# **Análise de Algoritmos: Coloração de Grafos**

**Víctor Hugo Gabriel dos Reis**

Faculdade de Computação - Universidade Federal de Uberlândia (UFU)

Caixa Postal 593 -38.408-100 - Uberlândia - MG, 38400-902

[viquitorreis@ufu.br](mailto:viquitorreis@ufu.br)

***Abstract*.** *This work presents the implementation and analysis of three graph coloring algorithms: an exact algorithm based on backtracking, a simple greedy algorithm, and a greedy algorithm with vertex degree ordering. The study evaluates execution time, success rate, and the number of colors required for graphs of different sizes and densities. To ensure fair comparison, a graph generator was developed, and multiple tests were executed with sparse, medium, and dense graphs. The results indicate the contexts in which greedy algorithms fail compared to the exact method, highlighting the trade-off between performance and solution quality.*

***Resumo.*** *Este trabalho apresenta a implementação e análise de três algoritmos de coloração de grafos: um algoritmo exato baseado em backtracking, um algoritmo guloso simples e um algoritmo guloso com ordenação por grau dos vértices. O estudo avalia tempo de execução, taxa de sucesso e número de cores necessárias em grafos de diferentes tamanhos e densidades. Para garantir uma comparação justa, foi desenvolvido um gerador de grafos e realizados múltiplos testes com grafos esparsos, médios e densos. Os resultados indicam em quais contextos os algoritmos gulosos falham em relação ao método exato, evidenciando o equilíbrio entre desempenho e qualidade da solução.*

**1. Informações Gerais**

O presente trabalho tem como objetivo a análise de algoritmos de coloração de grafos, um problema clássico de Teoria dos Grafos que consiste em atribuir cores aos vértices de um grafo de modo que vértices adjacentes não recebam a mesma cor. Esse problema possui aplicações em diversas áreas, como alocação de recursos, escalonamento de tarefas e otimização de processos.

Foram implementadas três abordagens distintas: um algoritmo exato baseado em backtracking, um algoritmo guloso simples e um algoritmo guloso com ordenação pelo grau dos vértices. A comparação entre eles é realizada considerando tempo de execução, número mínimo de cores utilizadas e taxa de sucesso.

A metodologia empregada inclui a geração automática de grafos com diferentes números de vértices e densidades (esparsos, médios e densos), além da execução repetida de experimentos para obtenção de valores médios e redução dos efeitos de aleatoriedade. A implementação foi realizada em linguagem Go, e os resultados são apresentados em formato de tabelas e gráficos.

**2. Informações Gerais**

A implementação do trabalho foi desenvolvida em linguagem Go (Golang), escolhida por sua eficiência, simplicidade e boas práticas na manipulação de estruturas de dados. A seguir, é descrito as principais estruturas da implementação.

**2.1. Informações Gerais**

A estrutura de dados escolhida para representar o grafo foi baseada em lista de adjacência, onde cada vértice mantém um mapa de suas arestas. Essa abordagem fornece complexidade O(1) para verificação de adjacência e O(grau) para iteração sobre vizinhos, o que se mostra eficiente em termos de memória, de simples implementação, e adequada para manipulação de grafos esparsos ou densos:

type Graph struct {

Vertices map[int]\*Vertex

Len int // Número de vértices

EdgesLen int // Número de arestas

}

type Vertex struct {

val int

edges map[int]\*Edge

}

type Edge struct {

weight int

dest \*Vertex // Ponteiro direto para o vértice destino

}

Na implementação apresentada, para garantir que o grafo seja não-direcionado, cada aresta é armazenada em ambos os vértices

**2.2. Gerador de Grafos**

Para garantir testes controláveis, foi implementado um gerador que produz grafos com densidade específica. O gerador retorna a interface do grafo, que permite a chamada de métodos a partir do grafo gerado.

func GenerateGraph(nVertices int, density float32) IGraph

O gerador:

1. Cria vértices numerados de 1 a n.
2. Calcula o número alvo de arestas, permitindo o limite máximo para um grafo não-direcionado (nVertices (nVertices - 1) / 2).
3. Adiciona arestas aleatórias até atingir a densidade desejada, evitando duplicatas e loops.
4. Permite a criação de grafos pouco, médios ou altamente densos, ajustando o parâmetro *density* (entre 0 e 1).

func GenerateGraph(nVertices int, density float32) IGraph {

graph := NewGraph()

for n := range nVertices {

graph.AddVertex(n + 1)

}

maxEdges := nVertices \* (nVertices - 1) / 2

targetEdges := int(float32(maxEdges) \* density)

edgesAdded := 0

for edgesAdded < targetEdges {

i := rand.Intn(nVertices) + 1

j := rand.Intn(nVertices) + 1

if i != j && !graph.HasEdge(i, j) {

weight := rand.Intn(10) + 1

if graph.AddEdge(i, j, weight) {

edgesAdded++

}

}

}

return graph

}

**2.3. Algoritmos de Coloração**

Todos os algoritmos seguem a assinatura *bool colore(g, k),* retornando verdadeiro se conseguiram colorir o grafo G com até k cores.

**2.3.1. Algoritmo Exato (Backtracking)**

O algoritmo exato implementado utiliza backtracking para encontrar uma coloração válida com até k cores. A estratégia consiste em:

1. Tenta atribuir uma cor a cada vértice, verificando se não há conflito com os vizinhos.
2. Caso não seja possível atribuir uma cor válida a um vértice, o algoritmo retrocede a decisão (backtrack) e tenta uma cor diferente nos vértices anteriores.
3. O processo se repete até encontrar uma solução válida ou esgotar todas as combinações.

func (g \*Graph) BacktrackColoring(n int) bool {

coloredMap, verticesList := g.makeColorMapAndVerticesList()

if g.doColor(verticesList, 0, n, coloredMap) {

g.printColors(coloredMap)

return true

} else {

fmt.Printf("nenhuma solução com %d cores\n", n)

return false

}

}

**Complexidade:** O algoritmo possui complexidade exponencial O(K^V), onde V é o número de vértices e k o número máximo de cores.

**Comportamento esperado:** Sempre encontra a solução ótima, mas torna-se inviável para grafos grandes e/ou de grande densidade devido ao crescimento exponencial do espaço de busca.

**2.3.2. Algoritmo Guloso Simples**

O algoritmo segue uma abordagem heurística, processando os vértices em ordem arbitrária e atribuindo a cada vértice a menor cor disponível que não seja usada por seus vizinhos. A estratégia consiste em:

1. Escolher uma ordenação dos vértices (aleatória, por valor, ou outra heurística simples). Para o estudo, foi adotado a ordenação dos vértices em ordem crescente.
2. Para cada vértice, identificar as cores já utilizadas pelos vizinhos.
3. Atribuir a menor cor disponível que não entre em conflito com as demais.
4. Repetir até todos os vértices estarem coloridos ou não ser possível encontrar uma cor válida.

func (g \*Graph) GreedyColoring(n int) bool {

coloredMap, verticesList := g.makeColorMapAndVerticesList()

sort.Ints(verticesList)

for \_, v := range verticesList {

availableColor := g.availableColor(n, v, coloredMap)

if availableColor == -1 {

return false

}

(\*coloredMap)[v] = availableColor

}

g.printColors(coloredMap)

return true

}

**Complexidade:** O algoritmo possui complexidade O(V²) no pior dos casos.

**Comportamento esperado:** Rápido e eficiente para grafos grandes, mas não garante a solução ótima. Pode falhar em grafos densos ou com estrutura complexa.

**2.3.3. Algoritmo Guloso com Ordenação por Grau**

Esta variação do algoritmo guloso prioriza a coloração de vértices com maior grau (mais vizinhos) antes dos demais. Essa heurística se mostrou mais eficiente para reduzir conflitos futuros, já que vértices altamente conectados são mais restritivos na escolha de cores.

func (g \*Graph) GreedyColoringByDegree(n int) bool {

coloredMap, verticesList := g.makeColorMapAndVerticesList()

ordered := g.sortByDegree(verticesList)

for \_, v := range ordered {

availableColor := g.availableColor(n, v, coloredMap)

if availableColor == -1 {

return false

}

(\*coloredMap)[v] = availableColor

}

g.printColors(coloredMap)

return true

}

**Complexidade:** Também O(n²), mas geralmente produz soluções melhores que o guloso simples.

**Comportamento esperado:** Mais eficiente que o guloso simples na maioria dos casos, com maior probabilidade de encontrar soluções próximas do ótimo, especialmente em grafos densos.

**3. Resultados**

Foram executados testes com diferentes cenários, variando o número de vértices e a densidade do grafo. Três algoritmos de coloração foram avaliados: Backtracking (Exato), Guloso Simples e Guloso por Grau.

As métricas coletadas foram:

* Tempo de execução de cada algoritmo
* Número mínimo de cores necessário para colorir o grafo sem conflitos

**Tabela 1. A Tabela 1 resume os resultados obtidos.**

| Cenário | Backtracking | Guloso | Guloso por Grau | Cores mínimas (BT/G/GD) |
| --- | --- | --- | --- | --- |
| 5v-0.1d | 9µs | 9µs | 11µs | 2/2/2 |
| 5v-0.5d | 29µs | 19µs | 23µs | 3/3/3 |
| 5v-1.0d | 137µs | 24µs | 37µs | 5/5/5 |
| 10v-0.1d | 37µs | 27µs | 40µs | 2/2/2 |
| 10v-0.5d | 345µs | 46µs | 162µs | 4/5/4 |
| 10v-1.0d | 3.11s | 67µs | 112µs | 10/10/10 |
| 20v-0.1d | 418µs | 88µs | 89µs | 3/3/3 |
| 20v-0.5d | 41.36s | 62µs | 337µs | 6/7/6 |
| 50v-0.1d | 54.39s | 194µs | 550µs | 4/5/4 |

Observações principais:

* O algoritmo Backtracking apresenta crescimento exponencial no tempo de execução à medida que aumenta o número de vértices e a densidade.
* Os algoritmos Guloso e Guloso por Grau têm desempenho significativamente superior em tempo, sendo adequados para quantidades de entradas maiores.
* Em termos de número de cores, o Backtracking encontra sempre a solução ótima. O Guloso por Grau apresenta melhor resultado que o Guloso puro na maioria dos cenários, ficando mais próximo do ótimo.
* Observa-se que, para grafos pequenos (até 10 vértices), a diferença de tempo entre os algoritmos é pouco significativa, mas a partir de 20 vértices com densidade 1.0, o Backtracking passa a ter um crescimento exponencial de tempo.
* No segundo conjunto de testes (com 100 vértices), verificou-se que o tempo do Backtracking cresce de forma impraticável, inviabilizando sua aplicação em grafos grandes.

**4. Conclusões**

O estudo desenvolvido permitiu analisar a aplicação de diferentes algoritmos de coloração de grafos em cenários variados, considerando a geração de grafos com diferentes densidades e tamanhos. A implementação em linguagem Go possibilitou tanto a representação de grafos quanto a aplicação de técnicas clássicas de coloração, como o algoritmo de backtracking e heurísticas aproximativas.

Os resultados mostraram que:

* Para grafos com 20 vértices, utilizando até 20 cores, o algoritmo de backtracking apresentou crescimento exponencial no tempo de execução a partir da densidade 1. Isso confirma a complexidade do problema de coloração exata.
* No segundo conjunto de testes, com 100 vértices, o aumento exponencial do tempo de execução se mostrou ainda mais rapidamente, tornando inviável a execução do backtracking em densidades mais elevadas.
* As heurísticas implementadas, por outro lado, apresentaram desempenho significativamente mais eficiente, mesmo em grafos mais densos, ainda que em alguns casos tenham resultado em soluções não ótimas.

De maneira geral, pode-se concluir que o algoritmo de backtracking é impraticável para instâncias maiores, confirmando seu uso apenas em cenários reduzidos ou como base comparativa. Já as heurísticas de coloração mostraram-se viáveis e escaláveis, oferecendo soluções próximas do ótimo com custo computacional muito menor.

Este trabalho evidencia a importância da escolha do algoritmo conforme as restrições do problema: para aplicações reais em grafos grandes, abordagens heurísticas e aproximativas se mostram indispensáveis.

**5. Apêndice**

O código-fonte desenvolvido neste trabalho encontra-se disponível no seguinte repositório público:

*https://github.com/viquitorreis/ed2/blob/main/graph/coloring/main.go*