Trabalho Prático 2
Algoritmos I - Desemprego

Wallace Eduardo Pereira - 2021032013

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

wallaceep@ufmg.br

1. Introdução

O problema proposto foi implementar um algoritmo que resolva um problema de matchings de duas formas diferentes, de forma gulosa e de forma exata. O objetivo do programa é resolver um problema específico na rede social LinkOut. A LinkOut é uma plataforma que tem como principal objetivo conectar usuários a vagas de emprego ideais. No entanto, a empresa está enfrentando uma crise de engajamento, com usuários reclamando de não serem recomendados para vagas, enquanto outros com qualificações semelhantes recebem diversas oportunidades. Para resolver o problema, o objetivo da solução é encontrar rapidamente pares únicos de usuários e vagas para maximizar o número de usuários atendidos. No entanto, é importante ressaltar que a solução gulosa muitas vezes é subótima, ou seja, não é a melhor solução possível.

2. Implementação

O programa foi desenvolvido na linguagem C++ e compilado pelo compilador G++ da GNU Compiler Collection.

3. Solução

3.1. Gulosa

O algoritmo greedyAlgorithm implementa um algoritmo guloso para encontrar correspondências entre usuários e trabalhos em um grafo representado por uma lista de adjacência. A função greedyAlgorithm() recebe como entrada um grafo representado pela variável adjList, que é uma lista de adjacência onde cada índice representa um usuário e os valores associados a esse índice representam os trabalhos disponíveis para esse usuário.

O algoritmo itera sobre cada trabalho disponível para o usuário atual (representado pelo loop for interno) e verifica se esse trabalho já foi atribuído a algum outro usuário. Se o trabalho ainda não tiver sido atribuído, ou seja, não estiver presente no vetor matchings, ele adiciona essa correspondência (usuário -> trabalho) ao vetor matchings e encerra o loop interno usando o comando break. Em resumo, o código implementa um algoritmo guloso que busca atribuir trabalhos a usuários, selecionando o primeiro trabalho disponível para cada usuário que ainda não tenha sido atribuído a outro usuário.

3.2. Exata

O código implementa o algoritmo de Ford-Fulkerson para determinar o fluxo máximo em um grafo direcionado com capacidades nas arestas, usando a busca em profundidade para encontrar caminhos aumentantes. Considerando que o fluxo máximo é também o máximo exato de matchings possíveis para o problema.

4. Modelagem

Para solucionar o problema de recomendação de vagas na rede social LinkOut, foi adotada uma abordagem baseada em um grafo bipartido. Nessa modelagem, os usuários são representados por um conjunto de vértices em uma partição do grafo, enquanto as vagas de emprego são representadas por outro conjunto de vértices na outra partição.

Cada vértice do grafo representa um usuário ou uma vaga de emprego. Para mapear os vértices, foi utilizado um identificador único para cada usuário e vaga, permitindo uma identificação precisa de cada elemento. Como foram feitas duas implementações, a modelagem foi dividida para explicar cada uma delas:

- **Gulosa:** A definição de um algoritmo guloso é fazer escolhas que pareçam ser as melhores em cada momento, com base em critérios locais, sem considerar o impacto futuro dessas escolhas.

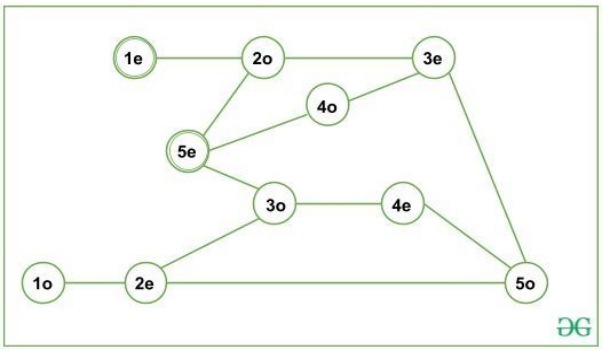
  A modelagem para a abordagem gulosa foi primeiro mapear as entradas das arestas para um *unordered_map<string, int> input;* de forma que um mesmo vértice não seria adicionado mais de uma vez e seu índice seria armazenado. Após isso, as arestas foram armazenadas com base nesses índices, foi adicionado na lista de adjacência do tipo *vector<vector<int>>* adjList; que foi utilizada no algoritmo guloso de fazer matchs.

- **Exata:** A solução exata, diferente da gulosa, é aquela que encontra a solução ótima para um problema, considerando todas as possibilidades e avaliando todas as soluções potenciais. Em outras palavras, uma solução exata garante que a solução encontrada é a melhor possível, com base em critérios pré-definidos.

  Para que a solução exata se adequasse ao algoritmo de Ford-Fulkerson, foi necessário fazer algumas modificações, tais como adicionar dois vértices, *source*, que é ligado a todos os vértices do tipo *userName,* e *sink,* que recebe aresta de todos os vértices do tipo *jobName*. Ademais, a aresta foi adequada para uma de fluxo, armazenando também o fluxo e a capacidade dos vértices, transformando assim em um grafo com fluxo preparado para ser utilizado no algoritmo de Ford-Fulkerson, que retorna a solução exata para o problema.

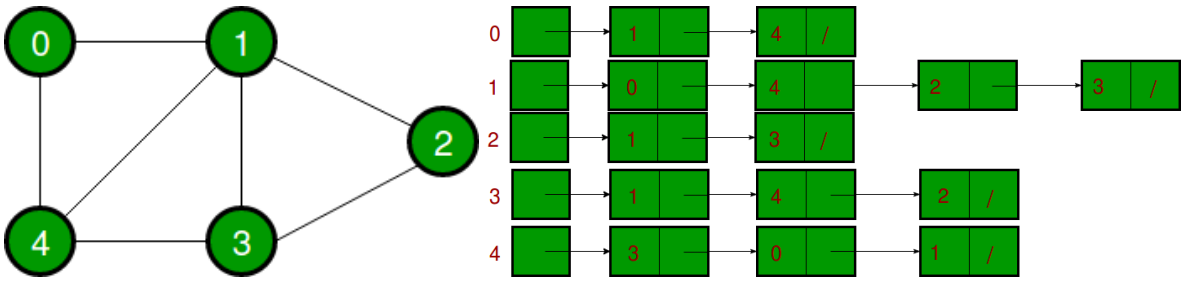
  Além disso, para que a lista de adjacência fosse criada, foi necessário utilizar uma *struct Edge*, como auxiliar para armazenar as informações das arestas, tais como sua capacidade e seu fluxo.

Além disso, o grafo é bipartido, ou seja, sempre possui uma aresta ligando um vértice de um conjunto conectada a um vértice de outro conjunto.

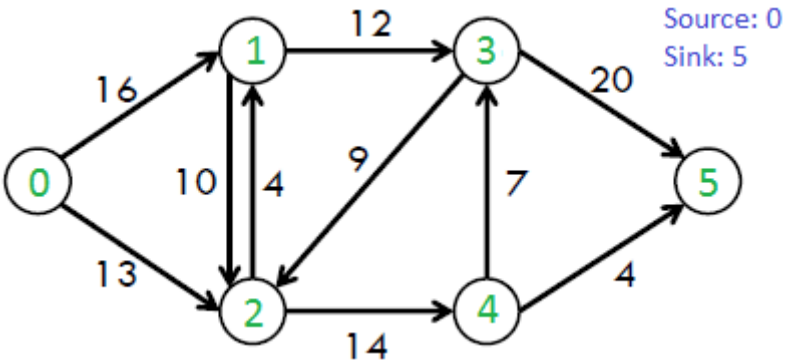


4.1. Estruturas de Dados

Ao armazenar o grafo, **para o algoritmo guloso**, foi utilizada uma lista de adjacência, uma forma simples de representar um grafo como uma lista de vértices, em que cada vértice tem uma lista de seus vértices adjacentes, como segue o exemplo da imagem a seguir:



Já **para o algoritmo exato**, também foi utilizada uma lista de adjacência, mas também foi transformado em um grafo de fluxo, que possui capacidade e fluxo nas arestas, como segue o exemplo da imagem a seguir:



Para representar a lista de adjacência, no guloso foi utilizada a biblioteca padrão de C++ <vector>, de forma que foi primeiro armazenado um *unordered_map,* mapeando as strings de entrada para inteiros. Depois foi salvo em um "vector de vector de inteiros", ou seja, um vetor que armazena os vértices, e para cada vértice um vetor que armazena um número inteiro, para representar o vértice sendo armazenado.

1.1. Classes

A resolução do problema possui somente uma classe, a classe Graph. A classe graph possui como atributos e funções:

- numV: valor inteiro que armazena o número de usuários e empregos, que representa o número de vértices do grafo.
- adjList: vetor que armazena outro vetor, e esse segundo vetor armazena um valor de inteiro que representa a ligação entre um usuário e um job, ou seja, o segundo vetor na prática armazena um inteiro que representa os vértices adjacentes a esse vértice.
- input: representa o grafo em um map, mapeando as entradas de string e atribuindo inteiros identificadores.
- inputE: mesma coisa do input.
- graph: armazena o grafo de fluxo com as arestas Edge auxiliando.
- Graph: construtor.
- verify: verifica se o vértice ja foi adicionado antes de adiciona-lo e liga-lo ao source ou sink.
- dfs: algoritmo de busca no grafo, verifica se há caminhos possíveis
- fordFulkerson: encontra o fluxo máximo do grafo, retornando a solução ótima
- addEdge: adiciona as arestas no grafo para o algoritmo guloso
- addEdgeExactly: adiciona as arestas no grafo e as prepara para rodar no ford-fulkerson
- greedy algorithm: implementa um algoritmo guloso para encontrar correspondências entre usuários e trabalhos em um grafo representado por uma lista de adjacência.

2. Análise de Complexidade

A complexidade de tempo do algoritmo guloso apresentado é O(|E| log |V|), onde |E| é o número total de arestas (ou trabalhos) e |V| é o número total de vértices (ou usuários). Isso ocorre porque, para cada vértice, é feita uma ordenação da lista de trabalhos disponíveis, que tem tamanho proporcional ao número de arestas conectadas a esse vértice. A ordenação é realizada com uma complexidade de O(n log n), onde n é o tamanho da lista de trabalhos. Portanto, a complexidade de tempo do algoritmo é O(|E| log |V|). Quanto à complexidade de espaço, o algoritmo utiliza um vetor matchings para armazenar as correspondências entre usuários e trabalhos. O tamanho desse vetor será no máximo igual ao número total de usuários, ou seja, |V|. Portanto, a complexidade de espaço é O(|V|). Em resumo, a complexidade de tempo do algoritmo é O(|E| log |V|) e a complexidade de espaço é O(|V|).

A complexidade de tempo do algoritmo de Ford-Fulkerson, considerando as funções dfs() e fordFulkerson() juntas, depende do número de arestas e do número de vértices no grafo.

Para a função dfs(), a complexidade de tempo é proporcional ao número de arestas do grafo. Em cada chamada recursiva, o algoritmo explora todas as arestas saindo de um vértice, verificando se a aresta não foi visitada e se ainda possui capacidade residual. Portanto, a complexidade de tempo da função dfs() é O(E), onde E é o número de arestas.

Na função fordFulkerson(), o algoritmo executa um loop enquanto houver caminhos aumentantes. A cada iteração, é chamada a função dfs() para encontrar um caminho aumentante. Portanto, o número máximo de iterações é limitado pelo número de caminhos aumentantes existentes no grafo. No pior caso, a complexidade de tempo é O(C * E), onde C é o valor máximo do fluxo no grafo. No entanto, a complexidade de tempo média do algoritmo de Ford-Fulkerson é geralmente menor que O(C * E) na prática.

Em relação à complexidade de espaço, o algoritmo utiliza um vetor visited na função dfs() para marcar os vértices visitados. O tamanho desse vetor é igual ao número de vértices no grafo. Portanto, a complexidade de espaço é O(V), onde V é o número de vértices.

Em resumo, a complexidade de tempo do algoritmo de Ford-Fulkerson é geralmente menor que O(C * E), e a complexidade de espaço é O(V), onde C é o valor máximo do fluxo, E é o número de arestas e V é o número de vértices.

5. Referências

https://www.geeksforgeeks.org/greedy-algorithms/
https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/
https://www.geeksforgeeks.org/graph-and-its-representations/
https://www.geeksforgeeks.org/shortest-path-with-even-number-of-edges-from-source-to-destination/amp/