Implementando Uma Heurística para o Problema do Clique Máximo

Iolanda Chagas Costa Paiva
Instituto Metrópole Digital - UFRN
iccpaiva@gmail.com

INTRODUÇÃO

O problema do clique máximo é um problema dentro do contexto de grafos. Ele se caracteriza em achar o maior subgrafo G' de um grafo G não direcional, tal que todos os vértices desse subgrafo sejam vizinhos uns dos outros.

O problema do clique máximo é um problema de decisão NP-difícil, o que implica que todo problema na classe NP pode ser reduzido polinomialmente a ele. No entanto, por ser NP-difícil, não existe um algoritmo que o resolva em tempo polinomial de forma exata, o que existem são heurísticas. Além disso, uma característica importante sobre esse problema é que caso seja possível encontrar um algoritmo aproximativo que o resolva, isso é provar que NP = P.

Para este relatório, apresenta-se algumas heurísticas desenvolvidas ao longo dos anos que atacam esse problema de maneiras diferentes, bem como traz a implementação da heurística de Tomita e Seki com a comparação de quantas execuções de ciclo são necessárias realizar para obter o valor mais aproximado do clique máximo.

FUNDAMENTAÇÃO TEÓRICA

Heurísticas são formas diferentes de enxergar um problema e tentar resolvê-lo dentro de um cenário estabelecido, o que significa que heurísticas são imprecisas, mas são uma forma eficiente de tratar problemas que não podem ser resolvidos em tempo polinomial.

O problema do clique máximo

O problema do clique máximo é uma extensão do problema do clique que passou a ser debatido em meados de 1950 por Luce e Perry[1]. Dado um grafo G não orientado, o que é chamado de clique é um subgrafo G’ tal que para cada vértice desse subgrafo, esse vértice tem uma aresta incidente a cada outro vértice do subgrafo.

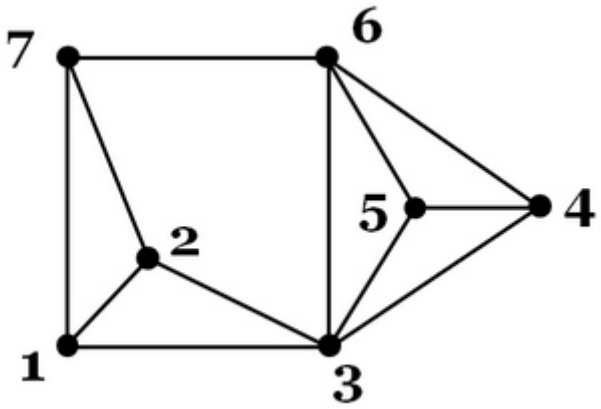


Figura 1. Um exemplo de cliques

fonte:<https://stackoverflow.com/questions/7297374/maximal-and-maximum-cliques>

Na figura 1, tem-se o grafo não orientado que nomeia-se G, um exemplo de clique desse grafo G poderia ser o conjunto de vértices (6, 4, 5) e as arestas que o ligam. Outro clique é o (4,3,5) e suas arestas.

O problema do clique máximo quer achar o maior entre todos os cliques de um grafo, então olhando novamente para a figura 1, pode-se identificar que o maior clique entre eles é o subgrafo (6,5,4,3) e as arestas que os ligam. Apesar disso ser facilmente visualmente identificado em um grafo pequeno como esse, o problema pode escalar bastante e isso se torna uma tarefa não trivial.

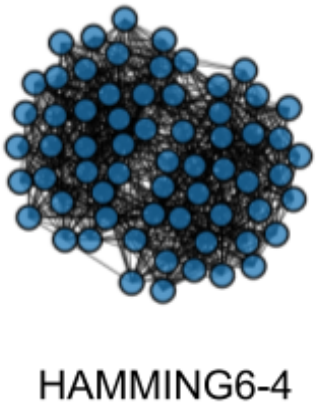


Figura 2. O grafo Hamming6-4 retirado de Network Data Repository

fonte: Ryan A. Rossi e Nesreen K. Ahmed

O grafo da figura 2 é um claro exemplo de quão complexo pode ser a identificação do clique máximo quando se tem uma situação que o grafo contém mais de 60 vértices e 700 arestas.

Então para automatizar esse processo algoritmos começaram a ser propostos. O primeiro algoritmo relacionado ao clique foi proposto em 1957 por Harary & Ross. Quanto

ao problema do clique máximo, destacam-se os algoritmos de Carragham e Pardalos, a heurística de Ostegard e o algoritmo de Tomita e Seki.

O algoritmo de Carragham e Pardalos

O algoritmo de Carragham e Pardalos [2] foi proposto em 1990, e em sua época era o algoritmo exato mais eficiente para resolver o problema, como é apontado em seu artigo.

O algoritmo dos autores trabalha com a densidade do grafo, um grafo mais denso é um grafo em que os vértices são mais conectados. Quanto mais conexões têm um grafo, mais complicado é para encontrar o clique máximo, pois mais comparações devem ser feitas.

A proposta dos autores é primeiramente ordenar os vértices do grafo por grau decrescente quando a densidade daquele grafo for menor que uma constante k. Isso já que vértices que têm um grau maior, têm por consequência uma probabilidade maior de estarem dentro do clique máximo, pois eles se conectam com mais vértices que a média. Mas para grafos mais densos, essa ordenação se torna trivial.

Após essa primeira etapa, a ideia é tentar formar cliques seguindo a ordem definida. Então percorrendo esse conjunto ordenado, deve-se passar por cada vértice o tomando como v_1 e tentando formar cliques a partir dele. O procedimento é, analisando os vizinhos de v_1 que é o conjunto $\Gamma(v_1)$, escolher o $v_2 \in \Gamma(v_1)$ e o adicionar ao clique que está sendo formado, após isso, continuar tentando encontrar um $v_k \in \Gamma(v_1) \cap \Gamma(v_2) \cap \ldots \cap \Gamma(v_{k-1})$ e seguir esse processo recursivamente.

Note que isso deve ser feito até que haja a intersecção vazia entre os elementos vizinhos de cada vértice dentro do clique ou então até que a união de um elemento no clique não suporte vizinhos suficientes para ultrapassar o tamanho do clique máximo. Essa função deve ser executada recursivamente, já que é pretendido analisar todos os cliques possíveis, daí que vêm o custo massivo. Abaixo pode ser visto um pseudocódigo para o algoritmo do Carragham e do Pardalos.

```
function clique(U, size)
 1:     if |U| = 0 then
 2:         if size > max then
 3:             max:=size
 4:             New record; save it.
 5:         end if
 6:         return
 7:     end if
 8:     while U ≠ ∅ do
 9:         if size + |U| ⩽ max then
10:             return
11:         end if
12:         i:=min{j | v_j ∈ U}
13:         U:=U\{v_i}
14:         clique(U ∩ N(v_i), size + 1)
15:     end while
16:     return
function old
17:     max:=0
18:     clique(V, 0)
19:     return
```

Figura 3. O algoritmo de Carragham e Pardalos
fonte: A fast algorithm for the maximum clique problem

O algoritmo de Ostegard

O algoritmo proposto pelo ostegard [3] está ilustrado na figura 4. Dado um conjunto de vértices S de tamanho n, considere que S_i é o conjunto de todos os vértices da posição i em S até n. Por exemplo, se $S = \{2, 3, 5, 6\}, S_2 = \{5, 6\}$e $S_1 = \{3, 5, 6\}$.

A função new proposta por ostegard, irá percorrer os conjuntos de S_n até S_1 obtendo o clique dentro do conjunto $S_i$que contém o primeiro elemento de S_i(o elemento s_i). Assim, é feita a intersecção entre S_i e $\Gamma(s_i)$. O array c irá armazenar o clique máximo encontrado em cada conjunto S_i.

Note que a ordem de execução desse algoritmo é inversa ao algoritmo passado, o autor explica que essa implementação possibilida uma nova estratégia de pruning. Além da comparação que já existia antes $size + |U| \leq max$, agora é possível ignorar casos em que $size + c[i] \leq max$.

Isso se dá pelo fato de que $c[i].size() = c[i+1].size()$ ou $c[i].size() = c[i+1].size() + 1$, já que S_i e S_{i+1}tem a diferença de um elemento somente. Assim, considerando v_j como elemento $i + 1$ de S_i, e como S_j já terá sido computado, é possível pular casos em que $size + c[i] \leq max$.

```
function clique(U, size)
 1:     if |U| = 0 then
 2:         if size > max then
 3:             max:=size
 4:             New record; save it.
 5:             found:=true
 6:         end if
 7:         return
 8:     end if
 9:     while U ≠ ∅ do
10:         if size + |U| ⩽ max then
11:             return
12:         end if
13:         i:=min{j | v_j ∈ U}
14:         if size + c[i] ⩽ max then
15:             return
16:         end if
17:         U:=U\{v_i}
18:         clique(U ∩ N(v_i), size + 1)
19:         if found = true then
20:             return
21:         end if
22:     end while
23:     return
function new
24:     max:=0
25:     for i:=n downto 1 do
26:         found:=false
27:         clique(S_i ∩ N(v_i), 1)
28:         c[i]:=max
29:     end for
30:     return
```

Figura 4. O algoritmo de Ostegard

fonte: A fast algorithm for the maximum clique problem

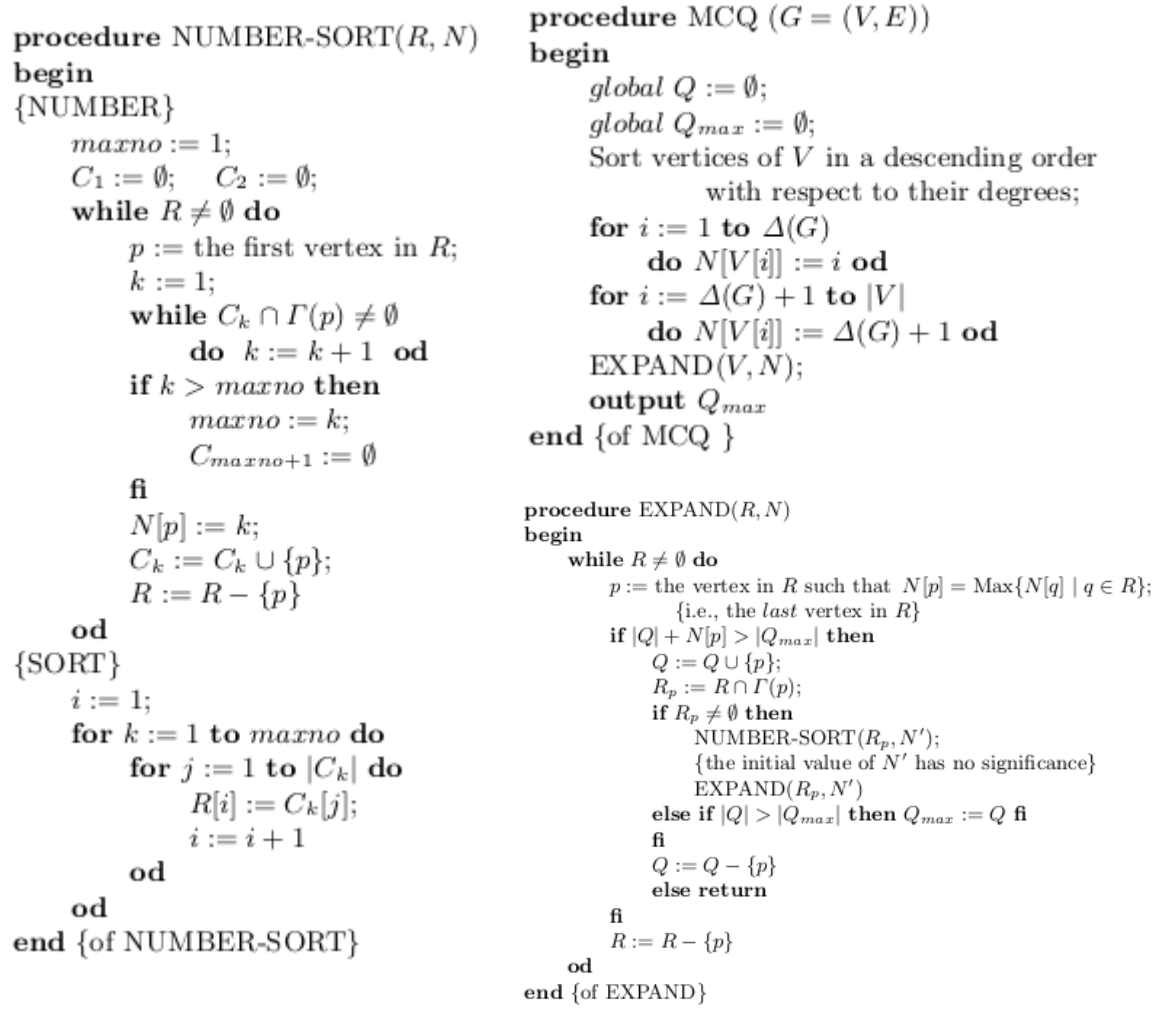
O algoritmo de Tomita e Seki

O algoritmo de Tomita e Seki[4] é uma heurística branch e bound que se mostrou mais eficiente que qualquer outra quando lançada em 2003. O algoritmo utiliza os conceitos de coloração de vértices para encontrar o clique máximo do grafo.

A relação entre coloração de grafos e cliques é que, na coloração, não existe vértice vizinho que tenha a mesma cor, assim, em um clique, todos os vértices têm cores diferentes. Esse conceito é utilizado como uma estratégia de prunning do algoritmo de Tomita e Seki, já que o tamanho do clique máximo será menor ou igual a quantidade total de cores utilizadas para colorir um grafo.

O algoritmo proposto pelos autores segue a ordem de execução ilustrada na figura abaixo. A função MCQ ordena inicialmente os vértices com base em seu degree e colore

inicialmente o grafo G, note que as cores variam de 1 ao grau máximo. Após isso, é chamado a função Expand. A decisão de adicionar uma ordenação inicial no MCQ, foi devido aos outros algoritmos, como o de carragham e pardalos, que obtiveram melhores resultados ao ordenar inicialmente os vértices com respeito ao grau.



```
procedure NUMBER-SORT(R, N)
begin
{NUMBER}
    maxno := 1;
    C1 := ∅;    C2 := ∅;
    while R ≠ ∅ do
        p := the first vertex in R;
        k := 1;
        while Ck ∩ Γ(p) ≠ ∅
            do  k := k + 1  od
        if k > maxno then
            maxno := k;
            Cmaxno+1 := ∅
        fi
        N[p] := k;
        Ck := Ck ∪ {p};
        R := R − {p}
    od
{SORT}
    i := 1;
    for k := 1 to maxno do
        for j := 1 to |Ck| do
            R[i] := Ck[j];
            i := i + 1
        od
    od
end {of NUMBER-SORT}
```

```
procedure MCQ (G = (V, E))
begin
    global Q := ∅;
    global Qmax := ∅;
    Sort vertices of V in a descending order
            with respect to their degrees;
    for i := 1 to Δ(G)
        do N[V[i]] := i od
    for i := Δ(G) + 1 to |V|
        do N[V[i]] := Δ(G) + 1 od
    EXPAND(V, N);
    output Qmax
end {of MCQ }
```

```
procedure EXPAND(R, N)
begin
    while R ≠ ∅ do
        p := the vertex in R such that N[p] = Max{N[q] | q ∈ R};
            {i.e., the last vertex in R}
        if |Q| + N[p] > |Qmax| then
            Q := Q ∪ {p};
            Rp := R ∩ Γ(p);
            if Rp ≠ ∅ then
                NUMBER-SORT(Rp, N′);
                {the initial value of N′ has no significance}
                EXPAND(Rp, N′)
            else if |Q| > |Qmax| then Qmax := Q fi
            fi
            Q := Q − {p}
            else return
        fi
        R := R − {p}
    od
end {of EXPAND}
```

Figura 4. O algoritmo de Tomita e Seki

fonte: An Efficient Branch And Bound Algorithm For The Maximum Clique Problem

A função Number-Sort atribui cores ao vetor de vértices analisado e também ordena o grafo de acordo com as cores, de forma que o grafo fique ordenado da cor mínima à máxima. Essa função é baseada no algoritmo aproximativo de coloração de Fujii e Tomita, em que os C_k representam os grupos de cores. A função Expand é a função recursiva que gera novos cliques sempre seguindo a ordenação de cor, para cada nova chamada, os vizinhos do clique e do vértice a ser adicionado são ordenados por cor.

Nessa função, sempre que um novo elemento é adicionado ao clique Q, o R é atualizado e chamado de Rp. A intersecção feita para encontrar o Rp garante que todo novo elemento a ser adicionado no clique pertence também a vizinhança de um elemento anteriormente adicionado no clique.

A estratégia de pruning na função expand é baseada na cor do elemento p analisado, isso porque quando é escolhido um elemento de cor máxima, significa que esse elemento não pode ser encaixado em nenhum dos grupos anteriores, por possuir vizinhos naquele grupo. Assim, se um elemento tem cor máxima 4, por exemplo, é possível dizer que existem pelo menos outros 3 elementos que se conectam a ele. Como a função number-sort gera o conjunto de vetores analisados, o elemento de maior cor é consequentemente o último elemento do vetor.

Quando a função Expand encontra um Q com o tamanho superior aQ_{max}, Q_{max} é substituído pelo valor de Q, ao fim de cada chamada recursiva, é retirado o elemento da vez de Q, para que a próxima combinação seja analisada.

Os resultados obtidos pelo algoritmo do Tomita e Seki funcionam muito bem para diversos tipos de grafos e seu desempenho se mostrou consideravelmente melhor que os algoritmos do Ostegard e de outros trabalhos.

IMPLEMENTAÇÃO DA HEURÍSTICA

A implementação da heurística [4] foi realizada utilizando a linguagem de programação c++ na versão 14. A estrutura de dados utilizada para representar as relações de vizinhança dos vértices foi o HashMap e utilizou-se um vetor auxiliar para a ordenação dos vértices do grafo. Todo o algoritmo irá rodar no processo principal.

A ordenação foi feita em tempo linear, modificando um pouco a lógica do algoritmo “counting sort” de forma que a função ordenasse os vértices não por seus valores, mas sim por seus graus.

A função Expand foi levemente alterada, foi inserido um parâmetro limit, inserido pelo usuário na execução do teste. Isso foi feito para limitar a quantidade de vezes que o while do expand será percorrido e possibilitar a análise da relação entre quantos vértices iniciais precisam ser analisados até encontrar um clique máximo.

ANÁLISE DE COMPLEXIDADE

Começando pela função Number-Sort, teremos um custo de $|R|$ no while mais de fora juntamente com o custo t para encontrar a intersecção entre C_k e $\Gamma(p)$, $t = Max\{x \mid x \in \{|C_k|, |\Gamma(p)|\}\}$. E também j, que é o custo para percorrer a intersecção encontrada, com $t > j$. O custo da operação de remover o elemento p do conjunto R tem custo $|R|$ e esse valor diminui a cada execução do while. Após isso, temos um custo $maxno * |C_k|$. O custo da função number-sort é então $O((|R| * (t + j + |R|) + maxno * |C_k|$ que em pior caso é $O(|R|^2)$.

A função expand percorre o conjunto R, tendo custo $|R|$ e chama a função Number-Sort e a própria expand para um novo conjunto R_p, o custo disso será,

respectivamente, $|R_p|^2$ e $log(|R|)$em que $R_p \subset R$, logo $|R| \leq |R_p|$. Como, no pior caso, $|R_p| = |R| - 1$, o custo de expand no pior caso será $O(|R| * ((|R| - 1)^2 + log(|R|)))$.

Por fim, a função MCQ foi implementada dentro da função main, a função main lê o arquivo passado como parâmetro e cria a NeighboursMatrix, em tempo linear. Esse hashmap criado é o que estabelece a relação de vizinhança entre os nodes. Após isso, a função main irá ordenar os nodes por grau decrescente em tempo linear, como já falado, e as outras operações de MCQ também são lineares, ou seja $O(|V|)$. Assim, a complexidade do algoritmo se resume a complexidade de expand, e a complexidade total do algoritmo é dita como $O(|V| * ((|V| - 1)^2 + log(|V|)))$, ou seja, complexidade cúbica.

FASE DE TESTES

Para realizar os testes, foi utilizado o banco de grafos disponibilizado pelo DIMACS através da plataforma Network Data Repository. Selecionou-se grafos utilizados em testes de outros artigos sobre o problema do clique máximo que tivessem uma certa variância em densidade e tamanho. A máquina utilizada possui um processador de 4 núcleos Intel® Core™ i5-5200U CPU @ 2.20GHz, com 8 Gb de RAM.

Os testes da tabela 1 irão apresentar a variância da constante k e como isso afeta o resultado obtido. A constante k representa a quantidade de vezes que o while da função expand foi limitado, por exemplo, caso o k seja 10, o while de expand só executou 10 vezes para qualquer chamada recursiva do expand.

Os dados da tabela 1 tem a seguinte forma: N, E e P caracterizam respectivamente a quantidade de vértices do grafo, a quantidade de edges e a densidade do grafo. As colunas no formato W (k) e T(k) representam o tamanho do clique máximo encontrado dado aquela limitação e o tempo em segundos que foi levado para chegar naquele resultado. As últimas duas colunas trazem o resultado obtido quando não foi aplicado um limite, ou seja, quando k atinge o valor que precisa atingir em cada execução da função clique.

Os testes apresentados na tabela 2 trazem uma comparação entre os resultados obtidos na tabela 1, e também os resultados de dos algoritmos de Ostegard [3] e dos resultados publicados pelo Tomita e Seki [4]. A coluna W apresenta o tamanho do clique máximo encontrado pela heurística, e o tempo de execução levado está apresentado à direita, na coluna heurística. A coluna W1 apresenta o tamanho do clique máximo encontrado pela implementação dos outros dois algoritmos.

Quando o teste atinge o tamanho correto do clique máximo, ele é marcado em negrito na tabela 1. Os resultados que não encontraram o clique máximo são sublinhados.

Tabela 1. Testes realizados

Grafo	N	E	P	W 10	T 10	W 20	T 20	W 30	T 30	W N	T N
brock200-1	200	14800	0.75	16	0.2529	18	1.1388	20	4.2911	**21**	5085.2
brock200-2	200	9876	0.49	11	1.4396	11	4.6032	11	6.8629	11	13.267
c-fat200-1	200	1500	0.08	11	0.0089	**12**	0.01634	**12**	0.0178	**12**	0.0326
c-fat200-2	200	3200	0.16	**24**	0.0531	**24**	0.0674	**24**	0.0817	**24**	0.1578
hamming6-2	64	1824	0.90	**32**	0.2263	**32**	0.2829	**32**	0.3261	**32**	0.3267
hamming6-4	64	704	0.34	**4**	0.0089	**4**	0.0123	**4**	0.0134	**4**	0.0144
johnson8-2-4	28	420	0.56	**4**	0.0040	**4**	0.0056	**4**	0.0062	**4**	0.0065
johnson8-4-4	70	1900	0.77	**14**	0.2284	**14**	0.2933	**14**	0.3146	**14**	0.3258
mann-a9	45	918	0.92	**16**	0.8864	**16**	0.9156	**16**	0.9106	**16**	0.9018
mann-a27	378	70600	0.99	-	-	-	-	-	-	**126**	-
p-hat1000-1	1000	122300	0.24	**10**	10.546	**10**	35.666	**10**	79.586	**10**	1255.9
san200-0-7-1	200	13900	0.70	22	10.484	29	7.6163	29	11.938	29	18.454
san200-0-9-1	200	17900	0.90	-	-	-	-	-	-	**70**	-
san1000	1000	250500	0.50	-	-	-	-	-	-	**15**	-
sanr-200-0-7	200	13900	0.69	17	103.84	**18**	369.82	**18**	654.21	**18**	873.72

fonte: Autoria própria

Tabela 2. Contraposição de testes

Grafo	W	Heurística	W'	[4]	[3]
brock200-1	**21**	5085.27	**21**	2.84	54.29
brock200-2	**11**	13.267	**12**	0.016	0.05
c-fat200-1	**12**	0.03267	**12**	0.0002	0.01
c-fat200-2	**24**	0.1578	**24**	0.0005	0.01
hamming6-2	**32**	0.3241	**32**	0.0003	0.01
hamming6-4	**4**	0.0116	**4**	0.0001	0.01
johnson8-2-4	**4**	0.0055	**4**	0.000022	0.01
johnson8-4-4	**14**	0.3258	**14**	0.0009	0.01
mann-a9	**16**	0.8936	**16**	0.0002	0.01

mann-a27	**126**	-	**126**	8.49	>10,000.00
p-hat1000-1	**10**	1255.9	**10**	0.86	5.84
san200-0-7-1	**29**	18.454	**30**	0.0169	0.56
san200-0-9-1	**70**	-	**70**	2.30	0.27
san1000	**15**	-	**15**	10.06	0.51
sanr-200-0-7	**18**	873.72	**18**	0.92	14.11

fonte: Autoria própria

CONCLUSÕES

Dos testes apresentados na tabela 2, é possível observar que a implementação do Tomita e do Seki obteve resultados extremamente melhores que a implementação feita para esse relatório, e apesar desses resultados terem sido influenciados pela máquina utilizada, possivelmente há uma implementação mais ágil ao se utilizar outra estrutura de dados que não seja o hashmap juntamente de vectors. Deixando essa diferença de lado, alguns outros pontos interessantes podem ser levantados.

Primeiro, observando a tabela 1, é importante notar que existem alguns casos em que muitas execuções são feitas somente para garantir que o clique é de fato máximo. Isto é, no caso c-fat-200-1, hamming-6-4, hamming-6-2, johnson-8-2-4 e mann-a9, um clique máximo foi encontrado logo nas primeiras execuções do while do expand, tornando qualquer outra execução após isso “desnecessária”, pois um clique maior não será encontrado.

Acontece que essas execuções são necessárias para garantir que o clique máximo é de fato máximo. Caso fosse implementado uma estratégia de prunning, como um “tamanho de satisfação” para o clique, melhores resultados poderiam ser obtidos em todos os casos. O obstáculo é justamente assumir um tamanho de satisfação para esse problema.

Outra coisa interessante a ser observada é que obteve-se resultados controversos quanto a densidade nos testes da tabela 1, observe que para o brock200-1 de densidade 0.75 e para o sanr-200-0-7 de densidade 0.69, apesar de terem tamanhos e densidades bem parecidas, o brock200-1 levou quase 6 vezes mais tempo para finalizar o algoritmo. Isso apoia o que é descrito nos testes do trabalho [4], que diz que o algoritmo de Tomita e Seki é levemente influenciado por densidade.

A implementação teve dificuldade quando o número de edges eram muito grandes, como nos casos san200-0-9-1, San1000 e mann-a27 onde não conseguiu finalizar. E mesmo em casos em que foi possível finalizar, como o p-hat1000-1, ele leva um tempo muito mais elevado.

Apesar da implementação ser fiel à implementação de Tomita e Seki, os resultados da implementação para este trabalho foram bem diferentes. O algoritmo não conseguiu

finalizar em um limite de 3 horas para alguns casos e mesmo quando finalizou, os resultados piores que os do trabalho [4]. Assim, conclui-se que é possível melhorar o algoritmo fazendo escolhas diferentes de estruturas para obter melhores resultados, como os dos autores do algoritmo.

REFERÊNCIAS

[1] LUCE, R Duncan; PERRY, Albert D. A method of matrix analysis of group structure. **Psychometrika.** v14. p. 95-116. mar. 1949.

[2] CARRAGHAN, Randy; PARDALOS, Panos. An exact algorithm for the maximum clique problem. **Operations Research Letters**. v9. pp. 375-382. 1990.

[3] ÖSTERGÅRD, Patric. A fast algorithm for the maximum clique problem.
Discrete Applied Mathematics. v120, 2002.

[4] TOMITA, E; SEKI, T. An efficient branch-and-bound algorithm for finding a maximum clique. **Lecture Notes in Computer Science.** p. 278-289.