Algoritmo de Dijkstra: Implementações para Abordagens MinMax e MaxMin

Gabriel de Cortez Mourão[1], **Luís Augusto Lima de Oliveira**[1], **Mateus Fernandes Barbosa**[1], **Victor Ferraz de Moraes**[1]

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

[2]Instituto de Ciências Exatas e Informática
ICEI

Abstract. *In this article, a detailed exploration of Dijkstra's algorithm, developed by Edsger W. Dijkstra in 1959, is presented. This algorithm calculates the shortest path between vertices in a graph. The implementation, complexity, and limitations of the algorithm are discussed. In addition, two variants of Dijkstra's algorithm are addressed: Min-Max and Max-Min. The Min-Max version aims to minimize the maximum weight along a path between two vertices, used in situations where the goal is to reduce bottlenecks in routes. The Max-Min version optimizes the maximum of the smallest edges, applied in situations such as traffic route optimization to reduce congestion.*

Resumo. *Neste artigo, é realizada uma exploração detalhada do Dijkstra, um algoritmo desenvolvido por Edsger W. Dijkstra em 1959, que calcula o caminho mais curto entre vértices em um grafos. É discorrido sobre a implementação, complexidade e limitações do algoritmo. Além disso, duas variantes do algoritmo de Dijkstra são abordadas: Minmax e Maxmin. A versão Minmax 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 Maxmin otimiza o máximo das menores arestas, aplicada a situações como a otimização de rota de tráfego para reduzir o congestionamento.*

1. Introdução

O problema de encontrar o menor caminho entre dois pontos em um grafo é fundamental em diversas áreas, como ciência da computação, engenharia de redes, transporte e otimização logística. Este artigo explora o algoritmo *Dijkstra*, um dos algoritmos que resolve esse problema, analisando suas características, complexidade e implementações. Também serão abordadas modificações do algoritmo para resolver problemas de *Maxmin* e *Minmax*, que mantêm a mesma performance, mas tratam de questões específicas na busca em grafos.

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.

O algoritmo pode ser usado 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 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 a partir da origem, é preciso fazer busca com todos os vértices.
  - 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 usa como representação do grafo usa uma Matriz de Adjacência, e como estrutura auxiliar para a pesquisa mais simples é uma 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 é de $O(|V|)$ em cada iteração, tanto para encontrar a menor distância quanto para acessar os vizinhos para um máximo de $|V|$ iterações que representa um custo de complexidade $O(|V^2|)$.

Outra implementação usa uma estrutura para a representação do grafo como uma lista de $|V|$ elementos, em que cada índice representa um vértice. Cada vértice contém uma lista interna que representa todas as arestas vizinhas a ele. A simples mudança para esta estrutura permite com que a busca por vértices vizinhos seja certeira. Logo, a busca de vértices vizinhos é reduzida de um custo total $|V^2|$ para custo total de $|E|$, sendo *E* a quantidade de arestas. Esta estrutura é chamada de Lista de Adjacência.

Para reduzir o custo de encontrar o vértice de menor distância não percorrido, a estrutura auxiliar de lista de vértices visitados 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 na com a nova distância, para ser posteriormente visitado.

Isto permite que toda busca pelo vértice seja de custo constante $O(1)$. Por consequência da organização da fila de prioridades, toda visita para um vértice o remove da fila, com um custo de $O(log|V|)$, feita no máximo $|V|$ vezes, e toda vez que uma melhor distância é encontrada, também adicionamos valores na fila com complexidade de $O(log|V|)$, o que seria feito no máximo $|E|$ vezes, ou seja, para todos as arestas.

Logo, a complexidade resultante do uso de uma estrutura de lista de adjacência e de uma estrutura auxiliar de fila de prioridades será de $O((|V| + |E|)log|V|)$ [Wikipedia 2024]. Esta implementação é simples e eficiente, e por isto será usada neste artigo para a criação do algoritmo de *Dijkstra* e de suas modificações.

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 é a necessidade de todos os pesos das arestas serem positivos, problemática 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.

2.3. Implementação do *Dijkstra*

Os algoritmos foram implementados na linguagem c++, com o auxílio de estruturas de dados e outras funções da biblioteca padrão *std*. A estrutura implementada para os grafos segue uma fila de adjacência, como explicado na seção de estrutura de dados.

- `Inicialização`
  - Uma lista de distâncias, chamada de *dist* é criada com o tamanho equivalente à quantidade de vértices, de forma que cada posição representa um índice de um vértice. Todos os valores se iniciam como o maior número positivo representável do tipo usado para distância, exceto o vértice de origem que possui distância 0.
  - Uma lista de parentes, chamada de *parent* também é instanciada com o tamanho equivalente à quantidade de vértices. Todos são inicializados com -1, indicando que nenhum vértice é o seu pai atualmente.
  - Uma fila de prioridade *min-heap* de pares (vértice, distância) é criada e ordenada pela distância. O vértice de origem é adicionado na fila com distância 0.
- `Exploração de Caminhos`
  - Para cada iteração, o primeiro valor da fila é armazenado em variáveis auxiliares e logo após removido da fila. Este será o vértice atualmente visitado pelo *Dijkistra*, e possuirá a melhor distância para o vértice. É possível que mais de um par de valores seja adicionado na fila para o mesmo vértice. Neste caso, o primeiro sempre possuirá o melhor valor, e os demais poderão ser ignorados. Isto condiz com a ideia do algoritmo em que o vértice só precisa ser visitado uma vez.
  - Para cada vértice v em visita, todos seus vizinhos N são explorados. Para cada $n \in \mathbb{N}$, se dist$[n] <$ dist$[v] +$ w(v, n), onde w representa o peso da aresta, temos que dist$[n] =$ dist$[v] +$ w(v, n), parent$[n] = v$ e n é adicionado na fila com o sua nova distância mínima.
- `Término do algoritmo`
  - Existem duas diferentes condições de término, dependente do resultado esperado:
    - Para encontrar a menor distância entre dois vértices, as iterações são executadas até o vértice de destino ser visitado
    - Para encontrar a menor distância entre todos os vértices para a origem, as iterações serão executadas até todos os vértices forem visitados, ou seja, até a fila de prioridade estiver vazia
  - O melhor caminho pode ser identificado através da análise dos vértices antecessores dados pela lista $parent$, partindo através de um vértice destino qualquer.

3. *Dijkistra* em problemas *Minmax*

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.

3.1. Implementação do *Minmax*

- `Inicialização`
  - Uma lista de pesos resultante é criada para representar o mínimo do máximo do peso de cada vértice. Cada valor é inicializado como o maior valor numérico atribuível, de tal forma que qualquer peso máximo tenha um valor menor. O vértice de origem recebe valor 0, pois é onde o cálculo começa.
  - É criada uma lista de parentes para cada vértice, com o intuito de obter o melhor caminho obtido para um vértice qualquer
  - É criada uma fila de prioridade min-heap que armazena pares (vértice, peso) para rastrear o caminho de menor *Bottleneck Edge* a ser explorado. O vértice de origem é adicionado na fila antes do loop, com peso 0.
- `Exploração de Caminhos`
  - A cada iteração, é removido um vértice de menor valor da fila de prioridade. Esse vértice representa o próximo a ser explorado.
  - Para o vértice removido da fila, seus vizinhos serão explorados. Para cada vizinho, se o peso máximo entre o peso da aresta de conexão com o vértice a ser explorado e o peso do vértice removido for menor que o peso armazenado no vértice adjacente, o valor do vértice adjacente é atualizado, e ele será adicionado na fila de prioridade com seu novo peso.
- `Término do algoritmo`
  - É terminado o algoritmo quando for explorado o vértice de destino, ou quando a fila de prioridade ficar vazia, assim como na implementação do Dijkstra
  - O melhor caminho para um vértice da origem pode ser identificada através da análise dos vértices antecessores (parentes).

4. Dijkstra em problemas *Maxmin*

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 *Maxmin* 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. Implementação do *Maxmin*

- `Inicialização`
  - Uma lista de pesos resultados é criada para representar o máximo do mínimo de cada vértice. O valor inicial de todos os pesos exceto para o vértice de origem será o menor valor atribuível para a variável numérica, de forma que qualquer outro peso tenha um valor maior do que este valor inicial, já que o algoritmo não aceita números negativos. O vértice de origem recebe o maior valor representável, uma vez que o mínimo dele com qualquer peso de qualquer aresta sempre resultará no valor da aresta.
  - É criada uma lista de parentes para cada vértice, com o intuito de obter a o melhor caminho obtido para um vértice qualquer.
  - 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`
  - É extraído o vértice com o maior valor mínimo da fila.
  - Para cada vizinho do vértice extraído, se o peso mínimo entre o peso da aresta de conexão com o vértice a ser explorado e o peso do vértice removido for maior que o peso armazenado no vértice adjacente, o valor do vértice adjacente é atualizado, e ele será adicionado na fila de prioridade com seu novo peso.
- `Término do algoritmo`
  - Repita o processo até que o vértice de destino seja alcançado, ou todos os vértices tenham sido processados.
  - O menor caminho pode ser identificado através da análise dos vértices antecessores (parentes).

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.

Wikipedia (2024). Dijkstra's algorithm — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Dijkstra's\%20algorithm&oldid=1242641432. [Online; accessed 10-October-2024].

[Cormen et al. 2009] [Weiss 2013]