clusters diferentes na solução $C'_1, \ldots, C'_k$

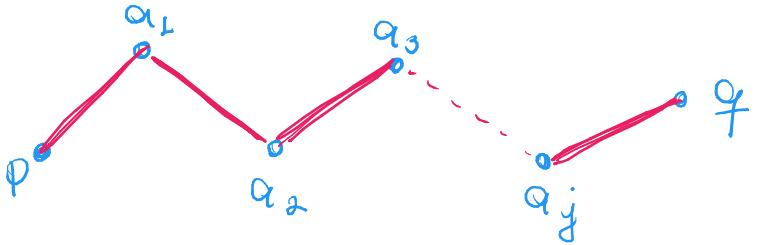
- temos que o espaçamento dela é no máximo L', i.e., $L' \leq d(p,q) \leq L$ ✓

⇒ Subcaso difícil: p e q não foram fundidos diretamente pelo algoritmo guloso,

- i.e., eles acabaram no mesmo cluster porque algum elemento do cluster
  - de p foi fundido com algum elemento do cluster de q

Observe que, como todo elemento começa isolado, podemos descrever

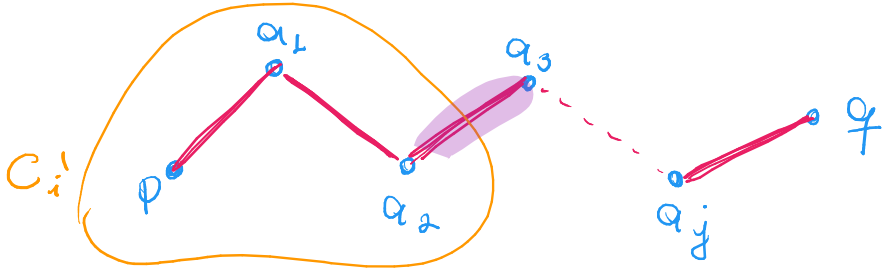
- um caminho de fusões que levaram p e q a terminarem juntos.



  - No desenho, cada aresta indica uma fusão direta.

Seja C_i' o cluster da outra solução em que está p. Observe que,

- como p e q estão em clusters distintos em $C_1', C_2', \ldots, C_k'$
- em algum ponto do caminho que começa em p e termina em q
  - devemos sair do cluster



No nosso exemplo, observe que o par (a_2, a_3) está no mesmo cluster

- na solução gulosa e está em clusters distintos em $(C_1', \ldots, C_k')$
- Em geral, chamemos de (a_2, a_1) o primeiro par do caminho de fusões
  - que atravessa a fronteira do cluster C_i'

⇒ caímos no caso fácil c/ (a_2, a_1)

Para concluir a prova basta observar que

- (a_2, a_1) foram fundidos diretamente na solução gulosa.
  - Portanto, $d(a_2, a_1) \leq L$
- (a_2, a_1) estão em clusters distintos em $C_1', \ldots, C_k'$
  - Portanto, $L' \leq d(a_2, a_1)$
- Assim, $\boxed{L' \leq d(a_0, a_3) \leq L}$ ✓