Resolução Heurística do Problema L(p,q)-Coloring

Seu Nome
Matrícula: XXXXX

DCC059 - Teoria dos Grafos
Universidade Federal de Juiz de Fora

21 de janeiro de 2026

Sumário

1 Introdução

O problema L(p,q)-coloring é uma generalização do problema clássico de coloração de grafos...

[Descreva o problema, sua importância e aplicações]

1.1 Definição do Problema

Dado um grafo simples não direcionado $G = (V, E)$ e dois inteiros não negativos p e q, o problema L(p,q)-coloring consiste em encontrar uma coloração $f : V \to \mathbb{Z}^+$ dos vértices tal que:

- Se u e v são vértices adjacentes (distância 1), então $|f(u) - f(v)| \geq p$
- Se u e v estão a distância 2, então $|f(u) - f(v)| \geq q$

O objetivo é minimizar a maior cor utilizada na coloração.

2 Algoritmos Implementados

2.1 Algoritmo Guloso

O algoritmo guloso implementado utiliza uma heurística construtiva aprimorada e determinística. A estratégia consiste em ordenar os vértices considerando tanto o grau direto quanto o número de vizinhos a distância 2, e então colorir cada vértice sequencialmente com a menor cor válida possível.

A ordenação aprimorada é crucial para o problema L(p,q)-coloring, pois vértices com maior grau combinado (direto + distância 2) tendem a ter mais restrições. Ao colori-los primeiro, reduz-se significativamente a chance de conflitos nas etapas finais. Para cada vértice, o algoritmo busca a menor cor (iniciando em 0) que satisfaça ambas as restrições: diferença $\geq p$ com vizinhos diretos e diferença $\geq q$ com vizinhos a distância 2.

Esta heurística mostrou-se superior à ordenação simples por grau, produzindo soluções de melhor qualidade em grafos densos.

Algorithm 1 Algoritmo Guloso para L(p,q)-Coloring

```
V ← vértices ordenados por (grau + |N2|) decrescente ▷ N2 = vizinhos a distância 2
coloring ← vetor de tamanho |V| inicializado com −1
maxColor ← 0
for cada vértice v ∈ V do
    cor ← 0
    while ¬ isValidColor(v, cor, coloring) do
        cor ← cor + 1
    end while
    coloring[v] ← cor
    maxColor ← max(maxColor, cor)
end for
return (coloring, maxColor)
```

2.2 Algoritmo Guloso Randomizado

O algoritmo guloso randomizado é baseado na técnica GRASP (Greedy Randomized Adaptive Search Procedure). Em vez de sempre escolher o vértice com menor custo, ele introduz aleatoriedade através de uma Lista Restrita de Candidatos (RCL).

O parâmetro $\alpha \in [0, 1]$ controla o grau de aleatoriedade: com $\alpha = 0$, o algoritmo é puramente guloso (escolhe sempre o melhor candidato), enquanto $\alpha = 1$ torna a escolha completamente aleatória. O custo de cada vértice não colorido combina a menor cor válida necessária com o **grau de saturação** (número de vizinhos já coloridos), priorizando vértices mais restritos. Especificamente, o custo é definido como:

$$cost(v) = minColor(v) \times 100 - saturationDegree(v)$$

A RCL contém todos os vértices cujo custo está dentro de um limiar $threshold = minCost + \alpha \cdot (maxCost - minCost)$.

O algoritmo executa múltiplas iterações (geralmente 30-50) e, ao final, aplica uma **busca local** para refinar a melhor solução encontrada, tentando recolorir vértices com cores altas em cores menores.

Algorithm 2 Algoritmo Guloso Randomizado

$bestSol \leftarrow \emptyset$
for $iter = 1$ até $iterations$ **do**
    $coloring \leftarrow$ vetor inicializado com -1
    $uncolored \leftarrow$ todos os vértices
    while $uncolored \neq \emptyset$ **do**
        for cada $v \in uncolored$ **do**
            $minColor \leftarrow$ findSmallestValidColor$(v, coloring)$
            $satDegree \leftarrow$ número de vizinhos coloridos de v
            $cost[v] \leftarrow minColor \times 100 - satDegree$
        end for
        $minCost \leftarrow \min(cost)$, $maxCost \leftarrow \max(cost)$
        $threshold \leftarrow minCost + \alpha \cdot (maxCost - minCost)$
        $RCL \leftarrow \{v \in uncolored : cost[v] \leq threshold\}$
        $v^* \leftarrow$ escolher aleatoriamente de RCL
        $coloring[v^*] \leftarrow$ findSmallestValidColor$(v^*, coloring)$
        $uncolored \leftarrow uncolored \setminus \{v^*\}$
    end while
    $bestSol \leftarrow$ atualizar se solução atual for melhor
end for
$bestSol \leftarrow$ localSearch$(bestSol)$ ▷ Refinamento
return $bestSol$

2.3 Algoritmo Guloso Randomizado Reativo

O algoritmo reativo é uma extensão do GRASP que adapta automaticamente o parâmetro α durante a execução. Em vez de usar um único valor fixo de α, ele trabalha com um conjunto de valores candidatos cuidadosamente escolhidos e ajusta dinamicamente as probabilidades de seleção de cada α baseado no desempenho observado.

Nossa implementação utiliza valores de α mais agressivos: $\{0.02, 0.05, 0.10, 0.15\}$, que favorecem escolhas mais gulosas (menor aleatoriedade), resultando em soluções de melhor qualidade para grafos densos. Valores menores de α concentram a busca em candidatos promissores, enquanto ainda mantendo diversificação suficiente.

O algoritmo divide as iterações em blocos (geralmente de tamanho 40-50). Ao final de cada bloco, ele avalia a qualidade média das soluções obtidas com cada α e atualiza as probabilidades de forma proporcional: valores de α que geraram soluções de melhor qualidade recebem probabilidades maiores de serem selecionados no próximo bloco. A qualidade é medida como $q = \frac{1}{1+maxColor}$, de modo que soluções com menor cor máxima têm maior qualidade.

Após todas as iterações, aplica-se uma **busca local mais extensa** (até 100 iterações) para refinar a melhor solução encontrada. A implementação é modularizada, com métodos auxiliares para construção de soluções (`buildSolution`) e atualização de probabilidades (`updateProbabilities`), facilitando manutenção e extensões futuras.

Este mecanismo de aprendizado permite que o algoritmo se adapte às características específicas de cada instância, privilegiando os valores de α mais eficazes.

Algorithm 3 Algoritmo Guloso Randomizado Reativo

$alphas \leftarrow \{0.02, 0.05, 0.10, 0.15\}$
$probabilities \leftarrow$ distribuição uniforme sobre $alphas$
$bestSol \leftarrow \emptyset$, $blockQuality \leftarrow 0$
for $iter = 1$ até $iterations$ **do**
    $\alpha \leftarrow$ selecionar de $alphas$ com probabilidades
    $sol \leftarrow$ buildSolution(α) ▷ Método modularizado
    $quality \leftarrow \frac{1}{1+sol.maxColor}$
    $blockQuality[\alpha] \leftarrow blockQuality[\alpha] + quality$
    $bestSol \leftarrow$ atualizar se sol for melhor
    if $iter \mod blockSize = 0$ **then**
        updateProbabilities($probabilities, blockQuality, blockUsage$)
        $blockQuality \leftarrow$ resetar para próximo bloco
    end if
end for
$bestSol \leftarrow$ localSearch($bestSol, 100$) ▷ Refinamento extenso
return $bestSol$

3 Metodologia Experimental

3.1 Instâncias Utilizadas

As instâncias utilizadas foram obtidas de [fonte]...

[Liste as instâncias usadas]

3.2 Configuração dos Experimentos

- Cada algoritmo foi executado 10 vezes para cada instância
- Valores de α testados: 0.1, 0.3, 0.5

- Algoritmo randomizado: 30 iterações
- Algoritmo reativo: 300 iterações com blocos de 30
- Valores de p e q: [especificar]

4 Resultados

4.1 Desvio Percentual da Melhor Solução

[Inserir tabela gerada pelo script analyze$_r esults.py$]

4.2 Desvio Percentual da Média

[Inserir tabela gerada pelo script]

4.3 Tempo Médio de Execução

[Inserir tabela de tempos]

5 Análise dos Resultados

[Discuta os resultados obtidos:]

- Compare o desempenho dos três algoritmos
- Analise o impacto do parâmetro α
- Discuta o comportamento do algoritmo reativo
- Avalie o trade-off entre tempo e qualidade

6 Conclusão

[Resuma os principais resultados e aprendizados]

Referências

[1] Griggs, J. R., & Yeh, R. K. (1992). *Labelling graphs with a condition at distance 2.* SIAM Journal on Discrete Mathematics, 5(4), 586-595.