Subconjunto independente máximo

Descrição do problema

O problema do subconjunto indepentende máximo, busca percorrer um grafo, e encontrar o subconjunto de vértices de maior cardinalidade que respeite à seguinte propriedade: nenhum vértice do conjunto pode estar conectado a outro, ou seja, a solução é um subconjunto de vértices do grafo, com cardinalidade máxima e nenhuma aresta entre os vértices do subconjunto. No que toca o escopo deste trabalho, estamos considerando a versão de otimização deste problema, ou seja, encontrar o conjunto máximo de vértices que atenda à propriedade mencionada. Sendo assim, o problema é da classe NP-Difícil, ou seja não há algoritmo conhecido que seja capaz de encontrar uma solução ótima em tempo polinomial. Vale ressaltar que a verificação da solução ser ótima ou não para determinado grafo, também é NP-difícil, visto que seria necessário verificar todos as possibilidades de subconjuntos existentes no grafo, e verificar se a solução a ser testada de fato possui a maior cardinalidade. Tal verificação é extramente custosa e necessita de um algoritmo de força bruta para conseguir garantir de fato a solução ótima.

Descrição das instâncias

As intâncias utilizadas para testar os algoritmos propostos e desenvolvidos foram as instâncias:

- **SW-10000-**
- **SW-10000-**
- **SW-10000-**
- **SW-10000-**
- **SW-10000-**
- **SW-10000-**
- **SW-10000-**
- **SW-10000-**
- **SW-10000-**
- **SW-10000-**

Obtidas no seguinte dataset: https://networkrepository.com/rand.php

As instâncias seguem a estrutura de arquivos texto no estilo:

Nó de início Nó de destino

Nó de início Nó de destino

...

Por exemplo:

1 2

0 1

10 11

Vale ressaltar que apesar de denomidados como nós de início ou destino, o grafo é não direcionado e o programa não está adaptado para esta situação. Outro ponto relevante é que, como especificado, o tamanho mínimo das instâncias escolhidas deveria ser ¿= 5000 nós, considerando isso, todas as escolhidas possuem 10 mil nós.

Algoritmos propostos

Para solucionar este problema, 3 tipos de algoritmos precisaram ser desenvolvidos, sendo eles:

- **Guloso**
- **Guloso Randomizado**
- **Guloso Randomizado Reativo**

Algoritmo Guloso

Para o algoritmo guloso, que é a base dos demais, a lógica utilizada foi: utilizar o vértice de menor grau disponível como escolhido para entrar na solução, e remover seus vizinhos que já não são mais elegíveis para a solução, realizando as correções necessárias inerentes à essas remoções. Teoricamente, remover o vértice de menor grau implica em remover menos possíveis candidatos para o conjunto solução, que indiretamente vai aproximar a solução de ter mais vértices elegíveis e escolhidos, aproximando a solução do conjunto máximo almejado. Considerando isso, a lógica utilizada no desenvolvimento do algoritmo foi:

criação do conjunto solução (um conjunto vazio de vértices), que será incrementado a cada iteração do método, adicionando um vértice elegível na solução, como um guloso deve operar. Uma lista de grau crescente inicializada com o tamanho n do total de nós, construída ao percorrer todos os nós do grafo, já sendo ordenada enquanto é construída por uma implementação do método Insertion Sort. A lista é uma lista de vértices, para que o Id, do vértice de menor grau esteja de fácil acesso para as etapas seguinte do código. O cerne da reconstrução do grafo a cada iteração, que envolve remover o vértice selecionado para o conjunto

solução, todos os seus vizinhos que automaticamente deixaram de ser candidatos elegíveis, e atualização do grau dos nós vizinhos desses descartados como candidatos, foi implementada com o uso de uma Busca em profundidade (Deep First Search - DFS) adaptada.

- **Busca em Profundidade Adaptada**

A busca em profundidade (Deep First Search - DFS) é um algoritmo de exploração de grafos que pode ser implementado de forma recursiva ou usando a estrutura de dados pilha. A partir de um nó inicial, o algoritmo marca o nó como visitado e segue para um nó adjacente não visitado. Esse processo é repetido recursivamente, explorando os vértices adjacentes. Quando não há mais vértices adjacentes não visitados disponíveis, o algoritmo retrocede para o nó anterior e continua a busca, até retornar ao nó de início ou explorar todos os vértices acessíveis.

Considerando isso as seguintes adaptações foram feitas, considerando o problema a ser solucionado:

Não há necessidade de percorrer todos os nós não visitados do grafo a partir do nó inicial (que é o escolhido para entrar na solução), apenas os nós vizinhos de seus vizinhos, depois disso, os nós não visitados em diante não vão sofrer nenhuma alteração em seu estado original, e portanto não precisam ser visitados. Dessa forma a busca em profundidade se torna muito mais eficiente para o problema a ser tratado, visto que percorrer toda a componente conexa em que o nó seleciodado para a solução se encontra, que no pior dos casos, é o grafo todo, é um processo desnecessário. Considerando isso, a DFS foi adapatada para, além de marcar o nó como visitado, atribuir um nível a ele, iniciando em 0 e indo até 2, da seguinte forma e considerando as seguintes lógicas:

- **Nível 0:** O nó a ser inserido na solução, que vai ser removido do conjunto dos candidatos.
- **Nível 1:** Todo nó vizinho do selecionado, que vai ser removido do conjunto dos candidatos por não ser mais um candidato válido para a solução.
- **Nível 2:** Todo nó vizinho dos nós de nível 1, os quais vão perder 1 no seu atributo grau, e o respectivo nó pai de nível 1 como vizinho. Após um nó de nível 2 ser atingido, o algoritmo pode começar a retroceder após realizar as operações respectivas para esse nível, visto que não há necessidade de visitar nós vizinhos dos de nível 2, os quais seriam nível 3, pois eles não vão ser impactados de nenhuma maneira pelas remoções anteriores ou pela inserção do nó de nível 0 na solução. Sendo assim, o nível 2 é o critério de parada dessa busca.

Quando o ciclo da busca em profundidade for concluído, todas as operações de remoção terão sido corretamente realizadas, permitindo assim que a lista de grau seja reconstruída pelo Insertion Sort, e que um novo ciclo se inicie até que o conjunto de candidatos à solução seja vazio. A cada iteração do algoritmo guloso, o nó [0] da lista de grau será sempre o,

ou um dos, nó(s) de menor grau disponível, permitindo que a Busca em profundidade seja novamente chamada, sempre em O(1).

- **Observações:**

Como todo nó de grau 0 não possui vizinhos, todos estão automaticamente na solução e não interferem em nenhum outro vértice, tornando desnecessário o uso da Busca em profundidade e permitindo que no início do programa, todos os nós de grau 0 sejam inseridos na solução e removidos do conjunto de candidatos. Além disso, pequenas modificações no código foram feitas para permitir que o algoritmo opere tanto com a estrutura de matriz, quanto de lista, como exigido no escopo do trabalho. Ressalto que esse trabalho utiliza como base o código enviado e desenvolvido para o trabalho parte 2.

Algoritmo Guloso Randomizado

O Algoritmo guloso convencional, toma decisões locais ótimas a cada passo, mas exatamente por isso, é vítima de ótimos locais durante a execução do algoritmo, e pode acabar deixando passar uma solução que seria o ótimo global. Visando corrigir esse problema, o algoritmo guloso randomizado surge para incluir um elemento de aleatoriedade a cada passo de iteração do guloso, dando ao algoritmo uma versatilidade relativamente maior na escolha do nó em cada passo, buscando com isso diversificar o conjunto disponível para a solução em cada iteração, que diferente do algortimo guloso, não é determinístico.

Para o caso deste trabalho, o elemento de aleatoriedade do algoritmo guloso será inserido na escolha do nó de menor grau para incorporar a solução, e para gerar um conjunto de soluções com um número razoável de soluções geradas, o algoritmo vai iterar 10 vezes para cada instância, e salvar a melhor solução encontrada dentre essas 10.

Para o critério de aleatoriedade em si, a cada iteração os 5 nós de menor grau da lista serão escolhidos, todos os graus serão copiados para um vetor de 5 posições, transformados pela operação $1/(\text{grau do nó})^2$, depois disso, todas os 5 valores serão somados e cada posição do vetor será normalizada de acordo com o valor dessa soma, que por fim representarão a possibilidade de cada um desses valores de ser escolhido.Para o uso correto dessas probabilidades calculadas, é necessário calcular as probabilidades acumuladas. Isso significa que, após a normalização, a probabilidade acumulada para cada nó será a soma das probabilidades dos nós anteriores, de forma que a probabilidade acumulada de um nó i será a soma das probabilidades dos nós 1 até i, essas probabilidades acumuladas irão representar intervalos contínuos, sendo que a última probabilidade acumulada será igual a 1, garantindo que a soma das probabilidades seja 1. Esses valores substituem os valores do vetor de valores original. A seguir, uma função que gera números pseudo aleatórios em c++, proveniente da biblioteca #include ¡random¿ é chamada. Por fim, o valor gerado pela biblioteca random será comparado iterativamente com cada índice do vetor, e se o valor escolhido for menor ou igual ao da probabilidade comparada, significa que aquele é o índice sorteado, caso contrário irá

comparar com o próximo valor, retornando para o algoritmo de escolha do nó, qual índice do vetor com 5 nós que deve ser escolhido.

Com essa modificação, garantimos uma versatilidade maior para a construção das soluções, e que o processo não será mais determinístico, tornando útil o uso de várias iterações por instância, uma vez que, diferente do guloso, não irá gerar sempre o mesmo resultado.

Observação: Para que o problema não enfrente problemas de índices inválidos ou de posições nulas no vetor dos 5 escolhidos, o tamanho do vetor a ser calculado vai ser passado como parâmetro, podendo variar no intervalo(2,5) posições, caso possua um único elemento, a chamada da aleatoriedade se faz desncessária.

Algoritmo Guloso Randomizado Reativo

A variação reativa do algoritmo visa permitir que escolhas potencialmente ruins feitas pelo algoritmo possam ser reanalisadas com base em um critério predefinido e, se necessário, corrigidas. Considerando esse conceito, a seguinte lógica foi implementada:

A cada escolha realizada pelo sorteio de probabilidades, uma verificação adicional será feita caso os elementos finais do vetor (posições 4 ou 5) sejam escolhidos. A verificação segue o seguinte critério: se

$$\frac{\text{grau do vértice selecionado}}{\text{grau do vértice selecionado} + \text{grau do primeiro vértice do vetor}} > 0.7$$

então um novo sorteio será realizado, descartando a posição anteriormente escolhida. No entanto, apenas um sorteio adicional pode ser realizado por iteração, evitando a necessidade de recalcular as probabilidades repetidamente e reduzindo o custo computacional.

Essa abordagem mantém a aleatoriedade na escolha do nó, evitando que o algoritmo se torne determinístico em casos que atendam ao critério. Caso a escolha fosse sempre substituída pelo nó de menor grau, o algoritmo tenderia a um comportamento semelhante ao do guloso, que pode ficar preso em mínimos locais e não alcançar soluções globalmente melhores.

O critério adotado pode ser facilmente ajustado no código, permitindo aumentar o número de resorteios ou modificar o limiar de 70% para otimizar os resultados conforme o comportamento da instância analisada. Como diferentes tipos de grafos podem exigir calibrações distintas, essa flexibilidade ajuda a melhorar a qualidade das soluções.

Por fim, o algoritmo será executado em ciclos de 10 repetições por instância, armazenando a melhor solução encontrada. Para um estudo mais detalhado dos parâmetros, o código pode ser adaptado para exibir o número de vezes que um segundo sorteio foi necessário, possibilitando ajustes mais precisos para diferentes instâncias do problema.

Referência das instâncias:

@inproceedingsnr, title=The Network Data Repository with Interactive Graph Analytics and Visualization, author=Ryan A. Rossi and Nesreen K. Ahmed, booktitle=AAAI, url=https://networkreposit

Tabela 1: Análise de tempo de execução para o algoritmo Guloso

Instância	**Tempo(s) Matriz**	**Tempo(s) Lista**
Instância 1		
Instância 2		
Instância 3		
Instância 4		
Instância 5		
Instância 6		
Instância 7		
Instância 8		
Instância 9		
Instância 10		

Tabela 2: Análise de tempo de execução para o algoritmo Guloso Randomizado

Instância	**Tempo(s) Matriz**	**Tempo(s) Lista**
Instância 1		
Instância 2		
Instância 3		
Instância 4		
Instância 5		
Instância 6		
Instância 7		
Instância 8		
Instância 9		
Instância 10		

year=2015

Tabela 3: Análise de tempo de execução para o algoritmo Guloso Randomizado Reativo

Instância	**Tempo(s) Matriz**	**Tempo(s) Lista**
Instância 1		
Instância 2		
Instância 3		
Instância 4		
Instância 5		
Instância 6		
Instância 7		
Instância 8		
Instância 9		
Instância 10		

Tabela 4: Características das Instâncias

Instância	**Total de Nós**	**Cardinalidade da Melhor Solução**
Instância 1		
Instância 2		
Instância 3		
Instância 4		
Instância 5		
Instância 6		
Instância 7		
Instância 8		
Instância 9		
Instância 10		

Tabela 5: Resultados dos Métodos Implementados

Instância	**Guloso**	**t(s)Guloso**	**Randomizado**	**t(s)Randomizado**	**Reativo**	**t(s)Reativo**
Instância 1						
Instância 2						
Instância 3						
Instância 4						
Instância 5						
Instância 6						
Instância 7						
Instância 8						
Instância 9						
Instância 10						