Relatório do EP2 de MAC0323

Algoritmos e Estruturas de Dados II

Grafos Legais

Aluno Heitor Barroso Cavalcante - N°USP: 12566101

Professor Carlos Eduardo Ferreira

Monitor Thiago Oliveira

22 de Maio de 2022

Conteúdo

Relatório do EP3 de MAC0323

1 Introdução

2 Processo de desenvolvimento do Exercício Programa

Para fazer esse EP, me deparei com um dilema inicial. Como o enunciado pedia que fizéssemos geradores de grafos de palavras, pensei que fazer uma estrutura de grafo o mais genérica possível seria o ideal. Nesse sentido, pensei em fazer com que fosse possível — utilizando templates do C++—, de fato criar grafos de palavras (strings) ou de qualquer outro tipo que o usuário decidisse. Para fazer isso, me inspirei na estrutura de lista de adjacências implementada com um vetor de listas ligadas que foi mostrada em aula pelo professor. Desse modo, tentando seguir essa ideia, implementei a estrutura que representa da seguinte maneira:

```
map<Item, vector<Item>>
```

Isto é, utilizando um `map` (Tabela de Símbolos da STL do C++), que atua como o vetor, de `vector`'s, que atuam como as listas ligadas. Contudo, como nem tudo são rosas, a performance das operações e da própria produção desses grafos genéricos ficou prejudicada, o que será mostrado nos testes adiante.

2.1 Construção da estrutura para representar o grafo

Como dito anteriormente, a ideia de utilizar um `map` de `vector`'s para representar os grafos nesse Exercício Programa foi inspirada na estrutura da lista de adjacências que consiste em um vetor de listas ligadas. Nesse sentido, cada `Key` do `map` é um vértice do grafo. O `Value` de cada `Key` é um `vector` o qual contém os vértices que são adjacentes ao vértice que está na `Key`.

2.2 Problema no input

Como a entrada do EP para construir o grafo consiste nas arestas desse grafo (pares de vértices), me deparei com um problema, que seria colocar no grafo os vértices desconectados dos demais. Caso estivesse utilizando uma estrutura de dados padrão para representar um grafo (lista de adjacências ou matriz de adjacências, por exemplo), não teria esse problema, pois, nesse

caso, os vértices correspondem à posições na lista ou matriz. Logo, ao declarar essa lista ou matriz de tamanho tal que contemplasse o total de vértices informado pelo usuário, todos os vértices já estariam sendo declarados implicitamente.

Contudo, no caso da minha implementação de um grafo genérico, — considerando um grafo de palavras, por exemplo — se um vértice está desconectado dos demais, ou seja, não é informado na entrada do usuário (que insere as arestas) não é possível saber qual seria esse vértice. Se fosse um grafo padrão, de inteiros, o vértice simplesmente seria um dos números restantes, mas no caso de palavras, por exemplo, não é possível definir qual seriam os vértices restantes.

Ao lidar com esse problema, segui dois caminhos diferentes no EP. O primeiro foi manter o input do usuário como o enunciado propunha. Mas, nesse caso, os vértices desconectados dos outros, de fato, não existem para a estrutura de grafo codificada.

A outra maneira de resolver esse problema foi desenvolver um outro tipo de input que leva em consideração essa questão.

2.3 Alteração do input

A alteração no input do grafo foi bem simples. Simplesmente, fiz um laço de repetição `while` que recebe os vértices que ficaram faltando enquanto o número de vértices adicionados pela estrutura for menor que o número de vértices informados pelo usuário. Nesse sentido, ao efetuar, simplesmente `adjacencias[vertice`, o map "`adjacencias`" cria uma chave "`vertice`" com seu `Value` sendo padrão. Ou seja, com seu `Value` sendo um `vector` vazio.

2.4 Distâncias

Para descobrir as distâncias de um vértice do grafo para os demais, utilizei um algoritmo baseado em busca em largura (bfs). Além disso, foram essenciais uma fila, e (devido à implementação genérica) dois `maps`:

- `map<Item, bool>`
  As `Key`'s são do tipo `Item` (string) e `Value`'s são booleanos, indicando se um vértice já foi visitado ou não.

- `map<Item, int>`
  As `Key`'s são do tipo `Item` (string) e `Value`'s são inteiros, indicando a distância de cada vértice do grafo ao vértice de partida.

Também fiz uso de um `map` dos predecessores de cada vértice, caso quisesse saber os caminhos, mas como essa funcionalidade não foi implementada, essa estrutura ficou um pouco avulsa no código.

Desse modo, o algoritmo (em um pseudocódigo para facilitar a compreensão) é tal que:

```
bfs(Vertice verticeDeOrigem){
    Dict<Vertice, bool> marked
    Dict<Vertice, int> dist
    Fila<Vertices> fila
    marked[verticeDeOrigem] = true
    dist[verticeDeOrigem] = 0
    fila.push(verticeDeOrigem)
    enquanto(!fila.vazia()){
        Vertice u = fila.pop()
        para cada Vertice v na lista de adjacencias de u{
            se(!marked[v]){// o vertice ainda nao foi visitado
                marked[v] = true
                dist[v] = dist[u] + 1
                fila.push(v)
            }
        }
    }
    return dist
}
```

Como essa função retorna um `map` das distâncias, é fácil extraí-las e saber as distâncias requisitadas pelo Exercício Programa.

2.5 Componentes Conexas

Para saber sobre as componentes conexas dos grafos, fiz uma função baseada em busca em profundidade (dfs). Para isso, utilizei uma função de busca em profundidade recursiva (`dfsR()`) e outra que a chama. Além disso, utilizei, novamente, outros dois `maps`.

- `map<Item, string> marked`
  As `Key`'s são do tipo `Item` (string) e `Value`'s são booleanos, indicando se um vértice já foi visitado ou não.

- `map<Item, int> comp`
  As `Key`'s são do tipo `Item` (string) e `Value`'s são inteiros, indicando a qual componente conexa cada vértice pertence.

Desse modo, o algoritmo (em pseudocódigo) é da seguinte forma:

```
comConexa(){
    int c = 0
    Dict<Vertice, bool> marked
    Dict<Vertice, int> comp
    para cada Vertice u no grafo{
        se(!marked[u]){ // o vertice ainda nao foi visitado
            dfsR(u)
            para cada Vertice v no grafo{
                se(marked[v] e comp[v] == -1){ // (1)
                    comp[v] = c
                }
            }
            c++
        }
    }
}
```

Esclarecimento sobre `(1)`:

Se o vértice já foi visitado e ainda não pertence a nenhuma componente, colocar em uma componente.

Isso funciona pois `dfsR(u)` explora as conexões de u. Então, os conectados ficam em uma mesma componente conexa.

```
dfsR(Vertice u){
    marked[u] = true
    para cada Vertice v conectado a u{
        se(!marked[v]){ // o vertice ainda nao foi visitado
            dfsR(v)
        }
    }
}
```

3 Geradores de grafos

3.1 Word Ladder

Para implementar esse modelo de gerador de grafos, me inspirei no conteúdo disponível em: https://panda.ime.usp.br

A ideia é separar as palavras que se diferenciam em apenas uma letra em "buckets" e, depois disso, unir todas essas palavras no grafo. Nesse sentido, primeiramente, recebo do usuário (via arquivo de texto) uma lista de palavras (uma por linha), todas possuindo a mesma quantidade x de letras. Depois, utilizo `ifstream`, a função `getline()` e um `vector` para ler e armazenar no `vector` todas as linhas do arquivo do texto.

Então, depois de ter essa lista de palavras, primeiro itero sobre essa lista adicionando todas as palavras no grafo — para adicionar a palavra de índice `i` basta fazer `adjacencias[palavras[i]]` — . Depois disso, itero sobre a lista outra vez, iterando sobre as letras de cada palavra. Agora, em um `map`, se um "bucket" correspondente a cada caractere da palavra não existir, eu o crio e adiciono a palavra nele, se esse "bucket" já existir, basta colocar a palvra dentro dele. Tal "bucket" é um `vector` em meu EP. Por exemplo, a palavra "bom" entrará e/ou dará origem aos seguintes "buckets":

- _om
- b_m
- bo_

Depois de já possuir o `map` de "buckets", basta iterar sobre cada "bucket no map" e se duas palavras que estejam nesse "bucket" forem diferentes, quer dizer que elas devem estar conectadas em nosso grafo.

3.2 Random Graphs

É importante ressaltar que, nesse gerador de grafo desenvolvido, em que os vértices são representados por inteiros, foi possível superar o problema apontado na seção 2.2.

Isso ocorreu porque, para contornar esse desafio, bastou (após saber a quantidade de vértices que o grafo possuirá) declarar todos eles como `Key`'s do `map` e então, caso tal vértice esteja conectado com outros, dar `push_back()`

com esses outros vértices no `vector` da `Key` em questão.

Então, esse gerador de grafos aleatórios funciona da seguinte maneira: depois de definir todos os vértices — inicialmente desconectados —, temos que, de acordo com a probabilidade de um par estar conectado (p dada pelo usuário), conectá-los. Para realizar essa tarefa, primeiramente foi necessário obter todas as combinações 2 a 2 de vértices desse grafo. Então, em cada uma dessas combinações, gerar um número aleatório entre 0 e 1. Se esse número pertencer ao intervalo $[0, p]$, então o par está conectado, caso contrário, está desconectado.

Para gerar as combinações 2 a 2, utilizei dois loops `for` encaixados. Por exemplo, querendo imprimir todas essas combinações em um grafo de 10 vértices, basta fazer:

```
for(int i = 1; i <= 10; i++)
    for(int j = i + 1; j <= 10; j++)
        cout << i << " " << j << endl;
```

Para obter um número aleatório aleatório entre 0 e 1 (`r`), basta utilizar a função `rand()` do C++, que retorna um inteiro aleatório e fazer:

```
r = (double)(rand() % 100)/(double)100;
```

Por fim, é interessante ressaltar que, ao adicionar os vértices, eu realizo um conversão de `int` para `string` usando a função `to_string()` da biblioteca de strings do C++. Tal tarefa é realizada pois, após realizar testes, percebi que o tipo string seria o mais genérico possível, visto que esse tipo seria o inserido pelo usuário na entrada padrão.

4 Testes

4.1 Componentes gigantes

A partir do gerador de grafos aleatórios, se brincarmos com o valor da probabilidade (`p`) de dois vértices estares conectados, é possível identificar a propriedade da Componente Gigante, mostrada por Erdös e Rényi. Ou seja, se $p \leq \frac{1-\epsilon}{n}$, então, com alta probabilidade as componentes terão $O(\log n)$ elementos. Já se $p \geq \frac{1+\epsilon}{n}$, surge uma componente gigante no grafo (sendo n a quantidade de vértices). Portanto, podemos observar tal propriedade nos seguintes testes:

4.1.1 $n = 100$

Para $p \leq 0.0088$, teremos componentes conexas pequenas.
Para $p \geq 0.0112$ teremos componente conexa gigante.

p	#comps. conexas	média de #elems. nas comps. conexas	#elems. na maior comp. conexa
.000880000	95	1.05263	2
.037920000	4	25	97
.074960000	1	100	100
.112000000	1	100	100

4.1.2 $n = 500$

Para $p \leq 0.00176$, teremos componentes conexas pequenas.
Para $p \geq 0.00224$ teremos componente conexa gigante.

p	#comps. conexas	média de #elems. nas comps. conexas	#elems. na maior comp. conexa
.000176000	479	1.04384	3
.007584000	14	35.7143	486
.014992000	1	500	500
.022400000	1	500	500

4.2 $n = 1000$

Para $p \leq 0.00088$, teremos componentes conexas pequenas.
Para $p \geq 0.00112$ teremos componente conexa gigante.

p	#comps. conexas	média de #elems. nas comps. conexas	#elems. na maior comp. conexa
.000088000	969	1.03199	3
.003792000	18	55.5556	982
.007496000	1	1000	1000
.011200000	1	1000	1000

4.2.1 $n = 5000$

Para $p \leq 0.000176$, teremos componentes conexas pequenas.
Para $p \geq 0.000224$ teremos componente conexa gigante.

p	#comps. conexas	média de #elems. nas comps. conexas	#elems. na maior comp. conexa
.000017600	4790	1.04384	4
000758400	105	47.619	4896
.001499200	2	2500	4999
.002240000	1	5000	5000

4.3 $n = 10000$

Para $p \leq 0.000088$, teremos componentes conexas pequenas.
Para $p \geq 0.000112$ teremos componente conexa gigante.

p	#comps. conexas	média de #elems. nas comps. conexas	#elems. na maior comp. conexa
.000008800	9579	1.04395	4
.000379200	224	44.6429	9765
.000749600	10	1000	9991
.001120000	1	10000	10000

Para fazer esses testes, fiz um script de bash, com a seguinte lógica, delimitei números de vértices que achei razoáveis: 100, 500, 1000, 5000 e 10000. Depois disso, para cada número de vértices, calculei a probabilidade para ter componentes pequenas ("$inferior$") e dividi por 10, calculei a probabilidade para ter componente gigante ("$superior$") e multipliquei por 10. Depois peguei esse intervalo e dividi por 3. Então, ficamos com 4 probabilidades para testar:

- $inferior$
- $inferior + \left(\frac{superior - inferior}{3}\right)$
- $inferior + 2\left(\frac{superior - inferior}{3}\right)$
- $superior$

Nesses testes, apesar de notar um pouco de lentidão, não vi necessidade de explicitar o tempo gasto. Isso será feito na próxima seção. De teste dos "seis graus de separação".

4.4 Seis graus de separação

Para realizar o teste dessa propriedade — que consiste na ideia de que todas as pessoas estão a seis ou menos conexões sociais de qualquer outra — fiz um grafo aleatório contendo um número n de vértices. Então, obtenho a maior componente conexa. Essa componente será o "universo" para este teste. Depois disso, para cada vértice, obtenho a distância dele para os demais, tiro a média dessas distâncias. Por fim, calculo a média das médias. Note que esse processo é custoso, então, como a estrutura genérica que implementei para construir o grafo "usando tabelas de símbolos" não é otimizada, tive tempos bem ruins ao efetuar os testes.

Os testes realizados, consistem em uma quantidade esperada de "conhecidos" que uma pessoa tem (conexões de um vértice). Ou seja, se em um grafo de 1000 vértices, esperamos que cada pessoa conheça outras 10, então, a probabilidade de um vértice estar conectado a outro deve ser de $\frac{10}{1000}$, pois o valor esperado de conexões nesse grafo será 10. Portanto, escolhi, arbitrariamente, construir grafos de 1000, 5000 e 10000 vértices e utilizando o conceito de uma pessoa possuir 5, 30 e 55 conexões. Assim, os resultados dos testes corroboraram a propriedade comentada, mas como dito anteriormente, obtive tempos de execução ruins.

4.4.1 $n = 1000$

Conexões esperadas	tempo	distância média
5	0m6.556s	4.48774
30	0m24.830s	2.37031
55	0m41.589s	1.99215

4.4.2 $n = 5000$

Conexões esperadas	tempo	distância média
5	3m50.266s	5.45423
30	13m54.892s	2.83477
55	21m43.352s	2.53035

4.4.3 $n = 10000$

Conexões esperadas	tempo	distância média
5	15m15.112s	5.89683
30	53m37.558s	2.9862
55	90m23.507s	2.73004

5 Conclusão

Ao fim deste relatório, concluo que consegui realizar o quê foi proposto no enunciado do Exercício Programa. Contudo, não posso afirmar que a maneira com que realizei as tarefas propostas foi a maneira esperada por quem elaborou o enunciado. Digo isso pois, para mim, o enunciado ficou muito aberto, sem delimitar objetivamente como o trabalho deveria ser feito. Nesse sentido, creio que o que desencadeou toda a ambiguidade do enunciado foi a questão dos grafos de palavras. Se tivesse ficado explícito que a ideia era (ou não) recebermos sempre inteiros para construir os vértices, a idealização de um grafo genérico seria confirmada ou descartada facilmente.

Além disso, outro ponto que ficou muito em aberto para mim foi a questão dos geradores de grafos. Não está claro no enunciado do EP se deveríamos implementar geradores de grafos ou geradores de entradas para o EP. Da maneira como fiz, a qual não consigo afirmar se era a esperada, o próprio programa gera seu grafo — sem requisitar entrada ao usuário — . Desse modo, fiquei um pouco inseguro quanto aos pontos mencionados.

Ademais, acredito que consegui exercitar conceitos importantes relativos ao conteúdo de grafos ao longo da construção desse Exercício Programa, ressaltando busca em largura (bfs) e em profundidade (dfs). Outro ponto interessante foi exercitar o uso de dicionários (`map`) e de `vector` em C++.