Projeto e Análise de Algoritmos **Oto Braz Assunção**
Trabalho Prático - Grafos **Data: 04/05/2018**

1 Introdução

Este relatório apresenta o processo de desenvolvimento do Trabalho Prático do segundo módulo da disciplina Projeto e Análise de Algoritmos. O problema abordado foi a identificação dos melhores corredores para alocação de uma equipe de determinada rede de comunicação nos estádios da Copa do Mundo. Estes são aqueles que terão maior tráfego dos atletas das equipes, considerando que os atletas sempre passam pelos caminhos mais curtos.

O problema foi modelado em um grafo orientado e ponderado, representado o estádio. Os vértices do grafo representam os locais dentro dos estádios, enquanto que as arestas representam os corredores por onde os atletas trafegarão.

2 Desenvolvimento

O trabalho foi desenvolvido em *C++* utilizando *Code::Blocks 16.01* como *IDE*. O processo de desenvolvimento pode ser dividido em quatro etapas: modelagem do problema, calculo das distâncias mínimas, identificação de corredores que possivelmente serão utilizados e identificação de corredores que sempre serão utilizados.

2.1 Modelando o Problema

O Estadio foi definido como a classe *Estadio.cpp apresentada a seguir:*

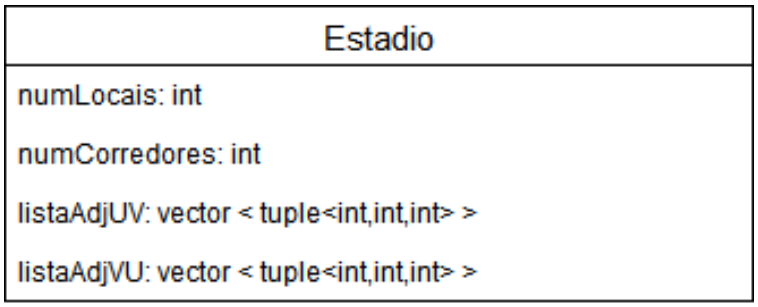

Figura 1: Classe - Estadio

O Estádio possui duas variáveis do tipo *int* para representar o número de locais (vértices) e corredores (arestas). Foi decidido utilizar listas de adjacências para representar o grafo e suas relações pois elas possuem melhor desempenho para grafos grandes e esparsos se comparadas a uma matriz de adjacência de tamanho nx. Foram utilizados *vectors* de *tuplas* para armazenamento do identificador do corredor, vértice adjacente e peso da aresta respectivamente. Duas listas foram definidas para facilitar a implementação dos algoritmos apresentados nas próximas subseções: *listaAdjUV*, representando as orientações originais,

e *listaAdjVU*, representando as orientações reversas. As duas são criadas e preenchidas ao mesmo tempo durante o processo de leitura do arquivo de entrada a fim de economizar tempo.

2.2 Calculando Distâncias Mínimas

Identificar as distâncias mínimas é o ponto central do trabalho. O restante dos algoritmos dependem diretamente desta etapa. A função implementada foi baseada no algoritmo de DIJKSTRA visto que este é o algoritmo de caminho mínimo com menor complexidade assintótica dado um grafo composto apenas por arestas de peso positivo. O Algoritmo 1 apresenta o pseudocódigo da função desenvolvida:

Algorithm 1 Two-Way Dijkstra

```
1:  function TwoWayDijkstra(s, e, d, dT, v, vT)
2:      counter ← 0
3:      Inicializar(e.numLocais, s, d, dT)
4:      q, qT ← ∅                                   ▷ Filas de prioridade
5:      for i ← 1 to e.numLocais do
6:          Q ← (i, d[i])
7:          QT ← (i, dT[i])
8:      while counter < e.numLocais and q ≠ ∅ do
9:          u ← q.pop()
10:         if d[u] = INF  or  d[u] ¡ u.d then continue
11:         for all vértice v ∈ e.AdjUV[u] do
12:             Relaxar(d, u, v, q)
13:         counter ← counter + 1
14:     counter ← 0
15:     while counter < e.numLocais and q ≠ ∅ do
16:         u ← qT.pop()
17:         if dT[u] = INF or dT[u] < u.d then continue
18:         for all vértice v ∈ e.AdjVU[u] do
19:             Relaxar(dT, u, v, qT)
20:         counter ← counter + 1
```

É TwoWayDijkstra é uma função que executa o algoritmo de DIJKSTRA em dois sentidos. Ela faz o cálculo das distâncias mínimas a partir da origem (d) e também a partir do destino (dT).

As filas de prioridade q e qT foram implementadas utilizando a estrutura de dados *priority_queue* da *STL*. Ela permite que, dada uma função de comparação, os dados sejam inseridos na fila de forma ordenada baseado nesta função. A desvantagem desta estrutura é que ela não dá suporte ao DecreaseKey, logo não é possível alterar as prioridades dos itens inseridos e reordenar a fila. Para contornar este problema, cada elemento é inserido novamente na fila caso seja necessário. Caso a distância $u.d$ do vértice u retirado da fila seja infinita ou maior que a distância mínima $d[u]$, a iteração atual é pulada.

Como definido por [1], cada elemento é considerado apenas uma vez e cada aresta é relaxada apenas uma vez. Assim, com o intuito de evitar processamento desnecessário devido as múltiplas inserções, foi utilizado um contador para manter controle sob o número de

iterações de cada *while loop*. O *loop* para quando a fila de prioridades fica vazia ou quando o contador for igual ao número de vértices, o que indica que não haverão mais arestas a serem relaxadas.

2.3 Encontrando Corredores que Possivelmente serão Utilizados

As distâncias d e dT calculadas pelo TwoWayDijkstra têm como objetivo permitir a identificação de todos os corredores que possivelmente serão utilizados(*CPU)* pelos atletas. O seguinte critério foi utilizado para definir se uma aresta fazia parte do conjunto solução ou não:

Seja e o corredor entre os locais u e v, e t o vértice de destino, e faz parte do conjunto solução se e somente se $d[t] = d[u] + dT[v]$.

O algoritmo desenvolvido para identificar tais corredores é apresentado a seguir:

Algorithm 2 Encontra Corredores que Possivelmente serão Utilizados (CPU)

```
1: function EncontraCPU(e, d, dT, nCP, c)
2:     camMIn = newEstadio(e.numLocais)        ▷ grafo composto por arestas dos caminhos mínimos
3:     for u ← 1 to e.numLocais do
4:         for all vértice v ∈ e.Adj[u] do
5:             if d[t] = d[u] + dT[v] then      ▷ adiciona a aresta nas listas de u e v
6:                 camMin.Adj[u] ∪ (u, v)
7:                 camMin.Adj[v] ∪ (v, u)
8:                 c[(u, v).cId] = 1                                  ▷ marca os CPUs
9:                 nCP = nCP + 1                                      ▷ conta o número de CPUs
    return camMin
```

O grafo *camMin* é um grafo não-orientado composto apenas pelas arestas que estão presentes nos caminhos mínimos. A não orientação é necessária para resolver o problema apresentado na subseção seguinte. Os corredores identificados pelo algoritmo são marcados no vetor $c[]$ com o número *1*.

2.4 Encontrando Corredores que Definitivamente serão Utilizados

CDUs são os corredores que sempre serão utilizados pelos atletas, ou seja, a remoção deles no grafo *camMin* fará com que chegar ao destino a partir do vértice de origem seja impossível. A impossibilidade se dá pelo fato do grafo ficar desconexo, sendo composto por duas componentes conexas. Este tipo de aresta é denominado *"ponte"* em Teoria dos Grafos. É garantido que as pontes fazem parte do conjunto solução porque *camMin* é composto apenas por arestas de caminhos mínimos, logo a desconexão do grafo impedirá que o destino seja alcançado.

O EncontraCDU(3), algoritmo implementado no trabalho, foi baseado no algoritmo de *Tarjan* apresentado em [2]. A princípio, similarmente ao algoritmo original, o EncontraCDU foi implementado de forma recursiva. No entanto, para grafos muito grandes, as inúmeras chamadas recursivas estavam estourando a pilha de execução e interrompendo a

execução do programa. Então, uma versão iterativa foi implementada no lugar. Ela faz uso de duas variáveis para simular a exploração da *Depth-First-Tree*:

- *int caminho[]:* representa a ordem em que os vértices são explorados
- *int posV:* indica a posição que o vértice explorado será inserido no vetor *caminho[]*

Algorithm 3 Encontra CDU

```
1:  function EncontraCDU(e, nCD, c)
2:      caminho[], posV ← 0
3:      lista[]
4:      for i ← 1 to e.numLocais do
5:          p[i] ← 0, visited[i] ← false
6:          lista[i] ← e.Adj[i].begin
7:      while true do
8:          if 1ª vez explorando vértice u then
9:              t = t + 1
10:             caminho[posV] ← u                          ▷ caminho de exploração
11:             posV ← posV + 1
12:             visited[u] = true
13:             d[u] = t, l[u] = t
14:         while lista[u] ≠ lista[u].end do
15:             v ← lista[u].v
16:             lista[u] ← lista[u] + 1
17:             if visited[v] = false then
18:                 p[v] ← u
19:                 break
20:             else if v ≠ p[u] then
21:                 l[u] ← min(l[u], d[v])
22:         if não há vértices a serem explorados then
23:             return
24:         else if todos os vértices adjacentes a u foram visitados then
25:             v ← u
26:             posV ← posV − 1                            ▷ menos um vértice a ser explorado
27:             u ← caminho[posV − 1]
28:             l[u] ← min(l[u], l[v])
29:             if l[v] > d[u] then
30:                 c[(u, v).cId] ← 2                      ▷ marca os CDUs
31:                 nCD ← nCD + 1                          ▷ conta os CDUs
```

3 Análise de Complexidade de Tempo

Esta seção apresenta a análise das principais funções desenvolvidas no trabalho. Ao fim, suas complexidades são consideradas em conjunto para obtenção da complexidade assintótica total do algoritmo. É importante destacar que as análises foram feitas de forma agregada com o objetivo de mostrar o limite assintótico mais firme possível. As complexidades das operações sob as estruturas de dados utilizadas foram obtidas em: http://www.cplusplus.com.

3.1 CriaGrafo

CriaGrafo é a função que faz a leitura do arquivo de entrada e cria o grafo a partir dos valores lidos. A complexidade se dá em função do número de arestas contidas no grafo. Cada aresta lida é adicionada em ambas as listas de adjacência do grafo. Considerações:

1. número de arestas = $|E|$
2. a complexidade de adicionar um elemento na lista é constante (armotizada)
3. grafo possui duas listas de adjacência ($vector < tuple < int, int, int >>$)

$$\begin{aligned} \text{Complexidade} &= 2(O(1) * |E|) \\ &= 2 * |E| \\ &= O(|E|) \end{aligned}$$

3.2 TwoWayDijkstra

TwoWayDijkstra(1) possui *loops* apenas em função do número de vértices do grafo. Algumas considerações:

1. número de vértices = $|V|$
2. dois vetores de distâncias mínimas
3. duas filas de prioridades *(priority_queue)*
4. dois *while loops* para cálculo das distâncias

Cada vetor de distância é inicializado com 0 para os vértices de origem e $infinito$ para os demais:

$$\begin{aligned} Complexidade &= 2(1 * |V|) \\ &= 2 * |V| \\ &= O(|V|) \end{aligned}$$

A fila de prioridade no pior caso leva $lg|V|$ para inserir um elemento na posição correta:

$$\begin{aligned} Complexidade &= 2(lg|V| * |V|) \\ &= 2 * |V|lg|V| \\ &= O(|V|lg|V|) \end{aligned}$$

Cada *while loop* é executado $|V|$ vezes e em cada iteração o vértice u é extraído da fila de prioridade($2lg|V|$) e sua lista de adjacência é examinada no *for loop*. Assim, o *for loop* é

executado $|E|$ vezes, fazendo o relaxamento de uma aresta por vez. Durante o processo de relaxamento, caso o $d[v]$ seja atualizado, ele é reinserido na fila de prioridade:

$$\begin{aligned} Complexidade &= 2(|E|(1 + lg|V|) + 2lg|V|) \\ &= 2(|E|lg|V| + 2lg|V| + |E|) \\ &= 2|E|lg|V| + 2|E| + 4lg|V| \\ &= O(|E|lg|V|) \end{aligned}$$

Logo, a complexidade total do algoritmo TwoWayDijkstra é:

$$\begin{aligned} &= O(|V|) + O(|V|lg|V|) + O(|E|lg|V|) \\ &= O((|V| + |E|) * lg|V|) + O(|V|) \\ &= O((|V| + |E|) * lg|V|) \end{aligned}$$

3.3 EncontraCPU

Função composta por dois *for loops* aninhados. O primeiro *loop* é executado uma vez para cada vértice, totalizando $|V|$ iterações. Em cada iteração o segundo *loop* percorre a lista de adjacência do vértice, buscando por arestas que fazem parte do caminho mínimo e as adicionando ao grafo de caminhos mínimos *camMin*. O grafo *camMin* é não-orientado, implicando na adição da aresta (u, v) nas listas de adjacência de ambos os vértices:

$$\begin{aligned} Complexidade &= 2 * |E| \\ &= O(|E|) \end{aligned}$$

3.4 EncontraCDU

O *for loop* da função faz a inicialização de três vetores para cada vértice do grafo *camMin*. Isto é feito $|V|$ vezes. Em seguida temos a parte principal da função, os dois *while loops* aninhados. O primeiro é executado até que não existam mais vértices a serem processados. O segundo faz o controle dos vértices sendo explorados, sendo executado uma vez para cada entrada na lista de adjacência. O *camMin* é um grafo não-orientado, o que implica em uma lista de adjacência de tamanho $2|E|$.

$$\begin{aligned} Complexidade &= |V| + 2|E| \\ &= O(|V| + |E|) \end{aligned}$$

3.5 GeraSaida

A saída é gerada a partir dos corredores marcados no vetor c em EncontraCPU e EncontraCDU. O vetor tem tamanho $|E|$ e é percorrido uma única vez para criação das

strings de saída.

$$\begin{aligned} Complexidade &= |E| \\ &= O(|E|) \end{aligned}$$

3.6 Análise do Programa

Cada uma das funções especificadas previamente é executada uma única vez. Logo, a complexidade total do programa é a soma das complexidades das funções:

$$\begin{aligned} Complexidade &= O(|E|) + O((|V| + |E|) * lg|V|) + O(|E|) + O(|V| + |E|) + O(|E|) \\ &= O((|V| + |E|) * lg|V|) + 4(O(|E|)) + O(|V|) \\ &= O((|V| + |E|) * lg|V|) \end{aligned}$$

Portanto, a complexidade final do programa é dada em função da complexidade do algoritmo TwoWayDijkstra.

4 Análise Experimental

A análise experimental do algoritmo foi realizada para quatro tipos de grafos. Foram criados diferentes instâncias de cada tipo alterando número de vértices e número de arestas. Os pesos das arestas também foram considerados em algumas análises a fim de saber se eles teriam influência sob o tempo de execução. O algoritmo foi executado dez vezes para cada instância e a média dos tempos de execução foi calculada. A partir dos tempos médios, foram calculadas quantas arestas o algoritmo processa por segundo para grafos com mais de *1000* vértices. Todos as execução ocorreram no mesmo ambiente: *Laptop Lenovo y510p. Processador Intel Core i7 2,4GHz. 8Gb RAM. HD 5400Rpm. S.O: Windows 10.*

4.1 Grafos Arbitrários

Estes grafos foram gerados seguindo os seguintes critérios:

- Em primeiro lugar, um caminho da origem ao destino foi criado passando por todos os vértices
- Para cada vértice, foram adicionadas arestas saindo deles para $|V|/2$ vértices aleatórios, totalizando $(|V| - 1) + |V|^2/2$ arestas.

Os resultados apresentados na Tabela 1 estão dentro do esperado. Primeiro, nota-se que o tempo de execução do algoritmo cresceu juntamente com o grafo já que mais vértices e arestas são processados pelo algoritmo. Os aumentos dos tempos médios se deram na proporção de aproximadamente *7*, *4* e *2,2* vezes entre os grafos de *1000* e *2500*, *2500* e *5000* e *5000* e *7500* vértices respectivamente. É interessante destacar que ambos os grafos com arestas de peso aleatório e máximo tiveram resultados semelhantes. Logo, os pesos não possuíram influência marcante sob os tempos para este tipo de grafo. O número de arestas processadas por segundo foi em média *423.803,67*.

Tempo Médio para Grafos Arbitrários			
		Tempo	
$\|V\|$	$\|E\|$	Distância Aleatória	Distância Máxima
10	59	0,0164s	0,0191s
100	5.099	0,0302s	0,0303s
1.000	500.999	1,1615s	1,1728s
2.500	3.127.499	7,605s	7,344s
5.000	12.504.999	29,935s	30,119s
7.500	28.132.499	64,539s	66,041s

Tabela 1: Resultados - GAD

4.2 Grafos Completos

Foram gerados quatro grafos completos: K_{10}, K_{100}, K_{1000} e K_{5000}. Grafos maiores não foram considerados por possuírem um imenso número de arestas. A Tabela 2 apresenta o resultado da execução dos algoritmos:

Tempo Médio para Grafos Completos		
$\|V\|$	$\|E\|$	Tempo
10	90	0,0191s
100	9.900	0,0444s
1.000	999.000	2,2941s
2.500	6.247.500	14,695s
5.000	24.995.000	61,0849

Tabela 2: Resultados - Grafos Completos

Para grafos completos foi considerado apenas arestas com pesos aleatórios, caso contrário as arestas entre a origem e destino sempre seriam escolhidas como o caminho mínimo. Assim como para os grafos arbitrários, conforme n cresce, o tempo de execução do algoritmo aumenta. Nota-se que os grafos completos possuem, para a mesma quantidade de vértices, aproximadamente o dobro do número de arestas dos grafos arbitrários gerados. Isto acarretou no dobro do tempo de execução para grafos com o mesmo número de vértices. Nos Grafos Completos, são processadas em média *402.049,14* arestas por segundo.

4.3 Grafos Acíclicos Dirigidos

Os *GADs* foram gerados da seguinte forma:

- Os primeiros $|V|/2$ vértices possuem uma aresta saindo para cada um dos seguintes $|V|/2$ vértices.
- A segunda metade dos vértices possuem uma aresta saindo para cada um dos seguintes vértices até o vértice final $|V|$.

  Baseados nos critérios definidos, o número de arestas de um grafo com $|V|$ vértices é igual a: $\left(\frac{|V|}{2}\right)^2 + \sum_{i=1}^{|V|/2} i$, fazendo com que um aumento de *10* vezes no número de vértices leve a um aumento de cerca de *100* vezes no o número de arestas.

  Os resultados para as instâncias geradas é apresentado na Tabela 3:

Tempo Médio para Grafos Acíclicos Dirigidos			
		Tempo	
$\|V\|$	$\|E\|$	Pesos Aleatórios	Pesos Máximos
10	35	0,0283s	0,0247s
100	3.725	0,0271s	0,0298s
1.000	374.750	0,9143s	0,9012s
2.500	2.343.125	5,683	5,941
5.000	9.373.750	23.311s	22,198s
7.500	21.091.875	50,105s	48,221s
10.000	37.497.500	90,0805s	90,5273s

Tabela 3: Resultados - GAD

Similarmente, para os grafos com poucos vértices *($|V| = 10$ e 100)*, o algoritmo executou em tempos praticamente iguais. A partir de *1000* vértices, a proporção no aumento do tempo de execução vai diminuindo conforme o grafo cresce. O tempo aumenta em aproximadamente *6* vezes entre os grafos de *1000* e *2500* vértices. Já para os grafos de *7500* e *10000* vértices, o aumento é de apenas *1,8* vezes. Isso é válido para ambos os grafos com pesos aleatórios e máximos, indicando que os pesos das arestas também não tiveram grande influência. O algoritmo processa cerca de *414.564,25* arestas por segundo para os GADs.

4.4 Grafos de Camadas

Estes grafos possuem *1* vértice em cada uma de suas camadas exteriores. O vértice de origem pertence à primeira camada e o de destino à última. O restante dos vértices estão distribuídos em n camadas interiores as quais possuem *50* vértices cada. Cada vértice de uma camada está conectado a todos os outros da camada seguinte. Logo, um Grafo em Camadas com total de $|V|$ vértices em suas camadas interiores possui:

- $|V| + 2$ vértices
- c camadas, sendo $c = \frac{|V|}{50}$
- $2 * 50 + 50^2 * c/2$ arestas

Tempo Médio para Grafos em Camadas					
				Tempo	
$\|V\|$	$\|E\|$	c	$\|V\|/c$	Pesos Aleatórios $(1 - 10^6)$	Pesos Máximos (10^6)
102	2.600	2	50	0,0368s	0,0495s
1.002	25.100	20	50	0,1164s	0,1583s
10.002	250.100	200	50	0,7634s	1,0644s
100.002	2.500.100	2.000	50	6,8186s	9,1508s
1.000.002	25.000.100	20.000	50	70,107s	92,3293s

Tabela 4: Resultados - Grafos em Camadas

Conforme exibido na Tabela 4, é possível perceber que, diferentemente do restante dos grafos, os pesos das arestas tiveram certa influência sob o tempo de execução. Os grafos que

possuem arestas de peso aleatório foram processados relativamente mais rápido do que aqueles compostos inteiramente por arestas de peso máximo. O número de arestas processadas por segundo, em média, é:

- *267.431,85* para os grafos com pesos aleatórios
- *198.007,02* para os grafos com pesos máximos

5 Conclusão

Foram abordados três subproblemas neste relatório: processamento caminhos mínimos de uma única origem, identificação das arestas pertencentes à eles e, dentre estas, aquelas que estão presentes em todos os caminhos mínimos identificados. Eles foram resolvidos utilizando como base o *Algoritmo de Dijkstra* e algoritmo para identificação de pontes proposto por *Tarjan*. Os algoritmos desenvolvidos foram analisados quanto as suas complexidades assintóticas, constatando-se que a complexidade final do programa é a mesma do *Algoritmo de Dijkstra*. Por fim, análises experimentais foram conduzidas executando os algoritmos para diferentes tipos de grafos, sendo calculado o tempo médio de execução para cada entrada. Além disso, também foi mostrado a quantidade média de arestas que o algoritmo processa para cada um dos quatro tipos de entrada.

Referências

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*. 2009.

[2] Halim, S., and Halim, F. *Competitive Programming 3: The New Lower Bound of Programming Contests: Handbook for ACM ICPC and IOI Contestants*. 2013.