UNIVERSIDADE ESTADUAL PAULISTA – UNESP

Algoritmos Elementares de Grafos

Vitor Hugo Parras

Presidente Prudente
2023

Sumário

1 Introdução

1.1 Apresentação do assunto

Os algoritmos de grafos são um conjunto fundamental de algoritmos na ciência da computação e na teoria dos grafos. Eles são usados para resolver problemas complexos e versáteis em várias aplicações, desde redes de computadores até biologia computacional e inteligência artificial. Este relatório fornecerá uma visão detalhada dos principais algoritmos de grafos, com foco em algoritmos elementares de grafos, Árvores Espalhadas Mínimas (Kruskal e Prim) e os conceitos de corretude e complexidade.

1.2 Importância dos algoritmos de grafos

Os algoritmos de grafos são vitais para a eficiência e eficácia de muitas aplicações e sistemas modernos. Eles são usados para resolver problemas que envolvem a estrutura ou rede de um problema, como encontrar o caminho mais curto entre dois pontos em um mapa, determinar a rota mais eficiente para a entrega de pacotes, ou analisar as redes sociais para identificar conexões importantes. Compreender esses algoritmos e seu funcionamento é fundamental para qualquer cientista da computação ou profissional de TI.

1.3 Resumo dos tópicos abordados

Este relatório abordará inicialmente os algoritmos elementares de grafos, a Busca em Largura (Breadth-First Search - BFS) e a Busca em Profundidade (Depth-First Search - DFS). Em seguida, discutirá as Árvores Espalhadas Mínimas (Minimum Spanning Trees - MST), incluindo os algoritmos de Kruskal e Prim, dois métodos populares para encontrar MSTs. O relatório também se aprofundará no entendimento dos conceitos de corretude e complexidade, fundamentais para a análise de algoritmos. O objetivo final é fornecer um entendimento abrangente desses algoritmos de grafos e de como eles são aplicados na resolução de problemas reais.

2 Algoritmos Elementares de Grafos

Os algoritmos elementares de grafos são a base da manipulação e análise de grafos. Nesta seção, discutiremos os dois algoritmos fundamentais de grafos: a Busca em Largura (Breadth-First Search - BFS) e a Busca em Profundidade (Depth-First Search - DFS). Esses algoritmos são usados para percorrer ou pesquisar um grafo na forma mais eficiente possível.

2.1 Busca em Largura (Breadth-First Search - BFS)

A Busca em Largura é um algoritmo de grafos que explora todos os vértices de um grafo a uma distância 'd' antes de explorar os vértices a uma distância 'd + 1'. O algoritmo começa a partir de um vértice raiz e explora todos os vértices adjacentes a ele antes de se mover para os próximos níveis.

2.1.1 Explicação do algoritmo

O algoritmo BFS começa a partir de um vértice raiz (que pode ser qualquer vértice no grafo) e explora todos os vértices adjacentes a ele. Em seguida, para cada um desses vértices mais próximos, ele explora os vértices não visitados adjacentes a eles, e assim por diante. Isso continua até que todos os vértices no grafo tenham sido visitados. O algoritmo BFS pode ser usado para encontrar o caminho mais curto de um vértice a todos os outros vértices em um grafo não ponderado.

2.1.2 Exemplos de uso

A BFS é comumente usada em uma variedade de aplicações, incluindo a localização de todos os vértices a uma distância 'k' de um vértice em um grafo, a verificação se um grafo é conectado, a determinação do caminho mais curto entre dois vértices e muitos outros.

2.1.3 Complexidade de tempo e espaço

A complexidade de tempo do algoritmo BFS é O(V + E), onde V é o número de vértices no grafo e E é o número de arestas. A complexidade do espaço é O(V) para armazenar informações sobre os vértices visitados e para manter uma fila dos vértices a serem visitados.

2.2 Busca em Profundidade (Depth-First Search - DFS)

A Busca em Profundidade é outro algoritmo fundamental para percorrer ou pesquisar um grafo. Diferentemente do BFS, que explora todos os vértices em um nível antes de se mover para o próximo, o DFS explora o grafo o mais profundamente possível antes de retroceder.

2.2.1 Explicação do algoritmo

O DFS começa em um vértice raiz e explora o máximo possível ao longo de cada ramo antes de retroceder. Isso é feito visitando um vértice adjacente

não visitado e continuando a fazê-lo até que não haja mais vértices não visitados ao longo do ramo atual. Então, o algoritmo retrocede e repete o processo para qualquer ramo não explorado. O DFS pode ser implementado de forma recursiva ou usando uma pilha para simular a recursão.

2.2.2 Exemplos de uso

O DFS é frequentemente usado em situações em que é útil visitar cada vértice de um grafo. Isso inclui encontrar um caminho entre dois vértices, detectar ciclos em um grafo, verificar se um grafo é bipartido, entre outros usos.

2.2.3 Complexidade de tempo e espaço

A complexidade de tempo do algoritmo DFS é O(V + E), onde V é o número de vértices no grafo e E é o número de arestas. A complexidade do espaço é O(V) para armazenar informações sobre os vértices visitados e para manter uma pilha dos vértices a serem visitados.

3 Árvores Espalhadas Mínimas

As Árvores Espalhadas Mínimas (Minimum Spanning Trees - MSTs) são uma parte crucial da teoria dos grafos e têm aplicações práticas significativas, especialmente na concepção de redes eficientes. Uma MST de um grafo é uma árvore que abrange todos os vértices e tem o peso mínimo possível, ou seja, a soma dos pesos de todas as arestas é a menor possível. Existem vários algoritmos para encontrar a MST de um grafo. Nesta seção, focaremos nos algoritmos de Kruskal e de Prim.

3.1 Algoritmo de Kruskal

O algoritmo de Kruskal é um algoritmo popular para encontrar a MST de um grafo. Ele é chamado de "algoritmo guloso"porque faz a escolha que parece ser a melhor naquele momento. No caso de Kruskal, sempre escolhe a aresta de menor peso que não forma um ciclo com as arestas já incluídas na MST.

3.1.1 Explicação do algoritmo

O algoritmo de Kruskal começa classificando todas as arestas do grafo em ordem crescente de seus pesos. Em seguida, ele adiciona a aresta de menor peso à MST, desde que a aresta não forme um ciclo com as arestas já presentes

na MST. O algoritmo continua adicionando a próxima aresta de menor peso que não forma um ciclo até que a MST contenha todos os vértices do grafo.

3.1.2 Exemplos de uso

O algoritmo de Kruskal é usado em uma variedade de aplicações, como na concepção de redes de computadores e telecomunicações, onde é desejável conectar todos os nós com o mínimo custo possível. Outras aplicações incluem a construção de estradas, redes de distribuição de água, e até mesmo em algoritmos de aprendizado de máquina para encontrar clusters de dados.

3.1.3 Prova de corretude

A corretude do algoritmo de Kruskal pode ser provada usando o corte de propriedades da MST. Um corte de um grafo divide o grafo em duas partes disjuntas. A propriedade do corte afirma que a aresta de peso mínimo que atravessa o corte deve pertencer à MST. O algoritmo de Kruskal sempre escolhe a aresta de peso mínimo que não forma um ciclo, o que está de acordo com a propriedade do corte, e, portanto, sempre produz uma MST.

3.1.4 Análise da complexidade

A complexidade de tempo do algoritmo de Kruskal é O(E log E), onde E é o número de arestas no grafo. Isso é porque o algoritmo precisa classificar todas as arestas, o que leva O(E log E) tempo. A complexidade do espaço é O(E + V) para armazenar as arestas e os vértices.

3.2 Algoritmo de Prim

O algoritmo de Prim é outro algoritmo popular para encontrar a MST de um grafo. Assim como o algoritmo de Kruskal, o algoritmo de Prim é um algoritmo guloso, mas a estratégia para construir a MST é diferente.

3.2.1 Explicação do algoritmo

O algoritmo de Prim começa selecionando um vértice arbitrário e adicionando-o à MST. Em seguida, em cada interação, escolhe a aresta de menor peso que conecta um vértice já incluído na MST a um vértice ainda não incluído. Este processo é repetido até que todos os vértices tenham sido incluídos na MST.

3.2.2 Exemplos de uso

O algoritmo de Prim é usado em uma variedade de aplicações, semelhantes às do algoritmo de Kruskal. Ele é particularmente útil quando se tem um grafo denso, onde o número de arestas é muito maior que o número de vértices, pois sua eficiência pode ser aumentada usando estruturas de dados apropriadas.

3.2.3 Prova de corretude

A corretude do algoritmo de Prim pode ser provada usando a propriedade de corte da MST, assim como no algoritmo de Kruskal. O algoritmo de Prim sempre escolhe a aresta de menor peso que conecta um vértice na MST a um vértice fora dela, o que está de acordo com a propriedade do corte. Portanto, o algoritmo de Prim sempre produz uma MST.

3.2.4 Análise da complexidade

A complexidade de tempo do algoritmo de Prim é O(E log V) com uma fila de prioridade binária, ou O(E + V log V) com uma fila de prioridade Fibonacci. Aqui, E é o número de arestas e V é o número de vértices. A complexidade do espaço é O(V) para armazenar os vértices e a fila de prioridade.

4 Corretude e Complexidade

A corretude e a complexidade são duas medidas importantes na análise de algoritmos. A corretude se refere à ideia de que um algoritmo faz o que é suposto fazer. A complexidade de um algoritmo, por outro lado, é uma medida de quanto tempo e espaço ele requer em relação ao tamanho da entrada.

4.1 Corretude dos Algoritmos de Grafos

A corretude de um algoritmo de grafo é estabelecida ao provar que o algoritmo atinge seu objetivo pretendido. Para algoritmos de MST, como Kruskal e Prim, a corretude é geralmente provada usando a propriedade do corte da MST.

4.1.1 Propriedade do Corte

A propriedade do corte para MSTs é uma importante ferramenta para provar a corretude dos algoritmos de MST. Ela afirma que, para qualquer corte de um grafo que respeite a MST (ou seja, que não atravesse nenhuma aresta já

incluída na MST), a aresta de menor peso que atravessa o corte deve pertencer à MST. Ambos, Kruskal e Prim, selecionam sempre a aresta de menor peso que não forma um ciclo com as arestas já na MST, que é precisamente a condição estabelecida pela propriedade do corte.

4.1.2 Prova de Corretude para Kruskal e Prim

A prova de corretude para os algoritmos de Kruskal e Prim é semelhante, uma vez que ambos os algoritmos baseiam-se na propriedade do corte. Cada algoritmo começa com uma MST vazia e adiciona arestas uma de cada vez. Em cada passo, a aresta adicionada é a de menor peso que atravessa algum corte que respeita a MST atual. Portanto, cada aresta adicionada satisfaz a propriedade do corte, e assim a corretude dos algoritmos de Kruskal e Prim é estabelecida.

4.2 Complexidade dos Algoritmos de Grafos

A complexidade de um algoritmo de grafo é uma medida de quanto tempo e espaço o algoritmo requer em relação ao tamanho da entrada. Para algoritmos de grafos, a complexidade de tempo geralmente depende do número de vértices e arestas no grafo, enquanto a complexidade do espaço geralmente depende do número de vértices.

4.2.1 Complexidade de Tempo

A complexidade de tempo dos algoritmos de grafos é geralmente expressa em termos do número de vértices (V) e o número de arestas (E). Por exemplo, os algoritmos BFS e DFS têm uma complexidade de tempo de O(V + E), pois cada vértice e cada aresta são visitados uma vez. O algoritmo de Kruskal tem uma complexidade de tempo de O(E log E) ou O(E log V) para a operação de classificação das arestas, enquanto o algoritmo de Prim tem uma complexidade de tempo de O(E log V) com uma fila de prioridade binária ou O(E + V log V) com uma fila de prioridade Fibonacci.

4.2.2 Complexidade de Espaço

A complexidade do espaço dos algoritmos de grafos é geralmente expressa em termos do número de vértices. Por exemplo, os algoritmos BFS e DFS têm uma complexidade de espaço de O(V), pois precisam armazenar informações sobre quais vértices foram visitados. O algoritmo de Kruskal tem uma complexidade de espaço de O(E + V) para armazenar as arestas e os vértices, enquanto o algoritmo de Prim tem uma complexidade de espaço de O(V) para armazenar os vértices e a fila de prioridade.

4.2.3 Otimização de Algoritmos de Grafos

A otimização dos algoritmos de grafos é um campo de estudo em constante evolução. A ideia principal é melhorar a eficiência dos algoritmos de grafos em termos de tempo e espaço.

Existem várias maneiras de otimizar os algoritmos de grafos. Uma estratégia comum é usar estruturas de dados eficientes. Por exemplo, no algoritmo de Prim, a escolha da estrutura de dados para a fila de prioridade pode ter um impacto significativo na eficiência do algoritmo. Usar uma matriz ou uma lista vinculada como uma fila de prioridade resulta em uma complexidade de tempo de $O(V^2)$, enquanto o uso de uma fila de prioridade binária ou Fibonacci reduz a complexidade de tempo para O(E log V) e O(E + V log V), respectivamente.

Outra estratégia para otimizar algoritmos de grafos é reduzir o número de operações realizadas. Por exemplo, no algoritmo de Kruskal, uma otimização comum é usar uma estrutura de dados chamada "conjunto disjunto"para verificar se a adição de uma aresta formaria um ciclo. Esta estrutura de dados permite que esta verificação seja feita em tempo quase constante, o que melhora significativamente a eficiência do algoritmo.

Finalmente, muitos algoritmos de grafos podem ser otimizados através de técnicas de programação paralela e distribuída. Estas técnicas permitem que o trabalho de processar o grafo seja dividido entre vários processadores ou máquinas, o que pode levar a melhorias significativas na eficiência para grafos grandes.

5 Conclusão

Os algoritmos de grafos são ferramentas essenciais em ciência da computação e em muitas outras disciplinas. Neste relatório, discutimos os algoritmos de busca em largura e profundidade, que são a base para muitos algoritmos de grafos mais complexos. Também discutimos os algoritmos de Kruskal e Prim para encontrar árvores de expansão mínima, que têm aplicações em uma ampla variedade de campos.

Além disso, abordamos a corretude e a complexidade dos algoritmos de grafos. A corretude é essencial para garantir que um algoritmo faz o que é suposto fazer, enquanto a complexidade é uma medida de quão eficiente é um algoritmo. Também exploramos algumas das maneiras pelas quais os algoritmos de grafos podem ser otimizados, incluindo o uso de estruturas de dados eficientes e técnicas de programação paralela e distribuída.

Embora este relatório tenha fornecido uma visão geral dos algoritmos de grafos, há muito mais para explorar neste campo. Algoritmos de grafos são um tópico ativo de pesquisa, com novos algoritmos e otimizações sendo desenvolvidos regularmente. Além disso, à medida que mais dados se tornam disponíveis e

as aplicações de algoritmos de grafos se expandem, é provável que a importância dos algoritmos de grafos continue a crescer.

Referências

[1] Carlos Eduardo Ferreira. *Grafos: teoria, algoritmos e aplicações*. Instituto de Informática, UFRGS, 1st edition, 2014.

[2] Luis Pinho and João F. Oliveira. *Grafos: Uma Introdução*. LTC, 2nd edition, 2018.

[3] Cássio Farias. *Grafos e Algoritmos*. Casa do Código, 1st edition, 2017.

[4] Nelma Moreira Carvalho and Edward Hermann Haeusler. *Grafos: conceitos, algoritmos e aplicações*. Elsevier, 1st edition, 2012.

[5] Eraldo Silveira Gouveia and Reinaldo Silva Lemos. *Teoria dos Grafos*. Editora Livraria da Física, 1st edition, 2009.

[6] Brasil Escola. Grafos. https://brasilescola.uol.com.br/matematica/grafos.htm. Recuperado em 29 de abril, 2023.

[7] InfoEscola. Teoria dos grafos. https://www.infoescola.com/matematica/teoria-dos-grafos/. Recuperado em 29 de abril, 2023.

[8] Wikipedia. Algoritmo de kruskal. https://pt.wikipedia.org/wiki/Algoritmo_de_Kruskal. Recuperado em 29 de abril, 2023.

[9] Wikipedia. Algoritmo de prim. https://pt.wikipedia.org/wiki/Algoritmo_de_Prim. Recuperado em 29 de abril, 2023.

[10] DevMedia. Introdução à teoria dos grafos. https://www.devmedia.com.br/introducao-a-teoria-dos-grafos/29539. Recuperado em 29 de abril, 2023.