Universidade Federal de Uberlândia
Faculdade de Computação
GBC052 - Análise de Algoritmos

Atividade prática

Algoritmos de aproximação em grafos

Felipe Augusto Nunes Cintra - 11811BCC039
felipeanc@ufu.br

1 Relatório prático da atividade

1.1 Descrição

Há determinados problemas no campo da Computação que, até os dias atuais, permanecem em aberto. Para esses, não foram encontrados algoritmos que gerem soluções ótimas em tempo polinomial. Após inúmeras pesquisas, abordagens que geram resultados próximos a solução ótima foram encontradas.

Neste documento será abordado um destes algoritmos de aproximação: coloração dos vértices de um grafo. Para ele, foi desenvolvido um algoritmo que visa solucionar o problema da melhor maneira possível. E mais a frente testaremos esse programa, e compararemos seus resultados com as soluções ótimas esperadas para cada caso de teste.

1.2 Implementação

O algoritmo previamente citado foi desenvolvido utilizando a linguagem de programação C++ (mais precisamente, revisão ISO/IEC C++17), implementado e testado em ambiente Windows, com o apoio da IDE CodeBlocks.

Tal escolha foi feita considerando os inúmeros recursos da linguagem e, é claro, por questão de afinidade. Serão utilizados recursos já implementados (*built-in*) da STL (*standard template library*), a qual oferece inúmeras estruturas de dados e algoritmos que nos auxiliarão na resolução dos problemas. Para representar o grafo foi utilizada uma lista de adjacência, e vetores auxiliares de inteiros (e.g. marcar visitados, guardar as cores). Para mensurar o tempo de execução do algoritmo, foi utilizada a biblioteca *chrono*.

2 Coloração dos vértices de um grafo

Dentre os problemas de coloração em grafos, a coloração de vértices é o mais comum. A ideia geral é: dadas m cores, encontrar uma forma de pintar os vértices do grafo utilizando o mínimo de cores possível, de modo que vértices adjacentes não possuam a mesma cor. Este número mínimo, é o que chamamos de **número cromático**.

Dentre as possíveis abordagens para solucionar esse problema, a escolhida para esse trabalho foi **backtracking**. De maneira geral, escolheremos um vértice inicial, e a primeira cor a se utilizar, e faremos uma busca em profundidade. Para cada vértice visitado, serão testada todas as cores até que se ache uma que atenda às restrições. Esse processamento foi dividido em duas funções: uma que percorre em profundidade o grafo, e outra que confere se uma dada cor pode ser aplicada à um vértice.

Abaixo apresentarei o método de busca no grafo. A implementação detalhada e completa do programa está disponível nesse mesmo repositório.

2.1 Busca em profundidade em grafos

Para percorrer um grafo, existem algumas possibilidades, aqui usaremos busca em profundidade (DFS). O algoritmo abaixo é responsável por atingir todos os vértices do grafo, e atribuir a eles uma cor.

Ele recebe como entrada um *vértice* u e uma *cor* c, marca u como visitado e marca ele com a cor c. Após isso, para cada *vértice* v conectado ao vértice u, serão testadas, pelo método **confere-cor(v, i)**, as cores disponíveis. Ao encontrar uma, o método de busca é invocado novamente, recebendo como parâmetro o vértice v, e a cor i.

```
//Algoritmo 1 - DFS
//Entrada: vertice u, cor c
void colorir(int u, int c){
    visitados[u] = VISITED;
    cor[u] = c;
    for(int j = 0; j < AdjList[u].size(); j++){
        int v = AdjList[u][j];
        if(visitados[v] == UNVISITED){
            for(int i = 0; i < m; i++) //m -> quantidade de cores
                if(confere_cor(v, i) && cor[v] == UNCOLORED)
                    colorir(v, i);
        }
    }
}
```

A princípio, algoritmos para percorrer grafos (tanto em largura, quanto profundidade) possuem complexidade de tempo igual à $\mathcal{O}(V + E)$, sendo V o número de vértices do grafo, e E o número de arestas (*edges*). No caso apresentado acima, teremos **mais um fator** a ser considerado para essa análise de tempo: a quantidade m de cores disponíveis para pintar o grafo.

Para cada vértice, testaremos as cores disponíveis para decidir qual usar. Considerando m possibilidades de cor, teremos no pior caso conferido todas elas. Então, para cada vértice, são m possibilidades de cor, resultando numa complexidade de $\mathcal{O}(V + E + V * M)$.

2.2 Testes

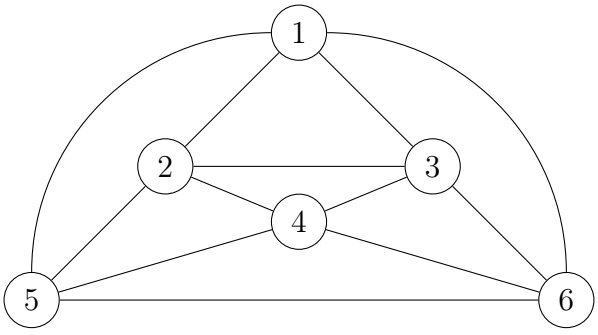
Para mensurar o desempenho dessa implementação, foram selecionados 5 grafos de tamanhos variados e um conjunto com 8 cores, que serão as mesmas para todos os testes. Adianto que dificilmente todas as cores serão utilizadas, normalmente se chegará a um limite superior igual ao maior grau entre os vértices do grafo + 1.

Cada grafo será executado 100 vezes, usaremos o tempo de execução para calcular a média da performance, e o desvio padrão desse conjunto de dados. O tempo de execução será medido em milisegundos com o auxílio da bibliotecha *crono.h*, e este representará o tempos em Ticks de CPU, diferentemente do tempo de um relógio normal (*wall time clock*).

O algoritmo, ao final de cada teste, gerará um arquivo CSV com os valores obtidos em cada iteração, descrevendo as cores que foram usadas para colorir cada vértice, o total de cores e o tempo gasto. Esses arquivos podem ser encontrados nesse mesmo repositório.

2.2.1 Grafo 1 - Grafo octaédrico

O grafo octaédrico é um grafo com 6 vértices e 12 arestas, que possui a conectividade de um octaedro (ver mais).

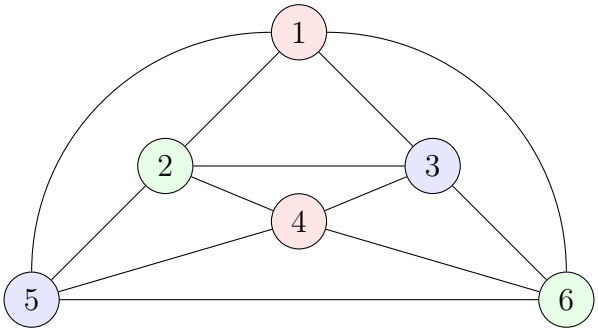


Abaixo temos a matriz de adjacência do grafo 1:

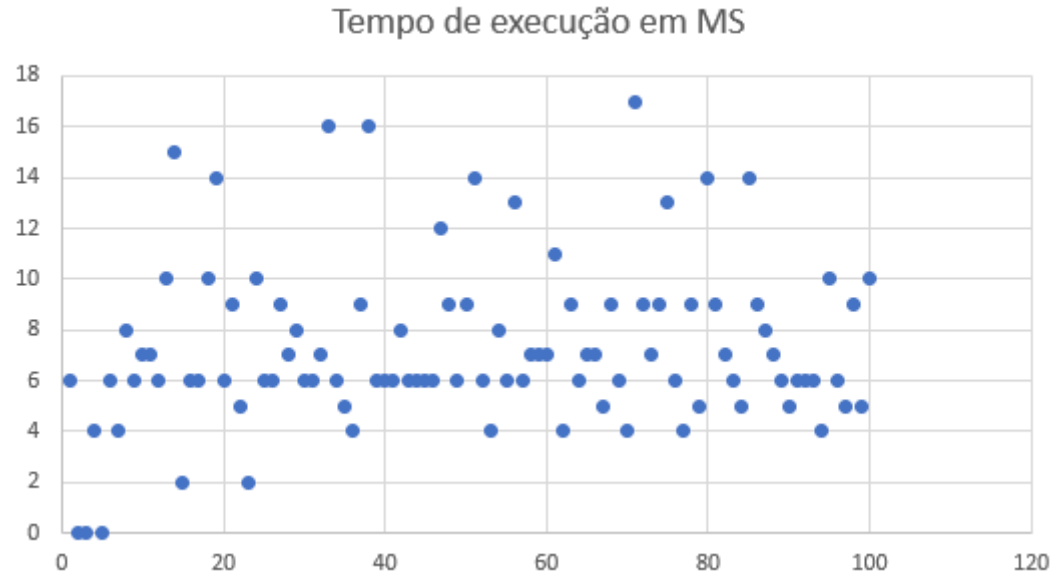
Vértices	1	2	3	4	5	6
1	0	1	1	0	1	1
2	1	0	1	1	1	0
3	1	1	0	1	0	1
4	0	1	1	0	1	1
5	1	1	0	1	0	1
6	1	0	1	1	1	0

Nesse grafo, todos os vértices têm grau igual d = 4, ou seja, o limite superior para o número de cores a ser usado é d+1 = 5. Entretanto, 3 cores já são suficientes para colorir todo o grafo, considerando as restrições do problema.

Após 100 execuções, o algoritmo teve um **tempo médio de execução** igual a **7.2000ms** e **desvio padrão de 4.3116ms**. Em todas iterações, foram utilizadas 3 cores para colorir todo o grafo. Ou seja, o algoritmo atingiu o resultado ótimo esperado. O grafo ficou assim:

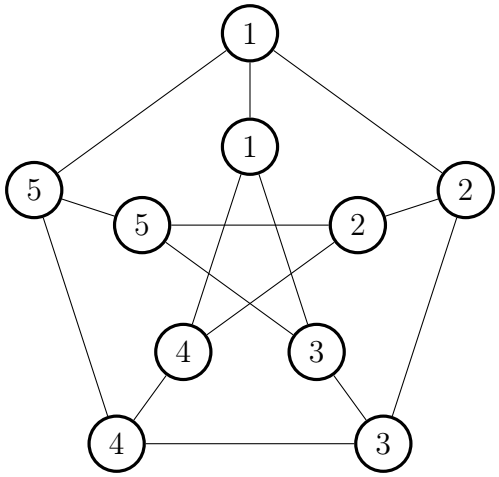


Abaixo temos um gráfico de dispersão com os valores do tempo de execução do algoritmo para o grafo acima. O eixo X representa as iterações (1 a 100) e o eixo Y são os valores em milisegundos. Percebemos que a maior concentração de valores está no intervalo [6, 9]. Temos valores extremos, como 2ms e 17ms.Por fim, é válido dizer que em algumas iterações o timer retornou 0ms, isso se deve ao fato de a biblioteca utilizada não ser totalmente consistente, podendo então em algumas situações não medir o tempo corretamente.



2.2.2 Grafo 2 - Grafo de Petersen

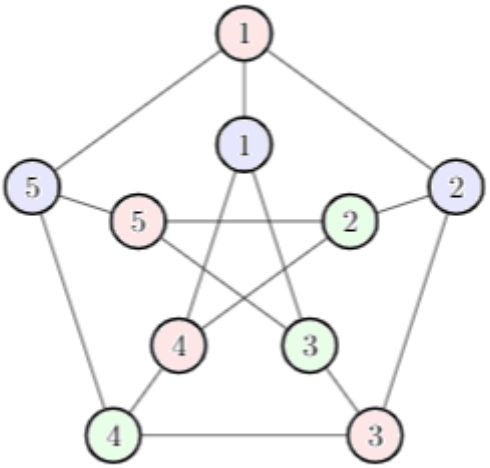
O Grafo de Petersen é um grafo não direcionado com 10 vértices e 15 arestas. É um grafo conhecido nos estudos de Teoria dos Grafos e utilizado em argumentações de vários problemas dessa área (ver mais).



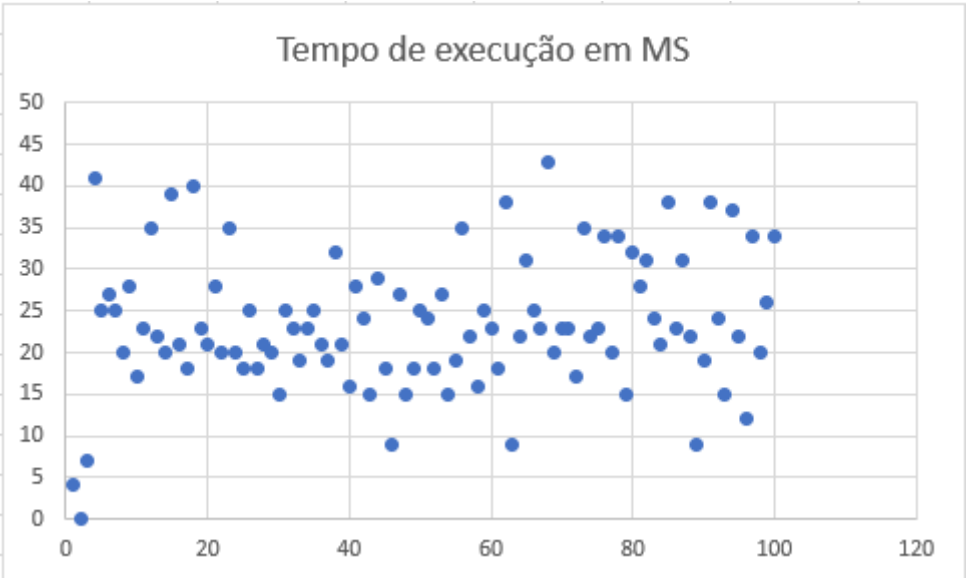
No grafo acima, o subgrafo interno (com formato de estrela) teve seus vértices numerados com os mesmos valores do subgrafo externo (1...5), porém na matriz de adjacência os vértices do subgrafo interno serão representados pelo seu valor acrescido de 5. Ou seja, o vértice 1 interno será 6, o vértice 2 será o 7, e assim sucessivamente. Abaixo temos a matriz de adjacência do grafo 2:

Vértices	1	2	3	4	5	6	7	8	9	10
1	0	1	0	0	1	1	0	0	0	0
2	1	0	1	0	0	0	1	0	0	0
3	0	1	0	1	0	0	0	1	0	0
4	0	0	1	0	1	0	0	0	1	0
5	1	0	0	1	0	0	0	0	0	1
6	1	0	0	0	0	0	0	1	1	0
7	0	1	0	0	0	0	0	0	1	1
8	0	0	1	0	0	1	0	0	0	1
9	0	0	0	1	0	0	1	1	0	0
10	0	0	0	0	1	0	1	1	0	0

Esse grafo pode ser, idealmente, colorido com apenas 3 cores, tendo um limitante superior igual d + 1 = 4, onde d é o grau máximo do grafo. Após 100 execuções, o algoritmo conseguiu atingir esse valor ideal. Em 100 iterações, ele coloriu o grafo com 3 cores, com tempo médio de execução igual a **23.4700ms** e desvio padrão de **8.0690ms**. A coloração resultante foi a seguinte:

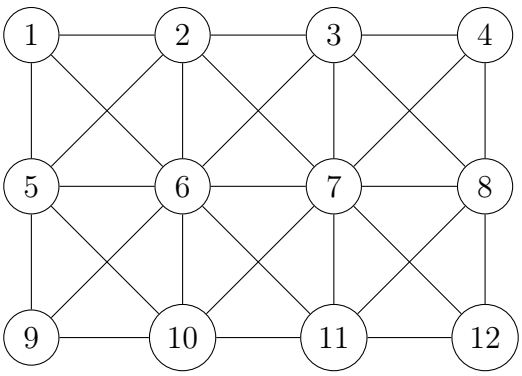


No gráfico de dispersão abaixo, a maior concentração de valores ficou no intervalo [15, 25]. E tivemos casos extremos, com o algoritmo resolvendo em menos de 5ms, ou quase 45 ms.



2.2.3 Grafo 3 - Grafo retangular

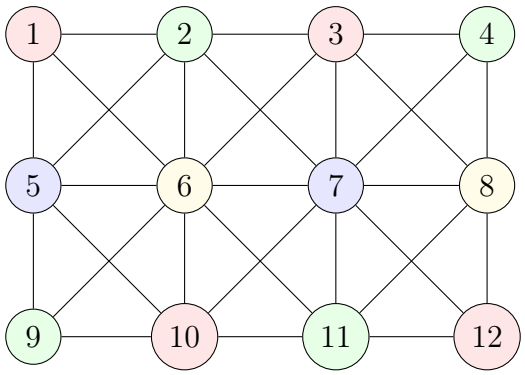
O próximo grafo contém 12 vértices e 29 arestas. Esse formato é comumente conhecido como *grid graph* (ver mais), porém o grafo utilizado apresenta arestas conectando diagonalmente os vértices. Utilizaremos um *grid* de 4x3.



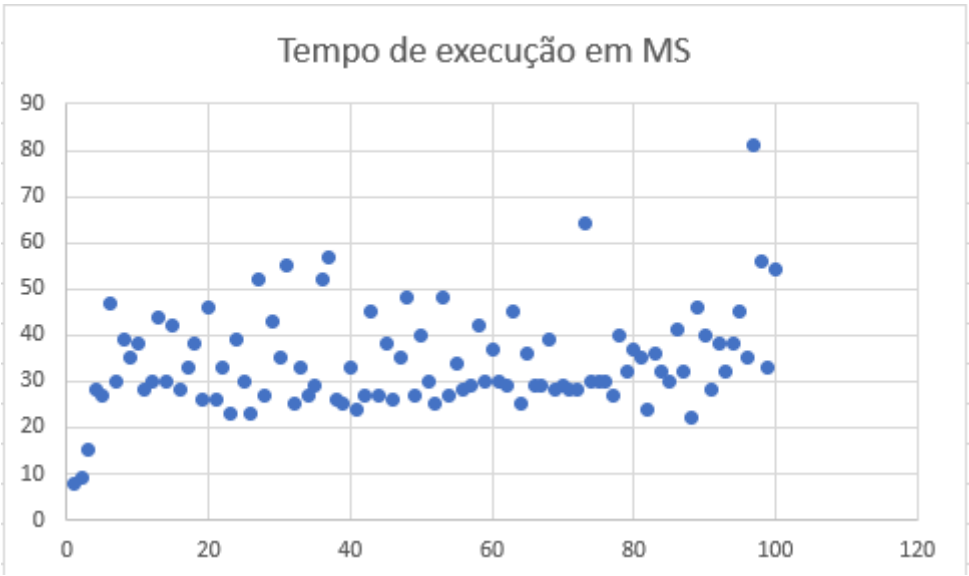
Abaixo temos matriz de adjacência desse grafo.

Vértices	1	2	3	4	5	6	7	8	9	10	11	12
1	0	1	0	0	1	1	0	0	0	0	0	0
2	1	0	1	0	1	1	1	0	0	0	0	0
3	0	1	0	1	0	1	1	1	0	0	0	0
4	0	0	1	0	0	0	1	1	0	0	0	0
5	1	1	0	0	0	1	0	0	1	1	0	0
6	1	1	1	0	1	0	1	0	1	1	1	0
7	0	1	1	1	0	1	0	1	0	1	1	1
8	0	0	1	1	0	0	1	0	0	0	1	0
9	0	0	0	0	1	1	0	0	0	1	0	0
10	0	0	0	0	1	1	1	0	1	0	1	0
11	0	0	0	0	0	1	1	1	0	1	0	1
12	0	0	0	0	0	0	1	1	0	0	1	0

Esse grafo, apesar do número de arestas , é bem simples e pode ser colorido, idealmente, usando apenas 4 cores. Apesar de seu limitante superior ser igual a $d + 1 = 7$, onde $d = 6$ é o grau máximo do grafo, o algoritmo em todas as 100 iterações utilizou apenas 4 cores. O tempo médio de execução foi de **34.2400 ms**, com um desvio padrão de **10.6951 ms**. A coloração final ficou da seguinte forma:

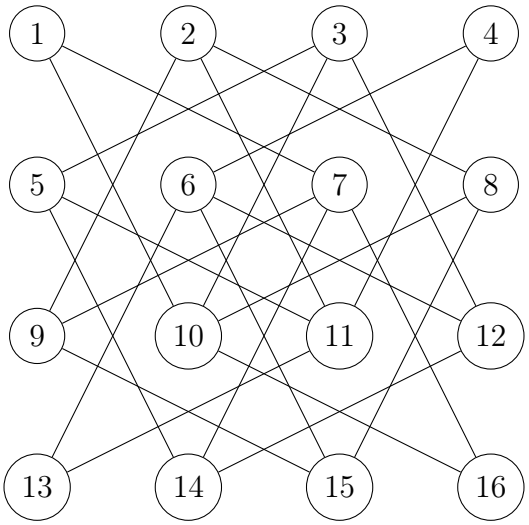


Da mesma forma como nos outros grafos, o gráfico de dispersão abaixo representa o tempo de execução em milisegundos das iterações de 1 a 100. A maior concentração de valores foi no intervalo [25, 35]. Tivemos também extremos de 8ms e 81ms.



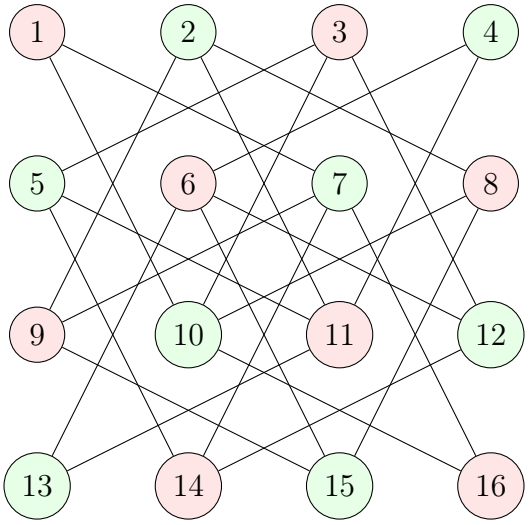
2.2.4 Grafo 4 - Grafo do Cavaleiro

O grafo do Cavaleiro, ou *Knight's graph* (ver mais) é um grafo onde cada vértice é uma posição de um tabuleiro de xadrez, e as arestas são os possíveis movimentos que o cavalo pode fazer (em L). Para esta ocasião, consideraremos um tabuleiro 4x4, ou seja, nosso grafo terá 16 vértices, e 24 arestas.

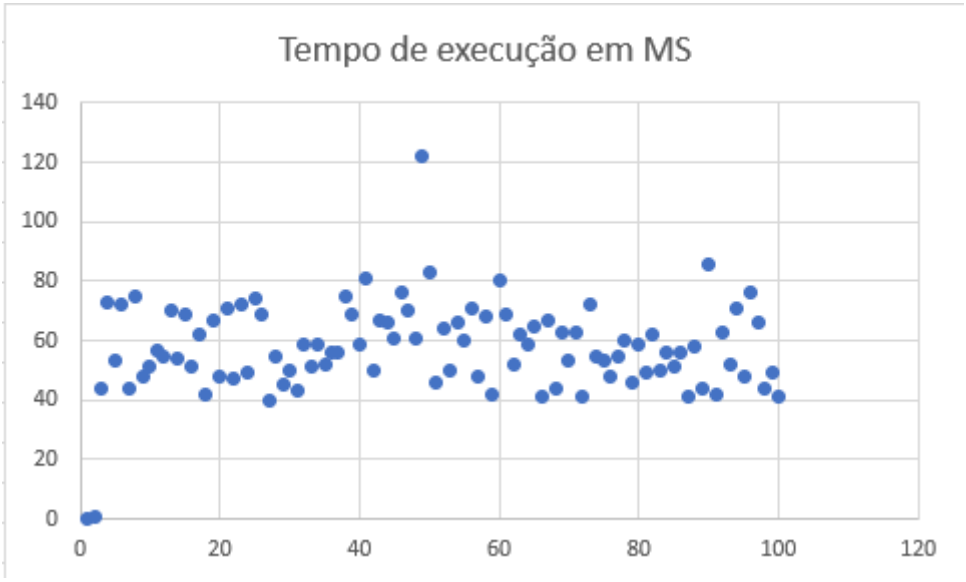


Considerando a complexidade desse grafo, infelizmente não acrescentarei ao relatório sua matriz de adjacência. Seguiremos direto para os resultados dos testes.

Mesmo complexo, o grafo do cavaleiro é, afinal, um grafo bipartido. Ou seja, ele pode ser dividido em 2 conjuntos, de modo que duas cores sejam suficientes para colorir todo ele. Após executar o algoritmo sobre esse grafo 100 vezes, obtemos um tempo médio de execução de **57.5100 ms** e um desvio padrão de **15.1436 ms**. A coloração final ficou da seguinte maneira:

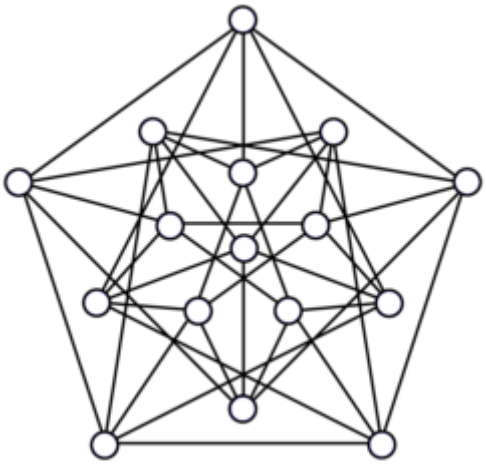


Abaixo temos o gráfico de dispersão contendo o tempo de execução das 100 iterações. Podemos notar que, diferentemente dos outros grafos, nesse os tempos de execução ficaram mais próximos, mais concentrados. No primeiro grafo, por exemplo, esses valores ficaram bem dispersos e com muitos valores extremos.

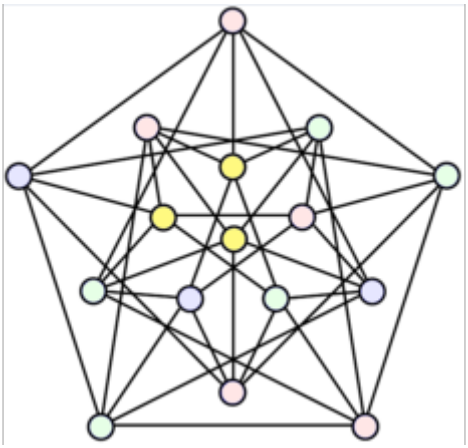


2.2.5 Grafo 5 - Grafo regular

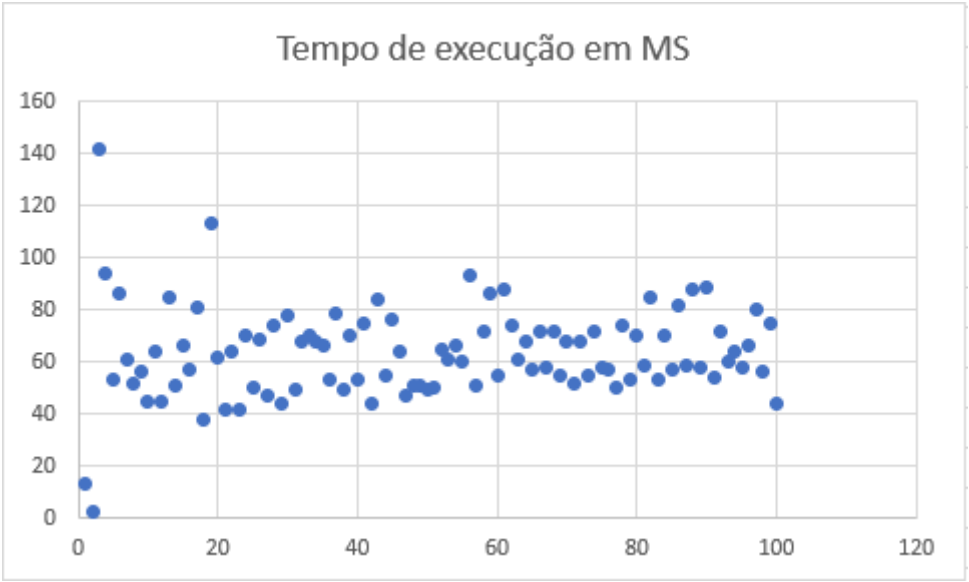
O último grafo de nossa seleção de testes será o grafo 5-regular, ou grafo de Clebsch (ver mais). Grafos regulares possuem uma característica: todos seus vértices possuem o mesmo número de adjacências, ou seja, todos têm o mesmo grau. Nessa ocasião, o grau do nosso grafo será $d = 5$. Seu número cromático é 4, e o limitante superior para o número de cores será igual a $d + 1 = 6$.



Em todas as 100 execuções, o algoritmo conseguiu atingir o número ideal de cores (4). Tal feito gastou em média **63.7000 ms** com um desvio padrão de **17.6741 ms**. O resultado final é o seguinte:



Pelos mesmos motivos do grafo 4, neste também não apresentarei a sua matriz de adjacência. Por fim, temos abaixo o gráfico de dispersão com os valores do tempo de execução em milisegundos do algoritmo.



Podemos perceber pelo gráfico que a maior concentração de valores está entre, aproximadamente, 50ms e 70ms. Mais uma vez, poucos extremos, valores bem precisos e coerentes com a dimensão do grafo.

2.3 Considerações finais

Ao analisar a perfomance do algoritmo ao executar os grafos apresentados, concluímos que, de fato, o tempo de execução está diretamente ligado a quantidade de vértices e arestas do grafo. Tivemos grafos pequenos, como o primeiro, que gastou em média 7.2ms para serem resolvidos, bem como grafos mais complexos, como o último, para o qual foram necessário em média 63.7ms, com extremos de mais de 120ms.

Outro fator interessante é que a quantidade de cores pouco, ou nada, influencia no tempo de execução. Em todos os casos de teste operamos com um conjunto de 8 cores, e alterando o tamanho desse conjunto, não obtêm-se mudanças significativas a ponto de dizer que o número de cores disponíveis influencia no tempo gasto.

Podemos dizer que o algoritmo faz escolhas "gulosas"ao colorir o grafo. Cada vez que ele pinta um vértice, a cor utilizada foi a próxima disponível no conjunto, ou seja, ele não seleciona uma cor aleatória. Com isso, garantimos que o algoritmo usará o mínimo de cores possível, adicionando mais uma cor ao resultado apenas quando necessário.

Referências

[1] Wolfram Alpha. *Graph Theory*. https://mathworld.wolfram.com/topics/GraphTheory.html. Acessado em: 07/06/2021.

[2] Múltiplos autores. Materiais disponibilizados em aula.

[3] Steve Halim and Felix Halim. *Competitive Programming 3. The new lower bound of programming contests*. 1st Edition. Lulu, 2013.