
Comunidades em Grafos

João C. Abreu¹

¹*Universidade Federal de Minas Gerais, DCC,
Avenida Antônio Carlos 6627, Belo Horizonte, Brazil,
joao.junior@dcc.ufmg.br*

Abstract Esse trabalho apresenta o problema de calcular comunidades e cinco algoritmos exatos para a resolução desse problema.

Keywords: Comunidades em Grafo, Centralidade, betweenness

1. Introdução

Seja um grafo $G = (V, E)$, onde V é o conjunto de vértices e E é o conjunto de arestas, com a cardinalidade $|V| = n$ e $|E| = m$. O peso de cada aresta $(i, j) \in E$ é dado pela função $w(i, j) = 1, \forall (i, j) \in E$. A centralidade (do inglês betweenness) de uma aresta $(i, j) \in E$ é definida como a quantidade de caminhos mínimos que utilizam a aresta $(i, j) \in E$. O problema de encontrar k comunidades em um grafo G pode ser visto como um problema de se retirar as arestas com maior centralidade no grafo até possuir k componentes conexas nesse grafo. O Objetivo desse trabalho é comparar cinco algoritmos exatos para o problema de cálculos de comunidade. A seção 2 apresenta as estruturas de dados utilizadas nesse trabalho, a seção 3 apresenta os cinco algoritmos exatos para o problema de cálculos de comunidades, a seção 4 apresenta os experimentos computacionais desse trabalho.

Para se encontrar k comunidades em um grafo, com $k < m$, calcula-se a centralidade de cada aresta $(i, j) \in E$ e retira-se do grafo a aresta $(i, j) \in E$ que possuir a maior centralidade. Após isso a quantidade de componentes conexas do grafo é verificada e se for igual a k o problema está resolvida, caso contrário o cálculo de centralidades deve ser novamente efetuado e a aresta $(i, j) \in E$ com maior centralidade é removida do grafo.

Todo código fonte produzido por esse trabalho pode ser obtido no endereço eletrônico: [https :
//github.com/joaojunior/feixo1_2scp](https://github.com/joaojunior/feixo1_2scp)

2. Estrutura de Dados

Essa seção apresenta as estruturas de dados que serão utilizadas nos algoritmos apresentados na seção 3. A seção 2.1 mostra a estrutura de dados que representa um grafo $G = (V, E)$ e a seção 2.2 apresenta a estrutura de dados fila de prioridades mínimas.

2.1 Grafo

Um grafo $G = (V, E)$ vai ser representado por uma matriz $n \times n$, onde $n = |V|$. Quando existe uma aresta $(i, j) \in E$ o valor armazenado na posição i, j da matriz será 1 e quando a posição i, j da matriz não corresponder a uma aresta será armazenado o valor do maior inteiro possível. Assim a ordem de complexidade da quantidade de memória utilizada para representar um grafo vai ser igual a $\theta(n^2)$ e a ordem de complexidade para verificar se uma aresta existe no grafo será $O(1)$. A estrutura de dados grafo está definida no arquivo `graph.h` e sua implementação pode ser obtida no arquivo `graph.c`.

2.2 Fila de prioridade Mínima

A fila de prioridade mínima [1] vai ser representada por um vetor de n elementos. Cada elemento armazena dois inteiros, um representa o peso do elemento e o outro a posição na fila de prioridade mínima. Um elemento pode possuir no máximo dois filhos, onde o peso do elemento dos filhos é igual ou menor do que o peso do elemento pai. Assim a ordem de complexidade da quantidade de memória para armazenar uma fila de prioridade mínima é $\theta(n)$, a ordem de complexidade para se retirar o elemento de menor peso da fila e para se diminuir o valor de um peso de um elemento existente na fila é $O(\log_2 n)$. A estrutura de dados fila de prioridade mínima está definida no arquivo `min_priority_queue.h` e sua implementação pode ser obtida no arquivo `min_priority_queue.c`.

3. Algoritmos

Essa seção apresenta um algoritmo genérico para o cálculo de comunidades e analisa a ordem de complexidade de cinco algoritmos baseados nesse algoritmo genérico. O Algoritmo 1 apresenta um algoritmo genérico para o cálculo de comunidades em um grafo $G = (V, E)$. Na linha 2 é calculado a centralidade de todas as arestas, o laço das linhas 3 a 6 é repetido até que o número k de comunidades é obtido. A linha 4 retira do grafo $G = (V, E)$ a aresta $(i, j) \in E$ que possui

a maior centralidade. A linha 5 recalcula os caminhos mínimos entre todos os pares de vértices $i, j \in V$. As seções 3.1, 3.2, 3.3, 3.4 e 3.5 analisam a ordem de complexidade dos algoritmos que se diferem do algoritmo genérico apenas pela cálculo de centralidades. As seções 3.1, 3.2, 3.3, 3.4 e 3.5 apresentam as análises dos algoritmos que calculam a centralidade utilizando, respectivamente, os algoritmos: Faster-All-Pairs-Shortest-Path[4], Floyd-Warshall[5], Johnson com fila de prioridades mínimas[6], Johnson com array[6] e Breadth First Search[2].

Entrada: Grafo $G = (V, E)$ e número k de comunidades
 Saída: Identificador da comunidade e label do nó em cada comunidade

```

1 inicio
2   Calcular centralidades de todas as arestas  $(i, j) \in E$ 
3   while o número de comunidades  $< k$  do
4     Remover a aresta que possuir a maior centralidade
5     Recalcular a centralidade de todas as arestas
6   end
7 fin
```

Algoritmo 1: Algoritmo genérico para o cálculo de comunidades em um grafo

3.1 Faster-All-Pairs-Shortest-Path

O cálculo de centralidades é feito utilizando o algoritmo Faster-All-Pairs-Shortest-Path. O Algoritmo Faster-All-Pairs-Shortest-Path possui ordem de complexidade igual a $\theta(n^3 \log_2 n)$. No melhor caso apenas k arestas serão retiradas do grafo e no pior caso m arestas serão retiradas do grafo. Assim o algoritmo de cálculo de comunidades utilizando o algoritmo Faster-All-Pairs-Shortest-Path terá ordem de complexidade igual a $\theta(kn^3 \log_2 n)$ no melhor caso e $\theta(mn^3 \log_2 n)$ no pior caso. A quantidade de memória utilizada por esse algoritmo será da ordem de $\theta(n^2)$, que é a ordem de utilização de memória dado pela estrutura de dados grafo.

3.2 Floyd-Warshall

O cálculo de centralidades é feito utilizando o algoritmo de Floydwarshall. O Algoritmo Floydwarshall possui ordem de complexidade igual a $\theta(n^3)$. No melhor caso apenas k arestas serão retiradas do grafo e no pior caso m arestas serão retiradas do grafo. Assim o algoritmo de cálculo de comunidades utilizando o algoritmo Floydwarshall terá ordem de complexidade igual a $\theta(kn^3)$ no melhor caso e $\theta(mn^3)$ no pior caso. A quantidade de memória utilizada por esse algoritmo será da ordem de $\theta(n^2)$, que é a ordem de utilização de memória dado pela estrutura de dados grafo.

3.3 Johnson com Fila de Prioridades Mínimas

O cálculo de centralidades é feito utilizando o algoritmo de Johnson que utiliza o algoritmo de Dijkstra[3] com fila de prioridade mínima. Nesse caso o Algoritmo de Johnson que utiliza o algoritmo de Dijkstra com fila de prioridades mínima possui ordem de complexidade igual a $O(nm \log_2 n)$. No melhor caso apenas k arestas serão retiradas do grafo e no pior caso m arestas serão retiradas do grafo. Assim o algoritmo de cálculo de comunidades utilizando o algoritmo de Johnson terá ordem de complexidade igual a $O(knm \log_2 n)$ no melhor caso e $O(nm^2 \log_2 n)$ no pior caso. A quantidade de memória utilizada por esse algoritmo será da ordem de $\theta(n^2)$, que é a ordem de utilização de memória dado pela estrutura de dados grafo.

3.4 Johnson com Array

O cálculo de centralidades é feito utilizando o algoritmo de Johnson que utiliza o algoritmo de Dijkstra[3] com um array. Nesse caso o Algoritmo de Johnson possui ordem de complexidade igual a $(n^3 + nm)$. No melhor caso apenas k arestas serão retiradas do grafo e no pior caso m arestas serão retiradas do grafo. Assim o algoritmo de cálculo de comunidades utilizando o algoritmo de Johnson terá ordem de complexidade igual a $O(k(n^3 + nm))$ no melhor caso e $O(mn^3 + nm^2)$ no pior caso. A quantidade de memória utilizada por esse algoritmo será da ordem de $\theta(n^2)$, que é a ordem de utilização de memória dado pela estrutura de dados grafo.

3.5 Breadth First Search

O cálculo de centralidades é feito utilizando o algoritmo Breadth First Search (BFS). O Algoritmo BFS possui ordem de complexidade igual a $O(n + m)$. No cálculo de centralidade utilizando o algoritmo BFS para cada vértice $v \in V$ é executado o algoritmo BFS. No melhor caso apenas k arestas serão retiradas do grafo e no pior caso m arestas serão retiradas do grafo. Assim o algoritmo de cálculo de comunidades utilizando o algoritmo BFS terá ordem de complexidade igual a $O(k(n^2 + nm))$ no melhor caso e $O(mn^2 + nm^2)$ no pior caso. A quantidade de memória utilizada por esse algoritmo será da ordem de $\theta(n^2)$, que é a ordem de utilização de memória dado pela estrutura de dados grafo.

4. Experimentos Computacionais

Os experimentos computacionais foram executados em uma máquina Intel Dual-Core de 2.81 GHz de clock e 2GB de memória RAM, rodando o sistema operacional Linux. Os algoritmos das seções 3.1, 3.2, 3.3, 3.4 e 3.5 foram implementados na linguagem de programação c e compilados com o compilador gcc na versão 4.8.2. A tabela 1 apresenta esses cinco algoritmos. Na coluna 1 está o nome que o algoritmo vai assumir aqui, a coluna 2 é a referência da seção que o algoritmo foi apresentado, a coluna 3 mostra o nome do arquivo de especificações do algoritmo e a coluna 4 apresenta o nome do arquivo com o código fonte do algoritmo.

Algoritmo	Seção	Arquivo .h	Arquivo .c
kc_faster	3.1	repeated_squaring.h	repeated_squaring.c
kc_floydwarshall	3.2	floyd_warshall.h	floyd_warshall.c
kc_johnson_min_queue	3.3	johnson.h	johnson.c
kc_johnson_array	3.4	johnson.h	johnson.c
kc_nbfs	3.5	bfs.h	bfs.c

Table 1: Algoritmos exatos para o problema de encontrar k comunidades em um grafo não direcionado e não ponderado

O modelo matemático apresentado em ?? foi implementado no Ilog CPLEX 12.5.1 e o algoritmo da seção ?? foi implementado em Python 2.7, sendo que o otimizador utilizado para resolver os modelos lineares presente nesse algoritmo foi o Ilog CPLEX 12.5.1. Foram utilizados quatro conjuntos de instâncias de testes nos experimentos computacionais e essas instâncias foram retiradas de [4]. Nessas instâncias de testes cada linha da matriz de incidência A é coberta por pelo menos duas colunas e cada coluna cobre pelo menos uma linha. O custo c_j de cada coluna j está entre $[1, 100]$. A tabela 2 resume esses conjuntos de instâncias. A coluna 1 dessa tabela representa o identificador do conjunto da instância de teste, as colunas 2 e 3 mostram, respectivamente, o número m de linhas e n de colunas da matriz de incidência A . A coluna 4 representa a densidade da matriz A que é calculado pela quantidade de 1's dessa matriz dividido pela quantidade total de elementos de A que é igual a mn e a coluna 5 mostra a quantidade de problemas em cada conjunto. Os dez problemas do conjunto 4 são nomeados como scp41-scp410, os cinco problemas do conjunto 6 são nomeados como scp61-scp65, e os problemas do conjunto A e B são nomeados respectivamente como scp1-scp5 e scpb1-scpb5.

No experimento desse trabalho foi comparado a performance do modelo proposto na seção ??, que será chamado aqui de IP , com o algoritmo proposto na seção ??, que será chamado $ARank1$. O modelo IP foi executado através do CPLEX com todos os parâmetros default. O

Conjunto	Linhas	Colunas	Densidade	Problemas
4	200	1000	2	10
6	200	1000	5	5
A	300	3000	2	5
B	300	3000	5	5

Table 2: Detalhes das instâncias de testes utilizadas

modelo presente no algoritmo *ARank1* foi executado pelo CPLEX com um tempo máximo de 120 segundos e foi setado para que o CPLEX encontra-se no máximo cinco soluções inteiras, explorando no máximo 50000 nós na árvore de Branch-and-Bound, encontra-se soluções com um valor objetivo sempre inferior a -0.05 e a ênfase na busca de soluções foi setado 4. O modelo *IP* e o algoritmo *ARank1* foram executados com um tempo de execução máximo de 7200 segundos.

A tabela 3 apresenta os resultados obtidos para o conjuntos de instância 4 e 6 e a tabela 4 apresenta os resultados para o conjuntos de instância A e B. Nessas tabelas a coluna 1 mostra o nome da instância de teste, as colunas 2 e 3 são resultados referentes ao modelo *IP* e as colunas 4,5,6 e 7 são resultados referentes ao algoritmo *ARank1*. A coluna 2 apresenta o custo da solução obtido pela modelo *IP* e a coluna 3 apresenta o tempo consumido para encontrar essa solução. A coluna 4 mostra o valor da relaxação linear obtida para o *SCP* no início do algoritmo *ARank1*, a coluna 5 apresenta o custo da solução obtido após procurar e inserir os cortes de Chvátal-Gomory de rank 1, a coluna 6 traz a quantidade de cortes que o algoritmo *ARank1* conseguiu adicionar e a coluna 7 mostra o tempo consumido pelo algoritmo *ARank1*. Para todas as instâncias do conjunto 4 e 6 o CPLEX conseguiu encontrar soluções ótimas em um tempo muito pequeno, conforme pode ser observado pelas colunas 2 e 3 da tabela 3. Para as instâncias scp41, scp42, scp43, scp45 e scp47 nenhum corte é possível de ser adicionado, pois a relaxação linear já provém uma solução inteira ótima para o *SCP*, conforme pode ser observado na coluna 4, linhas 1,2,3,5 e 7 da tabela 3. Para todas as outras instâncias desse conjunto o algoritmo *ARank1* conseguiu adicionar cortes de Chvátal-Gomory de rank-1, como pode ser observado pelas linhas 4,6,8,9,10,11,12,13,14 e 15 na coluna 6 da tabela 3. Para esse conjunto, a única instância que o algoritmo *ARank1* conseguiu resolver na otimalidade foi a instância scp410, conforme pode ser observado pela coluna 5, linha 10 da tabela 3. Nessa mesma tabela, retirando-se as instâncias que possuem uma relaxação linear inteira, pode-se observar que o tempo gasto pelo algoritmo *ARank1* foi bastane alto, conforme a coluna 7.

Instância	<i>IP</i>		Relaxação Linear	<i>ARank1</i>		
	Custo Solução	Tempo(s)		Custo Solução	#Cortes	Tempo(s)
scp41	429	0.84	429.00	429.00	0	0.00
scp42	512	0.85	512.00	512.00	0	0.00
scp43	516	0.86	516.00	516.00	0	0.00
scp44	494	0.86	494.00	494.00	6	863.71
scp45	512	0.85	512.00	512.00	0	0.00
scp46	560	0.89	557.25	558.94	32	1038.07
scp47	430	0.83	430.00	430.00	0	0.00
scp48	492	0.97	488.67	490.67	65	6118.55
scp49	641	0.90	638.54	639.87	75	8054.13
scp410	514	0.90	513.50	514.00	14	1349.04
scp61	138	1.23	133.14	133.53	52	7267.44
scp62	146	2.06	140.46	141.15	50	7284.06
scp63	145	1.26	140.13	141.46	72	7289.33
scp64	131	0.96	129.00	130.08	77	7251.63
scp65	161	1.94	153.35	154.13	50	7370.46

Table 3: Comparação entre os custos da solução e tempos obtidos entre o modelo *IP* e o algoritmo *ARank1* para as instâncias do conjunto 4 e 6.

Para todas as instâncias do conjunto A e B o CPLEX conseguiu encontrar soluções ótimas em um tempo pequeno, conforme pode ser observado pelas colunas 2 e 3 da tabela 4. Para essas instâncias o algoritmo *ARank1* não conseguiu encontrar a solução ótima para nenhuma delas, conforme pode ser observado pela coluna 5 da tabela 4. O algoritmo *ARank1* só conseguiu encontrar cortes de Chvátal-Gomory de rank-1 para as instâncias *scpa3*, *scpa5* e *scpb5*, conforme pode ser observado pela coluna 7 e linhas 3,5 e 10 da tabela 4. O tempo máximo de 120 segundos para o modelo de separação no algoritmo *ARank1* pode ter sido a causa de não encontrar cortes válidos para as demais instâncias. Para a instância *scpa3*, mesmo adicionando cortes válidos a solução encontrada possui o mesmo custo da solução na relaxação linear, conforme pode ser observado, comparando-se a coluna 4 e 5 da linha 3, indicando degenerações nas soluções.

Instância	<i>IP</i>		Relaxação Linear	<i>ARank1</i>		
	Custo Solução	Tempo(s)		Custo Solução	#Cortes	Tempo(s)
scpa1	253	9.95	246.84	246.84	0	251.75
scpa2	252	9.87	247.50	247.50	0	252.95
scpa3	232	9.48	228.00	228.00	6	1685.40
scpa4	234	8.76	231.40	231.40	0	248.60
scpa5	236	8.64	234.89	235.02	25	5826.96
scpb1	69	10.08	64.54	64.54	0	246.67
scpb2	76	10.87	69.30	69.30	0	256.28
scpb3	80	9.67	74.16	74.16	0	252.03
scpb4	79	11.52	71.22	71.22	0	250.54
scpb5	72	9.86	67.67	67.67	2	741.67

Table 4: Comparação entre os custos da solução e tempos obtidos entre o modelo *IP* e o algoritmo *ARank1* para as instâncias do conjunto A e B.

References

- [1] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. Introduction to Algorithms (pp. 162-164). MIT Press, 3rd edition, 2009.
- [2] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. Introduction to Algorithms (pp. 594-601). MIT Press, 3rd edition, 2009.
- [3] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. Introduction to Algorithms (pp. 658-662). MIT Press, 3rd edition, 2009.
- [4] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. Introduction to Algorithms (pp. 684-691). MIT Press, 3rd edition, 2009.
- [5] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. Introduction to Algorithms (pp. 693-697). MIT Press, 3rd edition, 2009.
- [6] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. Introduction to Algorithms (pp. 700-704). MIT Press, 3rd edition, 2009.