Iterated Local Search no Problema da Clique Máxima

Gabriel Cardoso de Carvalho

Resumo: Esse artigo mostra os resultados de uma implementação do Iterated Local Search no problema da Clique Máxima baseada em decisões aleatórias com o objetivo de comparar com os resultados de outras implementações disponíveis na literatura para as instâncias do DIMACS. Os resultados são, como esperado, piores do que em soluções gulosas, porém não são tão ruins quanto espera-se. Aqui comparamos o nosso algoritmo com o IKLS [1], o HSSGA [9] e o VNS para cliques máximas [8]. De fato, o algoritmo erra somente nas instâncias mais complexas, em que outras metaheurísticas como o HSSGA e o VNS também erram, e nos casos em que somente ele erra, ele o faz por uma margem bem pequena.

1 Introdução

O problema de encontrar a Clique Máxima (CM) é extremamente conhecido e estudado, pois inúmeros problemas práticos de diversas áreas diferentes, como biologia computacional, economia e análise de redes sociais podem ser modelados como CM. Além disso, a sua versão de decisão foi um dos primeiros problemas a serem provados NP-Completos.

Ele pode ser definido da seguinte maneira, seja o grafo $G = (V, E)$ onde $V = 1, 2, ..., n$ é o conjunto de vértices e $E \subseteq V \times V$ é o conjunto de arestas, uma Clique $C \subseteq V$é tal que $\forall i, j \in C, (i, j) \in E$, ou seja, todos os vértices em C são adjacentes entre si. Ou ainda, C é um subgrafo completo de G. O problema da clique máxima é o problema de encontrar a clique de cardinalidade máxima do grafo G.

Diversas soluções foram propostas, tanto métodos exatos quanto heurísticas e metaheurísticas [2, 3, 4]. As propostas no geral tendem a utilizar o sistema *breach and bound* nos métodos exatos e heurísticas gulosas em buscas locais, preferindo vértices de maior grau. De maneira geral, os algoritmos gulosos partem de uma clique C inicial que contém apenas um vértice e um conjunto N_C de vértices $v \in V$ que são os vértices vizinhos à C, ou seja, $\forall u \in C, v$ é adjacente à u. Daí o algoritmo adiciona vértices de N_C em C, escolhendo sempre o vértice de N_C que tem o maior grau no subgrafo $G(N_C)$, até que C seja *maximal*, ou seja, até que não exista uma clique C' maior que C tal que $C \subseteq C'$. Em outros trabalhos esse método é chamado de *Busca Local 1-opt* [5].

A metaheurística implementada nesse artigo é a *Iterated Local Search (ILS)*, que pode ser resumida como uma metaheurística que cria, de maneira iterativa, uma sequência de soluções geradas por uma heurística interna (ou busca local) [6]. É esperado que as soluções providas pelo ILS sejam melhores do que uma simples repetição da heurística de maneira aleatória.

Nesse artigo é proposta uma implementação do ILS focada na aleatoriedade, de modo a comparar seu desempenho com métodos gulosos, como a implementação do IKLS de *Katayama* [1], que utiliza a *Busca Local k-opt (KLS)* [5] como busca local, que é uma generalização da busca local 1-opt, onde adiciona-se k vértices à clique por vez, permitindo retirar vértices da clique para isso, de maneira dinâmica, ou seja, o k não é fixo, e uma perturbação baseada na *menor conecividade por arestas (LEC-KLS)*, onde escolhe-se um vértice v que não pertence à C, de maneira que v seja adjacente à menor quantidade de vértices de C. Então, adiciona-se v à C e remove de C todos os vértices que não são vizinhos à v.

A seção 2 apresenta como foi feita a implementação do ILS, enquanto a seção 3 cobre toda a implementação e os resultados experimentais do método sobre as instâncias do DIMACS [4]. A seção 4 apresenta as conclusões e os trabalhos futuros.

2 ILS

Esta implementação do ILS segue o padrão de Glover [6] utilizando o algoritmo 1, onde *GeraSolucaoInicial* trata-se da função que retorna uma clique maximal aleatória do grafo G, a *BuscaLocal* parte de uma clique maximal e busca nas vizinhanças uma clique maior, *Perturbacao* recebe uma clique maximal e retira uma quantidade aleatória de vértices dessa clique, em alguns casos retirando todos os vértices, caso em que ocorre uma reinicialização, e *CriterioAceitacao* que escolhe se a próxima perturbação será feita na solução antiga ou na atual.

Data: Grafo G, inteiro n_{iter}
s_0 = GeraSolucaoinicial(G);
$s*$ = BuscaLocal(s_0);
$k = 0$;
while k *for menor que* n_{iter} **do**
  s' = Perturbacao($s*$);
  $s*'$ = BuscaLocal(s');
  $s*$ = CriterioAceitacao($s*, s*'$);
  k_{++};
end

Algoritmo 1: Estrutura do ILS

Os detalhes de cada função são descritos nas subseções seguintes.

2.1 Geração da Solução Inicial

A função *GeraSolucaoInicial(G)* somente escolhe um vértice aleatório v de G e chama a função *geraSolucao(G, v)*.

A função *geraSolucao(G, v)* gera a solução inicial ao criar uma clique C de tamanho 1 utlizando o vértice v. Essa clique é um estrutura que contém os vértices que a compõe (denotados por C), o seu tamanho k, e um conjunto N_C de vértices que podem ser adicionados à ela. A partir daí, adiciona-se vértices aleatórios de N_C à C e atualiza seu tamanho e N_C até que a clique C seja maximal, como mostra o algoritmo 2.

Data: Grafo G, vértice v
s_0 = clique contendo v;
$N_{s_0} = V_v$;
$k = 1$;
while $N_C \neq \emptyset$ **do**
  u = vértice aleatório de N_C;
  $s_0 = s_0 + u$;
  $N_{s_0} = N_{s_0} \cap V_u$;
  k_{++};
end
return *clique* s_0 *maximal*

Algoritmo 2: função geraSolucao

No algoritmo 2, N_{s_0} representa os vértices que podem ser adicionados na clique s_0, enquanto V_v representa os vértices adjacentes ao vértice v. O algoritmo para quando não há mais nenhum vértice que possa ser adicionado a s_0, ou seja, s_0 é maximal.

Além de ser utilizada para criar a solução inicial, a função *GeraSolucaoInicial(G)* é chamada novamente sempre que a Perturbação decide que deve ser feita uma reinicialização.

2.2 Busca Local

A busca local vai receber uma clique C, e a partir dessa clique, vai tentar melhorá-la olhando para as vizinhanças de C. Temos duas vizinhanças, N_1 e N_2 que são explicadas mais a frente. Elas são conjuntos de cliques vizinhas a C, dada alguma propriedade. Dados esses conjuntos, a busca local irá simplesmente maximizar cada uma das cliques e escolher uma delas.

Definimos aqui $N_i(C)$ como o conjunto de cliques vizinhas a C, ou seja, a vizinhança da clique C, tal que, $\forall C' \in N_i(C)$,
$C' \subset C, k(C') = k(C) - i$. Portanto as vizinhanças que usaremos, N_1 e N_2, são subcliques de C com tamanhos $k-1$ e $k-2$, respectivamente, onde k é o tamanho da clique C.

N_1 tem tamanho k, ou seja, temos um vizinho para cada vértice de C. Dessa forma, é possível utilizar a técnica de *Best Improvement* (testar todos e

escolher o melhor) para N_1. Para N_2, por outro lado temos $\binom{k}{2}$ vizinhos, ou ainda, $\frac{k^2-k}{2}$. Caso N_1 não produza uma melhora, então testa-se para N_2, onde se faz necessário utilizar *First Improvement* (escolher o primeiro que melhorar a solução atual) devido ao seu tamanho. Por outro lado, mesmo usando First Improvement o tamanho de N_2 pode ficar enorme. Por isso, o algoritmo 3 mostra o pseudo-código da busca local com a maneira que utilizamos para buscar em N_2.

```
Data: Clique s
N_1 = conjunto de subcliques de s de tamanho k(s) − 1;
s' = s;
for c ∈ N_1 do
    c = maximiza(c);
    if k(c) > k(s') then
        s' = c;
end
if k(s') > k(s) then
    return clique s';
N_2 = conjunto de subcliques de s de tamanho k(s) − 2;
for c ∈ N_2 do
    c = maximiza(c);
    if k(c) > k(s') then
        s' = c;
end
return clique s
```

Algoritmo 3: BuscaLocal

Buscar todo o N_2 implica em complexidades de tempo e espaço grandes demais em grafos muito grandes, como as instâncias *C4000.5*, *MANN_a81* e *keller6* do DIMACS [4]. Dessa forma, como o tamanho de N_2 é muito grande, calcula-se apenas k dos $\frac{k^2-k}{2}$ elementos de N_2 de maneira aleatória.

As cliques, tanto em N_1 quanto N_2, são proibidas de voltarem às suas cliques originais, ou seja, seja $c \in N_1$ se $c = s + u$, onde s é a clique original de c e u é o vértice de s que não está em c, então u não faz parte de N_c.

A busca local portanto sempre vai retornar uma clique maior que a entrada dela, ou a própria entrada caso não consiga melhorá-la.

2.3 Perturbação

A perturbação simplesmente sorteia um valor $2 \leq n \leq k$ e remove de $s*$ n vértices aleatórios. Caso $n = k$, então é feita a reinicialização do procedimento, retornando a chamada da função GeraSolucaoInicial(G). Caso $n < k$ então simplesmente s' recebe $s*$ e depois são removidos os n vértices de s' . Depois de removidos os n vértices, atualiza-se N'_s e maximiza a solução de maneira aleatória. Algoritmo 4 ilustra esse processo.

Data: grafo G, clique $s*$
n = sorteie um número $\in \{2, k(s*)\}$;
if $n = k$ **then**
  return *GeraSolucaoInicial(G)*
$s' = s*$;
while $k(s') > k(s*) - n$ **do**
  v = vértice aleatório $\in s'$ $s' = s' - v$;
  $k(s')_{--}$;
end
recalcula $N_{s'}$;
return s'

Algoritmo 4: Perturbação

A perturbação é feita dessa forma pois ela deve fugir do ótimo local para que a busca local possa buscar nas proximidades da solução atual uma solução melhor. Dessa forma, a perturbação não pode ser muito forte ou muito fraca. No caso deste artigo, a força da perturbação é decidida de maneira aleatória, mas somente ocorre uma reincialização com probabilidade $p = \frac{1}{k-1}$, assim como ele aplicará a perturbação mais leve com a mesma probabilidade. Na média, a perturbação terá força entre os dois extremos.

2.4 Critério de Aceitação

O critério de aceitação tem o dever de escolher qual solução sofrerá a próxima perturbação. É bem simples e tem três opções: retorna o maior entre $s*$ e $s*'$, sempre retorna $s*'$, ou , sorteia um dos dois para ser o retorno.

O primeiro caso é o caso em que sempre retorna-se o melhor dos dois, que seria o guloso, enquanto o segundo é uma maneira também chamada de *random walk*, pois simplesmente caminha para a próxima solução sem se preocupar com a qualidade dela. O terceiro caso seria o meio termo entre os dois mas que ainda corre o risco de escolher uma solução pior sem nenhum ganho no resto do procedimento.

Nos experimentos apenas o primeiro caso é utilizado pois é certo que nenhum deles apresenta ganho de velocidade no procedimento geral em comparação com o outro, portanto é uma questão apenas de qual tem mais chance de dar bons resultados, que é, de maneira geral, o guloso.

3 Resultados Experimentais

Toda a implementação foi feita na linguagem JAVA e pode ser acessado por git [7]. Diversos testes foram feitos nas 37 instâncias do DIMACS, com foco nos mais simples, pois é possível testar mais vezes já que essas instâncias são menores.

Instances	DIMACS	Maior	Média(M)	Tempo	Média(T)
C125.9	34	34	33.866665	22.069	1.4712666
C250.9	44	44	42.2	79.872	5.3248
C500.9	57	53	51.066666	324.396	21.6264
C1000.9	68	62	59.133335	1413.653	94.24353
C2000.9	80	70	66.46667	6268.48	417.89865
DSJC1000_5	15	14	13.133333	111.005	7.400333
DSJC500_5	13	13	12.133333	34.25	2.2833333
C2000.5	16	15	14.266666	374.985	24.998999
C4000.5	18	16	15.4	4310.259	287.3506
MANN_a27	126	125	123.933334	1545.286	103.019066
MANN_a45	345	336	334.888888	48895.62	5432.82
MANN_a81	1100	1084	-	-	-
brock200_2	12	12	10.666667	5.219	0.34793332
brock200_4	17	16	15.266666	8.829	0.58860004
brock400_2	29	24	22.6	59.105	3.9403334
brock400_4	33	25	22.666666	61.049	4.0699334
brock800_2	24	20	18.933332	135.98	9.065333
brock800_4	26	19	18.466667	133.014	8.8676
gen200_p0.9_44	44	44	39.266666	36.969	2.4646
gen200_p0.9_55	55	55	47.0	51.641	3.4427333
gen400_p0.9_55	55	50	47.6	157.595	10.506333
gen400_p0.9_65	65	65	50.333332	156.574	10.438267
gen400_p0.9_75	75	75	59.733334	213.645	14.243
hamming10-4	40	40	37.133335	417.751	27.850067
hamming8-4	16	16	16.0	12.258	0.8172
keller4	11	11	11.0	4.891	0.32606664
keller5	27	27	24.4	228.479	15.231934
keller6	59	52	49.4	8234.637	548.97577
p_hat300-1	8	8	8.0	3.14	0.20933335
p_hat300-2	25	25	25.0	28.312	1.8874667
p_hat300-3	36	36	34.733334	58.169	3.8779333
p_hat700-1	11	11	9.333333	19.366	1.2910666
p_hat700-2	44	44	43.533333	274.399	18.293266
p_hat700-3	62	62	61.266666	498.792	33.2528
p_hat1500-1	12	12	10.4	94.455	6.297
p_hat1500-2	65	65	63.666668	1571.55	104.770004
p_hat1500-3	94	93	91.2	4536.831	302.4554

Tabela 1: Tabela com os resultados do procedimento nas 37 instâncias do DIMACS

A implementação tem somente um parâmetro, $nIter$ que foi setado como $nIter = 150$, que é o número de iterações do ILS, pois abaixo disso testes mostraram que as cliques encontradas eram muito menores que o valor esperado, enquanto maior que 150 não demonstrou melhoria significativa em comparação. Cada um dos testes foram repetidos 15 vezes para cada instância, com exceção de MANN_a45 (que foi rodado apenas 9 vezes), e MANN_81 que rodou apenas GeraSolucaoInicial.

Essas instâncias não foram completas pois o tamanho das cliques era muito grande, > 300 para MANN_a45 e > 1000 para MANN_a81, e por isso, N_1 e N_2 ficaram muito grandes. O gargalo fica no recálculo de N_c para cada $c \in N_1$ e $c' \in N_2$, que precisa checar todos os vizinhos de todos os vértices em c e c' para todas as cliques em N_1 e N_2. A tabela 1 mostra os resultados de todos os experimentos.

Na tabela 1, *Instância* representa a instância em que o experimento foi feito, *DIMACS* representa o maior valor que foi encontrado pelo DIMACS [4] para a dada instância, *Maior* representa a maior clique encotrada pelo nosso algoritmo, *Média(M)* representa o tamanho médio das cliques encontradas em cada uma das 15 tentativas, *Tempo* representa o tempo que levou para todo o experimento ser completo e *Média(T)* representa o tempo médio que cada uma das 15 tentativas levou para completar as 150 iterações.

Quando comparados com os melhores resultados do DIMACS, vemos que o ILS proposto se iguala ao melhor encontrado em 20 ocasiões das 37. Quando erra, por outro lado, tem erro médio de 10% comparado ao valor do DIMACS, sendo o maior erro de 27% na instância brock800_4 e o menor é de 1% nas instâncias MANN_a27, MANN_a81 (lembrando que nesta instância não houve sequer a buscal local) e p_hat1500-3. Entre os erros é válido colocar que nas instâncias DSJC1000_5, C2000.5, MANN_a27, brock200_4 e p_hat1500-3, a diferença de tamanho entre a maior clique conhecida pelo DIMACS e a nossa foi de apenas um vértice.

A tabela 2 mostra a comparação do nosso algoritmo (na tabela está como ILS) com alguns outros algoritmos da literatura, o IKLS [1], VNS para clique máxima [8] e o HSSGA [9].

É possível notar que nosso algoritmo se aproxima do HSSGA, sendo que sempre que o HSSGA erra o nosso também erra. Isso implica na possibilidade de que os grafos que são difíceis para o HSSGA são também difíceis para o nosso, enquanto o VNS erra somente no brock800_2 e brock800_4 que são duas das instâncias que o nosso algoritmo errou com maior diferença.

4 Conclusão

Neste artigo foi apresentada uma implementação do Iterated Local Search no problema da clique máxima. Essa implementação focou na tomada de decisões aleatórias em todas as fases do algoritmo do ILS, utilizando duas vizinhanças

Instances	DIMACS	ILS	IKLS[1]	VNS[8]	HSSGA[9]
C125.9	34	34	34	34	34
C250.9	44	44	44	44	44
C500.9	57	53	57	57	56
C1000.9	68	62	68	68	66
C2000.9	80	70	78	78	74
DSJC1000_5	15	14	15	15	15
DSJC500_5	13	13	13	13	13
C2000.5	16	15	16	16	16
C4000.5	18	16	18	18	17
MANN_a27	126	125	126	126	126
MANN_a45	345	336	345	345	343
MANN_a81	1100	1084	1100	1100	1095
brock200_2	12	12	12	12	12
brock200_4	17	16	17	17	17
brock400_2	29	24	29	29	29
brock400_4	33	25	33	33	33
brock800_2	24	20	24	21	21
brock800_4	26	19	26	21	21
gen200_p0.9_44	44	44	44	44	44
gen200_p0.9_55	55	55	55	55	55
gen400_p0.9_55	55	50	55	55	53
gen400_p0.9_65	65	65	65	65	65
gen400_p0.9_75	75	75	75	75	75
hamming10-4	40	40	40	40	40
hamming8-4	16	16	16	16	16
keller4	11	11	11	11	11
keller5	27	27	27	27	27
keller6	59	52	59	59	57
p_hat300-1	8	8	8	8	8
p_hat300-2	25	25	25	25	25
p_hat300-3	36	36	36	36	36
p_hat700-1	11	11	11	11	11
p_hat700-2	44	44	44	44	44
p_hat700-3	62	62	62	62	62
p_hat1500-1	12	12	12	12	12
p_hat1500-2	65	65	65	65	65
p_hat1500-3	94	93	94	94	94

Tabela 2: Tabela com os resultados do procedimento nas 37 instâncias do DIMACS

diferentes para a busca local, N_1 e N_2, que consistem nas subcliques da clique que passará pela busca local de tamanho $k-1$ e $k-2$ respectivamente, onde k é o tamanho da clique. A perturbação simplesmente retira uma quantidade aleatória de vértices da clique, e eventualmente reinicializa o processo.

Os resultados, como esperado, mostram que aleatoriedade é pior do que utilizar uma heurística gulosa para a busca local. A implementação desse artigo encontra uma clique em média 10% pior que a média do IKLS [1] e a maior discrepância encontrada é de 27% pior que o IKLS. Porém, este algoritmo errou, em geral, apenas nas instâncias mais difíceis.

Trabalhos futuros envolvem encontrar uma maneira mais eficiente de se calcular o $N_{\{C-v\}}$ de uma clique C que foi retirado o nó v, pois a forma atual faz com que o algoritmo seja muito lento. Além disso, testar diversas diferentes buscas locais e perturbações, de maneira a encontrar a melhor possível forma do ILS para o problema da clique máxima. Finalmente, uma comparação da melhor forma encontrada com outros tipos de mataheurísticas, como algoritmos genéticos, GRASP, evolutivos entre outros.

Referências

[1] Katayama, Kengo, Masashi Sadamatsu, and Hiroyuki Narihisa. "Iterated k-opt local search for the maximum clique problem." *Lecture Notes in Computer Science* 4446 (2007): 84.

[2] Wu, Qinghua, and Jin-Kao Hao. "A review on algorithms for maximum clique problems." *European Journal of Operational Research* 242.3 (2015): 693-709.

[3] I.M. Bomze, M. Budinich, P.M. Pardalos, and M. Pelillo. The maximum clique problem. In D.-Z. Du and P.M. Pardalos, editors, *Handbook of Combinatorial Optimization (suppl. Vol. A)*, pp. 1–74. Kluwer, 1999.

[4] D.S. Johnson and M.A. Trick. *Cliques, Coloring, and Satisfiability*. Second DIMACS Implementation Challenge, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1996.

[5] K. Katayama, A. Hamamoto, and H. Narihisa. An effective local search for the maximum clique problem. *Information Processing Letters*, Vol. 95, No. 5, pp. 503–511, 2005.

[6] Glover, Fred W., and Gary A. Kochenberger, eds. *Handbook of metaheuristics*. Vol. 57. Springer Science & Business Media, 2006

[7] https://github.com/Clique33/TrabalhoIntComp.git Acesso em 04/07/2017.

[8] P. Hansen, N. Mladenovic, and D. Urosevic. *Variable neighborhood search for the maximum clique*. Discrete Applied Mathematics, Vol. 145, No. 1, pp. 117–125, 2004

[9] A. Singh and A. K. Gupta. A hybrid heuristic for the maximum clique problem. *Journal of Heuristics*, Vol. 12, No. 1-2, pp. 5–22, 2006.