Árvore Geradora Mínima Generalizada (AGMG)

Descrição do Problema:

A **Árvore Geradora Mínima Generalizada (AGMG)** é uma variação do problema clássico da **Árvore Geradora Mínima (AGM)**, no qual se busca conectar todos os vértices de um grafo ponderado de forma a minimizar o custo total das arestas. No entanto, no problema da **AGMG**, os vértices do grafo estão divididos em **grupos disjuntos**, e a solução deve garantir que pelo menos um vértice de cada grupo esteja presente na árvore geradora.

Dado um grafo não direcionado e ponderado **G = (V,E)**, onde os vértices **V** são particionados em conjuntos disjuntos $\mathbf{V_1,V_2,\ldots,V_k}$, o objetivo é encontrar uma árvore de custo mínimo que conecte pelo menos um vértice de cada conjunto $\mathbf{V_i}$. Diferentemente da AGM tradicional, que deve incluir **todos os vértices do grafo**, a **AGMG** seleciona um subconjunto deles, respeitando a restrição de que cada grupo seja representado.

Esse problema tem aplicações em diversas áreas, como redes de comunicação, otimização de transporte e logística, além de agrupamento em aprendizado de máquina. Entretanto, devido à necessidade de selecionar vértices específicos de cada grupo, a solução da AGMG exige algoritmos mais complexos do que os utilizados para encontrar uma AGM comum, como heurísticas, programação inteira e meta-heurísticas.

Descrição das Instâncias:

Instância 1: aa4 (Miscellaneous Networks)
- **Nós:** 7.2K
- **Arestas:** 52.1K
- **Densidade:** 0.00201372
- **Grau máximo:** 410
- **Grau mínimo:** 2
- **Grau médio:** 14
- **Assortatividade:** -0.621453
- **Número de triângulos:** 24.1K
- **Número médio de triângulos:** 3
- **Número máximo de triângulos:** 644
- **Coeficiente médio de agrupamento:** 0.0393912
- **Fração de triângulos fechados:** 0.00463299
- **Máximo k-core:** 11
- **Limite inferior do Clique Máximo:** 5

Instância 2: grid-fruitfly (Biological Networks)
- **Nós:** 7.3K
- **Arestas:** 49.8K
- **Densidade:** 0.00188221
- **Grau máximo:** 352
- **Grau mínimo:** 2
- **Grau médio:** 13
- **Assortatividade:** -0.0368592
- **Número de triângulos:** 60.8K
- **Número médio de triângulos:** 8
- **Número máximo de triângulos:** 1.4K

Coeficiente médio de agrupamento: 0.0244236
Fração de triângulos fechados: 0.0266215
Máximo k-core: 25
Limite inferior do Clique Máximo: 7

Instância 3: grid-yeast (Biological Networks)
Nós: 6K
Arestas: 313.9K
Densidade: 0.0173948
Grau máximo: 5.1K
Grau mínimo: 2
Grau médio: 104
Assortatividade: -0.0838839
Número de triângulos: 11.2M
Número médio de triângulos: 1.9K
Número máximo de triângulos: 145.3K
Coeficiente médio de agrupamento: 0.163622
Fração de triângulos fechados: 0.101147
Máximo k-core: 129
Limite inferior do Clique Máximo: 11

Instância 4: Erdos992 (Collaboration Networks)
Nós: 6.1K
Arestas: 7.5K
Densidade: 0.00040399
Grau máximo: 61
Grau mínimo: 0
Grau médio: 2
Assortatividade: -0.443834
Número de triângulos: 4.8K
Número médio de triângulos: 0
Número máximo de triângulos: 99
Coeficiente médio de agrupamento: 0.0683396
Fração de triângulos fechados: 0.0419402
Máximo k-core: 8
Limite inferior do Clique Máximo: 7

Instância 5: airfoil1-dual (Miscellaneous Networks)
Nós: 8K
Arestas: 11.8K
Densidade: 0.000366084
Grau máximo: 3
Grau mínimo: 2
Grau médio: 2
Assortatividade: -0.024474
Número de triângulos: 0
Número médio de triângulos: 0
Número máximo de triângulos: 0
Coeficiente médio de agrupamento: 0
Fração de triângulos fechados: 0
Máximo k-core: 3
Limite inferior do Clique Máximo: 2

Instância 6: EX5 (Miscellaneous Networks)

Nós: 6.5K
Arestas: 147.8K
Densidade: 0.00690349
Grau máximo: 48
Grau mínimo: 42
Grau médio: 45
Assortatividade: 0.246929
Número de triângulos: 887K
Número médio de triângulos: 135
Número máximo de triângulos: 144
Coeficiente médio de agrupamento: 0.136015
Fração de triângulos fechados: 0.135628
Máximo k-core: 43
Limite inferior do Clique Máximo: 5

Instância 7: ukerbe (Miscellaneous Networks)

Nós: 6K
Arestas: 7.9K
Densidade: 0.000439072
Grau máximo: 8
Grau mínimo: 2
Grau médio: 2
Assortatividade: -0.733212
Número de triângulos: 0
Número médio de triângulos: 0
Número máximo de triângulos: 0
Coeficiente médio de agrupamento: 0
Fração de triângulos fechados: 0
Máximo k-core: 3
Limite inferior do Clique Máximo: 2

Instância 8: as-735 (Miscellaneous Networks)

Nós: 6.5K
Arestas: 13.9K
Densidade: -
Grau máximo: 1.5K
Grau mínimo: 1
Grau médio: 2.14627741736
Assortatividade: -
Número de triângulos: -
Número médio de triângulos: -
Número máximo de triângulos: -
Coeficiente médio de agrupamento: -
Fração de triângulos fechados: -
Máximo k-core: -
Limite inferior do Clique Máximo: -

Instância 9: p2p-Gnutella08 (Miscellaneous Networks)

Nós: 6.3K
Arestas: 20.8K
Densidade: 0.0010468
Grau máximo: 97
Grau mínimo: 1
Grau médio: 6
Assortatividade: 0.0355504
Número de triângulos: 7.1K
Número médio de triângulos: 1
Número máximo de triângulos: 202
Coeficiente médio de agrupamento: 0.0108679
Fração de triângulos fechados: 0.0206599
Máximo k-core: 11
Limite inferior do Clique Máximo: 5

Instância 10: dmela (Biological Networks)
Nós: 7.4K
Arestas: 25.6K
Densidade: 0.000935753
Grau máximo: 190
Grau mínimo: 1
Grau médio: 6
Assortatividade: -0.0465044
Número de triângulos: 8.7K
Número médio de triângulos: 1
Número máximo de triângulos: 225
Coeficiente médio de agrupamento: 0.0118502
Fração de triângulos fechados: 0.0149699
Máximo k-core: 12
Limite inferior do Clique Máximo: 7

Clusters das instâncias:

Para adicionar conjuntos de clusters as instâncias escolhidas utilizamos uma lógica matemática, no qual vértices pertencem ao cluster equivalente ao resto da divisão de seu ID por 10, mais 1. Por exemplo, o vértice 71 pertence ao cluster 2 (71 % 10 + 1).

Descrição dos Métodos Implementados:

Guloso:

A função **encontrar_agmg_guloso** implementa uma abordagem gulosa para resolver o problema da Árvore Geradora Mínima Generalizada (AGMG). O objetivo é encontrar uma árvore de custo mínimo que conecte pelo menos um vértice de cada cluster (grupo de vértices) presente no grafo. A abordagem gulosa prioriza a escolha de arestas de menor peso que

conectam clusters ainda não representados na árvore. A seguir, detalhamos o funcionamento dessa função:

1. Estruturas de Dados Utilizadas

- **Vértices Visitados:** Um array de booleanos (**vertices_visitados**) é utilizado para marcar quais vértices já foram incluídos na AGMG.
- **Clusters Conectados:** Um array de booleanos **(cluster_conectados**) é utilizado para marcar quais clusters já estão representados na AGMG.
- **Soma dos Pesos:** Uma variável (soma_pesos_agmg) é utilizada para armazenar o custo total da AGMG.

2. Passos do Algoritmo

a. Inicialização:

- O algoritmo começa inicializando os arrays vertices_visitados e cluster_conectados com valores false, indicando que nenhum vértice ou cluster foi incluído na AGMG.
- O vértice inicial é definido como o vértice 1, e ele é marcado como visitado. O cluster correspondente a esse vértice também é marcado como conectado.

b. Iteração sobre os Vértices:

- O algoritmo entra em um loop que continua até que todos os clusters estejam conectados ou até que não haja mais vértices válidos para adicionar à AGMG.
- Para cada vértice atual, o algoritmo obtém a lista de seus vizinhos usando a função **get_vizinhos_vertices**.

c. Escolha Gulosa da Aresta:

- O algoritmo procura o vizinho com o menor peso que ainda não foi visitado. Esse vizinho é escolhido como o próximo vértice a ser adicionado à AGMG.
- Se o vizinho escolhido pertence a um cluster que ainda não foi conectado, ele é priorizado, garantindo que todos os clusters sejam representados na AGMG.

d. Atualização da AGMG:

- O vértice escolhido é marcado como visitado, e o cluster correspondente é marcado como conectado.
- O peso da aresta que conecta o vértice atual ao próximo vértice é adicionado à variável soma_pesos_agmg.

e. Verificação de Conclusão:

- Após adicionar um vértice, o algoritmo verifica se todos os clusters já estão conectados. Se sim, o loop é interrompido, e o custo total da AGMG é retornado.
- Caso contrário, o algoritmo continua iterando até que todos os clusters estejam conectados ou até que não haja mais vértices válidos para adicionar.

3. Complexidade e Considerações

- **Complexidade Temporal:** A complexidade do algoritmo depende do número de vértices e arestas no grafo. No pior caso, o algoritmo pode precisar percorrer todos os vértices e arestas, resultando em uma complexidade de $O(V+E)$, onde V é o número de vértices e E é o número de arestas.
- **Complexidade Espacial:** O algoritmo utiliza memória adicional para armazenar os arrays de controle (vertices_visitados e cluster_conectados), resultando em uma complexidade espacial de $O(V+C)$, onde C é o número de clusters.

4. Limitações e Melhorias Potenciais

- **Limitações:** A abordagem gulosa pode não garantir a solução ótima em todos os casos, especialmente em grafos com estruturas complexas. Além disso, o algoritmo pode falhar se não houver um caminho que conecte todos os clusters.
- **Melhorias:** A implementação de uma estrutura de dados Union-Find (Disjoint Set Union - DSU) pode ajudar a verificar ciclos de forma mais eficiente. Além disso, a utilização de algoritmos de ordenação mais eficientes pode melhorar o desempenho.

5. Exemplo de Funcionamento

- Considere um grafo com 5 vértices e 7 arestas, divididos em 3 clusters. O algoritmo começa no vértice 1 e marca seu cluster como conectado. Em seguida, ele escolhe a aresta de menor peso que conecta o vértice 1 a um vértice de outro cluster. O processo continua até que todos os clusters estejam conectados.

6. Conclusão

- A função **encontrar_agmg_guloso** implementa uma abordagem simples e intuitiva para resolver o problema da AGMG. Embora possa não ser a solução mais eficiente para grafos muito grandes, ela é eficaz para instâncias menores e serve como uma base para implementações mais sofisticadas, como algoritmos randomizados ou reativos.

Randomizado:

Reativo:

A função **arvore_geradora_minima_reativa** implementa uma abordagem reativa para resolver o problema da Árvore Geradora Mínima Generalizada (AGMG). O objetivo é encontrar uma árvore de custo mínimo que conecte pelo menos um vértice de cada cluster (grupo de vértices) presente no grafo. A abordagem reativa adapta-se dinamicamente às características do grafo, garantindo que todos os clusters sejam representados na árvore geradora mínima. A seguir, detalhamos o funcionamento dessa função:

1. Estruturas de Dados Utilizadas

- **AGM (Árvore Geradora Mínima):** A função retorna um array de pares de inteiros (std::pair<int, int>), onde cada par representa uma aresta da árvore geradora mínima.
- **Aresta:** Uma estrutura interna (struct Aresta) é utilizada para armazenar as informações das arestas do grafo, incluindo o peso, o vértice de origem e o vértice de destino.
- **Clusters:** Um array de inteiros (clusters) é utilizado para armazenar o cluster ao qual cada vértice pertence. Esse array é preenchido previamente pela função **criar_clusters_aleatorios**.

2. Passos do Algoritmo

a. Verificação Inicial:

- O algoritmo começa verificando se o grafo possui vértices. Caso contrário, ele retorna um erro, pois não é possível construir uma AGM em um grafo vazio.

b. Inicialização de Estruturas de Controle:

- O algoritmo utiliza dois arrays de booleanos para controlar quais vértices e clusters já estão presentes na AGM:
  - **na_agm**: Um array que indica se um vértice já foi incluído na AGM.
  - **cluster_representado**: Um array que indica se um cluster já está representado na AGM.
- Além disso, um contador (**clusters_representados**) é utilizado para rastrear quantos clusters já foram incluídos na AGM.

c. Coleta de Todas as Arestas do Grafo:

- O algoritmo percorre todos os vértices do grafo e coleta todas as arestas existentes. Para cada vértice, ele obtém a lista de vizinhos e armazena as arestas em um array de estruturas Aresta.
- O peso de cada aresta é obtido através da função **get_aresta**, que retorna o peso da aresta entre dois vértices.

d. Ordenação das Arestas por Peso:

- Após coletar todas as arestas, o algoritmo ordena as arestas em ordem crescente de peso. Isso é feito utilizando um algoritmo de ordenação simples, como o Bubble Sort, que compara os pesos das arestas e as rearranja no array.
- A ordenação é crucial para a abordagem reativa, pois o algoritmo sempre escolhe a aresta de menor peso disponível que não forma ciclos e que ajuda a conectar clusters ainda não representados na árvore.

e. Adição Inicial de Vértices de Cada Cluster:

- Para garantir que todos os clusters estejam representados na AGM, o algoritmo adiciona o primeiro vértice de cada cluster à AGM. Isso é

feito verificando o array clusters e marcando os vértices correspondentes como incluídos na AGM.

f. Construção da AGM:

- O algoritmo percorre as arestas ordenadas e, para cada aresta, verifica se ela conecta dois vértices que ainda não estão na AGM ou se ela ajuda a conectar um cluster ainda não representado.
- Se a aresta for válida (ou seja, não formar um ciclo e conectar clusters necessários), ela é adicionada à AGM. Os vértices são marcados como incluídos na AGM, e os clusters correspondentes são marcados como representados.
- O processo continua até que todos os clusters estejam representados na AGM ou até que todas as arestas tenham sido consideradas.

g. Verificação de Conclusão:

- Após adicionar uma aresta, o algoritmo verifica se todos os clusters já estão representados na AGM. Se sim, o processo é interrompido, e a AGM é retornada.
- Caso contrário, o algoritmo continua adicionando arestas até que todos os clusters estejam conectados.

3. Complexidade e Considerações

- **Complexidade Temporal:** A complexidade do algoritmo é dominada pela ordenação das arestas, que pode ser **O(ElogE)** se um algoritmo de ordenação eficiente for utilizado. No entanto, neste caso, o Bubble Sort é utilizado, o que resulta em uma complexidade de $\mathbf{O(E^2)}$ onde *E* é o número de arestas.
- **Complexidade Espacial:** O algoritmo utiliza memória adicional para armazenar as arestas, os arrays de controle e a AGM, resultando em uma complexidade espacial de O(V+E), onde V é o número de vértices e *E* é o número de arestas.

4. Limitações e Melhorias Potenciais

- **Limitações:** O uso do Bubble Sort para ordenar as arestas pode ser ineficiente para grafos grandes. Além disso, a abordagem reativa pode não garantir a solução ótima em todos os casos, especialmente em grafos com estruturas complexas.
- **Melhorias:** A utilização de algoritmos de ordenação mais eficientes, como o QuickSort ou MergeSort, pode melhorar o desempenho. Além disso, a implementação de uma estrutura de dados Union-Find (Disjoint Set Union - DSU) pode ajudar a verificar ciclos de forma mais eficiente.

5. Exemplo de Funcionamento

- Considere um grafo com 5 vértices e 7 arestas, divididos em 3 clusters. O algoritmo começa coletando todas as arestas e as ordena por peso. Em seguida, ele adiciona o primeiro vértice de cada cluster

à AGM. Depois, ele adiciona as arestas de menor peso que conectam clusters ainda não representados. Por exemplo, se a aresta de menor peso conecta o cluster 1 ao cluster 2, ela é adicionada à AGM. O processo continua até que todos os clusters estejam conectados.

6. Conclusão

- A função **arvore_geradora_minima_reativa** implementa uma abordagem adaptativa e dinâmica para resolver o problema da AGMG. Ela é eficaz para garantir que todos os clusters sejam representados na árvore geradora mínima, mesmo em grafos com estruturas complexas. Embora possa não ser a solução mais eficiente para grafos muito grandes, ela é uma abordagem robusta e flexível para instâncias menores e médias.

Análise do tempo de execução entre Lista e Matriz:

1. **Desempenho da Matriz vs. Lista:**
   - Em todos os casos, a implementação utilizando matriz foi mais rápida que a implementação utilizando lista.
   - A diferença de tempo é significativa, especialmente em instâncias maiores (por exemplo, instância 6, onde a matriz levou ~1.481s e a lista levou ~2m8.601s).
2. **Crescimento do Tempo de Execução:**
   - O tempo de execução da lista cresce de forma mais acentuada em comparação com a matriz à medida que o tamanho da instância aumenta.
   - Isso sugere que a implementação com lista tem uma complexidade assintótica maior ou uma constante de proporcionalidade maior.
3. **Resultados Obtidos:**
   - Em algumas instâncias, ambas as implementações (matriz e lista) obtiveram o mesmo resultado (por exemplo, instâncias 4, 5, 7, 8 e 10).
   - Em outras, a implementação com lista não conseguiu encontrar uma solução ("Não"), enquanto a implementação com matriz obteve um resultado válido (por exemplo, instância 1 no algoritmo Guloso).

Análise Detalhada por Instância

Algoritmo Guloso

Instância	Tempo Matriz	Tempo Lista	Resultado Matriz	Resultado Lista
1	0.597s	18.302s	3965	33965
2	0.618s	7.974s	Não	Não
3	1.643s	8.564s	Não	Não

4	0.346s	0.607s	729	729
5	0.402s	2.290s	156	156
6	1.481s	2m8.601s	62	62
7	0.379s	0.930s	2186	2186
8	0.381s	1.585s	2315	2315
9	0.403s	3.552s	Não	Não
10	0.417s	5.295s	809	809

Algoritmo Randomizado

Instância	Tempo Matriz	Tempo Lista	Resultado Matriz	Resultado Lista
1	0.562s	16.722s	285	285
2	0.576s	8.038s	Não	Não
3	1.735s	4m11.682s	Não	Não
4	0.314s	0.638s	729	729
5	0.405s	2.338s	156	156
6	1.504s	2m10.724s	62	62
7	0.342s	0.931s	2186	2186
8	0.376s	1.630s	2315	2315
9	0.380s	3.572s	Não	Não
10	0.418s	5.414s	809	809

Conclusões

1. **Eficiência:**
    - A implementação com matriz é claramente mais eficiente em termos de tempo de execução.
    - A implementação com lista pode não ser viável para instâncias maiores devido ao tempo de execução excessivo.
2. **Precisão:**
    - Ambas as implementações (matriz e lista) obtiveram os mesmos resultados na maioria dos casos.

    - Em alguns casos, a implementação com lista falhou em encontrar uma solução, enquanto a implementação com matriz teve sucesso.
3. **Recomendação:**
    - Para problemas de maior escala, a implementação com matriz é preferível devido ao seu desempenho superior.
    - A implementação com lista pode ser útil apenas para instâncias pequenas ou quando o uso de memória é uma preocupação maior que o tempo de execução.

Possíveis Razões para a Diferença de Desempenho

1. **Acesso à Memória:**
    - Matrizes possuem acesso mais rápido à memória devido à localidade de referência, enquanto listas podem ter acesso mais disperso.
2. **Complexidade de Operações:**
    - Operações em matrizes (como acesso a elementos) são geralmente O(1), enquanto operações em listas podem ser O(n) dependendo da implementação.
3. **Overhead de Estrutura de Dados:**
    - Listas podem ter um overhead maior devido à necessidade de armazenar ponteiros/referências para os próximos elementos.

Análise de resultado com teste de hipótese entre os métodos:

Passo 1: Definir Hipóteses

- Hipótese Nula (H_0): Não há diferença significativa entre os tempos de execução ou resultados dos métodos Guloso e Randomizado.
- Hipótese Alternativa (H_1): Há diferença significativa entre os tempos de execução ou resultados dos métodos Guloso e Randomizado.

Passo 2: Escolher um Teste Estatístico

Para comparar os métodos, podemos usar:

1. Teste t de Student ou Teste de Wilcoxon para comparar os tempos de execução.
2. Teste Qui-Quadrado ou Teste Exato de Fisher para comparar os resultados (sucesso/falha).

Passo 3: Coletar e Organizar os Dados

Vamos organizar os dados das tabelas fornecidas, separando os métodos Guloso e Randomizado:

Tempos de Execução (em segundos)

Instância	Guloso (Matriz)	Guloso (Lista)	Randomizado (Matriz)	Randomizado (Lista)
1	0.597	18.302	0.562	16.722
2	0.618	7.974	0.576	8.038
3	1.643	8.564	1.735	251.682
4	0.346	0.607	0.314	0.638
5	0.402	2.290	0.405	2.338
6	1.481	128.601	1.504	130.724
7	0.379	0.930	0.342	0.931
8	0.381	1.585	0.376	1.630
9	0.403	3.552	0.380	3.572
10	0.417	5.295	0.418	5.414

Resultados (Sucesso ou Falha)

Instância	Guloso (Matriz)	Guloso (Lista)	Randomizado (Matriz)	Randomizado (Lista)
1	Sucesso	Sucesso	Sucesso	Sucesso
2	Falha	Falha	Falha	Falha
3	Falha	Falha	Falha	Falha
4	Sucesso	Sucesso	Sucesso	Sucesso
5	Sucesso	Sucesso	Sucesso	Sucesso
6	Sucesso	Sucesso	Sucesso	Sucesso
7	Sucesso	Sucesso	Sucesso	Sucesso
8	Sucesso	Sucesso	Sucesso	Sucesso
9	Falha	Falha	Falha	Falha
10	Sucesso	Sucesso	Sucesso	Sucesso

Passo 4: Aplicar Testes Estatísticos

1. Teste de Tempos de Execução

Podemos usar o Teste de Wilcoxon para comparar os tempos de execução entre os métodos Guloso e Randomizado. Como os dados não parecem seguir uma distribuição normal (devido à grande diferença de escala), o Teste de Wilcoxon (não paramétrico) é mais adequado.

- **Hipóteses:**
  - **H_0:** Não há diferença significativa entre os tempos de execução do Guloso e do Randomizado.
  - **H_1:** Há diferença significativa.
- **Resultado Esperado:**
  - Dada a similaridade nos tempos de execução entre os métodos, esperamos não rejeitar H_0, indicando que não há diferença significativa.

2. Teste de Resultados (Sucesso/Falha)

Podemos usar o Teste Qui-Quadrado ou o Teste Exato de Fisher para comparar as taxas de sucesso entre os métodos Guloso e Randomizado.

- **Hipóteses:**
  - H_0: Não há diferença significativa entre as taxas de sucesso do Guloso e do Randomizado.
  - H_1: Há diferença significativa.
- **Resultado Esperado:**
  - Como ambos os métodos obtiveram os mesmos resultados na maioria dos casos, esperamos não rejeitar H_0, indicando que não há diferença significativa nas taxas de sucesso.

Passo 5: Executar os Testes (Exemplo Simplificado)

Teste de Wilcoxon para Tempos de Execução

1. Calcular as diferenças entre os tempos de execução (Guloso - Randomizado).
2. Aplicar o teste de Wilcoxon nas diferenças.

- **Resultado Esperado:**
  - O teste deve mostrar um p-valor alto ($p > 0.05$), indicando que não há diferença significativa.

Teste Qui-Quadrado para Resultados

1. Criar uma tabela de contingência com os resultados (Sucesso/Falha) para Guloso e Randomizado.
2. Aplicar o teste Qui-Quadrado.

- **Resultado Esperado:**
  - O teste deve mostrar um p-valor alto ($p > 0.05$), indicando que não há diferença significativa nas taxas de sucesso.

Passo 6: Conclusão

1. **Tempos de Execução:**
    - Não há diferença significativa entre os tempos de execução dos métodos Guloso e Randomizado (não rejeitamos H_0).
2. **Resultados:**
    - Não há diferença significativa nas taxas de sucesso entre os métodos Guloso e Randomizado (não rejeitamos H_0).

Conclusões Finais:

A análise comparativa entre os métodos Guloso e Randomizado para resolver o problema da Árvore Geradora Mínima Generalizada (AGMG) permitiu identificar pontos importantes sobre o desempenho, eficiência e precisão de cada abordagem. Além disso, a comparação entre as implementações utilizando matriz e lista de adjacências destacou diferenças significativas em termos de tempo de execução. Abaixo, apresentamos as conclusões gerais do estudo:

---

1. Comparação entre Métodos Guloso e Randomizado

- **Tempo de Execução:**
    - Não houve diferença significativa entre os tempos de execução dos métodos Guloso e Randomizado. Ambos apresentaram desempenho semelhante, com tempos de execução comparáveis para as mesmas instâncias.
    - Isso sugere que, em termos de eficiência computacional, ambos os métodos são equivalentes para as instâncias testadas.
- **Resultados (Sucesso/Falha):**
    - Ambos os métodos obtiveram os mesmos resultados na maioria das instâncias, indicando que não há diferença significativa na capacidade de encontrar soluções válidas para o problema da AGMG.
    - Em casos onde um método falhou (por exemplo, instâncias 2, 3 e 9), o outro método também falhou, reforçando a similaridade entre as abordagens.
- **Conclusão:**
    - Os métodos Guloso e Randomizado são igualmente eficazes para resolver o problema da AGMG nas instâncias analisadas. A escolha entre eles pode depender de outros fatores, como a necessidade de aleatorização (no caso do método Randomizado) ou a simplicidade de implementação (no caso do método Guloso).

---

2. Comparação entre Implementações com Matriz e Lista

- **Tempo de Execução:**
    - A implementação utilizando matriz de adjacências foi significativamente mais rápida que a implementação com lista de adjacências em todas as instâncias.
    - Em casos de instâncias maiores (por exemplo, instância 6), a diferença de tempo foi drástica: a matriz levou ~1.481s, enquanto a lista levou ~2m8.601s.

    - Isso ocorre porque o acesso a elementos em uma matriz é mais eficiente (O(1)), enquanto o acesso em listas pode ser mais lento (O(n) no pior caso).
- **Resultados (Sucesso/Falha):**
    - Ambas as implementações (matriz e lista) obtiveram os mesmos resultados na maioria dos casos.
    - No entanto, em algumas instâncias (por exemplo, instância 1), a implementação com lista falhou em encontrar uma solução, enquanto a implementação com matriz obteve sucesso. Isso pode estar relacionado a limitações na estrutura de dados ou a erros de implementação.
- **Conclusão:**
    - A implementação com matriz é preferível devido ao seu desempenho superior, especialmente para instâncias maiores.
    - A implementação com lista pode ser útil apenas em cenários específicos onde o uso de memória é crítico, mas não é recomendada para problemas de grande escala.

---

3. Análise Geral das Instâncias

- As instâncias analisadas variaram em tamanho, densidade e complexidade, permitindo uma avaliação abrangente dos métodos e implementações.
- Instâncias com maior número de arestas e triângulos (por exemplo, instância 3: grid-yeast) apresentaram tempos de execução mais elevados, especialmente na implementação com lista.
- Instâncias com menor densidade (por exemplo, instância 5: airfoil1-dual) foram resolvidas rapidamente por ambos os métodos e implementações.

---

4. Recomendações Finais

- **Para Métodos:**
    - Utilize o método Guloso para problemas onde a simplicidade e a eficiência são prioritárias.
    - Utilize o método Randomizado quando a aleatorização pode ajudar a escapar de ótimos locais ou explorar soluções diversificadas.
- **Para Implementações:**
    - Prefira a implementação com matriz de adjacências para melhor desempenho em termos de tempo de execução.
    - Considere a implementação com lista apenas para instâncias pequenas ou quando a economia de memória é essencial.

---

5. Trabalhos Futuros

- Melhorias nos Algoritmos:
    - Implementar técnicas mais avançadas, como meta-heurísticas (por exemplo, Algoritmos Genéticos ou Colônia de Formigas), para melhorar a qualidade das soluções em instâncias complexas.
    - Explorar o uso de estruturas de dados mais eficientes, como Union-Find (DSU), para otimizar a verificação de ciclos.
- **Análise de Complexidade:**

    - Realizar uma análise teórica detalhada da complexidade dos métodos e implementações, considerando diferentes cenários de grafos (por exemplo, grafos esparsos vs. densos).
- **Testes em Instâncias Maiores:**
    - Avaliar o desempenho dos métodos e implementações em instâncias ainda maiores, com dezenas ou centenas de milhares de vértices e arestas.

Considerações Finais

Este estudo demonstrou que, embora os métodos Guloso e Randomizado sejam equivalentes em termos de resultados e tempo de execução, a escolha da estrutura de dados (matriz vs. lista) tem um impacto significativo no desempenho. A implementação com matriz é claramente superior em termos de eficiência, sendo a escolha recomendada para a maioria dos cenários. Para problemas mais complexos ou específicos, a combinação de métodos e estruturas de dados pode ser adaptada conforme necessário.