Problema dos Cliques

Link do Repositório Github
https://github.com/jpparreirap/Analise_de_Algoritmos_2020/tree/main/Proj_Final

João Paulo Parreira Peixoto
Universidade Federal de Roraima – Campus Paricarana
Ciência da Computação
Boa Vista, RR
jpparreirap@gmail.com

João Pedro Oliveira Silva
Universidade Federal de Roraima – Campus Paricarana
Ciência da Computação
Boa Vista, RR
joaopedro2195@gmail.com

***Resumo* — Atualmente o problema do clique, é um dos temas que mais vem sendo discutido. Nesse sentido, este trabalho visa apresentar uma introdução abrangente sobre o que é esse problema e quais são suas aplicações no mundo real, dando enfoque às redes sociais, que é onde este problema é mais comumente usado. O tipo de problemática abordado neste trabalho é sobre cliques máximos, que nele consiste em encontrar o maior número possível de vértices em um dado grafo, sendo este maior número um clique. Esse problema normalmente é utilizado para sistemas de recomendação, de produtos, mercadorias, amigos em comum, gostos e preferências em redes sociais. Tendo em vista essa descrição, dissertamos sobre o funcionamento do algoritmo de Bron-Kerbosch, no qual nos baseamos para a implementação do algoritmo para tentar solucionar este problema. Durante o relatório também apresentamos sobre os Tweets coletados e Benchmarks, os quais seriam usados para uma posterior análise de tempo de execução, números máximos de cliques, memória e iterações executadas.**

Palavras-Chaves — Problema do Clique, Aplicações, Tweets, Redes Sociais, Bron-Kerbosch, Benchmarks, Algoritmo, Máximo, Grafo, Vértice

***Abstract* — Currently, the click problem is one of the most discussed topics. In this sense, this work aims to present a comprehensive introduction about what this problem is and what its applications are in the real world, focusing on social networks, which is where this problem is most commonly used. The type of problem addressed in this work is about maximum clicks, which consists of finding the largest possible number of vertices in a given graph, this largest number being a click. This problem is usually used for recommendation systems, products, merchandise, mutual friends, tastes and preferences on social networks. In view of this description, we talk about the functioning of the Bron-Kerbosch algorithm, on which we base ourselves on the implementation of the algorithm to try to solve this problem. During the report we also presented about the collected Tweets and Benchmarks, which would be used for a later analysis of the execution time, maximum number of clicks, memory and iterations performed.**

Palavras-Chaves — Clique Problem, Applications, Tweets, Social Networks, Bron-Kerbosch, Benchmarks, Algorithm, Maximum, Graph, Vertex.

.

I. Introdução

O problema do clique no âmbito da ciência da computação, é um dos problemas mais conhecidos e discutidos atualmente. Ele possui como objetivo, encontrar cliques, que são denominados subconjuntos de vértices, todos adjacentes uns aos outros, podendo ser chamados até de subgrafos completos em um grafo. As formulações mais comuns para esse problema é o problema do número máximo de cliques, consiste em encontrar um ou mais cliques máximos (um clique com o maior número possível de vértices) em um dado grafo[1][2].

Atualmente essa problemática tem sido frequentemente abordada, e com isso a fez possuir diversas aplicações no mundo real. O principal dessas, seriam as redes sociais, onde os vértices do grafo representam as pessoas e as bordas do grafo representam o conhecimento mútuo umas das outras. Então nesse caso, um clique representa um subconjunto de pessoas que se conhecem, e esses vários algoritmos existentes para encontrar cliques podem ser usados para um sistema de recomendação, de produtos, mercadorias, amigos em comum, gostos e preferências nas redes sociais, além de várias outras aplicações que podem ser feitas nessas redes[1][2].

Além dessa aplicação, o problema do clique também tem muitas aplicações em outras áreas de atuação, como bioinformática e química computacional, entre outras[2].

Um dos problemas existentes neste assunto, seria os K-cliques, onde ao se admitir um valor para K muito alto, como 1.000 ou 10.000, iremos obter grafos extremamente grandes, e consequentemente o conjunto a ser verificado também, sendo tratado como um problema exponencial. O que também inclui ao tempo de execução para um problema dessa dimensão podendo ser exponencial.

O problema do clique é NP-Completo, pois podemos dizer que este problema está em NP. Além disso, a problemática do clique é redutível, podendo ser resolvido em tempo polinomial, o que nos infere que este problema é do tipo NP-Difícil. Ao se tratar desta redutibilidade, também podemos dizer que este problema pode ser reduzido a nível de SAT[4].

"Um algoritmo guloso começa com uma solução para o subproblema muito pequeno e aumenta-o sucessivamente para uma solução para o grande problema. O aumento é feito de forma 'gulosa', ou seja, prestando atenção ao ganho de curto prazo ou local, sem levar a uma boa solução de longo prazo ou global. Como na vida real, algoritmos gulosos às vezes levam à melhor solução, às vezes levam a soluções muito boas, e às vezes levam a soluções ruins. O truque é determinar quando será guloso." (Panberry, Ian. Problems on Algorithms, 1995)[6].

Um algoritmo guloso escolhe, em cada iteração, o objeto mais "apetitoso" que vê pela frente. O objetivo escolhido passa a fazer parte da solução que o algoritmo constrói. Estratégia gulosa é aquela por um montanhista que decide caminhar sempre "para cima", na direção de "maior subida", na esperança de assim chegar ao pico mais alto da montanha. Embora algoritmos gulosos pareçam

naturalmente corretos, a prova de sua correção é, em geral, difícil e sutil.[6][7]

Características que diferenciam um algoritmo guloso:

- abocanha a alternativa mais promissora (sem explorar as posteriores);
- é muito rápido,
- nunca se arrepende de uma decisão já tomada,
- não tem prova de correção simples.

De um algoritmo de programação dinâmica:

- explora todas as alternativas (mas faz isso de forma eficiente),
- é muito lento,
- a cada iteração, pode se arrepender de decisões tomadas anteriormente,
- tem prova de correção simples.[5][6]

II. Algoritmo de Bron-Kerbosch

Com base nessas informações, o algoritmo que nos inspiramos e escolhemos para a representação desta problemática de cliques máximos foi o algoritmo de Bron-Kerbosch. Na figura abaixo podemos visualizar o pseudocódigo deste algoritmo.

```
Algorithm CLIQUES(R, P, X)
1.  if P = ∅ and X = ∅
2.     then PRINT(R)
3.  for v ∈ P
4.        do CLIQUES(R ∪ {v}, P ∩ N(v), X ∩ N(v))
5.           P ← P − {v}
6.           X ← X ∪ {v}
```

Figura 1. Algoritmo de Bron-Kerbosch para enumeração de cliques máximos.

A lógica por trás do algoritmo de Bron-Kerbosch, conforme mostrado na Figura 1, consiste em primeiramente, como parâmetro da função é recebido três listas ou vetores, que irão representar subconjuntos de vértices de um dado grafo.

Esses parâmetros são nomeados como R, P e X, respectivamente. Sendo R, os nós remanescentes, do inglês, remaining nodes. P, como os cliques potenciais, do inglês, potential clique. E por fim, X, sendo a lista/vetor de nós não mais analisados, pulados, do inglês, skipped nodes.

Em seguida é realizada a verificação de se a lista/vetor dos cliques potenciais estiver vazia e se a lista/vetor de nós não analisados também estiver vazia, então esse subconjunto de nós remanescentes atual é considerado e contado como um clique.

Caso contrário ele continuará analisando e pegando para cada nó pertencente ao subconjunto de cliques potenciais, então ele realizará uma chamada recursiva, e nela passará como primeiro argumento a união do nodo atual, corrente, analisado com os nós remanescentes. Como segundo argumento, a interseção entre os vizinhos(Neighbours) do nó em análise com o subconjunto de cliques Potenciais. E como terceiro argumento, a interseção entre os vizinhos do nó analisado com o subconjunto de nós skipados.

Tendo realizado a chamada recursiva, então ele remove o nó atual analisado, do subconjunto de P, dos cliques potenciais, já que o mesmo já foi analisado. E então, adiciona, realiza a união desse nó com o subconjunto de nós skipados.

De formas gerais esse seria o funcionamento do algoritmo de Bron-Kerbosch. Vale ressaltar que a complexidade para o algoritmo descrito, é $O(3^{n/3})$ no pior caso de tempo de execução em um grafo gigante de n nós[8].

III. Tweets Coletados

Para coletarmos os tweets utilizamos o website Kaggle, que nele existem vários projetos de pesquisa já realizados e com isso vem os conjuntos de dados, ou seja, tweets já extraídos. Nele encontramos o dataset chamado "COVID19 Tweets", que devido aos acontecimentos atuais, é um assunto bem recente que tem sido bastante discutido, então optamos por utilizá-lo como referência aos problemas que viemos sofrendo ao longo desses tempos[9].

Como esta coleta de tweets, continha muitos dados que para nós não eram tão relevantes e além desses, muitos com campos vazios ou tipos de escritas com caracteres especiais, tivemos que realizar algumas sequências de filtros para limpar este dataset de forma que melhor nos atendesse.

Após feito isso, para definir o conjunto que deveria ser analisado como cliques, utilizamos as hashtags como chaves de coletas para identificar a relação entre eles e com isso foi possível a definição do conjunto que seria analisado como cliques.

IV. Conjunto de dados (dataset) utilizado

O conjunto de dados que nos baseamos foi de um professor, da universidade Beijing University of Aeronautics and Astronautics, chamado Ke Xu.

O dataset intitulado como "BHOSLIB: Benchmarks with Hidden Optimum Solutions for Graph Problems (Maximum Clique, Maximum Independent Set, Minimum Vertex Cover and Vertex Coloring)", é uma base de dados com soluções ótimas ocultas para problemas de grafos. E nela contém bases de testes para diversas aplicações como, conjunto independente máximo, cobertura mínima do vértice, coloração e por fim clique máximo, que é a aplicação que estamos abordando neste trabalho[3].

O dataset se mostrou ser de extrema qualidade, porém não conseguimos realizar os testes em cima dele, utilizando nosso próprio algoritmo, por problemas que nos deparamos com acesso de memória indevida que o nosso código estava tendo.

Porém realizamos uma análise rápida do próprio conjunto e com os resultados obtidos pelo autor Ke Xu. Esses resultados podem ser melhor observados na imagem abaixo.

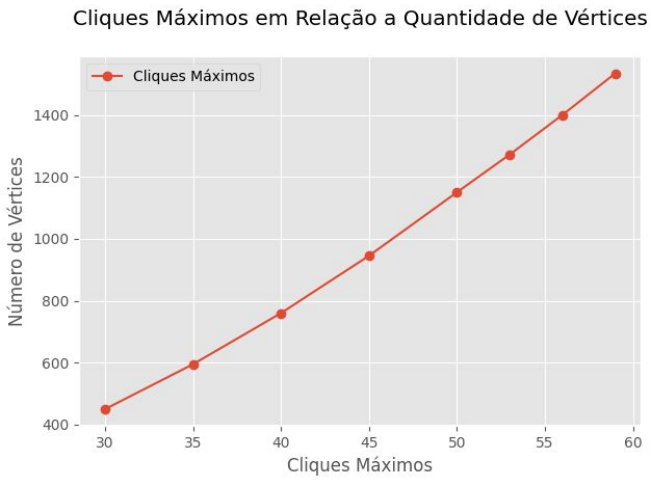

Figura 2. Gráfico da relação entre cliques máximos e quantidade de vértices.

Na Figura 2, podemos perceber e reafirmar a questão que abordamos anteriormente na introdução, sobre o problema do clique ser um problema exponencial para um dado grafo com n vértices. E nessa imagem essa percepção fica fácil pois na primeira entrada o número de vértices é 450 possuindo 30 cliques máximos e na proporção em que o número desses vértices aumentam, o número máximo de cliques também aumentam. A variação mostrada acima para vértices é 450, 595, 760, 945, 1150, 1272, 1400 e 1534, e para os cliques máximos respectivamente em relação aos vértices é 30, 35, 40, 45, 50, 53, 56, e 59 [3].

V. ALGORITMO IMPLEMENTADO E UTILIZADO NO TRABALHO

O algoritmo que foi desenvolvido por nós, foi feito na linguagem C++, utilizando o Visual Studio Code v1.51.1(user setup) como editor e mingw32-gcc-g++ v9.2.0-2 como compilador, no sistema operacional Microsoft Windows 10 Home (64-bits), utilizando o processador AMD Ryzen 5 3600 6-Core Processor 3.60GHz.

```
#include <iostream>
#include <list>
#include <iterator>
#include <algorithm>
using namespace std;

const int tam = 6;
int iteracoes = 0;
list<int> grafo[tam], all_nodes;
```

Figura 3. Variáveis globais e bibliotecas utilizadas

A iostream é a biblioteca básica do C++ para fazer a chamada das funções bases da linguagem, a list foi utilizada para criar as variáveis do tipo lista onde serão armazenados todos os vértices do grafo, o iterator foi utilizado para percorrer cada um dos elementos que contém em uma lista e o algorithm para utilizar as funções matemáticas de união e interseção.

```
int main() {

    leitura();

    imprime_Grafo();
    cout << endl;

    int total_cliques = find_cliques({},all_nodes,{});
    cout << "Total cliques found: " << total_cliques << endl;
    cout << "Iteracoes: " << iteracoes << endl;

    return 0;
}
```

Figura 4. Execução principal do programa

No momento em que o programa é inicializado, o algoritmo chama a função de "leitura" para poder ler os dados que contém todas as arestas do grafo analisado. Em seguida para uma melhor visualização para saber se o grafo está correto, é feito uma verificação imprimindo-o conforme a posição que cada cadeia de vértices estão em cada espaço de memória da variável (ex: lista[0]={1, 2, 3}, [1]={2, 5, 6}, ...). Em seguida é chamada a função principal do programa que é o que vai fazer a seleção dos cliques encontrados no grafo que foi lido, por fim será impresso o número de cliques encontrados e o número de iterações que foram feitas ao longo das chamadas recursivas da função "find_cliques".

```
void leitura() {
    FILE *arq;
    arq = fopen("teste.txt", "r");
    int v1, v2, nodes = 0;

    if(arq == NULL)
        cout << "Erro, nao foi possivel abrir o arquivo";
    else {
        while ( (fscanf(arq, "%d %d\n", &v1,&v2))!=EOF) {
            addEdge(grafo, v1-1, v2-1);
            if(v1 != nodes){ all_nodes.push_back(nodes++); }
        }
    }
    fclose(arq);
}
```

Figura 5. Função para a leitura de dados das arestas de cada vértice

Nessa função, sendo a mais importante para a montagem do grafo, é feita a instância da estrutura "FILE" no ponteiro "arq" que recebe a função para abrir o arquivo de texto e fazer somente a leitura do mesmo. As variáveis "v1" e "v2" são as que armazenarão as arestas de cada vértice, sendo no documento de texto composto somente por números (várias linhas e 2 colunas), chamando a função "addEdge" para adicionar a aresta na lista, a variável "nodes" será a responsável para colocar todos os vértices que existem no documento. E por fim a função para fechar o arquivo é chamada para finalizar a instância de "FILE".

```
void addEdge(list<int> adj[], int v1, int v2) {
    adj[v1].push_back(v2);
    //adj[v2].push_back(v1);
}
```

Figura 6. Função que adiciona arestas, criando o grafo propriamente dito

A função recebe uma lista de tamanho variável e dois valores inteiros, ao chamar a função "push_back" ela irá inserir o valor armazenado em "v2" na lista["v1"]. A parte comentada só serve para caso as informações do arquivo de

texto venham somente com as arestas existentes (ex: {1,2}, {1,3}) e não onde cada vértice está ligado com outro vértice (ex: {1,2}, {1,3}, {2,1}, {3, 1}).

```
void imprime_Grafo() {
    for (auto n : grafo) {
        cout << "[";
        for (auto m : n) cout << ' ' << m;
        cout << " ]" << endl;
    }
}
```

Figura 7. Função para imprimir o grafo construído

Nessa função é feito um for que percorre os índices da lista grafo e com o valor dos índices em "n", o "m" fará o trabalho de percorrer a lista que está armazenado dentro do índice "n", fazendo a impressão do grafo completo, onde "n" é o vértice e "m" são seus vizinhos.

```
int find_cliques(list<int>potential_clique, list<int>remaining_nodes, list<int>skip_nodes) {
    iteracoes++;
    list<int> rm = remaining_nodes;
    if(remaining_nodes.size() == 0 and skip_nodes.size() == 0) {
        cout << "This is a clique: ";
        imprime_lista(potential_clique);
        cout << endl;
        return 1;
    }

    int found_cliques = 0;

    list<int>::iterator node;
    for (node = rm.begin(); node != rm.end(); node++) {···

    return found_cliques;
}
```

Figura 8. Função para encontrar todos os cliques presentes no grafo

Essa função é o coração do algoritmo, que utiliza a lógica de Bron-Kerbosch, que localiza todos os cliques máximos. Primeiramente é feita uma incrementação em "iterações" para retornar no final o número de vezes que a função foi chamada, para sinalizar que o clique foi encontrado, tanto a lista dos nós remanescentes quanto dos nós ignorados precisam estar vazias para satisfazer a condição do IF dentro da função, finalizando assim a chamada recursiva.

```
list<int>::iterator node;
for (node = rm.begin(); node != rm.end(); node++) {
    list<int> new_potential_clique, new_remaining_nodes, new_skip_list, aux = {*node}, temp;

    set_union(potential_clique.begin(), potential_clique.end(),
              aux.begin(), aux.end(),
              back_inserter(new_potential_clique));

    set_intersection(remaining_nodes.begin(), remaining_nodes.end(),
                     grafo[*node].begin(), grafo[*node].end(),
                     back_inserter(new_remaining_nodes));

    set_intersection(skip_nodes.begin(), skip_nodes.end(),
                     grafo[*node].begin(), grafo[*node].end(),
                     back_inserter(new_skip_list));

    // for (auto n : remaining_nodes) {
    //     for (auto m : grafo[*node]) {
    //         if(n == m) new_remaining_nodes.push_back(m);
    //     }
    // }
    // for (auto n : skip_nodes) {
    //     for (auto m : grafo[*node]) {
    //         if (n == m) new_skip_list.push_back(m);
    //     }
    // }


    found_cliques += find_cliques(new_potential_clique, new_remaining_nodes, new_skip_list);
    remaining_nodes.remove(*node);
    set_union(skip_nodes.begin(), skip_nodes.end(),
              aux.begin(), aux.end(),
              back_inserter(temp));
    skip_nodes = temp;
}
```

Figura 9. Laço de repetição para filtrar os nós dentro de cada lista e realizar as operações matemáticas

Dentro do laço FOR, a variável "node" será do tipo lista atuando como uma iterator, para percorrer corretamente cada elemento dentro da lista dos nós remanescentes. Em seguida, serão criadas novas variáveis do tipo lista para armazenar todas as filtragens feitas nas operações matemáticas de união e interseção. A parte comentada do código é para mostrar exatamente como as interseções são feitas dentro do algoritmo. Então, finalmente é feita a chamada recursiva do algoritmo para poder aplicar mais filtragens até conseguir achar o clique, depois que todas as chamadas recursivas são feitas, o nó que foi analisado é removido da lista de nós remanescentes e adicionado na lista de nós ignorados. Ao final é retornada a quantidade de cliques que existem no grafo (Figura 10).

```
[ 1 2 4 ]
[ 0 2 3 5 ]
[ 0 1 3 5 ]
[ 1 2 4 5 ]
[ 0 3 ]
[ 1 2 3 ]

This is a clique: [ 0 1 2 ]

This is a clique: [ 0 4 ]

This is a clique: [ 1 2 3 5 ]

This is a clique: [ 3 4 ]

Total cliques found: 4
Iteracoes: 23
```

Figura 10 - Resultado do grafo de exemplo utilizado

REFERENCES

[1] DDA - Max Cliques, Tutorialspoint. Disponível em: <https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_max_cliques.htm>.

[2] Clique Problem. Wikipedia, 2020. Disponível em: <https://en.wikipedia.org/wiki/Clique_problem>.

[3] K. Xu, "BHOSLIB: Benchmarks with Hidden Optimum Solutions for Graph Problems (Maximum Clique, Maximum Independent Set, Minimum Vertex Cover and Vertex Coloring)", 1 de dezembro de 2014. Disponível em: <http://sites.nlsde.buaa.edu.cn/~kexu/benchmarks/graph-benchmarks.htm>.

[4] NP-Completo. Wikipedia, 2020. Disponível em: <https://pt.wikipedia.org/wiki/NP-completo>.

[5] Parberry, Ian. Problems on Algorithms, Prentice Hall, 1995.

[6] Feofiloff, Paulo. Algoritmos gulosos, 2020. Disponísvel em: <https://www.ime.usp.br/~pf/analise_de_algoritmos/bib.html#IP>.

[7] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 2nd edition, MIT Press & McGraw-Hill, 2001.

[8] Behar R; Sara C. Finding All Maximal Connected s-Cliques in Social Networks, 2018. Disponível em: <https://openproceedings.org/2018/conf/edbt/paper-28.pdf>.

[9] Preda, Gabriel. COVID19 Tweets, 2020. Disponível em: <https://www.kaggle.com/gpreda/covid19-tweets>