TB1 pt 2

Maria Luiza Fernandes 2018.1.08.015

15/03/2019

1 Introdução

O problema dado em sala de aula pede para que seja implementado um algoritmo que realize função de união entre conjuntos, de forma que se tenha um bom rendimento. Foi dado as seguintes especificações:

1. Deve-se criar uma estrutura, que pode ser considerada um conjunto (C), contendo números de 0 até N em que cada valor é um subconjunto contido em (C).

2. Deve-se implementar uma função denominada juntar, que tem como parâmetros dois números inteiros contidos em (C). Cada vez que essa função for chamada ela deve unir esses subconjuntos tornando-os apenas um.

3. Além disso, foi pedido para que se implemente uma outra função denominada teste, para que dado dois valores inteiros seja retornado "True" se eles estão no mesmo conjunto ou "False" caso contrário.

C = {{0},{1},{2},{3},{4}}
N = 4

juntar(1, 2)
juntar(3, 4)
C = {{0},{1,2},{3,4}}
teste(2,4)
true
teste(0,4)
false

Figure 1: Exemplo da aplicação

2 O Algoritmo

O Algoritmo foi desenvolvido em linguagem C.
Inicialmente foi criado um vetor para representar o conjunto que irá conter os subconjuntos de 0 até N, cada subconjunto nesse vetor é do tipo "no". O tipo "no" foi uma estrutura criada para facilitar a manipulação e a realização das funções.
Veja seu conteúdo:

```
typedef struct no_ no;

struct no_ {
   int valor;
   int prox;
   no* inicio;
   no* anterior;
   int grupo;
   int conjunto[TAMANHO];
};
```

Figure 2: Conteúdo da estrutura no

Posteriormente você entenderá o porque de cada variável dessa estrutura e onde ela é utilizada no código. Relembrando, no código tem-se um vetor do tipo "no" de tamanho pré-definido no início. Nota-se então, que cada elemento desse vetor possui a estrutura representada acima.
A função main é basicamente dividida em quatro funções. São elas:

1. preenche( ) - responsável por setar os valores iniciais em cada no do vetor (C).
2. juntar( ) - faz a operação de união dos subconjuntos de (C).
3. teste( ) - printa "True" ou "False" se determinados números estiverem ou não no mesmo conjunto.
4. desaloca( ) - libera as estruturas alocadas na memória.

Será explicado cada função separadamente para que por fim possa se entender o algoritmo por inteiro.

Veja abaixo o código da função preenche:

```
void preenche() {
   for (int i = 0; i <= TAMANHO; i++)
{
      C[i] = malloc(sizeof (no));
      C[i]->valor = i;
      C[i]->grupo = i;
      C[i]->prox = 0;
      C[i]->inicio = NULL;
      C[i]->anterior = NULL;
}
```

Figure 3: Implementação da função preenche

Contém um laço de repetição que preenche cada "no" do conjunto (C). Esse laço é limitado pelo "tamanho" que no caso é equivalente a N, o "tamanho" é definido no ínicio do código. Todos os valores setados nessa etapa em cada "no" serão utilizados como auxiliares nas funções juntar e teste.
A função juntar dentre as outras é a mais complexa. Observe que há quatro possibilidades em relação aos parâmetros "a" e "b":

1. "a" pertence a um subconjunto com um elemento (ele mesmo)
   "b" pertence a um subconjunto com um elemento (ele mesmo)

2. "a" pertence a um subconjunto com mais de um elemento
   "b" pertence a um subconjunto com um elemento (ele mesmo)

3. "a" pertence a um subconjunto com um elemento (ele mesmo)
   "b" pertence a um subconjunto com mais de um elemento

4. "a" pertence a um subconjunto com mais de um elemento
   "b" pertence a um subconjunto com mais de um elemento

Cada parte de código que será mostrado a seguir está contido dentro da função juntar, então no caso se unir todas essas partes você terá o código completo dessa função.
Abaixo tem-se uma condição que verifica se o prox de (C) na posição "a" e "b" é igual a 0, se caso for verdade significa que eles pertencem a um subconjunto com um elemento (ele mesmo).

```
if (C[a]->prox == 0 && C[b]->prox == 0)
{

      C[b]->grupo = C[a]->grupo;
      C[a]->inicio = C[a];
      C[b]->inicio = C[a];
      C[a]->prox = 1;
      C[b]->prox = 1;

}
```

Figure 4: Primeira parte da função juntar

Se caso eles estiverem em um subconjunto sozinhos, as seguintes alterações serão feitas:

1. O grupo de (C) na posição "b" recebe o grupo de (C) na posição "a", simbolizando que eles pertencem ao mesmo grupo agora.

2. O ínicio de (C) na posição "a" e de (C) na posição "b" recebem (C) na posição "a".

3. O prox de (C) na posição "a" e de (C) na posição "b" recebem 1 para simbolizar que eles pertencem a um subconjunto com mais de um elemento.

Abaixo tem-se uma condição que verifica se o prox de (C) na posição "a" é igual a 1 e se o prox de (C) na posição "b" é igual a 0, se caso for verdade significa que "a" pertence a um subconjunto com mais elementos e "b" pertence a um conjunto com um elemento (ele mesmo).

```
if (C[a]->prox == 1 && C[b]->prox == 0)
{

      C[b]->grupo = C[a]->grupo;
      C[b]->inicio = C[a]->inicio;
      C[b]->prox = 1;

}
```

Figure 5: Segunda parte da função juntar

Se caso "a" estiver em um subconjunto com mais elementos e "b" em um subconjunto sozinho, as seguintes alterações serão feitas:

1. O grupo de (C) na posição "b" recebe o grupo de (C) na posição "a", simbolizando que estão no mesmo grupo.

2. O inicio de (C) na posição "b" recebe o inicio de (C) na posição "a".

3. O prox de (C) na posição "b" recebe 1, que diz que ele pertence a um subconjunto com mais de um elemento.

Abaixo tem-se uma condição que verifica se o prox de (C) na posição "a" é igual a 0 e se o prox de (C) na posição "b" é igual a 1, se caso for verdade significa que "a" pertence a um subconjunto com um elemento (ele mesmo) e "b" pertence a um conjunto com mais de um elemento.

```
if (C[a]->prox == 0 && C[b]->prox == 1)
{

        C[a]->grupo = C[b]->grupo;
        C[a]->inicio = C[b]->inicio;
        C[a]->prox = 1;

}
```

Figure 6: Terceira parte da função juntar

Nota-se que essa condição faz as mesmas alterações que a segunda condição, com a diferença de que na onde é (C) na posição "a" é trocado por (C) na posição "b" e vice versa.
Por fim, temos a quarta condição que verifica se (C) na posição "a" e na posição "b" são iguais a 1, isso significa que os dois pertencem a subconjuntos com mais de um elemento.

```
if (C[a]->prox == 1 && C[b]->prox == 1) {

  C[a]->inicio->conjunto[C[b]->grupo] = 1;
  C[b]->inicio->conjunto[C[a]->grupo] = 1;
  C[a]->inicio->anterior = C[b]->inicio;
  C[b]->inicio->anterior = C[a]->inicio;

  continua...
}
```

Figure 7: Quarta parte da função juntar

Se a condição for atendida as seguintes modificações serão feitas:

1. O conjunto na posição "b" do início do (C) na posição "a" recebe 1.
2. O conjunto na posição "a" do início do (C) na posição "b" recebe 1.
3. O anterior do início do (C) na posição "a" recebe o inicio de (C) na posição "b".
4. O anterior do início do (C) na posição "b" recebe o início de (C) na posição "a".

Essas alterações são feitas para auxiliarem na função teste. Nesse quarto caso tratamos de dois subconjuntos com mais de um elemento, utilizando os conjuntos dos inícios seria possível testar qualquer união entre dois subconjuntos mas se esses já estiverem unidos com algum outro subconjunto irá dar erro, e é nesse caso que a variável anterior auxilia. Suponha que você uniu ((5 e 2) unido com (6 e 3)) e tenha unido (10 e 9) separadamente. você faz a união de (6 e 9), se você testar (5 e 10) retornará "falso" justamente por não ter um controle na função juntar de quando se une um conjunto que já está unido com outro conjunto. A solução é dividida basicamente em três condicionais que manipulam a variável anterior. A solução é dividida em 3 condicionais, mas será demonstrado apenas uma já que as outras são relacionadas.

Veja abaixo a solução:

```
if (C[a]->prox == 1 && C[b]->prox == 1) {
 continua...

 if(C[a]->inicio->anterior != NULL &&
   C[b]->inicio->anterior != NULL) {
     C[a]->inicio->anterior->conjunto[C[b]->grupo] = 1
     C[b]->inicio->anterior->conjunto[C[a]->grupo] = 1
  }

}
```

Figure 8: Quarta parte da função juntar - 1ª Condicional

É feito uma condicional que checa se o anterior de "a" e "b" é diferente de NULL para que não dê falha de segmentação, as outras duas condicionais fazem o mesmo só que uma checa se o anterior de "a" é diferente de NULL e "b" é igual a NULL e a terceira condicional faz o contrário da segunda, por isso não houve necessidade delas serem representadas. Basicamente essa condicional atribui no conjunto anterior do início de cada variável o valor 1, de maneira que no teste possa se provar que um conjunto que já foi unido com outro anteriormente, ao ser unido novamente não apresente erro ao testar algum número de qualquer conjunto que esteja unido com ele.
A função teste é dividida em quatro partes. São elas:

1. Verificar se o grupo de "a" é igual a "b", se for verdade printar "True".
2. Se caso o grupo for diferente, ele verifica se o conjunto na posição "b" do início de (C) na posição "a" é igual a 1 ou vice versa, se for verdade printar "True".
3. Se caso der falso as outras duas condições ele procura no conjunto do anterior do inicio.
4. Se nenhuma das anteriores forem verdadeiras, printar "False".

Abaixo temos a primeira parte da função teste.

```
if (C[a]->grupo == C[b]->grupo) {

        printf("\nTrue");
        return;

}
```

Figure 9: Primeira parte da função teste

Simplesmente verifica se o grupo de a é igual ao grupo de b. Abaixo temos a segunda parte da função teste.

```
if(C[a]->inicio->conjunto[C[b]->grupo] == 1)
{
        printf("\nTrue");
        return;

}
```

Figure 10: Segunda parte da função teste

Suponha que tem-se dois conjuntos, (2 e 3) e (5 e 6) se for pedido para juntar o 2 e o 6 ele irá mudar no vetor conjunto dos respectivos inícios de cada grupo para 1 (o de "a" na posição "b" e o de "b" na posição "a"), o que estamos fazendo é verificando se nessa posição desse vetor temos 1. Essa verificação é verdadeira quando acontecer de ser juntado conjuntos de mais elementos (quarta parte da função juntar).

A terceira parte é dividida em três condicionais igual foi feito na quarta parte da função juntar, como feito anteriormente, será mostrado apenas uma condição já que as outras possuem praticamente a mesma lógica. Veja:

```
if(C[a]->anterior != NULL &&
   C[b]->anterior != NULL) {

   if(C[a]->inicio->anterior->conjunto[C[b]->
grupo] == 1 || C[a]->inicio->anterior->
conjunto[C[b]->grupo] == 1) {
        printf("\nTrue");
        return;
    }
}
```

Figure 11: Terceira parte da função teste

Essa parte é necessária para quando se tem união de um ou dois conjuntos que anteriormente foram unidos com outro conjunto. Da mesma forma que na função juntar, essas três condicionais da função teste só alteram quem é igual ou diferente que NULL, e nos casos que alguma variável anterior é igual a NULL, é feito a atribuição apenas na variável anterior diferente de NULL, para que não apresente erro.
Tem-se no final a função desaloca, que fica responsável por dar um free em cada estrutura alocada anteriormente. Ela foi implementada da seguinte forma:

```
desaloca( ) {

    for(int i = 0; i < TAMANHO; i++)
    {
        free(C[i]);
    }

}
```

Figure 12: Função desaloca

3 Conclusão

Por fim, como foi visto o Algoritmo desenvolvido acessa diretamente as posições do vetor que é preciso para resolver cada caso possível nas funções. Tendo isso em vista, a complexidade é igual a O(1) porém esse Algoritmo ainda possui um problema. Pelo fato do vetor ser não dinâmico, dependendo do tamanho de N vai chegar uma hora que o programa não vai conseguir alocar tanto espaço na memória como conseguiria com alguma estrutura dinâmica, e assim o programa não irá funcionar.

O código completo do Algoritmo pode ser encontrado através do seguinte link: `https://github.com/malu0908/Uniao_Conjuntos`