Universidade Federal de São João del-Rei
Algoritmo e Estrutura de Dados III
Trabalho Prático 3

Felipe Francisco Rios de Melo
Thales Mrad Leijoto

1.Introdução

Em teoria dos grafos, coloração de grafos é um caso especial de rotulagem de grafos, é uma atribuição de rótulos tradicionalmente chamados "cores" a elementos de um grafo sujeita a certas restrições.

Uma coloração em um grafo, consiste na atribuição de cores aos vértices em de modo que vértices adjacentes tenham cores distintas. A solução do problema do número cromático em grafo tem como objetivo determinar o menor valor de que possibilite uma coloração.

A coloração de grafos goza de muitas aplicações práticas, bem como desafios teóricos. Ao lado dos tipos clássicos de problemas, limitações diferentes também podem ser definidas no grafo, ou na forma como uma cor é atribuída, ou mesmo sobre a própria cor. Ela chegou até a se popularizar com o público em geral na forma da popular série de quebra-cabeças Sudoku. A coloração de grafos ainda é um campo de pesquisa muito ativa.

O objetivo deste trabalho é o estudo de grafos aplicado à coloração de vértices. Para tal, faremos uma breve apresentação dos algoritmos implementados (um algoritmo exato e duas heurísticas), bem como a análise de complexidade e discussão dos resultados.

2. Modelagem e solução do problema

2.1. Problema proposto

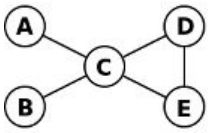
O problema proposto por este trabalho prático é, dado um grafo G = (V, A), não direcionado e não ponderado, deve se encontrar o menor número k de cores que é possível colorir o grafo, tal que cor(u) $\neq$ cor(v), para toda aresta (u, v). Em outras palavras, os vértices adjacentes devem ter cores diferentes.

2.2. Modelagem

A estrutura do grafo foi modelada como uma matriz de adjacência, uma das principais formas de representar um grafo.
Dado um grafo *G* com *n* vértices, podemos representá-lo em uma matriz *n* x *n*. De forma geral o valor a_{ij} guarda informações sobre como os vértices v_i e v_j estão relacionados (isto é, informações sobre a adjacência de v_i e v_j).

Para representar o tipo de grafo que foi usado neste trabalho, um grafo não direcionado, simples e sem pesos nas arestas, basta que as posições a_{ij} da matriz contenha 1 se v_i e v_j são adjacentes e 0 caso contrário.

Exemplo de uma matriz de adjacência:



	A	B	C	D	E
A	0	0	1	0	0
B	0	0	1	0	0
C	1	1	0	1	1
D	0	0	1	0	1
E	0	0	1	1	0

Foi criado uma estrutura, também, para representar cada vértices do grafo, de acordo com seus atributos, são eles: identificação do vetor, cor, cor definitiva, grau do vértice em relação a quantidade de arestas ligadas a ele e grau do vértice de acordo com a quantidade de vértices já coloridos ligados a ele (grau de saturação). Esta estrutura foi usada exclusivamente na implementação das duas heurísticas de coloração.

2.3. Entrada e saída

Os dados de entrada são lidos de um arquivo de texto que é estruturado da seguinte forma: a primeira linha contém a quantidade de vértice do grafo, e as seguintes possuem as ligações entre os vértices (arestas), por exemplo, em um arquivo podemos ter:

```
3
1 0
0 2
```

Isto, representa que temos 3 vértices no grafo, no qual o vértice 1 é ligado ao vértice 0 (e vice-versa), e o vértice 0 e 3 são ligados reciprocamente, pelo fato de ser um grafo não direcionado.

Escolhido um algoritmo entre os três ou então os três, a saída é composta pelo menor número k de cores que foi possível colorir o grafo, seguido do seu tempo de processamento.

2.4 Solução proposta

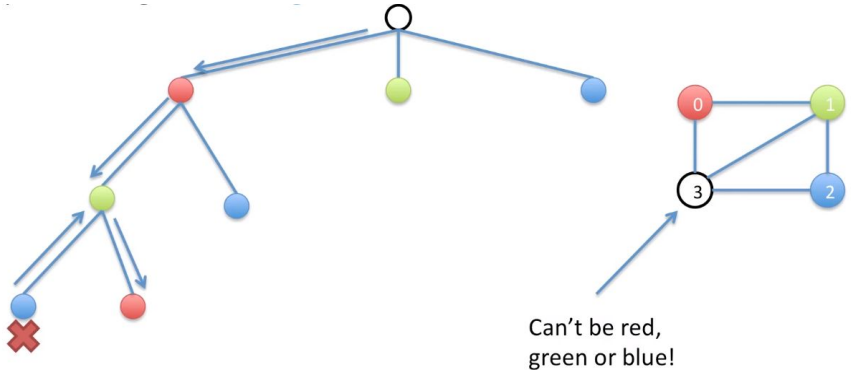
O problema do número cromático foi implementado, segundo três algoritmos distintos, um algoritmo que resulta sempre em uma solução ótima e resolve em tempo exponencial, e duas heurísticas que não garantem a solução ótima, mas que resolvem em tempo polinomial.

2.4.1. Algoritmo Exato

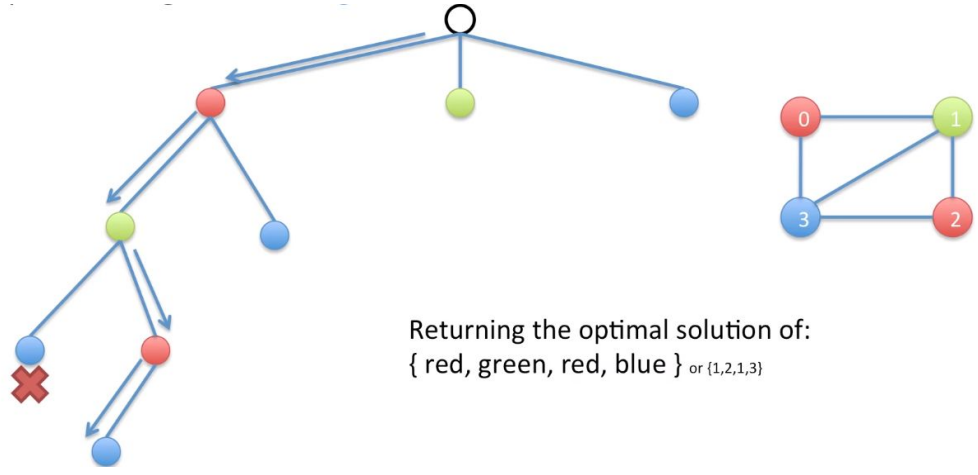
Para a implementação do algoritmo que garante sempre a solução ótima, foi usada a estratégia de força bruta, através do backtracking. O método backtracking soluciona problemas de otimização combinatória a partir da enumeração de possíveis soluções para um problema. Um algoritmo de backtracking é um método recursivo que gera uma árvore com todas as soluções possíveis da instância de entrada, de maneira que nós folhas representam soluções ótimas, e nós internos representam soluções parciais.

A ideia do algoritmo implementado é estabelecer cores aos vértices, um a um, começando do vértice de menor índice/identificador. Após atribuir uma cor a um vértice, checamos se a cor é segura, isto é, se a cor estabelecida segue a regra de que não se pode ter vértices adjacentes com a mesma cor. Se for encontrado uma cor segura, ela é adicionada como parte de uma possível solução ótima. Porém, se não for possível encontrar uma cor segura, e os vértices não estiverem totalmente coloridos, voltamos para trás na decisão, é o que chamamos de backtracking.

Por exemplo:
Estamos tentando colorir o seguinte grafo com apenas três cores, nos vértices 0, 1 e 2 foram encontrados cores seguras sem problemas, porém dependendo da escolha que fazemos no vértice 2, pode acarretar em problemas na hora de encontrar uma cor segura para o terceiro vértice. Foi o que aconteceu no seguinte esquema, foi atribuído a cor azul ao vértice 2, mas na hora de selecionar uma cor para o vértice 3 foi impossível encontrá-la, visto que o problema era usar apenas três cores e estas três cores já foram usadas por seus vértices adjacentes. Logo, é necessário dar um passo para trás.



Visto que a causa desse empecilho foi a cor estabelecida para o vértice 2, vamos alterar sua cor (de forma "segura"), e em seguida encontrar um cor para o vértice 3. Nessa configuração, agora foi possível colorir o vértice 3, com uma cor diferente de todos os seus vértices adjacentes, então concluímos que é possível colorir o vértice usando apenas três cores. Segue abaixo a representação da árvore:



Todo este processo é feito dentro de um laço de repetição, onde que, a cada iteração deste laço, o algoritmo de exaustão é executado, com um valor diferente de k passado como parâmetro. O laço começa com k igual a 1 e é incrementado 1 a cada repetição. O processo termina quando a função Backtracking() conseguir colorir todo o grafo com as k cores enviadas como parâmetro para a função. Assim, o valor retornado pelo algoritmo exato, é o número mínimo de cores necessárias para colorir o grafo.

2.4.2. Heurística 1

A estratégia da primeira heurística implementada consiste em colorir vértice por vértice, em ordem de vértice com maior grau até o vértice com o menor.
Primeiramente o algoritmo colore todos os vértices com a mesma cor e os ordena em ordem decrescente de acordo com o grau (que é medido através do número de arestas conectadas ao nodo). Então pegamos o primeiro vértice, colorimos ele com a primeira cor e verificamos se ele tem algum vértice adjacente com a mesma cor. Caso tenha, uma nova cor é atribuída a ele, e testamos novamentes as adjacências. Quando chegamos numa cor sem conflito, o programa prossegue colorindo todos os vértices adjacentes ao em questão (que ainda não passaram pelo algoritmo) com a mesma cor. E então o processo é repetido até ter passado por todos os vértices e descoberto qual o número mínimo de cores que são necessárias para colorir o grafo.

2.4.3 Heurística 2

A segunda heurística implementada é a DSatur criada por Daniel Brélaz em 1979. Essa estratégia colore os vértices baseando-se no grau de saturação de cada um, que é medido pelo número de cores diferentes ligadas em cada vértice. Quanto mais cores, maior o grau.
No começo, o algoritmo escolhe o vértice de índice 0, já que todos os nodos têm o mesmo grau de saturação, coloca a primeira cor nele, atualiza a saturação de todos os vértices (adjacentes a ele ficam com 1 de saturação) e pega o vértice mais saturado dentre os que restam. A partir daí o algoritmo passará por cada vértice fazendo a mesma coisa,
achando uma cor para ele, atualizando a saturação dos demais nós e passando para o próximo vértice mais saturado que ainda não tem uma cor definida. No caso da implementação feita neste trabalho, as cores são tratadas como números então, a maior cor, também significará quantas cores foram necessárias para colorir o grafo (esse valor é atualizado sempre que uma nova cor é adicionada).

3. Análise de Complexidade

Nesta seção faremos uma análise da complexidade de tempo e espaço das funções mais importantes do algoritmo. Cada função será analisada individualmente e em seguida calcularemos a complexidade total a partir da análise da função do programa principal. Ressaltamos que “V” é a quantidade de vértices, “A” a quantidade de arestas e “k” o número cromático. Cálculos e atribuições serão consideradas constantes, dessa forma, complexidade O(1).

3.1. verificaArquivo()

Tempo:
Percorre todo o arquivo, contando que o arquivo não está vazio, O arquivo chega ao fim quando é lido todas as arestas do grafo, temos como complexidade final, O(A), onde A é a quantidade de arestas.
Espaço: a função não possui alocação de memória, O(1).

3.2. alocaMatriz()

Tempo: Contêm um laço que vai de 0 até o valor do parâmetro linha passado para ela, que chamaremos de n, logo sua complexidade é O(n).
Espaço: É alocado na memória uma matriz bidimensional, complexidade $O(n^2)$.

3.3. inicializaGrafo()

Tempo:
Aloca a estrutura de dados grafos, bem como os componentes da struct (quantidade de vértices e a matriz de adjacência). A quantidade de vértice é obtida a partir da leitura da primeira linha do arquivo, O(1), e a matriz de adjacência é obtida a partir da função *alocaMatriz()*, passando como parâmetro a quantidade de vértice, O(V). Logo, a complexidade é O(max(1, V)), que é igual a O(V).

Espaço: A função que aloca a matriz (implicita na inicializaGrafo()), por fazer a alocação dinâmica de uma matriz bidimensional de tamanho V, contém uma complexidade espacial de $O(V^2)$.

3.4. constroiGrafo()

Tempo:
Esta função, possui um laço que termina quando se chega ao fim do arquivo, no qual dentro do laço, é lido a cada iteração uma linha do arquivo e usado o conteúdo lido nesta linha (vértice de origem e vértice de destino) como parâmetro para a função insereAresta(), função está que apenas atribui valores a uma posição da matriz de adjacencia, complexidade O(1). Contando que o arquivo chega ao fim quando é lido todas as arestas do grafo, temos como complexidade final, O(A), onde A é a quantidade de arestas.
Espaço: a função não possui alocação de memória, O(1).

3.5. Backtracking()

Tempo:
Há V^k possibilidades de solução, pois para cada vértice temos k escolhas. O algoritmo passa por todas elas, salvo as ramificações que são podadas através do backtracking. Porém, considerando que o pior caso será quando todas essas possibilidades forem testadas (sem podas), temos então a complexidade assintótica na ordem de $O(k^V)$.

Espaço: a função não possui alocação de memória, O(1).

3.6. AlgoritmoExato()

Tempo:
Em um loop, que inicia em k igual a 1, executa a função Backtracking(), até que a mesma retorne 1, o k é incrementado a cada iteração. O valor de k nunca será maior do que a quantidade de vértices, visto que caso k seja igual a quantidade de vértices, isso significa que podemos colorir cada vértice do grafo com uma cor distinta, logo o "teto" do laço é V. Temos então como complexidade $O(V * k^V)$.
Espaço: a função não possui alocação de memória, O(1).

3.7. calculaGrau()

Tempo:
Percorre um laço que vai de 0 até a quantidade de vértices, verificando e contabilizando a quantidade de arestas que o vértice possui, O(V).
Espaço: a função não possui alocação de memória, O(1).

3.8. inicializaVetorVertice()

Tempo:
Aloca memória para armazenar um vetor da struct vertice, e em um for que percorre todos os vértices (O(V)), seta todos os componentes do vetor de acordo com o vértice que os representa, todas estas atribuições são O(1), com exceção da atribuição do grau do vértice, que é computado pela função calculaGrau(), no qual possui complexidade O(V), até então, temos complexidade O(V * V), que é igual a $O(V^2)$. Se o parâmetro caso vier como 1, é rodado o algoritmo de ordenação insertionSort(). O insertionSort() é um algoritmo de ordenação conhecido e que possui complexidade no pior caso $O(V^2)$, que ocorre quando para toda iteração do laço existe uma troca de posição, $O(V^2)$. Logo, a complexidade quando é necessário executar o insertion sort é $O(\max(V^2, V^2))$ que é o próprio $O(V^2)$. E mesmo quando o parâmetro vier como 0, temos também a complexidade $O(V^2)$, que corresponde ao primeiro laço de atribuições.
Espaço:
A função contêm alocação de memória para um vetor da struct vértice, o tamanho deste vetor é dado pela quantidade de vértices no grafo, então a complexidade espacial do procedimento é O(V).

3.9. atribuiCor()

Tempo:
Essa função tem dois casos para serem analisados, quando é chamada pela heurística 1 e quando é chamado pela 2. No primeiro caso, a função executa dois laços aninhados que vão de 0 até V, então a complexidade é $O(V^2)$. No segundo caso, executa apenas um loop de 0 a V, complexidade O(V).
Espaço: a função não possui alocação de memória, O(1).

3.10. heuristica1()

Tempo:
Primeira chama a função inicializaVetorVertice() que tem complexidade O(V^2). A função tem um loop que passa por todos os vértices(O(V)). Dentro desse loop temos a função atribuiCor() que tem complexidade O(V^2) no pior caso e outro loop que também passa por todos os vértices O(V). Assim, a complexidade da heuristica1() é O(max(V^2,V*V^2,V*V)) = O(max(V^2,V^3,V^2)) = O(V^3).
Espaço: Chama a função inicializaVetorVertice() que tem complexidade O(V).

3.11. heuristica2()

Tempo:
Começa chamando a função inicializaVetorVertice() com complexidade de tempo O(V^2). Então a função prossegue com um loop passando por todos os vértices O(V) e tem alguns acontecimentos dentro desse loop: um outro loop que pode ter complexidade O(V), e dentro dos dois loops, a função *atribuiCor()* também O(V). Ainda dentro do primeiro loop, temos que atualizar a saturação e achar o novo vértice mais saturado ambas operações com custo O(V).
Então, a complexidade da heurística 2 ficou assim: O(max(V^2,V*V*V,V*V,V*V)) = O(max(V^2,V^3,V^2,V^2)) = O(V^3).
Espaço:
Chama a função inicializaVetorVertice() que tem complexidade O(V).

3.12. liberaArquivos()

Tempo: Apenas fecha os arquivos e libera a struct, O(1).
Espaço: a função não possui alocação de memória, O(1).

3.13. liberaGrafos()

Tempo:
Libera primeiramente a matriz de adjacência, por meio de um *for* de 0 a V, que percorre as colunas da matriz liberando-as, e depois fora do laço, libera-se as linhas, por fim é liberado a variável correspondente a struct grafo, todas estas instruções, com exceção do *for* que é O(V), possuem complexidade O(1), consequentemente, O(V) domina assintoticamente.
Espaço: a função não possui alocação de memória, O(1).

3.14. main()

Tempo:
A função *main* é a responsável por chamar todas estas funções citadas acimas, além de algumas outras que não entramos no mérito, por possuir complexidade O(1). Logo, concluímos que a complexidade assintótica do programa no pior caso é dado por O(max(1, V, A, V x k^V, V^3, V^3, 1, 1)), que é igual a O(V x k^V).

Espaço:
Quanto a complexidade espacial geral do programa, temos apenas a alocação da matriz bidimensional de adjacência e do vetor da struct vértice, contudo, o vetor é alocado somente quando se executa alguma das duas heurísticas, o backtracking não utiliza esta struct na implementação. Mas, supondo que estamos considerando o pior caso, então a pior complexidade de memória que o programa principal pode adquirir é $O(\max(V^2, V))$ que é igual a $O(V^2)$.

Observamos, então, que a função de que executa o algoritmo exato domina, no quesito tempo, assintoticamente as duas heurísticas no pior dos casos. Por outro lado, no quesito alocação de memória, os algoritmos heurísticos consomem mais espaço. E vale ressaltar que, mesmo a quantidade de arestas tendo um impacto sobre a complexidade, esse impacto é mínimo se comparado ao impacto da quantidade de vértices, então, realizamos a análise de complexidade em relação aos vértices e comentamos um pouco sobre a atuação das arestas no tópico 4 de resultados.

4. Experimentos e análise de resultados

Os testes foram realizados em um Intel Core i5 7ª geração i5-7200U, com 8gb de memória RAM. As entradas foram geradas através de um programa em C de terceiros que cria grafos podendo selecionar o número de vértices e arestas.

4.1. Tempo

A primeira análise feita baseado no número de arestas do grafo. Tomando como base um grafo com 10 vértices, foram salvos os tempos do algoritmo de força bruta e as heurísticas implementadas neste trabalho com a quantidade de arestas variando de 0 a 100% (sendo 0 um grafo sem arestas e 100% um grafo em que todos os vértices são adjacentes entre si).

GRÁFICO 1. Tempo do algoritmo de backtracking variando a proporção de arestas no grafo.

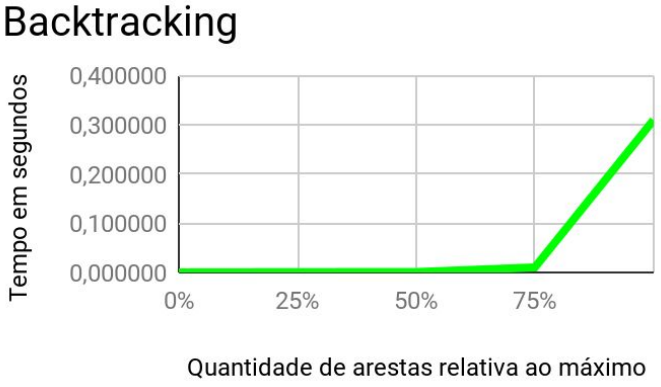
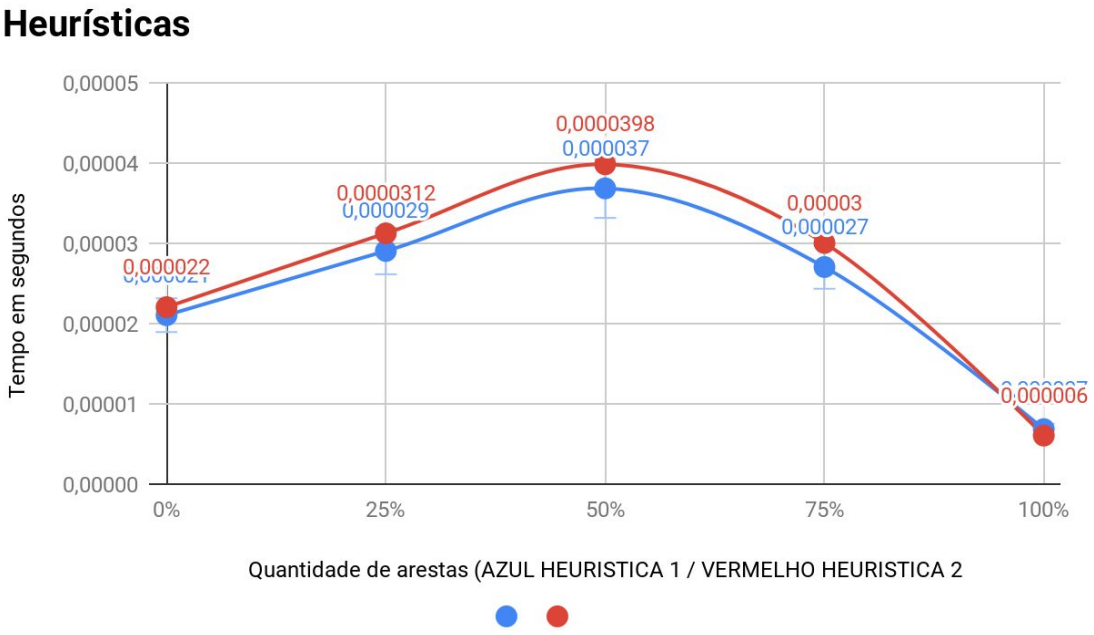


GRÁFICO 2. Tempo das heurísticas variando a proporção de arestas no grafo.



Note que não foi possível colocar o desempenho dos três algoritmos no mesmo gráfico devido a grande diferença de escala de tempo entre o backtracking e os demais.

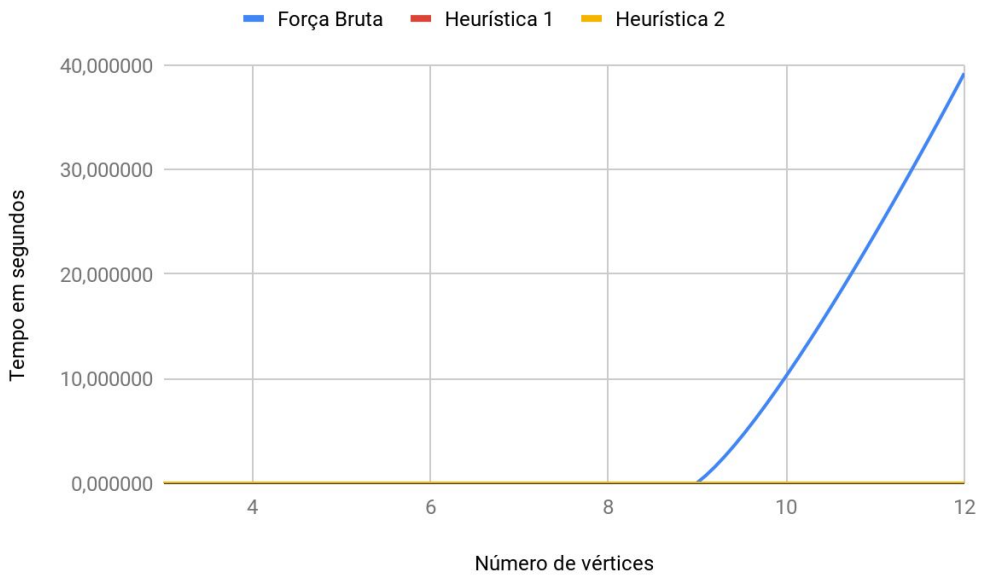
A partir do resultado deste teste é percebemos que o algoritmo de backtracking tem seu pior caso quando todos os vértices são adjacentes entre si, enquanto que as heurísticas têm um tempo de execução pior quando temos aproximadamente 50% de arestas do máximo que se pode ter para um grafo de 10 vértices.

O algoritmo de força bruta, possui pior tempo neste caso, devido ao aumento exacerbado das chamadas recursivas, que procuram uma cor segura para o vértice, mas só encontram na última cor segura testada. Logo o número mínimo de cor (k) será o número de vértice (todos os vértices são adjacentes entre si), no qual implica diretamente na complexidade $O(V * k^V)$.

Já as heurísticas, têm que fazer um número maior de comparações quanto menos arestas, porém tem que fazer um número maior de atribuições quanto mais arestas, são inversamente proporcionais as duas proporções. Por isso, as heurísticas encontram seu pior caso de tempo (em relação às adjacências) com 50% de arestas.

O próximo teste se trata da análise dos algoritmos quando submetidos ao seus respectivos piores casos relativos a quantidade de arestas (análise anterior), variando o número de vértices:

GRÁFICO 3. Tempo dos algoritmos em seus piores casos.



Aqui, observamos a relação entre a complexidade no pior caso entre os algoritmos, como era de se esperar o algoritmo de força bruta tem tempo de processamento excessivamente maior que as heurísticas, $O(V * k^V)$ contra $O(V^3)$.

Uma vez que o problema da determinação do número cromático é classificado como NP-completo, qualquer algoritmo exato empregado em sua resolução terá complexidade exponencial, desta maneira, técnicas heurísticas para solução aproximada desse problema em tempo polinomial são mais viáveis para grandes entradas. Porém, veremos a seguir, que não é necessário uma entrada tão grande assim, para se optar por algoritmos heurísticos.
Considerando o pior caso - todos os vértices adjacentes entre si - o algoritmo de força bruta consegue executar o problema da determinação do número cromático em um tempo satisfatório com uma entrada de até 13 vértices, gastando um tempo de 523,400036s, que equivale a 8 minutos e 43 segundos. Porém, diminuído a proporção de arestas é possível executar o algoritmo com uma quantidade de vértices maior.

Substituindo na ordem de complexidade, para V = 13 e k = 13, temos $13*13^{13}$, que é igual a 3.9373764×10^{15}. Se resolvermos testar o algoritmo para uma entrada 10 vezes maior, obtemos $130*130^{130}$, que resulta em 8.444601×10^{276}. Jogando em uma regra de 3 básica:

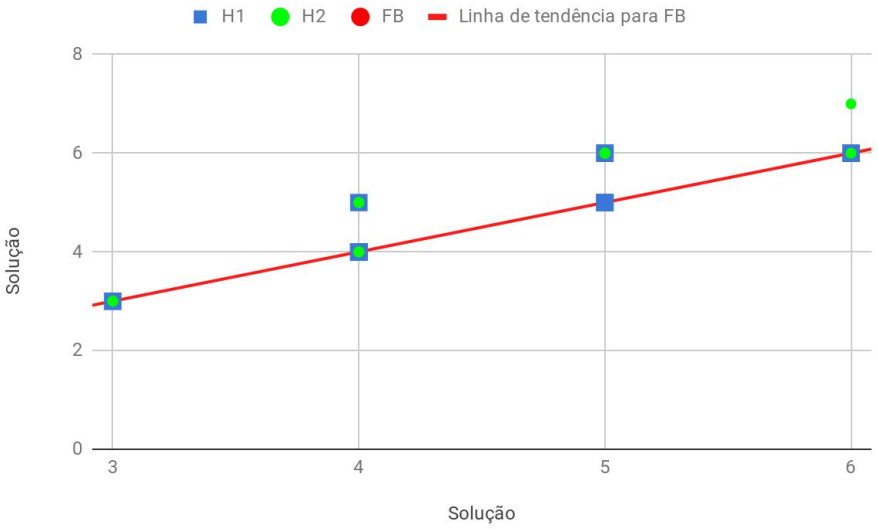
3.9373764×10^{15} --- 523.400036s
8.444601×10^{276} --- x

$x = 1.1225507 \times 10^{294}s = 3.56 \times 10^{283}$ milênios

Seria necessário 3.56×10^{283} milênios para executar o algoritmo de força bruta para uma entrada de 130 vértices.

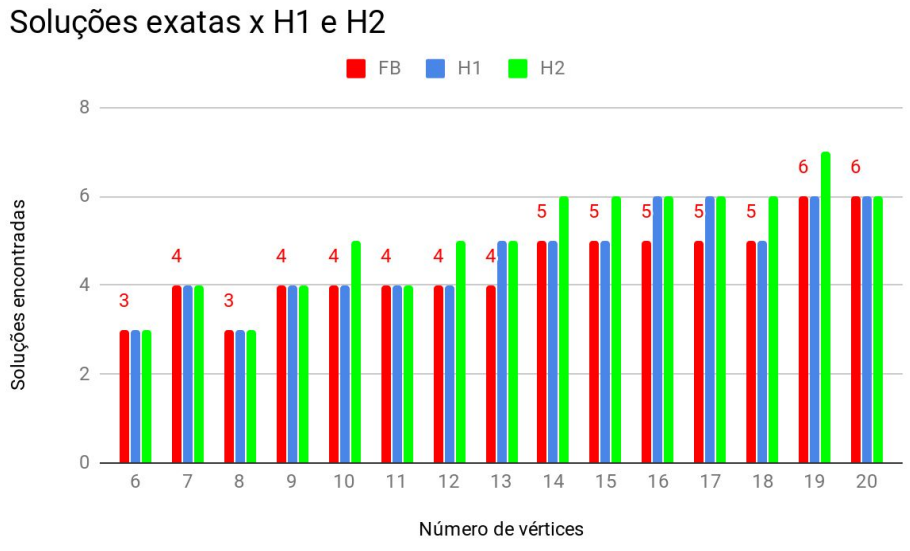
4.2. Análise empírica das soluções encontradas pelas heurísticas

GRÁFICO 4. Solução encontrada em razão da estratégia de algoritmo usado variando a quantidade de vértices.



O gráfico 4 apresenta as soluções geradas pelos algoritmos. Os pontos da linha vermelha são as melhores soluções possíveis, enquanto os quadrados azuis e os círculos verdes são as soluções encontradas pelas heurísticas 1 e 2 respectivamente. Como várias soluções ficaram umas por cima das outras, o gráfico 5 facilita a visualização, sendo as barras vermelhas as soluções ótimas e as outras barras das heurísticas. Empiricamente podemos ver que as heurísticas nem sempre alcançam as soluções exatas, porém, quando isso não ocorre, elas continuam bem próximas, o que as tornam muito úteis, pois o tempo é extremamente inferior ao tempo gasto pelo algoritmo de backtracking.

GRÁFICO 5. Solução encontrada em razão da estratégia de algoritmo usado variando a quantidade de vértices (com uma outra modelagem de gráfico).



4.3. Espaço

Quanto à análise de memória, utilizando o programa valgrind, foram feitos testes com situações adversas, para verificar a alocação de memória. Em todos, toda a memória alocada foi desalocada corretamente.

Exemplo de uma execução do *valgrind*:

```
==8459==
==8459== HEAP SUMMARY:
==8459==     in use at exit: 0 bytes in 0 blocks
==8459==   total heap usage: 18 allocs, 18 frees, 7,768 bytes allocated
==8459==
==8459== All heap blocks were freed -- no leaks are possible
==8459==
==8459== For counts of detected and suppressed errors, rerun with: -v
==8459== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

5. Conclusão

Foi possível observar, durante a realização do trabalho, o comportamento de um problema NP-completo e a eficiência de heurísticas para resolver tais problemas, comparado a eficiência de um algoritmo que garanta sempre a solução ótima e que é implementado somente em tempo exponencial.

Também foi notável que um algoritmo exato para resolver um problema NP-completo, é útil somente quando se trata de entradas pequenas, pois a medida que se aumenta a entrada o custo computacional cresce esdruxulamente. Podendo gastar milênios para executar o programa com uma entrada consideravelmente pequena.

Por fim, objetivo do trabalho, que consistia em implementar duas heurísticas e um algoritmo exato para solucionar o problema da determinação do número cromático, foi realizado com sucesso.

6. Referências Bibliográficas

Araújo Neto, Alfredo Silveira, and Marcos José Negreiros Gomes. "Problema e algoritmos de coloração em grafos-exatos e heurísticos." Revista de Sistemas e Computação-RSC 4.2 (2015).

Lima, Alane Marie de. "Algoritmos exatos para o problema da coloração de grafos." (2017).

Coloração de grafos. Disponível em: <https://pt.wikipedia.org/wiki/Colora%C3%A7%C3%A3o_de_grafos>

Coloração de grafos e suas aplicações. Disponível em: <https://www.ic.unicamp.br/~atilio/slidesWtisc.pdf>

m Coloring Problem | Backtracking-5. Disponível em: <https://www.geeksforgeeks.org/m-coloring-problem-backtracking-5/>

Dsatur. Disponível em: <https://en.wikipedia.org/wiki/DSatur>