Trabalho Prático - Grafos

Projeto e Análise de Algoritmos
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
1º Semestre de 2018

Maria Luísa Costa Pinto
dcc.ufmg.br/~maria.luisa

Resumo: Esse projeto tem por objetivo principal colocar em prática a resolução de problemas utilizando modelagem em grafos, otimizando a complexidade a fim de chegar a solução ótima. O problema a seguir é um aplicação fictícia da distribuição de repórteres em locais e corredores distintos. A implementação foi feita em linguagem Java.

1. INTRODUÇÃO

GRAFOS

Grafo é um campo de estudo da matemática que estuda as relações entre os objetos de um determinado conjunto. Para isso, são empregadas estas estruturas chamadas de grafos, que são compostas por dois conjuntos G(V,E), onde V é um conjunto não vazio de objetos denominados vértices (ou nós) e E (do inglês Edges - arestas) é um subconjunto de pares não ordenados de V. Um assim, constitui um conjunto de vértices e arestas.

Os grafos podem representar diversas estruturas que estão em toda parte e muitos problemas de interesse prático podem ser formulados como questões sobre certos grafos. Exemplo relacionamentos entre pessoas podem ser modelados com grafos, redes sociais, relações biológicas, a internet em si pode ser modelada em grafos, entre outros. O desenvolvimento de algoritmos para manipular grafos é um tema importante da ciência da computação.

DESCRIÇÃO DO PROBLEMA

O problema a ser resolvido consiste na resolução do seguinte problema:

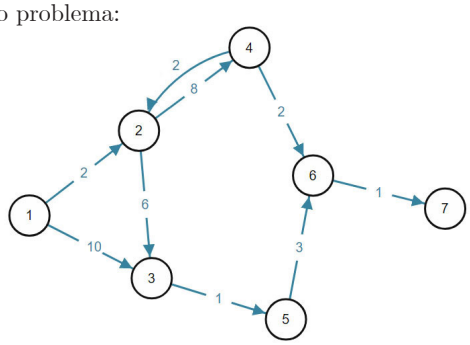
Na Copa do Mundo 2018, a FIFA decidiu que a imprensa só poderá ter acesso aos corredores existentes nos estádios, não sendo permitida a permanência de fotógrafos e jornalistas nos demais locais. Além disso, os corredores só poderão ser utilizados pelos atletas em uma única direção, ou seja, corredores que possuem o sentido da sala A para sala B, não poderão ser trafegados no sentido contrário.

A empresa que lhe contratou possui o mapa dos corredores onde é permitido o acesso e permanência da imprensa. Além disso, sabem que ao chegar no estádio os jogadores e comissão técnica sempre optam por percorrerem a menor distância entre o ponto de chegada no estádio e o vestiário. O desafio está no fato de poderem existir mais de um caminho cuja distância percorrida entre o ponto de chegada no estádio e o vestiário seja mínima, pois foi justamente este o fato que fez com que a companhia perdesse diversas oportunidades durante a última edição do evento. Diante deste problema, a grande rede de comunicação espera que você seja capaz de identificar todos os possíveis corredores que poderiam ser percorridos pelas seleções durante o trajeto entre o ponto de chegada e os vestiários, respeitando a regra de menor distância. Além disso, podem existir corredores que independentemente do trajeto escolhido pelos

atletas e comissão técnica, serão definitivamente utilizados. Tais corredores são com certeza os melhores pontos para alocação dos fotógrafos e jornalistas, portanto, é sua tarefa identificá-los também.

EXEMPLO

Abaixo é apresentado um exemplo do problema:



Neste exemplo os corredores são:

- Corredor #1 pode ser percorrido do local 1 para o local 2 com distância 2
- Corredor #2 pode ser percorrido do local 1 para o local 3 com distância 10
- Corredor #3 pode ser percorrido do local 2 para o local 3 com distância 7
- Corredor #4 pode ser percorrido do local 2 para o local 4 com distância 8
- Corredor #5 pode ser percorrido do local 3 para o local 5 com distância 3
- Corredor #6 pode ser percorrido do local 4 para o local 2 com distância 2
- Corredor #7 pode ser percorrido do local 4 para o local 6 com distância 2
- Corredor #8 pode ser percorrido do local 5 para o local 6 com distância 3
- Corredor #9 pode ser percorrido do local 6 para o local 7 com distância 1

2. MODELAGEM E SOLUÇÃO PROPOSTA

MODELAGEM EM GRAFOS

A própria descrição do problema já sugere uma modelagem em grafos, onde o conjunto denominado N, os pontos onde os repórteres não podem ficar são os vértices do grafo e o conjunto M, os corredores onde os repórteres podem ficar, são as arestas do grafo. O grafo é direcionado e o objetivo é encontrar caminhos mínimos nos grafos para saber em quais corredores a imprensa pode passar, e em quais ela certamente irá passar.

ALGORITMO DE DIJKSTRA

O algoritmo de Dijkstra, criado pelo cientista da computação holandês Edsger Dijkstra em 1956 e publicado em 1959, apresenta a solução do problema do caminho mais curto em um grafo dirigido ou não dirigido com arestas de peso não negativo, em tempo computacional O([m+n]log n) no melhor caso, onde m é o número de arestas e n é o número de vértices.

O algoritmo considera um conjunto S de menores caminhos, iniciado com um vértice inicial I. A cada passo do algoritmo busca-se nas adjacências dos vértices pertencentes a S aquele vértice com menor distância relativa a I e adiciona-o a S e, então, repetindo os passos até que todos os vértices alcançáveis por I estejam em S. Arestas que ligam vértices

já pertencentes a S são desconsideradas.

O algoritmo de Dijkstra é útil para encontrar menores caminhos em um grafo dirigido. Ele pode ser útil na resolução da primeira parte do problema proposto: encontrar todas as arestas que estão em algum caminho mínimo. Neste projeto o Algoritmo de Dijkstra foi o primeiro passo para a resolução. A seguir, nos proximos passos, será descrito como os resultados do Algoritmo de Dijkstra foi utilizado.

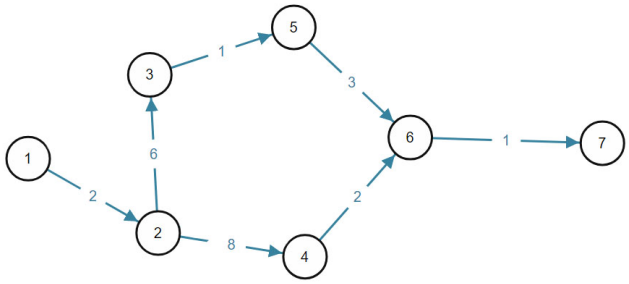
FUNCIONAMENTO DO ALGORITMO

O algoritmo de Dijkstra atribuirá alguns valores iniciais de distância e tentará melhorá-los passo a passo.

1. Marcar todos os nós não visitados. Crie um conjunto de todos os nós não visitados chamado Conjunto não visitado.
2. Atribua a cada nó um valor de distância provisório: configure-o para zero para o nosso nó inicial e para o infinito para todos os outros nós. Defina o nó inicial como atual.
3. Para o nó atual, considere todos os seus vizinhos não visitados e calcule suas tentativas de distâncias através do nó atual. Compare a tentativa de distância recém-calculada com o valor atual atribuído e atribua a menor. Por exemplo, se o nó atual A estiver marcado com uma distância de 6, e a borda que o conecta com um vizinho B tiver comprimento 2, a distância entre B e A será 6 + 2 = 8. Se B tiver sido marcado anteriormente com uma distância maior que 8, então mude para 8. Caso contrário, mantenha o valor atual.
4. Quando terminarmos de considerar todos os vizinhos do nó atual, marque o nó atual como visitado e remova-o do conjunto não visitado. Um nó visitado nunca será verificado novamente.
5. Vá para o próximo nó não visitado com as menores distâncias provisórias e repita as etapas acima, que verificam os vizinhos e marcam as visitas.
6. Se o nó de destino tiver sido marcado como visitado (ao planejar uma rota entre dois nós específicos) ou se a menor distância entre os nós no conjunto não visitado for infinito (ao planejar uma travessia completa; ocorre quando não há conexão entre o nó inicial e os nós restantes não visitados), pare. O algoritmo terminou.
7. Caso contrário, selecione o nó não visitado marcado com a menor distância tentativa, configure-o como o novo "nó atual" e volte para a etapa 3.

Ao planejar uma rota, na verdade não é necessário esperar até que o nó de destino seja "visitado" como acima: o algoritmo pode parar quando o nó de destino tiver a menor distância entre todos os nós "não visitados" (podendo ser selecionado como o próximo "atual").

O resultado do Algoritmo Dijkstra com o exemplo apresentado neste relatório é apresentado abaixo. As arestas são as que estão em caminhos mínimos do grafo original. O subgrafo G_1 é dado pelo conjuntos destas arestas.



DESIGUALDADE TRIANGULAR

O segundo passo para resolução do problema utiliza a propriedade da desigualdade triangular.

Desigualdade triangular: Seja s um vértice de um grafo com custos positivos nos arcos. Para cada vértice t ao alcance de s, seja d(t) a distância de s a t. Para qualquer arco v→w de custo c_{vw} tem-se $d(v) + c_{vw} \geq d(w)$ (mesmo que v ou w não estejam ao alcance de s).

Esta propriedade foi fundamental para saber se uma determinada aresta se encontra em algum dos caminhos mínimos. Primeiramente, vamos assumir que o Algoritmo de Dijkstra já foi executado e já sabemos qual a menor distância de cada vértice do vértice fonte s. A partir daí, para cada aresta v→w foi testado se a menor distância do vértice w ao vértice fonte s a menor distância do vértice origem da aresta v ao vértice fonte s mais o peso da aresta.

Seja p o peso da aresta v→w. E seja d(n) o resultado do Algoritmo de Dijkstra para um vértice n, ou seja, d(n) retorna a menor distância do vértice fonte s a n.

Logo o que foi feito foi:

- Comparar (d(v) + p) e d(w)
- Se os dois forem iguais então a aresta v→w está em algum caminho mínimo.

Este procedimento foi feito para todas as arestas do grafo, de forma que no final conseguimos obter todas as arestas que estão em algum caminho mínimo.

ARESTAS NECESSÁRIAS

O desafio nesta etapa consiste em encontrar todas as arestas que serão obrigatoriamente percorridas nos caminhos mínimos. Para resolver este problema, primeiramente enxergamos o conjunto de as arestas do passo anterior como um subgrafo denominado G_1. Precisamos saber quais arestas serão obrigatoriamente percorridas em G_1 partindo de um vértice s para um vértice t. Para isso usamos o conceito de Arestas Pontes em grafos, que são arestas que uma vez retiradas, desconectam o grafo.

Uma aresta de um grafo não-dirigido é uma ponte (***bridge***) se não pertence a um circuito. (Uma ponte, como qualquer outra aresta, consiste em dois arcos antiparalelos. Esses dois arcos formam um ciclo trivial, que não corresponde a um circuito.) Pode-se dizer que pontes e circuitos são conceitos complementares.

Pontes vs. conexidade: Uma aresta de um grafo não-dirigido é uma ponte se e somente se sua remoção aumenta o número de componentes conexas do grafo.

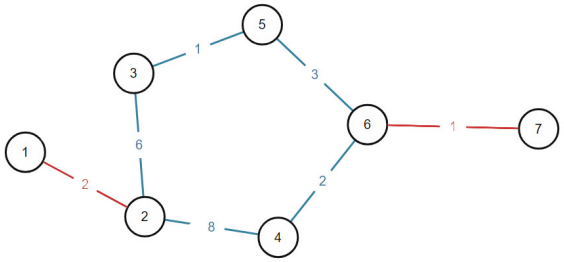
Para decidir se uma dada aresta u→v é uma ponte ou não, podemos remover u→v do grafo e verificar se existe um caminho de v a u no grafo restante, ou seja, se u e v pertencem à mesma componente conexa do grafo restante.

As pontes serão arestas que obrigatoriamente serão percorridas, pois elas são necessárias para que o grafo se mantenha conectado. Ou seja, se u→v é uma ponte, removendo u→v, o vértice u não conseguirá alcançar o vértice v e nem os vértices alcançáveis a partir de v, não há outra aresta que conecte o subgrafo formado pelo alcançáveis de u ao subgrafo formado pelos alcançáveis de v. Observe que para este fim de encontrar as arestas pontes, não é necessário que o grafo seja direcionado.

Desta forma, encontrando as arestas pontes sabemos quais arestas serãoa obrigatoriamente percorridas em G_1. A fim

de encontrar as arestas ponte foi implementado o algoritmo DFS - Depth-First Search (Busca em profundidade) verificando se as arestas eram ponte ou não. A descrição e explicação do Algoritmo está descrita no livro Competitive Programming, na página 132.

As arestas pontes identificadas pelo exemplo apresentado neste relatório estão marcadas em vermelho na imagem abaixo:



3. ANÁLISE DE COMPLEXIDADE

Abaixo serão apresentadas as complexidades de cada uma das etapas da modelagem com base na implementação desenvolvida:

ALGORITMO DE DIJKSTRA

A complexidade do Algoritmo de Dijkstra depende muito da implementação. A implementação desenvolvida neste trabalho armazena o conjunto de vértices Q em uma fila de prioridades, que foi implementada como um heap binário.

Assim a operação de remoção do menor item tem complexidade de tempo está em O(lg n) no pior caso. Desta forma o Dijkstra consegue ter uma complexidade de O((E + V) log V).

DESIGUALDADE TRIANGULAR

A complexidade do teste das arestas por meio da desigualdade triangular foi feita em tempo de execução O(V) pois com o resultado do Algoritmo de Dijkstra calculado é necessário apenas percorrer os vértices e fazer o teste.

ENCONTRAR AS PONTES

Este procedimento foi executado a partir do subgrafo $G_1 = (V_1, E_1)$, formado pelas arestas que compõe os caminhos mínimos. O algoritmo desenvolvido para encontrar as pontes do subgrafo G_1 possui complexidade $O(V_1+E_1)$. A implementação foi feita a partir da explicação e definições contidas no livro Competitive Programming e faz chamadas recursivas E vezes em um loop iterativo de V_1 chamadas.

LEITURA E ESCRITA NOS ARQUIVOS DE ENTRADA E SAÍDA

A leitura dos arquivos de entrada tem complexidade O(E), onde E é o conjunto de arestas do grafo passado no arquivo de entrada. A escrita no arquivo de saída tem complexidade $O(E_1)$ + O(R) onde E_1 são as arestas que estão em algum caminho mínimo e R são as arestas pontes encontradas.

COMPLEXIDADE TOTAL

Seja G = (V, E) um grafo direcionado onde E é o conjunto de arestas e V o conjunto de vértices, e $G_1 = (V_1, E_1)$ é o subgrafo não direcionado formado pelas arestas e vértices que compõe todos os caminhos mínimos de G. A complexidade total dada por T(G) é dada por:

$$T(G) = O((V+E) \log V) + O(V) + O(V_1+E_1) = O((V+E) \log V)$$

Considerando-se a leitura e escrita nos arquivos de entrada e saída, a complexidade é dada por:

$$T(G) = O((V+E) \log V) + O(V) + O(V_1+E_1) + O(E_1) + O(R) = O((V+E) \log V)$$

onde R são as arestas pontes encontradas.

4. O CÓDIGO

A documentação técnica de todas as classes, funções, propriedades e métodos desenvolvidos neste projeto se encontra no link: `http://homepages.dcc.ufmg.br/~maria.luisa/files/paa/grafos/documentacao_tp/`. Esta documentação foi feita seguindo o Javadoc, padrão de comentário de código em Java.

5. EXPERIMENTO E TESTES

Os códigos necessários para compilar o projeto são:

```
DijkstraAlgorithm.java
Edge.java
Graph.java
Heap.java
IOFiles.java
MainCode.java
NecessaryEdges.java
Node.java
TriangleInequality.java
compilar.sh
```

Além destes arquivos também é necessário o arquivo de entrada dos dados, inicialmente input.txt. O arquivo compilar.sh possui as instruções necessárias para que o código possa ser executado, inclusive o parâmetro com o nome dos arquivos de entrada e saida. Assim basta executar o compilar.sh para que o código possa começar sua execução. Ao final será gerado um arquivo de saída, inicialmente definido como output.txt com o retorno dos algoritmos implementados no projeto.

Foi criado também um conjunto de testes com arquivos de entrada bastante diversos para que o algoritmo pudesse ser testado de forma eficaz. O conjunto de teste pode ser encontrado no link: `http://homepages.dcc.ufmg.br/~maria.luisa/files/paa/grafos/testes_tp/`.

As máquinas usadas para testar foram máquinas do Departamento de Ciência da Computação, tanto as máquinas

da graduação, como a `tiete.grad` como uma máquina do laboratório de pesquisa `speed`. Testes pequenos foram feitos em um computador de uso pessoal, os demais testes apenas em máquinas do DCC. Os resultados foram dentro do esperado, não ultrapassando o valor de 5 segundos, nem mesmo para arquivos de entrada de grafos com 105 vértices e/ou arestas.

Inicialmente foi desenvolvida uma implementação do algoritmo Dijkstra com complexidade $O(v^2)$, porém o mesmo foi ineficiente, o que ocasionou uma segunda e atual implementação do algoritmo utilizando Heap binário, o que tornou os resultados satisfatórios.

Alguns problemas de performance foram notados ao longo do desenvolvimento, porém o problema não era devido a ineficiência da modelagem, mas sim dos recursos da linguagem que estavam sendo usados. Por exemplo, a escrita no arquivo de saída possui tempo de execução $O(E_1) + O(R)$ onde E_1 são as arestas que estão em algum caminho mínimo e R são as arestas pontes encontradas. A fim de reduzir esta complexidade eu tentei salvar os resultados (em ambos os casos, das arestas necessárias e das pontes) em uma string onde era concatenado os resultados a medida que eram descobertos, ao final, apenas era impressa a string no arquivo de saída, reduzindo estes loops da complexidade. Porém esta tentativa acabou sendo extremamente ineficiente pois a linguagem não possui bom desempenho em operações em larga escala com string. Esta alternativa acabou sendo inviável, embora reduzisse a complexidade, devido aos recursos da linguagem.

6. CONCLUSÃO

Embora possuam um caráter desafiador os trabalhos práticos são elementos estimuladores do conhecimento e aprendizagem. Neste projeto foi implementado e testado um algoritmo de resolução de problemas em grafos, encontrar caminhos mínimos, encontrar pontes, testar a desigualdade triangular, todos estes algoritmos que foram vistos em sala de aula. Assim, acredito que primeiramente o projeto cumpriu seu papel de colocar em prática os conhecimentos adquiridos em sala. Algo notório também foi como a escolha da linguagem afetou os resultados, o que permitiu ver e aprender sobre o comportamento dos diversos recursos em Java.

Foram implementadas diversas soluções até se chegar ao resultado apresentado neste relatório. A maior dificuldade era saber qual algoritmo resolveria cada subproblema da melhor forma possível. As referências ajudaram bastante a descobrir isso, assim como estudo e pesquisas. Acredito que este seja um dos maiores resultados em relação a aprendizado e desenvolvimento acadêmico por meio deste trabalho, aprender que a modelagem teórica deve ser feita e validada (baseada na literatura) antes de se começar a implementação.

7. REFERÊNCIAS

Introduction to Algorithms, 2nd Edition
Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein MIT Press, Hardcover, Published July 2001, ISBN 0070131511

Competitive Programming 3: The New Lower Bound of Programming Contests.
Steven Halim, Felix Halim. Handbook for ACM ICPC and IOI contestants (2013)

R. Sedgewick and K. Wayne, Algorithms, 4th Edition, Addison-Wesley, 2011.

Projeto de Algoritmos com Implementações em Pascal e C, 2a Edição
Nívio Ziviani, Editora Thomson, 2004