Teoria dos Grafos

Trabalho da Disciplina - Parte 2

Engenharia de Computação e Informação UFRJ

Carolina Santiago de Medeiros 122053305
Zuilho Rodrigues Castro Segundo 122064877

Link do Repositório: `https://https://github.com/ZuilhoSe/GraphLibrary`

1 Introdução

Este projeto é uma continuação do primeiro, e tem como objetivo aprimorar uma biblioteca C++ que lida com grafos, desta vez com pesos, permitindo a leitura de grafos a partir de arquivos, a conversão para o formato de vetor de adjacência. Assim como na versão prévia, existe a possibilidade de realizar buscas em largura ou profundidade, porém com a adição do algoritmo de Dijkstra para o cálculo de distâncias. O presente trabalho também visa fazer uma análise detalhada de estudos de caso, baseados nas situações-problema apresentadas e seus respectivos grafos.

2 Análise do Código

2.1 Entrada e Saída

2.2 Representação dos Grafos

A biblioteca utiliza uma representação de grafo baseada em lista de adjacência, que é uma abordagem eficiente para grafos não direcionados. Os grafos são armazenados como um vetor de listas de adjacência, onde cada vértice é representado como um elemento no vetor e suas arestas adjacentes são armazenadas na lista associada a esse vértice. A representação de grafos com pesos é alcançada associando um valor numérico (peso) a cada aresta na lista de adjacência.

2.3 Busca em Grafos

A biblioteca implementa duas técnicas de busca em grafos: Busca em Largura (BFS) e Busca em Profundidade (DFS). Ambas as técnicas são usadas para percorrer o grafo e explorar seus vértices e arestas. A Busca em Largura é implementada usando uma fila, enquanto a Busca em Profundidade é implementada usando uma pilha.

2.4 Distância entre vértices

A biblioteca implementa o algoritmo de Dijkstra para encontrar as distâncias mais curtas entre um vértice de origem e todos os outros vértices em um grafo ponderado, desde que não haja arestas com pesos negativos. No código, existem duas implementações distintas do algoritmo, cada uma com suas próprias características e eficiências: usando um vetor e usando um heap.

A principal diferença entre as duas implementações está na seleção eficiente do vértice com a menor estimativa de distância. A implementação com vetor requer uma busca linear para encontrar o vértice com a menor estimativa, o que resulta em uma complexidade de tempo de $O(n^2)$, onde n é o número de vértices. Em contraste, a implementação com heap permite uma seleção mais eficiente em O(log n) devido à sua estrutura de dados.

Ambas as implementações funcionam apenas quando o grafo não possui arestas com pesos negativos. Se houver arestas com pesos negativos, o algoritmo de Dijkstra não é adequado, e a biblioteca lida com essa situação informando ao usuário que o cálculo da distância não é possível nesse caso.

- Funções que implementam ambos os "Dijkstra":

```
std::vector<float> Graph::heapDijkstra(int v, bool e, std::string sFilename,
std::string sFilename2){
    if(this->hasNegativeWeight){
        cout << "Grafo possui arestas com peso negativo. Nao e possivel
        calcular a distancia." << endl;
        return std::vector<float>();
    }

    v = v - 1;
    int nV = this->getNVertices();
    std::vector<float> distance(nV, INF);
    std::vector<struct Edge>* L = this->getList();
    std::priority_queue<std::pair<float, int>,
    std::vector<std::pair<float, int>>, std::greater<std::pair<float, int>>> pq;

    std::vector<int> degree(nVertices, 0);
    std::vector<int> father(nVertices, -1);
    std::vector<int> level(nVertices, -1);

    distance[v] = 0.0f;
    father[v] = 0;
    level[v] = 0;
    degree[v] = 0;
    pq.push({distance[v], v});

    while (!pq.empty()) {
        float dist = pq.top().first;
        int u = pq.top().second;
        pq.pop();

        if (dist != distance[u]) {
            continue; // Skip if we've already found a shorter path to this vertex
        }

        for (const Edge &edge : L[u]) {
            int v = edge.dest;
            float w = edge.weight;
            degree[u]++;

            if (distance[u] + w < distance[v]) {
                distance[v] = distance[u] + w;
                father[v] = u + 1;
                level[v] = level[u] + 1;
                pq.push({distance[v], v});
            }
        }
    }

    this->nodesDegree = degree;
    this->nodesFather = father;
    this->nodesLevel = level;

    if(e){
        exportGenTreeToTxt(sFilename);
        exportDistancesToTxt(sFilename2, distance);
    }
```

```
    return distance;
}



std::vector<float> Graph::vectorDijkstra(int v){
    if(this->hasNegativeWeight){
        cout << "Grafo possui arestas com peso negativo. Nao e possivel calcular a
        ↪  distancia." << endl;
        return std::vector<float>();
    }
    v = v - 1;
    int numVertices = this->getNVertices();
    std::vector<float> distance(numVertices, INF);
    std::vector<struct Edge>* L = this->getList();

    std::vector<int> degree(nVertices, 0);
    std::vector<int> father(nVertices, -1);
    std::vector<int> level(nVertices, -1);

    std::vector<bool> visited(numVertices, false);

    distance[v] = 0.0f;
    father[v] = 0;
    level[v] = 0;
    degree[v] = 0;

    for (int i = 0; i < numVertices; i++) {
        // Find the unvisited vertex with the smallest distance
        int minVertex = -1;
        for (int v = 0; v < numVertices; v++) {
            if (!visited[v] && (minVertex == -1 || distance[v] < distance[minVertex])) {
                minVertex = v;
            }
        }

        // Mark the selected vertex as visited
        visited[minVertex] = true;

        // Update distances to its neighbors
        for (const Edge &edge : L[minVertex]) {
            int neighbor = edge.dest;
            float weight = edge.weight;

            degree[minVertex]++;

            if (!visited[neighbor] && distance[minVertex] + weight < distance[neighbor]) {
                distance[neighbor] = distance[minVertex] + weight;
                father[neighbor] = minVertex + 1;
                level[neighbor] = level[minVertex] + 1;
            }
        }
    }

    this->nodesDegree = degree;
    this->nodesFather = father;
    this->nodesLevel = level;

    exportGenTreeToTxt("../examples/graph_teste_out.txt");
```

```
return distance;}
```

3 Estudos de Caso

3.1 Tempo de Execução

Para calcular o tempo de execução das implementações do algoritmo de Dijkstra para encontrar a distância entre os vértices do grafo, foram realizados alguns testes utilizando vértices iniciais aleatórios, com k=2 (`vectorDijkstra`) e k=100 (`heapDijkstra`); "k"sendo o número de vezes que o teste foi feito. O tempo total de execução para cada implementação foi calculado através de uma média amostral. Os resultados desses testes estão apresentados na tabela a seguir:

	Tempo de Execução [ms]	
	vectorDijkstra k = 2	**heapDijkstra k = 100**
Grafo 1	2046	36
Grafo 2	12705	124
Grafo 3	202831	734
Grafo 4	20499234	9657
Grafo 5	X	53609 *k=1

Esses resultados fornecem uma perspectiva do desempenho das diferentes implementações do algoritmo de Dijkstra em relação ao tempo de execução, com base em um vértice inicial aleatório. Baseado nesses resultados, é perceptível que a implementação com heap é consideravelmente mais eficiente.

3.2 Distâncias e Caminhos Mínimos

Para calcular a distância e os caminhos mínimos entre o vértice 10 e os vértices 20, 30, 40, 50 e 60 no grafo, foi utilizado o algoritmo de Dijkstra com heap, e a função auxiliar de caminhos mínimos `exportMinPathToTxt`. Os resultados estão apresentados nas tabelas a seguir:

Grafos	**Distância Mínima**				
	(10,20)	**(10,30)**	**(10,40)**	**(10,50)**	**(10,60)**
1	2.38	1.72	2.05	1.2	1.66
2	1.81	1.72	1.92	1.57	1.58
3	0.8	0.89	0.84	0.87	0.97
4	2.66	2.02	2.52	2.69	2.41
5	17.34	17.24	17.37	15.43	17.99

Grafos	Vértices	Caminhos Mínimos
1	(10,20)	20 1034 9285 9853 4252 4627 2502 3578 8892 4746 2875 5094 2038 10
	(10,30)	30 4045 682 2658 2038 10
	(10,40)	40 1223 2195 6759 5941 682 2658 2038 10
	(10,50)	50 2527 2532 8078 3235 2258 7521 2038 10
	(10,60)	60 9082 4328 8182 3383 4333 1268 3598 2875 5094 2038 10
2	(10,20)	20 3907 23467 9448 13344 3358 8238 3448 1573 18571 15369 15491 10525 20401 10
	(10,30)	30 8688 22569 17348 12125 8863 16217 23713 22754 10
	(10,40)	40 11469 6984 12308 18966 23726 23673 3461 23713 22754 10
	(10,50)	50 14607 24612 24811 3594 14272 23978 627 8863 16217 23713 22754 10
	(10,60)	60 10430 7009 1848 10684 20167 5521 9146 13268 22754 10
3	(10,20)	20 15760 86355 16608 27694 86778 87862 68904 84450 26049 70540 41321 61279 39345 24445 3905 35293 2858 10
	(10,30)	30 26151 27642 93985 78071 42304 77935 87087 64898 53177 85187 96094 35293 2858 10
	(10,40)	40 46307 15930 19697 14327 86286 24445 3905 35293 2858 10
	(10,50)	50 19878 70948 94992 17622 82510 19040 84437 56663 10
	(10,60)	60 8718 59399 74717 79865 75728 96654 33719 66532 99253 2874 89584 24445 3905 35293 2858 10
4	(10,20)	20 807260 318070 634947 655748 663364 841402 805646 437166 430107 554617 877016 532245 10
	(10,30)	30 707581 740528 132461 417194 239369 35338 920911 183747 44636 207765 219466 694525 81769 261305 182382 873980 10
	(10,40)	40 477496 188007 386086 274827 730863 801708 277633 275098 418550 188455 557022 674971 182382 873980 10
	(10,50)	50 630598 994210 407659 472621 716970 9905 735867 349584 149827 663332 748624 382144 694525 81769 261305 182382 873980 10
	(10,60)	60 774634 822462 855615 358029 614364 188021 224210 161217 283156 883474 364950 227336 280367 27985 106317 674971 182382 873980 10
5	(10,20)	20 8885874 3908338 2031914 7155379 6766415 7788327 8790605 7103730 1295458 6528266 4615833 7622850 10
	(10,30)	30 8947758 5664093 1816279 7291082 5117196 3166991 8868678 4341576 3485759 5549785 8775277 2983314 6836812 9270546 202621 5413988 2846336 10
	(10,40)	40 9798152 4846487 9008138 7676193 3215052 5850281 660832 4105715 409605 5641294 9885733 1419794 1299954 10
	(10,50)	50 396789 6416250 9271362 700154 3724752 3716282 4276332 5368621 8451042 9927727 1299954 10
	(10,60)	60 1454010 7824884 3193827 7718765 1925405 1175531 2860740 8424191 2071841 3090234 3945175 264365 8941517 9104671 9611705 9445351 757290 7524819 1899178 1419794 1299954 10

3.3 Grafo de Rede de Colaboração entre Pesquisadores

3.3.1 Distâncias e Caminhos Mínimos entre Pesquisadores

Para calcular a distância e os caminhos mínimos entre o pesquisador Edsger W. Dijkstra e outros pesquisadores na rede de colaboração, utilizamos o algoritmo de Dijkstra. Os pesquisadores utilizados como vértice de origem são Alan M. Turing, J.B. Kruskal, Jon M. Kleinberg, Éva Tardos e Daniel R. Figueiredo. Os resultados estão apresentados nas tabelas a seguir:

	Distância Mínima
Turing	$+\infty$
Kruskal	4.21798
Kleinberg	2.81198
Tardos	2.88254
Figueiredo	3.48508

Origem	Caminhos Mínimos
Alan M. Turing	Não há caminho entre os pesquisadores!
J.B. Kruskal	J. B. Kruskal — Albert G. Greenberg — Richard E. Ladner — Amotz Bar-Noy — Baruch Schieber — Yishay Mansour — David A. McAllester — Robert Givan — Edwin K. P. Chong — Howard Jay Siegel — Dan C. Marinescu — John R. Rice — Edsger W. Dijkstra —
Jon M. Kleinberg	Jon M. Kleinberg — Prabhakar Raghavan — Eli Upfal — Franco P. Preparata — Roberto Tamassia — Michael T. Goodrich — Mikhail J. Atallah — John R. Rice — Edsger W. Dijkstra —
Éva Tardos	Éva Tardos — Zvi Galil — David Eppstein — Michael T. Goodrich — Mikhail J. Atallah — John R. Rice — Edsger W. Dijkstra —
Daniel R. Figueiredo	Daniel R. Figueiredo — Donald F. Towsley — John A. Stankovic — Krithi Ramamritham — Wei Zhao — Edwin K. P. Chong — Howard Jay Siegel — Dan C. Marinescu — John R. Rice — Edsger W. Dijkstra —