Relatório - Parte 2 - Teoria dos Grafos (COS242)

Thiago Barroso Perrotta e Guilherme Sales

6 de Novembro de 2013

Professores: Daniel Ratton Figueiredo e Ricardo Marroquim

1 Do objetivo

O objetivo desta segunda parte do trabalho foi de estender a primeira parte da biblioteca, acrescentando suporte a pesos nas arestas dos grafos, além de incluir diversos algoritmos, tais como Dijkstra – para o caminho mínimo, Prim – para a árvore geradora mínima (MST) e Floyd Warshall (*APSP* - All Pairs Shortest Paths) – para calcular a distância média do grafo.

Além disso, a biblioteca continuou a ser compatível com a primeira parte, podendo ser facilmente estendida ou utilizada em outros projetos.

2 Das decisões de projeto

Além do mais, utilizamos um modelo de desenvolvimento orientado a testes (TDD - Test-driven development), o que significa que adaptamos todo o desenvolvimento de novas *features* do programa da seguinte maneira: primeiro criamos o stub (assinatura; protótipo; header) das funções, assim como uma breve documentação de seu propósito; depois criamos os códigos de teste; e então, finalmente, vem a implementação. Esse modelo se mostrou extremamente útil ao longo do desenvolvimento, pois facilitou a administração do código e assegurou que novas linhas de código não quebrassem o trabalho anterior sem que isso fosse percebido.

Optamos por recomeçar o trabalho todo praticamente do zero, abandonando o modelo anterior de desenvolvimento e de organização dos arquivos. Agora passamos a utilizar programação orientada a objetos que a linguagem C++ fornece, utilizando de conceitos tais como herança, encapsulamento e polimorfismo para tornar a biblioteca mais fácil de manter para seus desenvolvedores, e mais amigável para seus usuários.

O framework de testes escolhido foi o **CppUnit**, utilizado de forma integrada à IDE (Integrated Development Environment) **NetBeans.** Utilizamos fortemente as facilidades de refatoração de código da mesma. Continuamos utilizando **git** como sistema de controle de versão, e ele se mostrou bem mais útil nessa segunda parte. Passamos a utilizar alguns aspectos mais avançados do mesmo, tais como **branches, diff** e (hard) resets.

Para a modelagem das classes de C++, fomos fortemente influenciados pelos conceitos de **UML**. Em particular, utilizamos o software **Umbrello**, do ambiente KDE, para representar as relações entre as classes. Na verdade, testamos vários softwares de UML antes de tomar essa decisão, tais como o ArgoUML, Dia e o Microsoft Visio 2010 e 2013, mas concluímos que o Umbrello era o melhor que atendia aos nossos requisitos, que eram essencialmente simplicidade com clareza.

Ao final do trabalho, contabilizamos em torno de 60 testes realizados (cada teste é uma função que procura cobrir diversos aspectos de cada classe). Acreditamos que esses testes contribuíram de forma significativa para reduzir diversos bugs mas, assim como todo software, não podemos garantir que nossa implementação é 100% bug-free. Mas, quanto mais testes, menor é a chance de algum bug passar despercebido.

Como *profiler,* pensamos em utilizar inicialmente o **Valgrind** para medir a memória de pico em algumas etapas das execuções dos programas, mas vimos que isso não era absolutamente necessário, já que utilizamos lista de adjacência e os grafos não eram densos demais. Além disso, para medir o tempo de execução dos programas, começamos utilizando a rotina clock() da biblioteca **time**.

Entretanto, tivemos alguns problemas com ela. Então passamos a utilizar a ferramenta **time** do Linux de forma integrada ao Netbeans, assim como a classe **progress_timer** da biblioteca Boost. As duas ferramentas se mostraram concordantes e confiáveis.

3 Da estrutura do projeto

O projeto está dividido em duas partes.

A primeira parte contém as classes **Graph** (Graph.cpp e Graph.h), **GraphList** (GraphList.cpp e GraphList.h) e **GraphMatrix** (GraphMatrix.cpp e GraphMatrix.h). As duas últimas são derivadas da primeira. A primeira é uma classe abstrata, o que significa que ela não pode ser instanciada. Ela contém basicamente todos os atributos e métodos que são gerais a grafos e que não dependem de maneira específica de sua representação (por exemplo, os graus dos vértices, assim como o número de nós e de arestas do grafo). Já as suas classes derivadas contém o que depende da representação do grafo (por exemplo, a forma na qual as arestas são guardadas).

Já a segunda parte contempla essencialmente todos os algoritmos específicos da biblioteca. Dentre elas (todas incluem um arquivo .h e um .cpp associados):

- Bfs (breadth-first search)
- Dfs (depth-first search)
- Dijkstra (caminho mínimo)
- FloydWarshall (distância média)
- Mst (Prim: árvore geradora mínima)
- InputHandler (para ler o grafo a partir de um arquivo)

Além disso existe um conjunto de funções que utilizamos diversas vezes durante os testes do programa; por isso resolvemos criar um arquivo **Utilities** para armazená-las. São funções com propósitos do tipo: comparar dois vetores, comparar dois arquivos de texto, imprimir o conteúdo de um vetor, etc.

De modo similar, os **headers** que eram utilizados com frequências (como vector, fstream, string) foram reunidos principalmente em **Graph.h** e num novo arquivo **Include.h**.

4 Estudo de Caso 1

4.1 Distância até o vértice 1

Rodamos cada programa somente **uma única vez**, aplicando o algoritmo de **Dijkstra** no vértice de origem 1, utilizando a representação de **lista de adjacência** do grafo. A execução de cada um deles foi bastante rápida, bastando uns poucos segundos. Nosso algoritmo de Dijkstra utilizou o heap que implementamos manualmente.

Grafo	Nó	Distância	Menor Caminho
1	10	19	$1, 100, 17, 8, 67, 10$
1	20	24	$1, 100, 17, 8, 73, 74, 19, 20$
1	30	21	$1, 100, 17, 8, 67, 26, 31, 30$
1	40	19	$1, 100, 17, 8, 40$
1	50	23	$1, 100, 17, 8, 73, 50$
2	10	2	$1, 52, 10$
2	20	3	$1, 144, 20$
2	30	2	$1, 4, 30$
2	40	3	$1, 331, 40$
2	50	3	$1, 337, 41, 50$
3	10	26	$1, 2, 3, 7739, 3782, 8405, 7014, 10$
3	20	38	$1, 2, 3, 7739, 3782, 3259, 9657, 1486, 4206, 2783, 19, 20$
3	30	34	$1, 2, 3, 7739, 7738, 7080, 995, 4307, 598, 599, 29, 30$
3	40	30	$1, 2, 3, 7739, 7738, 7080, 995, 3486, 877, 40$
3	50	27	$1, 2, 3, 7463, 8774, 6577, 7713, 4461, 50$
4	10	27	$1, 2, 40954, 6638, 24142, 5438, 5437, 30631, 18082, 9, 10$
4	20	17	$1, 2, 40954, 9187, 44780, 14530, 12285, 30632, 34758, 38687, 20$
4	30	40	$1, 2, 40954, 9187, 46430, 48174, 42945, 32, 31, 30$
4	40	21	$1, 2, 40954, 6638, 28947, 16279, 19115, 36243, 13509, 7746, 39, 40$
4	50	18	$1, 2, 40954, 46519, 11807, 5924, 5925, 8120, 23726, 50$
5	10	56	$1, 100000, 99999, 99998, 92827, 24681, 49217, 86633, 23787, 19564, 82822, \ldots$
			$\ldots, 24916, 79294, 79095, 35948, 30951, 30952, 30953, 70778, 10$
5	20	105	$1, 100000, 99999, 99998, 5098, 15254, 23575, 7300, 96904, 96903, 42404, 60810, \ldots$
			$\ldots, 60811, 60812, 89262, 41578, 12, 13, 14, 15, 16, 17, 18, 19, 20$
5	30	46	$1, 100000, 99999, 99998, 5098, 15254, 23575, 7300, 94647, 59430, 48919, 31, 30$
5	40	79	$1, 100000, 99999, 99998, 92827, 24681, 49217, 86633, 23787, \ldots$
			$\ldots, 19564, 72304, 78297, 7874, 74107, 35, 36, 37, 38, 39, 40$
5	50	44	$1, 100000, 99999, 99998, 92827, 24681, 49217, 21224, 14174, 99060, 72224, 50$

4.2 Árvore geradora mínima (MST)

Utilizamos a representação de **lista de adjacência** do grafo, rodando o algoritmo de **Prim** para encontrar a MST. Bastou executar uma única vez cada programa. Nessa parte o tempo de execução foi mais variado; em particular, o grafo 2 foi o mais demorado, e o 1 o mais rápido. Medimos o tempo internamente, no próprio código, utilizando a biblioteca padrão **time** e sua função clock().

Grafo	Tempo (s)	Custo da MST
1	$0,001$	336
2	$10,89$	999
3	$0,30$	31947
4	$1,21$	216236
5	$1,18$	608677

Notamos que o nosso heap implementado manualmente não gerou os resultados corretos para essa parte. Por exemplo, no grafo 5 tínhamos encontrado 60965, um valor ligeiramente maior do que o correto. O mesmo se repetiu para os grafos 3 e 4 (mas o 1 e o 2 ficaram corretos). Acreditamos que isso tem a ver com o fato de que não podemos atualizar algumas posições do heap além de (exceto) o topo. Para Dijkstra não houve esse problema.

4.3 Distância Média

Para calcular a distância média utilizamos o algoritmo de **Dijkstra** (SSSP) para os grafos 1 e 3, e o de **Floyd-Warshall** (APSP) para o grafo 2, utilizando a representação de **lista de adjacência** para todos os grafos.

Para os grafos 4 e 5, optamos pelo algoritmo de Dijkstra pois o algoritmo de Floyd, com complexidade $O(n^3)$, levaria tempo demais para ser executado (note que esses grafos são os que possuem o maior número de nós).

Em particular, para esses dois grafos, também acrescentamos uma operação adicional: a cada iteração o programa escrevia para um arquivo de saída, o qual atuava como um log. A razão disso é que previmos que o algoritmo demoraria várias horas para ser executado; portanto, o log foi a forma encontrada para monitorar tanto a) qual o progresso de execução do programa e b) se ele ainda estava sendo executado OK, ou se alguma exceção ou erro foi lançado ao longo de sua execução. Este foi o item mais demorado dentre os estudos de caso.

Grafo	Distância Média	Tempo
1	13.021	0.02 s
2	2.0859	9.12 s
3	16.0141	10 min 30 s
4	24.6328	2h 11 min
5	57.165	4,32 h

5 Estudo de Caso 2

5.1 Distância e caminho mínimo até Dijkstra

Usamos a representação de **lista de adjacência**, rodando o **algoritmo de Dijkstra**. **OBS.:** Edsger W. Dijkstra ID = 2722.

Colaborador	ID	Distância	Caminho Mínimo
Alan M. Turing	11365	∞	não existe
J. B. Kruskal	471365	3.48037	2722, 9490, 7200, 10343, 646765, 490368, 10746, 3655, 471365
Jon M. Kleinberg	5709	2.71595	2722, 217250, 11456, 768, 11448, 4704, ...
			..., 585432, 8629, 11834, 9608, 5709
Éva Tardos	11386	2.75351	2722, 217250, 11456, 768, 6479, 8528, 10572,...
			..., 357587, 11649, 3694, 318911, 11386
Daniel R. Figueiredo	343930	2.94283	2722, 9490, 7200, 391667, 371226, 4379, 68773, 11466, 343930
Ricardo Marroquim	309497	2.80935	2722, 9490, 408912, 2932, 371, 10304, 496285, 309497

- Tempo de execução médio:
  - Sem a flag de compilação `-O3` (debug): 32–33s
  - Com `-O3` (release): 9–10s

Vimos que o grafo é desconexo, já que não existe um caminho entre os célebres pesquisadores Turing e Dijkstra. Notamos que a otimização do compilador realmente faz **bastante** diferença em tempo de execução.

5.2 Árvore geradora mínima

Calculamos todos os resultados através do algoritmo de Prim, utilizando uma heap de mínimo. Geramos, como uma etapa intermediária, o arquivo de saída da MST, para apenas depois operarmos sobre ele para responder às questões pedidas.

- Representação utilizada: lista de adjacência;
- Tempo de execução médio para encontrar a MST: 21 s (usando a `priority_queue` da STL);
- Tempo de execução médio para obter as respostas da MST: 0,64 segundo.

1. Vizinhos de Dijkstra: 217250 (A.J.M. van Gasteren), 669617 (Carel S.Scholten), 672390 (W.Heise)

2. Vértices de maior grau: 473131 (Wei Li, grau 171), 268534 (Wei Wang, grau 146), 88809 (Wei Zhang, grau 143)

3. .

   (a) Vizinhos de Figueiredo: 255335 (Alexandre A. Santos), 337713 (André C. Pinho), 11466 (Donald F. Towsley)

   (b) Vizinhos de Marroquim: 496285 (Claudio Esperança)

   (c) Vizinhos em comum: Na MST encontrada não possuem vizinhos em comum (na verdade, não possuem vizinhos em comum nem no grafo da rede)