Trabalho Prático
Projeto e Análise de Algoritmos - Grafos

Pablo Goulart Silva

[1]DCC – Universidade Federal de Minas Gerais

pgoulart@dcc.ufmg.br

Introdução

Esse relatório apresenta os métodos utilizados para solucionar o problema de posicionamento de entidades em arestas que pertençam aos caminhos mínimos de um grafo $G(V, E)$. G é um grafo orientado e ponderado positivamente.

A modelagem em grafo se aplica ao contexto de posicionamento de fotógrafos nos corredores de um estádio para fazer a cobertura dos imagens dos jogadores no trajeto entre o desembarque até o vestiário. O objetivo é recomendar quantas e quais são as arestas atravessadas pelos jogadores - em caso de múltiplos caminhos - e as arestas que definitivamente serão atravessadas - caso onde os caminhos mínimos compartilham arestas.

Definição

O problema pode ser dividido em quatro etapas:

- Determinar o tamanho do caminho mínimo
- Determinar todas as arestas que pertençam a todos os caminhos mínimos possíveis
- Determinar se existem arestas que sejam compartilhadas por todos os caminhos mínimos
- Determinar quais arestas são comuns aos caminhos mínimos

Determinação da distância mínima com método de *Dijkstra*

O primeiro item pode ser solucionado utilizando-se o algoritmo de *Dijkstra*. O algoritmo de *Dijkstra* determina o caminho mínimo d entre a origem v_s e todos os vértices $v \in V \backslash v_s$ do grafo. O método retorna as arestas de um caminho mínimo entre a origem e o destino e as distâncias dos caminhos mínimos intermediários. Esse método pode ser aplicado em grafos ponderados de peso positivo.

A complexidade do método de *Dijkstra* é dependente das estruturas de dados utilizadas. A lista de prioridades foi implementada utilizando o tipo `set` do *C++*. `set` é implementado como *Red Black Trees*, e permite armazenar elementos não repetidos que são ordenados na estrutura. Essa estrutura de dados não permite a alteração de um valor armazenado.

Para implementar o método *DecreaseKey* é necessário que o elemento seja removido e novamente inserido. A complexidade dos métodos de inserção, busca e remoção é logarítmica [STL].

O algoritmo de *Dijkstra* chama *Insert* e *ExtractMin* uma vez para cada vértice. Cada operação é logarítmica, com complexidade agregada de $O(2V logV)$. *DecreaseKey* é chamado uma vez para cada aresta. Essa operação executa as operações *Remove* e *Insert*, portanto sua complexidade é $O(2ElogV)$. O custo agregado do algoritmo é $O(2(V + E)logV)$ = $O((V + E)logV)$.

Determinação das arestas dos caminhos mínimos

Para determinar todas as arestas v pertencentes a todos os caminhos mínimos - chamamos S o conjunto de arestas dos caminhos mínimos - é necessário definir uma função que mapeie o conjunto de arestas do grafo V em um conjunto S de arestas dos caminhos mínimos de G. As arestas dos caminhos mínimos, de comprimento w_{ij} onde i e j representam o índice dos vértices ligados pela aresta, podem ser identificadas através da equação 1, onde t é o vértice de destino e w_{ij} é o peso da aresta que liga o vértice i ao vértice j.

$$Dijkstra(s, j) + w_{ij} + Dijkstra(j, d) = t \tag{1}$$

O primeiro termo é conhecido do cálculo de *Dijkstra* para determinar a distância mínima. O segundo termo - peso das arestas do grafo - é parâmetro do problema. A distância mínima t também é resultado da aplicação do algoritmo de *Dijkstra*. O terceiro termo, $Dijkstra(j, d)$ pode ser obtido executando-se o método de *Dijkstra* sobre o grafo transposto. Cada aresta (u, v) do grafo original é mapeada em uma aresta (v, u) do grafo transposto. A complexidade dessa operação é $O(|V| + |E|)$.

Justificativa

Em um grafo direcionado G, o algoritmo de *Dijkstra* calcula a menor distância partindo da origem para todos os vértices do grafo. A execução do algoritmo de *Dijkstra* no grafo transposto G^T, calcula a distância do destino para todos os outros vértices do grafo transposto. A distância entre a origem e o destino no grafo original é a mesma distância entre o destino e a origem no grafo transposto. Pela propriedade do limite superior [Cormen 2009], quando as arestas $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$ são relaxadas, garante-se que $d[v_k] = \delta(v_k)$. Logo, pela mesma propriedade, quando $(v_k, v_{k-1}), (v_{k-1}, v_{k-2}), ..., (v_1, v_0)$ são relaxadas no grafo transposto, garante-se o menor caminho igual a $\delta[v_k]$.

Para determinar quais arestas pertencem ao caminho mínimo, iteramos sobre todas as arestas e verificamos se a distância mínima contendo o peso da aresta respeita a equação 1. Se, no grafo original, $d[v_u]$ é a distância mínima entre a origem e v_u, e no grafo transposto, $d[v_u]$ é a distância mínima entre o destino e o vértice v_u, essa distância deve fazer parte da distância mínima entre a origem e destino no grafo original pela propriedade de subestrutura ótima [Cormen 2009], pois $\delta(v_s, v_u) + \delta(d, v_u) = \delta(v_s, d)$. Estendendo essa propriedade, $t = \delta(v_s, v_u) + w(v_u, v_v) + \delta(d, v_v)$ é válido se a aresta (u, v) pertence à algum caminho mínimo entre v_s e d, comprovando a equação 1.

Determinação de Pontes do Grafo

Para se determinar as arestas que são comuns aos caminhos mínimos foi necessário encontrar as arestas que dominam o vértice destino v_t. Segundo [Italiano et al. 2012], uma aresta (u, v) é dominante do vértice w se todos os caminhos partindo de v_s até v_t contém

(u, v), dado um grafo $G(V, E)$ fortemente conectado - isto é, em que qualquer vértice $v \in V$ seja alcançável por v_s.

Para determinar as pontes do grafo, um grafo auxiliar $G'(V, S)$ foi criado. O conjunto de vértices do grafo G' é o conjunto de vértices V do grafo G. O conjunto de arestas é formado pelas arestas pertencentes ao conjunto de arestas de caminho mínimo S do grafo original G.

O algoritmo determina as pontes do grafo a partir de uma modificação do método *DFS*.

Algorithm 1 Determinação de pontes em grafo não-orientado

```
 1: procedure STRONGBRIDGES
 2:     Let dfs_low[1...n1], dfs_num[1...n1] and dfs_parent[1...n1]
 3:     Let visited[1...n1], enqueued[1...n1]
 4:     counter ← 0
 5:     Let Q be a new stack
 6:     Q.Push(root_vertex)
 7:     enqueued[root_vertex] = true
 8:     while Q is not empty do
 9:         vertex = Q.Pop
10:         dfs_low[vertex] = dfs_num[vertex] = counter
11:         visited[vertex] = true
12:         enqueued[vertex] = false
13:         for it in adjacency_list[vertex] do
14:             if !visited[it.vertex] and !enqueued[it.vertex] then
15:                 Q.Push(it.vertex)
16:                 dfs_parent[it.vertex] = vertex
17:                 enqueued[it.vertex] = true
18:             else if enqueued[it.vertex] then
19:                 dfs_parent[it.vertex] = vertex
20:             else if dfs_parent[vertex]! = it.vertex then
21:                 if dfs_low[it.vertex] <dfs_low[vertex] then
22:                     dfs_low[vertex] = dfs_low[it.vertex]
23:                     v_aux = dfs_parent[vertex]
24:                     while dfs_low[v_aux] != dfs_low[it.vertex] do
25:                         if dfs_low[it.vertex] <dfs_low[v_aux] then
26:                             dfs_low[v_aux] = dfs_low[it.vertex]
27:                         v_aux = dfs_parent[v_aux]
28:         counter ++
```

O algoritmo mantém três valores por vértice, $dfs\_low(u)$, $dfs\_num(u)$ e $dfs\_parent(u)$. $dfs\_num(u)$ armazena a iteração que o vértice u foi visitado pela primeira vez e $dfs\_low(u)$ armazena o menor dfs_num alcançável a partir do vértice u. $dfs\_parent(u)$ armazena o vértice pai do vértice u (linha 2). Inicialmente, $dfs\_low(u) = dfs\_num(u)$ quando o vértice é visitado pela primeira vez (linha 10). $dfs\_low(u)$ é re-atribuído quando há ciclos - aresta de retorno - e ele encontra um vértice com dfs_num

menor que seu valor atual. Se não houver nenhuma aresta de retorno, $dfs\_low(u)$ permanecerá igual a dfs_num. No laço `for` da linha 13, o algoritmo visita os vértices adjacentes ao vértice atual. Quando $dfs\_low(u)$ >$dfs\_low(v)$ então a aresta (u, v) é uma ponte (linha 21). Os vértices antecessores são atualizados utilizando dfs_parent os vértices (linhas $24 - 27$).

O algoritmo itera sobre todos os vértices do grafo (linha 8) em $O(V)$. O laço `for` da linha 13 visita todas as arestas do conjunto S, $O(V + E)$. Se houver arestas de retorno que satisfaçam a condição da linha 20, os antecessores do vértice são atualizados com o *dfs_low(v)* cuja complexidade é $O(cV)\forall c \in (0, 1)$, portanto $O(V+E+cV) = O(V+E)$.

Complexidade

A complexidade agregada do trabalho é dominada pelo algoritmo de *Dijkstra*, $O((V + E)logV)$. Os métodos de definição de arestas de todos os caminhos mínimos e o método de busca de pontes são lineares, com complexidade $O(V + E)$. O método de busca de pontes utiliza operações *Insert* em estruturas `set`, de complexidade $O(logV)$. Esse custo é desprezível em relação ao custo linear do algoritmo. Somados, os métodos apresentados nesse trabalho possuem complexidade $O(2(V+E)logV+(V+E+logV)+(V+E)) = O((V + E)logV)$.

Implementação

O trabalho foi implementado utilizando C++. A linguagem *C++* possui a sintaxe mais declarativa que *C*, *Java* e *Python*. Alguns recursos da linguagem utilizados nesse trabalho tornaram o código menos legível, como exemplo a classe shared_ptr , utilizada para evitar problemas de *memory leak* - isto é, variáveis alocadas durante a execução não desalocadas no fim do programa. Essa classe é um *proxy* de acesso, que gerencia o tempo de vida do objeto, chamando o destrutor da classe - que desaloca a variável alocada - quando não houver nenhuma referência para a variável. Devido à esse recurso, criou-se typedefs para simplificar a leitura.

O grafo G foi armazenado como uma lista de adjacências. Seu tipo é representado pelo código 1.

```cpp
typedef std::vector<std::vector<std::shared_ptr<Corredor>>> Grafo_t;
```

Código fonte 1: Tipo Grafo

```cpp
typedef std::vector<Tipos::peso_t> ListaPesos_t;

typedef std::vector<Tipos::vertice_t> ListaVertices_t;
```

Código fonte 2: Tipo lista de pesos e vértices de retorno do método de *Dijkstra*

A complexidade de espaço do grafo implementado como lista de adjacências é $O(|V| + |E|)$.

Cada elemento da lista de adjacências é armazenada no tipo Corredor. Esse tipo armazena o identificador da aresta (corredor), o identificador do vértice oposto da aresta o seu comprimento (código fonte 3).

```
struct Corredor {
    uint64_t num_corredor;
    uint64_t num_vertice;
    double distancia;

    Corredor(uint64_t, uint64_t, double);
};
```

Código fonte 3: Tipo Corredor

O método de *Dijkstra* recebe o tipo *Grafo_t* (trecho 1), o identificador do vértice de origem, e retorna uma tupla que contém os comprimentos do caminho mínimo entre cada vértice e a origem, e as arestas encontradas do caminho mínimo entre a origem e o destino (trecho 4).

```
std::tuple<ListaPesos_t, ListaVertices_t>  Dijkstra (
            Vertice_t,
            Grafo_t &
        );
```

Código fonte 4: Assinatura do método de *Dijkstra*

```
static void identifica_pontes_iterativo (
    Grafo_t grafo_original,
    Grafo_t grafo_arestas_caminho_min_nao_direcionado
);
```

Código fonte 5: Assinatura do método de identificação de pontes iterativo

O método de identificação de pontes (trecho 5) recebe o grafo original, o grafo não direcionado que contém as arestas pertencentes à todos os caminhos mínimos e o identificador do vértice inicial.

Resultados

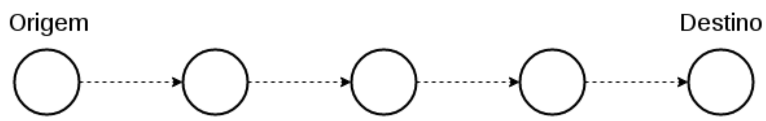

Figura 1. Grafo de testes com caminho mínimo de $V-1$ arestas. Todas as arestas são pontes.

Foram executadas variações de grafos com os limites de números de vértices, arestas e pesos segundo a especificação do trabalho. Os tempos de execução do algoritmo e as parametrizações do grafo estão armazenadas na tabela 1. O gráfico 8 exibe o tempo de execução do algoritmo para a soma dos parâmetros V e E. O tempo de execução corresponde a complexidade do algoritmo dominado pelo método de *Dijkstra*, executado duas vezes durante a execução e com complexidade $O((V+E)logV)$.

Conclusão

Grafos são abstrações utilizadas em diversas aplicações. O trabalho prático consistiu na utilização da abstração de grafos para resolver o problema de alocação de fotógrafos em

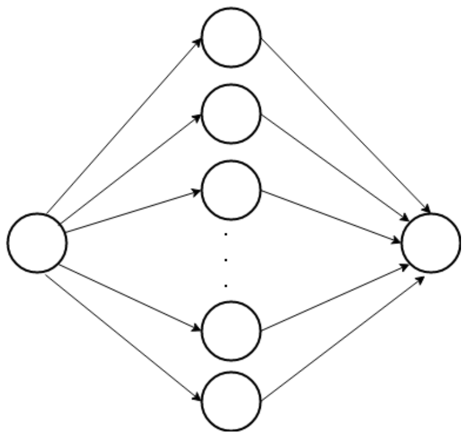

Figura 2. Grafo de testes $V-2$ caminhos mínimos e nenhuma ponte.

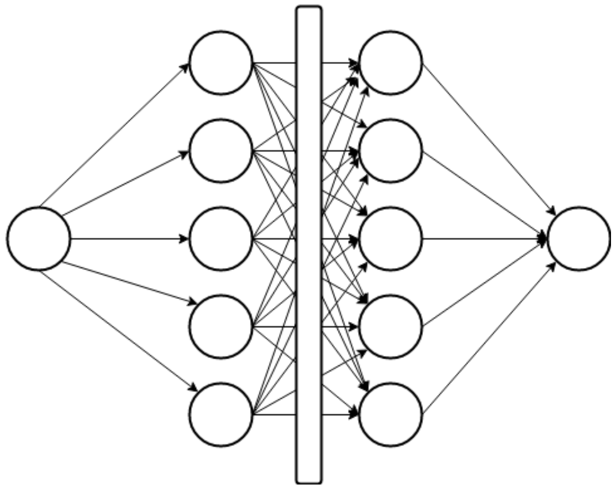

Figura 3. Grafo de testes totalmente conectado de N camadas com $(C)^N$ caminhos mínimos e nenhuma ponte, para C igual o número de vértices por camada e N.

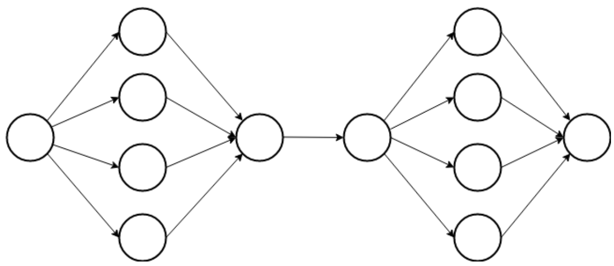

Figura 4. Grafo de testes com 2 camadas interligadas por uma ponte e C^2 caminhos mínimos, para C igual ao número de vértices de uma camada.

corredores de um estádio. Os corredores foram escolhidos para obedecer o critério de caminho mínimo, solucionado com o algoritomo de *Dijkstra*, e a seleção de corredores compartilhados foi solucionado pelo algoritmo de *Tarjan*. Determinamos os corredores através da utilização de propriedades dos grafos, pelo qual foi possível conhecer todos os caminhos mínimos do trajeto.

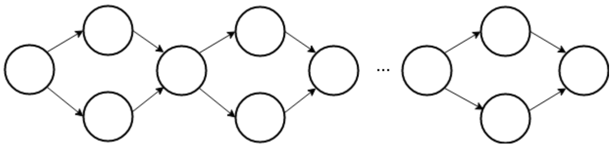

Figura 5. Grafo de teste redundante com $2A$ caminhos mínimos, para A igual ao número de anéis.

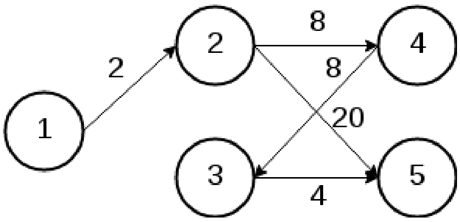

Figura 6. Grafo simples de teste com uma ponte.

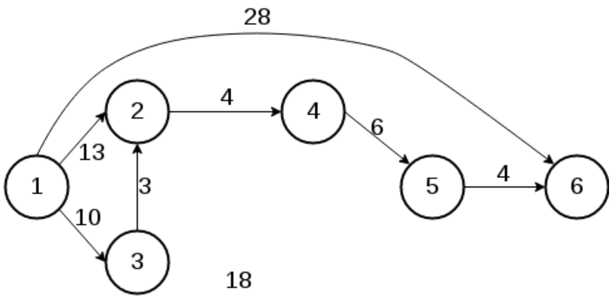

Figura 7. Grafo simples de teste com com um caminho mínimo e três pontes.

Tabela 1. Parametrização dos algoritmos por tempo de execução.

Figura	Vértices	Arestas	Pontes	Tempo
3	10004	980401	0	5.020s
2	100000	133332	0	0.877s
1	100000	99999	99999	0.774s
4	10000	19993	1	0.126s
7	6	8	3	<0.01s
6	5	5	1	<0.01s

Esse trabalho explorou diversas ferramentas para sua solução enquanto serviu como laboratório para modelagem de problemas em grafos. Algumas dificuldades encontradas foram na definição do método de busca de pontes. A literatura [Halim and Halim 2013] apresenta o problema em grafos não direcionados, enquanto que a abordagem utilizando grafos direcionados não especifica uma propriedade que permita a transformação do problema para grafos não direcionados.

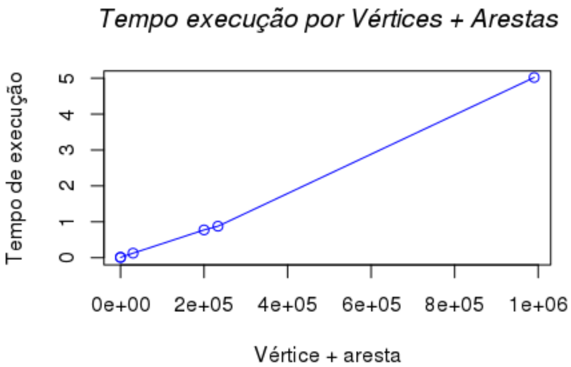

Figura 8. Tempo de execução por parametrização de acordo com tabela 1.

Referências

Stlset. `https://gcc.gnu.org/onlinedocs/gcc-4.6.3/libstdc++/api/a00284.html`. Accessed: 2018-05-03.

Cormen, T. H. (2009). *Introduction to algorithms*. MIT press.

Halim, S. and Halim, F. (2013). *Competitive Programming 3*. Lulu Independent Publish.

Italiano, G. F., Laura, L., and Santaroni, F. (2012). Finding strong bridges and strong articulation points in linear time. *Theoretical Computer Science*, 447:74–84.