Algoritmos e Estrutura de Dados III (2019-1)

Trabalho Prático 3

Heitor Lourenço Werneck
heitorwerneck@hotmail.com

31 de Maio de 2019

1 Introdução

O trabalho a ser apresentado consiste em desenvolver e avaliar soluções ótimas e heurísticas para o problema de coloração de grafos. Uma coloração de k cores de um grafo $G = (V, E)$, não direcionado, é uma função $c : V \rightarrow \{1, 2, \ldots, k\} | (\forall (u, v) \in E)(c(u) \neq c(v))$. O problema de coloração de grafos consiste em determinar o mínimo de cores necessário para colorir um grafo. O menor número de cores utilizado para colorir um grafo é o número cromático de G que é representado por $\chi(G)$. Determinar o número cromático de um grafo é conhecido por ser um problema NPC, mais especificamente o seguinte problema de decisão: "Dado um grafo G, $\chi(G) \leq k$?". [9, 6]

Coloração de grafos e comumente usada em resolução de problemas de agendamento onde há um conjunto T de trabalhos(todos com o mesmo tempo de realização) que deve ser feito por agentes em alguma maquina. Alguns trabalhos não podem ser feitos no mesmo tempo, porque ele deve ser feito pelo mesmo agente ou pela mesma maquina. Esse problema é basicamente o problema de agendamento em uma escola no qual os agentes são os professores, aulas as maquinas e as disciplinas os trabalhos. [5]

Outros problemas que a coloração de grafos resolve é alocação de registradores, sudoku e coloração de mapa.

Nesse trabalho será apresentado soluções envolvendo um algoritmo *branch and bound* e duas heurísticas, DSatur e Recursive Largest First(RLF).

2 Estruturas de dados

Nesse trabalho existe como objetivo determinar a menor complexidade assintótica de tempo possivel para a solução do problema, para atender a esse parâmetro o grafo é representado como uma matriz de adjacência. Como o grafo desse problema é não ponderado então a estrutura do grafo foi definida como:

```
typedef bool weightType;
struct graph{
  int v_qnt;// vertexes
  weightType **adj;// table of edges
};
```

ou seja o grafo será representado como:

$$\begin{bmatrix} v_{11} & v_{12} & \cdots & v_{1j} \\ v_{21} & v_{22} & \cdots & v_{2j} \\ \vdots & \vdots & \ddots & \vdots \\ v_{i1} & v_{i2} & \cdots & v_{ij} \end{bmatrix} \tag{1}$$

Com $v_{ij} \in \{0(\text{Não tem aresta}), 1(\text{Existe aresta})\}$ para $i = 1, ..., |V| \wedge j = 1, ..., |V|$.

Para também ser possível a ordenação do grafo por algum fator sem fazer isomorfismo diretamente no grafo e sim criar uma função que possibilite tal coisa e preservar os nomes dos vértices foi utilizado um tipo de dado que atrela um valor, esse valor foi especificado como grau, ao vértice e tem um campo que guarda seu rótulo fazendo assim o rótulo ser independente do indice de posição do vetor.

```
typedef struct vertex{
  int label,degree;
}vertex;
```

3 Solução

A primeira solução a ser abordada será a de força bruta com *branch and bound*, esta solução para o problema dará o número cromático do grafo.

3.1 Branch and bound

A solução força bruta com *branch and bound* irá passar pelas possibilidades de coloração evitando buscar soluções onde não será achado uma solução melhor do que a que já se obteve. No pior caso todas possibilidades serão enumeradas se não houver nenhum corte de soluções. O corte se da comparando uma variável que guarda o menor número de cores achados até então, para uma solução, com uma variável que guarda o número de cores usados até então na solução que está sendo construída.

Visto que os algoritmos gulosos dão soluções validas e um limite superior para o número de cores o algoritmo *branch and bound* então pode ser utilizado em conjunto com os algoritmos gulosos delimitando um limite de busca, então se eles derem soluções boas muitos cortes irão ocorrer. Seus algoritmos tem complexidade polinomial como poderá ser visto na seção de análise de complexidade, então são validos pois não irão aumentar a complexidade assintótica, e suas complexidades assintóticas são polinomiais e se poder diminuir o tempo do algoritmo não polinomial ja é algo bom.

No algoritmo 1 logo após achar um número máximo de cores é chamado o algoritmo que faz o processo de busca exaustiva com o número máximo de cores para ser feito os cortes.

Algoritmo 1 Branch and bound.

Input: G **Output:** $\chi\{G\}$

1: **procedure** COLORINGBRANCHNBOUND
2: $\quad maxColor \leftarrow minimum(coloringDSatur(G), coloringRLF(G))$
3: $\quad coloringBranchnBoundR(g, colors, colorsCount, maxColor, 0, 0)$
4: $\quad$ **return** $maxColor$

O algoritmo de *branch and bound* (algoritmo 2) começa verificando se a quantidade de cores utilizadas é maior ou igual ao número minimo de cores utilizado, se for então não é necessário continuar na construção dessa solução pois não dará uma solução melhor que a já obtida logo retorna o número minimo de cores já obtido. Na linha 4 começa a tentativa de colorir o vértice de

$G.|V|$(quantidade de vértices de G) cores, primeiro checa se a cor é valida na linha 5 se for então colore o vértice(linha 6).

Depois de colorir o vértice ainda é necessário manter a propriedade da variável **colorsUsed** que é a de guardar a quantidade de cores utilizadas, basicamente será suposto que em uma solução a maior cor será a quantidade de cores diferentes utilizadas, claro que esse fato elimina algumas outras soluções e também pode dar uma informação errada em uma sequencia de cores do tipo $(1, 2, 7)$, ou seja na suposição considerada o número de cores diferentes nessa 3-tupla é 7, oque é errado porém sempre há uma solução ótima com essa propriedade, ou seja sempre é possível obter uma solução ótima mesmo com essa suposição feita. Para não ter esse tipo de inconveniência poderia ser calculado o número de cores diferentes em cada chamada porém isso acrescentaria na complexidade de tempo do algoritmo.

Após a manutenção da variável **colorsUsed** chega a parte de recursão do algoritmo, se chegou no ultimo vértice então retorna o número de cores usadas, se não chegou no ultimo vértice 9 chama a função recursivamente para o próximo vértice, isso irá retornar a quantidade de cores usadas, se a quantidade for menor que a quantidade já obtida então atribui a quantidade de cores mínima.

Algoritmo 2 Branch and bound recursão.

Input: G,colors :: refToInteger,colorsCount :: refToInteger,maxColor :: refToInteger,vertex,colorsUsed **Output:** $\chi\{G\}$

```
1:  procedure coloringBranchnBoundR
2:  |  if *maxColor ≤ colorsUsed then
3:  |  |  return *maxColor
4:  |  for c = 1 to G.|V| do
5:  |  |  if adjNotSameColor(G, colors, c, vertex) then
6:  |  |  |  colors[vertex] = c
7:  |  |  |  if c > colorsUsed then
8:  |  |  |  |  colorsUsed = c
9:  |  |  |  if vertex + 1 < G.|V| then
10: |  |  |  |  result = coloringBranchnBoundR(G, colors, colorsCount, maxColor, vertex + 1, colorsUsed)
11: |  |  |  |  if result ≠ 0 ∧ result < *maxColor then
12: |  |  |  |  |  *maxColor = result
13: |  |  |  else
14: |  |  |  |  return colorsUsed
15: |  return *maxColor
```

3.2 DSatur

A primeira heurística que foi implementada é o algoritmo DSatur que foi originalmente proposto por Brélaz (1979) [8]. A ideia do algoritmo de qual vértice colorir é baseada primeiramente no grau de saturação dos vértices (O grau de saturação de um vértice é o número de cores diferentes nos vértices adjacentes ao mesmo) e seguidamente do grau para ser possível escolher um vértice entre os vértices com maior saturação.

Um fato importante sobre a coloração de grafo é que existe um limite superior bem definido de que $\chi\{G\} \leq \Delta(G) + 1$, onde $\Delta(G)$ é o grau maximo do grafo G.

No algoritmo 3 é possível ver o funcionamento do algoritmo, primeiro descolore todos vértices e zera a saturação consequentemente. Após isso é calculado o grau de todos vértices que é feito a partir de uma passagem pela matriz de adjacência. Essa matriz de adjacência é ordenada com o método da bolha e depois o vértice com maior grau é colorido. Com o vértice colorido a saturação também deve ser atualizada então o próximo passo é esse, o algoritmo descreve bem esse passo, porém é importante notar o fato que a saturação é feita com base nos índices ordenados através do uso da seguinte expressão para checagem $vertexList[<\textbf{vértice}>].label$, essa expressão é uma

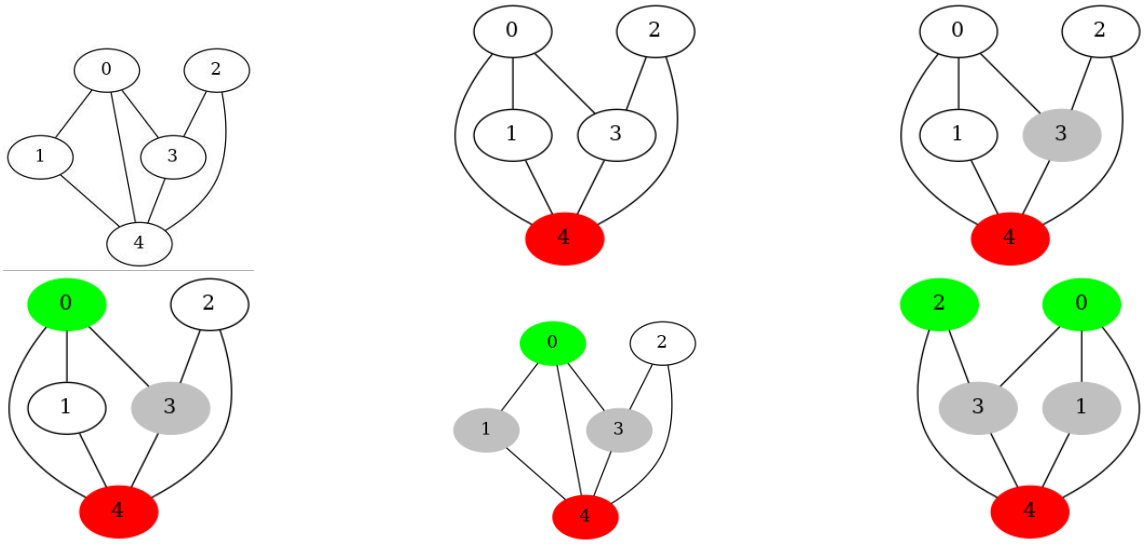


Figura 1: DSatur ilustração da execução.

função bijetora $f : V \to V^{'}$, e depois $saturation[i]$. A consequência do vetor $saturation$ usar os índices ordenados é que na busca pela maior saturação, no caso de saturações iguais, somente percorrendo o vetor e não parando de pegar uma saturação maior ou igual a maior atual a maior final será a com maior grau das com maior saturação.

Após estes passos começa o processo de construção da solução, esse processo irá passar por todos vértices (linha 11), o primeiro passo é pegar a maior saturação esse é um processo trivial de navegação pelo vetor. O passo de colorir basta criar um vetor que diz se uma cor k existe ou não nos vértices adjacentes e depois achar uma cor que não tenha sido colorida, com a coloração do vértice é necessário atualizar a saturação dos adjacentes a ele com em um laço. A remoção do vértice se da com uma troca do rótulo do vértice para um que sinaliza remoção.

Algoritmo 3 DSatur.

Input: G **Output:** número de cores

```
1:  procedure COLORINGDSATUR
2:      for i = 0 to G.|V| − 1 do
3:          saturation[i] ← 0
4:          colors[i] ← 0
5:      vertexList ← Calcula o grau de todos vértices também guardando seu rotulo original
6:      BubbleSort(vertexList)
7:      colors[vertexList[G.|V| − 1].label] = 1                ▷ colore vértice com maior grau
8:      for i = 0 to G.|V| − 1 do                              ▷ Atualiza saturação
9:          if i ≠ vertexList[G.|V| − 1].label ∧ G.adj[vertexList[G.|V| − 1].label][vertexList[i].label] then
10:             saturation[i] ← saturation[i] + 1
11:     while Não leu todos vértices do
12:         vertexLabel ← maxSaturation(saturation)            ▷ Pega maior saturação O(G.|V|)
13:         Colore o vértice com uma cor diferente dos vértices adjacentes
14:         Atualiza saturação dos vértices adjacentes a o normalVertexLabel
15:         Remove o vértice de saturation
```

Algumas constatações sobre o algoritmo DSatur é que ele é exato para grafo bipartido, ciclos e roda. [8] A execução do algoritmo é ilustrada na figura 1.

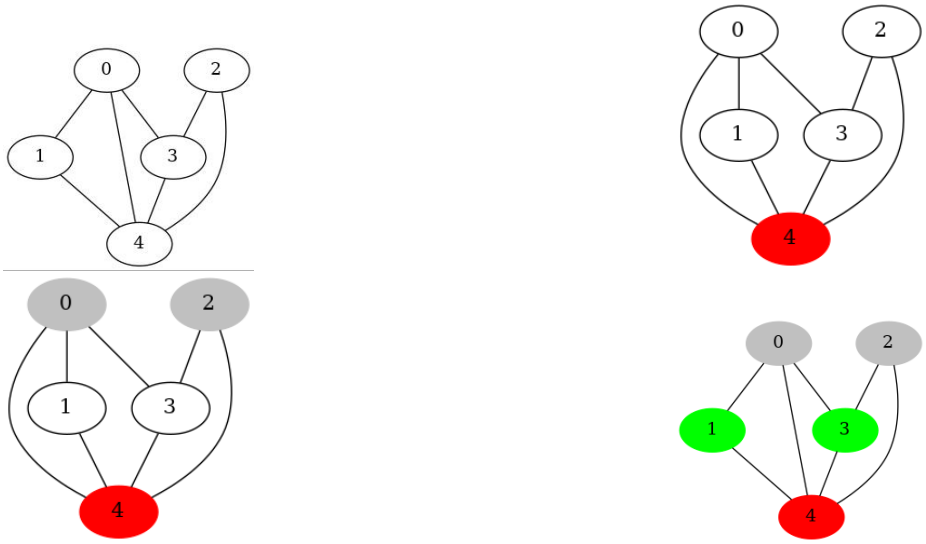


Figura 2: RLF ilustração da execução.

3.3 RLF

O algoritmo Recursive Large First(RLF) segue um padrão guloso de construção diferente do DSatur. O RLF foi originalmente feito por Leighton(1979). O algoritmo foca em colorir de uma cor de cada vez, diferente do DSatur que colore um vértice de cada vez. O algoritmo usa uma heurística para achar um vértice independente e pintar todos vértices desse conjunto independente da mesma cor, após serem coloridos o conjunto independente é removido do grafo. A heurística utilizada para achar o conjunto independente é pegar o vértice de maior grau e depois achar os vértices que não são adjacentes a esse e não são adjacentes entre si.(linha 13)

O algoritmo 4 mostra o funcionamento.

Algoritmo 4 RLF.

Input: G **Output:** número de cores

```
1:  procedure COLORINGRLF
2:      for i = 0 to G.|V| − 1 do
3:          colors[i] ← 0
4:      vertexList ← Calcula o grau de todos vértices também guardando seu rotulo original
5:      remainingVertex ← G.|V|, c ← 1
6:      while remainingVertex > 0 do
7:          ChosenVertex ← higherDegree(vertexList)
8:          U1 ← Todos vértices não adjacentes a ChosenVertex(incluindo ele mesmo)
9:          U2 ← vertexList − U1
10:         U1.grau ← Grau de todos os vértices de U1 em relação aos vértices de U2
11:         bubbleSort(U1)
12:         for v in U_1 do
13:             Checa se algum vértice de U1 é adjacente a v e o transfere de U1 para U2
14:             colors[v.label] ← c; vertexList ← vertexList − v; remainingVertex ← remainingVertex − 1
15:             Decrementa o grau dos vértices adjacentes ao vértice removido
16:         c ← c + 1
17:     return c − 1
```

Um fato sobre o algoritmo RLF é que ele é exato para os mesmos casos que o algoritmo DSatur. A execução do algoritmo é ilustrada na figura 2.

4 Análise de complexidade

Antes da analise de complexidade é necessário definir alguns termos que darão a complexidade do algoritmo.

$$\underbrace{V}_{\text{quantidade de vértices}}, \quad \underbrace{k}_{\text{quantidade de cores}} \in \mathbb{N} \tag{2}$$

4.1 Complexidade de tempo

4.1.1 DSatur

O algoritmo 3 faz uso de diversos algoritmos auxiliares que necessitam de análise assintótica. De acordo com o funcionamento de cada componente a complexidade do algoritmo é a seguinte:

$$\begin{aligned} coloringDSatur(V) \in O(max(V, CalculaGrau, \\ BubbleSort, AtualizaSaturacao, V \cdot (maxSaturation + Pinta + AtualizaSaturacao))) \\ CalculaGrau \in O(V^2); BubbleSort \in O(V^2); \\ AtualizaSaturacao \in O(V); maxSaturation \in O(V); Pinta \in O(V) \\ coloringDSatur(V) \in O(max(V, V^2, V^2, V \cdot (V + V + V)) \\ \therefore coloringDSatur(V) \in O(V^2) \end{aligned} \tag{3}$$

4.1.2 RLF

O algoritmo *coloringRLF* faz uso de diversos algoritmos auxiliares que necessitam de análise assintótica, alguns iguais ao DSatur.

O laço interno ao "while"do algoritmo tem seus elementos executados V vezes somente ao invés de V^2, isso pois o conjunto "anda"junto com os vértices remanescentes, logo as funções internas do laço da linha 12 são executados $O(V)$ vezes.

De acordo com o funcionamento de cada componente a complexidade do algoritmo é a seguinte:

$$\begin{aligned} coloringRLF(V) \in O(max(V, CalculaGrau, \\ V \cdot higherDegree, V \cdot ConstrucaoU1U2, V \cdot GrauU1, \\ V \cdot BubbleSort, V \cdot AdjacenteVerifica, V \cdot PintaERemove, \\ V \cdot AtualizaGrau)) \\ CalculaGrau \in O(V^2); higherDegree \in O(V); ConstrucaoU1U2 \in O(V); \\ GrauU1 \in O(V); AdjacenteVerifica \in O(V); \\ PintaERemove \in O(1); AtualizaGrau \in O(V) \\ coloringRLF(V) \in O(max(V, V^2, V, V^2, V^2, V^3, V^2, V, V^2)) \therefore coloringRLF(V) \in O(V^3) \end{aligned} \tag{4}$$

4.1.3 Branch and bound

Uma analise mais intuitiva do algoritmo *branch and bound* é sobre a enumeração das possibilidades, no pior caso do algoritmo ele irá enumerar todas possibilidades de cores. A equação de recorrência do algoritmo *branch and bound* é a seguinte:

$$t(V) = \begin{cases} k \cdot t(V-1) & \text{if } V > 1 \\ k & \text{if } V = 1 \end{cases} \tag{5}$$

$$t(V) = k \cdot t(V-1) = k^2 \cdot t(V-2) = k^{V-1} \cdot t(V-(V-1)) = k^{V-1} \cdot t(1) = k^V \tag{6}$$

Como o k no algoritmo é sempre a quantidade de arestas então a complexidade será finalmente:

$$k = O(V); t(V) = k^V; k = V \implies t(V) = V^V \therefore coloringBranchnBoundR \in O(V^V) \quad (7)$$

Já a função principal utiliza as heurísticas para ser feito cortes. A complexidade é a mesma complexidade porém é importante incrementar explicitamente essa situação

$$\begin{aligned} coloringBranchnBound(V) &\in O(V^V + coloringDSatur + coloringRLF) \\ O(V^V + V^2 + V^3) &\implies coloringBranchnBound(V) \in O(V^V) \end{aligned} \quad (8)$$

A complexidade encontrada é bem ruim no pior caso porém existem algoritmos exatos com complexidade assintótica menor. O melhor algoritmo conhecido para resolver esse problema é o que utiliza o principio de inclusão-exclusão e a transformada Z [4] esse algoritmo tem complexidade $O(n \cdot 2^n)$.

4.2 Complexidade de espaço

Antes de se fazer a análise de complexidade de espaço é preciso evidenciar que os algoritmos que possuem em sua lista de parâmetros uma referência para um vetor não apresentará complexidade de espaço relacionada com o espaço ocupado por esse valor referenciado(Ex. $O(n^c), c \in \mathbb{N}$) pois o parâmetro guarda uma constante(referência para o vetor) ou seja a complexidade de espaço é constante.[2]

4.2.1 DSatur

O algoritmo DSatur faz uso de diversas listas para realizar as operações necessarias. Existe uma lista de cores dos vertices, saturação e outros. As funções que o algoritmo chama são todas de complexidade $O(1)$. Logo como faz uso de listas de uma dimensão somente então a complexidade do algoritmo 3 é $O(V)$ pois o grafo é passado com complexidade constante para a função por ser uma referencia. Porém no contexto geral desse algoritmo no programa principal sua complexidade pode ser vista como $O(V^2)$.

4.2.2 RLF

O algoritmo RLF faz uso também de diversas listas como o DSatur, todas elas unidimensionais e todas função auxiliares chamadas tem complexidade de espaço $O(1)$ então a complexidade do algoritmo 4 é $O(V)$. Porém assim como o DSatur em um contexto geral sua complexidade pode ser vista como $O(V^2)$.

4.2.3 Branch and bound

A complexidade de espaço do algoritmo 2 está atrelada a sua recursão pois existe um custo de guardar o contexto atual a cada chamada, na lista de parâmetros todas as variaveis tem custo $O(1)$ e o algoritmo no máximo guarda os parâmetros V vezes devido a seu comportamento recursivo, logo a complexidade do algoritmo é $O(V)$.

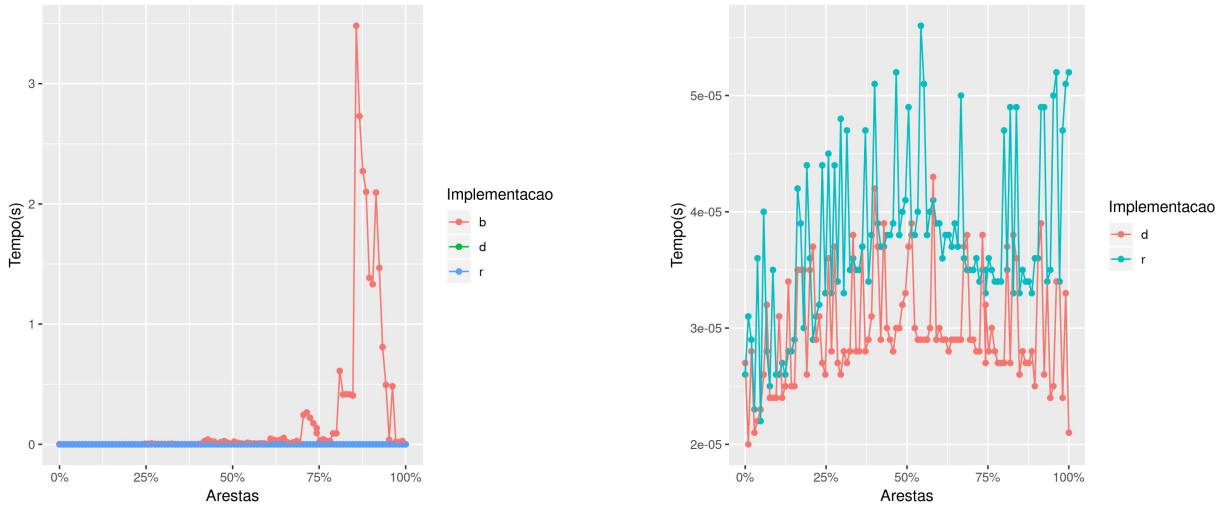
4.3 Análise geral

A tabela 1 mostra todas complexidades obtidas, com ela pode-se ver que o melhor algoritmo é o DSatur em questão de complexidade assintótica de tempo e o pior é o *branch and bound*. No geral também pode-se ver que todos algoritmos tem a mesma complexidade de espaço logo nenhum deles tem alguma vantagem sobre o outro nesse quesito.

O programa principal necessita de alocar a matriz de adjacência que representa o grafo, isso tem custo $O(V^2)$.

Complexidade	Tempo	Espaço
DSatur	$O(V^2)$	$O(V)$
RLF	$O(V^3)$	$O(V)$
Branch and bound	$O(V^V)$	$O(V)$
Principal	$O(V^V + V^2 + V^3)$	$O(V^2)$

Tabela 1: Complexidades.



(a) Branch and bound e outros.

(b) DSatur e RLF.

Figura 3: Crescimento de acordo com variação de arestas com disposição aleatória(15 vértices).

5 Resultados

A maquina utilizada para os experimentos possui as seguintes especificações: Intel(R) Core(TM) i3-4005U CPU @ 1.70GHz e 4GiB de memória RAM.

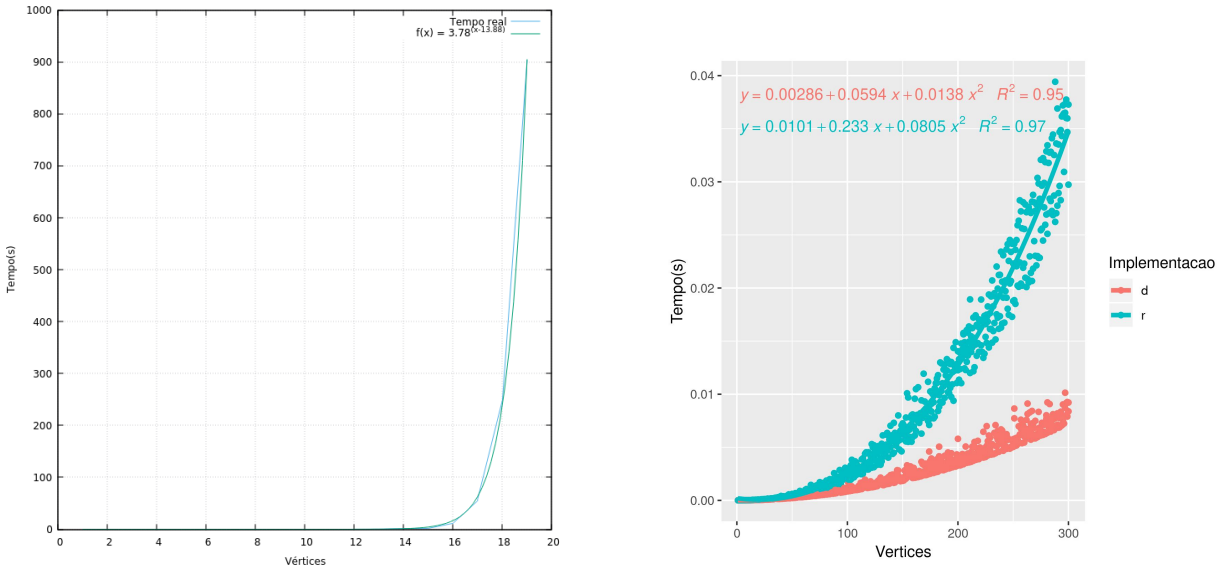
5.1 Tempo

Antes da análise de resultados é importante refletir sobre quais fatores influenciam no tempo de execução. Nesse caso há os vértices e arestas. Primeiro então é necessario observar como o tempo se comporta com a variação desses parâmetros. È fato que também a maneira que as arestas estão dispostas afetam no tempo porém a analise desse fator é muito complexa e não será abordada.

Na figura 3a o algoritmo força bruta parece ter o maior tempo de execução quando o número de arestas esta proximo de 87% da capacidade total. Já o DSatur e RLF, de acordo com a figura 3b, aumentam seu tempo quando o número de arestas se aproxima de 55%.

Agora para verificar se as complexidades obtidas que estão na tabela 1 são corretas é preciso gerar entradas que exploram o pior caso para obter o pior tempo. Com base no pior tipo de conteúdo em uma entrada, que acabou de ser descoberto, de cada algoritmo, basta variar o número de vertices.

A regressão feita no algoritmo *branch and bound*, como pode ser visto na figura 4a, é melhor que a complexidade obtida na tabela 1, isso pois o valor 3.78 da função $f(x) = 3.78^{x-13.88}$ é uma constante e mesmo assim a regressão funcionou. Isso mostra que o *branch and bound* é um bom algoritmo dentre algoritmos exatos, pois um tentativa e erro não cortaria tantas soluções como o branch and bound é capaz. O maior valor obtido foi com 19 vértices e um tempo de 902.958661 segundos. Para uma entrada 10 vezes maior o tempo necessário será de $5.10335e101$ segundos.



(a) Branch and bound.

(b) DSatur e RLF.

Figura 4: Complexidade de tempo, regressão.

Na figura 4b há a regressão dos outros dois algoritmos. O algoritmo DSatur foi comprovado sua complexidade $O(V^2)$. Já o algoritmo RLF mostrou uma complexidade assintótica melhor que a esperada de $O(V^3)$ porém como observado em [7] o algoritmo se comporta como $O(V^2)$ para colorir grafos que $k \times E \approx V^2$, onde E é o numero de arestas.

5.2 Espaço

Para provar a complexidade de espaço do algoritmo principal basta monitorar as alocações feitas pelo algoritmo para cada tamanho de entrada. O monitoramento será feito somente da memória *heap* [3] que é a parte da memória onde a alocação dinâmica é feita e o espaço pode-se variar dinamicamente permitindo-se assim obter uma função de complexidade correta.

Pela figura 5 o espaço utilizado pelos 3 algoritmos foi comprovado e como dito os 3 algoritmos tem regressões polinomiais parecidas que provém do algoritmo principal na alocação de memória para o grafo de entrada(matriz de adjacência). Isso fica óbvio vendo que o fator do termo de ordem 2 é igual para todos algoritmos e é essa uma parte da quantidade de memória usada pelo programa principal.

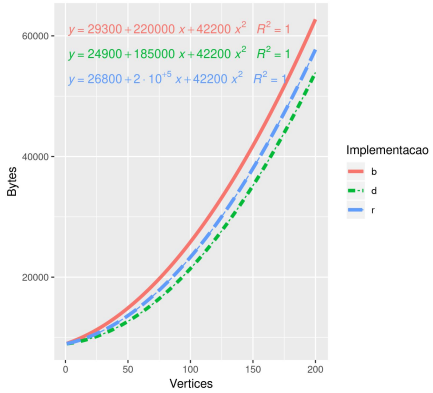


Figura 5: Espaço usado pelos algoritmos.

(a) DSatur. **(b)** RLF.

Figura 6: Taxa de acerto das heurísticas.

(a) DSatur. **(b)** RLF.

Figura 7: Taxa de acerto das heurísticas, diagrama de caixa.

5.3 Estrategia gulosa

A taxa de acerto das heurísticas é mostrado na figura 6, com uma quantidade pequena de vértices o algoritmo achou a solução ótima nesses gráficos porém com o aumento de vértices a solução piora ou pelo menos a incerteza aumenta. O algoritmo RLF apresenta uma taxa de acerto alta, já o algoritmo DSatur dá soluções que tem 50% de erro(figura 7) nesses casos aleatorizados. Porém é importante notar que a taxa de acerto pode variar com o tipo do problema(disposição das arestas).

Na tabela 2 é possível ver o comportamento do erro dos algoritmos em problemas reais. O algoritmo DSatur se destacou em relação a taxa de acerto que apresenta, porém assim como ele tem o menor erro também possui a pior solução porém na média isso se torna irrelevante devido a seu desempenho ótimo. Mas na média o algoritmo RLF também não fica tão atrás do DSatur. Porém é importante salientar que o algoritmo DSatur melhora muito sendo aplicado em problemas reais, ou pelo menos em problemas de certo tipo.

Um outro fator que será comprovado empiricamente são os casos de soluções ótimas dos algoritmos.

6 Conclusão

Com esse trabalho foi possível observar a intratabilidade de um problema NPC na prática, com as análises de gráficos e regressões dos algoritmos a solução exata se mostrou inaplicável para problemas sequer de tamanhos médios. Porém isso tudo depende da quantidade de arestas no grafo, a variação dos algoritmos em relação a quantidade de arestas também foi analisada, em

Base	RLF cores	DSatur cores	$\chi\{G\}$	RLF Erro	DSatur Erro
homer	14	13	13	0.92857143	1
le450_25a	30	25	25	0.83333333	1
miles500	21	20	20	0.95238095	1
miles750	35	31	31	0.88571429	1
mulsol.i.1	49	49	49	1	1
mulsol.i.2	31	31	31	1	1
queen11_11	17	18	11	0.64705882	0.61111111
queen13_13	21	23	13	0.61904762	0.56521739
Média				0.85826331	0.89704106

Tabela 2: Eficiência da heurística em problemas variados. [1]

Tipo do grafo	V	E	$\chi\{G\}$	solução DSatur	solução RLF
Roda	10	18	4	4	4
Bipartido	10	9	2	2	2
Ciclo	15	15	3	3	3
Ciclo	4	4	2	2	2
Roda	4	6	4	4	4
Bipartido	5	7	2	2	2

Tabela 3: Soluções em casos ótimos.

certas quantidades de arestas pode-se esperar que o algoritmo *branch and bound* seja mais rápido que o usual.

As heurísticas propostas demonstraram grande taxa de acerto, o algoritmo RLF com grande acerto para entradas aleatorizadas e o algoritmo DSatur obtendo a solução ótima muitas vezes em problemas reais.

O algoritmo *branch and bound* foi atribuído uma complexidade assintótica $O(V^V)$, na prática a regressão obteve uma complexidade menor o que comprova que o algoritmo *branch and bound* é melhor que as soluções triviais de algoritmos exatos. O algoritmo RLF foi também melhor na prática assim como o *branch and bound*, se comportando com uma complexidade de $O(V^2)$ enquanto tinha sido obtido sua complexidade $O(V^3)$. O algoritmo DSatur foi exatamente como esperado, a complexidade obtida foi comprovada e ele foi o algoritmo mais rápido obtido com complexidade assintótica $O(V^2)$.

A complexidade de espaço dos algoritmos foram comprovadas empiricamente e seus fatores são extremamente iguais, logo nenhum algoritmo se sobressai expressivamente nesse quesito. Então em uma abordagem prática para ver qual algoritmo aplicar iria-se estudar a quantidade de arestas em relação ao total possível e ver se há a possibilidade de aplicar o força bruta, quando não estiver perto de uma porcentagem de arestas de 87%. Se o força bruta for inviável a escolha da heurística se dará simplesmente por qual área cada uma é melhor, o observado nos resultados é que o RLF será aplicado para problemas com natureza aleatória e o DSatur para problemas com alguma ordem.

Referências

[1] Graph coloring instances. `https://mat.gsia.cmu.edu/COLOR/instances.html#XXREG`. Accessed: 2019-05-29.

[2] Space complexity. `https://www.cs.northwestern.edu/academics/courses/311/html/space-complexity.html`. Accessed: 2019-04-23.

[3] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2018.

[4] Andreas Björklund, Thore Husfeldt, and Mikko Koivisto. Set partitioning via inclusion-exclusion. *SIAM Journal on Computing*, 39(2):546–563, 2009.

[5] Daniel Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, 1979.

[6] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Complexity of Computer Computations. Springer US, 1972.

[7] F.T. Leighton. A graph coloring algorithm for large scheduling problems. *Journal of Research of the National Bureau of Standards*, 84(6):489, Nov 1979.

[8] R.M.R. Lewis. *A Guide to Graph Colouring*. Springer International Publishing, 2016.

[9] Larry Stockmeyer. Planar 3-colorability is polynomial complete. *ACM SIGACT News*, 5(3):19–25, 1973.