

Felipe Soares, Kaiky Gomes, Mateus Ramos

**Projeto e Análise de Algoritmos**

**Procedimento de Busca Aleatória Gulosa Adaptativa**

*para solucionar o problema de coloração de vértices*

13 de Março de 2024

Campos dos Goytacazes

Sumário

[1.0 Manual de uso do programa 2](#_Toc161232905)

[2.0 O problema 2](#_Toc161232906)

[3.0 A metaheurística 3](#_Toc161232907)

[3.1 Características 4](#_Toc161232908)

[4.0 Resolução do problema 4](#_Toc161232909)

[4.1 Adaptando a lógica 4](#_Toc161232910)

[4.2 Parâmetros 4](#_Toc161232911)

[4.3 Leitura dos dados 5](#_Toc161232912)

[4.4 Busca de solução inicial com algoritmo guloso 7](#_Toc161232913)

[4.4.1 Função geraVetorCor 7](#_Toc161232914)

[4.4.2 Função atualizaMelhorResultado 9](#_Toc161232915)

[5.0 Evolução dos resultados 9](#_Toc161232916)

[5.1 Função “buscaLocal” 10](#_Toc161232917)

[5.1.1 Variáveis 10](#_Toc161232918)

[5.1.2 Função “contarCoresDiferentes” 11](#_Toc161232919)

[5.1.2 Escolha de uma cor aleatória 11](#_Toc161232920)

[5.1.3 Verificação de conflitos 12](#_Toc161232921)

[5.1.4 Confirmando resultados 13](#_Toc161232922)

[5.2 Função “perturbação” 13](#_Toc161232923)

[6.0 Fim do código 14](#_Toc161232924)

[7.0 Resultados encontrados 14](#_Toc161232925)

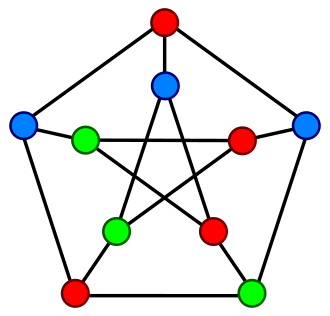
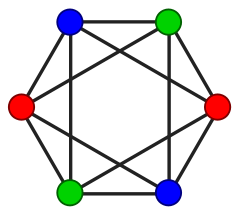
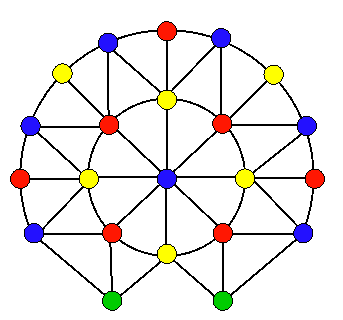
# 1.0 Manual de uso do programa

Para utilizar o programa, basta utilizar um arquivo denominado “grafo.txt” dentro do mesmo diretório do programa e executá-lo.

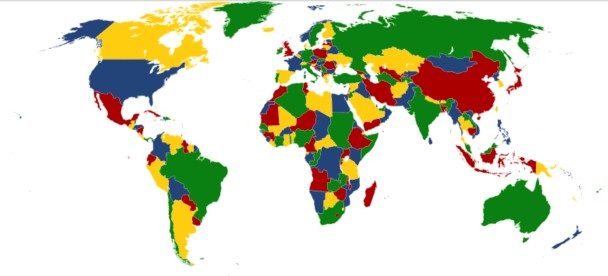
Note que o arquivo mochila.txt deve conter na primeira linha a capacidade da mochila, na segunda linha o número n de itens totais da mochila e em cada uma das próximas n linhas, o valor e peso de cada item.

# 2.0 O problema

O problema de coloração de vértices é um desafio em teoria dos grafos, onde o objetivo é atribuir cores a cada vértice de um grafo de modo que vértices adjacentes não compartilhem a mesma cor. A ideia é minimizar o número de cores utilizadas. Formalmente, dada uma quantidade finita de cores, a tarefa é colorir o grafo de modo que nenhum par de vértices adjacentes tenha a mesma cor.

A resolução eficiente desse problema é importante em várias aplicações práticas, como alocação de recursos em redes de computadores, design de mapas de registro de circuitos e agendamento de exames. O problema de coloração de vértices é um exemplo clássico de um problema NP-difícil, o que significa que não se conhece um algoritmo polinomial para resolvê-lo em tempo razoável, a menos que P seja igual a NP. Portanto, as soluções eficientes para casos específicos ou aproximações são comumente exploradas na prática.



# 3.0 A metaheurística

Agora que sabemos um pouco mais sobre como funciona o problema a ser trabalhado, vamos entender melhor a ferramenta que vamos utilizar para resolver esse problema.

O Greedy Randomized Adaptive Search Procedure (GRASP) é uma poderosa abordagem de otimização que combina estratégias gulosas e aleatórias para enfrentar problemas complexos de otimização combinatória. Projetado para lidar com situações onde encontrar a solução ótima é desafiador devido à natureza computacionalmente difícil do problema, o GRASP se destaca ao equilibrar a exploração do espaço de busca. Sua flexibilidade e adaptabilidade tornam-no uma ferramenta valiosa em diversos domínios, proporcionando uma solução eficaz para problemas práticos que variam desde alocação de recursos até design de redes

## 3.1 Características

O GRASP é uma técnica de otimização heurística que combina uma abordagem gananciosa com aleatoriedade para construir soluções iterativamente. Utiliza heurísticas de construção para formar soluções parciais, introduzindo aleatoriedade para explorar amplamente o espaço de solução.

Após a construção inicial, emprega busca local para aprimorar a solução. Sua adaptabilidade e aplicação abrangente em problemas complexos o tornam eficaz em situações onde soluções ótimas são desafiadoras de serem encontradas.

É composto por:

* **Algoritmo Guloso -** Produz uma solução inicial, com base em um algoritmo guloso usado para a geração de uma solução inicial sub-ótima;
* **Busca Local -** Retorna uma solução melhorada em relação a inicial com base na sua vizinhança;
* **Perturbação -** Modifica a solução corrente guiando a uma solução intermediária, evitando que o algoritmo prenda em ótimos sub-ótimos locais;
* **Critério de Aceitação -** Decide de qual solução a próxima perturbação será aplicada;

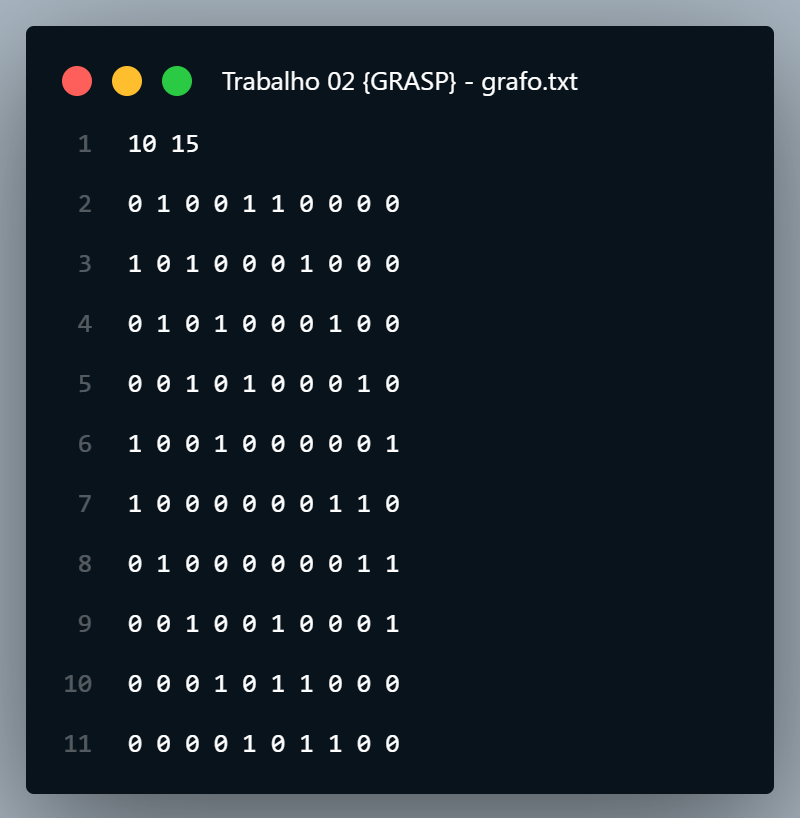
# 4.0 Resolução do problema

## 4.1 Adaptando a lógica

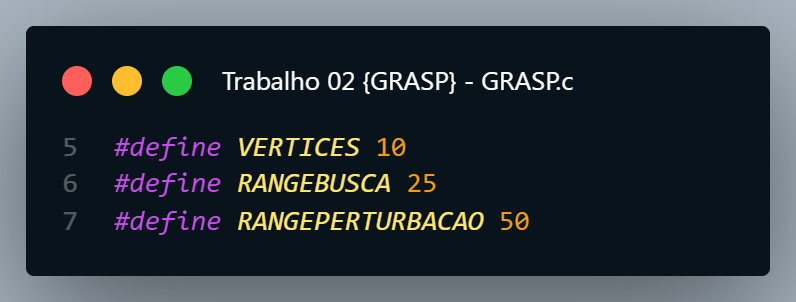
Para resolvermos o problema da coloração de vértices utilizando o GRASP primeiro vamos cruzar os conceitos. O GRASP trabalha com geração e modificação de combinações, buscando gerar um melhor resultado possível a partir dos parâmetros dados a ele. Com isso vamos levar em conta que a construção feita pelo algoritmo será de uma combinação de cores que será melhorada até chegar no melhor resultado possível, ou seja, no menor número de cores.

## 4.2 Parâmetros

Como parâmetros iniciais foram usados os dados de entrada base do algoritmo: quantidade de vértices, quantidade de arestas e uma matriz adjacência. Todos eles adicionados por um arquivo “grafo.txt”.

*.*

Além dos dados adicionados pelo arquivo de texto, temos também alguns valores criados como variáveis globais, que auxiliam no funcionamento do algoritmo:



A variável “VERTICE” contém o número de vértices da nossa entrada do arquivo de texto, ela existe para a passagem da matriz adjacência para outras funções como parâmetro. A variável “RANGEBUSCA” define a quantidade de vezes que nosso algoritmo vai realizar a busca local. E, por fim, a variável “RANGEPERTURBACAO” define o critério de aceitação do algoritmo, ou seja, quantas vezes o nosso algoritmo vai alterar entre busca local e perturbação antes de encerras as buscas.

Tanto o “RANGEBUSCA” quanto o “RANGEPERTURBACAO” poderiam ser trocados por um critério de aceitação baseado na melhora da combinação de entrada, mas como os testes com esse algoritmo vão ser feitos usando grafos pequenos, o critério de aceitação por repetição é melhor adaptável.

## 4.3 Leitura dos dados

Tendo os parâmetros definidos e a nossa lógica em mente, agora começaremos a leitura dos dados. Os dados se encontram em um arquivo .txt no nosso diretório. Dentro deles estão os principais dados que variam do problema, eles são respectivamente:



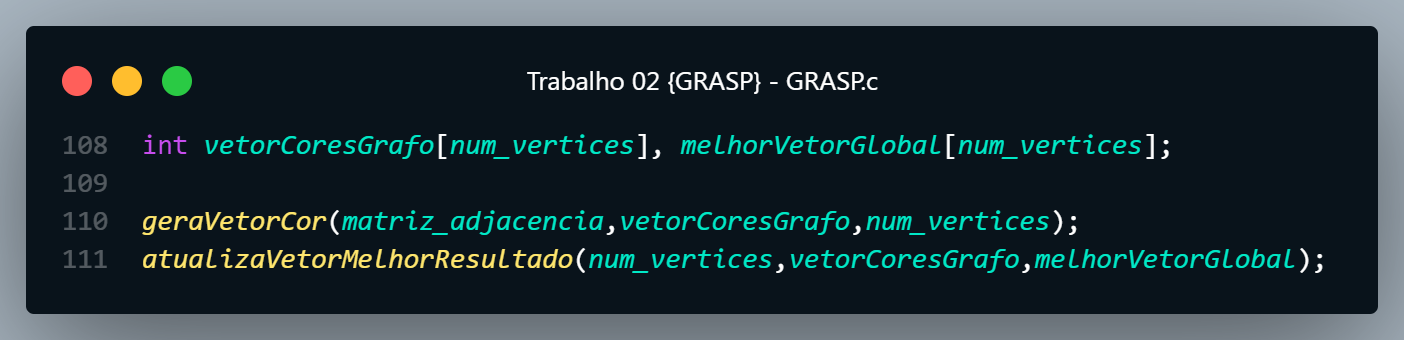
*(valores unicamente ilustrativos)*

Para leitura dos dados utilizamos as funções de leitura de arquivos normais com uma verificação de erro caso o programa não consiga ler o arquivo. Após isso, lemos as 2 primeiras linhas, que possuem os dados referentes a quantidade de vértices e quantidade de arestas, respectivamente. E terminamos a leitura com um loop que vai armazenar os valores de adjacência em suas respectivas casas na matriz adjacência.



## 4.4 Busca de solução inicial com algoritmo guloso

Para iniciar a busca por uma solução inicial válida, primeiro temos que criar as variáveis que vamos usar. Elas são, respectivamente, “vetorCoresGrafo” representando a combinação atual que vai ser alterada por todo o algoritmo, e “melhorVetorGlobal” representando a melhor combinação encontrada até o momento.



Após esse passo, iniciamos a função “geraVetorCor” que é o algoritmo guloso que terá como papel a criação dessa primeira solução.

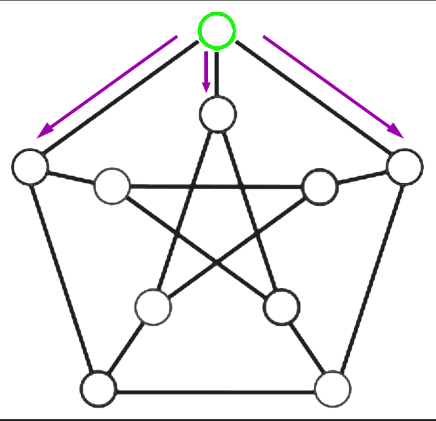
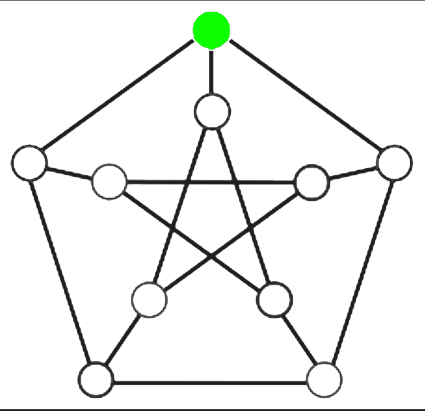
### 4.4.1 Função geraVetorCor



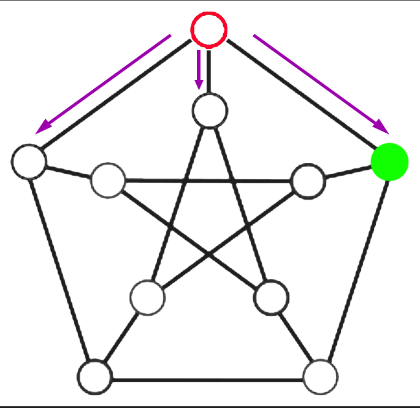
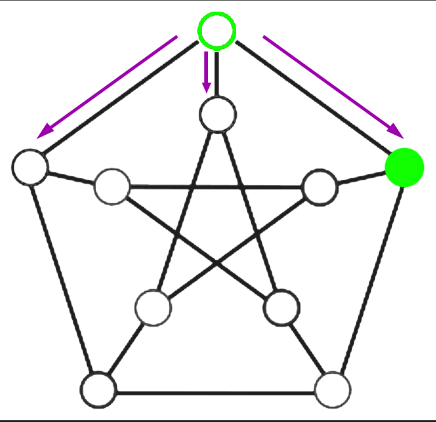
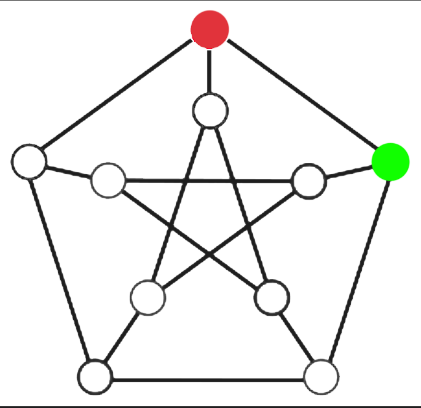
Nessa função, vamos trabalhar com dois loops de variável comum ‘for’, onde a variável “i” representa o vértice que estamos tentando colorir, o chamaremos de vértice assistido, e a variável “j” representa os vértices restantes do grafo, de modo que nos permita fazer comparações entre o vértice assistido e os demais vértices do grafo.

Dentro do primeiro loop iniciamos uma variável “c” com o número 1, ela representa as possíveis cores para o grafo, cada número de 1 a n representa uma cor aleatória. Após esse primeiro loop, o segundo trabalha para comprar o primeiro vértice aos demais e as condicionais internas impõem que caso haja adjacência entre o vértice assistido e algum outro e a cor do vértice que faz adjacência for igual a cor que queremos colorir nosso vértice ele trocará a cor desejada pela próxima e voltará a comparação do início.

Para entender melhor, vamos observar visualmente como funciona:

Primeiro é selecionado um vértice para ser o nosso vértice assistido (contornado em verde), após isso ele observará os vértices que tem adjacência com ele e comparará se a cor que ele deseja adotar é a mesma de algum dos vértices. Caso não haja o problema de algum vértice adjacente ter a cor desejada, ele fixa a cor como a escolhida para a combinação.

Caso haja um problema de compatibilidade de cores entre vértices o algoritmo guloso vai descartar a cor pretendida atual, selecionando uma nova, e fazer uma nova verificação. Caso não restem mais problemas ele fixa a cor como a escolhida para a combinação. Essa estratégia pode se repetir diversas vezes até que ele encontre uma cor que possa ser utilizada no vértice sem problemas.

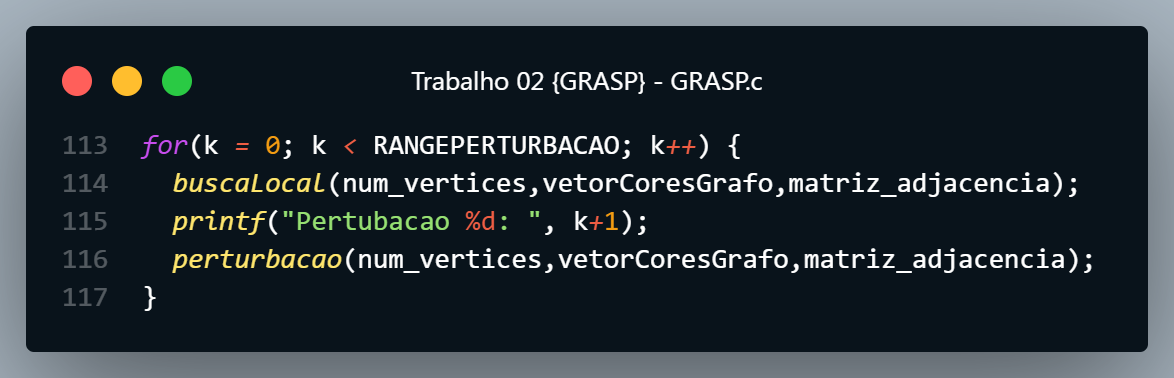
### 4.4.2 Função atualizaMelhorResultado

Por fim, como o resultado gerado pelo algoritmo guloso é nosso resultado inicial, ele é armazenado no nosso vetor de melhor resultado encontrado até o momento. Para isso é utilizada uma função simples que resgata os valores da combinação que estão no vetor de combinação atual para o vetor de melhor resultado, é importante lembrar que isso é feito pois o vetor de combinação atual terá o seu valor modificado várias vezes, tanto para melhor quanto para pior.



# 5.0 Evolução dos resultados

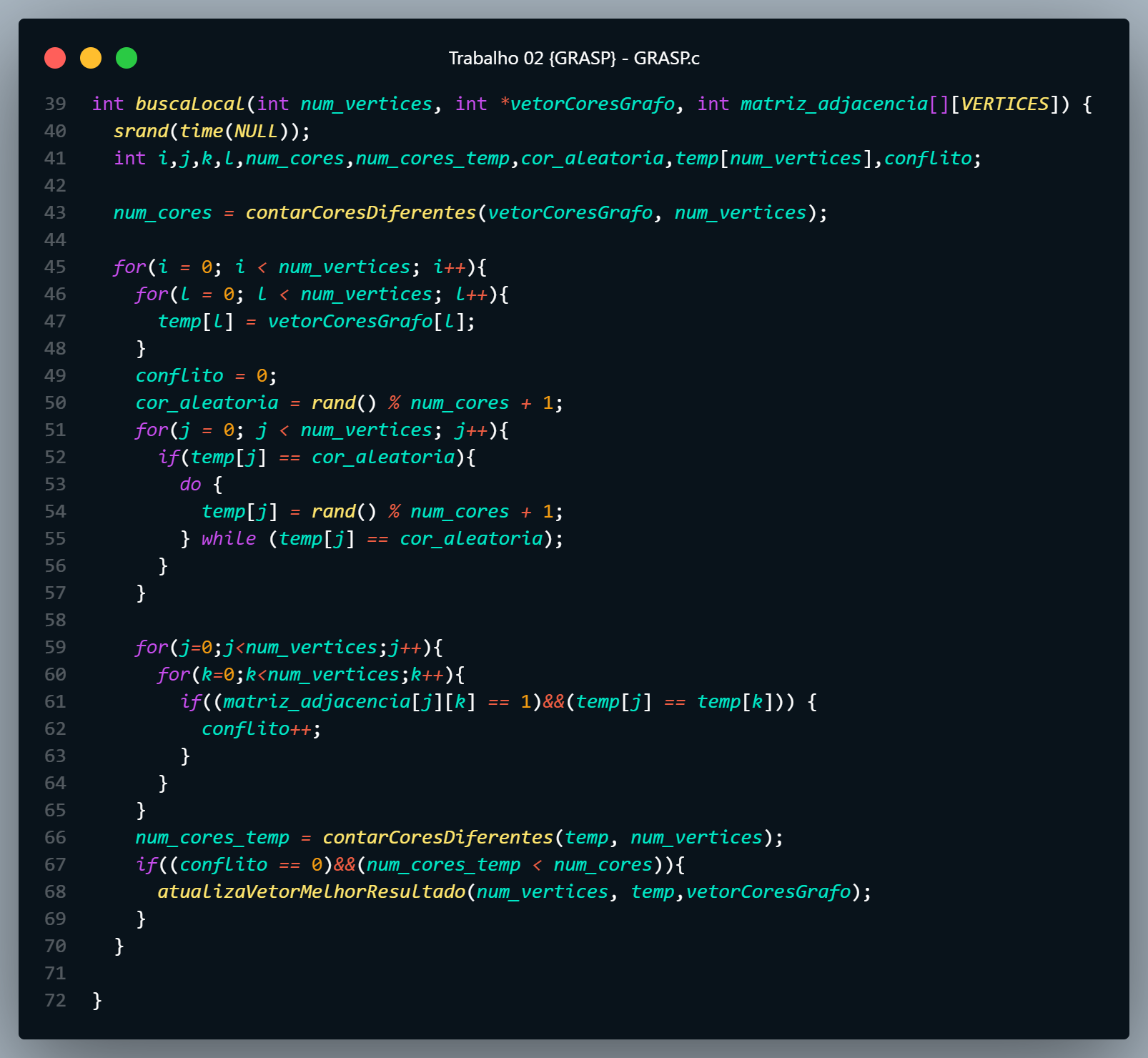
Após a geração de um resultado inicial, com ajuda de um algoritmo guloso, é necessário aprimorar o resultado localmente. Para isso é usada a junção das estratégias de busca local e perturbação de resultado.



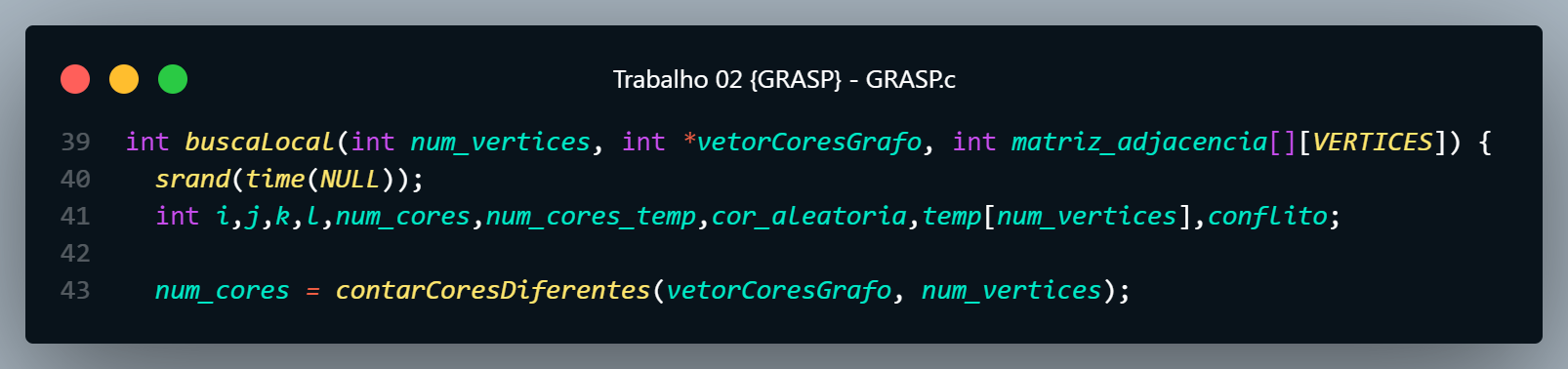
Usando um loop de variável simples, percorremos a busca local e a perturbação um número pré-determinado de vezes, como é dado na variável “RANGEPERTURBAÇÃO” como citado anteriormente. Deste modo, a busca local será responsável por percorrer a vizinhança da nossa combinação atual e a perturbação garantirá que o resultado não fique preso em uma única vizinhança.

## 5.1 Função “buscaLocal”

Essa função é responsável por melhorar o resultado garantido pela perturbação dentro da vizinhança do próprio, segue abaixo a análise detalhada de cada parte da função:



### 5.1.1 Variáveis



A função de busca local conta com as variáveis base de entrada (“num\_vertices”, “vetorCoresGrafo” e “matrizAdjacencia”), variáveis de contagem para loop (“i”, “j”, “k” e “l”), variáveis que armazenam a quantidade mínima de cores de cada combinação (“num\_cores\_temp” e “num\_cores”), variáveis de comparação (“cor\_aleatoria” e “conflito”) e uma variável de vetor temporário (“temp”).

### 5.1.2 Função “contarCoresDiferentes”

Como o próprio nome evidencia, a função “contarCoresDiferentes” é responsável por contar quantas cores únicas temos dentro de cada vetor de combinação, ou seja, o mínimo de cores necessário para a combinação em questão.



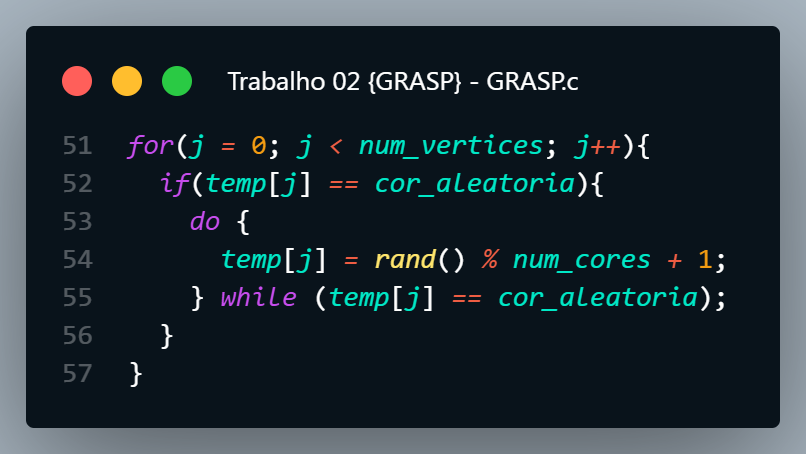
Para isso, ela percorre o vetor observando um vértice de cada vez, ao observar um vértice ela percorrerá os vértices a frente dele no vetor. Caso haja algum vértice a frente dele com a mesma cor, ele não fará nada. Ao chegar ao ultimo vértice, que quando comparado não existe nenhum outro na sua frente com a mesma cor que a sua, ele adiciona um a um contador de cores.

### 5.1.2 Escolha de uma cor aleatória

Com as variáveis com seus determinados valores, é iniciado o corpo da função. Ele consiste de um loop principal que mantém todo o processo funcionando n vezes, sendo n o número de vezes que o algoritmo é ajustado para fazer, representado pela variável global “RANGEBUSCA”, como já foi falado anteriormente.



Após isso ele irá criar uma cópia do vetor trabalhado, para que não perca a combinação trabalhada, enquanto busca outra melhor na vizinhança. Além disso ele seleciona valor para duas variáveis, “conflito” recebendo 0, já que cada busca vai usar essa variável como parâmetro para adoção de um novo resultado atual, e “cor\_aleatoria” recebendo uma cor aleatória dentro das cores já usadas, buscando escolher uma cor para ser removida da combinação e substituída por outras, já que o objetivo da melhora do resultado é diminuir a quantidade mínima de cores necessárias para colorir o grafo.



Nessa sequência do código é possível observar como é feita a troca da cor aleatória escolhida por uma das cores restantes. Isso é feito através de um loop de variável simples que percorrerá todo o vetor combinação. Dentro desse for ele vai observar cada vértice e se o vértice em questão possuir a cor que foi escolhida para ser excluída, ele irá fazer a troca utilizando um loop condicional; “While”.

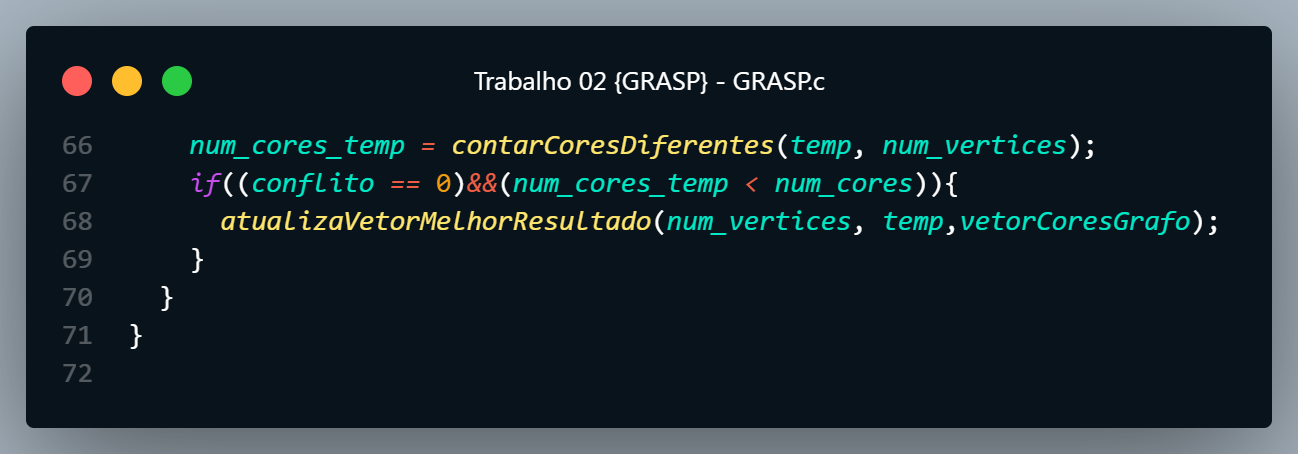
### 5.1.3 Verificação de conflitos

Após remover a cor escolhida o algoritmo deve verificar se houve algum conflito com as regras do problema, ou seja, se algum vértice possui e mesma cor que um vértice adjacente a ele. Para isso ele percorrerá a matriz de adjacência verificando, para cada vértice, se há adjacência e, se houver, se as cores dos dois vértices são iguais.



### 5.1.4 Confirmando resultados

Com os dados de conflitos salvos, a função verifica a quantidade de cores únicas dentro do vetor criado e salva esse valor em uma variável. Logo após, uma condicional é feita para confirmar se não houveram conflitos e se a quantidade de cores da combinação criada atual é menor do que a da melhor combinação que nosso algoritmo já encontrou. Se essas condições forem satisfeitas, a função “atualizaMelhorResultado” é chamada para atualizar o vetor que armazena o melhor resultado enc1ontrado pelo algoritmo até agora.



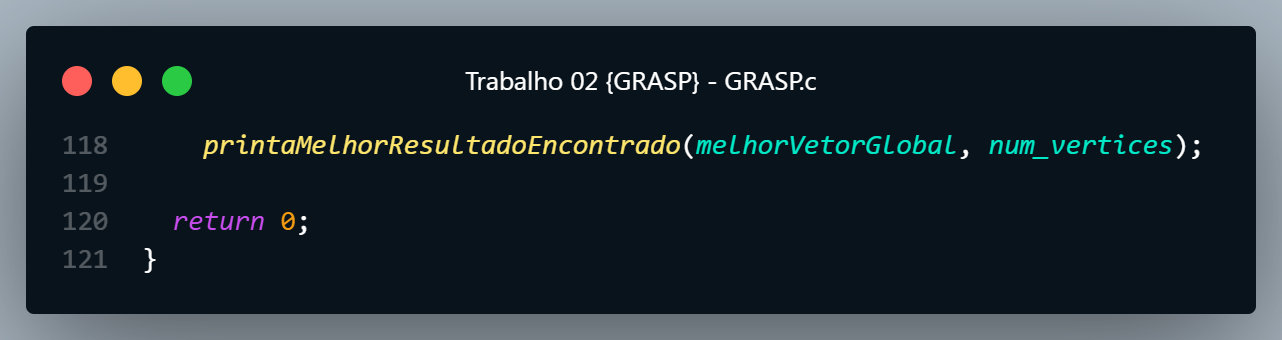
## 5.2 Função “perturbação”

Por fim, após cada busca local é necessário renovar o escopo de buscas do algoritmo. Isso é feito por meio de uma perturbação, que gera uma nova combinação aleatória que possua o mínimo número de cores igual a da melhor combinação atual. A partir dessa nova combinação é feita uma nova busca local esse processo se repete até o critério de aceitação ser completo (fazer a busca n vezes).



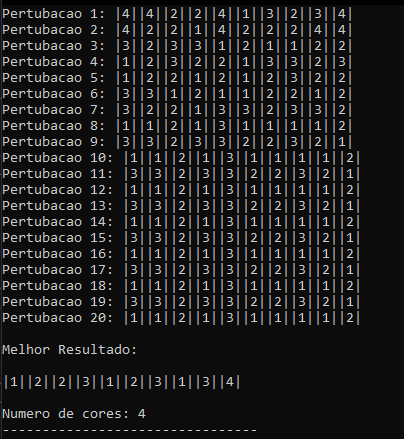
# 6.0 Fim do código

Finalmente, o melhor resultado encontrado pelo algoritmo durante sua execução é mostrado na tela, juntamente com o valor mínimo de cores necessárias para essa combinação.



# 7.0 Resultados encontrados

Resultados aleatorizados por cada perturbação e o melhor resultado gerado pelo algoritmo, testados no grafo de grafo de Petersen.



# 7.0 Referências

FELICE, Mário César San. **s2v04 Algoritmos Gulosos: Coloração (IOCA)**. Local: https://www.youtube.com/watch?v=Qyv20alPqec. (Acesso: 10/02/2024)

FERREIRA, Marcelo Rego; GAMBINI, Haroldo Santos. **Algoritmos para o Problema de Coloração de Grafos**. Local: http://www.decom.ufop.br/menotti/paa111/files/PCC104-111-ars-11.1-MarceloFerreiraRego.pdf. (Acesso: 10/02/2024)

FREDDO, Ademir Roberto; BRITO, Robison Cris. **Implementação da Metaheurística GRASP para o Problema do Caixeiro Viajante Simétrico**. Local: https://www.inf.ufpr.br/aurora/disciplinas/topicosia2/downloads/trabalhos/GraspTSP.pdf (Acesso: 10/02/2024)

SANTIAGO, Pedro Henrique Rivera. **Método GRASP e ACO em Otimização**. Local: https://www.ime.unicamp.br/~mac/db/2015-1S-120022.pdf. (Acesso: 10/02/2024)

BORDIN, Airton Junior. **Metaheurística GRASP**. Local: https://airtonbjunior.github.io/mestrado/metaheuristics/grasp/presentation/MH\_5.pdf. (Acesso: 10/02/2024)

SANTOS, Philippe Leal Freire. **TEORIA ESPECTRAL DE GRAFOS APLICADA AO PROBLEMA DE ISOMORFISMO DE GRAFOS**. Local: https://repositorio.ufes.br/server/api/core/bitstreams/47e4049d-e1c3-49d0-becf-5e254537bbb2/content. (Acesso: 10/02/2024)

Štefaníková, P; Váňa, P; Faigl, J. . **Greedy Randomized Adaptive Search Procedure for Close Enough Orienteering Problem**. Local: https://www.youtube.com/watch?v=ek6mfGX73kg . (Acesso: 10/02/2024)