Algoritmo de Dijkstra: Implementações para Abordagens MinMax e MaxMin

Gabriel de Cortez Mourão[1], **Mateus Fernandes Barbosa**[1], **Victor Ferraz de Moraes**[1]

[1] Pontifícia Universidade Católica de Minas Gerais (PUC Minas)
Caixa Postal 30535-901 – Belo Horizonte – MG – Brazil

[2]Instituto de Ciências Exatas e Informática
ICEI

Abstract. *In this article, we conducted a detailed exploration of Dijkstra's algorithm, a classic graph algorithm developed by Edsger W. Dijkstra in 1959, which calculates the shortest path between vertices in a graph. As discussed, the algorithm works by updating the known shortest distances from a source vertex to its neighbors in the graph, with applications in navigation systems, route optimization, and more. We covered the implementation, complexity, and limitations of the algorithm, highlighting its inability to handle graphs with negative edge weights. Additionally, we presented two modified variants of Dijkstra's algorithm: Min-Max and Max-Min. The Min-Max version aims to minimize the maximum weight on a path between two vertices, used in situations where the goal is to reduce bottlenecks on routes. The Max-Min version optimizes the maximum of the smallest edges, applied in cases like traffic route optimization to reduce congestion. Both versions use priority queues to ensure efficient exploration of vertices and edges.*

Resumo. *Neste artigo, fizemos uma exploração detalhada do algoritmo de Dijkstra, um algoritmo clássico de grafos desenvolvido por Edsger W. Dijkstra em 1959, que calcula o caminho mais curto entre vértices em um grafos. Como discutido, o algoritmo opera atualizando as menores distâncias conhecidas de um vértice de origem para os vizinhos no grafo, tendo aplicações em sistemas de navegação, otimização de rota etc. Discutimos a implementação, complexidade e limitações do algoritmo, chamando a atenção para a incapacidade do algoritmo de lidar com grafos que contêm arestas de peso negativo. Além disso, apresentamos duas variantes modificadas do algoritmo de Dijkstra: Min-Max e Max-Min. A versão Min-Max visa minimizar o peso máximo em um caminho entre dois vértices, usado em situações em que a meta é reduzir os gargalos nas rotas. A versão Max-Min otimiza o máximo das menores arestas, aplicada a situações como a otimização de rota de tráfego para reduzir o congestionamento. Ambas as versões empregam filas de prioridade para garantir a exploração eficiente de vértices, arestas.*

1. Introdução

O problema de encontrar o menor caminho entre dois pontos em um grafo é uma questão fundamental em diversas áreas, como ciência da computação, engenharia de redes, transporte e otimização logística. Este artigo procura explorar não apenas o algoritmo de grafos de *Dijkstra* para encontrar uma solução precisa para o problema, mas suas características, complexidades e implementações. Uma visão mais abrangente do algoritmo também é tratada com sua modificação para abordagens de diferentes problemas como os problemas de maior caminho mínimo e menor caminho máximo

2. *Dijkstra*

Dijkstra, desenvolvido pelo cientista Edgsget W. Dijkstra, é um algoritmo de grafos para encontrar o menor caminho entre dois vértices. Foi desenvolvido em 1959 e foi o primeiro algoritmo a abordar esse problema. Muitos problemas de distância podem ser abstraídos em grafos para computar o melhor caminho possível entre dois pontos, como ao calcular rotas de navegação de veículos.

A aplicação pode ser usada para encontrar a menor distância de um vértice de origem com os demais vértices, ou a menor distância de um vértice de origem para outro de destino. Sua implementação se baseia na atualização da distância do vértice de origem para os vizinhos do vértice atual da busca, em que:

- `Inicialização das distâncias`: No inicio a distância inicial de todos os vértices da origem será infinita, definindo que os vértices não são inatingíveis. Logo após, a distância da origem com ela mesma é definida como 0.
- `Exploração de vértices`: A busca começa na origem. Caso exista um melhor caminho do vértice atual para o seus vizinhos, atualizamos a distância do vizinho para refletir este melhor caminho.
- `Escolha de vértices`: O algoritmo assegura que cada vértice precise analisar seus vizinhos uma única vez. Para garantir isto, é sempre escolhido o vértice de menor distância registrada para a origem que ainda não foi explorada, pois é possível assegurar que, caso não exista arestas de peso negativo, os demais vértices terão distância maior ou igual a este vértice, e logo não podem encontrar um caminho de menor distância
- `Terminação do algoritmo`: O processo continua até que o objetivo do algoritmo seja atingido:
  - Para encontrar a menor distância de todos os vértices até a origem, é preciso fazer busca com todos os vértices. Neste caso, temos uma complexidade de
  - Para encontrar a menor distância entre dois vértices, o algoritmo pode ser encerrado ao iniciar a busca no vértice de destino, pois é possível concluir que a melhor rota até este vértice foi encontrada

A sua garantia de escolher a menor distância para cada vértice em que está efetuando a busca não será garantida caso existam arestas negativas, restringindo o escopo dos grafos aceitáveis para o algoritmo.

A complexidade e custo médio do algoritmo pode variar dependendo da facilidade de acesso do vértice de menor distância e de encontrar os vizinhos de cada vértice.

2.1. Estruturas de Dados

A implementação mais simples do algoritmo usa uma Matriz de Adjacência, e uma estrutura auxiliar de lista de vértices já visitados, em que:

- `Custo de encontrar menor distância`: É necessário percorrer todos os vértices ainda não explorados, através de uma lista de elementos auxiliar que contém apenas estes elementos, ou que identifica todos os vértices como explorados ou não explorados. De forma geral, o custo de pesquisa nesta lista será de $O(|V|)$.
- `Acesso dos vizinhos`: Para verificar todos os vizinhos de um vértice da matriz, é conferido toda a coluna da matriz de adjacência para um vértice v, tendo custo constante $|V|$. Logo, sua complexidade será $O(|V|)$

O custo de $O(|V|)$ em cada iteração para no máximo $|V|$ iterações representa um custo de complexidade $O(|V^2|)$.

Outra implementação usa uma estrutura de lista de tamanho $|V|$, cada uma delas com uma lista que representa todas as arestas vizinhas ao vértice. A simples mudança para esta estrutura permite com que a busca por vértices vizinhos seja certeira. Logo, ao máximo a quantidade de arestas exploradas será de tamanho $|E|$, resultando em uma complexidade total de $O(|V|^2 + |E|)$, ou seja, a quantidade de arestas também pode impactar na performance do algoritmo. De fato, quanto mais esparso o grafo é, melhor será o desempenho quando comparado ao custo fixo da estrutura de Matriz de Adjacência.

Para reduzir o custo de encontrar a menor distância, a estrutura auxiliar de lista pode ser substituída por uma fila de prioridade de *min-heap*, que possui apenas os vértices vizinhos aos visitados, com complexidade de pesquisa, remoção e adição de $O(log|V|)$. Para cada vez que um vértice é explorado, ele é removido da fila, e cada alteração na distância mínima de um vértice para a origem o adiciona na fila. A ordenação do vértice *min-heap* normalmente é baseada na distância mínima recebida.

Neste caso, toda busca pelo vértice de menor distância e acesso a ele se tratam de uma pesquisa e remoção de complexidades totais de $O(|V|log|V|)$ e o custo de adicionar uma aresta é de $O(|E|log|V|)$. Logo, a complexidade resultado do algoritmo será de $O((|V| + |E|)log|V|)$. Por ser uma implementação simples e eficiente, os algoritmos deste artigo se baseiam nela.

2.2. Limitações do algoritmo

O algoritmo possui complexidade elevada, e por isto nem sempre é adequado para soluções velozes em grafos de grande volume de vértices e arestas. Outros algoritmos que se baseiam em *Dijkstra* com menor tempo de execução, como o A*, são usados como base para grandes aplicações, como *"Google Maps"* e *"Waze"*. Essa implementação, embora possua a chance de encontrar resultados considerados ruins ou imprecisos, na grande maioria das vezes encontra rotas que atendem as necessidades da maior parte dos usuários com um tempo de execução menor que o necessário para encontrar a melhor rota possível, utilizando o *Dijkstra*.

Outra restrição problemática é a necessidade de todos os pesos das arestas serem positivos, especialmente em cenários onde os custos associados às arestas podem ser negativos, como em rotas com descontos, promoções ou incentivos. Em situações em que

arestas podem ter valores negativos, o algoritmo de *Dijkstra* pode produzir resultados incorretos, pois não espera que o valor da distância para a origem de um vértice abaixe ainda mais após já ser explorada. Outro problema seria com ciclos negativos, em que os valores diminuem para cada loop infinitamente.

3. *Dijkistra* em problemas *Min-Max*

O algoritmo de Dijkstra modificado para encontrar o peso Min-Max em um grafo direcionado, também chamado de *Minimum Bottleneck Spanning Arborescence (MBSA)*, tem como objetivo determinar o menor valor entre os maiores pesos das arestas que compõem os caminhos possíveis de uma origem a um destino. O problema lida com grafos direcionados conexos com arestas de pesos positivos. O maior peso em qualquer caminho entre dois vértices é conhecido como a *Bottleneck Edge*. O foco do algoritmo é minimizar esse peso, encontrando, assim, o caminho cujo maior peso é o menor possível, comparado a outros caminhos. [Murali]

Este algoritmo pode ser utilizado para minimizar o efeito de gargalos em um caminho entre dois vértices. Por exemplo, considere uma companhia de entregas de cargas frágeis que precisa determinar a melhor rota entre duas cidades (A e B). A prioridade não é apenas encontrar o caminho mais curto, mas também garantir que a maior capacidade de peso ao longo do trajeto seja a menor possível, para minimizar o risco de danos às cargas. Neste contexto, o objetivo seria encontrar o caminho em que o peso máximo de uma aresta (estrada) seja o menor possível ao longo de todo o trajeto.

- `Inicialização`
  - Defina todos os vértices com um valor inicial de infinito para representar que nenhum caminho foi encontrado ainda. O vértice de origem recebe valor 0, pois é onde o cálculo começa.
  - Crie uma fila de prioridade (tipicamente implementada como uma heap mínima) que armazena pares (vértice, valor) para rastrear o caminho de menor Bottleneck Edge a ser explorado.
- `Exploração do Vértice Inicial`
  - A cada iteração, remova o vértice de menor valor da fila de prioridade. Esse vértice representa o próximo a ser explorado.
- `Atualização dos Caminhos`
  - Para o vértice removido da fila, explore seus vizinhos (vértices adjacentes). Para cada aresta que conecta o vértice atual a um vizinho, compare o maior peso entre as arestas do caminho até o vizinho.
  - O peso armazenado no vértice vizinho é atualizado apenas se o maior peso no caminho para o vizinho for menor do que o maior peso atualmente conhecido para esse vértice.
  - Insira ou atualize o vértice vizinho na fila de prioridade com o valor atualizado, garantindo que ele será processado posteriormente.
- `Terminação`
  - O algoritmo continua explorando os vértices até alcançar o vértice de destino ou até que a fila de prioridade esteja vazia.
  - Quando o vértice de destino for removido da fila de prioridade, o valor associado a ele será o menor Bottleneck Edge, ou seja, o menor entre os maiores pesos de todas as arestas ao longo dos possíveis caminhos.

A execução do algoritmo tem uma complexidade de $O((|V| + |E|)log|V|)$, onde $|V|$ é o número de vértices e $|E|$ o número de arestas. A fila de prioridade otimiza a seleção do próximo vértice a ser explorado, garantindo eficiência.

4. Dijkistra em problemas *Max-Min*

4.1. Definição:

Uma abordagem aplicada em um grafo direcionado conexo, em que os pesos, das arestas, são representados por valores positivos e pertencentes ao conjunto de números reais, que visa encontrar o menor peso de cada caminho/subconjunto de vértices e arestas, e dentre as arestas, de menor peso, encontradas selecionar a maior.

4.2. Utilidade:

A variação ”Max-Min” do algoritmo Dijkstra possui inúmeras aplicações, tratando principalmente da ocorrência de gargalos, também conhecido como *maximum capacity path problem*. Por exemplo: em um conjunto de ruas, visa-se diminuir o trânsito/congestionamento de carros, para tanto, o algoritmo ”Max-Min” é utilizado para garantir que o gargalo, ou a diferença entre a quantidade de carros que podem transitar por uma dessas ruas, seja a menor possível, buscando encontrar, dentre as ruas pequenas, ou de menor fluxo, a maior delas, assim sendo, o tempo gasto por cada veículo, para percorrer o trecho congestionado, será o menor possível, uma vez que, tal valor, tempo gasto no congestionamento, é determinado pelo trecho de menor capacidade/largura.

4.3. Procedimento:

- Inicialização
  - Atribuir um valor de peso máximo infinito ao vértice de origem e menos infinito para todos os outros vértices.
  - Usar uma fila de prioridade (*max-heap*, com preferência por pesos maiores) para processar os vértices, priorizando aquele com o maior peso mínimo até o momento, ou seja, o maior dos menores.
- Exploração de Caminhos
  - Extrair o vértice com o maior valor mínimo da fila.
  - Para cada vizinho do vértice extraído, calcular o menor valor entre o valor mínimo atual e o peso das arestas.
  - Atualizar o valor mínimo do vizinho se o novo valor for maior do que o valor já registrado.
- Atualização da Fila
  - Adicione os vizinhos atualizados à fila de prioridade.
- Finalização
  - Repita o processo até que o vértice de destino seja alcançado, ou todos os vértices tenham sido processados.

5. Referências

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT press, 3rd edition.

Murali, T. M. Applications of minimum spanning trees. https://courses.cs.vt.edu/~cs5114/spring2009/lectures/lecture08-mst-applications.pdf.

Weiss, M. A. (2013). *Data Structures and Algorithm Analysis in C++*. Benjamin/Cummings Publishing Company, Inc., 4nd edition.

[Cormen et al. 2009] [Weiss 2013]