Solução para o problema ”Halting Wolf” da XXV Maratona de Programação da SBC

Daniel Vitor Pereira Rodrigues[1], **Wladimir Araújo Tavares**[1]

[1]Universidade Federal do Ceará - Campus Quixadá
Av. José de Freitas Queiroz, 5003 - Cedro, Quixadá - CE, 63902-580

danielvitor@alu.ufc.br, wladimirufc@gmail.com

Abstract. *This article describes the solution to the ”Halting Wolf”problem of the national phase of the XXV SBC 2020 Programming Marathon. The solution must be efficient in terms of time and memory used, so the choice of algorithm must be careful. For this, the problem was modeled as a weighted graph and the Edmonds-Karp algorithm was used, which is an implementation of the Ford-Fulkerson method to calculate the maximum flow in a flow network.*

Resumo. *Este artigo descreve a solução do problema ”Halting Wolf”da fase nacional da XXV Maratona de Programação da SBC 2020. A solução deve ser eficiente em termos de tempo e memória utilizados, por isso a escolha do algoritmo deve ser cuidadosa. Para isso, o problema foi modelado como um grafo com pesos e foi usado o algoritmo de Edmonds-Karp, que é uma implementação do método de Ford-Fulkerson para calcular o maior fluxo em uma rede de fluxos.*

1. Introdução

A Maratona de Programação é um evento realizado anualmente pela Sociedade Brasileira de Computação (SBC) desde o ano de 1996, evento este que surgiu das competições regionais classificatórias do concurso de programação Internacional Collegiate Programming Contest (ICPC).

Esta competição é destinada a estudantes de graduação e pós-graduação da área de Computação como Ciência da Computação, Engenharia de Computação, Sistemas de Informação, Matemática, etc. Os times são compostos por 3 alunos que são desafiados a resolverem o máximo possível dos 10 a 13 problemas em um tempo de 5 horas, com certos limites de tempo e memória para execução do algoritmo.

O problema abordado neste artigo apareceu na última edição da fase nacional da Maratona, em 2020. O problema ”Halting Wolf”pode ser solucionado fazendo modelagem de grafos e utilizando algum algoritmo de fluxo máximo, como Edmonds-Karp ou push-relabel [Cormen et al. 2009].

Na Seção 2 serão apresentadas as preliminares para a compreensão do algoritmo. Na Seção 3 será apresentado o algoritmo de Edmonds-Karp para computar o fluxo máximo. Na Seção 4 será apresentada a descrição do problema proposto. E na Seção 5 o problema será modelado como um grafo para posterior implementação do algoritmo apresentado na Seção 3.

2. Preliminares

Uma **rede de fluxo** é um grafo orientado $G = (V, E, c, s, t)$, onde V é o conjunto de vértices, E é um conjunto de arestas, $c : V \times V \to \mathbb{Z}_+$ é uma função chamada de capacidade, $s \in V$ é um vértice especial chamado fonte e $t \in V$ é um vértice chamado sorvedouro.

Um **fluxo** em uma **rede de fluxo** é uma função $f : V \times V \to \mathbb{Z}_+$ que satisfaz as três restrições:

- *Restrição de capacidade:* O fluxo em uma aresta não pode exceder sua capacidade, $f(u, v) \leq c(u, v)$
- *Conservação do fluxo* A soma dos fluxos entrando em um vértice v deve ser igual a soma dos fluxos saindo desse mesmo vértice v, exceto na fonte e no sorvedouro.

$$\sum_{(u,v)\in E} f(u, v) = \sum_{(v,u)\in E} f(v, u) \quad \forall v \in V \setminus \{s, t\}$$

A fonte só tem fluxo de saída, assim como também o sorvedouro só tem fluxo de entrada. Além disso, temos que:

$$\sum_{(s,u)\in E} f(s, u) = \sum_{(u,t)\in E} f(u, t)$$

O valor do fluxo em uma **rede de fluxos** é definido como $|f| = \sum_{v\in V} f(s, v)$

O problema do fluxo máximo pode ser estabelecido da seguinte maneira:

$$\max \sum_{v\in V} f(s, v)$$

$$\sum_{(u,v)\in E} f(u, v) = \sum_{(v,u)\in E} f(v, u) \quad \forall v \in V \setminus \{s, t\}$$

$$0 \leq f(u, v) \leq c(u, v)$$

3. Método de Ford-Fulkerson

Antes de apresentar o método, vamos definir o que é **capacidade residual** e **rede residual**: a capacidade residual é, para uma aresta $(u, v) \in E$, a diferença entre sua capacidade e o fluxo que passa pela aresta, ou seja, $c(u, v) - f(u, v)$. Além disso, podemos também definir, para cada aresta $(u, v) \in E$, uma aresta reversa (v, u) tal que sua capacidade seja 0 e o seu fluxo seja tal que $f(v, u) = -f(u, v)$. Assim, a rede de fluxo que considera a capacidade residual como capacidade e tem os mesmos vértices e arestas será a **rede residual**.

O método de Ford-Fulkerson funciona da seguinte forma. Primeiro o fluxo em todas as arestas é 0. Em cada iteração, o valor do fluxo será aumentado, encontrando um **caminho aumentante**. Um caminho aumentante pode ser entendido um caminho da fonte

até o sorvedouro na rede residual utilizando apenas as arestas com capacidade residual positiva. Seja C a menor capacidade residual do caminho aumentante. O fluxo total será incrementado em C e, para cada aresta (u, v) no caminho aumentante, adicionaremos $+C$ em $f(u, v)$ e $-C$ em $f(v, u)$. Quando não houver caminhos aumentantes para serem encontrados, o fluxo encontrado será máximo. O Algoritmo 1 apresenta o pseudocódigo do método de Ford-Fulkerson:

Algorithm 1 Ford-Fulkerson

```
1:  function FORD-FULKERSON(V, E, s, t)
2:      FluxoMaximo ← 0
3:      for cada aresta (u, v) ∈ E do
4:          f(u, v) ← 0
5:          f(v, u) ← 0
6:      end for
7:      while existir um caminho p da fonte até o sorvedouro na rede residual do
8:          C ← min{c(u, v) − f(u, v) : (u, v) ∈ p}
9:          FluxoMaximo ← FluxoMaximo + C
10:         for cada aresta (u, v) ∈ p do
11:             f(u, v) ← f(u, v) + C
12:             f(v, u) ← f(v, u) - C
13:         end for
14:     end while
15:     return FluxoMaximo
16: end function
```

O algoritmo Edmonds-Karp é uma implementação do método de Ford-Fulkerson, no qual o algoritmo usado para encontrar os caminhos aumentantes é a Busca em Largura (BFS) [Cormen et al. 2009]. O algoritmo de Ford-Fulkerson tem complexidade de tempo no pior caso $O(E * |f^*|)$, onde f^* é o valor do fluxo máximo encontrado pelo algoritmo, enquanto o algoritmo de Edmonds-Karp tem complexidade de tempo no pior caso $O(VE^2)$.

4. Descrição do Problema

O problema foi originalmente disponibilizado em inglês, e nesse artigo será feita uma tradução livre para melhor compreensão do problema.

Senoof ama linguagens de programação, e a única coisa que ele ama mais que usar elas é criar novas delas. Sua última invenção é a Wolf Programming Language, uma linguagem de programação muito simples que consiste somente de dois tipos de instruções. Elas são numeradas consecutivamente e escritas uma abaixo da outra para formar o programa. A execução inicia na linha 1 e continua até o programa ficar preso. Os dois tipos de instruções são:

- *"*$K\ L_1\ L_2\ L_3 \ldots L_k$*"* é um salto finito. Cada valor L_i é o número de uma instrução no programa, enquanto K indica quantas delas são especificadas. Quando um pulo finito é executado, um dos valores L_i é escolhido, e a execução continua com a instrução L_i. Mas isso não é tudo! O programa altera a instrução de salto finito

de modo a consumir o valor escolhido. Se o programa executa o salto finito sem valores disponíveis, ele fica preso e para.
- *"* L"* é um salto infinito. Quando ele é executado, o programa continua com a instrução L, deixando a instrução de salto infinito não modificada.

Eu sei, Senoof é louco, mas não é tão difícil. A Figura 1 abaixo mostra um exemplo, onde a instrução atual é indicada com um sinal ▷, e um valor consumido é indicado com um sinal de ⊔. O programa, em (a), inicia a execução na instrução 1, que é um salto finito. Suponha que o segundo valor é escolhido, isto é, a execução continua com a instrução 2 e seu valor é consumido na instrução 1, que produz a situação mostrada em (b). Como a instrução 2 é um salto infinito que salta para a instrução 3, a execução continua com essa instrução, sem consumir nenhum valor da instrução 2. Agora imagine que da instrução 3 a execução salta para a instrução 4, então para a instrução 1 e então novamente para a instrução 1, consumindo os valores correspondentes. A situação neste ponto é mostrada em (c). Como você pode ver, o programa fica preso e para, pois não existem valores disponíveis para saltar.

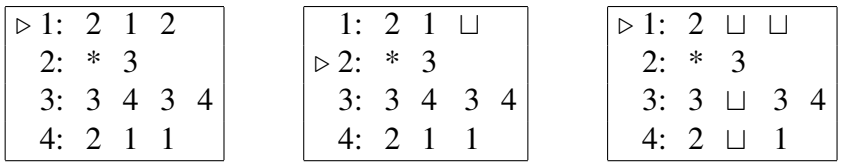

Figura 1. Exemplo de um programa em que a instrução 1 é executada duas vezes e o programa fica preso e pára

Depois de brincar um pouco, Senoof percebeu que os programas escritos em Wolf podem ser executados para sempre, o que não implica que uma dada instrução pode ser executada infinitamente. A Figura 2 mostra exemplo de um programa que pode ser executado para sempre, embora a instrução 1 possa ser executada no máximo duas vezes. Dado um programa escrito em Wolf, você deve determinar o número máximo de vezes que a instrução 1 pode ser executada.

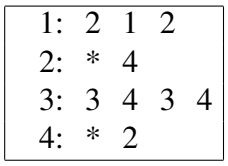

Figura 2. Exemplo de um programa em que a instrução 1 é executada duas vezes e o programa pode ser executado para sempre

Entrada

A primeira linha contém um inteiro N ($1 \leq N \leq 100$), o número de instruções que o programa tem. Cada uma das N linhas descreve uma instrução. Um salto finito é representado com um inteiro não negativo K seguido por K inteiros, $L_1, L_2, L_3, \ldots, L_k$ ($1 \leq L_i \leq N$ para $i = 1, 2, \ldots, K$). Por outro lado, um salto infinito é descrito com

o caractere "*"(asterisco) seguido por um inteiro L ($1 \leq L \leq N$). É garantido que a quantidade total de instruções mencionada em saltos finitos é no máximo 10^4.

Saída

Imprima uma única linha com um inteiro indicando o número máximo de vezes que a instrução 1 pode ser executada, ou o caractere "*"(asterisco) se a instrução 1 pode ser executada infinitamente.

5. Modelagem do problema

Nesta seção, vamos mostrar como construir uma rede de fluxo a partir da entrada do problema apresentado na seção anterior. Seja I o conjunto de instruções. Cada instrução $i \in I$ será representada por um vértice $i \in V$, exceto pela instrução 1 que será representada por dois vértices especiais, a fonte $1 \in V$ e o sorvedouro $N + 1 \in V$. Para cada instrução $i \in I$, seja $R \subseteq I$ o conjunto de instruções referenciadas em i. Se a instrução i for do tipo 1, então para cada instrução $j \in R$ com $j \neq 1$, será criada uma aresta, com peso 1, que sai do vértice que representa i para o vértice que representa j, ou seja, a aresta $(i, j, 1)$. Se a instrução i for do tipo 2, será criada uma aresta, com peso $\inf$, do vértice que representa i para o vértice que representa a única instrução $j \in R$ com $j \neq 1$, ou seja, a aresta $(i, j, \inf)$. No caso da instrução i, se ela referencia a instrução 1, adicionaremos a aresta $(i, N + 1, 1)$ se a instrução for do tipo 1, $(i, N + 1, inf)$, caso contrário.

Observe que um caminho entre o vértice 1 e o vértice $N + 1$ representa que a instrução 1 foi executada uma vez e o programa conseguiu saltar para a instrução 1 que poderá ser executada novamente. Caso o algoritmo de Fluxo Máximo encontre um caminho aumentante no grafo residual de custo inf então a instrução poderá ser executada infinitamente. Logo, se $FluxoMaximo = \inf$, a instrução 1 pode ser executada infinitamente, e o programa deverá imprimir '$*$' como resposta. Caso $FluxoMaximo \neq \inf$, a quantidade de vezes máxima que a instrução 1 pode ser executada será '$FluxoMaximo + 1$'.

6. Conclusões

Neste artigo foi apresentada uma forma eficiente de abordar o problema "Halting Wolf"que foi apresentado na fase nacional da Maratona de Programação da SBC 2020. O algoritmo de Edmonds-Karp foi utilizado no grafo que foi modelado a partir das informações do problema. A solução discutida foi implementada[1] na linguagem C++ e submetida na plataforma de juiz online CodeForces (codeforces.com). O tempo de execução nos testes foi de 31 ms, dentro do limite proposto no problema, de 250 ms, e também dentro do limite que foi proposto na competição oficial, de 100 ms. Além disso, o uso de memória foi de 4400 KB, também dentro do limite estabelecido no problema, de 1024 MB.

O estudo dos algoritmos de fluxo em grafos permite a resolução de uma ampla variedade de problemas, pois é uma poderosa ferramenta de modelagem, capaz de representar diversos problemas do mundo real.

Referências

[Cormen et al. 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms*. MIT press.

[1] https://github.com/danielvitor2d/XXVMaratonadeProgramacaoSBC/blob/main/H.cpp