Comunidades em Grafos

João C. Abreu[1]

[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. Para se encontrar k comunidades em um grafo, com $k \leq n$, 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 resolvido, caso contrário o cálculo de centralidades deve ser novamente efetuado e a aresta $(i, j) \in E$ com maior centralidade é removida do 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 e a seção 5 apresenta as considerações finais.

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 x 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 análisa 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 a centralidade de todas as arestas. 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 prioridade mínima[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) ∈ 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 prioridade mínima possui ordem de complexidade igual a $O(nmlog_2n)$. 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 $O(knmlog_2n)$ no melhor caso e $O(nm^2log_2n)$ 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 encontradar k comunidades em um grafo não direcionado e não ponderado

Foram utilizados cinco conjuntos de instâncias de testes nos experimentos computacionais, três dessas instâncias foram retiradas de [7] e as outras duas foram geradas por esse trabalho. As instâncias de teste retiradas de [7] foram: karate, lesmis e adjnoun. A instância karate representa uma rede social de 34 membros de um clube de karate, a instância lesmis representa o número de ocorrências de determinados caracteres em um livro e a instância adjnoun representa o número de ocorrências de palavras em um livro. As instâncias geradas vão ser chamadas de path e completo. Essas instâncias foram geradas pois vão explorar o melhor e o pior caso dos algoritmos apresentados. Uma instância path é um grafo onde o nó $i \in V$ está ligado ao nó $i + 1 \in V$ formando apenas um caminho simples. Foram gerados três instâncias path com 50, 100 e 200 nós. Uma instância completo é um grafo completo, ou seja, existe uma aresta entre cada par de nós do grafo. Foram geradas três instâncias completo com 50, 100 e 200 nós. A tabela 2 apresenta todas as instâncias de teste. Na coluna 1 dessa tabela está o nome da instância, a coluna 2 apresenta o número de nós da instância e a coluna 3 mostra o número de aresta de cada instância.

No primeiro experimento foi comparado a performance dos algoritmos das seções 3.1, 3.2, 3.3, 3.4 e 3.5 nas instâncias retiradas de [7]. A figura 1 mostra os resultados obtidos com a execução dos algoritmos para a instância karate. Essa figura mostra que o algoritmo mais rápido

Instância	# Nós	# Arestas
karate	34	78
lesmis	77	254
adjnoun	112	425
path_50	50	49
path_100	100	99
path_200	200	199
completo_50	50	2450
completo_100	100	9900
completo_200	200	39800

Table 2: Características das instâncias de testes

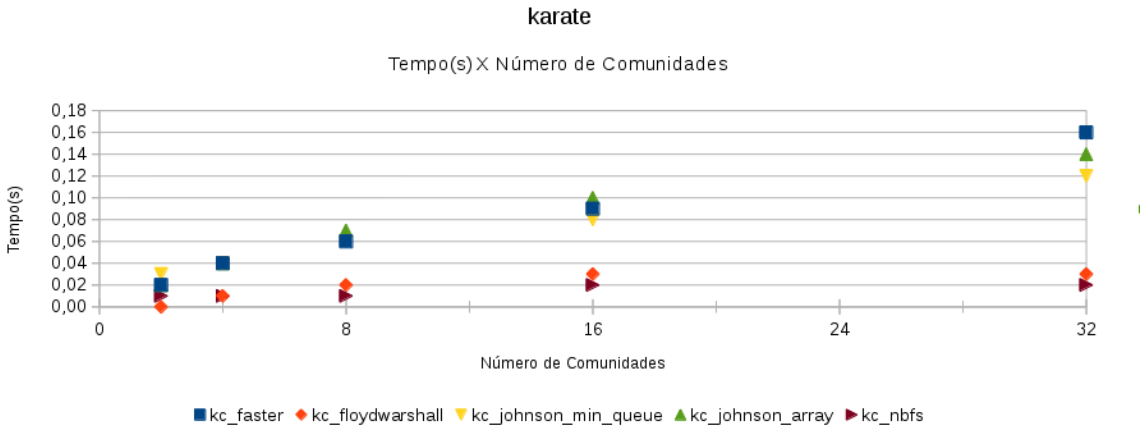

Figure 1: Comparação de tempo em segundos pelo número de comunidades:2, 4, 8, 16 e 32 para os cinco algoritmos apresentados nas seções 3.1, 3.2, 3.3, 3.4 e 3.5 e excutado na instância de teste karate

foi o de kc_nbfs seguido do kc_floydwarshall, essa figura também mostra que a medida que o número de comunidades aumenta o algoritmo kc_faster passa a ser o mais lento. O algoritmo kc_johnson_min_queue passa a ser mais rápido que o algoritmo kc_johnson_array com o aumento do número de comunidades.

A figura 2 mostra os resultados obtidos com a execução dos algoritmos para a instância lesmis. Essa figura mostra que o algoritmo mais rápido foi o de kc_nbfs seguido do kc_floydwarshall, essa figura também mostra que a medida que o número de comunidades aumenta o algoritmo kc_faster passa a ser o mais lento. O algoritmo kc_johnson_min_queue passa a ser mais rápido que o algoritmo kc_johnson_array com o aumento do número de comunidades.

A figura 3 mostra os resultados obtidos com a execução dos algoritmos para a instância adjnoun. Essa figura mostra que o algoritmo mais rápido foi o de kc_nbfs seguido do kc_floydwarshall, essa figura também mostra que a medida que o número de comunidades au-

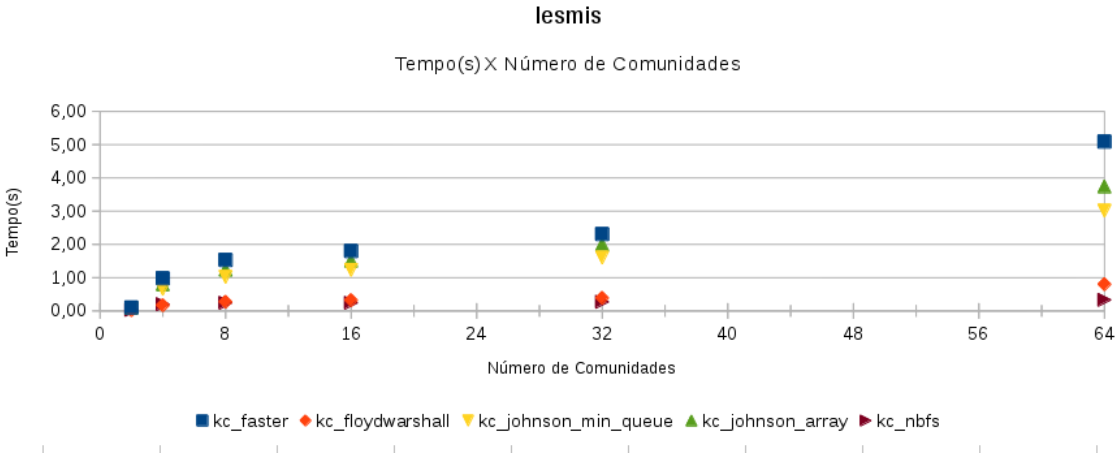

Figure 2: Comparação de tempo em segundos pelo número de comunidades:2, 4, 8, 16, 32 e 64 para os cinco algoritmos apresentados nas seções 3.1, 3.2, 3.3, 3.4 e 3.5 e excutado na instância de teste lesmis

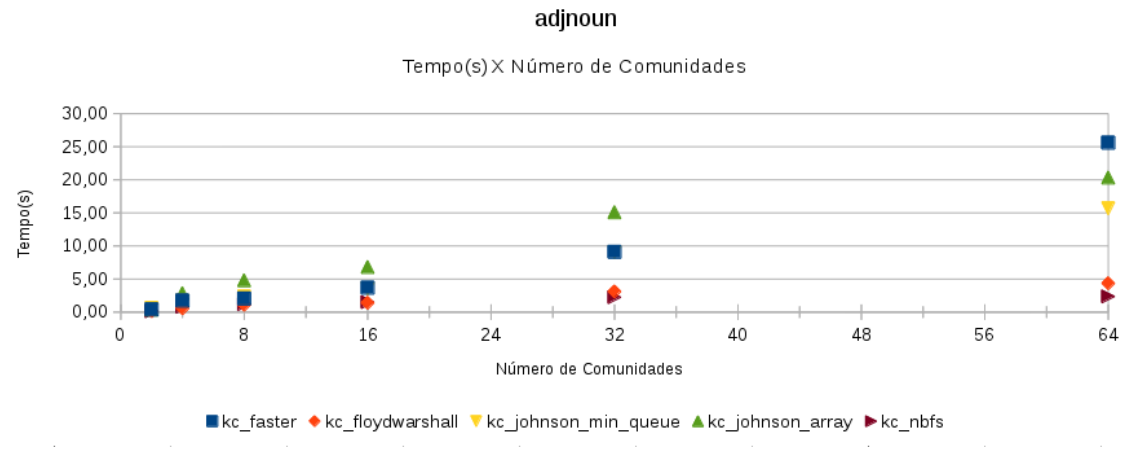

Figure 3: Comparação de tempo em segundos pelo número de comunidades:2, 4, 8, 16, 32 e 64 para os cinco algoritmos apresentados nas seções 3.1, 3.2, 3.3, 3.4 e 3.5 e excutado na instância de teste adjnoun

menta o algoritmo kc_faster passa a ser o mais lento. O algoritmo kc_johnson_min_queue passa a ser mais rápido que o algoritmo kc_johnson_array com o aumento do número de comunidades.

No segundo experimento foi comparado a performance dos algoritmos das seções 3.1, 3.2, 3.3, 3.4 e 3.5 no melhor e pior caso. No melhor caso, os algoritmos foram executados nas instâncias path50, path100 e path200 com o número de comunidades k igual a 50, 100 e 200 respectivamente, assim todas as $k-1$ arestas do grafo precisaram ser retiradas. No pior caso, os algoritmos foram executados nas instâncias completo50, completo100 e completo200 com número de comunidades k igual a 50, 100 e 200 respectivamente, obrigando a retirada de todas as arestas do grafo. Para comparar os algoritmos, o tempo de execução de cada um foi dividido

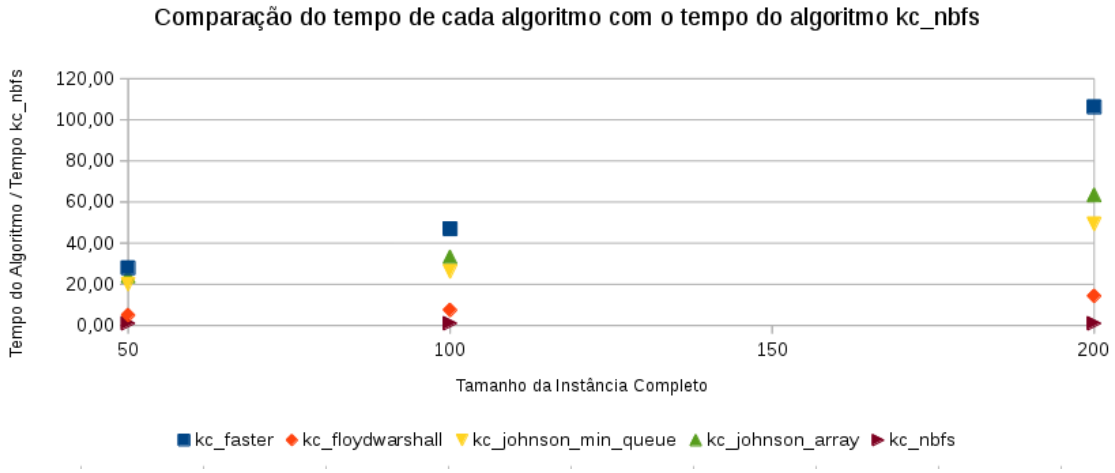

Figure 4: Comparação da razão do tempo gasto dos algoritmos pelo tempo gasto pelo algoritmo mais rápido nas instâncias path50, path100 e path200

pelo tempo de execução do algoritmo kc_nbfs que foi o algoritmo que obteve o menor tempo de resposta. A figura 4 apresenta os resultados obtidos para as instâncias path50, path100 e path200. Nessa figura o eixo vertical representa o tempo que o algoritmo gastou dividido pelo tempo gasto pelo algoritmo kc_nbfs e o eixo horizontal representa cada uma das instâncias path. Obervando essa figura vemos que o algoritmo kc_faster é o algoritmo que apresenta os piores resultados e com o aumento do número de arestas no grafo path a razão do tempo gasto por esse algoritmo e o algoritmo kc_nbfs aumenta muito. O algoritmo kc_floydwarshall consumiu tempos similares aos tempos do algoritmo kc_nbfs. O algoritmo kc_johnson_min_queue foi mais rápido do que o algoritmo kc_johnson_array.

A figura 5 apresenta os resultados obtidos para as instâncias completo50, completo100 e completo200. Nessa figura o eixo vertical representa o tempo que o algoritmo gastou dividido pelo tempo gasto pelo algoritmo kc_nbfs e o eixo horizontal representa cada uma das instâncias completo. Obervando essa figura vemos que o algoritmo kc_faster é o algoritmo que apresenta os piores resultados e com o aumento do número de arestas no grafo completo a razão do tempo gasto por esse algoritmo e o algoritmo kc_nbfs aumenta muito. O algoritmo kc_floydwarshall consumiu tempos similares aos tempos do algoritmo kc_nbfs. O algoritmo kc_johnson_min_queue foi mais rápido do que o algoritmo kc_johnson_array.

Comparando as figuras 4 e 5 a razão entre o tempo dos algoritmos e o tempo do algoritmo kc_nbfs diminui, isso pode ser explicado porque com o aumento do número de arestas no grafo o algoritmo kc_nbfs tende a ser muito pior.

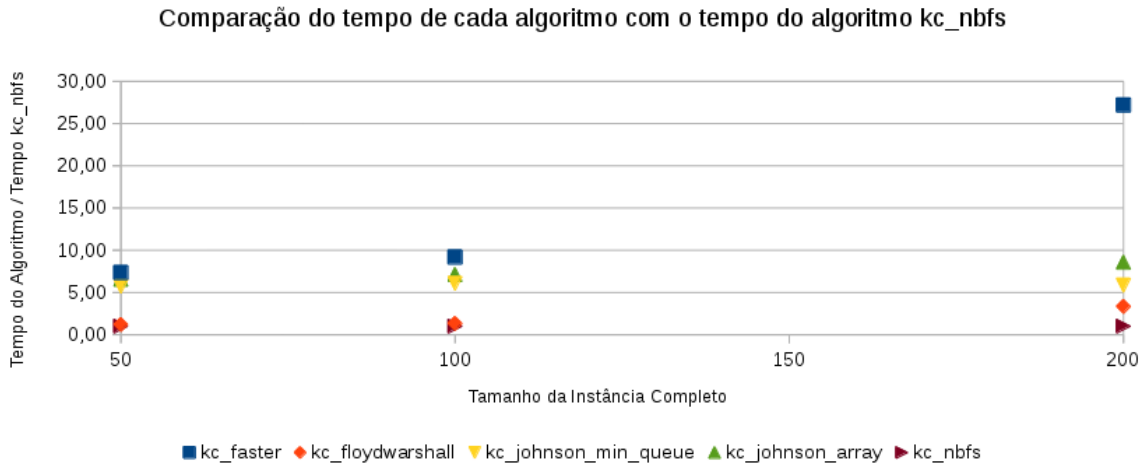

Figure 5: Comparação da razão do tempo gasto dos algoritmos pelo tempo gasto pelo algoritmo mais rápido nas instâncias completo50, completo100 e completo200

5. Considerações finais

Esse trabalhou apresentou o problema de encontrar k comunidades em um grafo não dirigido e não ponderado e cinco algoritmos exatos para a resolução desse problema. O algoritmo kc_nbfs foi o que obteve o menor tempo de execução em todos os experimentos realizados. O Algoritmo kc_floydwarshall obteve tempos de execução pouco pior do que o algoritmo kc_nbfs. O algoritmo kc_faster foi o que obteve os piores tempos de execução em todos os experimentos realizados. O algoritmo kc_johnson_min_queue obteve tempos sempre melhores do que o algoritmo kc_johnson_array.

Todo código fonte produzido por esse trabalho pode ser obtido no endereço eletrônico: *https* : *//github.com/joaojunior/busca_por_comunidades*

References

[1] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. Introduction to Algorithms (pp. 162-164). MIT Press, 3nd edition, 2009.

[2] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. Introduction to Algorithms (pp. 594-601). MIT Press, 3nd edition, 2009.

[3] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. Introduction to Algorithms (pp. 658-662). MIT Press, 3nd edition, 2009.

[4] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. Introduction to Algorithms (pp. 684-691). MIT Press, 3nd edition, 2009.

[5] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. Introduction to Algorithms (pp. 693-697). MIT Press, 3nd edition, 2009.

[6] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. Introduction to Algorithms (pp. 700-704). MIT Press, 3nd edition, 2009.

[7] Mark Newman. Network data, April 2014.