TP1 – O Plano de Campanha

Algoritmos 1

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brazil

chbmleao@ufmg.br

1. Identificação

Carlos Henrique Brito Malta Leão

Matrícula: 2021039794

2. Introdução

O problema que precisa ser resolvido apresenta extrema aplicabilidade no mundo real, e trata da escolha, ou não, de n itens baseado nas preferências de m indivíduos ou escolhas. Nesse sentido, em um contexto mais específico do trabalho, um deputado federal precisa se reeleger, para isso, realizou uma pesquisa com seus seguidores para saber a opinião de cada um em relação às propostas de governo, em que cada entrevistado precisa escolher duas propostas para serem mantidas e duas para serem removidas do plano governamental. Dessa forma, deseja-se saber se existe um conjunto de propostas que agrade a todos estes seguidores simultaneamente, em que, agradar um seguidor significa manter pelo menos uma das propostas aceitas e remover pelo menos uma das propostas rejeitadas. Por fim, concluímos que esse problema é um clássico problema de **satisfatibilidade**.

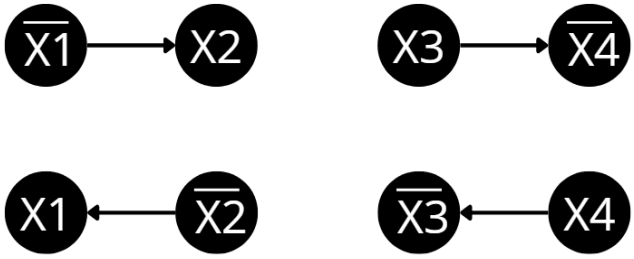
Na lógica matemática, **satisfatibilidade** é um conceito elementar da semântica. Nesse sentido, uma fórmula é satisfazível se é possível achar uma interpretação que torne a fórmula verdadeira. No aspecto do problema supracitado, consideraremos que as propostas escolhidas por cada seguidor formarão a equação que deve ser verificada, em que cada conjunto de duas propostas, escolhidas para serem mantidas ou tiradas, formam uma cláusula com dois literais. Exemplificando, se um seguidor opina que as propostas $X1$ e $X2$ devem ser mantidas e $X3$ e $X4$ devem ser retiradas, teremos a seguinte expressão booleana para verificar a **satisfatibilidade**:

$$(X1 \vee \mathrm{X2}) \wedge (\overline{\mathrm{X3}} \vee \overline{\mathrm{X4}})$$

Nesse sentido, considerando que a equação deve ser verdadeira, se $X1 = 0$, então necessariamente $X2$ precisa ser igual a 1, e vice-versa. De modo semelhante, para que a equação seja verdadeira, se $X3 = 1$ e $\overline{\mathrm{X3}} = 0$, então, necessariamente, $X4 = 0$ e $\overline{\mathrm{X4}} = 1$ e o mesmo ocorre com o estado contrário. Por fim, após aplicar essa lógica aritmética, teremos a final equação:

$$(\overline{\mathrm{X1}} \rightarrow \mathrm{X2}) \wedge (\overline{\mathrm{X2}} \rightarrow \mathrm{X1}) \wedge (X3 \rightarrow \overline{\mathrm{X4}}) \wedge (X4 \rightarrow \overline{\mathrm{X3}})$$

Com a equação booleana acima formada, podemos descobrir se ela pode ser satisfeita utilizando um grafo, em que cada proposta que deve ser mantida é representada por um vértice e cada proposta que deve ser rejeitada é representada pelo mesmo vértice negado. Dessa forma, é possível criar as arestas entre os vértices a partir de cada cláusula da equação booleana. Por exemplo, seguindo a equação booleana acima, o grafo teria um formato semelhante ao seguinte esquema:



Para descobrir se uma expressão é satisfazível ou não, existe um problema de computação muito conhecido chamado **Boolean Satisfiability Problem (SAT)**. Particularmente, a fórmula Booleana é dada na forma normal conjuntiva, que é a conjunção de múltiplas cláusulas, em que cada cláusula é a disjunção de dois literais, assim como a fórmula representada anteriormente. De forma mais específica, no caso do problema supracitado, lidamos com, exatamente, dois literais por cláusula, dessa forma, estamos lidando com um problema do tipo **2-SAT**.

Em um problema do tipo **2-SAT**, para encontrar uma solução para a equação booleana, é preciso e suficiente que, para toda variável Xi, os vértices Xi e $\overline{Xi}$ devem estar em diferentes **Componentes Fortemente Conexos (CFC)** no grafo criado. Um **Componente Fortemente Conexo** de um grafo orientado é um subconjunto C de vértices tal que para todo par de vértices u e v em C, existe um caminho orientado de u a v e vice-versa.

Em resumo, ao verificar se para todo vértice, ele e o seu complemento não estão no mesmo **CFC**, significa que a expressão booleana é satisfazível. O detalhamento dos algoritmos e estruturas de dados escolhidos está disposto na secção seguinte.

3. Modelagem

3.1 Estruturas de Dados

Primeiramente, foi preciso decidir qual a melhor forma de representar o grafo nessa situação. Nesse sentido, poderia ser utilizada uma **Matriz de Adjacência**, em que cada linha e coluna representam os vértices do grafo, em que, se o elemento da linha x e coluna y apresenta o valor 0, significa que o vértice x não aponta para y, caso apresente o valor 1, x aponta para y. Este método é funcional, porém, para acessar todos os vizinhos do vértice x é preciso percorrer todos os n elementos, além de que a complexidade de espaço é $O(n^2)$.

Dessa forma, foi utilizado uma **Lista de Adjacência**. Esta estrutura de dados utiliza outra estrutura de dados chamada **Lista Encadeada**, em que cada elemento aponta para o próximo da estrutura, formando uma lista de elementos. Assim, é criado um vetor de listas de tamanho n, em que cada posição do vetor aponta para o endereço de uma lista, assim, na posição x do vetor, existe uma lista que apresenta os vértices que o vértice x aponta. Nesse sentido, para acessar todos os vizinhos de x ao invés de iterar n elementos, essa função acessa apenas o número de vizinhos que x possui, porém, a complexidade assintótica da função não diminui e permanece linear. Por outro lado, a complexidade de espaço torna-se linear $O(n + m)$, em que n representa o número de vértices e m representa o número de arestas, em que cada lista apresenta cerca de $\frac{n}{m}$ arestas.

Além disso, foi preciso utilizar duas **Listas de Adjacência**, uma para a ordem normal do grafo, e outra com as arestas invertidas, ou seja, se no grafo normal existe a aresta $X \rightarrow Y$, no grafo invertido, existe $Y \rightarrow X$. Ademais, também foram utilizados vetores de inteiros para armazenar os vértices já visitados e a ordem de visitação. Estas estruturas terão suas funcionalidades melhores explicadas na secção seguinte.

3.2 Algoritmo

Após compreender quais são as **Estruturas de Dados** utilizadas no programa, já é possível entender como estas são utilizadas dentro do programa. Com isso em mente, a primeira coisa a ser feita é ler os valores de entrada do arquivo *input.txt* para saber o número de propostas e o número de seguidores à serem analisados. Nesse sentido, as propostas aceitas por cada seguidor representarão os vértices positivos do grafo, enquanto as rejeitadas serão os vértices negativos. Nesse sentido, precisarão ser alocadas na **Lista de Adjacência** $2n$ listas (em que n representa o número de propostas), 2 para cada proposta, uma representando a proposta aceita e outra a proposta rejeitada. Dessa forma, as listas enumeradas de 0 até $n-1$ representam os vértices positivos, enquanto as listas n até $n-2$ são os vértices negativos.

Em seguida, após alocar o espaço para a **Lista de Adjacência**, é preciso preenchê-la. A lógica dos vértices e arestas já foi melhor explicada na secção 2, mas é necessário um pouco mais de detalhamento para entender todo o processo do código. Portanto, segue os casos de transformação dos números de propostas para o preenchimento da matriz de adjacência.

- Primeiramente, como a lista de adjacência começa com o valor 0, é preciso decrementar 1 do índice de cada proposta.
- Para inserir os vértices na lista de adjacência existem duas lógicas diferentes:
    - Caso a proposta seja aceita, basta acessar a posição do índice da proposta no vetor.
    - Caso a proposta seja rejeitada, é preciso acessar a posição do índice da proposta somada ao número n de propostas.
- No tratamento de propostas aceitas X e Y, é preciso inserir na lista de adjacência:
    - $\bar{X} \rightarrow Y$, ou seja, adiciona Y na lista de vizinhos de $\bar{X}$.
    - $\bar{Y} \rightarrow X$, ou seja, adiciona X na lista de vizinhos de $\bar{Y}$.
- Já no tratamento de propostas rejeitadas $\bar{X}$ e $\bar{Y}$, é preciso inserir na lista de adjacência:
    - $Y \rightarrow \bar{X}$, ou seja, adiciona $\bar{X}$ na lista de vizinhos de Y.
    - $X \rightarrow \bar{Y}$, ou seja, adiciona $\bar{Y}$ na lista de vizinhos de X.
- O seguidor também possui a opção de deixar uma escolha em branco, ou seja, não escolher uma proposta, seja esta de mantimento ou remoção. Dessa forma, teremos dois casos possíveis:
    - Na escolha de apenas uma proposta de mantimento $X1$, teríamos a expressão booleana $(X1 \vee \mathrm{X1})$, que aplicando a lógica aritmética já detalhada anteriormente, seria $\overline{X1} \rightarrow X1$. A partir desse ponto, basta aplicar a lógica já explicada acima.
    - Na escolha de apenas uma proposta de remoção $\overline{X1}$, teríamos a expressão booleana $(\overline{X1} \vee \overline{X1})$, que aplicando a lógica aritmética já detalhada anteriormente, seria $X1 \rightarrow \overline{X1}$. A partir desse ponto, basta aplicar a lógica já explicada acima.
- Por fim, o seguidor pode não escolher ambas propostas, tanto de mantimento como de remoção. Neste caso, não é preciso fazer nenhuma alteração na **Lista de Adjacência**.

Ao fim deste processo de criação da **Lista de Adjacência**, nós já temos nosso grafo estruturado e preenchido. Portanto, já é possível inicializar o processo de caminhamento por todos os vértices. Para realizar este caminhamento, utilizaremos o algoritmo **Depth First Search (DFS).**

O algoritmo de caminhamento no grafo **DFS**, utiliza um vetor auxiliar para marcar os vértices já visitados, ou seja, o vetor tem o tamanho igual ao número de vértices do grafo. Caso o vértice já tenha sido visitado, sua posição é marcada com o valor 1, caso contrário, é marcado com o valor 0. Dessa forma, o propósito do algoritmo é marcar cada vértice do grafo como visitado, evitando ciclos, em outras palavras, marcar todos os vértices com o valor 1.

O algoritmo pode começar por qualquer vértice do grafo, que ao realizar a chamada da função **DFS** é marcado como visitado. Em seguida, a função **DFS** é chamada recursivamente para cada vértice apontado pelo vértice atual. E o processo segue, até que o vértice não tenha vizinhos, ou todos os vizinhos já tenham sido visitados. Por fim, a função adiciona o vértice em um vetor que armazena a ordem dos vértices acessados.

Este algoritmo de caminhamento no grafo é muito eficiente, já que cada vértice é visitado apenas uma vez. Nesse sentido, já ocorre a execução de n procedimentos. Além disso, para cada vértice, todos os vizinhos do mesmo são verificados se já foram visitados, ou seja, todas as arestas que apontam de cada vértice são iteradas, considerando m como o número de arestas, ocorre a execução de m instruções. Por fim, concluímos que a complexidade do algoritmo **DFS** é $O(n + m)$, em que n representa o número de vértices e m o número de arestas.

Caso o grafo seja desconexo, ou seja, não existir pelo menos um caminho entre cada par de vértices do grafo, não é possível acessar todos os vértices dependendo do vértice inicial. Para resolver esse problema, é realizado um laço externo à função que itera todos os vértices do grafo, caso um deles ainda não tenha sido visitado, a função DFS é chamada para este. Assim, o algoritmo ainda mantém a complexidade assintótica de $O(n + m)$.

Agora que já sabemos como realizar o caminhamento no grafo, precisamos definir os **Componentes Fortemente Conexos (CFC)**, para assim, verificar a **satisfatibilidade** da expressão booleana. Para isso, utilizamos um famoso algoritmo chamada **Algoritmo de Kosaraju**, que é um algoritmo que funciona em tempo linear para encontrar os **CFCs** de um grafo conexo. Portanto, o **Algoritmo de Kosaraju** auxilia na verificação de **satisfatibilidade** do nosso problema.

O **Algoritmo de Kosaraju** realiza o caminhamento por **DFS** duas vezes. Nesse sentido, a realização de um **DFS** no grafo produz uma árvore se todos os vértices são alcançados pelo vértice inicial de entrada, caso contrário produz uma floresta, formada por várias árvores. Assim, o **DFS** de um grafo com apenas um **CFC** sempre produz uma árvore. Por outro lado, o DFS de um grafo pode produzir tanto uma árvore como uma floresta quando existe mais de um **CFC**, isso depende do vértice inicial de entrada. Portanto, como podemos encontrar essa sequência de escolha de vértices que funcionam como pontos de partida do **DFS**?

Nesse sentido, se realizarmos o **DFS** de um grafo e armazenar os vértices de acordo com sua ordem de retorno, é possível garantir que esta ordem de um vértice, que se conecta a outros **CFCs** (diferentes do seu próprio **CFC**), será sempre maior que a ordem de retorno dos vértices em outro **CFC**. Dessa forma, para utilizar essa propriedade, realizamos a função **DFS** no grafo e colocamos cada vértice finalizado em um vetor, que conterá a ordem de retorno do caminhamento.

No próximo passo, é preciso inverter o grafo. Nesse sentido, utilizamos a **Lista de Adjacência Inversa**, citada na secção 3.1, em que as arestas que conectam dois vértices são invertidas. Assim, se realizarmos o caminhamento **DFS** deste grafo reverso, usando a ordem de retorno do caminhamento **DFS** ao contrário, é o necessário para encontrar qual **Componente Fortemente Conexo** cada vértice está inserido.

Para realizar este processo, utilizamos um segundo algoritmo **DFS**, que, em base, apresenta a mesma lógica do primeiro **DFS** supracitado, porém, os vértices são iterados na ordem de retorno inversa que foi registrada na primeira chamada do algoritmo. Além disso, o segundo **DFS** também recebe uma variável de contagem, para definir o número do componente conexo de cada vértice. Nesse sentido, a cada chamada da função **DFS** no laço principal, esta variável de contagem é incrementada, então todos os vértices encontrados em uma **DFS** são atribuídos com um mesmo valor diferente de outras chamadas de **DFS**. Em outras palavras, cada chamada do algoritmo de caminhamento representa um **CFC** distinto.

Por fim, ao final da chamada do segundo **DFS**, todos os vértices do grafo apresentam um **Componente Fortemente Conexo** definido. Portanto, é preciso verificar para todos os vértices se o seu complemento está no mesmo **CFC** que o próprio, caso verdadeiro, concluímos que a expressão booleana do problema é insatisfazível, caso contrário, se todos os vértices estiverem em **CFCs** diferentes de seus complementos, a expressão é satisfazível. Em outras palavras, se existe Xi, e se Xi e $\overline{Xi}$ estão no mesmo **Componente Fortemente Conexo**, a expressão booleana não é satisfazível, caso contrário, a expressão é satisfazível. Caso a expressão seja satisfazível é impresso “sim” no console, caso contrário é impresso “nao”.

Em conclusão, é possível resolver o problema de descobrir se existe um conjunto de propostas que agrade a todos os seguidores de um deputado utilizando um algoritmo com complexidade assintótica linear desenvolvido neste trabalho prático. Nesse sentido, o algoritmo como um todo apresenta complexidade de $O(n + m)$, em que n representa o número de propostas e m representa o número de arestas do grafo formado. Além disso, o programa apresenta complexidade espacial também de $O(n + m)$, já que foi utilizada a **Lista de Adjacência** como estrutura de dados principal do programa.