Tópicos Avançados em Algoritmos

Hamilton José Brumatto

Bacharelado em Ciências da Computação - UESC

25 de abril de 2019

Conjuntos Disjuntos

Conjuntos

- O STL possui um tipo próprio para representar conjuntos: `set`
- O `set` possui as seguintes características:
  - Cada elemento é único.
  - Os elementos possuem ordem e são guardados ordenados.
  - As principais operações são: `find`, `insert` e `erase`
- Apesar disto, este tipo não oferece as operações de conjuntos: $A - B$, $A \cup B$, $A \cap B$ e $A \triangle B$.

Conjuntos

- Nós podemos substituir o `set` por um vetor ordenado, teremos o mesmo efeito, com a preocupação de manter os objetos únicos.
- Além das operações: `find`, `insert` e `erase`, agora teremos também:
  - `set_difference` $= A - B$
  - `set_union` $= A \cup B$
  - `set_intersection` $= A \cap B$
  - `set_symmetric_difference` $= A \triangle B = (A \cup B) - (B \cup A) = (A - B) \cup (B - A)$
  - `unique`: Remove elementos duplicados de um conjunto.

Uri - 2222 - Brincando com Conjuntos

Dabriel é um menino fissurado por matemática, ele acaba de aprender em sua escola operações sobre conjuntos. Após passar a tarde toda brincando com alguns conjuntos que ele possui, chega a hora de resolver as lições de casa, porém ele já está muito cansado e com medo de que possa cometer alguns erros, solicitou sua ajuda. Dabriel deseja um programa de computador que dado **N** conjuntos e os elementos de cada conjunto, ele possa realizar algumas operações, são elas:

- **1 X Y**: Retorna a quantidade de elementos distintos da intersecção entre o conjunto **X** com o **Y**.
- **2 X Y**: Retorna a quantidade de elementos distintos da união entre o conjunto **X** com o **Y**.

Entrada:

A entrada é composta por diversas instâncias. A primeira linha da entrada contém um inteiro **T** indicando o número de instâncias. Cada instância inicia com um inteiro **N** ($1 \leqslant \mathbf{N} \leqslant 10^4$), representando a quantidade de conjuntos que Dabriel possui. As próximas **N** linhas começam com um inteiro **Mi** ($1 \leqslant \mathbf{Mi} \leqslant 60$), que indica o total de elementos que o conjunto **i** possui, segue então **Mi** inteiros **Xij** ($1 \leqslant \mathbf{Xij} \leqslant 60$), que representam o valor de cada elemento. Na próxima linha contém um inteiro **Q** ($1 \leqslant \mathbf{Q} \leqslant 10^6$), representando quantas operações Dabriel deseja realizar. Nas próximas **Q** linhas terá a descrição de uma operação.

Saída:

Para cada operação seu programa deverá imprimir a quantidade de elementos, conforme explicado na descrição.

Solução: Estratégia

- Construção:
  Para cada t lemos o valor de n e criamos um vetor de vetor de inteiros:
  `vector<vector<int> > v(n);`

  Para cada n (i de 0 a n):
  - lemos m e iniciamos o vetor: `v[i].push_back(k)` (k foi lido).
  - ordenamos: `sort(v[i].begin(),v[i].end())`
  - eliminamos números repetidos: `auto last = unique(v[i].begin(),v[i].end())`
  - redimensionamos o vetor: `v[i].resize(last-v[i].begin());`
- Consulta:
  - Interseção: criamos um vetor com o mesmo tamanho de um deles:
    ```
    vector<int> inter(v[i].size());
    auto last = set_intersection(...inter.begin());
    cout << (last - inter.begin());
    ```
  - União é semelhante:
    ```
    vector<int> uni(v[i].size()+v[j].size());
    ...
    ```

STL e eficiência

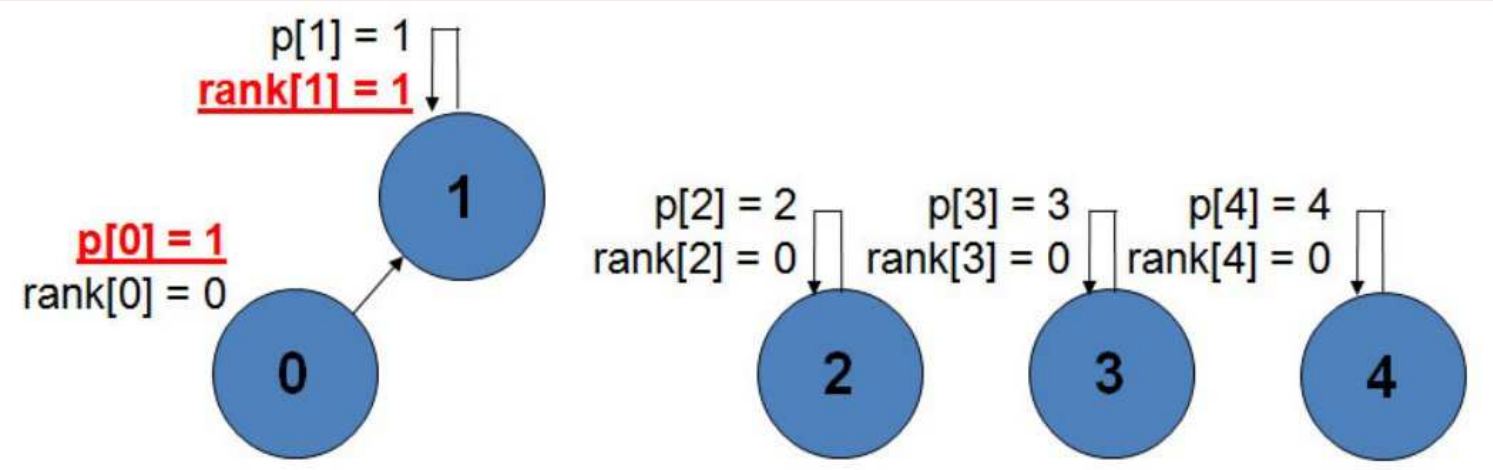
- O problema das funções definidas na bibliotecas do STL para trabalhar com conjuntos, é a eficiência, apesar de intuitiva.
  - Inserir um elemento em uma posição ($O(n)$ - tem de deslocar os demais).
  - Remover um elemento ($O(n)$).
  - Encontrar um elemento ($O(\log n)$ - pensando em busca binária).
  - Construir um conjunto ($O(n \log n)$ - precisa ordenar).
  - União, interseção, diferença, ... ($O(n)$).
- Existe uma outra estrutura que permite buscar elementos em $O(1)$, ou verificar se 2 elementos pertencem ao mesmo conjunto.
- Exemplos de problemas que recaem nestas necessidades: Árvore Geradora Mínima (Prim), Componentes fortemente conexas em grafos, ...

Union-Find Disjoint Sets

- União-Busca em Conjuntos Disjuntos (UFDS) representa uma classe com uma estrutura que organize conjuntos de forma que as buscas de elementos (ou ao conjunto a que pertença) seja em $O(1)$.
- Nesta representação, um item será indicado como representante do conjunto.
- Desta forma, um conjunto pode ser representado por uma árvore onde o raiz é o representante.
- Uma coleção de conjuntos disjuntos seria uma floresta.
- Um conjunto é então um vetor onde cada índice do vetor representa um elemento, e o conteúdo do vetor é, para cada elemento, o seu representante, que pode ser si mesmo, se ele for representante.

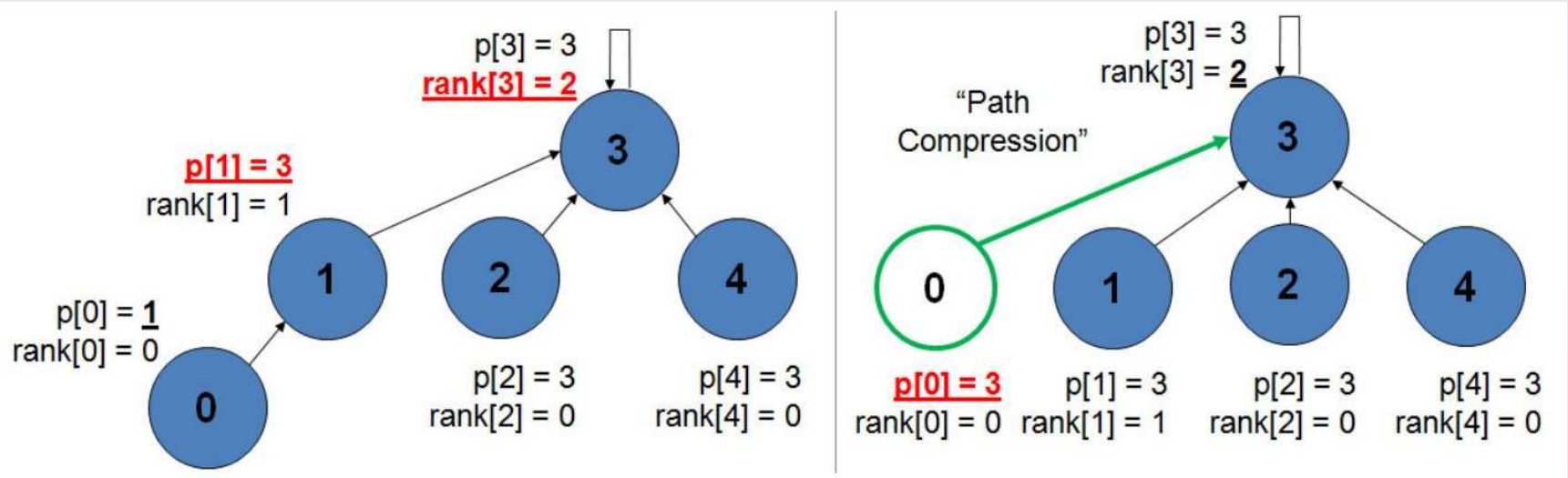
Union-Find Disjoint Sets

- A principal inovação nesta estrutura é escolher o item 'pai' representante para representar o conjunto.
- Se cada conjunto é representado por um único item, então identificar se dois itens pertecem a um mesmo conjunto é simples.
- Além do vetor de inteiros representando os 'pais', existe um vetor de inteiros que representa a altura de um elemento na árvore.



Union-Find Disjoint Sets: Busca

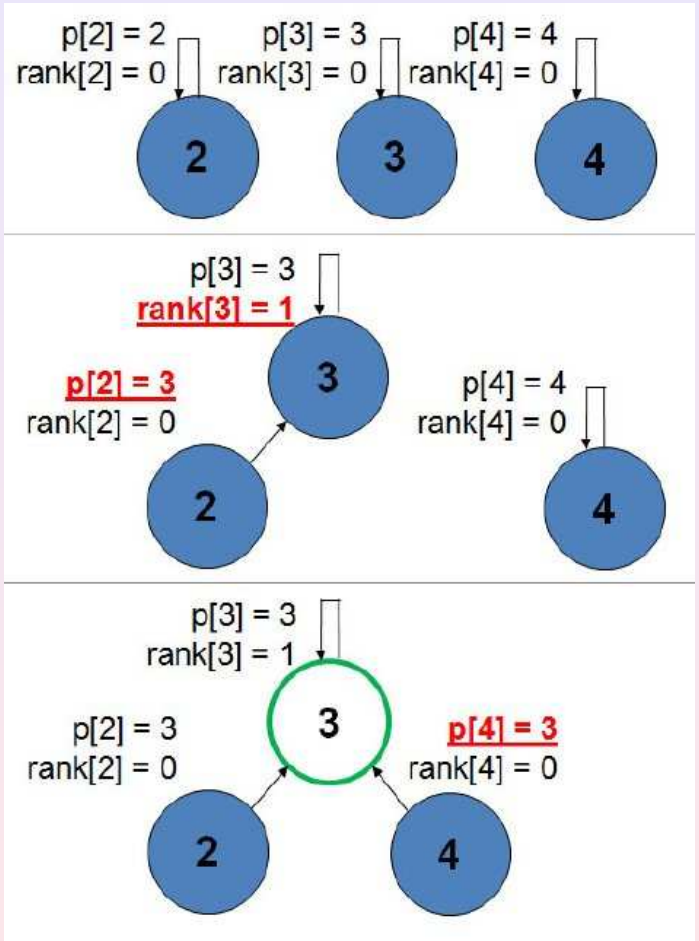
- Descobrir se um elemento pertence a um conjunto é feita percorrendo a árvore até o raiz.
- Para acelerar a busca, se o 'pai' imediato não é o raiz, ao realizar a busca, o pai é atualizado para ser o raiz. Isto é chamado de "Compressão de Caminho".



Union-Find Disjoint Sets: Union

- Para unir dois conjuntos disjuntos, basta tomar o item (raiz) representativo de um conjunto passar a ser o novo ‘pai’ do item representativo do outro conjunto.
- A escolha do representativo final será pelo ‘rank’ que representa a altura da árvore, escolher o de maior ‘rank’ como representativo mantém a árvore curta.
- Se ambos tem o mesmo ‘rank’ escolhe-se um e aumenta o seu ‘rank’.

Union-Find Disjoint Sets: Union



Classe UFDS

```
class UnionFind {
private:  vector<int> p, rank;
public:
 UnionFind(int N) {
  rank.assign(N, 0); p.assign(N, 0);
  for (int i = 0; i < N; i++) p[i] = i;
 }
 int findSet(int i) {
  return (p[i] == i) ?  i :  (p[i] = findSet(p[i]));
 }
 bool isSameSet(int i, int j) {
  return findSet(i) == findSet(j);
 }
 void unionSet(int i, int j) {
  if (!isSameSet(i, j)) { // Se forem de conjuntos disjuntos
   int x = findSet(i), y = findSet(j);
   if (rank[x] > rank[y]) p[y] = x;
   else {                 p[x] = y;
                          if (rank[x] == rank[y]) rank[y]++;
   }
  }
 }
};
```

URI - 2657 - Sensate

Sensates, ou Homo sensoriums, são uma espécie de humanos que são conectados telepaticamente com outras pessoas pelo mundo. Um grupo de sensates é chamado de cluster, e os membros de um cluster podem contactar um ao outro independente de onde eles estiverem no mundo. Após saber dos sensates os organizadores de maratonas de programação ficaram muito preocupados, pois mesmo que uma pessoa não saiba programar, se alguém do seu cluster souber ela irá se sair bem na competição mesmo sem saber nada. Por causa disso os organizadores de maratonas criaram novas regras para determinar se uma pessoa poderá participar de uma competição ou não. Primeiramente cada pessoa no planeta irá receber uma nota de um a dez, com o intuito de medir quão bem ela programa. Após isso para a pessoa participar ela terá que se encaixar em uma das categorias abaixo:

1. Não ser um sensate.
2. Ser um sensate e não ter ninguém no mesmo cluster com nível de programação maior que cinco.
3. Ser um sensate e ter seu nível de programação maior ou igual a 5.

Desenvolva um programa que consiga determinar se uma pessoa poderá ou não participar de competições de programação.

Entrada:

A primeira linha contém três inteiros **N**, **M** e **Q** ($1 \leqslant$**N**, **M**, **Q**$\leqslant 10^4$) representando respectivamente o número de pessoas, quantas ligações existem entre essas pessoas e quantas consultas serão realizadas. Cada uma das **N** linhas seguintes contém uma string **S** ($1 \leqslant |\mathbf{S}| \leqslant 10$) e um inteiro **V** ($0 \leqslant$**V**$\leqslant 10$) representando respectivamente o nome da pessoa e seu nível de programação. As próximas **M** vão conter duas strings **S1** e **S2** ($1 \leqslant |\mathbf{S1}|, |\mathbf{S2}| \leqslant 10$), representando que a pessoa **S1** e **S2** estão no mesmo cluster. As próximas **Q** linhas vão conter uma string **T** ($1 \leqslant |\mathbf{T}| \leqslant 10$) que representa o nome da pessoa que pretende participar da competição.
As strings serão compostas apenas de letras maiúsculas e minusculas.

Saída:

Para cada uma das **Q** consultas imprima `"S"` se a pessoa pode competir ou `"N"` caso contrário.

Resolvendo o URI 2657

- Vamos usar a classe UnionFind com alguns recursos a mais:
  - Além de *p* e *rank*, vamos incluir: *cluster*, que tal como *p* guarda o nível máximo de programação dos elementos de uma árvore. No unionSet o id representativo de ambos grupos ganham o maior de seus valores cada um.
  - Também o *sensate* que guarda se um elemento participa ou não de um grupo, inicialmente todos é 0 (não), se participar ganha 1 (sim)
  - Como cada elemento tem como ID um nome, criamos um `map<string, pair<int, int> > sensor` que irá representar cada elemento. Mapeado por nome, e os ints do pair são: ID (do UnionFind) e nível.

Resolvendo o URI 2657

- Também foram incluídos novos métodos:
  - `setSensor(nome,id,nivel)` inclui um sensor com ID e nível;
  - `getSensorID(nome)` retorna o ID de um sensor;
  - `getSensorNivel(nome)` retorna o nível de um sensor;
  - `setSensate(id)` transforma o elemento id em sensate (participante de um grupo);
  - `getSensorSensate(nome)` retorna se um sensor é um sensate; e
  - `getClusterNivel(nome)` retorna o nível de programação maior do grupo que o elemento pertence.

Solução URI 2657 - main()

```
int main() {
 int n, m, q, nivel, id1, id2;
 string nome, outro;
 cin >> n >> m >> q;
 UnionFind uf(n);
 for(int i=0; i<n; i++) {
  cin >> nome >> nivel;
  uf.setSensor(nome, i, nivel);
 }
 for(int i=0; i<m; i++) {
  cin >> nome >> outro;
  id1 = uf.getSensorID(nome); id2 = uf.getSensorID(outro);
  uf.unionSet(id1,id2); // unionSet chama setSensate e atualiza cluster
 }
 for(int i=0; i<q; i++) {
  cin >> nome;
  if(!uf.getSensorSensate(nome)) cout << "S" << endl;
  else if(uf.getSensorNivel(nome) >= 5) cout << "S" << endl;
  else if(uf.getClusterNivel(nome) <= 5) cout << "S" << endl;
  else cout << "N" << endl;
 }
 return 0;
}
```