

Trabalho Prático 1

O Plano de Campanha

Ítalo Dell’Areti

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

italodellareti@ufmg.br

1. Introdução

O problema a ser resolvido é, dado às propostas de campanha de um deputado, as quais ele tem com intenção de abranger e satisfazer a maior quantidade de eleitores possível, determinar se existe a possibilidade de agradar todo seu eleitorado. Mas para saber a opinião de seu eleitorado, ele cria uma campanha para que seus seguidores façam 4 escolhas, aprovar 2 propostas e recusar 2 propostas, dado todas o conjunto das suas propostas. Como é difícil que se aprove as propostas dos eleitores, para atender um seguidor ao menos uma das propostas votadas a favor deve continuar no plano de campanha e ao menos uma das propostas com voto contrário deve ser retirada do plano.

Ao final, devemos conseguir determinar se é viável a realização dessa escolha, afirmando a possibilidade ou negando a existência de solução. Sendo assim, com base nas características e propriedades grafos e seus componentes, tipos de busca e ordenação, podemos implementar este problema de forma a respeitar todas as nuances da problemática. Tendo em vista disso, este documento possui como objetivo explicitar o funcionamento desta implementação.

2. Implementação

O programa foi desenvolvido utilizando a linguagem de programação C++, compilada pelo G++ da GNU Compiler Collection. Ademais, o programa foi produzido e testado em um ambiente Linux distribuição Ubuntu 20.04 LTS utilizando Visual Studio Code.

2.1 - Modelagem

A priori, tendo em vista o problema, devemos discretizá-lo e expressar suas necessidades matematicamente. Como requerido na implementação, devemos utilizar grafos em nossa modularização e como vamos encaixar essa estrutura.
















Como entrada do arquivo, temos dois números inteiros S e P, que respectivamente representam o número de seguidores que responderam a enquete

e o número de propostas do plano de campanha. Após isso, temos os votos de cada participante, aprovando 2 propostas e recusando 2 propostas. Considerando que, cada proposta pode ser aprovada e desaprovada, sim ou não, é viável se utilizar da lógica proposicional e booleana para o início da resolução. Sendo assim vamos utilizar o problema 2-SAT, o qual determina se uma fórmula booleana é satisfatível ou insatisfatível.

Mas para a construção da Forma normal conjuntiva, precisamos estabelecer algumas regras. Dado duas variáveis quaisquer P e Q, temos que :

- $\neg P$ é o complemento de p. Se x é verdadeiro, $\neg P$ é falso.
- $(P \vee Q)$ representa P OR Q .Verdadeiro quando ao menos um deles é verdadeiro.
- $(P \wedge Q)$ representa P AND Q . Verdadeiro só quando ambas proposições são verdadeiras.
- $(P \rightarrow Q)$ representa P implica Q. Verdadeiro só quando P é verdadeiro, tem que ser verdadeiro. Caso contrário, Q pode ser verdadeiro ou falso.
- $(P \Leftrightarrow Q)$ representa (P implica Q) \wedge (Q implica P). verdadeiro somente se P e Q tiverem o mesmo valor.

Diante do exposto, agora podemos ter uma melhor compreensão de como o grafo vai se estruturar, dado que S variáveis (seguidores) e P cláusulas (propostas), e simplificando as fórmulas podemos inferir que a proposição deve suceder em verdadeiro para que uma proposição seja satisfeita.

		Votos			
		Manter		Remover	
Seguidores		1  2 		3  4 	
		3  4 		1  2 	
		2  3 		1  4 	

Utilizando o exemplo fornecido, dados x_1, x_2, y_1 e y_2 , teríamos as expressões:

$$(x_1 \vee x_2) \wedge (\neg y_1 \vee \neg y_2)$$

$$(\neg x_1 \vee \neg x_2) \wedge (y_1 \vee y_2)$$

$$(\neg x_1 \vee x_2) \wedge (y_1 \vee \neg y_2)$$

Por tudo isso, agora podemos criar um gráfico direcionado de implicações resultantes simplificadas, onde cada nó é uma proposição e a aresta (u,v) é direcionada de u para v. Portanto, vamos estruturar um grafo direcionado onde cada nó é uma proposta.

Como nosso interesse está relacionado à quantidade máxima de propostas com a intenção de manter ou descartar a proposta (relação de conexões fortemente

conectadas) e não requer algo como menor distância, foi utilizado um tipo especial de DFS (depth-first search) para percorrer o grafo, o Kosaraju.

Depois do grafo estar montado e de adicionarmos todas as arestas, devemos encontrar todos os componentes relacionados no grafo, pois são eles que vão nos resultar em nossa lista de adjacências e poderemos determinar se é possível ou não chegar em uma determinação.

Para realizar esta função, vamos ter que utilizar o algoritmo de Kosaraju, pois apesar da etapa de uma busca em profundidade revelar uma componente forte do grafo, Infelizmente isto só é válido se o vértice inicial de cada nova etapa for escolhido de uma maneira específica. Sendo assim, para realizar essa escolha determinada, o algoritmo de Kosaraju começa por colocar os vértices numa certa ordem, determinada por uma busca em profundidade e, ao final de todo processo, teremos as listas de adjacência. Após contar os componentes podemos verificar os dois lados do grafo, podemos dizer se sim, ou não é possível agradar a todos.

Em resumo, construímos o grafo de acordo com as expressões (2 sat de uma expressão grande como as opiniões que estão em pares) e separando as variáveis a favor e contra em cada lado do grafo(direcionado), aplicamos o kosaraju para contar quantos componentes estão ligados fortemente(contar quantas conexões fortes cada grafo tem) e depois vemos se o número de componentes de cada par é igual, se for não é possível satisfazer(verifica se existe qualquer variável onde ela negada tem o mesmo número de componentes. Se existir, não dá para satisfazer. Pela contradição uma variável não pode assumir 2 valores na expressão).

Além disso, uma das decisões que ainda não foram citadas mas é notória, é o caso do voto 0. Tendo em vista que, quando alguém se abstém e a entrada fica 0 mas no modelo usando, nós não podemos ter apenas uma variável e também não podemos assumir que ela tá junto com algo sempre verdadeiro, por que ela pode se anular, já que é um OR e não um AND.Depreende-se, então a utilização da idempotência lógica, onde duplicamos a proposição e não alterando seu valor,sendo $(A \wedge A) \Leftrightarrow A$ e $(A \vee A) \Leftrightarrow A$

2.2 - Organização e Análise

Algumas das funções do programa são:

void Campanha::aloca_e_inicializa_estruturas() : Como o próprio nome já diz, aloca os vetores no heap e inicia as estruturas. Complexidade de espaço é $O(n)$ e complexidade de tempo $O(n)$.

void Campanha::adicionar_grafos() : Adiciona grafos, dada uma origem e um destino, invertendo a origem e destino do grafo transposto.Complexidade de espaço é $O(n)$ e complexidade de tempo $O(n)$.

void Campanha::dfs1() : Primeira parte de algoritmo de kosaraju, o qual possui a função de encontrar os componentes fortemente conectados de um gráfico direcionado. Complexidade de espaço é $O(n)$ e complexidade de tempo $O(m+n)$.

void Campanha::dfs2() : Parte 2 do algoritmo de kosaraju.Complexidade de espaço é $O(n)$ e complexidade de tempo $O(m+n)$.

bool Campanha::satisfaz() : Verifica a satisfatibilidade dada 2 proposições, a partir do número de proposições e da quantidade de seguidores.Para apurar as respostas temos 4 opções, a favor de ambas, a favor apenas da primeira, a favor apenas da segunda e contra ambas. Quando o seguidor se abster, não se importa, mas os 0 não estarão presentes em nenhum array, pois as variáveis foram duplicadas para resolver a lógica do voto nulo (idempotência lógic)

Ademais, temos uma struct chamada Campanha, na qual temos como elementos seguidores e propostas como inteiros,um ponteiro para o grafo e para o grafo transposto, 2 ponteiros booleanos para os vértices visitados do grafo e o visitado transposto,um ponteiro para os componentes e um inteiro para a quantidade de componentes que começa com 1.

2.3 - Estrutura de dados

No princípio, para criar as estruturas eram somente arrays grandes com valor determinado.Para usar a complexidade de espaço $2n$ teria que criar com new, inicializar os valores manualmente e depois dar delete. Mas se compreendesse mais de 50 mil variáveis poderia dar segfault.

```
constexpr int TAM = 100000;

struct Campanha {
    const int seguidores;
    const int propostas;

    std::vector<int> grafo[TAM];
    std::vector<int> grafo_transposto[TAM];
    std::stack<int> pilha;
    bool visitados[TAM] = {false};
    bool visitados_transposto[TAM] = {false};
    int componentes[TAM] = {0};
    int quantidade_componentes = 1;
```

Como era anteriormente antes da modificação

Desse modo, para resolver isso foi decidido alocar $2n+1$ na heap, deixando mais robusto e sendo a única limitação do programa a ram do computador.

```
Campanha::Campanha(int seguidores, int propostas) : seguidores(seguidores), propostas(propostas) { aloca_estruturas(); }

Campanha::~Campanha() {
    delete[] grafo;
    delete[] grafo_transposto;
    delete[] visitados;
    delete[] visitados_transposto;
    delete[] componentes;
}
```

Graças a esta nova implementação, o programa funciona com qualquer número de variáveis e também fica otimizado para poucas variáveis também, por exemplo 100, vai utilizar apenas 201 posições em vez de 100000, e funcionando para entradas maiores que 49999, mais eficiente que colocar um número arbitrário que fosse grande o suficiente para abstrair isso.

Instruções para compilação e execução

Para executar o programa basta dar um comando **make** para executar ou **make clean** para remover objetos e o executável. Além disso, o arquivo de texto de entrada **deve estar no diretório raiz da pasta do programa para o seu funcionamento. Sempre que mudar quaisquer valores de entrada**, você deverá executar o comando **./tp1 < arquivo_de_entrada.txt** para atualizar os valores da saída. Para salvar os valor de saída em um arquivo de saída basta digitar o comando

./tp1 < arquivo_de_entrada.txt > arquivo_de_saida.txt

Vídeo com a implementação funcionando

<https://www.dropbox.com/s/tz3z69ytr369dn6/TP%201%20ALG%20FINAL.mp4?dl=0>

<https://drive.google.com/file/d/10WYHXAnpw2GDjxJSqPqHZkUAxQ5kBZNm/view?usp=sharing>

Bibliografia

[Geeks for geeks: Depth First Search or DFS for a Graph](#)

[Geeks for geeks: 2-Satisfiability \(2-SAT\) Problem](#)

[USP IME: Algoritmo de Kosaraju-Sharir para componentes fortes](#)

[Geeks for geeks: Strongly Connected Components](#)